

Cloud Gaming and Simulation in Distributed Systems

Khen Cruzat

**Submitted in accordance with the requirements for the degree of
Computer Science**

2015/16

The candidate confirms that the following have been submitted.

<As an example>

Items	Format	Recipient(s) and Date
Final Report (2 copies)	Report	SSO (10/05/16)
Final Report (digital)	Report	VLE (10/05/16)
Project Code	GitHub Repository	Supervisors, Assessor (10/05/16)
User Manual	Report Appendix	SSO (10/05/16)

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

Summary

Cloud computing attempts to enable access to high-end graphics intensive games to a wider audience by using powerful data centers. The process of delivering cloud gaming includes processing the user input for the game engine then encoding this as video and streamed to the end user. Due to another added layer of complexity, other problems arise and the aim of this project is to tackle networking issues in the cloud to reduce the latency experienced by the user.

The Software-Defined Networking paradigm allows the management of network technology without the need of touching individual switches. This project will discuss the effects of using SDN for cloud gaming in a distributed system.

Acknowledgements

I would like to thank my supervisors Jie Xu and Peter Garraghan for all the guidance and advice they have given to me throughout the project. Weekly meetings were very helpful to keep my motivation going as well as a time to reflect on current progress on the project and to discuss challenges faced.

Another person I would like to thank is my assessor David Duke. The feedback given to me for the Planning and Scoping document and Progress Meeting was very helpful so I could narrow down my problem.

I would also like to thank the university IT support team especially John Hodrien for their help regarding the use of the university School of Computing cloud testbed. My personal tutor Karim Djemame was supportive as a meeting was held with the other members in the tutor group to contemplate on the successes and challenges of each others projects which was helpful to keep me motivated.

Contents

1	Introduction	1
1.1	Context	1
1.2	Project Aim	1
1.3	Project Objectives	1
1.4	Deliverables	1
1.5	Methodology	2
1.5.1	Project Schedule	2
1.5.2	Methodology	3
1.5.3	Version Control	3
2	Background Research	5
2.1	Problem Overview	5
2.2	Cloud Computing	5
2.3	Cloud Gaming	6
2.4	Latency Mitigation	8
2.4.1	Speculative Execution: Outatime	9
2.4.2	Software-Defined Networking	10
2.5	Related Works	11
3	Design	13
3.1	Cloud Gaming System	13
3.2	Game Design	16
3.3	Virtual Network	19
3.4	SDN Application Design	20
4	Implementation	23
4.1	Game Implementation	23
4.2	Cloud Gaming	24
4.3	Networking Implementation	26
4.3.1	Mininet Virtual Network	26
4.3.2	OpenDaylight SDN Controller	28
4.3.3	Load Balancer	28
5	Testing and Evaluation	39
5.1	Test Cases	39
5.1.1	Case 1	39

5.1.2	Case 2	40
5.1.3	Case 3	42
5.2	Results	44
5.3	Evaluation	49
6	Conclusion	51
6.1	Conclusion	51
6.2	Future Work	52
6.3	Personal Reflection	53
	References	55
	Appendices	59
A	External Material	61
B	Ethical Issues Addressed	63
C	User Manual	65
C.1	Cloud Gaming System	65
C.1.1	Game Server	65
C.1.2	Client Program	65
C.2	Networking	66
C.2.1	Mininet Virtual Network	66
C.2.2	Open vSwitch	66
C.2.3	OpenDaylight SDN Controller	67
C.2.4	Load Balancer	67

Chapter 1

Introduction

1.1 Context

Cloud computing has always been seen as a way of improving compute performance by the use of multiple computers connected to each other. The games industry has rapidly evolved along the years and a great deal of demand is apparent. Developers of games have pushed computer hardware to meet the needs of consumers for more complicated games and realism. Even with computer hardware becoming cheaper and more of a commodity, the costs of driving graphics rendering for high-end games to run at the optimal settings of 1080p at 60fps are still relatively high.

1.2 Project Aim

The aim of the project is to produce a solution which uses software-defined networking to reduce the network latency in a network in terms of a cloud gaming system.

1.3 Project Objectives

- A simple game program, that is computationally expensive enough to not perform optimally on a single machine (simple flight simulator with real time procedurally generated trees).
- A simplified cloud gaming system where the game created is launched on the cloud and input on the client side in the form of button presses on the keyboard is sent to the game on the server. The game frames produced are then sent to the client's screen.
- Produce a virtual network with simulated cloud game traffic and delay. With the use of SDN, reduce latency in the network.

1.4 Deliverables

The deliverables of the project include:

- Code that demonstrates a simple game/simulation rendering graphics on a server and controlled by a client remotely and a manual on how to setup the client and server for the cloud gaming system.

- Code that creates a virtual network and software-defined networking load balancer as well as a manual on how to set it up and run the code and software.
- Project report that explains the problem the project is trying to solve and the schematics of the solution produced as well as an evaluation of the solution.

1.5 Methodology

1.5.1 Project Schedule

	February					March				April				May	
Task	01/02	08/02	15/02	22/02	29/02	07/03	14/03	21/03	28/03	04/04	11/04	18/04	25/04	02/05	09/05
Scoping and Planning															
Background Research															
Literature Review															
Architecture Design															
Prototyping Game															
Prototyping Cloud															
Prototype SDN															
Prototyping Testing															
Prototype Evaluation															
Implementation															
Testing & Evaluation															
Report Write Up															
Deadline Submission															

Figure 1.1: Gantt chart of project schedule

1.5.2 Methodology

The approach that was used for this project was iterative. Iterative process means doing an initial plan followed by an iteration of planning, design, implementation, testing then evaluation. This iteration is repeated again to improve the prototype. An iteration of the project development is outlined below.

The problem and scope of the project was first defined and this includes background research After doing background research, the next process was literature review of the information I have found. Comparisons of different techniques that reduce or mask latency and increase the overall performance of the system was made. A single technique was then chosen to focus on and this was software-defined networking. The next step was to design the architecture for the cloud gaming system along with how it will be implemented:

A prototype of the game was produced and was made sure it was running properly on a single system in the DEC-10 Lab. The game rendered a procedurally generated tree

using Lindenmayer system which is computationally expensive to produce in real time. The game also uses lighting and shadows which adds more to the compute power needed. The game area can be navigated around by the player using simple flight simulator controls.

When this was completed, a server-client system where the server can receive user input from a client and relay the commands to the OpenGL program was prototyped. The intention of this was to stream the rendered frames in video format to the client's window and the bandwidth of the video traffic was to be recorded. Simulated traffic was then used on a virtual network for the SDN solution which was tested against different test scenarios and evaluated.

1.5.3 Version Control

The project uses version control for both the code development and report write up. A public GitHub repository was created and development changes were saved as the project progressed. This is so backups are created regularly just in case some new code broke the implementation, an older version can be easily loaded back up. The code repository is also hosted remotely so it can be accessed using multiple machines and not in one machine. This protect against losing files accidentally if it was stored locally. The GitHub public repository is hosted using the following link:

<http://github.com/kvcruzat/cloudgaming/>.

Chapter 2

Background Research

2.1 Problem Overview

Compute power of games can be offloaded to a server of much powerful computers and then streamed to a client with lower specification hardware such as a laptop or a mobile device. With this comes risks and shortcomings that need to be factored. One of these problems are the latency of the game. Latency can be huge factor in the gameplay such as high-paced games like first-person shooters or fighting games. The delay in pressing a button on the gamepad to seeing the action performed on the screen needs to be kept to a minimum. This idea of interaction delay tolerance being different from genre to genre of games is discussed by Shea et al. [38]. As stated above, a player of FPS games can only tolerate the least which is around 100ms whereas Role playing game (RPG) gamers can tolerate around 500 ms.

Another problem that is directly linked to delay in the system is the effect of packet loss. As stated in the *Eight Fallacies of Distributed Computing* [7], it should be assumed that latency is never zero as mentioned above as well as network is not always reliable. This means that packet loss can occur which in terms of cloud computing can mean the degradation of image quality. In the investigation conducted by Jarschel et al. [14] in which they surveyed average consumers about the importance of packet loss and delay. Generally the quality of the video streamed to the clients plays an important role as the participants were open to using such as a service if provided in good quality.

2.2 Cloud Computing

According to the National Institute of Standards and Technology [24], cloud computing is a means of providing on demand access to computing resources over the network. This should be executed with minimal management effort or service provider interaction. With the shared computing resources, cloud computing aims to process larger data and solve large scale computation.

This cloud computing model can be separated in to three different service models: Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) [13]. 'SaaS' model attempts to eliminate the need to install and run the application on the user's system. An example of this is the Microsoft Office 365 package

which provides productivity software through the web browser. 'PaaS' model provides the consumer a computing platform using the cloud infrastructure to allow running and building their own applications. The consumer does not need to manage the "underlying cloud infrastructure including networks, servers, operating systems or storage [24]".

'IaaS' model provides the capability to control the processing, storage and networks to an extent. The consumers will have to install the OS images and related application software on the cloud infrastructure [18].

A disadvantage stated by Grossman [10] is that since cloud services are often remote, it can suffer from latency and bandwidth related issues. Data centres can be physically located anywhere in the world, so the number of router hops from the client to the server may attribute to the latency. As well as distance, delays are also introduced by network hardware and error correction on data packets.

2.3 Cloud Gaming

Cloud gaming is new technology that can be seen as an alternative by having the games run remotely on a server and then streamed to the user. Performing computations remotely as with streaming games remotely is believed to gain traction in the future in the same way how streaming videos and audio have become ubiquitous through services such as Netflix and Spotify. NVIDIA's GRID Cloud Gaming advancements have shown that this is becoming the case. As stated by Mariano in *Is cloud gaming the future of the gaming industry* [23], cloud gaming is increasingly becoming an attractive option for consumers as higher end games can then work on simpler, cheaper clients as well as with devices that they may already own also known as thin clients through the use of powerful server GPUs.

These thin clients are responsible for displaying the game frames rendered on the cloud server side in the form of video frames. Also, it has to collect and process the game control inputs from the user and send these to cloud to be registered as inputs on the game engine. According to Shea et al [38], cloud gaming would be of great benefit to the game industry as it would open the user base to the thin clients. For example the recommended specifications to run the 2015 Game of the Year title *Witcher 3* [8] would require a system that has [28]:

- CPU: Intel Core i7 3770 3.4 GHz / AMD FX-8350 4 GHz
- GPU: GeForce GTX 770 / AMD Radeon R9 290
- RAM: 8GB

A system that has these components would cost around £400 and this does not include peripherals such as keyboard, mouse and monitor. The latest tablets and laptops can barely if at all meet the recommended specifications to run the game natively. Furthermore, games will have to deal with running on different hardware architectures and operating systems. Cloud gaming will make it easier for developer as they will not have to deal with platform compatibility and per platform tuning.

In the paper *Cloud Gaming: A Green Solution to Massive Multiplayer Online Games* [6] it mentions that NVIDIA has introduced SHIELD which is a mobile gaming device that can be connected to a desktop PC with a compatible NVIDIA GPU and stream gameplay to the device via 802.11n WiFi. Another feature is the ability to connect to one of NVIDIA's data centres to play games from their vast library of stream-ready games. One of the benefits of this service is the convenience of not having to wait for the download and installation of the game as you simply pick a game and instantly start playing. The games will not be stored on the client's device so storage limitations will not be a problem. The service also boasts gameplay performance of up to 1080p at 60fps which are deemed by gamers to be the target performance.

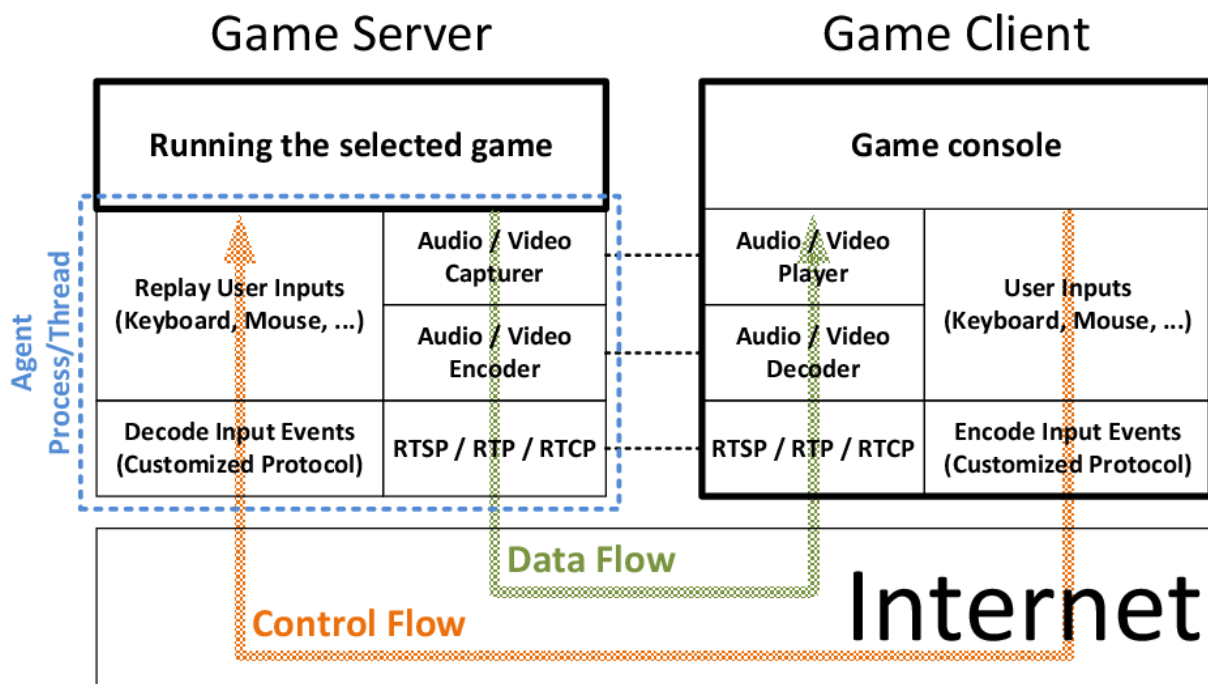


Figure 2.1: Cloud gaming system architecture [9]

A simple cloud gaming architecture consists of several procedures which adds on top of the game engine process as shown in Figure 2.1. The client would send user inputs through some form of controller like a keyboard, mouse, or gamepad to the thin client. The game client then encodes these commands so it can easily be sent to the game server via network. When the server receives the user inputs, it simulates them so the game that is running recognises them. The game renders the frames produced as a

result of the inputs and are captured and encoded to video frames. This is done so it can be easily streamed to the client through the use of Real Time Streaming Protocol (RTSP) which is a network control protocol that manages delivery of data with real time properties [33]. The thin client uses RTSP to receive the video frames and decodes them to be displayed on a video player. The user then sees the results of the button presses sent from the game running on the server side.

Due to the many different processes involved in the architecture of a cloud gaming system, managing latency has become a problem. A traditional gaming system already experiences latency and as shown in Figure 2.2, this arises from the game pipeline and display lag. The game pipeline latency is the amount of time it takes for the game to compute and render a frame and the display lag is the time it takes to display the frames on the screen. Display lag can be caused by the display’s scaler since current displays have a fixed resolution and expensive image processing such as dynamic contrast and motion interpolation [39]. Cloud gaming introduces latency from capturing/encoding game frame to video frame on the server side, network latency with the transmission of data to both sides and decoding on the client side.

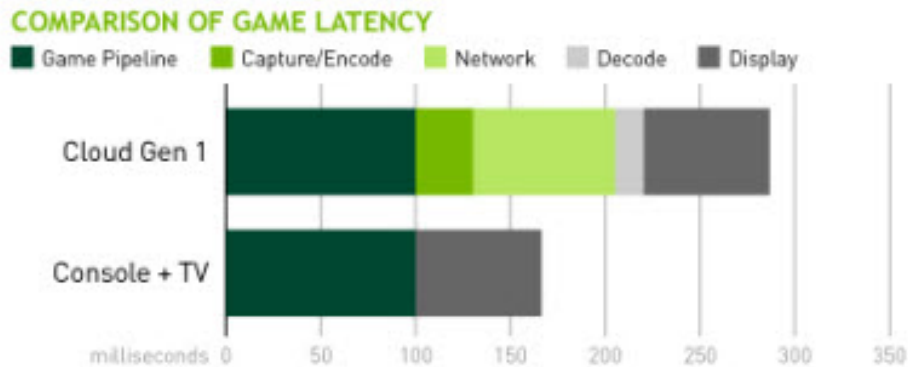


Figure 2.2: Latency in cloud gaming [15]

The aim of this project is to reduce the network latency which is the element that introduces the most latency in cloud gaming.

2.4 Latency Mitigation

One method of latency mitigation in the network is to simply move the server closer to the clients. This means that traffic will have to travel less distance therefore latency will decrease. Unfortunately, this solution is not feasible since building and maintaining data centres are expensive. Video streaming services such as Netflix and YouTube use buffering which loads video data before playing the video in order for continuous

playback. This cannot be done for live cloud gaming so other solutions to improve experiences of latency sensitive games must be explored.

2.4.1 Speculative Execution: Outatime

Microsoft's Outatime which uses *speculation to enable low-latency continuous interaction for mobile cloud gaming* [19] is a form of latency mitigation. This method acknowledges that there will be latency and attempts to reduce its effects since studies have shown that players are sensitive to latency from 75-100ms, decreasing accuracy in shooting games and decreasing their scores by almost up to 50% [4].

Outatime basically predicts multiple possible frame outputs that may appear in the future of the game's render scene on the client side. It has to predict what frames may be needed at least a full end host round trip time (RTT) ahead of time the client actually produces game input controls. So unlike standard cloud gaming where clients receive a response after more than one RTT, Outatime delivers a response immediately since the possible frames are all ready to be displayed.

Outatime was extensively tested by having people play a shooting game without knowing if they were using a traditional cloud gaming system or Outatime. The tests were conducted multiple times at various network delays and players were asked qualitatively their opinion on the playability of the game. The results show that Outatime was favoured by the players in all cases even with experienced players.

2.4.2 Software-Defined Networking

Another form of latency mitigation that is being explored is software-defined networking. As stated by Kirkpatrick [17], software-defined networking (SDN) is a new networking architecture that allows programmers to quickly reconfigure and define network usage. Whilst significant advances have been in other areas of technology, networking has not been able to evolve in the same pace.

Similar to mobile phones shifting to the world of smartphones with the help of APIs (application program interface), in an SDN environment, applications can communicate with network switches through an API. The API can be used to quickly reconfigure the resources of the network to accommodate the needs of the applications being executed. This main benefit of using SDN is also discussed in *Improving network management with software defined networking* [16]. Kim et al mentions that network operators will not need to configure all the network devices individually to make network behaviour changes, but instead make network-wide traffic forwarding decisions. The SDN

controller is used for this and would have global knowledge of the state of the network.

SDN consists of two planes, the data and control plane. The data plane also known as the forwarding plane is the part of the network that carries user traffic by forwarding them to the next hop along the path to its destination in accordance to the logic in the control plane [32]. The control plane has command where the traffic is sent by creating the routing tables and is also responsible for managing connections between switches, handling errors and exceptions [31]. This separation allows the control plane to be directly programmable and the network elements in the data plane to be abstracted for networking services.

The controller is the core component for the software-defined networking. It is an application that manages the flow control of data between nodes. In order for SDN applications to communicate with the network switches it has to go through the controller which can be accessed by applications with REST API (a way to communicate to between machines through HTTP requests). This allows network administrators to manage how the data plane handles network traffic and its routing.

2.5 Related Works

There are a few researches that has been conducted that is related to latency mitigation in cloud gaming systems using software-defined networking. Amiri et al. [3] explored using SDN to reduce latency by using SDN to share the load in a network. The shortest path is found in the network and the path cost is computed by pinging the machines at each node to discover the end-to-end delay. Incoming packets are then split among the paths depending on the weights so the load is evenly shared in the network.

Another research that is related is also by Amiri et al [2] and aims to reduce network latency by considering the type of game being requested and using this to optimally assign a game server using this property. Different types of games have different network delay sensitivities such as a shooting game would have a higher sensitivity than a turn-based game. With this taken in to account it uses a weighted priority system when using SDN to find game servers that have paths that allow for this network delay.

This project used this idea of load sharing, but measures the current load in each path and forwards traffic to the path with the lowest load dynamically. Also, multiple test cases that reflect real world scenarios of cloud gaming system usage are used to better understand the effectiveness of software-defined networking.

Chapter 3

Design

This chapter outlines the proposed design of the cloud gaming system as well as the game that will be used to run on the system. Also at a high level, this chapter discusses the virtual network design that will be used to simulate a cloud data centre as well as the proposed solution to use software-defined networking to reduce latency in the network.

3.1 Cloud Gaming System

As a result of the background research conducted, the cloud gaming system will aim to offload the game engine to the cloud data centre. This includes the game logic, physics and graphics rendering. Due to this, it will leave the client's application to just take control of receiving physical button input and send this to the game server as well as receiving and displaying the game video frames.

It can be seen at Figure 3.1 that multiple players should be able to connect to the cloud server and a new virtual machine should be generated for each player. For each virtual machine, an instance of the game will be executed with the required resources such as RAM and processing power provided by the cloud's resource manager. The resources required can be specified as a template with the parameters already set. This makes sure that each game instance has sufficient resources to run at smoothly.

A design improvement on this is to allow the client to specify the video settings to use such as 1080p resolution at 60 frames-per-second and 720p resolution at 30 frames-per-second. Enabling this option will give the client an option to manually improve their gameplay experience since sending higher resolution frames as well as more of them per second will require higher bandwidth. This also helps the cloud data centre to use different templates for virtual machine resources so they will not be wasted by providing too much and therefore can be allocated to other virtual machines.

The software-defined networking controller will manage the networking inside the data centre. It will have global knowledge of the all switches and the links between them. A load balancing application will be used to make sure that traffic generated with the video streaming is routed appropriately to help keep latency in the network to a minimum. This will be discussed later on section 3.4.

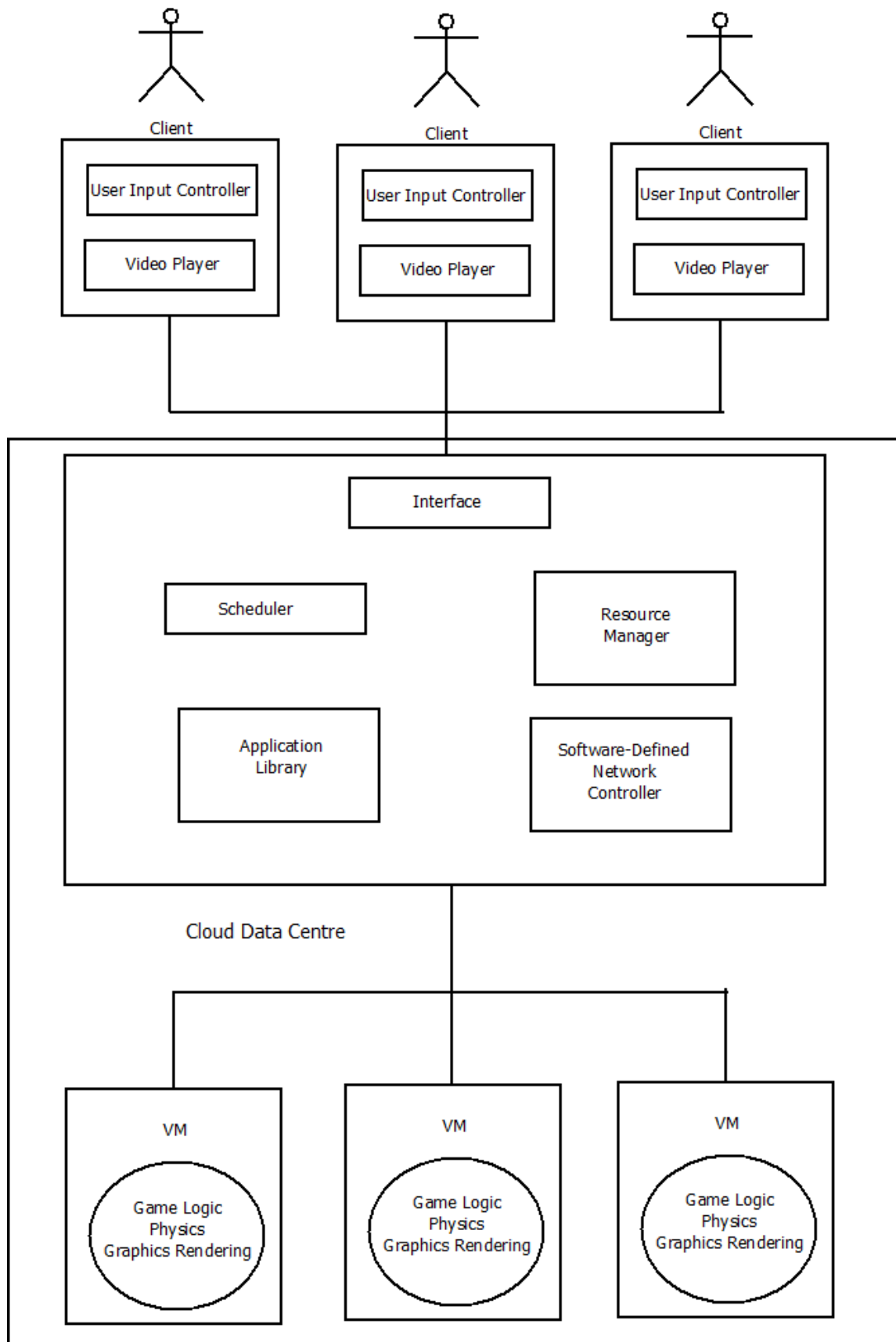


Figure 3.1: Cloud Gaming System Model

As for the framework of the cloud gaming system, it will start with a local program on the client's machine. This program will connect to a server on another machine using its IP address and the port number it listening at. Once a connection is made the server will launch the game and start capturing the rendered frames and encoded them to video to be streamed to the client. The client program will receive the video frames and display them using a video player embedded on the program. It will also be actively waiting for player interaction through the keyboard. The only commands that will register with the client program will be the ones binded for the flight simulator controls. Once the player has pressed one of these, it will send the input to the server where it is processed and simulated as an in game command. Then similarly to traditional games, the game pipeline uses the game command to process the game logic with this new information for rendering updated video frames. This process is displayed in Figure 3.2 as a cycle where the player receives video feedback of the game and responds with keyboard game commands.

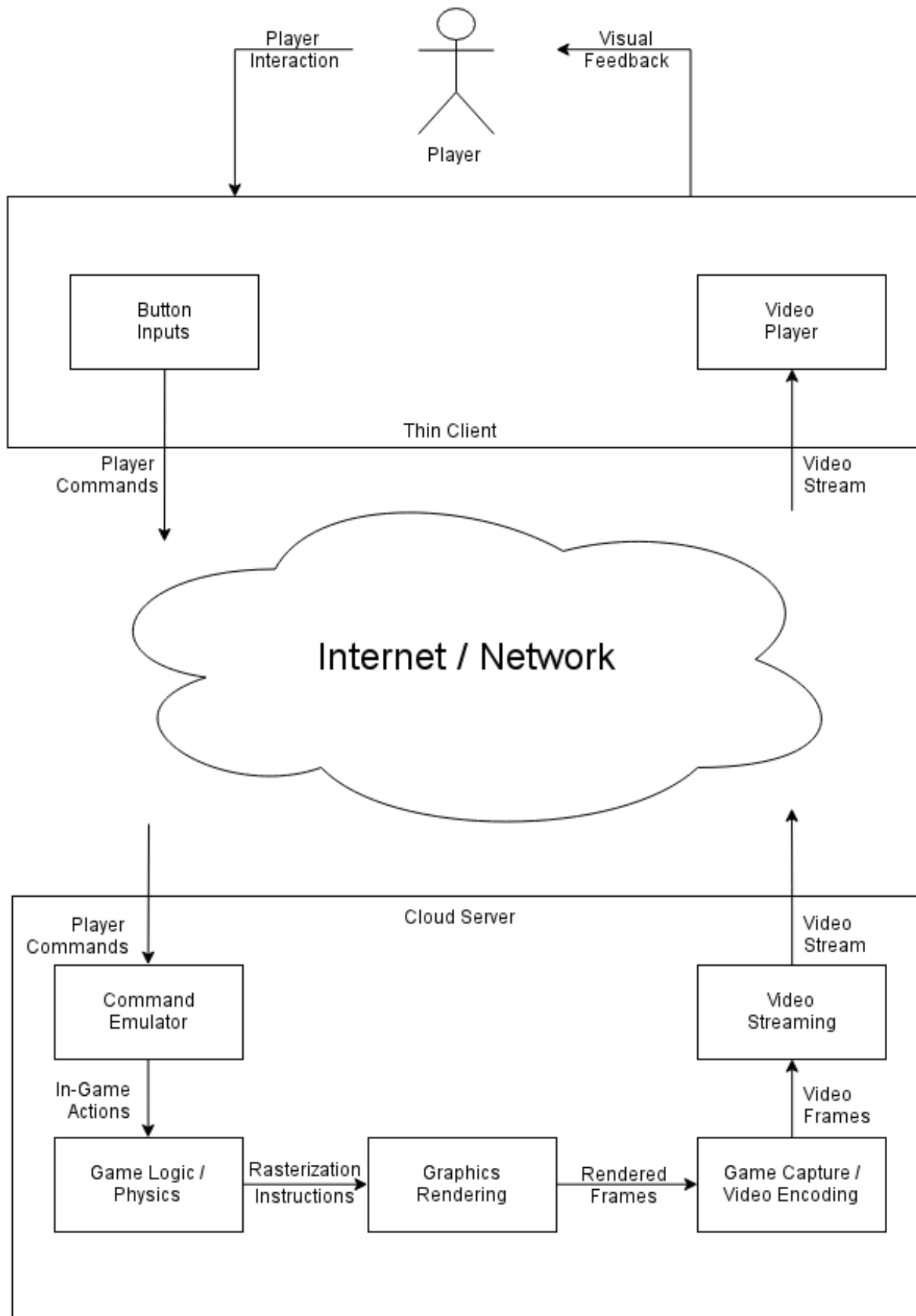


Figure 3.2: Cloud Gaming Framework [6]

3.2 Game Design

This section will indicate the design of the game at a high level that will developed to be used as the program to benchmark on the cloud gaming system. The aim of this game is to be somewhat computationally expensive to warrant as a program that will improve by being on a system that is more powerful than a mainstream consumer system. The game should be configurable so it can test out the limits of the system by increasing the amount of computer power it needs. It should also implement a method to measure latency that will arise not just numerically but through user input and immediate visual feedback. The client will experience this lag when a button is pressed, but the expected visual feedback is not being displayed as soon as expected.

To make the game computationally challenging, trees will be modelled and generated in real time. Plants and trees are part of the natural environment and are often used in games to create a realistic scenery. They are geometrically complex and difficult to generate realistic models in real time. Such trees are usually pre-generated and saved locally so it can be loaded easily when needed, but this means the game is limited to using those models and tree models will have to be repeated. This degrades the player's experience as they see duplicate trees in the scenery reminding them that they are in a game which breaks the immersion.

Lindenmayer systems or 'L-systems' is a part of formal language theory to write parallel grammars describing growth similar to the way DNA is a programming language of the human body [29]. Plants tend to have patterns in their growth, but fundamentally they grow forward, rotate then branch out in a hierarchy starting from the root. L-systems are stated as production rules and correspond to each stage of growth for each part of the plant according to a fixed pattern. The notation in the pattern can be simplified to:

- F : move forward and draw
- +, - θ : rotate θ around x-axis
- &, ^ θ : rotate θ around y-axis
- /, \ θ : rotate θ around z-axis
- [,] : push / pop

Using the symbols above, grammars for tree generation can be produced that will specify the pattern to be followed. The symbols '[' and ']' which represent push and pop respectively refer to a Last In First Out stack. When the symbol '[' is reached, the current position and angle is saved and are restored when the symbol ']' is encountered. Variations can be implemented through randomized parameters such as random angles

for rotation and length of branches. For more complex games, environmental factors can be used to determine the parameters such as competition for light, food, diseases and animals [5]. For the game produced from this project, random rotation angles will be used to generate variation. The production rules below are recursive and will produce two branches for each branch on each recursion.

$$Trunk \rightarrow F[+\theta/\theta Branch][- \theta \setminus \theta Branch]$$

$$Branch \rightarrow F[+\theta/\theta Branch][- \theta \setminus \theta Branch]$$

To better illustrate how a tree is produced at each recursion, Figure 3.3 shows a two dimensional version. The production rules are based on the Pythagoras tree which is a tree constructed from squares and is named due to each triple of squares enclose a right angle triangle. The tree starts with one square and with each recursion two squares are used for each square from the previous recursion. As shown in Figure 3.3, the amount of branches grows in size and complexity pretty quickly, in fact it grows exponentially where the total branches equal to $2^n - 1$ with n being the number of recursions.

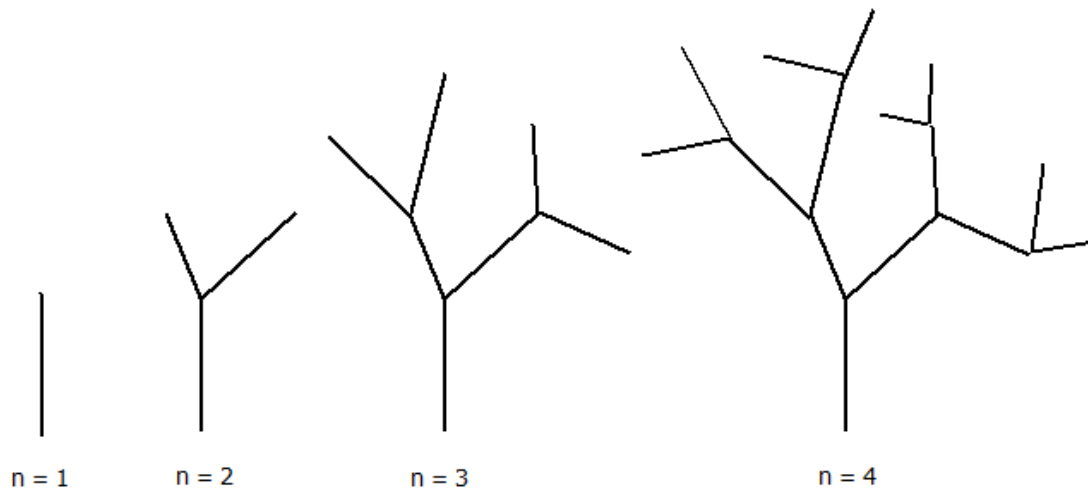


Figure 3.3: 2D Pythagorean L-system tree

Computing the vertices of these many branches in real time will be expensive and adding lighting and shadows will add even more to the resource requirement. Projective shadows is a shadow technique that casts shadows by point light sources onto planes. It takes advantage of a shadow matrix that uses perspective transformation to render an object on to the plane depending on the position of the light source [11]. The problem with this is that it doubles the computation since the object is being rendered twice; the actual object and its shadow. To measure the performance, a frames-per-second counter should be used. The higher the number the FPS counter shows, it means the higher performance the system is capable of since it can push more frames at a given time which means the player will have a smoother gameplay experience. An important factor that should also be considered is the variance in the framerate. A steady framerate

should be the aim as frame drops would cause stuttering.

In order for latency to be evident when playing the game, a form of interaction needs to be implemented. A simple flight simulator was chosen so the user can traverse the game and be a way to view the tree model from different angles. The flight simulator will support basic controls such as acceleration and deceleration and roll, yaw and pitch movement. Latency will be easily detected when the user inputs a command then the game's camera doesn't move immediately.

3.3 Virtual Network

Due to limitations with the School of Computing cloud testbed, the proposed solution of software-defined networking to mitigate latency will not be able to be conducted. An alternative solution is to simulate the video traffic that would have been using the data centre's network. This can be done through virtual networks where it can run real kernel, switch and application code on a single machine. Simulating a network also means that switches and links can be easily configurable for software-defined networking controller to work and to simulate real world factors such as latency and bandwidth limitations.

Figure 3.4 shows a basic topology of a cloud network that is currently in use by three different clients. Each client is connected to a single switch which is the core switch acts as an entry point to the data centre. The core switch is linked to other switches called aggregation switches and is eventually linked to a Top of Rack (ToR) switch which is connected to the cloud server host machine running multiple virtual machines. This network design is called fat tree topology. A tree topology is used since it is commonly used in data centres for its good scalability, accessibility, cost effectiveness and low latency [41]. The advantages of using fat tree instead of a binary tree is that multiple paths will be available between nodes which improves fault tolerance and increases inter-rack bandwidth [1]. The main disadvantage that has been considered with using this topology is its single point of failure at the root switch. If this core switch has a fault the entire tree network collapses. Solving this problem is beyond the scope of this project and this topology is used mainly for the fact that multiple paths will be available between two nodes.

Simulated video traffic will be generated at a cloud game server source and sent to the client machines. Large amounts of video data will have to be transmitted which makes the network links prone to congestion. Latency will be simply measured by pinging between host machines while network is under load.

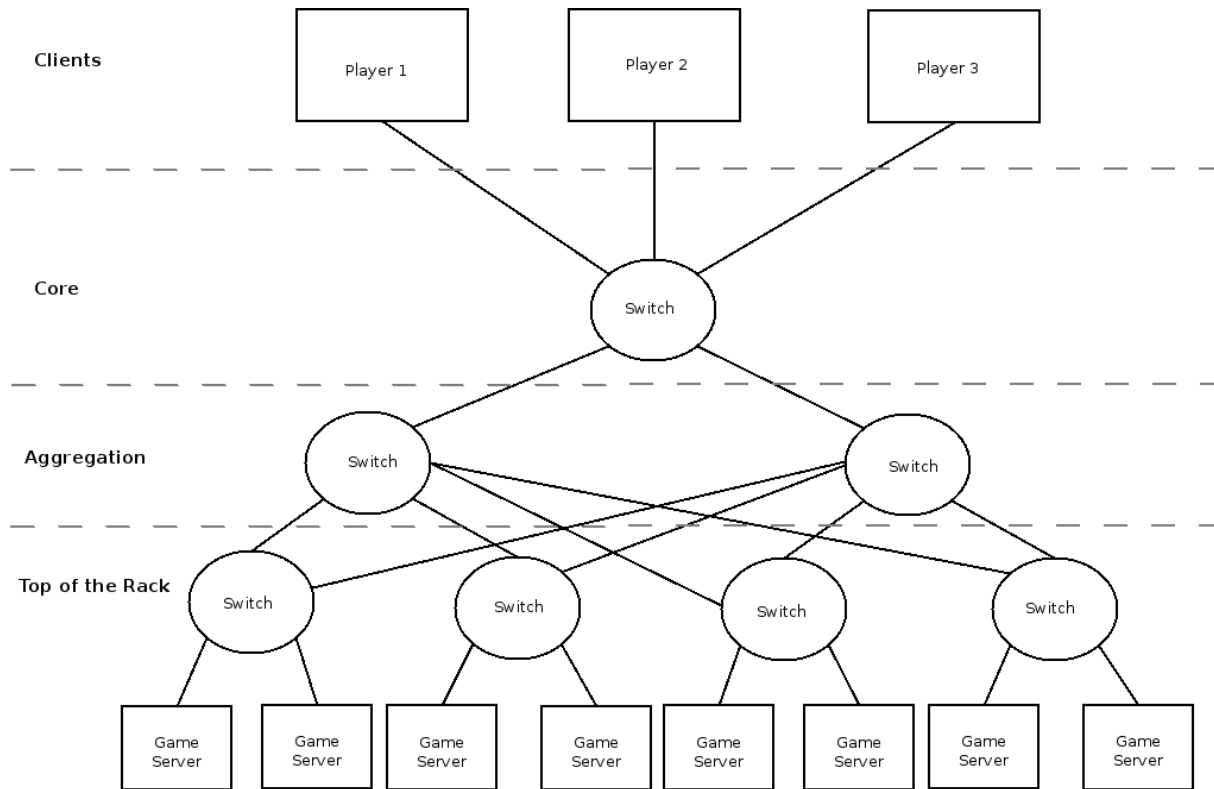


Figure 3.4: Virtual Network Topology

3.4 SDN Application Design

In a traditional cloud gaming system, several methods allow optimal paths in the network to be determined but use link status and parameters provided when the network was set up. Current states of network links when in use such as the amount of bandwidth currently available are not considered. The software-defined network application that will be used to control network traffic through a centralized network controller will aim to lower the latency. This controller will have global knowledge of the network by listening on a certain port where switches can set up a connection. It will use the information collected from switches to compute the best route for packets.

A load balancing concept aims to evenly spread traffic occurring in the network by making use of available paths between a source and destination. This in turn should lower the congestion in network links and reduce variance in network load so the user will experience less stuttering and delay. In order to find optimal paths, the shortest paths between two nodes needs to be computed. The algorithm that will be used to do this is Dijkstra's algorithm which will result in finding all the paths of the shortest length to reduce the inspection for load statistics. The path with the least load is determined from the set of all shortest path. The load will be computed by querying the switches for the amount of packets received and transmitted. The flows for the path with the lowest cost is pushed to the flow tables of all the switches in that path so new

packets will know to be routed using that path.

The main difference with this method is unlike traditional networking this process will be executed repeatedly so links costs are dynamically updated which leads to the path between the two nodes to always be the optimal path for every communication. The load balancer should only update about every minute so it will not use a lot of computation resources. This load balancing solution will make sure that all possible paths are being utilised evenly so no path is heavily congested. Using the virtual network topology above, improvement in latency should still show while performing transmission of simulated video traffic. Better results may show when using this solution with large data centre networks since more paths will be available, but it means the SDN application will have more computation to do as there will be more links and nodes that will have to be factored.

The load balancer application will communicate with the SDN controller by utilising its Northbound APIs. This will be done using REST calls that are already predefined with the SDN controller [35]. Parameters and information can be sent using XML format and responses can be received using JSON format due to lower overhead that comes with these serialized formats. As shown in Figure 3.5, the SDN network controller takes the information from the load balancer application and processes this in to instructions that can be sent to the network elements through the Southbound APIs communication protocols. OpenFlow is an example of a software adapter that the SDN controller can use to communicate with the switches [36].

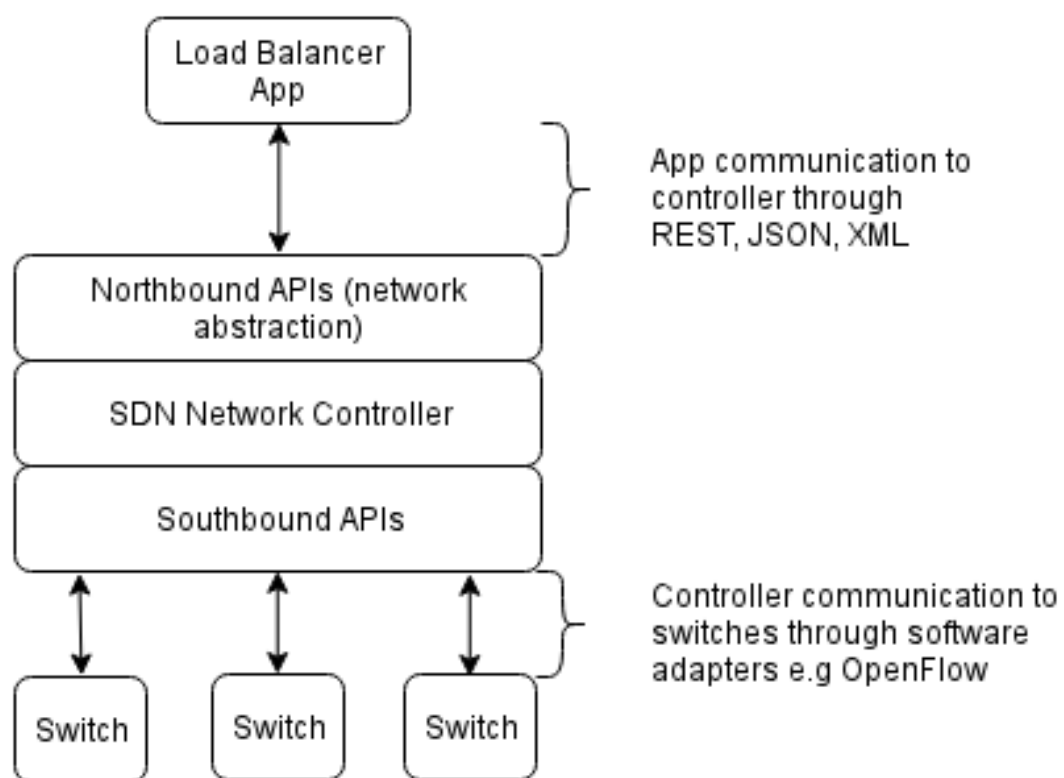


Figure 3.5: SDN Basic Architecture [26]

Chapter 4

Implementation

4.1 Game Implementation

This section will outline how the game designs in section 3.2 in the design chapter were implemented and their justification. The programming language used was C++ because it is known for its wide usage in commercial games. One of the main advantages of using C++ is that it is a low level language which gives more performance and performance is an important factor in games.

Another reason that C++ is used is that it supports many libraries and one of the libraries that is supported is OpenGL. OpenGL stands for Open Graphics Library and is used to render 3D graphics and supports multiple platforms (Windows, Linux, OSX) [27]. Since OpenGL only deals with 3D rendering, Qt is used as a framework to help build the game. Qt is a framework that is used to develop applications which also supports multiple platforms like OpenGL [22]. Qt has an OpenGL module to make developing in OpenGL easier and can take advantage of the whole Qt API for non-OpenGL specific functionality such as networking which will be explained in section 4.2.

As shown in Figure 4.1, a procedurally generated tree is being modelled along with lighting and shadows. The interface controls on the side are used for testing such as moving the tree to test the shadow and increasing the complexity of the tree. Frames per second counter is implemented and is outputted in the terminal as shown in 4.2. The flight simulator can be controlled using the following keyboard commands:

- W / S - Accelerate / Decelerate
- Q / E - Roll Left / Roll Right
- Up / Down - Pitch Down / Pitch Up
- Left / Right - Yaw Left / Yaw Right
- Space - Stop Movement
- F - Toggle FPS counter



Figure 4.1: Screenshot of game with a procedurally generated tree

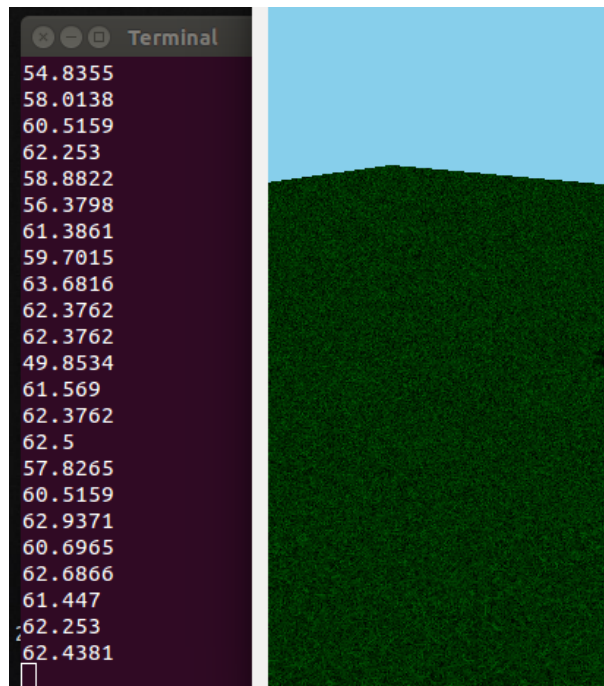
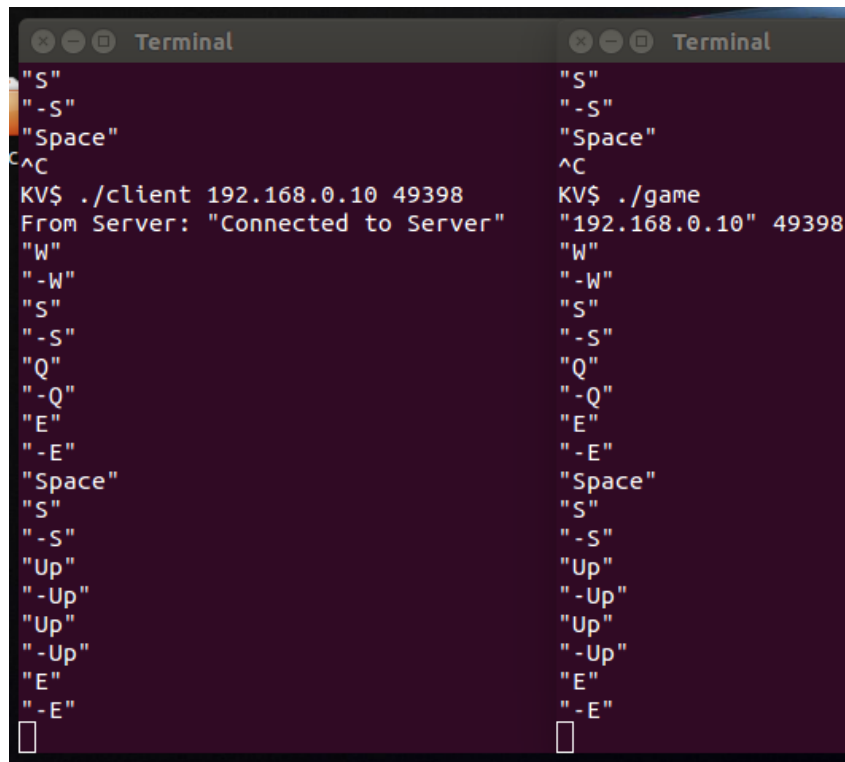


Figure 4.2: Screenshot of frames per second beind display in the terminal

4.2 Cloud Gaming

The basic cloud gaming system implementation uses Qt's framework for the networking in terms of client and server connection. Since Qt was already used for OpenGL, using Qt for networking will make communication easier. The client program is simply a Qt program that connects to the server program through IP address and port number using Qt's networking module. Once a connection is made, a Qt window appears that will listen to keyboard commands from the user. Only commands that is mapped for the flight simulator controls is accepted as well as held keys being taken in to account then these commands are sent to the server. The server accepts these inputs and simulates the game commands so the game engine can process them to produce the next frames as shown in Figure 4.3.



```

Terminal
"S"
"-S"
"Space"
^KC
KV$ ./client 192.168.0.10 49398
From Server: "Connected to Server"
"W"
"-W"
"S"
"-S"
"Q"
"-Q"
"E"
"-E"
"Space"
"S"
"-S"
"Up"
"-Up"
"Up"
"-Up"
"E"
"-E"
█

Terminal
"S"
"-S"
"Space"
^KC
KV$ ./game
"192.168.0.10" 49398
"W"
"-W"
"S"
"-S"
"Q"
"-Q"
"E"
"-E"
"Space"
"S"
"-S"
"Up"
"-Up"
"Up"
"-Up"
"E"
"-E"
█

```

Figure 4.3: Screenshot of client connection to server and sent commands

Due to time constraints and setbacks, the video streaming portion of the cloud gaming system was not completed so the initial implementation plans are outlined. The rendered frames on the server side needed to be rendered off screen so it will not display on the server machine. This is done by using Qt's `QOffscreenSurface` class that allows rendering with OpenGL on an arbitrary thread without the need to create a `QWindow` then these offscreen rendered frames are then captured in to `QImage` format [21]. Capturing frames to `QImage` format was actually already implemented but further work on it was not carried out.

The captured QImage frames are then to be encoded in to streamable H.264 MP4 video format since it is compressed so video traffic data size is smaller. This was supposed to be done using a Qt wrapper called QtFfmpegWrapper[30] for Ffmpeg which helps encode and decode video. It uses Qt QImage to exchange video frames with the encoder/decoder. Unfortunately, this code has not been updated for three years so some functions have been deprecated with recent Qt and Ffmpeg versions leading to it not being able to be compiled. With the time constraints, learning the ffmpeg library to manually encode QImage data would have been too time consuming with the networking implementations and report deliverables still to be completed.

The next step would have been to stream the video frames to the client using live555 library. The live555 media server is a complete RTSP (Real Time Streaming Protocol) server application [20]. The thin client program can then receive and play these video frames back to the window using Qt's media player RTSP compliant capabilities so the user can see the response of their keyboard commands.

4.3 Networking Implementation

The original networking implementation idea was to use the University of Leeds' School of Computing cloud testbed for testing software-defined networking in a real data centre network. Open vSwitch is an example of a virtual switch which is a software program that enables communications between virtual machines [37] which makes it essential to SDN deployments in data centres. Even though the cloud testbed is capable of using Open vSwitch, it was not enabled when it was first set up which is the only time it can be enabled. The other option is to use a virtual network to simulate a network and its latencies as well as the video traffic.

4.3.1 Mininet Virtual Network

Mininet is a tool that can be used to create a virtual network on a single machine with the ability to use virtual switches such as Open vSwitch. Latencies and bandwidth limits can be set on the links between the switches to simulate real world link delays and limitations. The bandwidth limitations that is set on the links between the player hosts and the core switch of the data centre is based on the UK's average network bandwidth in different areas [12] as shown in Table 4.1. The link delay from the players to the core switch is set to 15ms for all players since 30ms is an average ping time to a typical data centre location. The link delays in the data centre are all set to 1ms since there should be little to no latency where server machines are close to each other. Figure 4.4 shows a diagram of the mininet virtual network along with the IP address of the hosts and port numbers of switch links as well as host and switch names for easier referencing.

Connection	Bandwidth	Link Delay
h1 ->s1	70 Mbps	15 ms
h2 ->s1	50 Mbps	15 ms
h3 ->s1	30 Mbps	15 ms

Table 4.1: Table showing player host link parameters

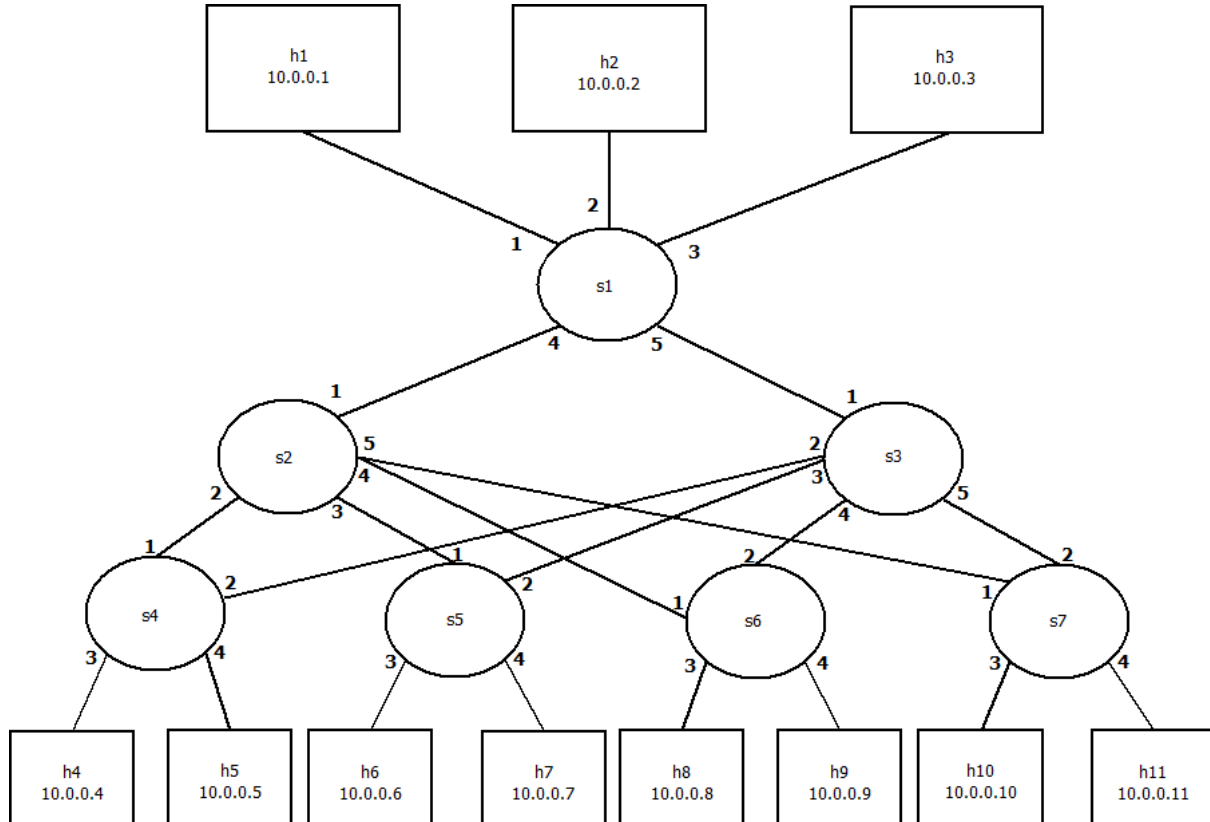


Figure 4.4: Mininet network topology with port numbers and host names and IP address

4.3.2 OpenDaylight SDN Controller

Several SDN controllers were explored such as NOX which is the first SDN controller initially developed with OpenFlow by Nicira Networks then was made open source. Other open source controllers include POX and Beacon which is one of the most popular [34]. Beacon is a Java-based OpenFlow Controller and which many current commercial controllers are based on such as Floodlight. The SDN controller that was used for this is OpenDaylight which is the controller companies such as HP, Cisco and IBM used after using controllers based off Beacon.

OpenDaylight was released in 2014 with its first code release names "Hydrogen". The version that is used in this project is OpenDaylight's fifth release which as of this project it is OpenDaylight's current version. OpenDaylight provides an easy to use command line interface and web graphical user interface for flow statistics, also it uses OpenFlow as its SDN protocol.

Mininet can connect to this controller so the virtual network can be managed by OpenDaylight. Using the pingall command in Mininet's CLI (Command Line Interface) will have every host ping all other hosts so they can discover each other. The network topology can then be easily viewed in OpenDaylight's web GUI as shown in Figure 4.5. The host machines are shown using black boxes and switches are shown using blue boxes.

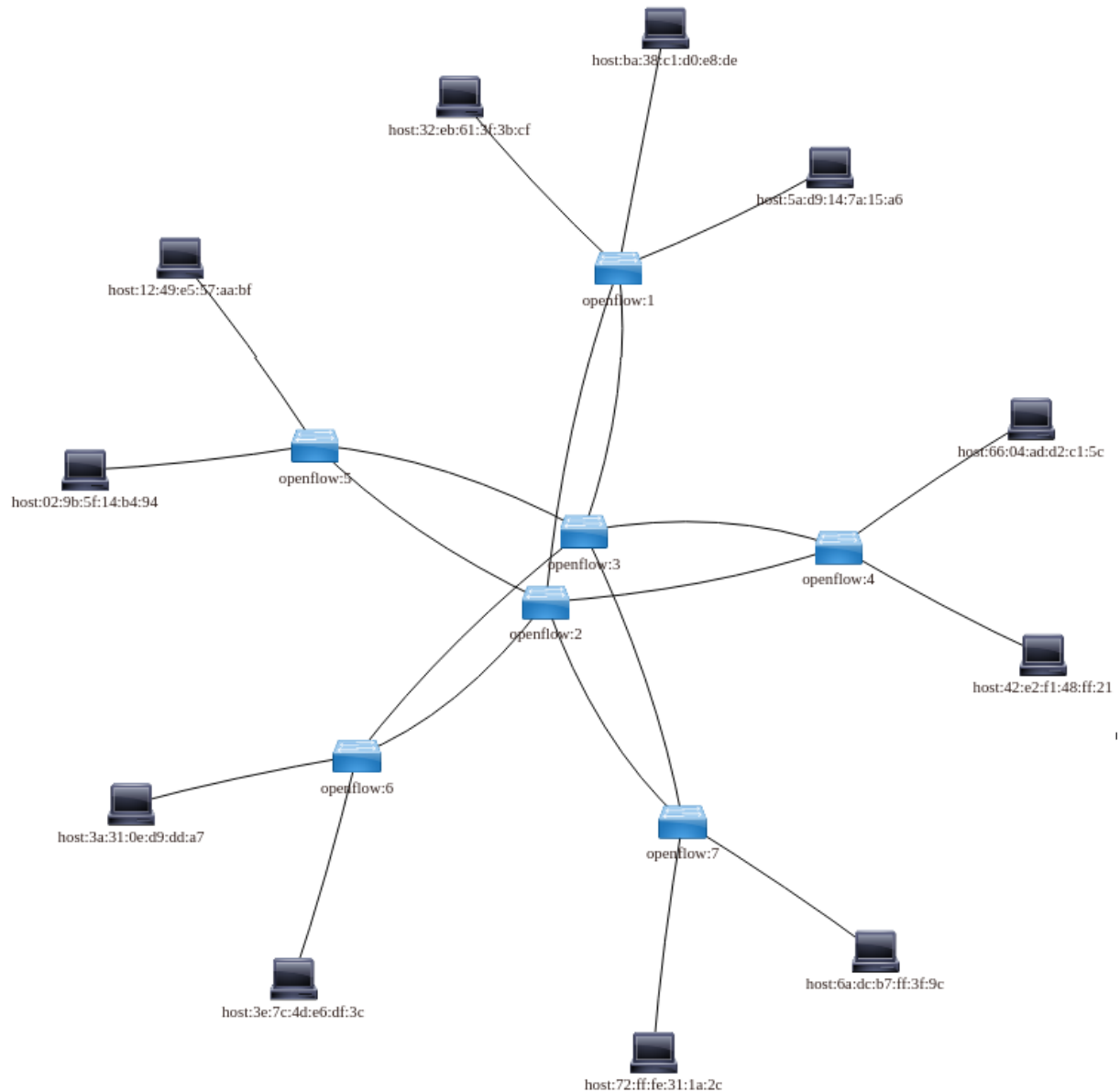


Figure 4.5: OpenDaylight network topology

4.3.3 Load Balancer

To communicate with the OpenDaylight controller, its northbound API was used through REST API calls. Initially, Postman which is a Google Chrome extension tool to make API calls was used to test out retrieval of flow statistics and to push flows. The

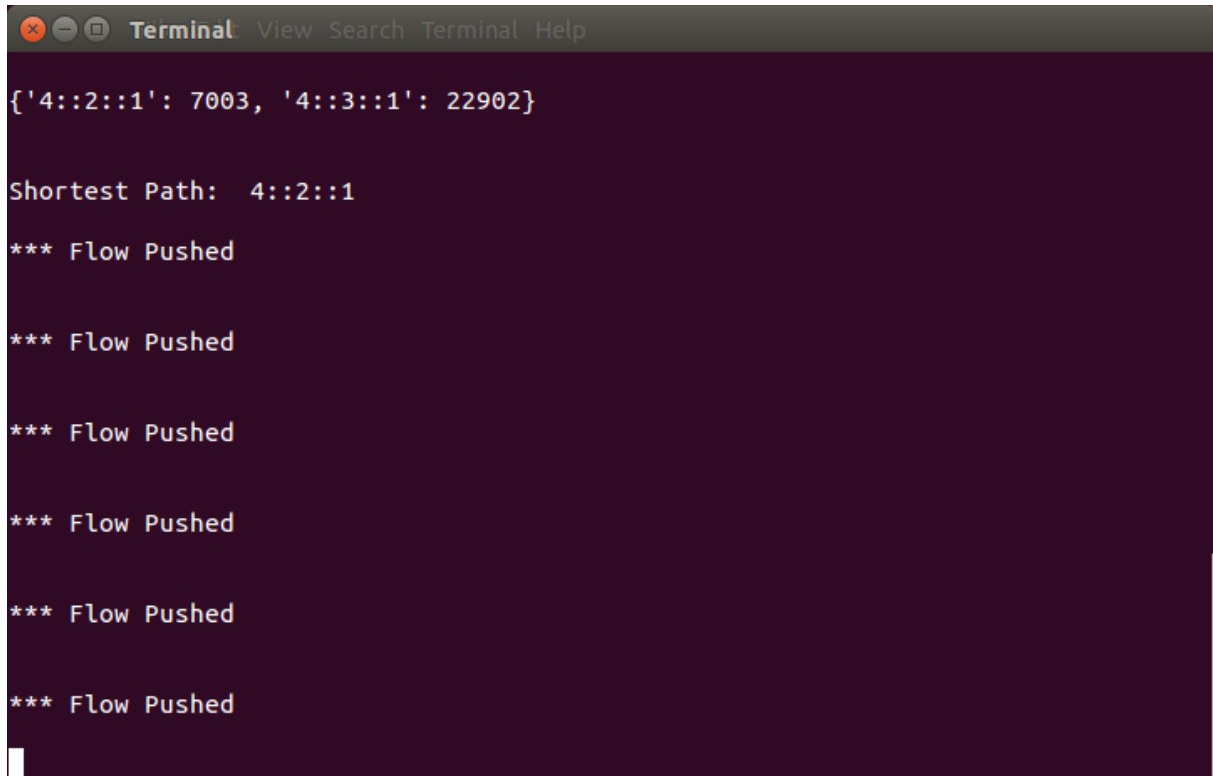
load balancer application was created in python since it can be used to easily create simple programs since it will mainly be making REST API calls. The load balancer code was based on code at <https://github.com/nayanseth/sdn-loadbalancing> and YANG UI in the OpenDaylight web GUI. The YANG UI helps in development of applications by listing and executing APIS for the SDN controller.

The load balancer application asks for user input for two machines, the source host machine and destination host machine. An API call is made to retrieve the network topology which includes the host names and switches and the links. The python package NetworkX is used to store the retrieved network. NetworkX also include a built in Dijkstra's algorithm function and is used to compute all the shortest paths. For each switch of each shortest path, multiple REST calls are made so packet statistics can be obtained. The amount of packets received and transmitted are summed up and then subtracted from the amount of packets received and transmitted two seconds later. This gives the transmission rate for that switch. The transmission rate is accumulated for every switch in a path for the cost of that path to be computed. The path with the minimum cost is then chosen to be the flow route between the host machines. This flow is then pushed to the SDN controller with REST calls.

The original code was improved on by allowing for multiple instances of load balancing as the original could only work for a single flow. If multiple flows were attempted to be pushed it would overwrite previously pushed flows. This means load balancing in the virtual network will not work created for this project will not work. The improved version allows this so multiple flows can be pushed to their shortest path for evenly spread traffic reducing congestion. These flows are deleted by a python script that loops through every switch and deletes every possible flow that could have been installed through REST API calls.

Load balancing was tested by using a network packet analyser tool called Wireshark [40]. Once load balancing is executed, Wireshark was used on a switch that is a part of the lowest load path then filter all packets that is from an IP address or going to an IP address of a host machine that is part of the load balanced path. UDP packets were being displayed in Wireshark which means that the traffic was using the load balanced path. To further confirm this, Wireshark was used on a switch that is part of a path that is not the pushed flow. Due to the design of the network, there can only be two shortest paths from a player host to a server host. This means Wireshark should not pick up any UDP packets as it should be using the path set by the application which was the case.

The following figures shows the working load balancer application using the network topology shown in 4.4. The simulated traffic is from h4 to h1 host machines and another traffic going from h5 to h2. The two possible paths for both traffic flows is through the switches s4::s2::s1 and s4::s4::s1.

A screenshot of a terminal window with a dark background and light-colored text. The window title bar shows 'Terminal' and some window control icons. The terminal output consists of several lines of text: a JSON-like object, a path selection message, and a series of flow push notifications.

```
{'4::2::1': 7003, '4::3::1': 22902}

Shortest Path: 4::2::1

*** Flow Pushed

*** Flow Pushed

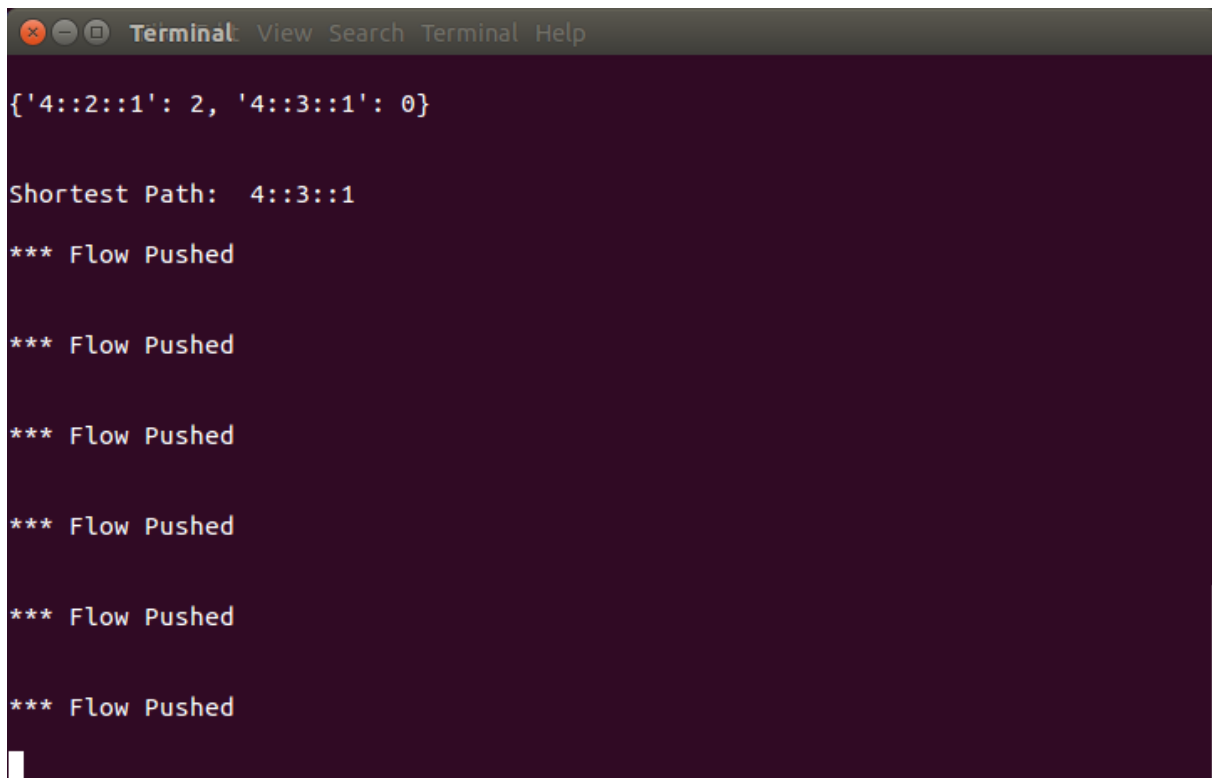
*** Flow Pushed

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed
```

Figure 4.6: Screenshot of load balancer for h4 to h1 traffic. Path s4::s2::s1 was chosen as least load path so this path was pushed to the sdn controller.

A terminal window with a dark purple background and light green text. The title bar shows 'Terminal' with standard window controls. The output text is as follows:

```
{'4::2::1': 2, '4::3::1': 0}

Shortest Path: 4::3::1

*** Flow Pushed

*** Flow Pushed

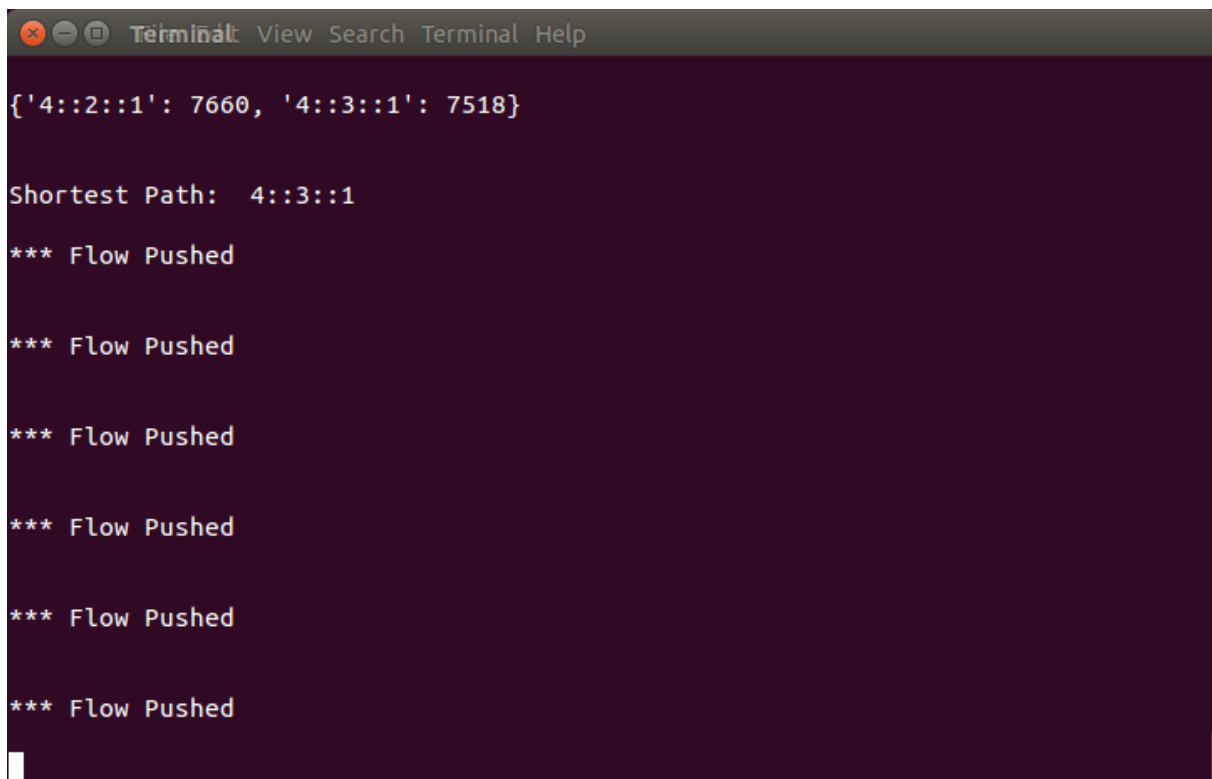
*** Flow Pushed

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed
```

Figure 4.7: Screenshot of load balancer for h5 to h2 traffic. Path s4::s3::s1 was chosen as least load path.

A terminal window with a dark purple background and light green text. The title bar shows 'Terminal' with standard window controls. The output text is as follows:

```
{'4::2::1': 7660, '4::3::1': 7518}

Shortest Path: 4::3::1

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed

*** Flow Pushed
```

Figure 4.8: Screenshot showing load balanced paths with almost equal computed costs for each path of 76660 and 7518 packets

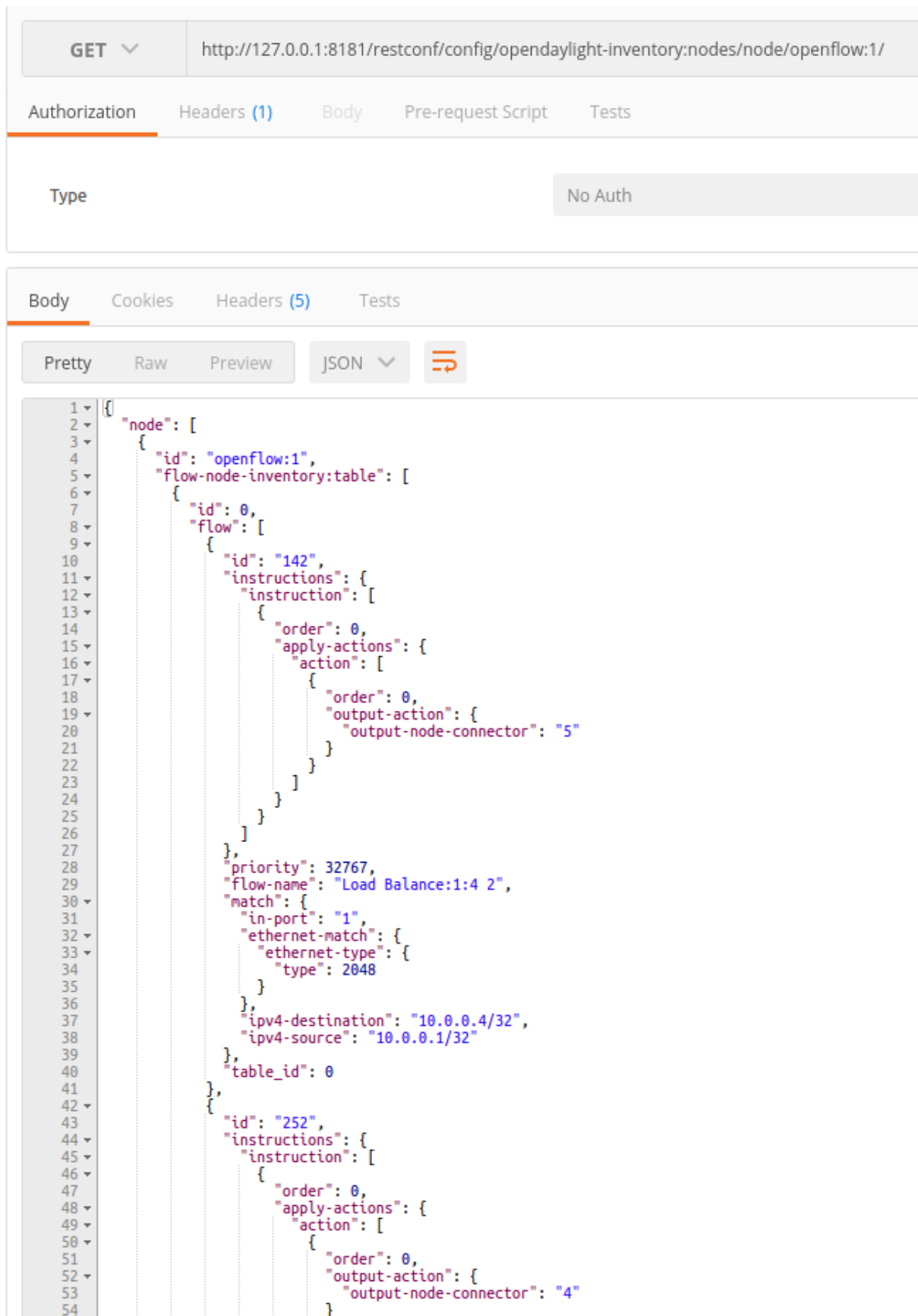


Figure 4.9: Postman request showing pushed flows in switch s1

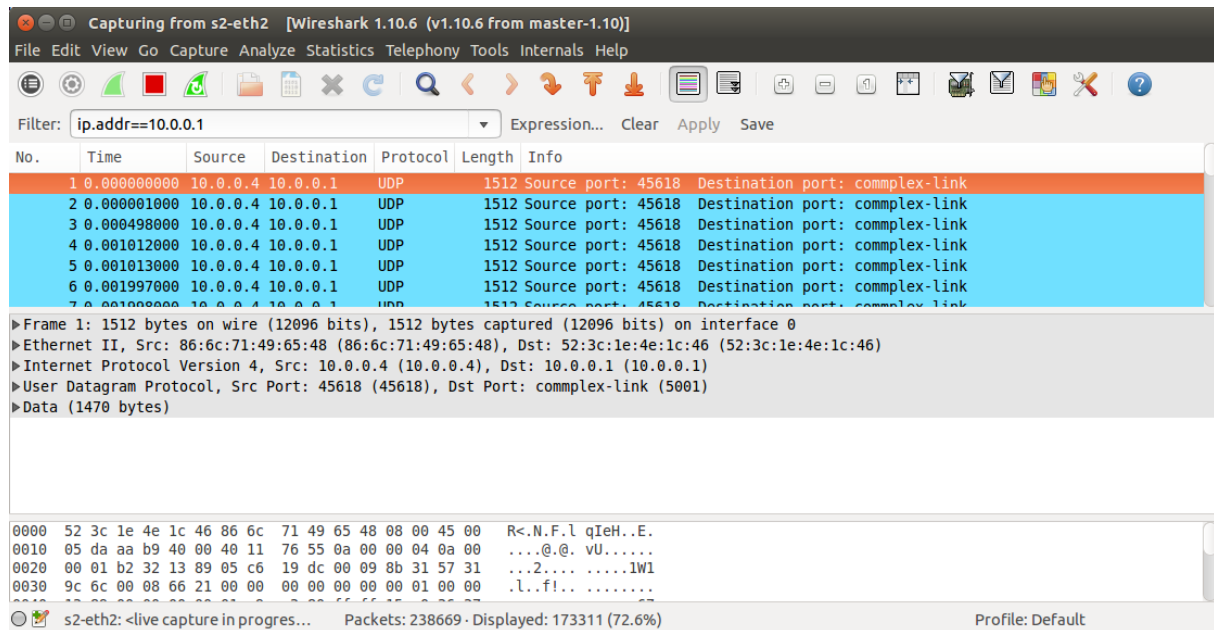


Figure 4.10: Wireshark showing UDP packets for h4 to h1 traffic going through s2 switch.

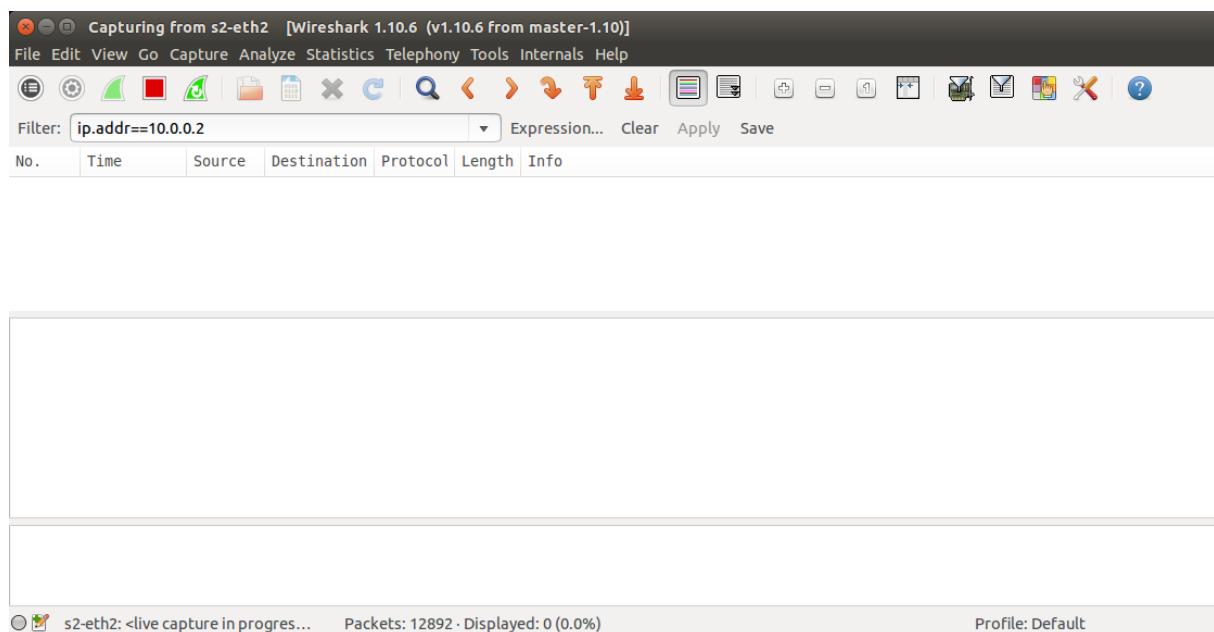


Figure 4.11: Wireshark showing no packets are going through s2 switch for h5 to h2 traffic

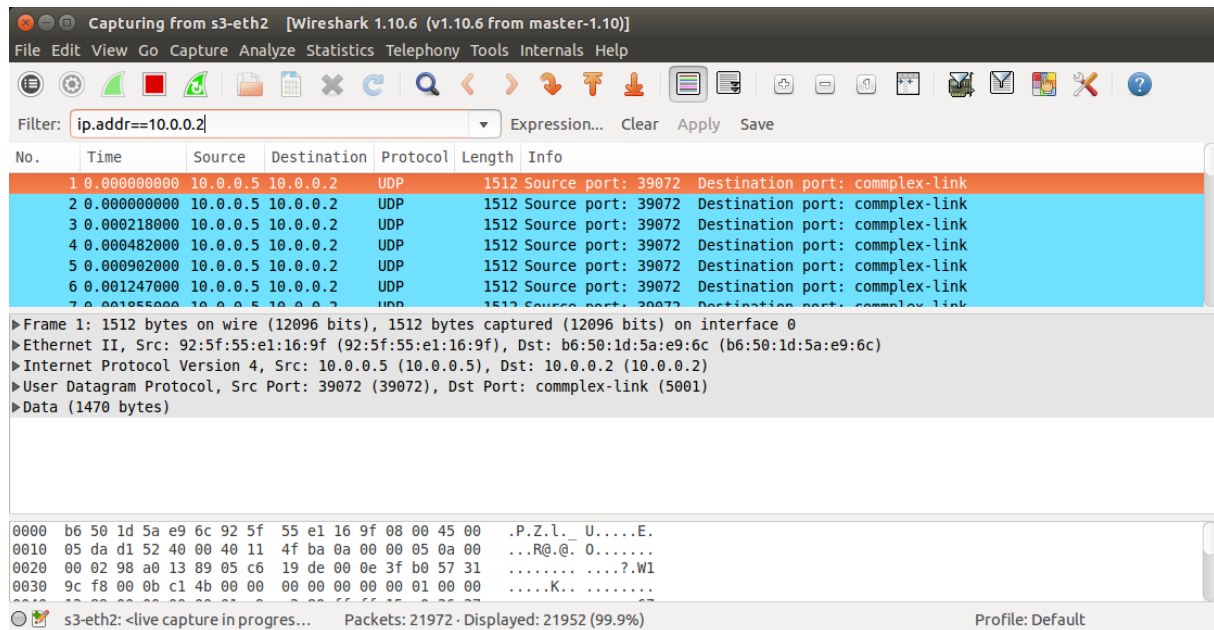


Figure 4.12: Wireshark showing UDP packets going through switch s3 for h5 to h2 traffic

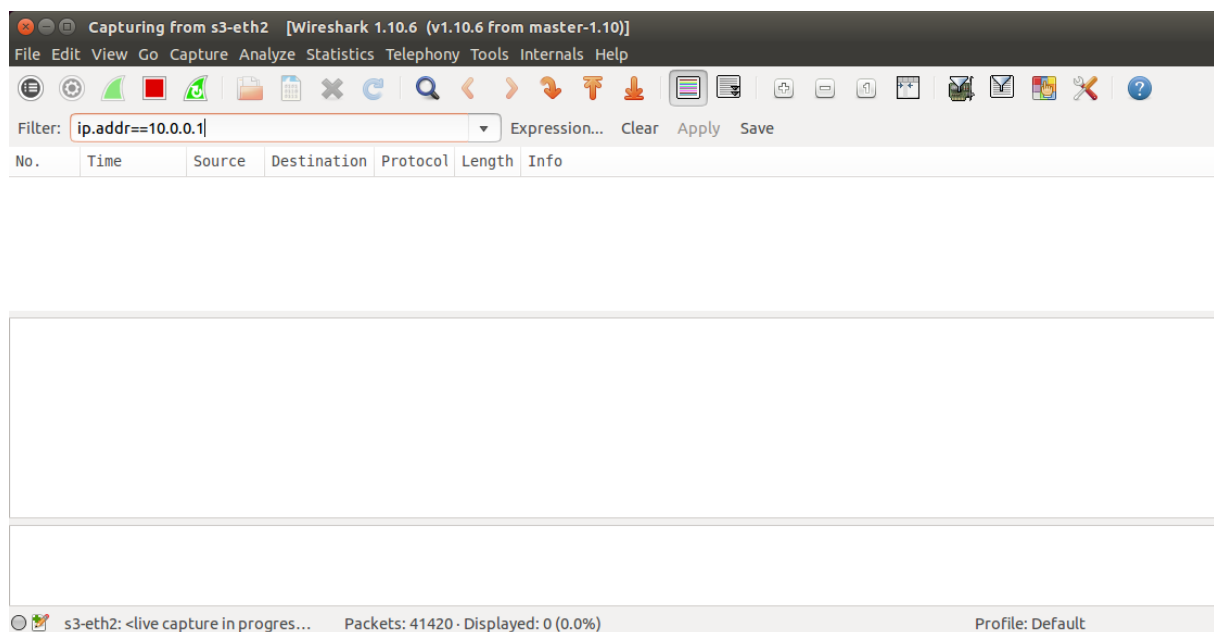


Figure 4.13: Wireshark showing no packets going through switch s3 for h4 to h1 traffic

Chapter 5

Testing and Evaluation

5.1 Test Cases

In order to test the load balancing application, test scenarios were designed to see how it would effect real world cases. Using the virtual network created with the mininet program and the iperf network bandwidth measurement tool, traffic is generated between the player hosts and game server hosts. The bandwidth of this traffic will be based on network requirements suggestions for 1080p at 60FPS, 720p at 60FPS and 720p at 30FPS given by nVidia's GeForce Now cloud gaming system requirements [25] as shown in Table 5.1. The iperf tool used to generate the traffic will set the player hosts in server mode and the game servers in client mode. This is so the game server can send UDP traffic to it's connected player host machine. UDP is used since its traffic bandwidth can be set and is the protocol that is used for streaming video data over TCP.

Latency will be simply tested by sending 10 ping requests from player host to game server whilst the iperf traffic is being sent. Each test case will contain multiple tests using each of the settings in Table 5.1 with tests before load balancing and then with load balancing in effect. To make sure th tests without load balancing doesn't depend on routes specified by previous load balanced runs, flows created by previous load balanced runs are deleted using a python script that makes REST API calls to the OpenDaylight SDN controller.

Video Settings	Network Bandwidth
1080p + 60FPS	50 Mbps
720p + 60FPS	30 Mbps
720p + 30FPS	10 Mbps

Table 5.1: Traffic Simulation Settings

5.1.1 Case 1

The first case is two players using the cloud gaming system with their game instance being in two separate servers but under the same switch. Player host 1 'h1' (10.0.0.1) and player host 2 'h2' (10.0.0.2) are used and are connected to 'h4' (10.0.0.4) and 'h5' (10.0.0.5) respectively. This shows a basic example of multiple players connected to different game servers but using the same Top of the Rack switch. As shown in Figure

5.1, there are two possible routes for traffic flowing from the core switch 's1' to switch 's4' (s1::s2::s4 and s1::s3::s4). When the load balancer is running for each flow, it will assign a different route for each of them. Figure 5.1 shows one out of the two possible load balanced routes with the other being just a swap with the red and blue path between 's1' and 's4' switches.

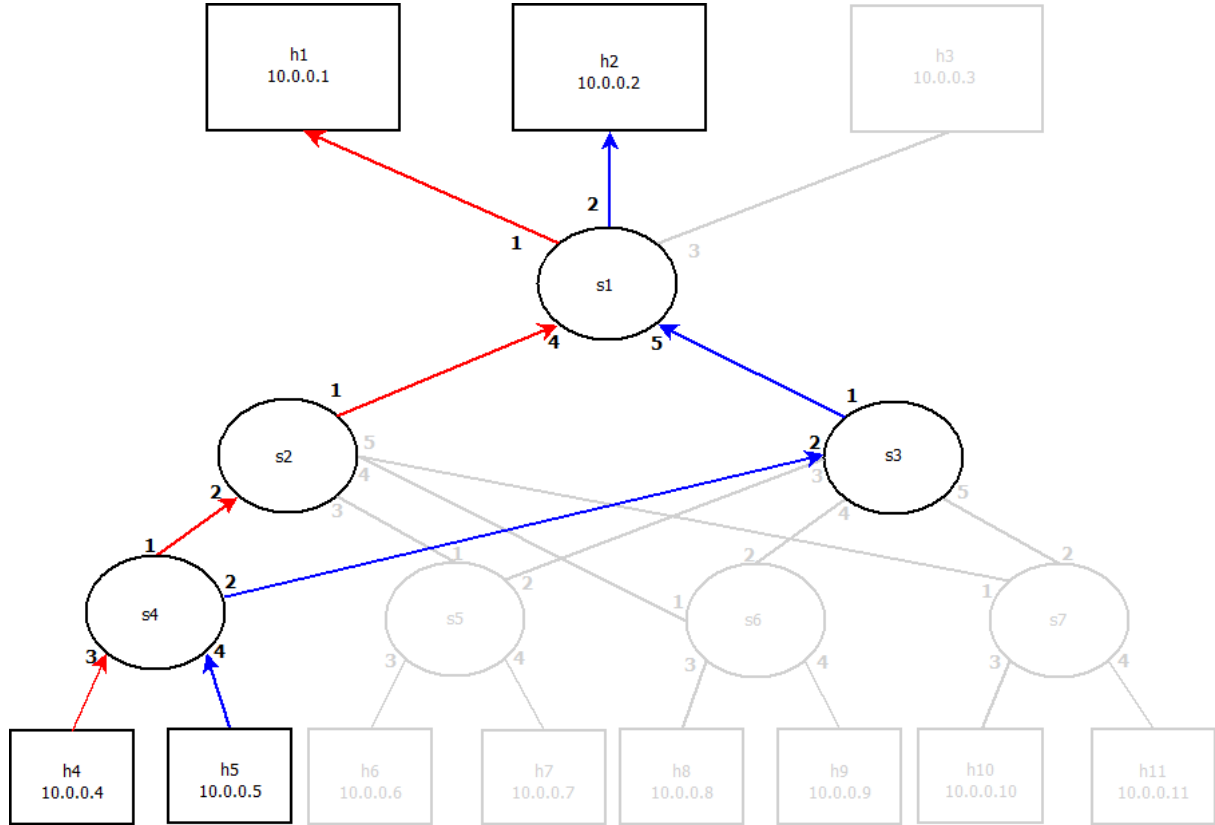


Figure 5.1: Test Case 1: Possible load balanced routes

5.1.2 Case 2

The second case is when two players are connected to the same physical game server host since it can contain multiple virtual machines with multiple game instances. Also it can represent two players playing the same game instance such as a multiplayer game so two video traffic will be needed to be generated and sent to the two separate clients. For this test case, hosts 'h1' and 'h2' will be used to be connected to host 'h4'. The possible load balanced routes between the switches will be the same as in test case 1 but with two separate traffic being generated from 'h4' as shown in Figure 5.2.

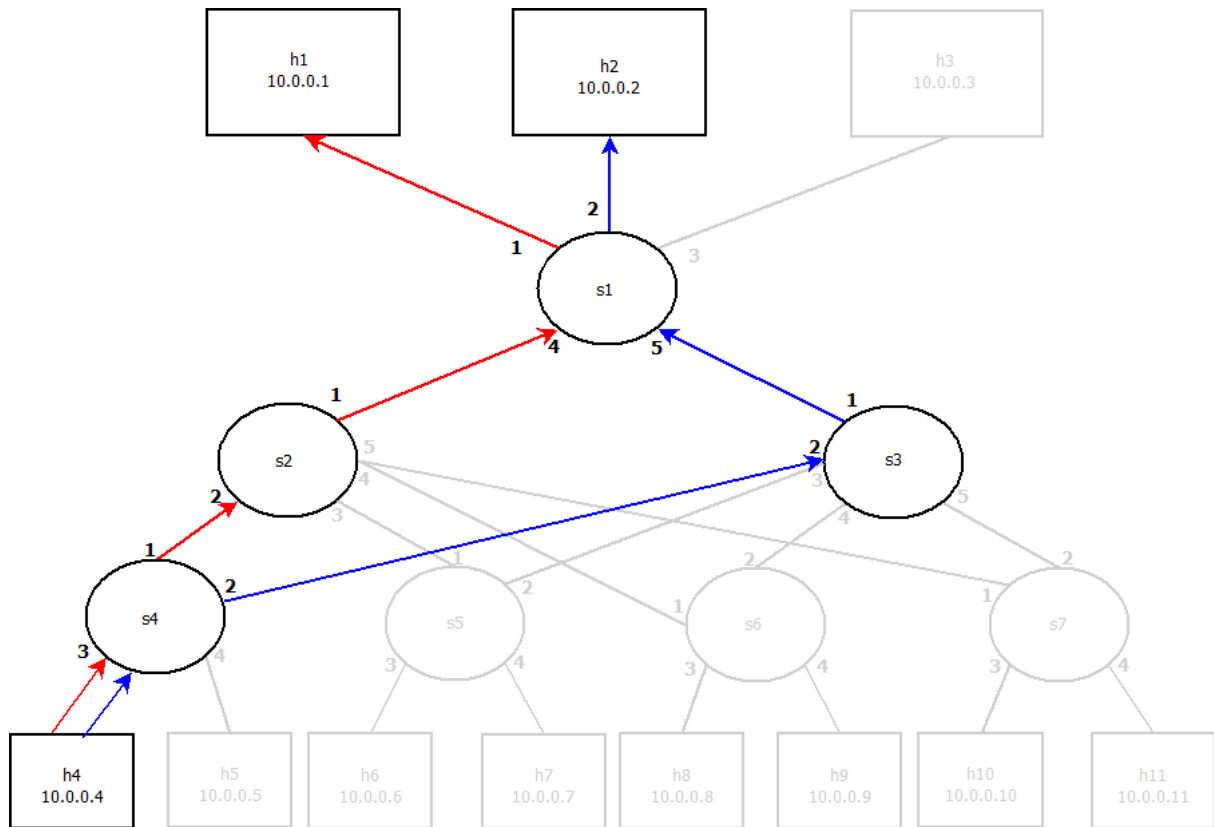


Figure 5.2: Test Case 2: Possible load balanced routes

5.1.3 Case 3

The third test case uses three players simultaneously connected to the cloud data centre. Game server hosts 'h4', 'h5' and 'h6' each generate traffic and are routed to the player host 'h1', 'h2' and 'h3' respectively as shown in Figure 5.3. This test case represents a scenario of a highly congested network with multiple traffic flows and possible paths. The results of using the load balancer application in this test case provides a good insight on it's capabilities and effectiveness.

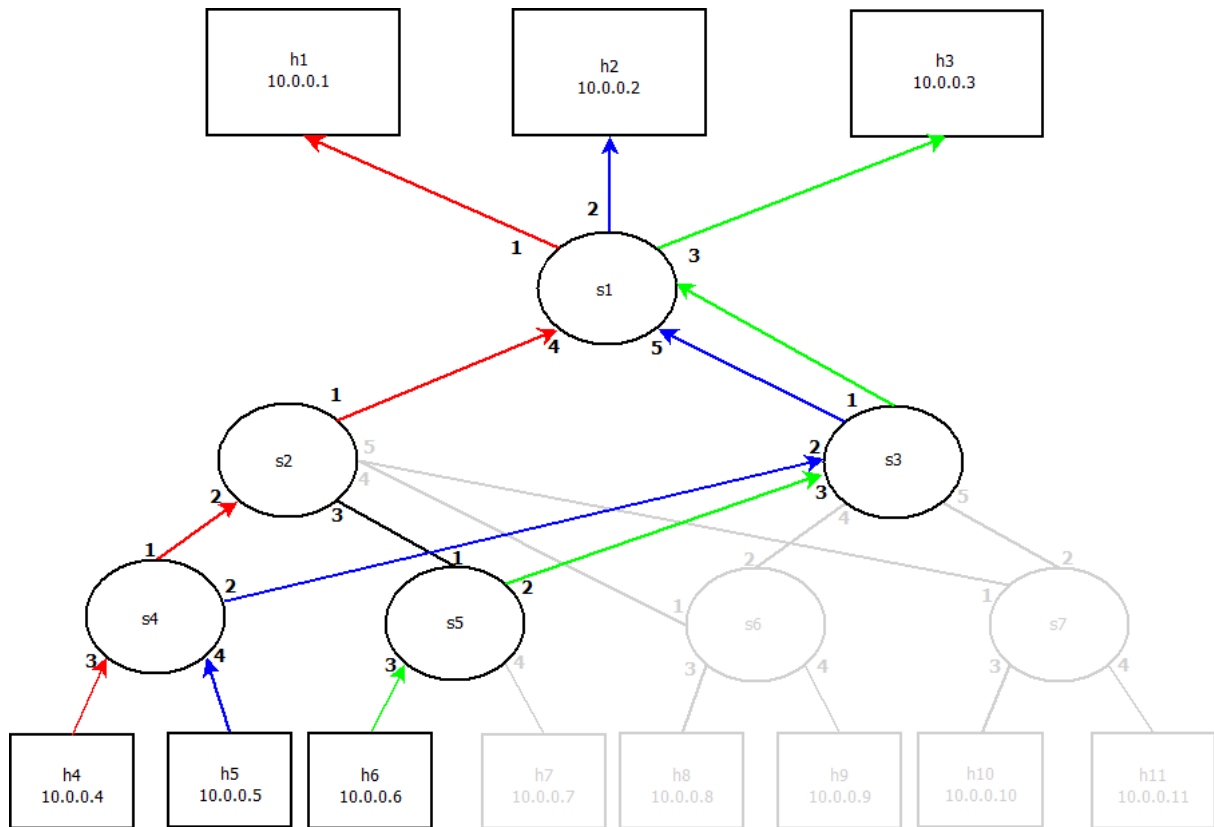


Figure 5.3: Test Case 3: Possible load balanced routes

5.2 Results

This sections shows the results of each test case scenario. The tables contains results collected from the ping tests which shows the different round trip times summary with 10 ping packets. The minimum round trip time as 'min', maximum as 'max', average as 'avg' and deviation as 'mdev'. The deviation result is the average of how far each ping RTT is from the mean RTT. The higher 'mdev' is, the more the RTT changes which results to a poorer user experience. Since the focus of this project is on reducing the network latency in a data centre for cloud gaming, the charts below the tables show a better representation of how the load balancing application affected the delay under certain loads.

	Before Load Balancing				With Load Balancing			
Connection	min	avg	max	mdev	min	avg	max	mdev
50 Mbps								
h1 ->h4	210.317	223.223	244.866	9.440	37.409	40.268	47.533	2.685
h2 ->h5	278.072	280.276	281.873	1.522	265.200	269.559	274.839	2.182
30 Mbps								
h1 ->h4	57.012	75.600	108.208	15.111	36.755	37.891	41.927	1.428
h2 ->h5	274.877	280.685	286.498	4.096	36.628	38.28	45.249	2.373
10 Mbps								
h1 ->h4	41.814	43.108	46.786	1.469	37.211	38.107	39.057	0.596
h2 ->h5	42.486	43.516	44.502	0.675	37.550	38.200	39.239	0.653

Table 5.2: Test Case 1 Results

	Before Load Balancing				With Load Balancing			
Connection	min	avg	max	mdev	min	avg	max	mdev
50 Mbps								
h1 ->h4	268.850	296.824	320.158	18.250	209.526	213.216	216.632	3.011
h2 ->h4	357.026	377.026	404.797	20.273	272.451	274.735	275.937	1.167
30 Mbps								
h1 ->h4	213.000	221.689	229.219	6.238	38.468	39.232	39.960	0.448
h2 ->h4	296.504	302.919	321.211	7.733	38.525	39.243	39.995	0.469
10 Mbps								
h1 ->h4	42.832	43.443	44.234	0.484	39.985	38.016	38.572	0.432
h2 ->h4	41.622	42.396	44.543	0.936	36.992	38.007	39.062	0.739

Table 5.3: Test Case 2 Results

Connection	Before Load Balancing				With Load Balancing			
	min	avg	max	mdev	min	avg	max	mdev
50 Mbps								
h1 ->h4	208.303	216.48	220.662	4.465	37.631	40.998	49.488	3.256
h2 ->h5	269.297	273.377	277.458	4.113	269.133	272.394	274.474	2.096
h3 ->h6	426.205	435.244	443.197	6.979	421.068	424.151	425.826	1.589
30 Mbps								
h1 ->h4	205.742	210.754	216.244	3.768	36.542	38.055	40.507	1.168
h2 ->h5	271.333	273.377	278.936	2.629	36.397	37.288	38.337	0.657
h3 ->h6	425.217	427.494	431.469	2.499	425.185	426.292	430.231	1.489
10 Mbps								
h1 ->h4	41.517	42.602	44.918	0.966	37.082	38.156	40.023	0.975
h2 ->h5	41.602	43.132	44.731	0.943	37.598	38.386	38.999	0.387
h3 ->h6	426.631	428.373	430.332	1.518	36.981	37.702	38.944	0.523

Table 5.4: Test Case 3 Results

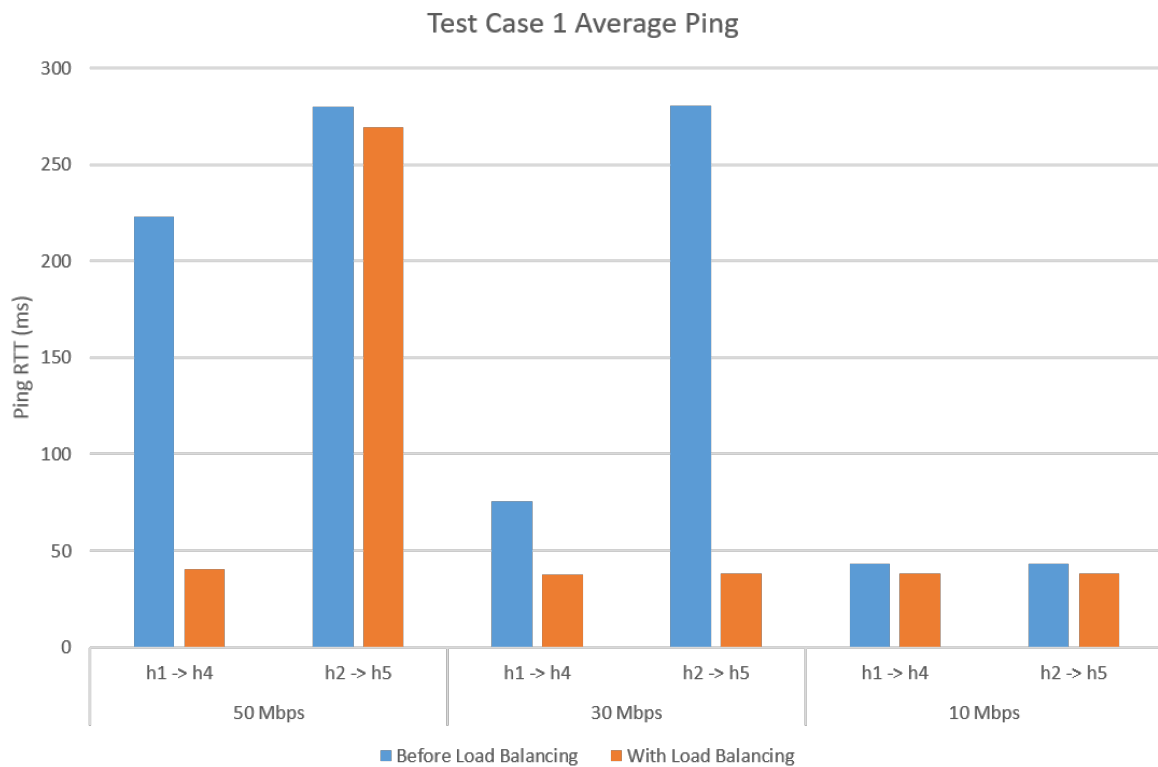


Figure 5.4: Test Case 1: Average ping

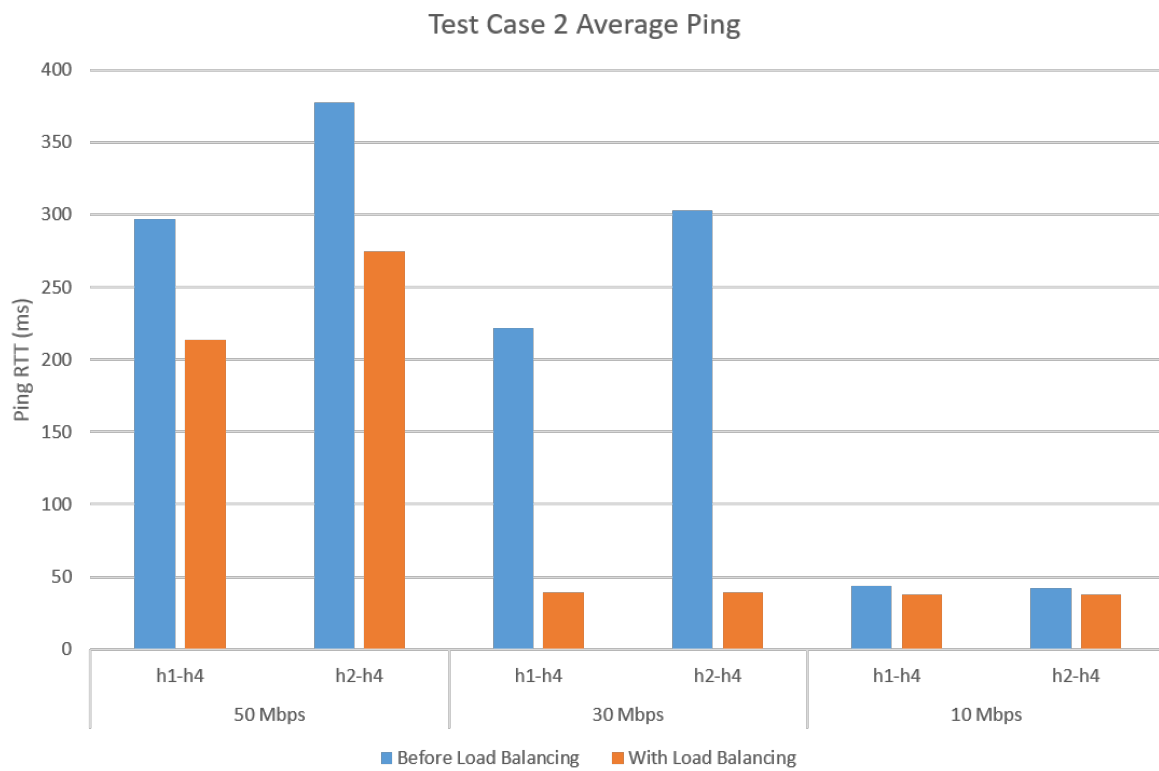


Figure 5.5: Test Case 2: Average ping

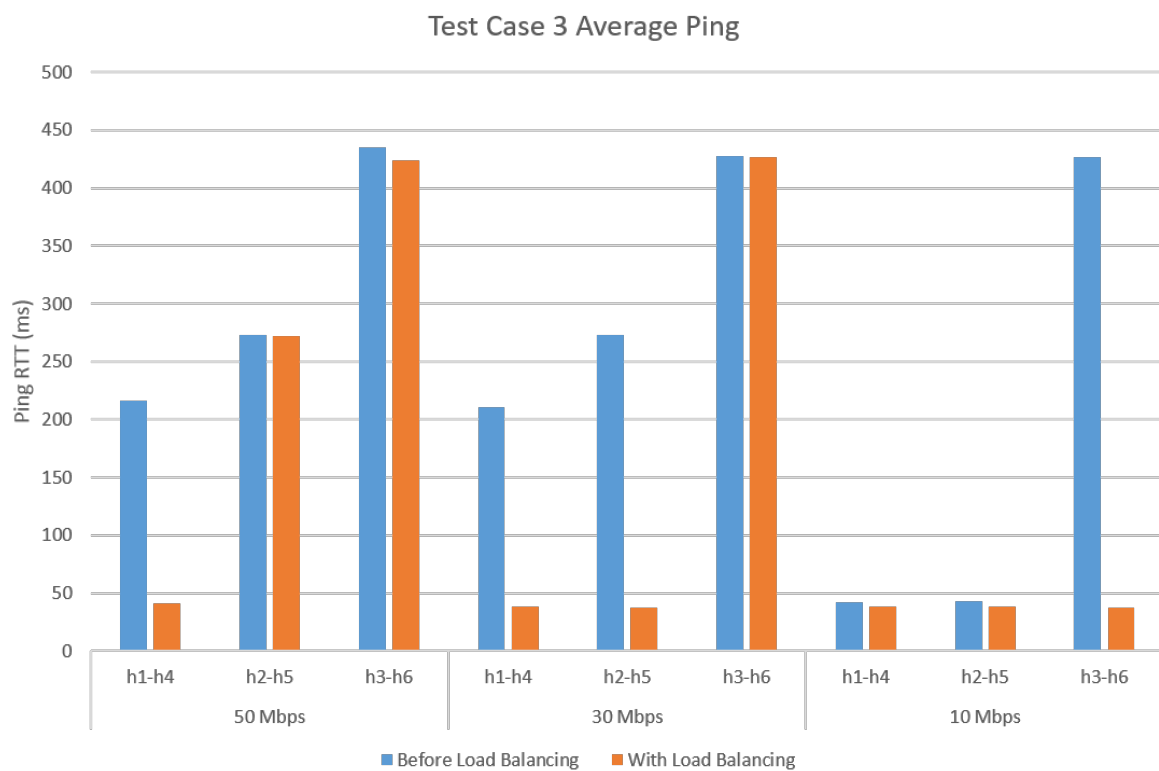


Figure 5.6: Test Case 3: Average ping

5.3 Evaluation

In test case 1, the load balancer application assesses the current traffic in the shortest routes between the core switch 's1' and the ToR switch 's4' for hosts 'h4' and 'h5'(s1::s2::s4 and s1:s3::s4). By calculating the the amount of packets being received and transmitted, it picks the route with the lowest traffic and uses it for the flow assigned for the load balancer instance. The load balancer instances for each flow once fully functioning uses a separate route for each flow. This is how the ping times decreased after load balancing. For the 50 Mbps test in Table 5.2, the average ping time for the 'h1' to 'h4' connection decreased from 223ms to 40ms when load balancing is used but the 'h2' to 'h5' connection only decreased from 280ms to 265ms. This is likely due to the bandwidth limit on 'h2' to core switch link being the same as the traffic throughput of 50 Mbps. The 30 Mbps test further supports this since load balancing decreased 'h2's' connection ping from 280ms to 38ms. Even with 10 Mbps, there is still some latency before load balancing at around 43ms but this was reduced to 38ms when load balancing is used. This is very close to the best possible ping time theoretically possible in this virtual network of 36ms with its built in latency in the links.

Ping times are generally higher in test case 2 compared to test case 1 as shown in table 5.3 since the traffic source is the same host even though the same routes are being used as in test case 1. the effects of the load balancing application is not as effective in the 50 Mbps test. Test case 1 load balancing reduced the average ping time for 'h1' to 'h4' connection to 40ms, but test case 2 it as only been reduced to 213ms. This is still an 86ms reduction which means that the load balancing application is still useful. The average ping time using 30 Mbps traffic before load balancing for the 'h1' to 'h4' connection for test case 1 is 75ms and dramatically increases in test case 2 to 221ms. Load balancing has reduced both connections to 39ms. The 10 Mbps results are similar to the results in test case 1, but slightly better.

For test case 3, the ping times shows in Table 5.4 is similar to the results in test case 1 for the connections 'h1' to 'h4' and 'h2' to 'h5'. On the other hand the third connection in this scenario suffers a lot from long delay with ping times over 420ms before load balancing. Even with load balancing, average ping times are still over 420ms when 50 Mbps and 30 Mbps traffic is used. Load balancing greatly reduced the ping time in the 10 Mbps test, from 428ms to 37ms. This findings further supports the claim made above in test case 1, where the bandwidth limit in the 'h3' link to the core switch of 30 Mbps cannot handle the throughput being pushed.

According to the test results, the dynamic load balancing application has seemed to reduced ping times in every scenario. It also shows that in every case that when the

throughput of the simulated traffic is high, the ping times are also high which is due to the high congestion in the routes. Another finding is that when traffic is forced on to connections with not high enough bandwidth to support it, latency increases and load balancing does not help to reduce it or it does not reduce it enough to make the game experience more playable.

Chapter 6

Conclusion

6.1 Conclusion

For this project, the problem that was undertaken was to explore the feasibility and the effects of using software-defined networking in a cloud gaming system. In order to do this a cloud gaming system and a game was implemented to better understand latency and how it arises in cloud gaming. Background research and literature review was conducted so the problem can be specified and narrowed down to a size where the project can be undertaken with the time constraints. Also, research was needed to be done on the feasibility of the solution to see if it could be done with current technical and programming skills and knowledge.

A simple game with flight simulator controls was designed and implemented. The game also modelled a real time procedurally generated tree with lighting and shadows which uses expensive compute resources so moving it to a more powerful data centre would be beneficial. A cloud gaming system was attempted to be implemented but was not fully accomplished. A solution was complete up to the point of sending user inputs and simulating them to the game on the server side. The video streaming of the game frames to the client was not finished which meant the amount of traffic produced by it could not be recorded. This was supposed to be used with the software-defined network tests to generate traffic of the same throughput.

For the networking portion of the solution to the problem, a virtual network was created and used to simulate a data centre network that could have been used in a cloud gaming system along with latency and bandwidth limits in the network links. Software-defined networking is then used by deploying a load balancer application on to the network with the help of a SDN controller. The load balancer finds the shortest paths between two nodes in the network and computes the usage of the links to find the least used route. This route is then pushed to the SDN controller so any traffic between the two nodes use this path. Multiple instances of this load balancer can be running so every flow can be load balanced also it will update repeatedly so the load can be balanced even when new flows are introduced which made it dynamic.

With the results from the different test scenarios, it has shown that deploying load balancer using software-defined networking helps with reducing latency in a network. Under different loads and scenarios, load balancing generally improved the ping times

between player hosts and game servers. It was concluded that in the tests where ping times did not improve or improved by a small amount, it was due to trying to force throughput higher than the bandwidth limit in the link. A limitation with the tests is that it was conducted with simulated networks, traffic and parameters which cannot fully replicate a real network so results may vary in a real case. With this aside, the project has shown that software-defined networking in terms of dynamic load balancing improves network latency in a cloud gaming system so therefore a player's gameplay experience will be improved with this reduction of lag.

6.2 Future Work

There are many areas in the project which can be improved on or how it could be used to carry out further research. This section will outline this and how it would give better, reliable results and insight to the problem.

One of the improvements is to complete the cloud gaming system by implementing the video streaming aspect. This is to get the rendered frames on the server side encoded in to video streaming format and sent to the client where the client program displays it on the window. The planned implementation ideas were outlined in section 4.2 and if this was completed the video traffic produced can be recorded using a packet analyser like Wireshark which would give more accurate simulated traffic in the test cases. Also, interaction delay which is the amount of time for a button to be pressed and the corresponding action displayed on the screen can then be measured.

Another improvement that could be made is to deploy the solution on a real world data centre which has support for software-define networking. This means implementing a fully working cloud gaming system where it would create a virtual machine and run a separate game instance for each connected client. Also, a fully functioning software-defined networking solution to be deployed on the data centre where it would automatically run a load balancer instance for each new client and have it run in the background whilst constantly updating for new load balanced routes. This will give the most accurate test results since it is deployed on a real physical network with no simulated data.

An improvement that would be interesting to implement once a full is to develop a game that is capable of multiple players running the same game instance so essentially a multiplayer game. With multiple players communicating with the same game engine and logic, seeing how deploying this on a cloud gaming system affects the latency and how effective load balancing is with this case will give results that would be helpful for future cloud multiplayer gaming implementations.

6.3 Personal Reflection

In this project, there were aspects that proved to be more difficult than initially expected and this section will highlight them and methods to avoid or reduce their effects if the project or a similar project was to be undertaken again.

One of the areas that I deemed to be the most difficult is in the beginning stages where the problem the project is trying to solve needs to be specified. At first, I was under the impression that the project would just consist of developing a game and cloud gaming system that would offload the computation on the server side. I found it difficult to find a problem case that I could handle with my technical skills under the imposed time constraint. Background research proved to aid finding a problem and a feasible solution to it. The problem was not enough research was done early on and not having specific problem and solution to work with brought up other problems.

A problem that came up is that many solutions that was attempted in the implementation stage came to a dead end. These road blocks were met and more background research had to be done to find alternative solutions. The main example of this is with the use of the School of Computing cloud testbed. From early on in the project I was already preparing the environment and virtual machines that will be used for deploying the cloud game system and game. There was problems in connecting with the cloud remotely off campus and problems with trying to get a working OpenGL environment. Another problem was when trying to get the software-defined networking controller and virtual switches working, but was not successful upon finding out Open vSwitch not being enabled. So a virtual network was the alternative solution at the end, but a lot of time was consumed due to the lack of a solid plan and research. If knowledge of SDN being used along with its requirements was known earlier on then there would have been more time on other aspects of the project.

Another challenging aspect of the project was learning software-defined networking. With the distributed systems module and past internship with HPC in the university, I already had prior knowledge of networking and data centre architecture, but software-defined networking was a new paradigm that I had trouble understanding. I tried working with SDN controllers and virtual switches with not enough background reading and lead to a slower learning rate.

On the other hand, an area that went smoothly is during the developing stage of the game. This is because of the knowledge and experience from the computer graphics module that was recently completed. I managed to complete the development of the game within the allotted time frame given in the project schedule. Previous

development in OpenGL and Qt was very useful in producing the game which gave more time to be spent on the rest of the project

In conclusion, the project was a challenging experience with its many setbacks, but rewarding due to the knowledge gained from it. An important factor to take away from this project is to efficiently conduct background research and plan early before getting too deep with the development. Without proper understanding of the problem and preparation for the development stage, many routes in developing can lead to dead ends and be time consuming. What I felt that was done effectively was adapting to these problems and finding alternative solutions.

References

- [1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.
- [2] M. Amiri, H. Al Osman, S. Shirmohammadi, and M. Abdallah. Sdn-based game-aware network management for cloud gaming. *Proc. ACM/IEEE Network and Systems Support for Games, Zagreb, Croatia*, 2015.
- [3] M. Amiri, H. Al Osman, S. Shirmohammadi, and M. Abdallah. An sdn controller for delay and jitter reduction in cloud gaming. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*, pages 1043–1046. ACM, 2015.
- [4] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151. ACM, 2004.
- [5] H. Carr. 15: Natural phenomena: Terrain & plants. University Lecture, 2015.
- [6] S.-P. Chuah, C. Yuen, and N.-M. Cheung. Cloud gaming: a green solution to massive multiplayer online games. *Wireless Communications, IEEE*, 21(4):78–87, 2014.
- [7] P. Deutsch. The eight fallacies of distributed computing. URL: <http://today.java.net/jag/Fallacies.html>, 1994.
- [8] GameCentral for Metro.co.uk. The witcher 3 wins game of the year at golden joysticks 2015, 2015. [Online]. [Accessed 15 April 2016]. Available from: <http://metro.co.uk/2015/10/30/the-witcher-3-wins-ultimate-game-of-the-year-at-golden-joysticks-2015-5471812/>.
- [9] Gaming Anywhere. Cloud gaming architecture, 2014. [Online]. [Accessed on 18 April 2016]. Available from: http://gaminganywhere.org/gaming_anywhere.files/arch.png.
- [10] R. L. Grossman. The case for cloud computing. *IT professional*, 11(2):23–27, 2009.
- [11] K. Hawkins and D. Astle. *OpenGL game programming*. Cengage Learning, 2001.
- [12] M. Jackson. Ofcom 2016 report - average uk home broadband speeds, 2016. [Online]. [Accessed on 7 May 2016]. Available from:

- <http://www.ispreview.co.uk/index.php/2016/03/ofcom-2016-report-average-uk-home-broadband-speeds-reach-28-9mbps.html>.
- [13] Y. Jadeja and K. Modi. Cloud computing-concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, pages 877–880. IEEE, 2012.
 - [14] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld. An evaluation of qoe in cloud gaming based on subjective tests. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 330–335. IEEE, 2011.
 - [15] A. Kharatian. E3 2012: Nvidia’s grid changes the face of cloud gaming, 2012. [Online]. [Accessed on 18 April 2016]. Available from: <http://nerdreactor.com/wp-content/uploads/2012/06/gridlatency.jpg>.
 - [16] H. Kim and N. Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013.
 - [17] K. Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.
 - [18] A. Kumawat. Cloud service models (iaas, saas, paas) + how microsoft office 365, azure fit in, 2013. [Online]. [Accessed on 22 April 2016]. Available from: <http://www.cmswire.com/cms/information-management/cloud-service-models-iaas-saas-paas-how-microsoft-office-365-azure-fit-in-021672.php>.
 - [19] K. Lee, D. Chu, E. Cuervo, J. Kopf, A. Wolman, Y. Degtyarev, S. Grizan, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. *GetMobile: Mobile Computing and Communications*, 19(3):14–17, 2015.
 - [20] Live Networks, Inc. The live555 media server, 2015. [Online]. [Accessed on 7 May 2016]. Available from: <http://www.live555.com/mediaServer/>.
 - [21] T. Q. C. Ltd. Qoffscreensurface class | qt gui 5.6, 2016. [Online]. [Accessed on 7 May 2016]. Available from: <http://doc.qt.io/qt-5/qoffscreensurface.html>.
 - [22] T. Q. C. Ltd. Qt opengl, 2016. [Online]. [Accessed on 6 May 2016]. Available from: <http://doc.qt.io/qt-5/qtopengl-index.html>.
 - [23] B. Mariano and S. G. Koo. Is cloud gaming the future of the gaming industry? In *Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on*, pages 969–972. IEEE, 2015.

- [24] P. Mell and T. Grance. The nist definition of cloud computing. *NIST*, 2011.
- [25] nVidia. What are the system requirements for geforce now? | shield, 2016. [Online]. [Accessed on 4 May 2016]. Available from: <https://shield.nvidia.co.uk/support/geforce-now/system-requirements/1>.
- [26] Open Networking Foundation. Openflow, 2016. [Online]. [Accessed on 1 May 2016]. Available from: <https://www.opennetworking.org/images/stories/sdn-resources/openflow/12-8-OpenFlow-Diagram.jpg>.
- [27] OpenGL. Faq - opengl.org, 2014. [Online]. [Accessed on 6 May 2016]. Available from: <https://www.opengl.org/wiki/FAQ>.
- [28] D. Piner. Witcher III system minimum and recommended system requirements and upgrade guide, 2016. [Online]. [Accessed 15 April 2016]. Available from: <http://www.tentonhammer.com/guides/witcher-iii-system-minimum-and-recommended-system-requirements-and-upgrade-guide>.
- [29] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.
- [30] D. Roggen. Qtffmpepwrapper repository, 2013. [Online]. [Accessed on 7 May 2016]. Available from: <https://code.google.com/archive/p/qtffmpepwrapper>.
- [31] M. Rouse. What is control plane (cp)?, 2013. [Online]. [Accessed on 18 April 2016]. Available from: <http://searchsdn.techtarget.com/definition/control-plane-CP>.
- [32] M. Rouse. What is data plane (dp)?, 2013. [Online]. [Accessed on 18 April 2016]. Available from: <http://searchsdn.techtarget.com/definition/data-plane-DP>.
- [33] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol, 1998. [Online]. [Accessed 20 April 2016]. Available from: <https://www.ietf.org/rfc/rfc2326.txt>.
- [34] SDxCentral. What are sdn controllers?, 2015. [Online]. [Accessed on 8 May 2016]. Available from: <https://www.sdxcentral.com/resources/sdn/sdn-controllers/>.
- [35] SDxCentral. What are sdn northbound apis?, 2015. [Online]. [Accessed on 1 May 2016]. Available from: <https://www.sdxcentral.com/resources/sdn/north-bound-interfaces-api/>.
- [36] SDxCentral. What are sdn southbound apis? - where they are used., 2015. [Online]. [Accessed on 1 May 2016]. Available from: <https://www.sdxcentral.com/resources/sdn/southbound-interface-api/>.

- [37] SDxCentral. What is open vswitch (ovs)? - definition, 2015. [Online]. [Accessed on 7 May 2016]. Available from:
<https://www.sdxcentral.com/resources/open-source/what-is-open-vswitch/>.
- [38] R. Shea, J. Liu, E. Ngai, and Y. Cui. Cloud gaming: architecture and performance. *Network, IEEE*, 27(4):16–21, 2013.
- [39] A. Soomoro. What is input lag: The breakdown, 2013. [Online]. [Accessed 21 April 2016]. Available from:
<http://www.displaylag.com/what-is-input-lag-the-breakdown/>.
- [40] Wireshark. Wireshark faq, 2015. [Online]. [Accessed on 5 May 2016]. Available from: <https://www.wireshark.org/faq.html>.
- [41] C. Wu and R. Buyya. *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center*. Elsevier Science, 2015.

Appendices

Appendix A

External Material

External libraries and software used:

- OpenGL
- Qt 5.3.1 (QtOpenGL and QtNetwork module)
- NetworkX (Python module)
- OpenDaylight 0.5.0 SDN controller
- Open vSwitch virtual switches
- Mininet virtual network simulator
- Wireshark network packet analyser

The load balancer code was based on code found in:

<https://github.com/nayanseth/sdn-loadbalancing>. The code was improved by allowing for multiple traffic flows to be load balanced.

Appendix B

Ethical Issues Addressed

This project does not have any ethical issues involved since there are no human participants. Also no private or personal data was used in the process.

Appendix C

User Manual

GitHub repository at: <http://github.com/kvcruzat/cloudgaming/>

C.1 Cloud Gaming System

In order for the cloud gaming system to run, an environment with the following software/libraries is needed:

- C++ and g++ compiler
- OpenGL
- Qt 5.3.1 or above

C.1.1 Game Server

The game folder is located in the GitHub repository in the path:
`cloudgaming/Implementation/game/`

To compile and execute the game a *make.sh* script is included in the folder for compilation and can be run with the following commands:

```
$ sh make.sh
$ ./game
```

C.1.2 Client Program

The client program can be run locally on the same machine or on a different machine on the same network. The client folder is located in the current path:
`cloudgaming/Implementation/client/`

To compile and execute the game a *make.sh* script is included in the folder for compilation. To execute the client, IP address or host name and port number needs to be supplied as arguments. The IP address and port number is printed out on the terminal where the game server is executed. Here are example commands

```
$ sh make.sh
$ ./client comp-pc3085 4871
```

C.2 Networking

In order for the SDN networking to run, an environment with the following software/libraries is needed:

- Java 8
- Python 2.7.6 or above

C.2.1 Mininet Virtual Network

To install Mininet the following commands can be used:

```
$ git clone git://github.com/mininet/mininet
$ $ git checkout -b 2.2.1
$ ./util/install.sh -a
```

To run the virtual network with mininet the topology is located in the path: `cloudgaming/Implementation/networking/` To execute Mininet with the *fatcloudtopo.py* topology, OpenDaylight needs to be running in another terminal (instructions in section C.2.3) then use the following command:

```
$ sudo mn --controller=remote,ip=127.0.0.1 --custom fatcloudtopo.py
--topo cloudtopo --link tc
```

C.2.2 Open vSwitch

To install Open vSwitch, the following installation instructions can be used:

<http://dannykim.me/danny/openflow/57620>

The following commands are used to start Open vSwitch:

```
$ sudo ovssdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock
--remote=db:Open_vSwitch,Open_vSwitch,manager_options
--private-key=db:Open_vSwitch,SSL,private_key
--certificate=db:Open_vSwitch,SSL,certificate
--bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert --pidfile --detach
$ sudo ovs-vsctl --no-wait init
$ sudo ovs-vswitchd --pidfile --detach
```

C.2.3 OpenDaylight SDN Controller

The controller can be downloaded from the following URL:

[https://nexus.opendaylight.org/content/repositories/.opendaylight.](https://nexus.opendaylight.org/content/repositories/.opendaylight)

snapshot/org/.opendaylight/integration/distribution-karaf/0.5.0-SNAPSHOT/

To run the controller, Open vSwitch needs to be running (instructions in section C.2.2). The controller can be run with the following commands:

```
$ cd distribution-karaf-0.5.0-SNAPSHOT
$ ./bin/karaf
```

In order for the controller to function correctly for this project, a few features need to be installed. The following command needs to be executed in the controller's command line interface:

```
<opendaylight-user@root> feature:install odl-dlux-all odl-restconf-all
odl-l2switch-all
```

The OpenDaylight web GUI can be accessed using the following link in a browser while the controller is running: <http://localhost:8181/index.html>

C.2.4 Load Balancer

To run the load balancer tests, the SDN controller (section C.2.3) needs to be running in the background and the fat tree topology running with Mininet (section C.2.1). In Mininet CLI ping all hosts (Figure C.1) and connect to hosts where traffic will be sent and received using the example following commands:

```
mininet> pingall
mininet> xterm h1 h2 h4 h5
```

iPerf traffic simulation can be run by setting destination node in server mode and the source in client mode with the commands:

```
h1 (10.0.0.1: Server mode):
$ iperf -s -u
h4 (10.0.0.4: Client mode):
$ iperf -c 10.0.0.1 -u -b 50M -t 60
```

Delay in the connection can then be tested with a simple ping command with 10 packets in the Mininet CLI:

```
mininet> h1 ping -c 10 h4
```

The load balancer can be run using another terminal and the path for the python script *load_balancer.py* is in: `cloudgaming/Implementation/networking`. The load balancer for the connection between h1 (10.0.0.1) and h4 (10.0.0.4) can be executed as follows:

```
$ python load_balancer.py
```

```
Enter Host 1
```

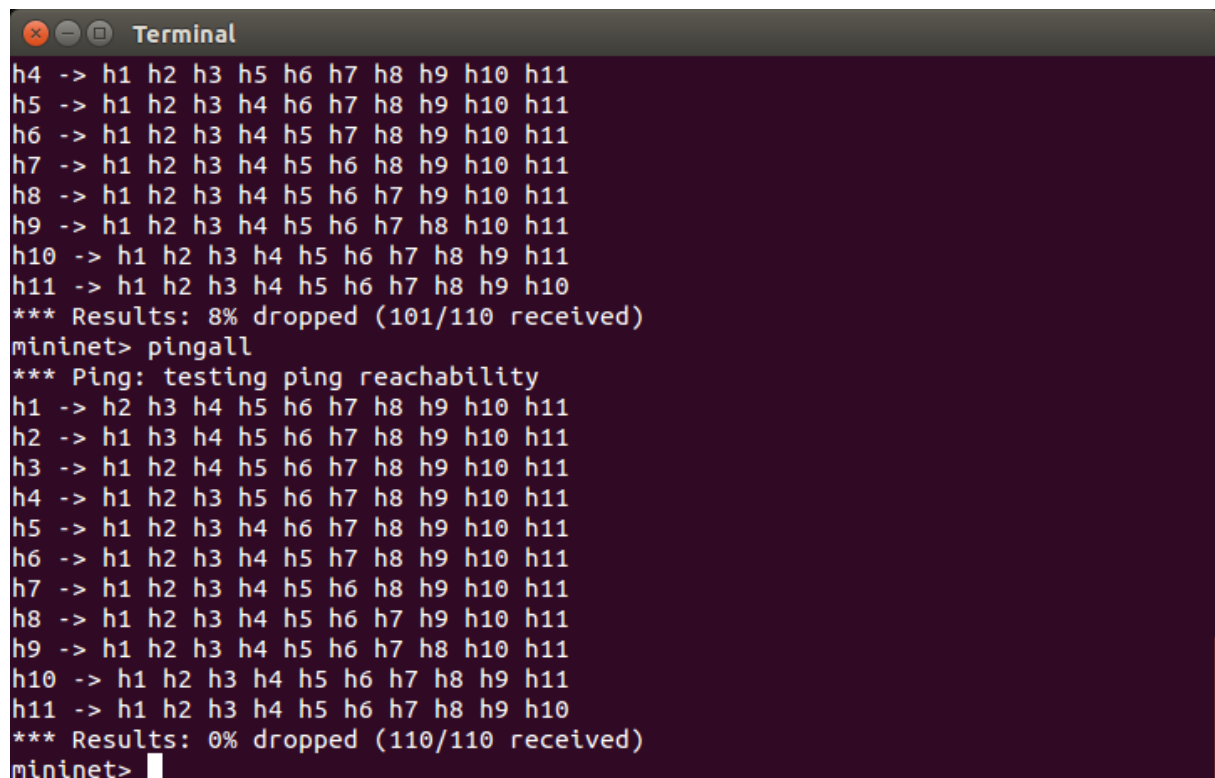
```
1
```

```
Enter Host 2
```

```
4
```

The installed flows can be deleted by running the python script *deleteflows.py*

```
$ python deleteflows.py
```



```
Terminal
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Results: 8% dropped (101/110 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Results: 0% dropped (110/110 received)
mininet> 
```

Figure C.1: Screenshot of all hosts discovering each other through pingall command