ASSESSMENT 2: Adaptive Video Streaming over SDN

COMP3004
Advanced Computing and Networking Infrastructures
2022/2023

Table of Contents

1. Abstract

A large majority of Internet users use services such as Facebook, YouTube, Twitch, Twitter etc. The platforms all stream video content to millions of users' devices each day. To keep customers engaged with these platforms there is a huge demand for high quality, fast media.

This report intends to undergo an experiment wherein network conditions are investigated to make informed deductions as to what factors have the biggest impact on video streaming quality.

2. Introduction

This section of the report introduces the technology being used within this experiment and gives explanations of what each technology is and why it is used. Furthermore, this section will also further discuss the aims of this project.

Aims

This project aims to undergo an investigation into the impact of bandwidth and bitrate on video streaming. Bandwidth is a term used to describe the throughput of a means of transmission. In other words, bandwidth is the maximum amount of data that can be handled at a given time. For example, in this project the network upon which the video is being streamed across would have a bandwidth. On the other hand, bitrate is the number of bits (which are small bits of data) that are transmitted or processed per unit of measurement, which is most often seconds. For example, N bits can be processed per second.

It is important to understand that the higher the bitrate, the better the quality of the video that is being streamed. However, as a higher bitrate entails more bits, the file size of the video will also be larger. Another crucial note is that bitrate is limited by the amount of available bandwidth – data cannot be uploaded or downloaded beyond the capacity of the network. Therefore, larger bandwidths can overall be advantageous, so they do not bottleneck communication by limiting bitrate.

Within this investigation the concept of Adaptive Video Streaming (AVS) is intended to be presented and justified. AVS is a technique wherein the bitrate of the video being displayed is adjusted in relation to the ongoing network conditions. This technique proposes that visual quality should be sacrificed, rather than pausing the video whilst it is buffered. The effectiveness of this technique is further discussed within section: *6. Conclusions.*

Technology

SDN. Software-Defined Networking describes a network architecture approach wherein there is enhanced control over a network. This is achieved by having a central design that is programmable and virtual. The benefits that this provides include increased flexibility and visibility of network devices. This architecture is abstracted into 'data' and 'control' planes, where the purpose of the former is to forward data and the latter's purpose is to describe how data is forwarded. These two components of the architecture are bridged by the SDN Controller, which is the network's Operating System, and the controller will create forwarding tables, which are data that store network destinations within the network and are used to route traffic. Overall, SDN is an architecture that will be used to manage the network being created within this project.

DASH. Dynamic adaptive streaming over HTTP is a streaming tool that is used to decompose a media file into ordered segments that are then served over HTTP to enable high quality media streaming. Segments are typically encoded at different bit rates (from low to high quality) from which the DASH tool can determine which segment can be downloaded and played after the current segment without creating any stalls. Overall, DASH takes segmented media, each encoded to represent a different size and quality, and automatically determines which of these encoded segments is most suitable to play next for seamless playback. DashJS is a media player used alongside DASH to enable videos to be loaded and for the process to be monitored.

The process of integrating these technologies into the project are found within section: *3. Setup, Configuration, and Code.*

Tasks

A brief explanation of the steps taken within the project follows:

Firstly, two virtual machines were created and were setup to run Ubuntu (18.04.6). One virtual machine was built on the Ubuntu Desktop image and the other Ubuntu Server. The DashJS application was built within the Desktop via Apache2 web servers and the Mininet library was used to build a virtual network through which one host was used to run the web server, and the other was used as a client and ran Firefox to access the web server content, which was the DashJS application. The Ryu SDN Controller was installed and setup within the Server machine.

Secondly, a test video was encoded in three different bitrates (qualities) and each encoded video was then segmented so that the video could be dynamically loaded into the DashJS player. However, to showcase Adaptive Video Streaming, the .mpd file of each three of the segmented videos were combined into a single .mpd file. When this specific .mpd file is loaded into the player it should theoretically enable the DashJS application to determine whether the network conditions allow for the higher bitrate video segments to be downloaded and displayed. If not, then the player would display either the middle option or lowest option. The quality of each encoded video is distinguishable.

Thirdly, experiments were conducted on the application to determine whether Adaptive Video Streaming was in effect. This experiment involved limiting the bandwidth of the hosts via Mininet. Options for low, medium, and high bandwidths were tested.

Structure of this report

Section 3 (*Setup, Configuration, and Code*) illustrates the implementation of the above tasks within this project. Screenshots of the configurations and automation of dependency setup are provided. Code is also provided where necessary. Full code and configurations can be found within the Appendix.

Section 4 (*Experiments, Results, and Analysis*) covers how the experiment into bandwidth and Adaptive Video Streaming was undertaken. Results are discussed here as well as justified.

Section 5 (*Discussions and Network Monitoring*) explores how the SDN controller can collect network statistics despite being external. The results of these statistics and their importance for network management are highlighted.

3. Setup, Configuration, and Code

This section explains how the project was setup, configured, and carried out. Full files for the configurations and code can be found within the Appendix.

Creating Virtual Machines

To start the project, two virtual machines were created within WMware Workstation Player. One was named Server and built from the Server build of Ubuntu 18.04. The other VM was built from the Desktop build of the same version of Ubuntu. By default, the VMs were configured as NAT network type. The first step was to change the network type to Bridged so that both VMs would show within the same network.

Installing Dependencies

The most necessary technologies within this project have been highlighted within the Introduction of this report. However, there were a lot of smaller packages that were required. Firstly, to speed up workflow and keep a record of all commands that were being used to build up the project, shell scripting was integrated within the project. A separate shell script was made for both VMs as they had different dependencies and requirements. This report will cover the Desktop VM's setup first.

Using the built in nano text-editor a file called 'init.sh' was created (Appendix 8.1). Within this file, the following packages were installed:

- Git
- Mininet
- Python3
- Apache2
- x264
- GPAC - for the MP4Box tool
- Unzip

Where the **sudo apt-get** command was used, it was accompanied by the option **–y** so that if any prompts were made to overwrite or confirm the download of the package, it was automatically accepted. This was useful, especially as the shell file was also being used outside of downloading the dependencies.

The Mininet package was installed via Git to get the latest, most up-to-date source code, and therefore the Git package also had to be installed. Mininet came packed with its own shell script to execute (which installs the package to the virtual machine). Mininet is necessary for creating the virtual network.

Python3 was installed because it was needed to work in conjunction with Mininet and to create the network topology.

Apache2 was installed because it was the chosen HTML web server used to host the DASH application.

x264 and MP4Box were installed for video segmentation and encoding purposes.

Unzip was installed so that the DashJS.zip file that was downloaded manually, rather than via a package, could be extracted. This zip file was manually placed within a child directory at '/avsProject' (see Figure 1). The original test video was also stored here. The sample video chosen was a one-minute sample from Big Buck Bunny. This file was named bbb1.mp4.

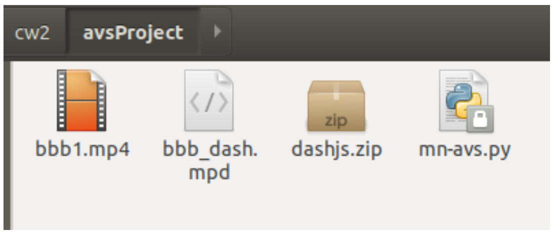


Figure 1: /cw2/avsProject directory

In the Appendix, the init.sh file for the Server VM can be found (Appendix 8.2). Within this VM, the dependencies to install were:

- Git
- Python3
- Ryu

Git was needed so that the Ryu source code could be cloned into the VM. Python was needed so that Ryu's requirements could be installed from the Git directory. Ryu was a Controller that was used to run a REST API alongside the virtual switches.

Both init.sh files were made executable with the command: **chmod +x init.sh**
And they were run with the command: **./init.sh**

Apache2 Server, DASH.js, and Segmented Videos

Now that the dependencies were installed and all required components were ready, it was time to start assembling the project. To do this, a few more console commands were needed and therefore they were added to the Desktop init.sh file to record the process and make it easy to restart when issues were encountered.

The first step was to build up the Apache web app by adding the necessary files to the correct directory. The files that needed to be moved were: *bbb1.mp4* and dashjs.zip. These files were moved using the **cp** command and were copied into the root folder for Apache2: /var/www/html/ .

dash.js was then extracted within this root folder via the **unzip** command. Accessing the web server from the Firefox browser showed that the default Apache2 landing page had been replaced with the DASH.JS player.

Finally, the videos were encoded and segmented. The bbb1.mp4 video was encoded three separate times at different bitrates: 100, 1500, and 5000. These values show a very low bitrate, a medium bitrate and an high bitrate. Each video was encoded at 720p and 30fps. This made the file smaller, however the segmentation process was still lengthy as multiple files were being generated for each of the three encoded files.

When the files were segmented, they were directly extracted to the Apache root folder where DASH.js was set up. These files were also uniquely named meaning that they didn't overwrite each other.

At this stage in the project, the web server could be accessed, and videos loaded into the player and watched at their set bitrates. The video that was encoded to a bitrate of 100 was very low quality, whilst the other two were normal quality (see Figure 2 and 3).

In Figure 2 there is noticeable compression whilst in Figure 3 the video is viewable in a normal quality.

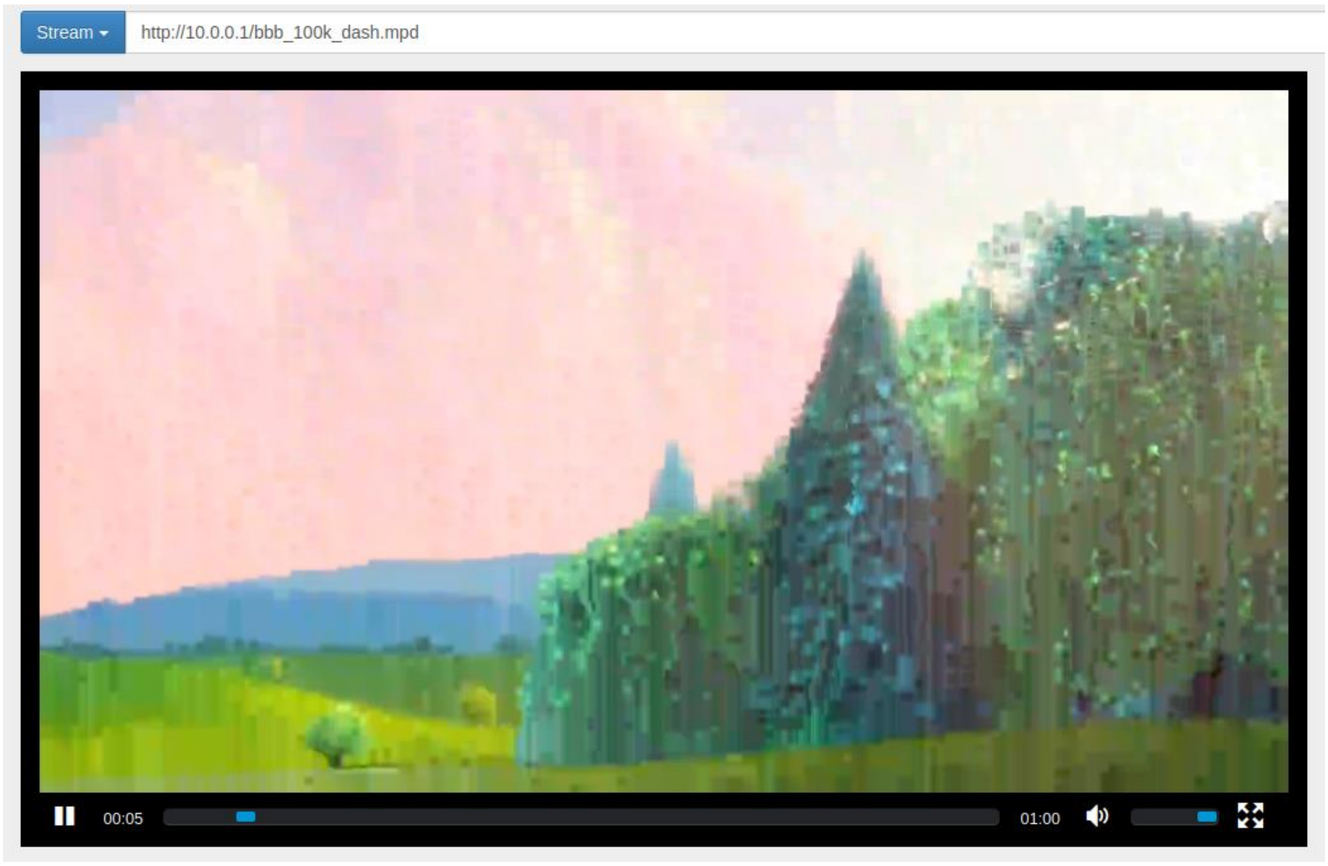


Figure 2: Lowest bitrate video

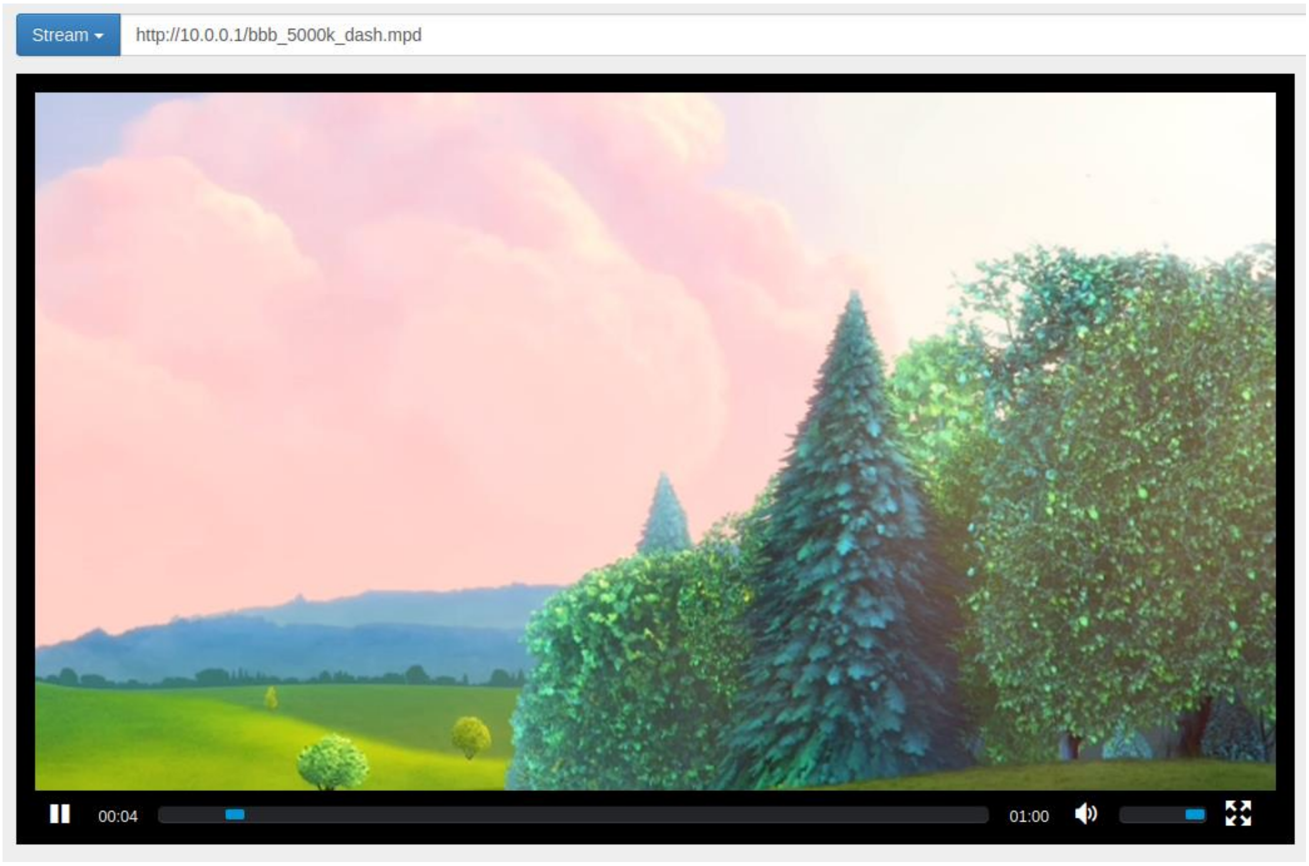


Figure 3: Highest bitrate video

Combined MPD File

A combined MPD file was created by creating a duplicate of one of the existing MPD files in /var/www/html/ and then adding the information necessary from the two other MPD files into the duplicate that was just made. Another duplicate of the finished MPD file was made and stored within the /cw2/avsProject directory (see Figure 1) so that in the init.sh script the project could be easily reset and this file would not need to be remade manually each time.

The combined MPD file is in the Appendix (8.3).

Mininet

Appendix 8.4 shows the Python3 code written to create the virtual network. This code was initially generated via running the MiniEdit.py file and designing the network topology via the GUI. Figure 4 shows the GUI topology that was created.

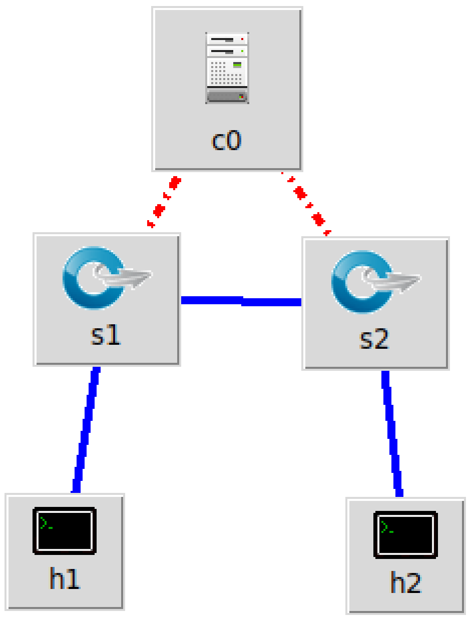


Figure 4: Topology in Mininet GUI

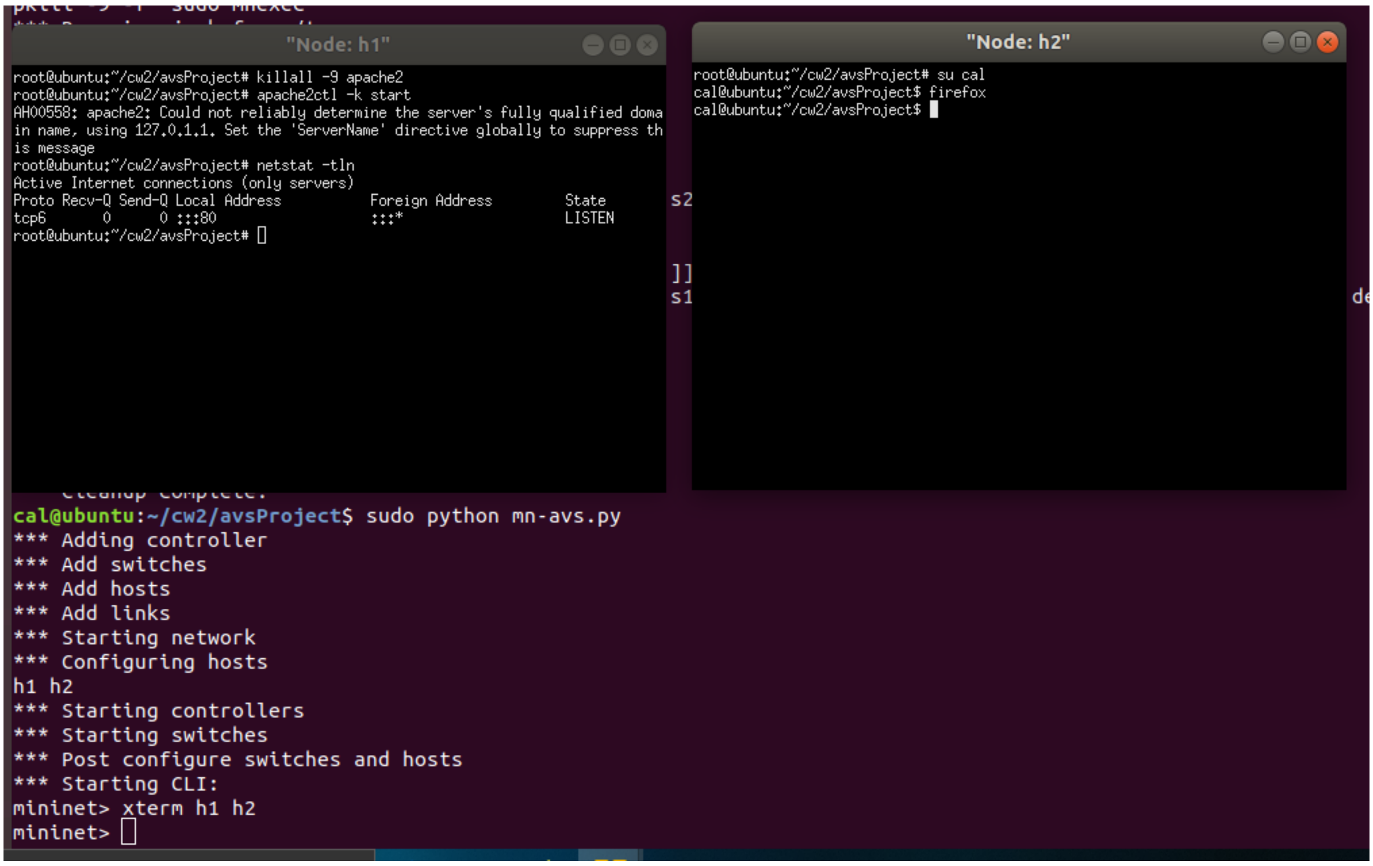
In both the code and the above figure, there are two hosts, two switches and one controller. Each host connects to a singular switch. The switches connect to each other and the controller.

Within this project, h1 is used to host the Apache2 web server and therefore the DASH.js application, whilst h2 is used to access a browser and then access the application via the IP of the Controller.

The network is created via the following command: **sudo python mn-avs.py**
This command runs the topology python code and builds the network. Within this Python application, the bandwidth is controlled. The python file also specifies that the Controller is remote and can be found at a specific IP address.

Once the network is created, the command **xterm h1 h2** is used to open consoles for each host. In the h1 host, the Apache service is run: **apache2ctl -k start**

In the h2 host, Firefox is opened.



```
root@ubuntu:~/cw2/avsProject# killall -9 apache2
root@ubuntu:~/cw2/avsProject# apache2ctl -k start
AH00558: apache2: Could not reliably determine the server's fully qualified doma
in name, using 127.0.1.1. Set the 'ServerName' directive globally to suppress th
is message
root@ubuntu:~/cw2/avsProject# netstat -tln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp6       0      0 :::80                   :::*                    LISTEN
root@ubuntu:~/cw2/avsProject# 
```

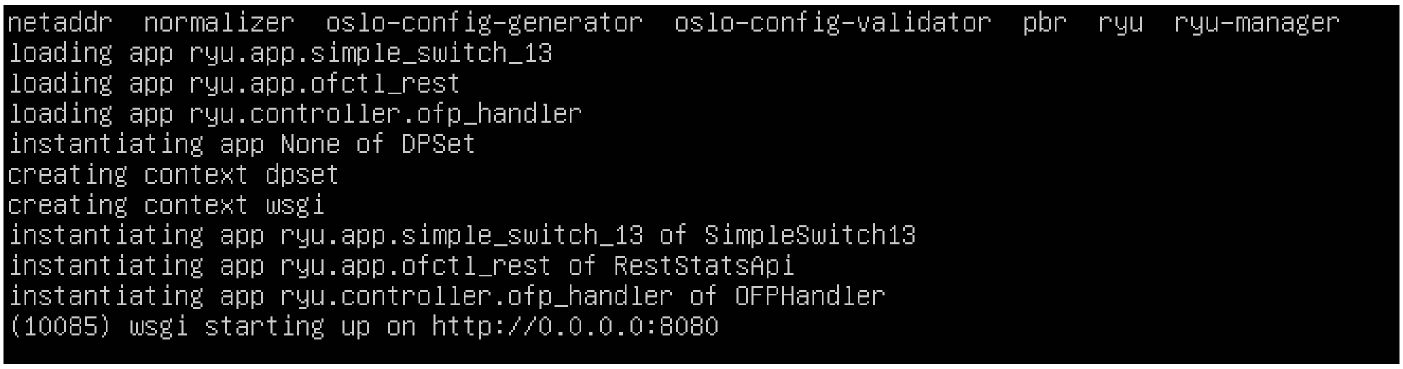
```
root@ubuntu:~/cw2/avsProject# su cal
cal@ubuntu:~/cw2/avsProject$ firefox
cal@ubuntu:~/cw2/avsProject$ 
```

```
cal@ubuntu:~/cw2/avsProject$ sudo python mn-avs.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h1 h2
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> xterm h1 h2
mininet> 
```

Figure 5: Running the Hosts via the virtual network Mininet.

Ryu Controller

The Controller was run via the ryu-manager executable. This executable was used to run the applications for: simple_switch13 and the REST API. These commands were written within the init.sh for the Server (Appendix 8.2). Figure 6 shows the API running.



```
netaddr  normalizer  oslo-config-generator  oslo-config-validator  pbr  ryu  ryu-manager
loading app ryu.app.simple_switch_13
loading app ryu.app.ofctl_rest
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.controller.ofp_handler of OFPHandler
(10085) wsgi starting up on http://0.0.0.0:8080
```

Figure 6: Rest API on Server VM

4. Experiments, Results, and Analysis

Bitrates of 100, 2500 and 5000 were chosen for this experiment.

A low bandwidth of 1 Mbps was chosen however in this investigation there was no changes to the quality of the player. The quality remained good and there was zero packet loss.

A medium bandwidth of 5 Mbps was chosen to also investigate if there would be any impact on the application. However, once again there was no deterioration in quality.

This leads me to believe that the bandwidth limits were placed on the network were either not working fully or these chosen bandwidths were simply bad targets to measure against.

5. Discussions and Network Monitoring/Management

6. Conclusions

Overall, a SDN testbed was built alongside a Ryu controller. An application was built within Apache2 server. Mininet was used to build a virtual network – which ran two hosts. One host ran the Apache2 server whilst another was used as a client to access the DASH.js application. Meanwhile, the Ryu controller was running in a separate Virtual Machine and the Mininet code established that this virtual machine's IP address was the controller.

Improvements could be made by encoding more versions of the mp4 video so that a greater range of qualities are available.

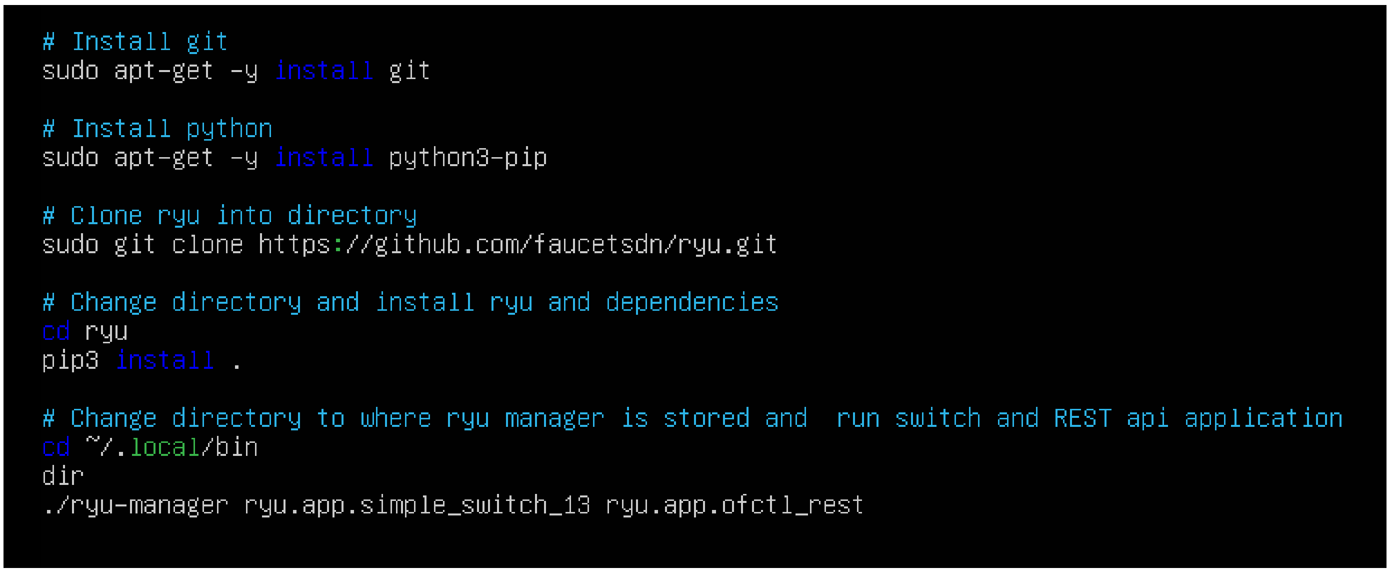
7. References

8. Appendix

8.1. init.sh for Desktop

```
#!/bin/bash

# Install git
sudo apt install git

# Install mininet
#sudo git clone https://github.com/mininet/mininet
#sudo mininet/util/install.sh -a

# Install Python
sudo apt-get -y install python3-pip

# Install Apache Web Server
sudo apt-get -y install apache2

# Install x264 and MP4Box
sudo apt-get -y install x264
sudo apt-get -y install gpac

# Install unzip
sudo apt-get -y install unzip

# Change directory
mkdir avsProject
cd avsProject

# Copy mp4 video into Apache2 root folder
sudo cp bbb1.mp4 /var/www/html/

# Copy dashjs.zip into Apache2 root folder
sudo cp dashjs.zip /var/www/html/

# Change directory to Apache2 root folder
cd /var/www/html/

# Extract zip file
sudo unzip -o dashjs.zip

# 1. Encode video in H.264/ACV
# 2. Add video to mp4 container
# 3. Segmentation

sudo x264 --output bbb_1500k.264 --fps 30 --bitrate 1500 --video-filter resize:width=1280,height=720
./bbb1.mp4
sudo MP4Box -add bbb_1500k.264 -fps 30 bbb_1500k.mp4
sudo MP4Box -dash 4000 -frag 4000 -rap -segment-name segment_1500k_ bbb_1500k.mp4

sudo x264 --output bbb_5000k.264 --fps 30 --bitrate 5000 --video-filter resize:width=1280,height=720
./bbb1.mp4
sudo MP4Box -add bbb_5000k.264 -fps 30 bbb_5000k.mp4
sudo MP4Box -dash 4000 -frag 4000 -rap -segment-name segment_5000k_ bbb_5000k.mp4

sudo x264 --output bbb_100k.264 --fps 30 --bitrate 100 --video-filter resize:width=1280,height=720
./bbb1.mp4
sudo MP4Box -add bbb_100k.264 -fps 30 bbb_100k.mp4
sudo MP4Box -dash 4000 -frag 4000 -rap -segment-name segment_100k_ bbb_100k.mp4


# Copy mpd file to Apache2 root folder
cd ~/cw2/avsProject
sudo cp bbb_dash.mpd /var/www/html/
```

8.2. init.sh for Server



```
# Install git
sudo apt-get -y install git

# Install python
sudo apt-get -y install python3-pip

# Clone ryu into directory
sudo git clone https://github.com/faucetsdn/ryu.git

# Change directory and install ryu and dependencies
cd ryu
pip3 install .

# Change directory to where ryu manager is stored and  run switch and REST api application
cd ~/.local/bin
dir
./ryu-manager ryu.app.simple_switch_13 ryu.app.ofctl_rest
```

8.3. bbb_dash.mp4

```
<Period duration="PT0H1M0.067S">
  <AdaptationSet segmentAlignment="true" maxWidth="1280" maxHeight="720" maxFrameRate="30"
par="16:9" lang="und">
   <Representation id="1" mimeType="video/mp4" codecs="avc1.64001f" width="1280" height="720"
frameRate="30" sar="1:1" startWithSAP="1" bandwidth="104797">
    <SegmentList timescale="30000" duration="97941">
     <Initialization sourceURL="segment_100k_init.mp4"/>
     <SegmentURL media="segment_100k_1.m4s"/>
     <SegmentURL media="segment_100k_2.m4s"/>
     <SegmentURL media="segment_100k_3.m4s"/>
     <SegmentURL media="segment_100k_4.m4s"/>
     <SegmentURL media="segment_100k_5.m4s"/>
     <SegmentURL media="segment_100k_6.m4s"/>
     <SegmentURL media="segment_100k_7.m4s"/>
     <SegmentURL media="segment_100k_8.m4s"/>
     <SegmentURL media="segment_100k_9.m4s"/>
     <SegmentURL media="segment_100k_10.m4s"/>
     <SegmentURL media="segment_100k_11.m4s"/>
     <SegmentURL media="segment_100k_12.m4s"/>
     <SegmentURL media="segment_100k_13.m4s"/>
     <SegmentURL media="segment_100k_14.m4s"/>
     <SegmentURL media="segment_100k_15.m4s"/>
     <SegmentURL media="segment_100k_16.m4s"/>
     <SegmentURL media="segment_100k_17.m4s"/>
     <SegmentURL media="segment_100k_18.m4s"/>
     <SegmentURL media="segment_100k_19.m4s"/>
    </SegmentList>
   </Representation>

   <Representation id="2" mimeType="video/mp4" codecs="avc1.64001f" width="1280" height="720"
frameRate="30" sar="1:1" startWithSAP="1" bandwidth="1438382">
    <SegmentList timescale="30000" duration="97941">
     <Initialization sourceURL="segment_1500k_init.mp4"/>
     <SegmentURL media="segment_1500k_1.m4s"/>
     <SegmentURL media="segment_1500k_2.m4s"/>
     <SegmentURL media="segment_1500k_3.m4s"/>
     <SegmentURL media="segment_1500k_4.m4s"/>
     <SegmentURL media="segment_1500k_5.m4s"/>
     <SegmentURL media="segment_1500k_6.m4s"/>
     <SegmentURL media="segment_1500k_7.m4s"/>
     <SegmentURL media="segment_1500k_8.m4s"/>
     <SegmentURL media="segment_1500k_9.m4s"/>
     <SegmentURL media="segment_1500k_10.m4s"/>
     <SegmentURL media="segment_1500k_11.m4s"/>
     <SegmentURL media="segment_1500k_12.m4s"/>
     <SegmentURL media="segment_1500k_13.m4s"/>
     <SegmentURL media="segment_1500k_14.m4s"/>
     <SegmentURL media="segment_1500k_15.m4s"/>
     <SegmentURL media="segment_1500k_16.m4s"/>
     <SegmentURL media="segment_1500k_17.m4s"/>
     <SegmentURL media="segment_1500k_18.m4s"/>
     <SegmentURL media="segment_1500k_19.m4s"/>
    </SegmentList>
   </Representation>


   <Representation id="3" mimeType="video/mp4" codecs="avc1.64001f" width="1280" height="720"
frameRate="30" sar="1:1" startWithSAP="1" bandwidth="4721495">
    <SegmentList timescale="30000" duration="97941">
     <Initialization sourceURL="segment_5000k_init.mp4"/>
     <SegmentURL media="segment_5000k_1.m4s"/>
     <SegmentURL media="segment_5000k_2.m4s"/>
     <SegmentURL media="segment_5000k_3.m4s"/>
     <SegmentURL media="segment_5000k_4.m4s"/>
     <SegmentURL media="segment_5000k_5.m4s"/>
     <SegmentURL media="segment_5000k_6.m4s"/>
     <SegmentURL media="segment_5000k_7.m4s"/>
     <SegmentURL media="segment_5000k_8.m4s"/>
     <SegmentURL media="segment_5000k_9.m4s"/>
     <SegmentURL media="segment_5000k_10.m4s"/>
     <SegmentURL media="segment_5000k_11.m4s"/>
     <SegmentURL media="segment_5000k_12.m4s"/>
     <SegmentURL media="segment_5000k_13.m4s"/>
     <SegmentURL media="segment_5000k_14.m4s"/>
     <SegmentURL media="segment_5000k_15.m4s"/>
     <SegmentURL media="segment_5000k_16.m4s"/>
     <SegmentURL media="segment_5000k_17.m4s"/>
     <SegmentURL media="segment_5000k_18.m4s"/>
     <SegmentURL media="segment_5000k_19.m4s"/>
    </SegmentList>
   </Representation>

  </AdaptationSet>
  </Period>
</MPD>
```

8.4. mn-avs.py

```
#!/usr/bin/env python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
from mininet.log import setLogLevel

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c0=net.addController(name='c0', controller=RemoteController, ip='192.168.1.111', port=6633)

    info( '*** Add switches\n')
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)

    info( '*** Add links\n')
    net.addLink(h1, s1, bw=1)
    net.addLink(h2, s2, bw=1)
    net.addLink(s2, s1)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s2').start([c0])
    net.get('s1').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
```

```
if __name__ == '__main__':
	setLogLevel('info')
	myNetwork()
```