Technical University of Denmark

Paulius Preibys, s181324
Radheshyam Singh, s186394
Yichen Zhang, s181390
Supervisor: Henrik Wessing

Network Emulation and Verification: Final Report

Synthesis Project for Telecommunication, 34249

Report, June 2019, Version 3

DTU Fotonik
Department of Photonics Engineering

Abstract

Course 34249 ’Synthesis Project for Telecommunication’ participants are assigned a large system design project - an audio/video transmission system, with an addition of the transmitting node being a drone.

This paper is a system description report on ’Network Emulation and Verification’ part of the assigned project. The report concentrates on describing the sub-project’s functionality in detail, considerations on implementation methods, implementation process and stating the interfacing towards other parts of the audio/video transmission system. Purposes and results of different kinds of tests are also included.

Table of Contents

List of Figures

1 Introduction

1.1 Project Overview

Course 34249 'Synthesis Project for Telecommunication' participants are split into groups of 2-6 students and each group is assigned a sub-project, responsible for designing, implementing and testing a specific segment of the desired system - an audio/video transmission system, where a transmitter is said to be on a drone. There is a total of seven sub-projects, each responsible for a system block: signal modulation, network emulation and verification, video encoding and decoding, framing and error correction, security, application based monitoring and control, drone API and control. Each system block interfaces with other blocks and all sub-projects integrate together to perform the complete audio/video transmission.

1.2 Network Emulation and Verification sub-project

'Network Emulation and Verification' sub-project's goal is to provide an alternative (to a direct link) path for video transmission between the transmitter and receiver of the system and monitor its performance. The alternative path provided needs to be resilient, of configurable bit rate and be able to introduce relevant testing scenarios. Performance verification parameters of the emulated network are required to be collected and displayed in an efficient way, introduce as little interference/overhead to the 'user' traffic as possible, and above all, be as accurate as possible. Key parameters to be monitored are packet latency, bit rate and loss over the designed network. A high level diagram of 'Network Emulation and Verification' block is given in figure 1.1.

All the above mentioned requirements pose certain questions and considerations when choosing implementation path and designing the structure of 'Network Emulation and Verification' system block. Additionally, interfacing with neighbouring system blocks must also be considered. This report discusses the chosen technologies for sub-project implementation, interfaces with other groups and includes a detailed testing description.

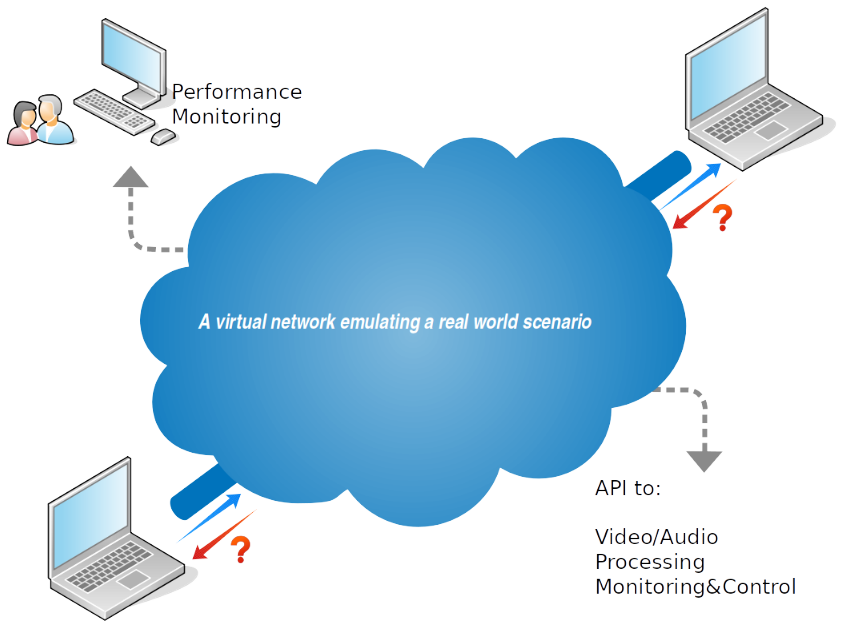


Figure 1.1: A high level diagram of 'Network Emulation and Verification' block

2 Network Emulation

Virtual network architecture, implementation methods and future improvements are discussed in this chapter.

2.1 Overview

As mentioned in chapter 1, some of the key requirements for the virtual network are high resiliency, the ability to pre-configure desired bit rate for links in the network, and flexibility to introduce relevant testing scenarios over the emulated network (e.g. network link failure and re-route effect on video stream quality, higher/lower network link bit rate effect on video stream quality). Based on these requirements, a network architecture was designed, a plan for implementation drawn up and some relevant testing conducted. The following sub-chapters introduce the design process in detail.

2.2 Architecture

In order to make the alternative path and testing scenario more realistic, an architecture resembling that of a SP (Service Provider) network was chosen.

2.2.1 L3 Topology

A layer 3 topology of the network is given in figure 2.1.

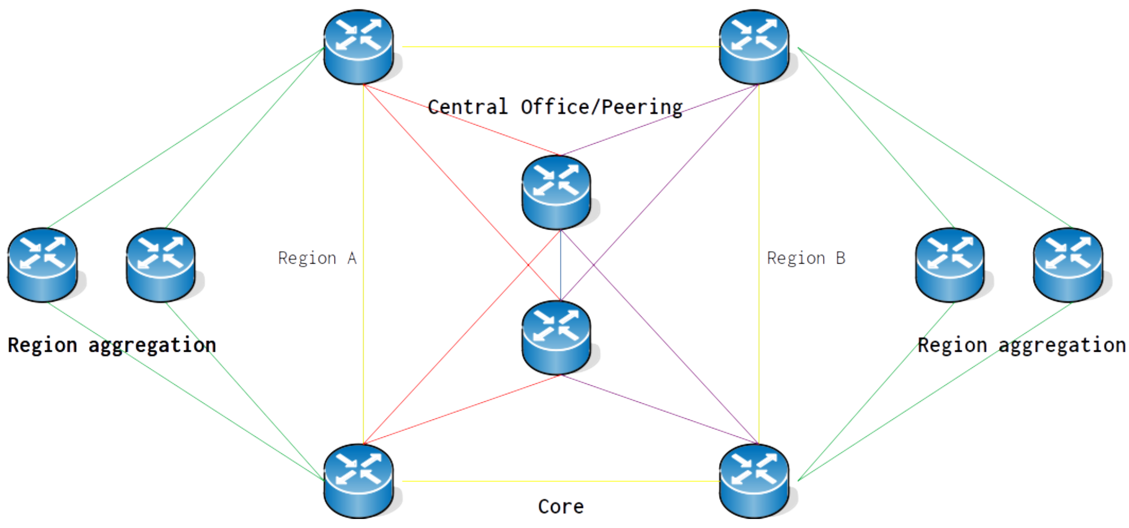


Figure 2.1: A L3 topology of the emulated network

The network topology in figure 2.1 contains three layers:

- a peering layer, that would be used to connect to the Internet, or other internal ISP services. In our project's scenario, these nodes could be used to insert 'noise' traffic into the network;
- a core layer, which is used to aggregate and route regional traffic. Regional traffic to the Internet or other ISP services is routed towards peering layer, while internal network traffic between regions is routed inside the network core layer. Each core layer device has links to each peering layer node and a single link to neighbouring network region. Two devices used in each region of the core ensure device redundancy, while L3 links, housed on disjoint physical connections, between regional core nodes, to neighbouring regions and to the peering layer nodes ensure link redundancy.
- a regional aggregation layer, which contains regional nodes aggregating client traffic. Each node has a L3 link to both region core nodes - L3 links are implemented in disjoint physical links.

In this (figure 2.1) L3 topology, transmitting and receiving nodes would be connected to different regions, while the peering nodes could be used to generate some 'noise' traffic. Technologies chosen for routing, connectivity establishment are implementation dependant, thus are discussed in section 2.3.

2.2.2 L1 Topology

The L1 topology designed is depicted in figure 3.6. The main segments of the diagram are:

- core network ring - all core network nodes are interconnected by 4 optical fibre links, providing high link redundancy and two disjoint paths to any other node in the core;
- region network rings - region aggregation nodes are also interconnected by 4 fibre rings, providing link redundancy and two disjoint paths towards the core network. Region rings are connected to the core network at region core nodes;
- Central Office/Peering core network node could be seen as another region in the topology, however, for simplicity and due to the fact that it will only be used for inserting 'noise' traffic, a single ROADM is used.

Each two neighbouring nodes of the network are interconnected by four fibre links, where two fibres are used for live traffic (solid lines in 3.6) and two are used for protection (dotted lines in 3.6). If all the links and nodes in the network are operational, traffic across such network is forwarded over the shortest path, e.g. bidirectional traffic between two region core nodes and central office node is forwarded directly between them and not over the longer segment of the ring. In case of a live traffic link failure, protection fibres between adjacent nodes are utilised. However, if the protection links in a ring segment fail at the same time as live links, traffic is redirected over the longer segment of the ring. In case of a node failure, traffic is redirected around the ring. Such ring network architecture is known as Bidirectional Line-Switched Ring (BLSR) [5]. It should be pointed out that BLSR architecture could also be implemented by using two fibres for neighbouring node, however, in such case, the only protection path is over the longer segment of the ring.

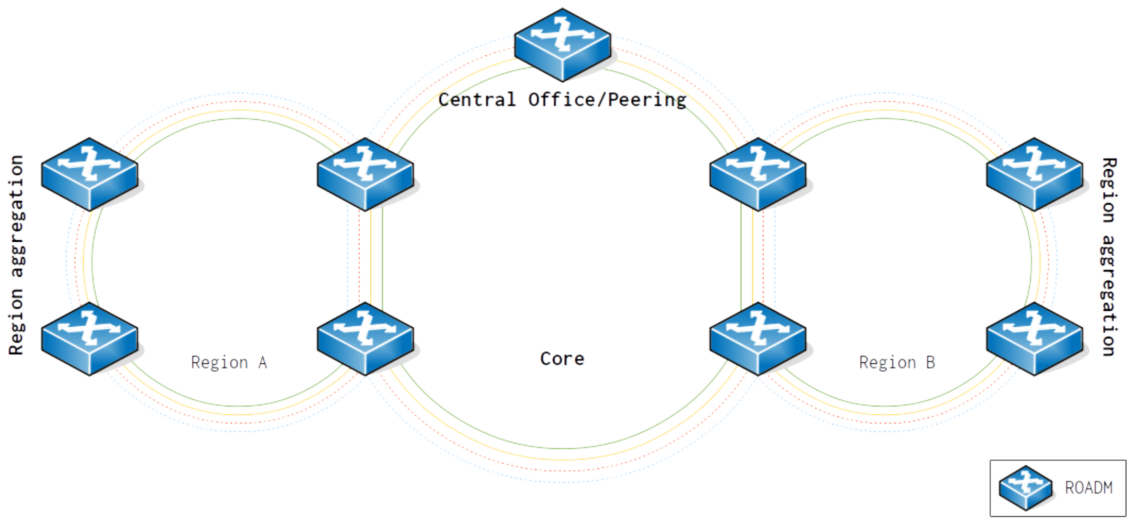


Figure 2.2: A L1 topology of the emulated network

2.3 Implementation

2.3.1 Virtualisation

Since we have decided to emulate a network in a virtual environment, multiple decisions had to be made regarding means of virtualisation and hardware used. First of all, a laptop computer was chosen to be used as hardware platform. Multiple tests have shown that the available resources (24GB RAM, 4 CPU (8 multi-threaded)) are sufficient to transport video of up to 45 Mbit/s bit rate over the virtual network without visual disturbance at the receiver end. Since all the physical hardware is shared for network emulation and monitoring, one must be vary of the fact that monitoring can impact transmission performance and vice versa.

Oracle VM Virtualbox [6] was chosen to be used as hypervisor. The choice was based on the fact that Virtualbox is open source, free, tried and tested and easy to use. Another option considered (not based on virtualisation) was to use Linux Containers for some parts of the system (e.g. SDN controller) to save some physical resources of the laptop, however Virtualbox VMs were chosen due to ease of use.

2.3.2 Network Emulation

Network emulator

Mininet network emulator [7], capable of emulating switches, hosts and links, was chosen as the key component of the virtual network. It utilises Linux namespaces for running the 'virtualised' components of the network, thus is similar to Linux Containers or other container orchestration systems and relies on standard Linux/Unix networking applications. Mentioned characteristics of mininet allow it to emulate realistic system behaviour.

Network nodes

Two types of software switches are used within the designed virtual network environment:

- Open vSwitch [8] is used to emulate the packet switched nodes. OVS is an open source software switch, used by default in mininet setup. OpenFlow is used to insert forwarding rules to device tables.
- LINC software switch [9], together with LINC-OE extension is used to emulate ROADMs in the network. Additional python classes are used in mininet environment to startup and integrate the LINC-OE switches to mininet network. Same as OVS, LINC-OE relies on OpenFlow to deliver forwarding instructions to the device tables. The forwarding instructions are based on 'physical' signal parameters - LINC-OE switches add/strip additional headers to/from packets entering or leaving 'optical' domain. The additional headers contain signal information (frequency channel) on which forwarding is based.

Network controller

OpenFlow [10] protocol is used to control forwarding behaviour of network nodes. Therefore, an SDN controller is needed to generate the rules and orchestrate the insertion of required forwarding rules. Open source ONOS SDN [11] controller was chosen for that purpose for a couple of reasons:

- previous knowledge of the controller - open source SDN controllers are complex software programs and generally have a rather steep learning curve, thus choosing one where some level of knowledge has already been acquired is essential;
- available applications - ONF ONOS SDN controller focuses on providing an SDN controller to service providers, therefore a number of projects (namely CORD [12]) are being developed under the supervision of ONF, resulting in a number of applications designed for use in service provider environments. Since our goal is to emulate a service provider network, this proves to be a huge benefit;
- level of documentation - often open source projects are lacking in documentation - while some of the references provided by ONOS community are outdated or no longer true for latest releases of the controller, the available documentation still is relatively detailed and useful. Adding to that, the community seems to be active in helping new users by discussions in Google Groups.

A range of other options in choosing an opensource SDN controller exist, including OpenDaylight, Floodlight, ryu, pox/nox and multiple other projects, however, due to reasons stated above, ONOS was chosen.

2.3.3 Implementation testing

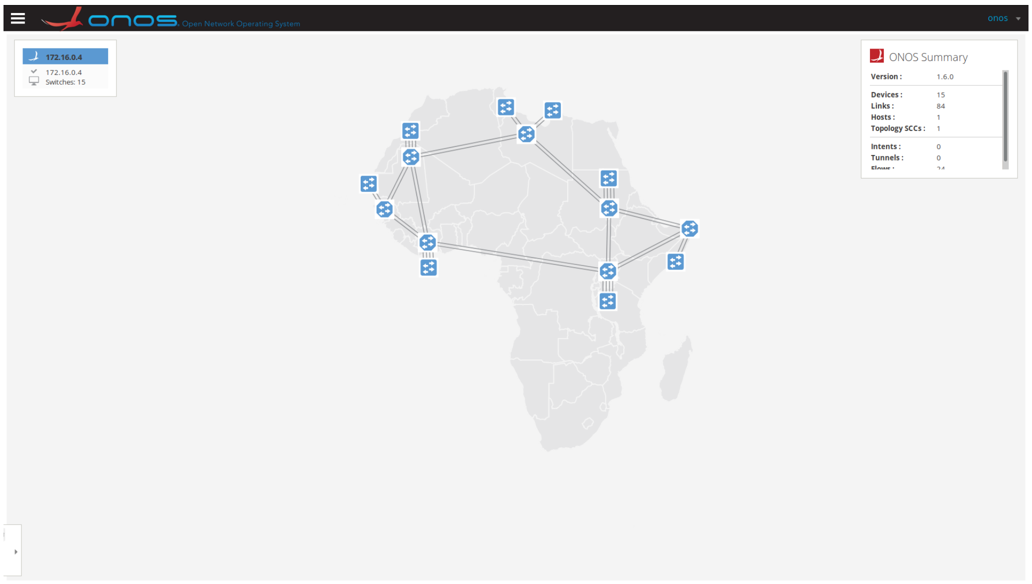
In order to test our chosen implementation, the first step was to emulate the network based on architecture discussed in section 2.2. For that purpose, two virtual machines were created - an Ubuntu Server 14.04 VM (Virtual Machine) for network emulation and Ubuntu Server 16.04 VM for the controller. Mininet, LINC-OE and ONOS controller were installed on respective virtual machines. Both controller and network emulation VMs were further configured by adding some linux environment variables, such as address of the controller or controller authentication credentials (not essential but eases workflow). Both VMs were given Internet access for easier installations and an additional network adapter for mutual interconnection and connection to the host. Final versions of the VMs are configured as:

- network VM - 4vCPU, 8 GB of RAM, 3 network adapters;
- controller VM - 1 vCPU, 8 GB of RAM, 1 network adapter;

Following extensive system testing, it could be observed that the limiting factor for the transmission rates and monitoring accuracy is CPU processing capacity in the network VM (and the physical hardware).

Once the VMs were prepared, a python script was written implementing the L1 and L3 topology. Script creates a network consisting of 7 ROADMs (5 in the core ring and one in each regional ring) and 10 OVS switches (6 in the core network and two in each region networks (additional OVS switches in region networks are needed to utilise mininet's TCLink class, allowing for configurable link parameters)), adds two hosts for testing/noise traffic purposes, adds links in between network nodes, and bridges regional switch ports to specified VM interfaces. Only two links (instead of four) are added in between adjacent ROADMs due to them being bidirectional. Interfaces between OVS and LINC-OE nodes are implemented as Linux tap interfaces and their interconnection is provided by opticalUtils classes and methods from ONOS testing tools [13]. 4 links are added between each core ROADM and core OVS switch, as only one connection can be configured per tap interface. Once all network elements are created, network topology is pushed to the ONOS controller.

Once the topology is pushed to the controller and OpenFlow connection between ONOS and network nodes is established, the network topology can be inspected in ONOS GUI (figure 2.3).



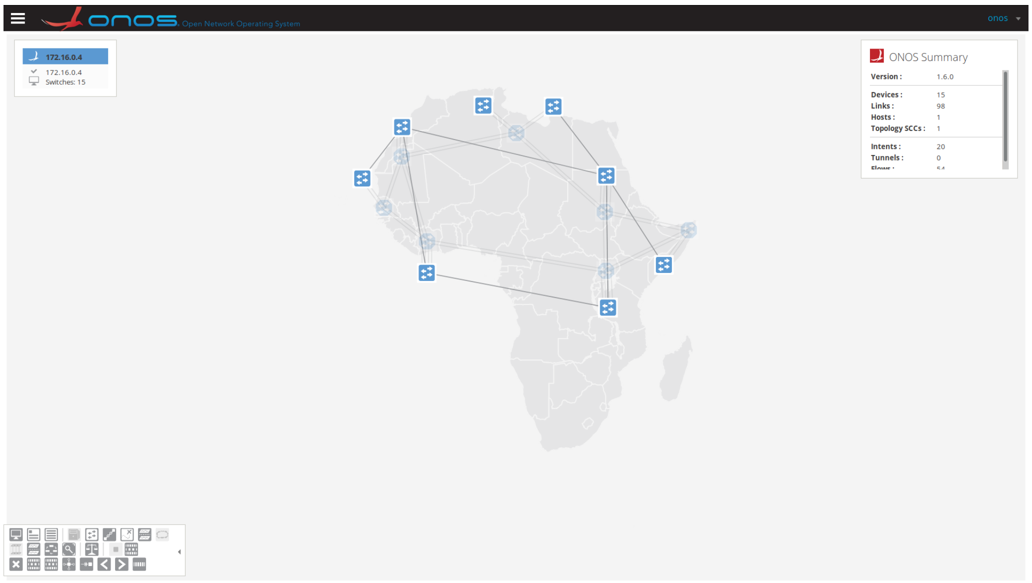
172.16.0.4
172.16.0.4
Switches: 15

Version : 1.6.0
Devices : 15
Links : 84
Hosts : 1
Topology SCCs : 1
Intents : 0
Tunnels : 0

Figure 2.3: Emulated topology in ONOS web UI

Next step towards testing forwarding behaviour is to set up connectivity between nodes. ONOS application 'newoptical' is used for that purpose. The application allows user to set up optical connectivity between two OVS switches provided there is an optical path.

If a path or paths exist, shortest one is selected and flow rules are inserted in relevant ROADM nodes. The rules are based on mapping a specific frequency channel over required input and output ports. Frequency channel is emulated by adding an additional packet header in LINC-OE ROADMs. If primary path fails, connectivity is reestablished on the second shortest path. An example of the same topology with some of the packet switched connections established is given in figure 2.4. If host-to-host connectivity is desired, one should also add host-to-host intents between hosts for L2 forwarding (or enable a L2 forwarding application).



Version : 1.6.0
Devices : 15
Links : 98
Hosts : 1
Topology SCCs : 1
Intents : 20
Tunnels : 0

Figure 2.4: Emulated topology in ONOS web UI where some of the packet switched connections are established (highlighted)

Forwarding & Capacity Testing

System's forwarding behaviour can be tested using mininet virtual hosts. Such a test is simple, quick and provides an insight on general system capabilities. More elaborate testing is required to make sure the emulated network is able to perform and carry the required video data. To be more specific, a test including real hosts connected to the emulated network is required.

For that purpose, a setup depicted in figure 2.5 was constructed. Two PCs running iperf network testing software were connected to the laptop chosen for implementation. In order to forward real traffic over the emulated virtual network, the physical implementation laptop interfaces must be mapped to the guest OS 'Network VM'. To facilitate this mapping, Virtual Box interface bridging is used. Once the interfaces were mapped to 'Network VM', they need to be further mapped to ingress & egress mininet switches (corresponding to Region A & B aggregation devices in figure 2.1). To successfully achieve VM network interface to OpenvSwitch port mapping, mininet scripts were adjusted accordingly. Furthermore, provided LINC-OE integration to mininet software was modified to successfully facilitate the mapping, as such 'link to the real world' inclusion in mininet causes errors in unmodified LINC-OE integration code (in LINC configuration json file construction functions). Once the interfaces are correctly mapped, topology and forwarding behaviour can be set up according to previous description. Finally, iperf testing can be conducted.

Iperf was used to test network performance under UDP traffic load to mimic video transmission to be used in the project. The results conclude that under current Network VM setup (8 GB RAM, 4 vCPUs), the emulated network is capable of achieving bit rates of approx. 130 Mbit/s. It should be noted, that additional CPU cores would increase the achievable bit rates.

Further testing included the use of video streaming software instead of iperf. In this case, it can be concluded that the selected implementation is capable of forwarding 45 Mbit/s bit rate video without visual disturbance at the receiving end.

Testing scenarios are described in a detailed manner in chapter 4.

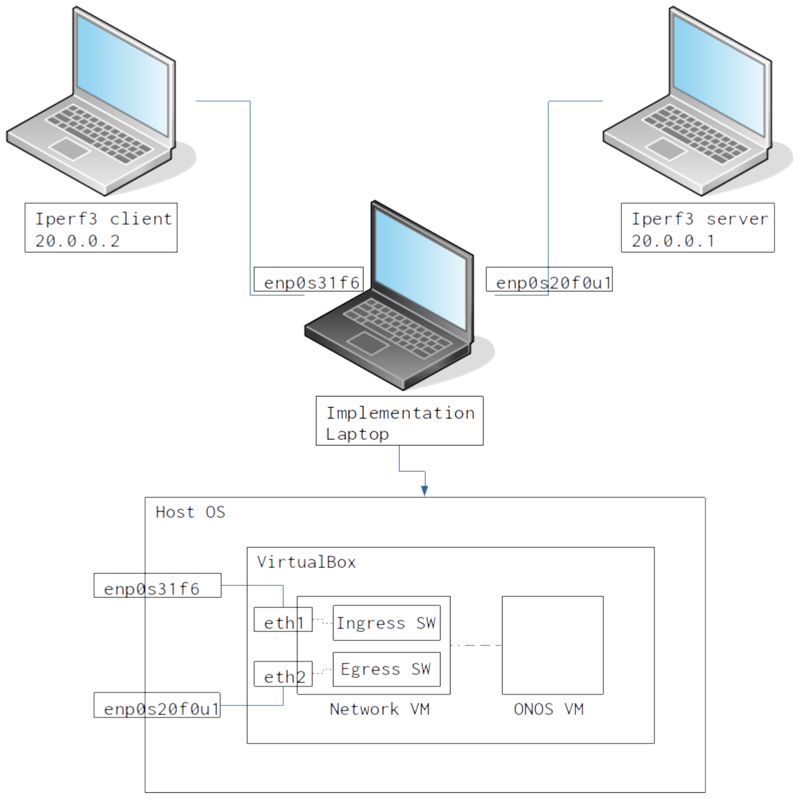


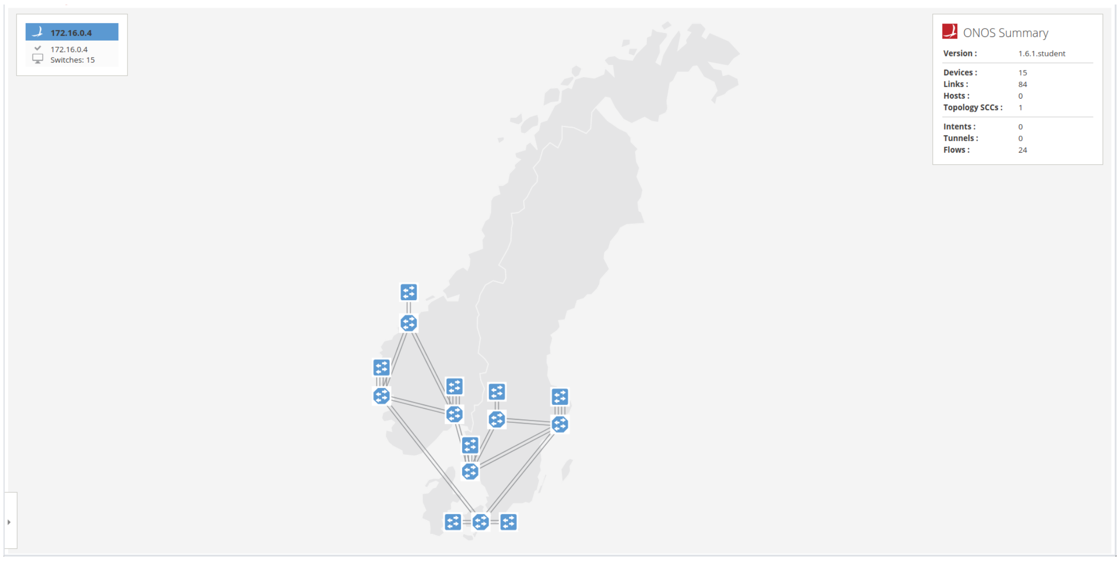
Figure 2.5: Iperf testing setup with real hosts

2.3.4 Alternative Map Inclusion

ONOS controller allows for background map inclusion in network topology representation for convenience purposes. The built in maps available for selection include large regions or specific countries (e.g. Africa or South Korea). Initial implementation was represented over an Africa map (figures 2.3 and 2.4). Since it is an open source project, users are welcome to include their own maps for desired areas. Based on feedback received, it was decided to include an alternative map for a more familiar region.

To do that, one must obtain map data (e.g. from online sources). The obtained data is likely to include multiple regions or the whole world, thus the required data should be filtered out (e.g. data for Denmark, Norway and Sweden). Map data in ONOS is included in *topojson* format, therefore the gathered data should also be in this format. Furthermore, the filtered map data might not be centred and include complex shapes, placing network nodes in obscure locations and/or increasing page load times. To avoid this behaviour, map data should be tweaked and simplified (online/offline tools available).

Once the map data is prepared, an application allowing for new map selection should be developed or current GUI application code tweaked. While both options present certain difficulties, tweaking current application code to include additional maps is a more practical option. To achieve that, ONOS source code should be downloaded, required Java classes found and modified accordingly. Finally, the project should be built and tested. Additionally, the tweaked ONOS source code can be packaged in a more user friendly archive format, including the launching scripts. An example of an updated representation over Denmark, Norway and Sweden is given in figure 2.6.


172.16.0.4
172.16.0.4
Switches: 15

ONOS Summary

Version :	1.6.1.student
Devices :	15
Links :	84
Hosts :	0
Topology SCCs :	1
Intents :	0
Tunnels :	0
Flows :	24

Figure 2.6: Emulated topology in ONOS web UI, alternative map

3 Monitoring

3.1 Overview

In the monitoring subproject, the mission is to monitor the packets passing through the emulated network and analyse different network parameters. Monitoring technology can be active monitoring or passive monitoring, and they are suitable for different scenarios.

Active monitoring mode injects the test packets into the network, following them and detecting network performance. Active monitoring enables us to monitor acquired parameters and get control on the generation of packets for measurement scenarios, thus get real-time feedback from the measurement. However, active monitoring would increase the network load and only gives feedback on a small amount of data, making it suitable for dealing with a specific metric. Passive monitoring mode monitors over the packets being transmitted over selected segments or interfaces and analyses packets based on the pre-defined capture filters. Passive monitoring adds little load on the network hardware and is ideal for network state and load monitoring.

Based on the comparison above, passive mode monitoring was chosen to be implemented in our system block. The video group sends UDP packets containing payloads of video data to our emulated network through the input interface, the network routes UDP packets to the destination and outputs them through the output interface. The monitoring part captures UDP packets, compares packets at the two interfaces and checks the network latency/packet loss for UDP packets.

The following sections describe technologies chosen for implementation of the monitoring part for the sub-project, a detailed explanation on implementation design of network monitoring is also provided.

3.2 Implementation Methods

In this sub-project, we use Python scripts to do the packets capture and analysis. The used operating system is Linux which is compatible with Mininet and can run python scripts in the terminal control window under Mininet environment.

3.2.1 Scapy

Python is a high level programming language used for many purposes and it supports many standard tools, including some useful libraries for packet capture. Using scapy, a user can create, decode, send and analyse the packets. Scapy performs a basic function to match the packets, it sends all the packets and receives the answers. Scapy can be a alternative for the tools like tshark, hping, arping etc. These tools have some inbuilt functions for some specific task and user can not do feasible changes according to their needs but by using scapy users can do reasonable changes and design a application according to their needs[1]. Figure 3.1 shows some basic abilities of scapy, such as:

- Scanning
- Attacking
- Packet Forging
- Sniffing

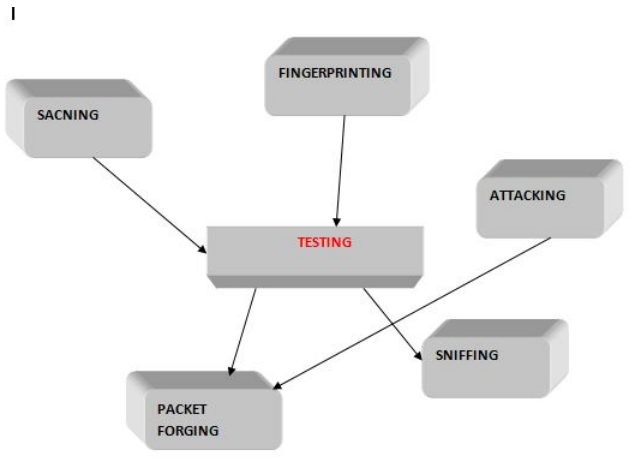


Figure 3.1: Scapy abilities[1]

While doing some UDP tests using mininet we found out that when bit rate is high (in the order of Mbit/s) than CPU's capacity is consumed more than 85% by the python monitoring code and that was happening because of scapy packet manipulation tool. Scapy has a bug and because of it scapy loses packets when data rate is high[14]. Here we are doing a live monitoring to calculate the latency, number of received packets and number of lost packets. Now we can say that while using scapy loss of data packet is more and it uses maximum CPU capacity, so using scapy for monitoring continuously running data packets, is not an optimum solution. To avoid this excess use of CPU and to make the monitoring code more efficient instead of scapy, raw socket is used.

3.2.2 Raw Socket

Using a raw socket, a software application can send and receive data without disturbing the operating system of a computer. In other words, we can say that a raw socket act as a network socket to send and receive the data packets directly to the user's application without involvement of operating system. In figure 3.2, we can see that data packets are directly sent to the user application by-passing the normal TCP/IP processing [15].

Raw socket has some features, such as:

- Since data is not explicitly handle by the operating system, because of this system's overhead is reduced.

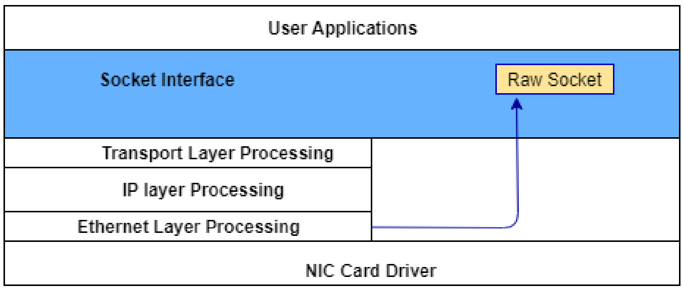


Figure 3.2: Raw socket graphical representation [2]

- Raw socket act as a pipeline between the application and external source and because of this there is no interference in raw connection caused by operating system or any other applications[15].
- Using raw socket user can specify the TCP, IP header and send it to application[16].

3.2.3 Influx DB

Apart from the video group, we also need to interface with the control group, providing our analysis data to them. InfluxDB was chosen as the database tool to share information to the control group. InfluxDB is a Time Series Database, which makes it ideal for storing data of packets with time stamps. A Time Series Database is built specifically for handling a large range scans of many records that are time-stamped. A TSDB is optimized for measuring change over time [17].

The control group created a database on the internet using InfluxDB, and we inserted the monitoring results into the database so that they can access to the network information. Official standard InfluxDB-Python API client libraries are used to let our Python script get access to InfluxDB.

3.3 Implementation

3.3.1 Packets Capture

Data is transmitted through the network according to certain formats of various network protocols. When the data to be sent is packaged, addresses (MAC address, IP address, etc.) of the sender and the receiver are packaged together and sent out. A packet has different layers, the header of each layer contains some information regarding the transmission. Basic format of an IP packet is shown below:

Ether IP UDP/TCP RTP

Basically, the packets capture technology looks at these layers and sets filters to filter needed packets. Socket library has a function "socket.socket()" which creates socket objects and listens to incoming packets. A standard expression of socket.socket function has several parameters to configure [18].

- First parameter is the address family. Several types of address family are available depending on the communication way, like TCP/UDP communication between servers or communication locally.
- Type is used to declare socket constants. Packets can be passed to and from the interface without any changes or the Ethernet header can be removed.
- Protocol number is default 0. It uses IEEE 802.3 protocol number in network bytes order, and different protocols own different numbers.
- File number is default none. If a specific file number is defined, then the remaining parameters in the function will be auto-detected.

Raw data will be captured by socket and then other functions will take and analyse useful UDP packets. The socket.socket() function is written as:

sniff = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

- socket.AF_PACKET parameter captures and manages the traffic. Other optional choices could be AF_INET used for TCP and UDP communication.
- socket.SOCK_RAW receives packets in original format, all packet data will be reserved without any removal, i.e.the Ethernet header being removed.
- socket.ntohs(3) enables to parse all kinds of packets, like ICMP, UDP, etc..
- Fileno would be the default none as it has not been defined.

Captured raw data by socket should be unpacked and further filtered. The first step is to unpack Ethernet packets and select IP packets. In the Ethernet frame structure, the first 14 bytes of the header include an Ethernet type filed as shown in figure 3.3. A filter will check if the Ethernet type is IP by asking if the type number equals to 8 [19].

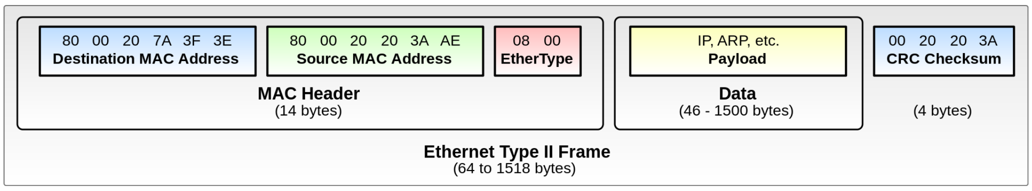


Figure 3.3: Ethernet Frame Format Structure. [3]

After getting IP packets, we should unpack IP packets and check if a packet is a UDP packet. A field in IPv4 packets called protocol stores a number which identifies the protocol type 3.4. A function checks if this protocol number is 17 corresponding to UDP [20].

In a word, socket captures raw data on the interface and we unpack raw data, setting two filters to pick IP packets and UDP packets respectively. On both input and output interfaces, the same mechanism for getting UDP packets is implemented.

However, the script only captures all UDP packets in the network which can not be applied on a network has multiple services. To capture required packets for a specific service, filters for the UDP/TCP destination port and UDP/TCP source IP address should be added when unpacking packets.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.4: IPv4 Header Format. [4]

3.3.2 Packets Analysis

To do the packet analysis, meaning latency calculation, data size calculation and packet loss detection, we have to capture and analyse packets at both ingress interface and egress interface simultaneously. Multiprocessing in Python is used to execute multiple processes at the same time using different processors [21]. Multiprocessing provides a couple of ways to enable shared data between processes, on the other hand, it improves efficiency compared to using one processor for multiple threads. We use multiprocessing to allow capture on the input interface and output interface simultaneously.

Two processes of capturing functions created by socket are running, one at the ingress interface and another one at egress. Upon UDP packets capture, we extract the packet timestamp assigned by the Linux OS at both interfaces. To identify the same packet at two interfaces, a unique ID that is unique for each packet should be picked.

Hash value of the UDP payload has been chosen as the ID to identify packets. Hashing is a safe way to match the packet because hash value of the payload is unique [22]. In rare occasions, hash values could be conflicting resulting in misidentifying packets in analysis. Such a situation would cause erroneous packet latency calculation. Additional steps could be taken to prevent such errors (such as additional dictionary look-ups), however conflicts in our case are unlikely to happen because the hash values of packets are cleared periodically which will be discussed later. Another alternative is UDP checksum. The checksum field in the UDP packet is calculated partly from the carried data [23]. For UDP packets which carry different data (like in our case, video data varies in each UDP packet), the checksum for each packet is mostly unique. However, there is no guarantee that the checksum is always unique. For those packets have the same data accidentally, coincident values can cause errors.

The initial way to do the analysis was invoking another function to store hash values and ingress or egress timestamps in arrays. The latency then can be calculated by performing a subtraction from arrival packet time at egress to arrival packet time at ingress. Yet using arrays to do the packets match and latency calculation could face some problems.

Problem with Arrays

Arrays are ordered lists of elements and elements stored in array can be accessed randomly using the index number. To monitor the latency of packets, arrival time of packets at two interfaces and hash values of packets are needed.

Latency = Arrival time of a particular packet at output - Arrival time of the same packet at input

So for calculating the latency the piece of code should take the same packet and associate arrival time at input and output. While using arrays, the identification for a packet, array matches only the index position of the element in the input array and in the output array. The problem is that for the same packet, the index position of a packet at input and output could be different, because the sequence of packets at output may be random. Even if the hash value of a packet is found in the output array, because of the wrong order, calculation of the latency might be wrong.

Consider the following example. Suppose there are 4 packets at input in In_array and in Out_array. From the diagram 3.2, In_array= [Pac1, Pac2, Pac3, Pac4] and Out_array=[Pac1, Pac2, Pac4, Pac3]. So while calculating the latency it will match Pac3 to the Pac4 and Pac4 to the pac3 and calculation of latency will be wrong.

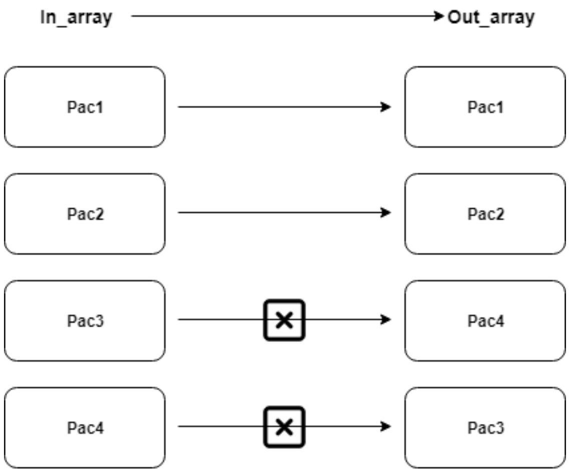


Figure 3.5: Matching a Packet in Array.

The problem shown in figure 3.6 can be avoided by mapping hash values and respective capture times. In this case, for calculating the correct latency for a data packet, finding the exact data packet at output and matching with the input data packet is essential. So in place of array, a dictionary is used. Even dictionary has various types of keys, but finding an element in a dictionary takes less time compared to lists or arrays. In Python, dictionaries are implemented as hash tables. Array or list takes more time to search for an element because they iterate through entire array element or list, whereas dictionary looks for a KEY and every KEY is unique in a dictionary. So dictionary takes constant time for look-ups and array takes linear time [24].

In short, using a dictionary users don't have to care about the sequence of the elements, lookups are faster than array, can handle a bigger data set and find a relationship between pairs of objects.

Dict={'Hash value of the payload' : Arrival Time}

Hash value of the packet payload is used as a key as it is unique. So while calculating the latency, the program searches for hash values of the payload, irrespective of the location of the packets in the ingress and egress dictionaries and grab the arrival time associated to that hash value. Considering the following figure 3.6, where we can clearly see that dictionary is not matching the data packet according to the index position but it uses a KEY for matching it.

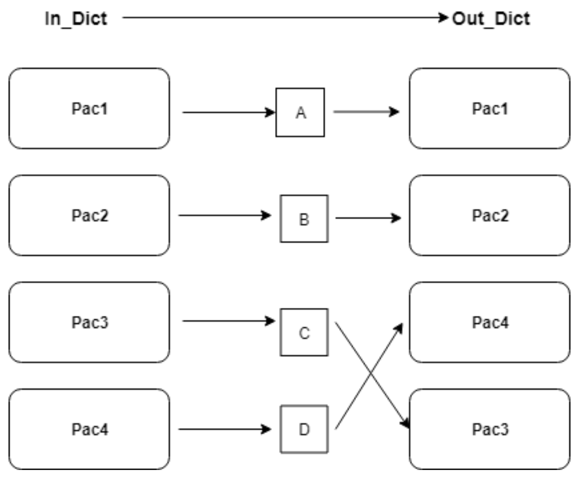


Figure 3.6: Matching a Packet in Dictionary.

Calculations Implementation

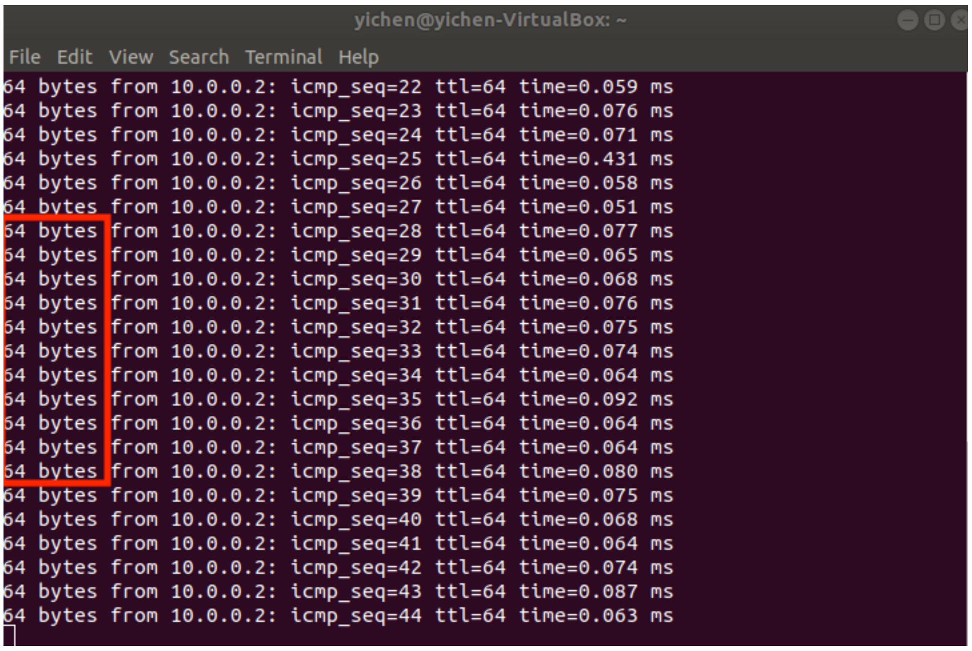
The capture of packets is running periodically. Following the capture time period, analysis and reporting function is called to output results. The finishing of the capture invokes a detection in the dictionary. A search in the dictionary will find the timestamps of each matched UDP packet at egress and ingress interface. Then the latency will be calculated from a subtraction of two timestamps. The values of latency for each packet are stored into an array and the calculation of average latency could be done simply by taking values from this array. It is super easy to get the number of received packets and lost packets now as we capturing packets at two interfaces. The number of received packets equals to the number of captured packets at egress interface in a capture period. The number of sent packets equals to the number of captured packets at ingress interface in a capture period. Estimated packet loss analysis is shown below:

Number of lost packets = Number of packets at ingress - Number of packets at egress

Packet loss rate = 100% * Number of lost packets / Number of packets at ingress

In fact, this is not really a good way to calculate the estimated packet loss. The problem is that the start and end capturing time at two interfaces could be slightly different. The small capturing time deviation between two interfaces would cause errors in captured packets numbers. Especially for a high video bit rate, a small deviation can probably cause unstable capturing results. A more accurate solution relies on active monitoring. Some sample packets can be injected into the network, and packet loss rate is acquired from the statistics of these sample packets.

Another parameter to be measured is the data size. Carried video data size can be obtained from the payload of the packet. Because the data is encapsulated in different layer headers, the way we get the transmitted data size is to get the length of the payload in bytes plus the length of the UDP/IP/Ethernet header. To test the accuracy of the data size calculation, experiments of ICMP transmission have been implemented many times. Tests tested for the payload data size and the transmitted data size. We created a single two hosts network on Mininet and generated packets flow by ping from one host to another host. Two hosts were at two Ethernet ports on one switch, and host one communicated with host two with ICMP packets. The total length of transmitted ICMP packet is 64 bytes as shown in figure 3.7. The calculated data size is 64 bytes per packet corresponding to the theoretical one.


yichen@yichen-VirtualBox: ~
File Edit View Search Terminal Help

```
64 bytes from 10.0.0.2: icmp_seq=22 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=23 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=24 ttl=64 time=0.071 ms
64 bytes from 10.0.0.2: icmp_seq=25 ttl=64 time=0.431 ms
64 bytes from 10.0.0.2: icmp_seq=26 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=27 ttl=64 time=0.051 ms
64 bytes from 10.0.0.2: icmp_seq=28 ttl=64 time=0.077 ms
64 bytes from 10.0.0.2: icmp_seq=29 ttl=64 time=0.065 ms
64 bytes from 10.0.0.2: icmp_seq=30 ttl=64 time=0.068 ms
64 bytes from 10.0.0.2: icmp_seq=31 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=32 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=33 ttl=64 time=0.074 ms
64 bytes from 10.0.0.2: icmp_seq=34 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=35 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=36 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=38 ttl=64 time=0.080 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=0.068 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=0.074 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=0.063 ms
```

Figure 3.7: Transmitted ICMP Packets.

Bit rate is also a parameter to indicate the network performance, a proper bit rate should be maintained to ensure the transmission not exceed the network capacity. The virtual network we built is aimed to deliver the video data. Data transmission bit rate can be seen as transmitted data size per second. A standard formula to calculate bit rate would be:

Bit rate = Data size (bits) / Transmission time (seconds)

Data carried by IP packets is measured in bytes, and the transmission time is the time period between the ingress interface and the egress interface. Simply transform the formula above we can get a new formula to calculate the bit rate for a single packet:

Bit rate (Mb/s) = 8 * Transmitted data size / (Latency*1024*1024)

Since packets are analysed per capture period, the transmission bit rates at two interfaces are calculated from:

Bit rate (Mb/s) = 8 * Total transmitted data size at the interface / (Capture period*1024*1024)

One thing worth mentioning is that all the packet capture and analysis happen periodically when the capture period begins.

- The packet capture is periodic and lasts a specific set time value (the value can be adjusted depending on bit rates and system capabilities). As soon as packet capture is over, the analysis and reporting function is called, which, as its first task, copies and empties data stores used by the capturing processes. Once the mentioned data stores are emptied and copied, the analysis and reporting function notifies the capturing processes to restart.
- Packets captured within one capture cycle are analysed. Average packet latency, bit rate, number of captured packets and packet loss rate are calculated and reported to the terminal and the influxDB per each capture cycle. Total data transferred value is additive and reports the total number of bytes carried during script run time.
- In case of long capture periods at high bit rates, the reporting function might take a long time to execute. To avoid the situation where multiple reporting threads are running at the same time, one can set a specific waiting time for the capturing process to restart after reporting function's notification.

3.3.3 Database Interface

Once we have our desired output results, we can upload them to the database.

The first thing to do is to install influxdb-python libraries, which would let the Python script support connection with InfluxDB. The next step is to create a new instance of the InfluxDBClient, with domain name/user name/password of the server that we want to access. Then the client is configured to use the database provided by the control group. We are sending these information to the database: number of captured packets per period, average latency per period, number of lost packets and packet loss rate per period, transmission bit rate at egress and ingress interface per period, sum of transmitted data size by all packets.

3.4 Achieved Outcomes

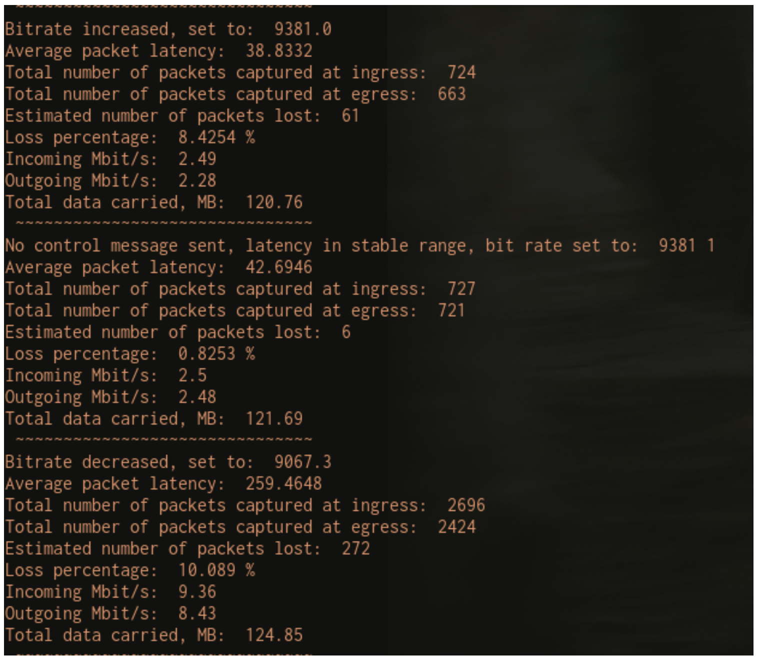
The monitoring part is used to analyse performance of the emulated network. Considering useful parameters for the network performance, we have achieved measurements on some parameters in order to help optimise the data transmission. Achieved outcomes are:

- Number of received packets and lost packets per period.
- Packet loss rate per period.
- Average latency for received packets per period.
- Bit rate at ingress and egress interfaces per period.
- The sum of data size included in all captured packets so we can analyse if we have lost a lot of data compared with the initial inputting video group.

- Able to detect the status of the network, and send notifications back to the video group, informing a more suitable bit rate is required.
- Store all results in the database letting the control group have access to the network performance.

The first four parameters are used to judge if the overall network is capable to transmit videos for the course project. The last three parameters are more like analysing the performance from the view of real world network transmission. These measurements are aimed to monitor the network comprehensively and are helpful to adjust the packets transmission.

An example from tests in chapter 4 is shown in figure 3.8.



```
Bitrate increased, set to:  9381.0
Average packet latency:  38.8332
Total number of packets captured at ingress:  724
Total number of packets captured at egress:  663
Estimated number of packets lost:  61
Loss percentage:  8.4254 %
Incoming Mbit/s:  2.49
Outgoing Mbit/s:  2.28
Total data carried, MB:  120.76
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
No control message sent, latency in stable range, bit rate set to:  9381 1
Average packet latency:  42.6946
Total number of packets captured at ingress:  727
Total number of packets captured at egress:  721
Estimated number of packets lost:  6
Loss percentage:  0.8253 %
Incoming Mbit/s:  2.5
Outgoing Mbit/s:  2.48
Total data carried, MB:  121.69
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Bitrate decreased, set to:  9067.3
Average packet latency:  259.4648
Total number of packets captured at ingress:  2696
Total number of packets captured at egress:  2424
Estimated number of packets lost:  272
Loss percentage:  10.089 %
Incoming Mbit/s:  9.36
Outgoing Mbit/s:  8.43
Total data carried, MB:  124.85
```

Figure 3.8: Achieved Outcomes of the Monitoring.

4 Tests

4.1 Streaming Tests

Several kinds of streaming videos tests within the group have been tried for many times. Based on different variables, streaming tests were performed to examine different network and monitoring capabilities. All tests were taken using the emulated network and the developed monitoring script. Two laptops running VLC acted as the sender (server) and the receiver (client) respectively. VLC is an application which supports multiple ways of streaming videos, like a format of RTP packets, UDP packets, HTTP packets, etc.. The sender streamed a streaming video using UDP as the IP transport protocol to a predefined port on the receiver laptop through the emulated network built on the implementation laptop. All laptops were connected by Ethernet cables, figure 4.1 illustrates the setup for streaming tests.

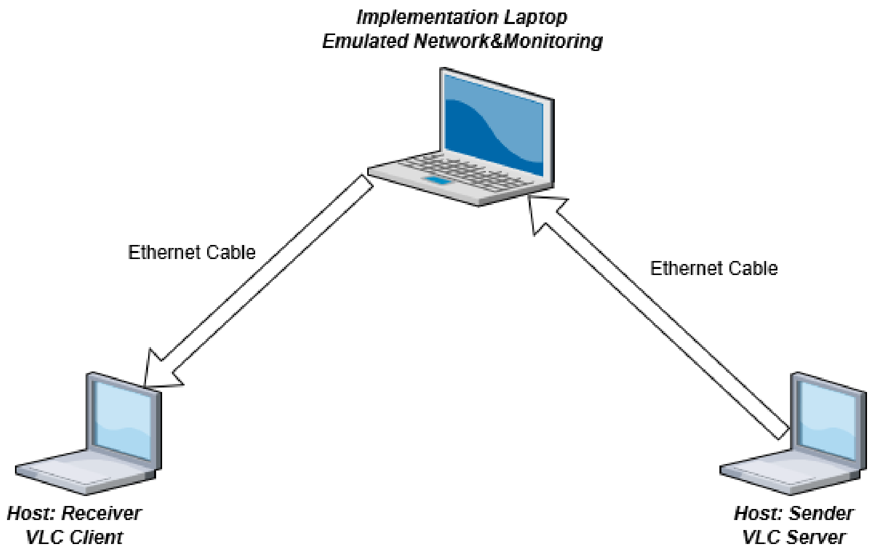


Figure 4.1: Setup for Streaming Tests.

Variables to be controlled for streaming tests include: video bit rate, link bit rate, link latency, link packet loss percentage.

4.1.1 Bit-rate Capacity Streaming Tests

Bit rate control streaming tests were aimed to test the highest capability the emulated network could have. No link capacity was assigned to links in the emulated network in order to get the highest affordable bit rate. The sender streamed a series of increasing bit rate videos, the implementation laptop provided the emulated network and did the monitoring. A series of videos with the encoding bit rate from 10 Mb/s to 60 Mb/s were used for the streaming tests. Performance indicators of the test are the smoothness of the video on the receiver laptop and the accuracy of the monitoring script regarding packets analysis.

- The highest encoding bit rate of the streaming video was 45 Mb/s without any delay on the receiver laptop. Meanwhile, the monitoring script performed less stable and less accurate compared to a 10 Mb/s scenario.
- When the video encoding bit rate reached 50 Mb/s, small delays happened sometimes and the video was less watchable on the receiver laptop.
- When the video encoding bit rate was more than 55 Mb/s, delays were happening frequently, the video on the receiver laptop had poor quality.
- Streaming tests were taken under two kinds of emulated networks. One was a small scale network with only a few nodes (two or three). Another was a larger scale network with 15 nodes. The results under these two networks were similar - the satisfied highest video encoding bit rate was 45 Mb/s.

The monitoring script could always get a relatively accurate transmission bit rate and average latency, while with the video encoding bite rate becoming higher, other parameters could become unstable. Recalling that the way we calculate the packet loss is depending on number of captured packets at the ingress interface and egress interface respectively within the capture period. With the video bit rate getting higher, transmitted packets per second also increase so that errors from the time deviation is increasingly magnified. As a result, tests showed at 10 Mb/s, the script worked perfectly with only few minor errors for usually 1 - 2 deviation on the number of captured packets. However, big errors in measured parameters would happen when the video bit rate reached 30 Mb/s, i.e. up to negative 100 lost packets may happen.

4.1.2 Latency Control Streaming Tests

Purposes of latency control tests are to test the tolerable latency range for the network and accuracy of the latency calculation. Streamed video encoding bit rate was 10 Mb/s and the link capacity was 15 Mb/s, some delay was added on two links between two streaming hosts manually.

100 ms delay for each link

The video was streamed without any interruptions. The monitored average latency was around 200 ms which can be seen accurate, because after passing two delayed links packets were supposed to get 200 ms delay in total. The number of estimated lost packets became higher, one possible reason is that increased latency caused some packets to not arrive at the egress interface before the capturing period ended.

1000 ms delay for each link

The stream quality dropped, video stuttered several times and had visible delays. Regarding latency monitoring, since the capture period was set to 1.5 seconds and the latency was longer than 2 seconds, no packets could be compared during one single capture period, thus the results from the monitoring were not right as we were not comparing the same group of UDP packets.

When the assigned latency was 100 ms, the streamed video was at a good quality. The calculated latency corresponded to the added latency, thus concluding the accuracy of the monitoring script. The tests also proved that when the link latency increases the streamed video's quality drops.

4.1.3 Packet Drop Streaming Tests

Packet drop rate was the only variable in the tests, video encoding bit rate was fixed 10 Mb/s and the link capacity had no limitation. Purpose of tests are evaluating the accuracy of packet loss monitoring. 50% packet drop was added on two links respectively, so after one link 50% packets should be dropped and after these two links only 25% packets would be passed. The links which dropped packets are between two streaming hosts, desired packet loss rate should be 75% at the receiver host.

The video could not be played because it stopped all the time. The monitored estimated packet loss rate was around 75%, proving the rough accuracy of packet loss monitoring function of the monitoring script.

4.1.4 Bit-rate Control Streaming Tests

In these tests, two controllable links between two streaming hosts in the emulated network were assigned a limitation of transmission bit rate. The video encoding bit rate was 10 Mb/s while link limitations were set to different levels. Bit rate control streaming tests are used to evaluate the quality of streamed videos under different link bit rate capacity and the effect of link capacity on the latency and packet loss. As the network became "overloaded", the video on the receiver laptop and the monitoring results would indicate different network status.

15 Mb/s link capacity limitation

A smooth video was playing on the receiver laptop, and the monitoring script had stable and accurate estimated results.

10 Mb/s link capacity limitation

The video on the receiver laptop had lags more than 5 seconds and even stopped playing sometimes. The monitoring results showed that the period average latency went up to 0.5 - 1 second and around 20% packets have been lost.

5 Mb/s link capacity limitation

The streamed video on the receiver laptop could not be watched at all because the video almost looked like displaying a figure. The packet loss percentage was around 50% which corresponds to half of the link capacity.

From above testing scenarios, it is obvious that when the link capacity is higher than the transmitted video bit rate, the network can transmit the video perfectly. On the other hand, when the link capacity equals to or is lower than the video bit rate, the network is overloaded and the video quality is not satisfied. The network status can not only be seen from the video quality, but also can be seen from the monitoring results. The monitoring script gives accurate indications of whether the network is overloaded from the estimated packet loss rate and the average latency.

4.1.5 Noise Streaming Tests

Noise streaming tests mimic a real-life network scenario where multiple services are carried over the same network. Video bit rate was 10 Mb/s and no link capacity limitation was assigned. 5 Mb/s noise was generated by hosts located on links between two streaming hosts.

When the noise host was sending out traffic on links, the video was not playing smooth continuously. For some periods, the estimated packet loss rate reached as high as 40%.

All results of streaming tests have been documented from which we can define the emulated when network is capable of satisfying transmission requirements. Combining all monitoring results and the video view quality, we found that when the latency attained around 0.2 second the video began to be lagging. Therefore, integration with the video group can be based on the average latency. Communication with the video group uses socket TCP server/client which generates TCP packets from the client to the server. An indicated number is carried inside the payload of the TCP packet, from which the video group could calculate the new desired bit rate. Table 4.1 shows limitations set for different bit rate requirements.

Bit-rate	**Latency**	**Indicated Number**
Go Higher	0 - 90 ms	+ 1
Keep	90 - 200 ms	keep
Go Lower	>200 ms	- 5

Table 4.1: Integration with the Video Group Based on the Latency.

When the monitoring detects the bit rate should be changed, a TCP packet with the indicated number encapsulated in the payload will be sent to the video group. The adjustable encoding bit rate range is between 32 Kb/s and 16 Mb/s, the corresponding level is indicated as a number from 0 to 255. If the average latency is too high (more than 0.2 second), the value of the indicated number decreases by 5 from 255 step by step. Every time the average latency is higher than 0.2 second, the number decreases by 5. When the latency is lower than 0.09 second, meaning the network is capable of delivering a higher bit rate video, the indicated number carried in the TCP packet will increase by 1.

4.2 Integrated Tests

Integrated tests with the video group were taken to test if all functions provided by our group work properly. Also the bit rate control between two groups is one parameter to be considered during the integrated tests. Setup for tests is illustrated in figure 4.2.

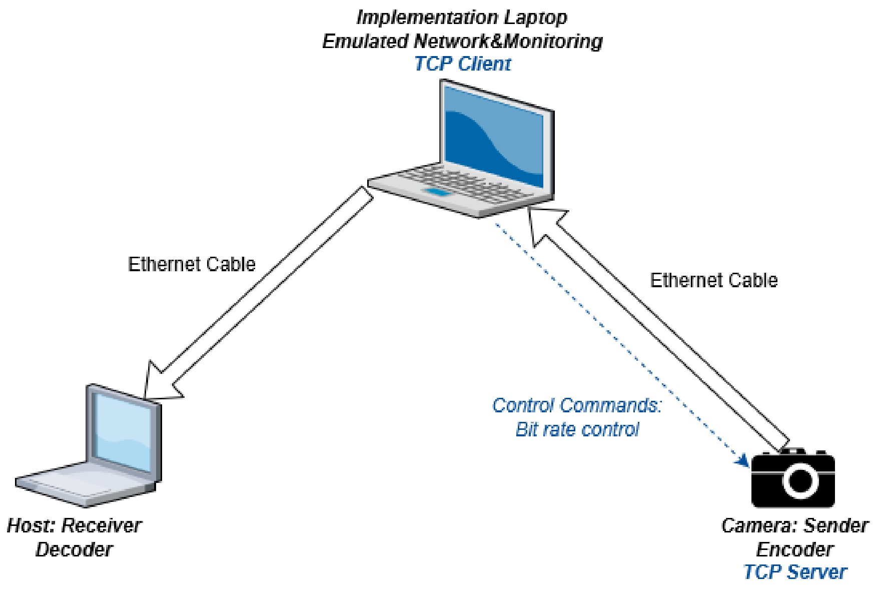


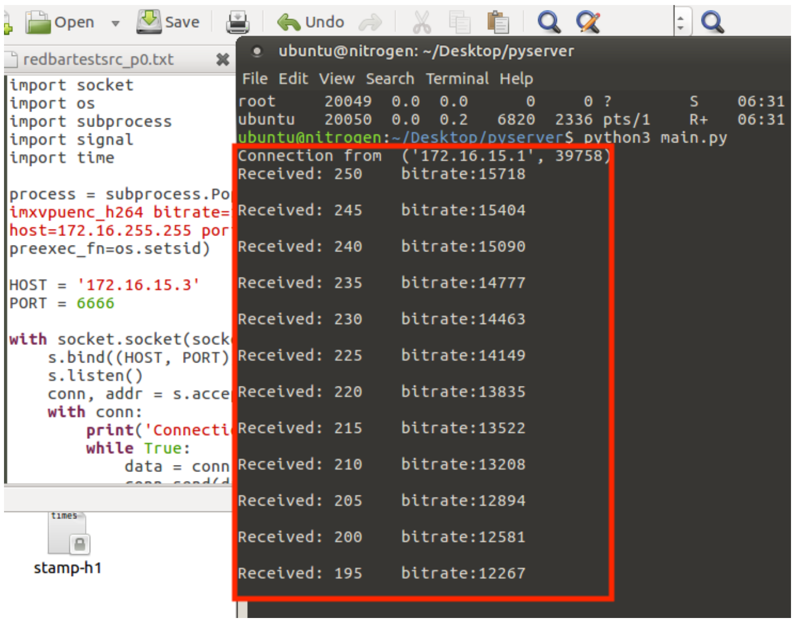
Figure 4.2: Setup for Integrated Tests.

A camera filming live videos encodes the video and sends UDP packets through a Ethernet cable to the implementation laptop. The implementation laptop passes packets across the emulated network to the receiver laptop who decodes packets and plays the video. Feedback for the video bit rate is delivered by sending control strings in TCP packets back to the sender.

Initial transmission video bit rate was 16 Mb/s while the network capacity was 10 Mb/s. The video on the receiver laptop was blurred and lagging at first, the camera received commands requiring lower bit rates as shown in figure 4.3.

When the bit rate reached at around 10 Mb/s, commands can be seen from figure 4.4. Obviously, the bit rate control messages were helping the video bit rate keep at a suitable range. The bit rate shown in figure 4.3 and 4.4 was the bit rate assigned to the encoder on the sender camera. The monitoring application located upon the emulated network usually monitored a higher bit rate, therefore the transmitted bit rate was around 10 Mb/s.

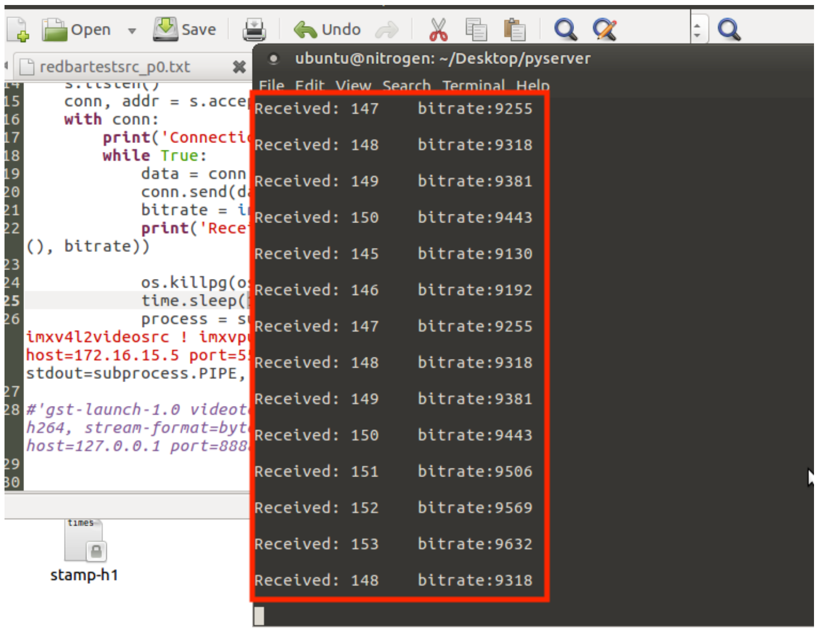
The integrated tests proved the good capability of the emulated network and the monitoring script. Also the communication through the socket TCP server/client for the bit rate control was stable and reliable.


Open
Save
Undo
redbartestsrc_p0.txt
ubuntu@nitrogen: ~/Desktop/pyserver
File Edit View Search Terminal Help
stamp-h1

```
import socket
import os
import subprocess
import signal
import time

process = subprocess.Po
imxvpuenc_h264 bitrate=
host=172.16.255.255 por
preexec_fn=os.setsid)

HOST = '172.16.15.3'
PORT = 6666

with socket.socket(sock
    s.bind((HOST, PORT)
    s.listen()
    conn, addr = s.acce
    with conn:
        print('Connecti
        while True:
            data = conn
```

```
root      20049  0.0  0.0      0     0 ?        S    06:31
ubuntu    20050  0.0  0.2   6820  2336 pts/1    R+   06:31
ubuntu@nitrogen:~/Desktop/pyserver$ python3 main.py
Connection from  ('172.16.15.1', 39758)
Received:  250     bitrate:15718

Received:  245     bitrate:15404

Received:  240     bitrate:15090

Received:  235     bitrate:14777

Received:  230     bitrate:14463

Received:  225     bitrate:14149

Received:  220     bitrate:13835

Received:  215     bitrate:13522

Received:  210     bitrate:13208

Received:  205     bitrate:12894

Received:  200     bitrate:12581

Received:  195     bitrate:12267
```

Figure 4.3: Commands Requiring Lower Bit-rates.


Open
Save
Undo
redbartestsrc_p0.txt
ubuntu@nitrogen: ~/Desktop/pyserver
File Edit View Search Terminal Help
stamp-h1

```
    conn, addr = s.acce
    with conn:
        print('Connecti
        while True:
            data = conn
            conn.send(d
            bitrate = i
            print('Rece
(), bitrate))

            os.killpg(o
            time.sleep(
            process = s
imxv4l2videosrc ! imxvp
host=172.16.15.5 port=5
stdout=subprocess.PIPE,

#'gst-launch-1.0 videot
h264, stream-format=byt
host=127.0.0.1 port=888
```

```
Received:  147     bitrate:9255

Received:  148     bitrate:9318

Received:  149     bitrate:9381

Received:  150     bitrate:9443

Received:  145     bitrate:9130

Received:  146     bitrate:9192

Received:  147     bitrate:9255

Received:  148     bitrate:9318

Received:  149     bitrate:9381

Received:  150     bitrate:9443

Received:  151     bitrate:9506

Received:  152     bitrate:9569

Received:  153     bitrate:9632

Received:  148     bitrate:9318
```

Figure 4.4: Commands for Stable Bit-rates.

5 Interfaces

’Network Emulation and Verification’ system block interfaces with two other blocks - application based monitoring and control system block and video encoding and decoding system block. Both interfaces have agreed with sub-project teams responsible for neighbouring system blocks.

In order to present data to the ’Application Based Monitoring and Control’ group we insert our gathered monitoring data (discussed in chapter 3) into a InfluxDB database created by the control group.

’Video Encoding and Decoding’ team sends video data over the emulated network, thus one of interfacing points is an Ethernet connection over a switch receiving the video data. The second interfacing point is used to provide feedback on network performance (with purpose of changing video bit rates). A socket TCP server/client connection is set up at both ends, communication between two groups uses the same Python libraries. A specific string encapsulated in TCP packet as the payload being sent to the video group on a specific port would indicate a poor network performance (based on monitoring latency statistics). The video group decodes the payload and gets an indicated number from which the new bit rate can be calculated. Detailed integrated implementation and integrated tests have been described in chapter 4.

6 Conclusion

Design and implementation of the 'Network Emulation and Verification' system block are discussed in this report. Network architecture is designed and chosen to emulate a service provider network. The emulated network is implemented in a virtual SDN environment, where key components include LINC-OE, OVS switches, Mininet network emulator and ONOS SDN controller. Traffic transmitted over the virtual network is analysed by a python application, which uses socket packet capture tool. The results of said analysis are available for the 'Application Based Monitoring and Control Group' by sharing data in an InfluxDB database. Interface with the 'Video Encoding and Decoding' system block uses a TCP client to send bit rate feedback to a TCP server running on the video encoding node, enabling the 'Video Encoding and Decoding' system block to adjust the video bit rate.

Tests include streaming tests within the system block and integrated tests with other system block. Emulated network performance has been tested using real hosts and traffic. Streaming test results indicate sufficient capabilities for the purpose. Performance of the monitoring application has also been proved reliable and helpful within a suitable range of bit rates from streaming tests. Integrated tests with the 'Video Encoding and Decoding' system block show stable capabilities of the emulated network and the monitoring application regarding processing real video data.

References

[1] About scapy. https://scapy.readthedocs.io/en/latest/introduction.html. Accessed: 2019-06-20.

[2] Does libpcap use raw sockets underneath them? https://stackoverflow.com/questions/7856509/does-libpcap-use-raw-sockets-underneath-them, . Accessed: 2019-06-20.

[3] Transmission of ip datagrams over ethernet networks. https://sanmati4.com/transmission-of-ip-datagrams-over-ethernet-networks-rfc-894/, . Accessed: 2019-06-12.

[4] J. Postel. *RFC 791: Internet Protocol: Darpa Internet Program Protocol Specification*. Internet Engineering Task Force (IETF), 1981.

[5] B. Mukherjee. *Optical WDM Networks*. Springer US, 2006.

[6] Virtualbox user manual. https://www.virtualbox.org/manual/ch01.html. Accessed: 2019-04-07.

[7] Mininet overview. http://mininet.org/overview/. Accessed: 2019-04-07.

[8] What is open vswitch? http://docs.openvswitch.org/en/latest/intro/what-is-ovs/. Accessed: 2019-04-07.

[9] Linc - openflow software switch. https://github.com/FlowForwarding/LINC-Switch, . Accessed: 2019-04-07.

[10] H. Balakrishnan G. Parulkar L. Peterson J. Rexford S. Shenker J. Turner N. McKeown, T. Anderson. Openflow: Enabling innovation in campus networks, 2008.

[11] Onos wiki home. https://wiki.onosproject.org. Accessed: 2019-04-07.

[12] Cord: About. https://opencord.org/about/. Accessed: 2019-04-07.

[13] opticalutils.py, classes and methods for integrating mininet and linc-oe. https://github.com/opennetworkinglab/onos/blob/master/tools/test/topos/opticalUtils.py. Accessed: 2019-04-07.

[14] scapy - interactive packet manipulation tool. https://linux.die.net/man/1/scapy, . Accessed: 2019-06-20.

[15] What is a raw socket? https://www.wisegeek.com/what-is-a-raw-socket.html, . Accessed: 2019-06-12.

[16] raw - linux ipv4 raw sockets.
http://man7.org/linux/man-pages/man7/raw.7.html, . Accessed: 2019-06-20.

[17] Influx data official introduction.
https://www.influxdata.com/time-series-database/. Accessed: 2019-04-07.

[18] Socket official documentation.
https://docs.python.org/3/library/socket.html#creating-sockets.
Accessed: 2019-06-12.

[19] Internet assigned numbers authority: Ieee 802 numbers.
https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1, . Accessed: 2019-06-12.

[20] Internet assigned numbers authority: Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml, .
Accessed: 2019-06-12.

[21] Python multiprocessing official documentation.
https://docs.python.org/2/library/multiprocessing.html. Accessed:
2019-06-13.

[22] Ensuring data integrity with hash codes. https://docs.microsoft.com/en-us/dotnet/standard/security/ensuring-data-integrity-with-hash-codes.
Accessed: 2019-05-28.

[23] J. Postel. *RFC 786: User Datagram Protocol*. Internet Engineering Task Force (IETF), 1980.

[24] Python: Dictionary and its properties.
https://www.rookieslab.com/posts/python-dictionary-and-its-properties.
Accessed: 2019-05-28.

A Contribution

A.1 Report Writing

Paulius:

- Abstract
- Introduction
- Network Emulation
- Interfaces
- Conclusion
- Editing

Radheshyam:

- Monitoring_Implementation Methods_Scapy
- Monitoring_Implementation Methods_Raw Socket
- Monitoring_Implementation_Packet Analysis_Problem with Arrays

Yichen:

- Monitoring_Overview
- Monitoring_Implementation Methods_InfluxDB
- Monitoring_Implementation (All)
- Monitoring_Achieved Outcomes
- Tests
- Interfaces
- Conclusion

A.2 Project Implementation

Paulius:

- The emulated network - topology design, virtual SDN-based network setup and everything related to it.
- Initial monitoring script using scapy - packet capture and analysis.
- Monitoring script using raw sockets - packet capture, analysis, control messages, influxDB integration, multiple versions and scenarios for improved efficiency.
- Integration.
- Testing using iperf, video streaming and video group devices.

Radheshyam:

- Monitoring script using scapy: dictionary implementation

- Testing using iperf, video streaming and video group devices.

Yichen:

- Monitoring script using Scapy: all calculation functions for analysis, multiple versions for suitable monitoring parameters/outcomes
- Raw monitoring script using Pycap: pycap capture
- Monitoring script using sockets: discussions over calculations with Paulius
- Testing using iperf, video streaming and video group devices.

B Revision History

Some of the revisions include additions and editing to previously included report sections, however some of them are completely new to this report while some have been completely changed compared to previous reports. Sections added or changed after the midterm report:

- Socket packets capture and packets analysis
- Tests description
- Bit rate control integration with the video group
- InfluxDB integration with the control group
- Changes to implementation section in Network Emulation chapter