ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

School of Information Sciences and Technology/Department of Informatics

MSc in Information Systems Development and Security

Implementing a Recirculation-Based Selective Forwarding Unit Using P4

Efthymios Papageorgiou f3312410

Supervisor

Prof. Georgios Xylomenos

Athens, 2026

Abstract

Modern video conference applications rely on Selective Forwarding Units (SFU) to replicate packet traffic between participants. Traditional SFUs commonly operate in user space and as such are required to move packets to and from it so they can copy them. These operations have a significant negative impact on latency. Switches with programmable dataplanes offer a solution to this problem enabling SFUs to run directly on switches and thus not to incur such costs. Using P4, an open-source programing language for switches, specifically its recirculation feature, in which packets can reenter a switch pipeline instead of being transmitted, a more efficient SFU was developed. After its development, it was compared with a Python-based SFU implementation running in user space on a Linux host, instead of on a switch, similarly to traditional SFUs. Even in simulated hardware, the P4-Based SFU outperformed its counterpart by demonstrating lower latency, increased stability and reduced packet loss. The results highlight the potential of programmable dataplanes for the development of high performance SFUs.

Περίληψη

Οι σύγχρονες εφαρμογές βιντεοδιάσκεψης βασίζονται σε Selective Forwarding Units (SFU) για την αναπαραγωγή της κυκλοφορίας πακέτων μεταξύ των συμμετεχόντων. Οι παραδοσιακές SFU λειτουργούν συνήθως στον χώρο χρήστη και, ως εκ τούτου, απαιτείται η μεταφορά πακέτων από και προς αυτόν προκειμένου να αντιγραφούν. Αυτές οι λειτουργίες επιβαρύνουν σημαντικά την καθυστέρηση. Οι μεταγωγοί με προγραμματιζόμενα επίπεδα δεδομένων (programmable dataplane) προσφέρουν μια λύση στο πρόβλημα, επιτρέποντας στις SFU να εκτελούνται απευθείας στον μεταγωγό, κι έτσι να αποφεύγουν αυτό το κόστος. Χρησιμοποιώντας την P4, μια γλώσσα προγραμματισμού ανοιχτού κώδικα για μεταγωγούς, και συγκεκριμένα την ανακυκλοφορία (recirculation), μέσω της οποίας τα πακέτα μπορούν να εισέλθουν εκ νέου στον αγωγό επεξεργασίας αντί να μεταδοθούν, αναπτύχθηκε μια πιο αποδοτική SFU. Μετά την ανάπτυξή της, συγκρίθηκε με μια υλοποίηση SFU σε Python που εκτελείται σε χώρο χρήστη σε έναν υπολογιστή Linux, όπως συμβαίνει στις παραδοσιακές SFU. Ακόμη και σε προσομοιωμένο υλικό, η SFU που βασίζεται στην P4 υπερείχε της αντίστοιχης υλοποίησης, παρουσιάζοντας χαμηλότερη καθυστέρηση, αυξημένη σταθερότητα και μειωμένη απώλεια πακέτων. Τα αποτελέσματα αναδεικνύουν τις δυνατότητες των προγραμματιζόμενων επιπέδων δεδομένων για την ανάπτυξη SFU υψηλής απόδοσης.

Acknowledgments

I would like to thank my supervisor Prof. Georgios Xylomenos for his guidance and feedback throughout this project. I am also grateful to Pavlos Tsikrikas, who, alongside my supervisor authored the paper “A Selective Forwarding Unit Implementation in P4”, which provided the foundation this project was built. Finaly, I would like to thank the Onassis Foundation for providing the scholarship which enabled me to pursue my master’s degree.

Table of Contents

List of Figures

List of Tables

List of Acronyms

P2P: Peer-to-Peer

MCU: Multipoint Control Unit

SFU: Selective Forwarding Unit

RTP: Real-time Transport Protocol

UDP: User Datagram Protocol

P4: Programming Protocol-Independent Packet Processors

PSA: Portable Switch Architecture

TNA: Tofino Native Architecture

I2E: Ingress-to-Egress Clone

E2E: Egress-to-Egress Clone

API: Application Programming Interface

BIER: Bit Index Explicit Replication

BIER-FRR: Bit Index Explicit Replication – Fast Reroute

BMv2: Behavioral Model v2:

IP: Internet Protocol

OVS: Open vSwitch

1. Introduction

1.1. Real-time conferencing and latency

In today's increasingly digital world, video conferencing applications like Microsoft Teams, Google Meet and Zoom are becoming more common in many aspects of life. A major factor for the smooth operation of such real-time video streaming applications is keeping latency as low as possible to avoid potential communication disruptions, since most users generally tolerate delays up to 150ms [1]. At the same time, with resolution and bitrate increasing (e.g., 4K conferencing), the need for fast packet replication and forwarding becomes more pressing to facilitate the smooth operation of video conferencing applications. One of the main factors that need to be taken into consideration when attempting to maintain low latency is the application architecture, and especially, its handling of multiparty communications.

1.2. Types of video conference application architecture

There are multiple architectural approaches for allowing communication between several hosts developers can follow [2] [3]:

- Full-mesh peer-to-peer networks (P2P): In this case every host receives packets from and transmits packets to every other host at the video conference. This is a very simple concept that has several advantages, mainly the ability to easily use end-to-end encryption and give users the ability to customize exactly which streams they want to receive. At the same time, it results in each host receiving and sending a number of packet streams that can be up to the number of hosts it shares the conference with. With many hosts, this can result in a very large bandwidth cost, making this type of connection impractical, even though it has the lowest possible delay [4] [3].

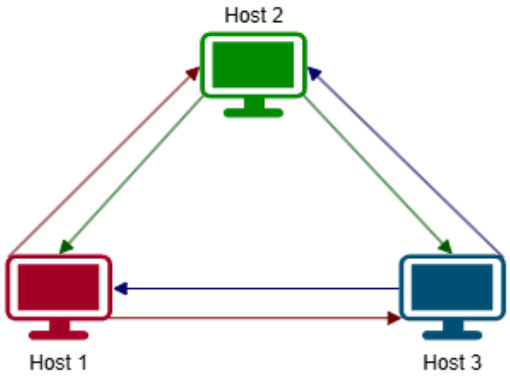

Figure 1 : P2P architecture

- Multipoint Control Units (MCUs): In this type of architecture a server acts as an intermediary for the conference. Each host sends its stream exclusively to the server, which then combines the streams from all hosts into a single stream. While this drastically reduces the number of streams that the hosts need to deal with and, thus, the required bandwidth, it also introduces a host of new issues. Firstly, it introduces the need for synchronization between all the different streams sent to the server, meaning that delays in one host's transmission can negatively impact the streams from all other hosts. The computational overhead required is also increased, since there is a need to decrypt and re-encrypt the streams to enable combining them, while also introducing potential security risks and privacy concerns. Finally, combining all streams into one means that hosts are forced to receive the data of every stream, even if they have no use for it, meaning increased bandwidth requirements for receiving essentially useless data. As an example, in a digital classroom setting while the teacher might want to be able to receive the video streams of all participants, the students might only wish to see the teacher's stream, but they are forced to receive each other's streams anyway [5] [3].

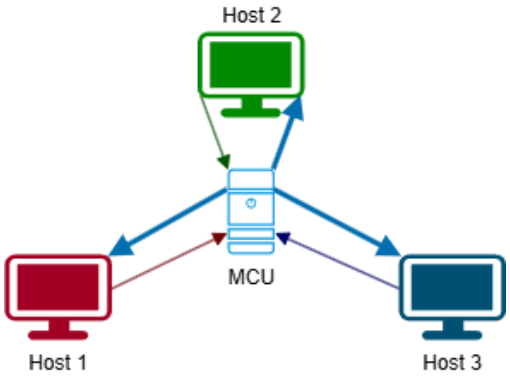

Figure 2 : MCU architecture

- Selective Forwarding Units (SFUs): SFUs are the current dominant video conference architecture, since they manage to combine the centralized control of MCUs while keeping some of the advantages of peer-to-peer connections like their versatility. Like an MCU, the SFU is also located in the "center" of a video conference and receives the streams transmitted by all participants. Unlike an MCU, the streams are not combined into one, instead the SFU forwards to each host multiple streams, but only the ones that the host has asked for. This means that each host is transmitting only one stream, keeping the low upload requirements of the MCU architecture. While it does receive multiple streams, since it is only the ones it needs, it is most likely that their combined required download bandwidth is lower than an MCU's single large stream. SFUs are also commonly used alongside layered coding, where each host creates multiple streams, each with a different resolution. This way, others in the video conference can select the resolution their network can best support, without requiring the SFU to perform transcoding, thus reducing overhead and delays. Finally, unlike MCUs, SFUs can make use of end-to-end encryption, matching the privacy advantages of P2P architectures [6] [3].

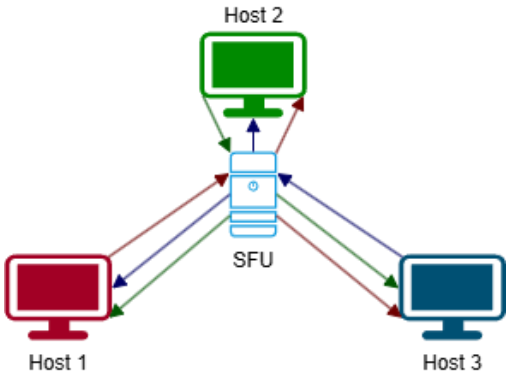

Figure 3 : SFU architecture

1.3. How SFUs work in detail

The main purpose of a Selective Forwarding Unit (SFU) is to decide which participant in a conference gets forwarded what stream, based on the rules of the conference. It does not transcode or mix streams, it only forwards and, if needed, makes copies of the stream's packets so multiple participants can receive the same stream. Media flows from each participant to the SFU and then to other participants over RTP/UDP. The forwarding logic is based on the SFU maintaining lists of participants and the streams they have subscribed to. When a packet arrives at the SFU, the source stream is first identified and then the packet is replicated and forwarded to all the stream's subscribers. SFUs usually support features like taking decisions based on network conditions, dynamic subscriptions with the participants being able to change which streams they receive on the fly and scalable video coding, where participants send streams of different quality levels and the SFU decides which to forward based on the receiver's bandwidth. [3]

1.4. Programmable dataplanes and P4

Traditional networking devices like routers and switches have fixed dataplanes meaning that the functions they perform like parsing, forwarding and filtering are hardcoded. Devices with programmable dataplanes allow their operators to define how packets are processed and potentially change those definitions at a later date. This ability is particularly useful when it comes to trying out prototypes or specialized applications that require unusual forwarding logic.

P4 stands for Programming Protocol-Independent Packet Processors and is described as "a high-level language for programming protocol-independent packet

processors" [7]. It was developed with the goal of providing a way for switch operators to reconfigure their behavior after deployment as well as to provide target independence giving developers the ability to describe packet processing functionality independent of the hardware they are using. The basic concepts from P4 relevant to this project include:

Pipeline

In P4 the dataplane is organized as a packet processing pipeline. When a packet enters the switch, it goes through the following steps [7]:

- Parser: extracts the headers from the raw packet and stores them in fields that are carried in the rest of the pipeline along the packet.
- Ingress: does some of the processing, like rewriting headers or deciding forwarding.
- Egress: process the packet further, for example, using recirculation.
- Deparser: reassembles the headers into the packet for transmission.

It should be noted that the above stages exist in the v1model architecture which was followed by this project; other architectures for P4 (such as PSA or TNA) also exist and might make use of additional pipeline stages [8] [9]. All processing takes place as the packet goes from stage to stage.

Metadata

Metadata is temporary information carried alongside a packet throughout the pipeline, without them being part of the packet itself. Unlike headers, they are created by the P4 program itself and never leave the switch. They usually relate to data like flags or calculated values [7], which are specific to a packet.

Registers

In P4 registers are stateful memory arrays inside the dataplane. They are defined with a type and fixed size and unlike metadata and headers, they persist across packets. They can be read and written using the appropriate P4 operations [10], and serve as persistent memory in the switch.

Match action tables

They are tables that match packet fields and apply some predetermined action when a match is found. The match keys usually include header fields or metadata [7].

Recirculation

Unlike traditional programming languages, the P4 pipeline is completely linear and does not include loops. Recirculation in P4 allows a packet that has completed the pipeline to be sent back to the beginning of the pipeline to be processed again. In the v1model architecture, which was used by this project, a packet can be selected for recirculation by being redirected to a designated recirculate port [11] [12].

Packet Cloning

Cloning simply creates a duplicate of the packet. In this project two types of cloning are relevant [9] [8]:

- Ingress to egress (I2E): The clone is created based on the packet as it was at the start of the ingress and is injected at the beginning of the egress.
- Egress to egress (E2E): The clone is created based on the packet as it was at the start of the egress and is injected back into the egress.

P4Runtime

P4Runtime is a control-plane API designed to manage devices programmed in P4. While the P4 language specifies how packets are processed in the dataplane, P4Runtime enables external controllers to reconfigure P4 switches, without the need for recompilation, for example, by modifying the registers or updating the match action tables [10] .

1.5. Motivation and related work

The main motivation behind this project was to create a more realistic alternative to the P4-Based SFU created for the paper “A Selective Forwarding Unit Implementation in P4” authored by Pavlos Tsikrikas and George Xylomenos. In this work the authors of the paper created a P4-Based SFU taking advantage of the fact that P4 runs on a switch rather than a server, leveraging the inherent advantages of hardware-based packet processing. Switches are optimized for fast operations, avoiding the overhead of system calls and kernel–user space crossings present in server implementations. They concluded that their SFU had major advantages when it came to minimizing latency compared to a user space SFU written in Python which they also created [13]. However, one limitation of their study was that the SFU developed relied on multicasting to create multiple copies of each packet, therefore it had to be located in such a way that each destination would be served by a different port, which would only make sense for a small local area scenario. In practice, most SFU servers make use of only a limited number of ports to communicate with users and are usually located on the edges of networks.

The paper “P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast” written by Daniel Merling, Steffen Lindner and Michael Menth, served as inspiration for a more realistic P4-based SFU. In this paper the authors implement Bit Indexed Explicit Replication (BIER) using P4; in BIER, packets are replicated using a bitmap included in their header. While BIER itself is not directly related to this project, the authors implemented the novel idea of making use of recirculation in conjunction with the use of metadata flags to achieve the replication of packets, thus avoiding the limitations imposed by P4’s lack of real loop functionality. Although this approach is slower than multicast groups, it avoids the need for dedicating a port to each host connected to the SFU [12].

The main purpose of this project was to attempt to develop a new SFU in P4, using recirculation to avoid multicast and, as in the original publication, compare it with a similar SFU developed in Python, in order to determine if it maintained its lower latency advantage and how its latency scaled when increasing the number of recipients the SFUs had to transmit to. The project also had the secondary goal of making the P4-Based SFU dynamically configurable, to take further advantage of P4’s capabilities; for example, this could be used to allow adding and dropping senders and receivers from a conference.

All source code, scripts, and experimental data used in this thesis are available at: https://github.com/efthpapag/p4-SFU .

2. Design and implementation of the SFUs

2.1. Design assumptions

In order to assess the benefits of using P4 two SFUs were created. One was programmed using P4_16 and could be executed by P4 compatible programmable switches. For this project, it ran on a BMv2 software switch and, more specifically, BMv2's simple_switch target, the reference software implementation of the v1model architecture [14]. This type of setup provided a programmable dataplane that supported all the actions needed for the implementation of the SFU's functionality. While BMv2's performance does not match that of a hardware switch, it is commonly used for prototyping and testing P4 programs [8].

The other SFU was a standard user space SFU programed using Python 3 and ran on a Linux server. Unlike the SFU written in P4, this one had to use socket calls to receive and send packets, performing packet duplication on the user level. For the sake of keeping the comparison fair, it worked in a similar way to the P4-Based SFU.

Another assumption made about the system was that each host that took part in a conference would only have the ability to transmit a single stream with a steady bitrate and packet size. This way the SFUs would only need to consider the host's IP address when determining where they had to address the replicated packets to. The hosts and the SFUs would communicate with each other through a switch located in the center of the network.

The way in which the SFUs decided which host they would send copies of a packet to in this project was based on a subscription model similar, though not completely identical to, the way a lot of real SFUs work [3]. The basic concept is that there are groups each of which correspond to the IP address of a host that transmits a stream of packets; hereafter this host will be referred to as the presenter. This group is essentially a list with the IP addresses of all the hosts that wish to receive the presenter's stream. These hosts will from now on be referred to as the viewers. The idea is that when a new host enters a video conference a new empty group is created with it as the presenter. When another host wants to receive its stream, it subscribes to the stream by adding itself as a viewer in the presenter's group. So essentially every host sends its stream to the viewers of the group it is the presenter of. It is important to note that for this project it was assumed that each host sends only one stream, in reality this usually is not the case and hosts often send multiple streams, for example different streams for audio, camera feed, screen sharing etc. [2]

In addition to the SFUs, a few supporting scripts were created to assist with testing as well as the setup of the system. The first was a Python script which would transmit a number of packets from the host it was running, using Python's Scapy library. Scapy is a Python library that enables the creation, manipulation, transmission, and capture of network packets directly within Python scripts [15]. Later on in the project, the way the packets were transmitted was changed in the experimental part of the project, as detailed in section "3.1. Environment configuration". A second Python script, also using Scapy, this time with the purpose of recording any packets received by the host was also created. In addition to the two Python scripts, a shell code script was created to set up some parts of the network. This script's uses included configuring the non-programmable software switch that was in the center of the network, assigning an IP address to the port of the BMv2 switch that the SFU was receiving packets from, configuring a clone/mirroring session on the same switch and, finally, resetting the BMv2's registers.

2.2. P4-based SFU

In this section the development and function of the P4-Based SFU is described. Firstly, the main idea behind it is described, then the main changes that happened during its development and, finally, its final version is described in detail.

The central idea behind the implementation of the P4-Based SFU is recirculation. When the SFU receives a packet, it must make a clone of it for each host that needs to receive it. Since the goal was to use only one port for outgoing packets, multicasting could not be used, since it would require a number of ports equal to the number of viewers. Sending multiple copies of the packet requires the original packet passing through the pipeline once for each viewer, with a copy being made and sent for every pass. In order to circumvent P4's lack of loops, packet recirculation would be used. When using recirculation, the packet passes through the pipeline as usual, but at the end of the egress stage, instead of being sent to the network, it goes back to the beginning of the pipeline as shown in Figure 4. The egress stage can be informed which packets are to be recirculated and which are not by a metadata field acting as a flag. In P4 metadata are fields carried alongside a packet through the pipeline, without them being part of the packet itself. P4 can also preserve selected metadata, so that they can follow the packet between recirculations.

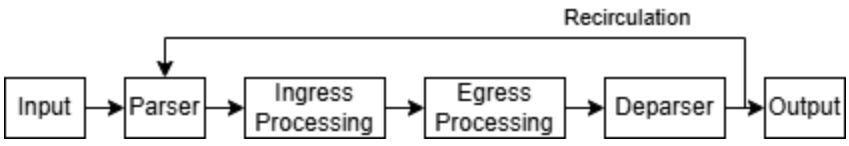

Figure 4 : Recirculation mechanism

Development

Initially, the SFU was not created as a simple forwarding switch, to make sure the pipeline's basic components had been correctly implemented. It is also important to note that during the development phase of the project the SFU did not have its own IP address; instead, the central switch directed all packets not coming from the SFU towards it.

The first step towards implementing the actual SFU functionality was overriding a packet's destination, so that it can be sent to a host selected by the SFU. The SFU would override the destination of all packets with the IP address of one of the hosts, named h1, which had an IP address of 10.0.0.1, as can be seen in Figure 5.

```
action forward_to_h1() {
    meta.force_h1 = 1;
    hdr.ethernet.srcAddr = SWITCH_MAC;
    hdr.ethernet.dstAddr = 0x080000000111; // h1 MAC
    hdr.ipv4.dstAddr = 0x0a000101;         // 10.0.1.1
    standard_metadata.egress_spec = 2;     // port 2
}
```

Figure 5 : Forward to h1 action

The next step was to implement duplication, which required cloning. The goal was to detect packets where h1, the presenter, was a packet's source, and clone it. The host named h2, with an IP address of 10.0.2.2, and the packet's original destination host would act as viewers. To achieve this, when a packet arrived from h1 it would be cloned, the clone would be sent to the network unmodified, ending up at the original destination, while the original would continue down the ingress pipeline when its destination changed to h2, as seen in Figure 6.

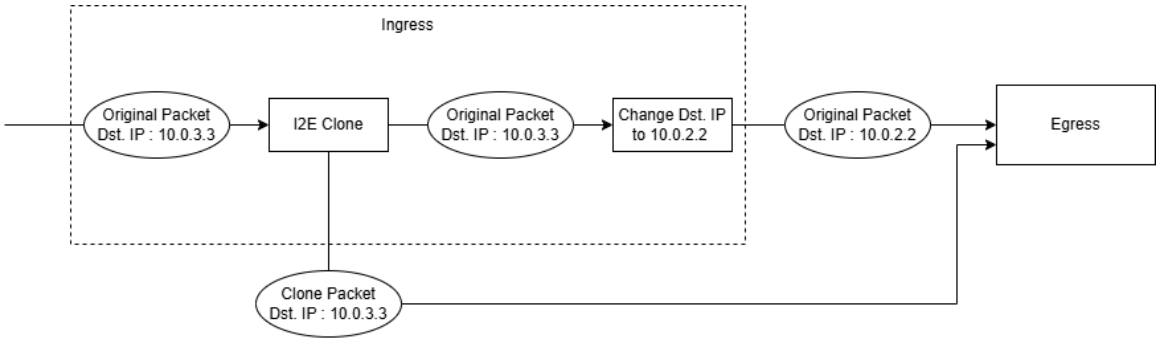

Figure 6 : Cloning example

Based on [12] the type of cloning used was ingress to egress (I2E). This type creates a clone packet with the same contents as the original at ingress and then immediately injects it into egress.

Next, it was time to give the program proper SFU functionality, by implementing the recirculation of packets. To implement it, a new metadata field called *viewer_index* was added and used to keep track of the number of viewers that had already been sent copies of the packet. Another metadata field called *recirculate_next* was used as a flag to notify the egress if the packet should be recirculated or not. When a packet entered ingress, if *viewer_index* was less than the number of viewers, *recirculate_next* was set to 1 to indicate the packet should be recirculated. Otherwise, it was set to 0 and *viewer_index* was also reset to zero. The IP and MAC addresses of the viewers were hardcoded and used based on the index. Index 0 corresponded to h2's address, index 1 to h3's and index 2 to h4's.

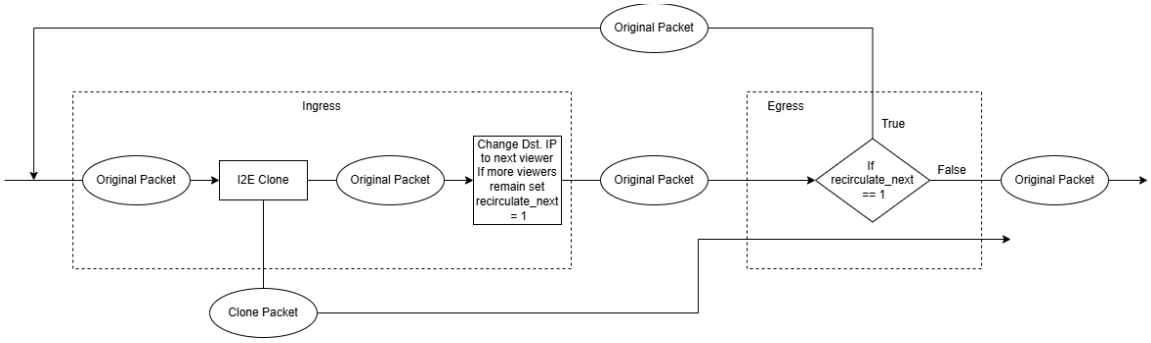

Figure 7 : Pipeline with recirculation and cloning added

```
action recirculate(bit<32> max_viewers) {
    // Rewrites packet headers for the current viewer.
    // If more viewers remain, recirculate the packet.
    // Otherwise, send it out.
    hdr.ethernet.srcAddr = SWITCH_MAC;
    hdr.ethernet.dstAddr = meta.viewer_mac;
    hdr.ipv4.dstAddr     = meta.viewer_ip;
    hdr.ipv4.ttl         = hdr.ipv4.ttl - 1;

    if (meta.current_viewer_index < (max_viewers - 1)) {
        // More viewers to reach, prepare for recirculation
        meta.current_viewer_index = meta.current_viewer_index + 1;
        meta.recirculate_next = 1; // Signal to Egress to recirculate
        standard_metadata.egress_spec = 2;
    } else {
        // Last viewer, send out
        standard_metadata.egress_spec = 2;
    }
}
```

Figure 8 : Recirculate action

At this point, a bug that will be referred to as the *extra packet bug* first appeared; it would not be solved until the end of the development phase. Although the SFU had no IP yet and the packets were sent to it automatically by the central switch, the packets still needed to have a valid destination, so the presenter set one of the viewers' IPs as the destination address. It was observed that an extra copy of the packet was sent to that viewer, *before* all the expected copies were transmitted. As an example, if h1 addressed the packet to h3, the packets sent from the SFU would go to h3, h2, h3, h4, in that order. If the packet was addressed to a host that was not a viewer, then the first packet would still go to it once.

After the basic functionality of SFU had been implemented, it was time to make it work dynamically, beginning with the viewers. The foundation of the SFUs dynamic functions was that all IP and MAC addresses would be read by the P4 program from the switch's registers. Registers are independent from the P4 program and as such, unlike hardcoded values, their values can be modified without the need to restart the SFU. These registers could be modified using an external shell code script. Also, the *viewer_index* variable was renamed to *dynamic_viewer_count* and indicated the register which stored the address of the next viewer.

```
register<bit<32>>(1) viewer_count;
register<ip4Addr_t>(MAX_VIEWERS) viewer_ips;
register<macAddr_t>(MAX_VIEWERS) viewer_macs;
```

Figure 9 : Register definition

```
apply {
    if (hdr.ipv4.isValid()) {
        // If the packet is from the presenter
        if (detect_presenter.apply().hit) {
            // Load viewer IP and MAC from registers
            viewer_ips.read(meta.viewer_ip, meta.current_viewer_index);
            viewer_macs.read(meta.viewer_mac, meta.current_viewer_index);

            // Read the current viewer count from control plane
            bit<32> dynamic_viewer_count;
            viewer_count.read(dynamic_viewer_count, 0);

            // Rewrite and recirculate
            recirculate(dynamic_viewer_count);
        } else {
            // Forward to presenter
            forward_to_h1();
        }
    }
}
```

Figure 10 : SFU using registers to read viewer addresses

Next, in order to be able to have multiple presenters that could be changed dynamically, they were also stored in registers. When a packet entered ingress, its source IP would be matched to that of a presenter. Each register containing a

presenter's IP corresponded to a numbered group, essentially a list, of registers containing the information of the viewers that had subscribed to it. Originally there was only one presenter register, so matching the source IP with its contents was done through the match-action table shown in Figure 11, but when multiple presenters were added this was done thought a series of If statements, as shown in Figure 12.

```
table detect_presenter {
    key = {
        hdr.ipv4.srcAddr: exact;
    }
    actions = {
        initiate_fanout;
        NoAction;
    }
    size = 32;
    default_action = NoAction();
}
```

Figure 11 : Presenter matching table

```
bit<32> ip0; presenter_ips.read(ip0, 0);
if (hdr.ipv4.srcAddr == ip0) {
    matched_group = 0;
}

bit<32> ip1; presenter_ips.read(ip1, 1);
if (hdr.ipv4.srcAddr == ip1) {
    matched_group = 1;
}

bit<32> ip2; presenter_ips.read(ip2, 2);
if (hdr.ipv4.srcAddr == ip2) {
    matched_group = 2;
}
```

Figure 12 : Examples of presenter matching using If statements

In order to make register modifications easily, a P4Runtime controller was created. The controller used a combination of functions provided by P4Runtime and some supporting shell code scripts called by the P4Runtime, with the ability to add new groups and their presenter information, add viewers to existing groups, remove viewers from groups and delete groups. The reason for calling shell code scripts for some functions was that P4Runtime does not yet support read and write register operations for simple_switch [14].

```
Presenter group 1: IP=167772674, MAC=8796093022754, Viewers=3
Viewer 0: IP=167772417, MAC=8796093022481
Viewer 1: IP=167772931, MAC=8796093023027
Viewer 2: IP=167773188, MAC=8796093023300

Presenter group 2: IP=167772931, MAC=8796093023027, Viewers=3
Viewer 0: IP=167772417, MAC=8796093022481
Viewer 1: IP=167772674, MAC=8796093022754
Viewer 2: IP=167773188, MAC=8796093023300

Presenter group 3: IP=167773188, MAC=8796093023300, Viewers=3
Viewer 0: IP=167772417, MAC=8796093022481
Viewer 1: IP=167772674, MAC=8796093022754
Viewer 2: IP=167772931, MAC=8796093023027

Presenter group 4: IP=0, MAC=0, Viewers=0
```

Figure 13 : Registers before using the controller

```
(p4dev-python-venv) p4@p4dev:~/tutorials/exercises/SFU_Project_v8.0$ python3 controller.py
 add_group --group 4 --ip 1111 --mac 1111
presenter_ips[4] = 1111
presenter_macs[4] = 1111
viewer_counts[4] = 0
(p4dev-python-venv) p4@p4dev:~/tutorials/exercises/SFU_Project_v8.0$ python3 controller.py
 add_viewer --group 1 --ip 1111 --mac 1111
viewer_ips[15] = 1111
viewer_macs[15] = 1111
viewer_counts[1] = 4
(p4dev-python-venv) p4@p4dev:~/tutorials/exercises/SFU_Project_v8.0$ python3 controller.py
 remove_viewer --group 2 --ip 167773188
viewer_ips[26] = 0
viewer_macs[26] = 0
Removed viewer with IP 167773188 from group 2 at index 26
(p4dev-python-venv) p4@p4dev:~/tutorials/exercises/SFU_Project_v8.0$ python3 controller.py
 delete_group --group 3
presenter_ips[3] = 0
presenter_macs[3] = 0
viewer_ips[39] = 0
viewer_macs[39] = 0
```

Figure 14 : Example uses of the controller

```
Presenter group 1: IP=167772674, MAC=8796093022754, Viewers=4
Viewer 0: IP=167772417, MAC=8796093022481
Viewer 1: IP=167772931, MAC=8796093023027
Viewer 2: IP=167773188, MAC=8796093023300
Viewer 3: IP=1111, MAC=1111

Presenter group 2: IP=167772931, MAC=8796093023027, Viewers=3
Viewer 0: IP=167772417, MAC=8796093022481
Viewer 1: IP=167772674, MAC=8796093022754
Viewer 2: IP=0, MAC=0

Presenter group 3: IP=0, MAC=0, Viewers=0

Presenter group 4: IP=1111, MAC=1111, Viewers=0
```

Figure 15 : Registers after using the controller

The final major change to the P4-Based SFU was to resolve the extra packet bug mentioned previously. At first glance, the cause would appear to be obvious, since the SFU clones the original packet before changing its IP, so an unmodified clone of the original packet would be transmitted in the first loop. However, reversing the order of operations and placing the rewrite before cloning did not resolve the bug! The next possible solution was dropping the packet, but all attempts to drop it at either the ingress or egress failed. The solution was to remove the ingress-to-ingress (I2E) clone from the ingress stage and replace it with an egress-to-egress (E2E) clone in the egress stage. It would appear that cloning in the ingress was too early, since I2E

cloning creates a clone immediately when the packet enters the ingress stage, no matter where the actual clone command is located, meaning that the header rewrite was always happening after the clone, since both were in the ingress stage.

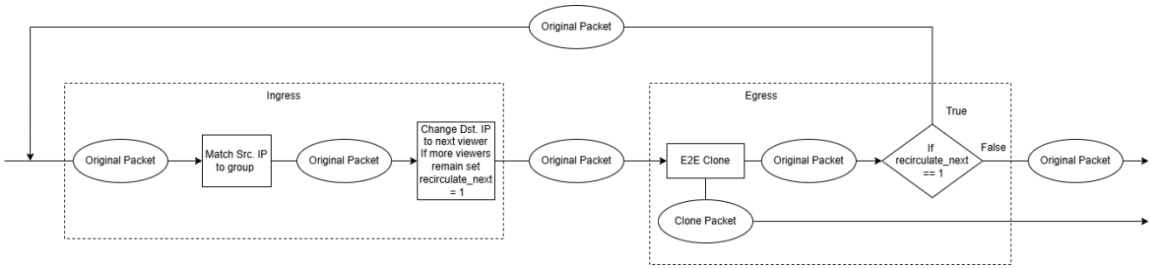

Figure 16 : Final version of the P4-Based SFU pipeline

Summary of the final version

To summarize, a description of how the final version of the P4-Based SFU works follows. When a packet enters the SFU, it is parsed and processed in the ingress stage. The source IP is checked against the presenter registers through a series of if statements; if a match is found, the corresponding group ID is noted. The current viewer's IP and MAC are read from the viewer registers using *meta.group_id * MAX_VIEWERS + meta.current_viewer_index* to find the registers location and the number of viewers in the group is also read.

```
bit<32> ip11; presenter_ips.read(ip11, 11);
if (hdr.ipv4.srcAddr == ip11) {
    matched_group = 11;
}

// Proceed if a valid presenter group was matched
if (matched_group < MAX_GROUPS) {
    meta.group_id = matched_group;

    bit<32> base = meta.group_id * MAX_VIEWERS + meta.current_viewer_index;
    viewer_ips.read(meta.viewer_ip, base);
    viewer_macs.read(meta.viewer_mac, base);

    bit<32> dynamic_viewer_count; // Number of viewers in group
    viewer_counts.read(dynamic_viewer_count, meta.group_id);

    prepare_recirculation(dynamic_viewer_count);

    meta.emit_clone = 1;
}
```

Figure 17 : Ingress pipeline (excluding most If statements used for group matching)

The packet's destination headers are rewritten with the viewer's information. If more viewers remain, the *recirculate_next* flag is set to 1 otherwise it is set to 0.

```
action prepare_recirculation(bit<32> dynamic_viewer_count) {
    // Rewrite headers
    hdr.ethernet.srcAddr = SWITCH_MAC;
    hdr.ethernet.dstAddr = meta.viewer_mac;
    hdr.ipv4.dstAddr     = meta.viewer_ip;
    hdr.ipv4.ttl         = hdr.ipv4.ttl - 1;

    // Decide whether to recirculate
    if (meta.current_viewer_index < dynamic_viewer_count - 1) {
        meta.current_viewer_index = meta.current_viewer_index + 1;
        meta.recirculate_next = 1;
    }else {
        meta.recirculate_next = 0;
    }

}
```

Figure 18 : Recirculation action

In egress, the packet is cloned (*CloneType.E2E*) so that a copy is emitted. If *recirculate_next* is set, the packet is recirculated back to ingress without ever leaving the switch to serve the next viewer. If not, the packet simply exits the switch after serving the final viewer.

```
control EgressProcessing(inout headers hdr,
                         inout metadata meta,
                         inout standard_metadata_t standard_metadata) {

    apply {
        if (meta.emit_clone == 1) {
            clone(CloneType.E2E, 100);
            meta.emit_clone = 0;
        }
        if (meta.recirculate_next == 1) {
            meta.recirculate_next = 0;
            recirculate_preserving_field_list(RECIRC_FIELDS);
        }
    }
}
```

Figure 19 : Egress pipeline

2.3. user space SFU

The user space SFU was implemented as a Python 3 script that was executed on a server and was made to work in a similar way to the P4-Based SFU, transmitting one packet at a time using a single port. This meant that the implementation was intentionally simple and unoptimized. It used only a single thread and pure Python packet handling. The goal was not to build a production grade SFU but to mirror the P4-Based implementation as closely as possible to keep the comparison fair.

Replication was achieved by looping over the set of intended recipients. In this SFU the groups were hardcoded and represented as a dictionary where the key is the group's presenter, and the value is a list with the IP addresses of the group's viewers. The script was scanning the server port that was connected to the rest of the network trying to detect any packets that arrived at it. When a packet entered the SFU its source address was read and through the previously mentioned dictionary the addresses of those that should receive it. Since Python, unlike P4, can make use of loops, there is no need for recirculation; the script executed a loop. In every repetition a clone of the original packet was created with the IP and MAC addresses of a viewer as its destination. The clone was then transmitted using the *sendp* function of Scapy. When clones had been sent to all viewers in the group the loop ended.

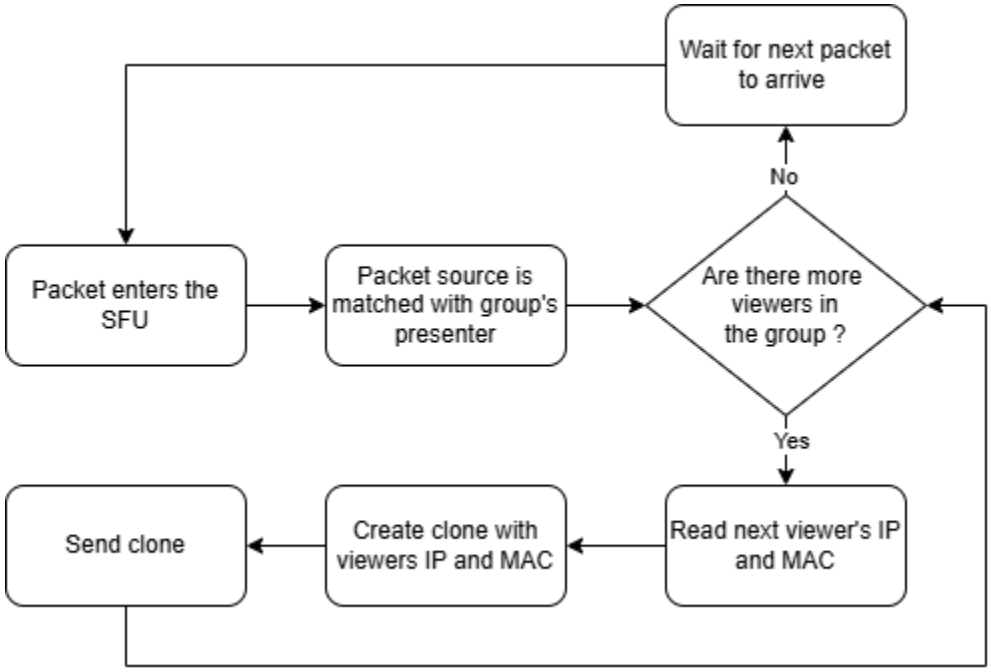

Figure 20 : user space SFU flowchart

3. Experimental setup

3.1. Environment configuration

The experiments were executed on a virtual machine running on Oracle VirtualBox 7.1 with Ubuntu 24.04.2 LTS installed. The virtual machine already had most of the necessary tools needed to use P4 preinstalled [16]; all that was needed was to execute the *./p4-guide/bin/install-p4dev-v8.sh* installation script. The project was developed and ran in the tutorial folder already installed in the virtual machine. In this manner, the configuration for the automated network generation for these tutorials could be used for the development and testing of the project.

The virtual network was simulated using the network emulator Mininet [17]. For traffic generation a Python script was used to create the dummy packets needed and write them in a .pcap trace file. Then, the script used the Linux command line to call tcpreplay, an open-source suite of tools used for editing or sending packets recorded in .pcap trace files [18], to transmit the packets it had created to the network at the rate required for the experiments. It should be noted that in some earlier iterations of the setup we used Python's Scapy library to transmit the packets, but Scapy could not achieve some of the higher bitrates required for the experiments. In addition, it also lacked the desired accuracy when it came to sending the packets at regular time intervals. To execute the Python scripts on the hosts created by Mininet, including the send script, the user level SFU and a script used to record any packets the host received, *xterm* was used to access the desired host's command line.

To record the traffic sent towards and generated by the two SFUs, a shell script was created which was ran on the host virtual machine; it started Wireshark captures at the points of the network where required. After its termination, it stored the recordings at a predetermined location. To analyze the recorded data a Python script was created which read the contents of the .pcap files created by Wireshark and in turn generated two CSV files and a graph based on the recordings. More details on the contents of these files are in the "3.3. Measurement and analysis method" section.

3.2. Network and application settings

The goal of the experiments was to attempt to simulate a video conferencing application. For that purpose, in both the P4-based SFU and the user space SFU experiments, the network was arranged in a star-like topology with an Open vSwitch (OVS) instance at its center. OVS, a non-programmable virtual switch [19], facilitated the communication between the hosts and the SFU by forwarding any packets it

received to their designated destination. This switch was connected with each host using a different port.

In the case of the P4-Based SFU, the SFU ran on a second switch which maintained a single connection with the central switch, as shown in Figure 21. This switch was a BMv2 programmable software switch that could execute P4 programs. It should be noted that unlike traditional Layer-2 switches, the P4-Based SFU switch was configured with an IP address on its ingress interface. This allowed hosts to treat the switch as an IP endpoint and send packets directly to it.

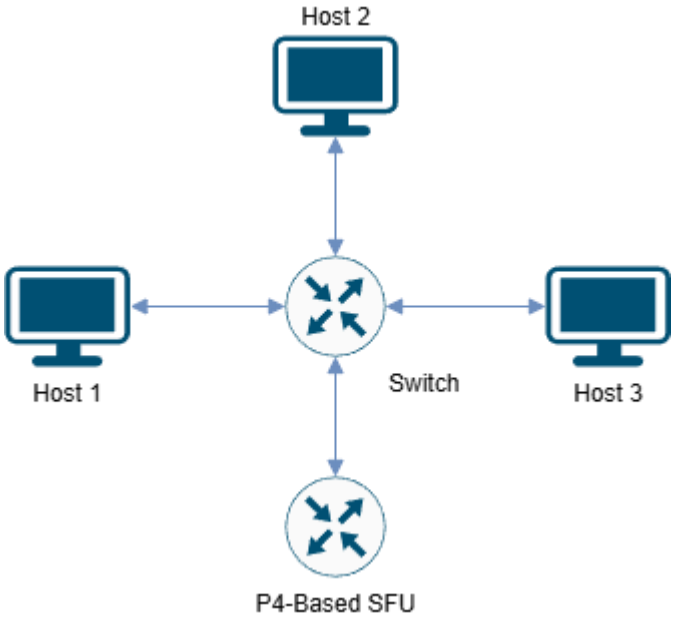

Figure 21: Network topology for experiments with the P4-Based SFU

The user space SFU ran on a Linux host and also maintained a single connection with the central switch, leading to a topology identical to that of the P4-Based SFU experiments, as shown in Figure 22.

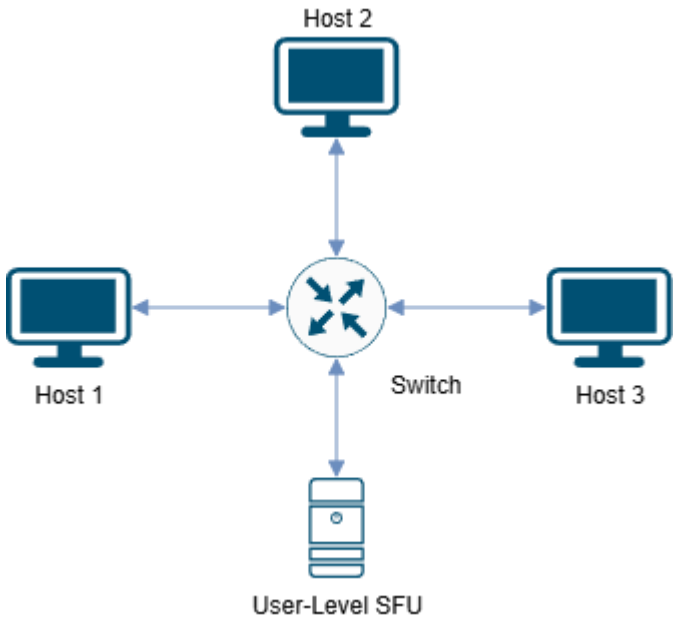

Figure 22 : Network topology for experiments with the user space SFU

All packets transmitted during the experiments were UDP packets with a total size of 1400 bytes. This size is less than the Ethernet Maximum Transmission Unit (MTU) of 1500 bytes [20], but it was large enough to minimize the number of packets required to achieve a specific bitrate and, by extension, the strain on the system that results from small transmission intervals. It was also selected to more accurately reflect the relatively large size of video packets. The payload of each packet included the name of the host which transmitted it, a sequence number indicating its position in the burst and, finally, enough padding to get it to the desired size of 1400 bytes.

Preliminary testing showed that the delay of the P4-Based SFU, for a given bitrate and number of copies, started stabilizing when testing with bursts of around 50 packets. Despite that, it was decided that bursts of 300 packets should be used to minimize the effects of potential spikes in delay time due to unrelated processes running on the host virtual machine, like the Wireshark recordings. All experiments were repeated three times; in each run, we sent a burst of 300 packets to the SFU, for a given number of participating hosts. The SFU then transmitted copies of those packets to the specified number of hosts, beginning with one host and increasing the number of hosts by one in each subsequent set of three runs. Each experiment was ran for both the P4-Based SFU and the Python (user space) SFU.

Initially, we attempted to use source bitrates of 0.15 Mbps, 2.5 Mbps and 4 Mbps, since those are the minimum, recommended and best performance bitrates for the Microsoft Teams Application, respectively [21]. However, as the user space SFU relied on Python's limited ability to handle relatively fast bitrates, the tests resulting in large amounts of packets being lost, so we added experiments with the slower bitrate of 0.076 Mbps. 0.076 is the bitrate recommended for best performance by Microsoft Teams for audio only meetings, so while not totally unrealistic, since this project is mostly concerned with video calls, it is not ideal either.

3.3. Measurement methodology

The primary objective of the experiments was to compare the performance of the two SFU implementations when it came to delay for a given number of hosts, the scaling behavior of delay when increasing the number of hosts and the maximum number of hosts they could be supported for a given bitrate.

To record the packets in our experiments, a shell script was used that started four Wireshark instances that captured packets for each 300-packet run and stored them in .pcap files. For the P4-Based SFU, Wireshark captured all UDP packets arriving at the interface used by the SFU to receive packets from the central switch with the P4-Based switch as their destination, and all packets leaving from the SFU towards the central switch with destination IPs other than that of the SFU. For the user space SFU, both Wireshark instances recorded traffic in the central switch's interface that was connected to the host the SFU was running on. This setup was due to the fact that in a realistic setting it would be necessary for the user space SFU to always have a switch between itself and the rest of the network; to make the two topologies comparable, even though the P4-Based SFU could run on the central switch, we decided to run it in a separate switch connected to the central switch. One Wireshark instance recorded incoming packets, that is, packets with the SFU's IP as the destination, and another one outgoing packets, with any IP destinations except the SFU's host.

To analyze the results, a Python script was created that read all .pcap files of an experiment, matching each original packet received by the SFU with its copies transmitted form the SFU, based on the packet's unique payload consisting of the source host, its sequence number and padding for example h1001[padding to reach 1400 bytes] and produced a number of metrics about them. It should be noted that while for each 300-packet run both the P4-Based SFU and the user space SFU related Wireshark captures were running, for practical purposes, each experiment was ran

only for one of the SFUs at a time, so in practice while four .pcap files were produced, only two contained packets in them with the other two being empty.

The results of the analysis of each experiment produced a graph showing the relation between the average delay and the number of hosts, a CSV file in which each row represented a 300-packet run and included data about the bitrate the experiment was run with, average packet size to confirm it was 1400 Bytes, the number of viewers in the run, the duration of the trial (calculated as the difference between the arrival time of the first and the last packets sent to the SFU), the minimum delay to process a packet, the maximum delay to process a packet, the average delay to process a packet, the standard deviation between the delays of all packets in the run, the number of packets sent to the SFU, the number of packets transmitted by the SFU, packets lost in, meaning packets that where sent to the SFU but did not get replicated, packets lost out, meaning packets that where transmitted by the SFU but did not match an original packet, volume in Kb received by the SFU, volume in Kb transmitted by the SFU, average delay compared to the previous trial of the same experiment (to check the consistency between identical trials) and, finally, the true bitrate of the trial, calculated by dividing the volume of data received by the SFU with the duration of the trial; this was done since some early experiments failed to achieve the desired bitrate and modifications to the way packets were sent to the SFU where needed.

A second CSV file is created by the Python script, summarizing the results per number of hosts. This included data about the number of viewers this row contains data about, the average of all average delays for this number of viewers, the average of average delays compared to the previous number of receiving hosts, a count of the number of runs made for this number of hosts to make sure it was three, as planned, the standard deviation of the delay across the runs for each number of hosts as a measure of stability, plus the average packets lost in and out for this number of viewers.

4. Evaluation Results

This section presents the results of the experiments, grouped by the bitrate used in said experiments, in the order that they were conducted. Then, general observations and a comparison between the performances of the two SFUs are presented. For brevity, only the tables from the summary CSVs are included in this section; the extended results can be found in the appendix.

4.1. Bitrate of 4 Mbps

P4-Based SFU results

Table 1: Summary results for P4-Based SFU at 4 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.001013	0.007868	0.001692		0.000213	0
2	0.001782	0.018843	0.002945	74.11%	0.000473	0
3	0.003347	0.025315	0.006555	122.54%	0.001996	0
4	0.004434	0.161159	0.044819	583.78%	0.009311	0
5	0.006676	0.299062	0.141954	216.72%	0.015069	0
6	0.007707	0.610527	0.256515	80.70%	0.019832	0
7	0.008008	0.750984	0.353978	38.00%	0.013223	0
8	0.011734	1.066183	0.486725	37.50%	0.015256	0
9	0.012599	1.184818	0.579637	19.09%	0.031377	0
10	0.012622	1.473549	0.702944	21.27%	0.029031	0
11	0.01472	1.804093	0.854824	21.61%	0.033296	0

In the first experiment, the P4-Based SFU was tested with packets being sent with a bitrate of 4 Mbps. Up to the third viewer, the added latency remains very low; as we can clearly see in Figure 23, it scales almost completely linearly with the number of viewers. With the addition of a fourth host the delay starts to increase nonlinearly, reaching around 45ms, and by the fifth viewer it has risen dramatically to slightly above 140ms, with similar increases following the addition of further viewers. From this, we can easily calculate that the SFU with the current setup can transmit with a rate of around 16Mbps (4 viewers * 4 Mbps), while keeping the delay increase

linear. It is interesting to note that the later increases appear to follow a new linear scale. When it comes to the standard deviation between runs with the same number of viewers, it remains consistently low, indicating the relative stability of the SFU, though it also begins rising after the addition of the fourth viewer. It is also important to note that even with the maximum number of eleven viewers, there was no packet loss observed. These results align with expected bottleneck dynamics in programmable switches: once queues saturate, delay dominates while throughput remains intact.

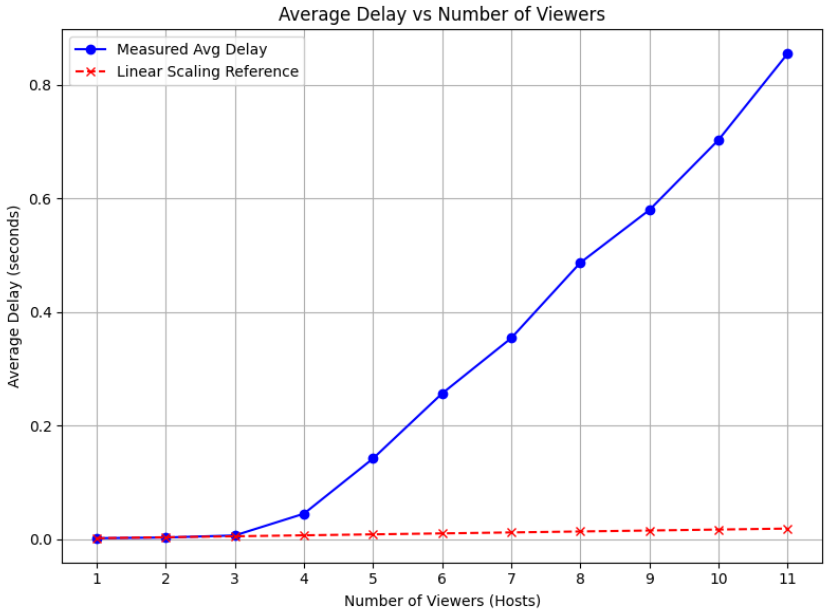

Figure 23 : Average delay vs number of viewers graph for the P4-Based SFU at 4 Mbps

user space SFU results

Table 2 : Summary results for user space SFU at 4 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.012521	2.170998	1.037004	-	0.081095	210.33

When testing the user space SFU, it quicky became obvious that it could not handle 4 Mbps, with more than a third of the packets sent to it producing no clones

and being lost even with a single user. As a result, no further testing was conducted at this bitrate.

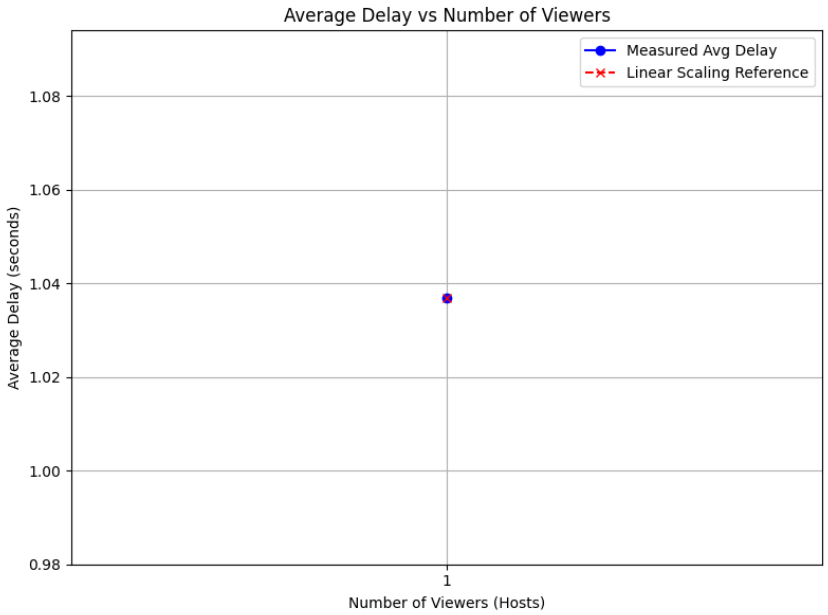

Figure 24 : Average delay vs number of viewers graph for the user space SFU at 4 Mbps

4.2. Bitrate of 2.5 Mbps

P4-Based SFU results

Table 3 : Summary results for P4-Based SFU at 2.5 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.000977	0.00899	0.00173		0.000112	0
2	0.001842	0.020609	0.003032	75.28%	0.000344	0
3	0.002798	0.023963	0.005009	65.18%	0.00081	0
4	0.003808	0.01526	0.005586	11.53%	0.000191	0
5	0.005408	0.045238	0.008742	56.49%	0.00129	0
6	0.007504	0.02831	0.014479	65.63%	0.003086	0
7	0.009179	0.319847	0.142304	882.83%	0.040563	0
8	0.008522	0.411645	0.194082	36.39%	0.018613	0

9	0.009325	0.754314	0.348544	79.59%	0.017672	0
10	0.012828	0.920747	0.446259	28.03%	0.030696	0
11	0.010544	1.161713	0.582184	30.46%	0.028054	0

When experimenting with the P4-Based SFU at a bitrate of 2.5Mbps, the behavior observed is similar to when using a bitrate of 4Mbps. This time the average delay remains low and increases linearly up to six viewers, where it reaches 14.5ms. After that point, there is a major increase in both the rate at which the delay increases, as well as the standard deviation, indicating increased instability. The maximum bitrate at which the SFU can transmit comfortably is 15Mbps (2.5 Mbps x 6 viewers), close to the 16Mbps observed in the previous test run. Once again, no packet loss was observed during the experiment.

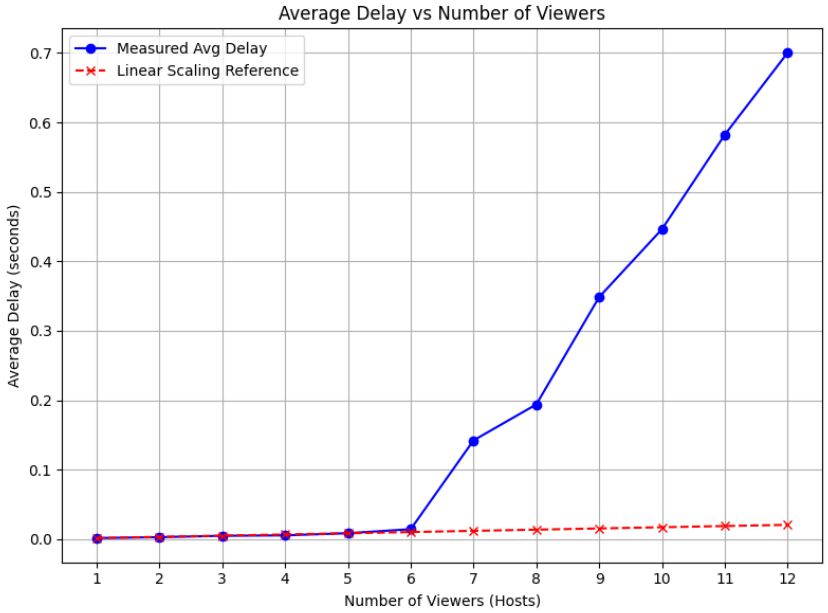

Figure 25 : Average delay vs number of viewers graph for the P4-Based SFU at 2.5 Mbps

user space SFU results

Table 4 : Summary results for user space SFU at 2.5 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost

1	0.014899	2.137533	1.029144	-	0.094613	185

As in the experiment using a 4Mbps bitrate, major packet loss is observed already from the very first viewer, with more than half of the packets being lost and extremely high latency, taking more than a second for each packet to be cloned.

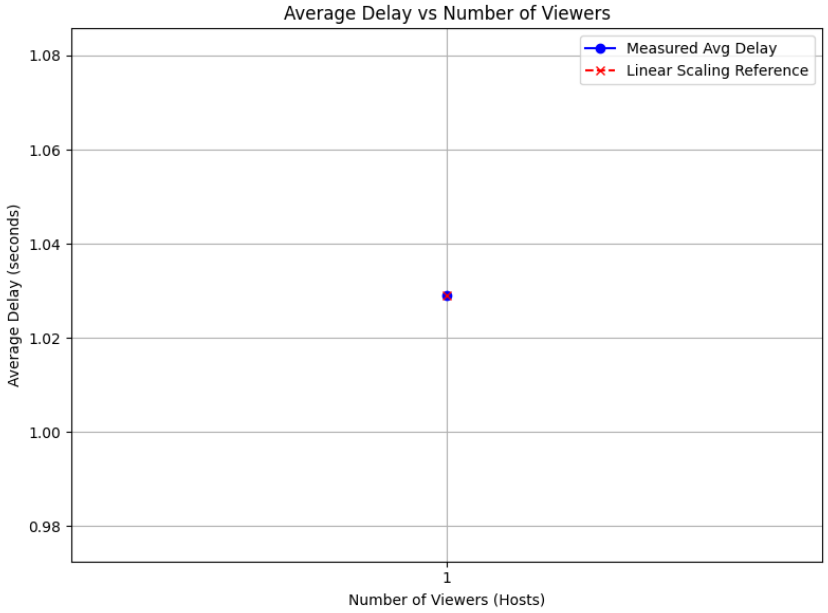

Figure 26 : Average delay vs number of viewers graph for the user space SFU at 2.5 Mbps

4.3. Bitrate of 0.15 Mbps

P4-Based SFU results

Table 5 Summary results for P4-Based SFU at 0.15 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.000964	0.015707	0.001896		0.000255	0
2	0.001719	0.010157	0.002971	56.72%	0.000232	0
3	0.002554	0.018129	0.004402	48.14%	0.000268	0
4	0.003417	0.028455	0.005436	23.49%	0.000806	0
5	0.004075	0.025499	0.008154	50.01%	0.000557	0
6	0.004998	0.0241	0.009864	20.98%	0.000424	0
7	0.005958	0.017037	0.00872	-11.60%	0.000764	0

8	0.006463	0.025187	0.011216	28.63%	0.002905	0
9	0.008239	0.033649	0.013401	19.48%	0.000477	0
10	0.008776	0.040953	0.014568	8.71%	0.002072	0
11	0.009499	0.029708	0.013122	-9.93%	0.002606	0
12	0.010649	0.040122	0.015455	17.78%	0.00166	0
13	0.011838	0.056409	0.018733	21.21%	0.000829	0
14	0.012522	0.055912	0.018242	-2.62%	0.002147	0
15	0.011456	0.036355	0.020823	14.15%	0.002889	0
16	0.012733	0.049815	0.019491	-6.40%	0.002791	0
17	0.014798	0.053912	0.023127	18.65%	0.001318	0
18	0.014973	0.044334	0.01896	-18.02%	0.001654	0
19	0.018484	0.054291	0.025165	32.73%	0.001099	0

When using a bitrate of 0.15 Mbps, the P4-Based SFU once more maintains zero packet loss. Its latency increases in a nearly linear fashion and for the duration of this experiment there was no sharp increase, as with the bitrates of 2.5Mbps and 4Mbps. Since in the previous experiments conducted with the P4-Based SFU the sharp increase happened at the point when the SFU transmitted with around 15Mbps that would mean that for 0.15Mbps it would take around one hundred viewers to reach that point, which the virtual machine host would probably not be able to support, so it was decided to stop the experiment at the nineteenth viewer.

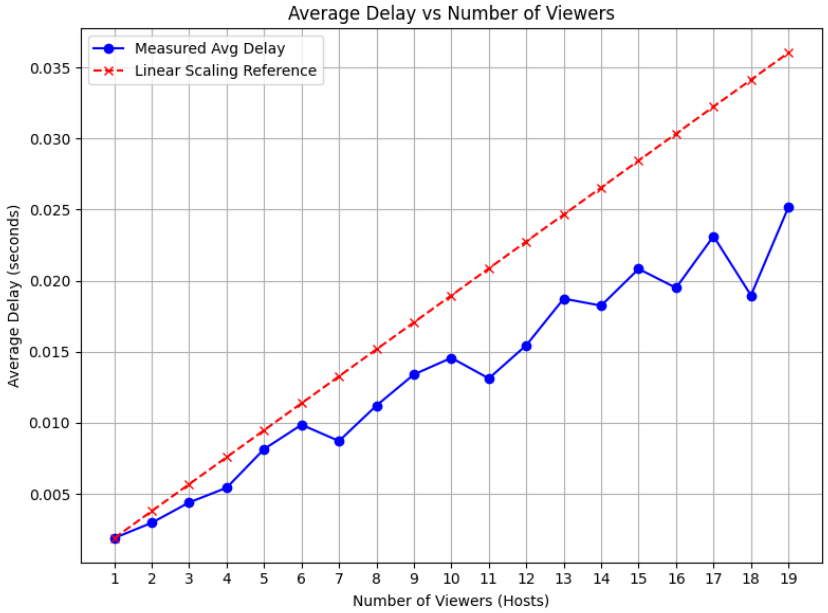

Figure 27: Average delay vs number of viewers graph for the P4-Based SFU at 0.15 Mbps

user space SFU results

Table 6 : Summary results for user space SFU at 0.15 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.005123	1.28837	0.054924	-	0.053063	0
2	0.023196	1.509373	0.190789	247.37%	0.213357	0
3	0.034977	2.024044	0.516701	170.82%	0.338272	7.67
4	0.065941	10.078093	3.026833	485.80%	1.806817	118.33
5	0.106272	12.109511	3.98331	31.60%	1.843851	168

When sending a stream with a bitrate of 0.15Mbps to the user space SFU, latency initially increases relatively linearly for only the first two hosts, with a noticeable increase from 191ms to 517ms at the third. When just one more host is added then the average delay skyrockets to more than 3 seconds. As far as packet loss is concerned for the first two hosts there is none, and at viewer number three it's 2.5%, just above the maximum acceptable in applications like Zoom [22]. From the fourth viewer and onwards, packet loss becomes clearly unacceptable with more than a third of the packets being lost. The standard deviation of delay between runs also increases significantly at that point.

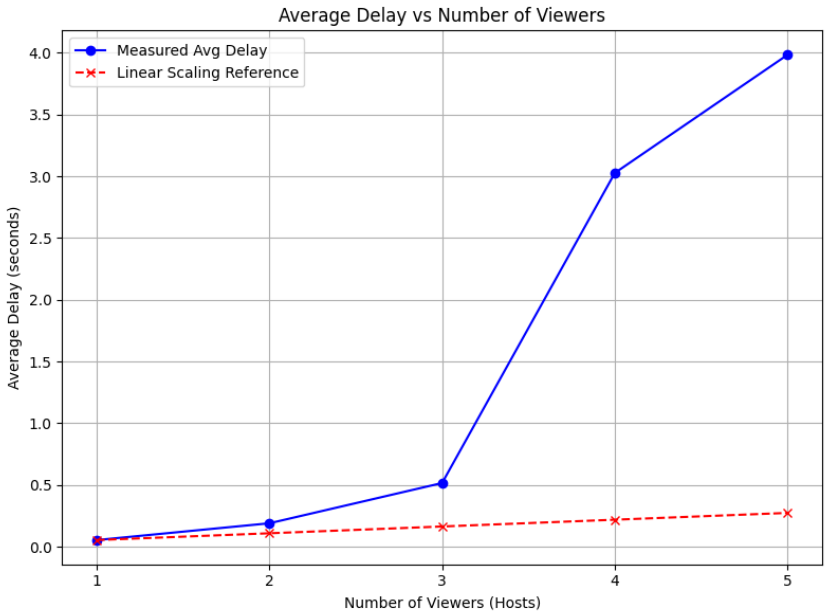

Figure 28 : Average delay vs number of viewers graph for the user space SFU at 0.15 Mbps

4.4. Bitrate of 0.076 Mbps

P4-Based SFU results

Table 7 : Summary results for P4-Based SFU at 0.076 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.00093	0.023949	0.00181	-	0.000269	0
2	0.001754	0.021029	0.003233	78.64%	0.000346	0
3	0.002545	0.026238	0.00399	23.40%	0.00027	0
4	0.003376	0.023085	0.00617	54.64%	0.000482	0
5	0.004112	0.033264	0.006438	4.35%	0.000767	0
6	0.005401	0.020137	0.007918	22.99%	0.000948	0
7	0.005733	0.027613	0.008599	8.60%	8.70E-05	0

Once again, with a bitrate 0.076Mbps the average delay for the P4-Based SFU scales relatively linearly. This experiment was ended at seven viewers to match the

corresponding user space SFU experiment, since it was not part of the original set of experiments and was added just to provide more data for comparing the two SFUs.

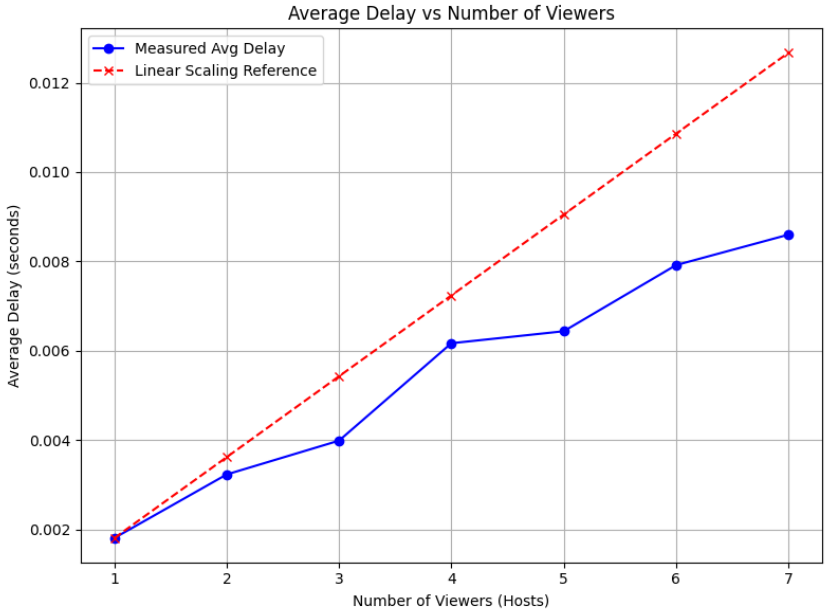

Figure 29 : Average delay vs number of viewers graph for the P4-Based SFU at 0.076 Mbps

user space SFU results

Table 8 : Summary results for user space SFU at 0.076 Mbps

Number of viewers	Minimum delay in seconds	Maximum Delay in seconds	Average delay in seconds	Average delay compared to previous number of viewers	Standard deviation of delay between runs for the given number of viewers	Average number of packets lost
1	0.005578	1.041137	0.061121		0.053466	0
2	0.024309	2.071009	0.154619	152.97%	0.116966	0
3	0.037442	4.173761	0.320757	107.45%	0.34386	4.67
4	0.060191	7.401644	1.405437	338.16%	0.55846	32.33
5	0.072576	4.772319	1.162259	-17.30%	0.970617	29
6	0.107365	14.352738	3.816721	228.39%	2.388305	98.67
7	0.113907	6.157235	1.997344	-47.67%	0.875714	78.33

The user space SFU can deal with up to three viewers at a bitrate of 0.076Mbps with the delay scaling linearly and small or non-existent packet loss. However, both delay and packet loss increase by a large margin with the addition of a fourth host. The most striking result of this experiment is probably the large degree of instability that can be observed in the results both in the relatively high standard deviation as well as the fact that from the fourth host and onwards the average delay does not maintain its upwards trend, instead having noticeable increases and decreases in its average delay.

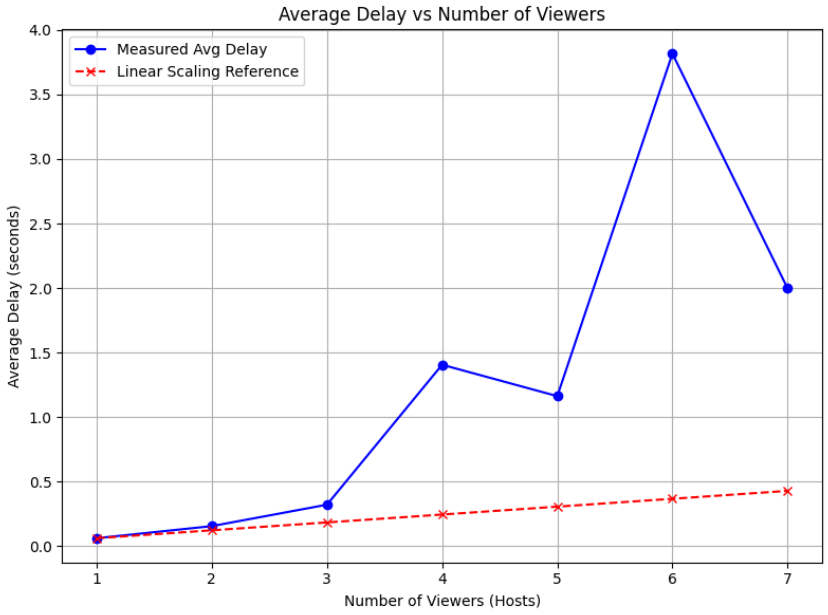

Figure 30 Average delay vs number of viewers graph for the user space SFU at 0.076 Mbps

4.5. General observations and comparison

Comparing the two SFUs it becomes clear that the P4-Based implementation has a distinct advantage in all metrics. When it comes to lower latency, which was the main objective of this project, the P4-based SFU is much faster in all cases, with delays consistently in the 2ms to 20ms range in most cases. As the number of viewers increases, the latency increases in a mostly linear fashion until the SFU has to transmit at a rate of more than 15Mbps. It can be relatively safely assumed that if this implementation used real hardware or software running on better hardware than what we had available, this ceiling would be higher. On the other hand, the user space SFU is much slower with delays extending to several seconds with the P4-based SFU being from 95% to 100% faster. The scalability of the user space implementation also leaves much to be desired, with delays increasing at a non-linear rate even at low viewer counts.

Table 9 : Comparison of P4-based SFU and user space SFU at 0.15Mbps

Number of viewers	Average delay in seconds for P4-based SFU	Average delay in seconds for user space SFU	P4-based Faster (%)
1	0.001896	0.054924	96.54795718
2	0.002971	0.190789	98.44278234
3	0.004402	0.516701	99.14805661
4	0.005436	3.026833	99.82040635
5	0.008154	3.98331	99.79529587

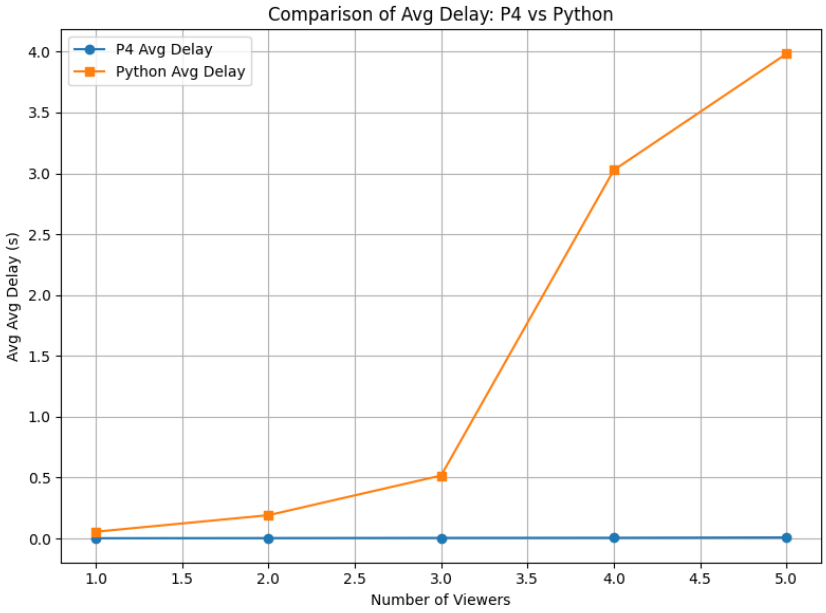

Figure 31 : Comparison graph of P4-based SFU and user space SFU at 0.15Mbps

Table 10 : Comparison of P4-based SFU and user space SFU at 0.0.76Mbps

Number of viewers	Average delay in seconds for P4-based SFU	Average delay in seconds for user space SFU	P4-based Faster (%)
1	0.00181	0.061121	97.03866102
2	0.003233	0.154619	97.90905387
3	0.00399	0.320757	98.75606768
4	0.00617	1.405437	99.56099064
5	0.006438	1.162259	99.44607871
6	0.007918	3.816721	99.79254444
7	0.008599	1.997344	99.56947827

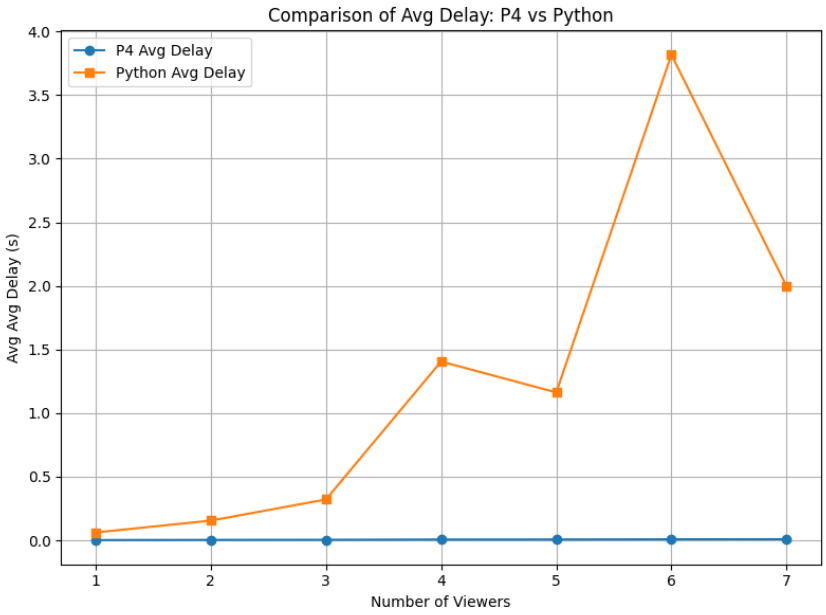

Figure 32 : Comparison graph of P4-based SFU and user space SFU at 0.0.76Mbps

It can also be observed that the P4-based SFU is significantly more stable, with usually low standard deviation, even when delay begins to rise. This is to be expected since the user space SFU must deal with overhead from user space scheduling, garbage collection etc.

When it comes to packet loss, in all tests the P4-based SFU caused no loss of packets, unlike its counterpart which experienced significant packet loss, especially when having to deal with higher bitrates, where the packet loss was so severe that the user space SFU became unusable.

Finaly, when comparing this project to the work “A Selective Forwarding Unit Implementation in P4” [13] which was the main inspiration behind this project, although the SFU in that work relies on multicast replication inside the switch pipeline, an approach that is theoretically ideal for minimizing overhead, its evaluation was conducted under extremely low traffic rates. The authors used packets of only 352–376 bits and transmission intervals of 36–62 ms, resulting in effective bitrates of approximately 6–9 kbps per stream. In contrast, the SFU evaluated in this work operates under substantially higher traffic loads, with per-viewer bitrates an order of magnitude greater. Despite this heavier load, the P4-based SFU developed here maintains low delay, predictable scaling, and zero packet loss across a wide range of viewer counts. This suggests that while multicast replication is conceptually optimal, the practical performance of a P4 SFU under realistic traffic conditions depends more on pipeline design and efficiency than on

the specific replication mechanism used. Consequently, the results of this work complement the findings of the original paper by demonstrating that a P4-based SFU can remain stable and performant, even when subjected to significantly higher bitrates than those used in prior evaluations.

5. Conclusions and future research

During this project, two SFUs were developed: one implemented in P4 and executed on a programmable switch, and one implemented in Python and executed in user space. The evaluation demonstrated clear advantages for the P4-based SFU, which consistently achieved lower latency and exhibited significantly better scalability as the number of viewers increased.

However, some limitations must be acknowledged. Due to the system being emulated, its performance differed compared to testing on real hardware. BMv2's performance is significantly lower than hardware programable switches, while at the same time both Mininet and the use of a virtual machine likely introduced some timing inaccuracies when running the experiments. The user space SFU also had some inherent disadvantages built into it, since it was not intended to be as optimized as possible, but to instead mirror its P4-based counterpart.

The natural next step would be to run the tests on different machines with the P4-based SFU on a P4 compatible hardware switch to evaluate the real word performance benefits of a switch based SFU developed in P4. Additionally, while the P4-based SFU presented had the ability to be reconfigured dynamically, this was done only through the machine it was ran on and not with the use of signaling. In theory at least, it would be relatively simple to use the full capabilities of the P4Runtime API to enable it to receive signals from the hosts connected and run the appropriate controller scripts to respond to their requests. Such an extension would bring the SFU a step closer to being a realistic production ready system. Another additional step to increase realism would be adding multiple streams of different qualities being transmitted by each host.

6. Appendix: Extended results

Abbreviations:

Pkt Size: Packet size

Viewers: Number of viewers

Dur: Duration

Min Dly: Minimum Delay

Max Dly: Maximum Delay

Avg Dly: Average Delay

Std Dly: Standard deviation of delay in this run

In Pkts: Number of packets that entered the SFU

Out Pkts: Number of packets that exited the SFU

Lost: Number of packets that entered the SFU but whose content did not much any packets that exited (they were not copied)

Vol In (KB): Total volume of packets that entered the SFU in Kilobytes

Vol Out (KB): Total volume of packets that exited the SFU in Kilobytes

ΔAvg Dly vs Prev: Average delay compared to the previous run

True BR: True bitrate

Table 11 : Extended results for P4-based SFU at 4Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	0.840071	0.001126	0.005749	0.001992	0.000761	300	300	0	3360	3360		3999.662
1400	1	0.840039	0.001013	0.007868	0.001525	0.000625	300	300	0	3360	3360	-23%	3999.814
1400	1	0.839808	0.001014	0.005361	0.001558	0.000482	300	300	0	3360	3360	2%	4000.914
1400	2	0.839873	0.001899	0.007724	0.003033	0.000742	300	600	0	3360	6720	95%	4000.605
1400	2	0.83989	0.001782	0.0057	0.002327	0.000488	300	600	0	3360	6720	-23%	4000.524
1400	2	0.840075	0.001953	0.018843	0.003476	0.002089	300	600	0	3360	6720	49%	3999.643
1400	3	0.839416	0.003662	0.00881	0.004904	0.001028	300	900	0	3360	10080	41%	4002.783
1400	3	0.839988	0.003575	0.025315	0.009363	0.005057	300	900	0	3360	10080	91%	4000.057
1400	3	0.83989	0.003347	0.020565	0.005397	0.002357	300	900	0	3360	10080	-42%	4000.524

1400	4	0.839522	0.005637	0.091801	0.052478	0.028233	300	1200	0	3360	13440	872%	4002.277
1400	4	0.840005	0.005102	0.062977	0.031714	0.011654	300	1200	0	3360	13440	-40%	3999.976
1400	4	0.840002	0.004434	0.161159	0.050266	0.041674	300	1200	0	3360	13440	58%	3999.99
1400	5	0.839764	0.009125	0.299062	0.160502	0.081817	300	1500	0	3360	16800	219%	4001.124
1400	5	0.839759	0.006813	0.280549	0.141767	0.074852	300	1500	0	3360	16800	-12%	4001.148
1400	5	0.839833	0.006676	0.256672	0.123592	0.071931	300	1500	0	3360	16800	-13%	4000.795
1400	6	0.839645	0.009633	0.478654	0.244178	0.141494	300	1800	0	3360	20160	98%	4001.691
1400	6	0.839985	0.008251	0.461733	0.24087	0.13393	300	1800	0	3360	20160	-1%	4000.071
1400	6	0.840004	0.007707	0.610527	0.284496	0.172007	300	1800	0	3360	20160	18%	3999.981
1400	7	0.839803	0.008008	0.750984	0.366745	0.215756	300	2100	0	3360	23520	29%	4000.938
1400	7	0.839951	0.009536	0.656126	0.335761	0.188818	300	2100	0	3360	23520	-8%	4000.233
1400	7	0.839559	0.010009	0.696611	0.359427	0.202036	300	2100	0	3360	23520	7%	4002.101
1400	8	0.839856	0.011768	0.928352	0.471629	0.266175	300	2400	0	3360	26880	31%	4000.686
1400	8	0.839942	0.011825	1.066183	0.507622	0.308019	300	2400	0	3360	26880	8%	4000.276
1400	8	0.83995	0.011734	0.907618	0.480925	0.255181	300	2400	0	3360	26880	-5%	4000.238
1400	9	0.839882	0.012903	1.088686	0.559527	0.317864	300	2700	0	3360	30240	16%	4000.562
1400	9	0.840127	0.016214	1.184818	0.623948	0.335394	300	2700	0	3360	30240	12%	3999.395
1400	9	0.840036	0.012599	1.150188	0.555436	0.324165	300	2700	0	3360	30240	-11%	3999.829
1400	10	0.839429	0.013649	1.42612	0.738477	0.412874	300	3000	0	3360	33600	33%	4002.721
1400	10	0.83936	0.013587	1.473549	0.70299	0.418946	300	3000	0	3360	33600	-5%	4003.05
1400	10	0.840008	0.012622	1.351014	0.667366	0.38786	300	3000	0	3360	33600	-5%	3999.962
1400	11	0.839805	0.01472	1.690892	0.842292	0.478328	300	3300	0	3360	36960	26%	4000.929
1400	11	0.839989	0.01791	1.804093	0.900398	0.524497	300	3300	0	3360	36960	7%	4000.052
1400	11	0.839992	0.01616	1.616436	0.821781	0.462946	300	3300	0	3360	36960	-9%	4000.038

Table 12 : Extended results for user space SFU at 4 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	0.833683	0.017725	1.432714	0.931715	0.483148	300	88	212	3360	985.6		4030.309
1400	1	0.823529	0.012521	1.958873	1.050276	0.6424	300	91	209	3360	1019.2	13%	4080.002
1400	1	0.839714	0.015385	2.170998	1.129021	0.712034	300	90	210	3360	1008	7%	4001.362

Table 13 : Extended results for P4-based SFU at 2.5 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	1.343813	0.001135	0.00899	0.001816	0.000908	300	300	0	3360	3360		2500.348
1400	1	1.34407	0.001019	0.004263	0.001572	0.000404	300	300	0	3360	3360	-13%	2499.87
1400	1	1.344105	0.000977	0.005838	0.001802	0.000909	300	300	0	3360	3360	15%	2499.805
1400	2	1.343977	0.001882	0.007602	0.00283	0.000613	300	600	0	3360	6720	57%	2500.043
1400	2	1.344294	0.001875	0.003832	0.002751	0.000314	300	600	0	3360	6720	-3%	2499.453

1400	2	1.344138	0.001842	0.020609	0.003516	0.001949	300	600	0	3360	6720	28%	2499.743
1400	3	1.343859	0.002957	0.023963	0.004181	0.00238	300	900	0	3360	10080	19%	2500.262
1400	3	1.344189	0.003218	0.020329	0.004736	0.002394	300	900	0	3360	10080	13%	2499.648
1400	3	1.343988	0.002798	0.022569	0.006109	0.003578	300	900	0	3360	10080	29%	2500.022
1400	4	1.344082	0.003833	0.011537	0.005708	0.001162	300	1200	0	3360	13440	-7%	2499.847
1400	4	1.343776	0.003808	0.01526	0.005316	0.001253	300	1200	0	3360	13440	-7%	2500.417
1400	4	1.344123	0.004028	0.013965	0.005734	0.001179	300	1200	0	3360	13440	8%	2499.771
1400	5	1.34371	0.005408	0.014408	0.007562	0.00153	300	1500	0	3360	16800	32%	2500.54
1400	5	1.343959	0.005628	0.017488	0.008127	0.001814	300	1500	0	3360	16800	7%	2500.076
1400	5	1.343981	0.005452	0.045238	0.010536	0.00683	300	1500	0	3360	16800	30%	2500.035
1400	6	1.343847	0.007583	0.025876	0.014642	0.005074	300	1800	0	3360	20160	39%	2500.285
1400	6	1.3439	0.007504	0.019039	0.010621	0.002581	300	1800	0	3360	20160	-27%	2500.186
1400	6	1.343989	0.008191	0.02831	0.018174	0.004183	300	1800	0	3360	20160	71%	2500.02
1400	7	1.344106	0.009795	0.248936	0.126561	0.069747	300	2100	0	3360	23520	596%	2499.803
1400	7	1.343896	0.009225	0.319847	0.197948	0.082656	300	2100	0	3360	23520	56%	2500.193
1400	7	1.345989	0.009179	0.202395	0.102403	0.051119	300	2100	0	3360	23520	-48%	2496.306
1400	8	1.343551	0.008522	0.411645	0.199867	0.111314	300	2400	0	3360	26880	95%	2500.835
1400	8	1.34395	0.011341	0.351202	0.168951	0.098837	300	2400	0	3360	26880	-15%	2500.093
1400	8	1.344007	0.010203	0.403061	0.213429	0.109685	300	2400	0	3360	26880	26%	2499.987
1400	9	1.343893	0.012035	0.754314	0.373421	0.215317	300	2700	0	3360	30240	75%	2500.199
1400	9	1.344121	0.009325	0.654079	0.338185	0.189101	300	2700	0	3360	30240	-9%	2499.775
1400	9	1.344001	0.0117	0.647287	0.334027	0.179859	300	2700	0	3360	30240	-1%	2499.998
1400	10	1.343783	0.012828	0.84396	0.422706	0.242778	300	3000	0	3360	33600	27%	2500.404
1400	10	1.343705	0.016222	0.920747	0.489616	0.261245	300	3000	0	3360	33600	16%	2500.549
1400	10	1.344053	0.013295	0.83519	0.426454	0.24396	300	3000	0	3360	33600	-13%	2499.901
1400	11	1.344045	0.015992	1.161713	0.613042	0.333556	300	3300	0	3360	36960	44%	2499.916
1400	11	1.344026	0.010544	1.131568	0.588351	0.326693	300	3300	0	3360	36960	-4%	2499.952
1400	11	1.344094	0.015664	1.046544	0.545159	0.294937	300	3300	0	3360	36960	-7%	2499.825

Table 14 : Extended results for user space SFU at 2.5 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	1.330182	0.019233	2.137533	1.15143	0.645071	300	110	190	3360	1232		2525.97
1400	1	1.339242	0.014899	1.931699	1.015033	0.588386	300	121	179	3360	1355.2	-12%	2508.882
1400	1	1.34403	0.020327	1.508451	0.92097	0.452034	300	114	186	3360	1276.8	-9%	2499.944

Table 15 : Extended results for P4-based SFU at 0.15 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	22.400506	0.001029	0.015707	0.001585	0.000927	300	300	0	3360	3360		149.997
1400	1	22.399612	0.001031	0.00492	0.001893	0.000724	300	300	0	3360	3360	19%	150.003

1400	1	22.399861	0.000964	0.005231	0.00221	0.000655	300	300	0	3360	3360	17%	150.001
1400	2	22.39976	0.001927	0.007938	0.003156	0.000858	300	600	0	3360	6720	43%	150.002
1400	2	22.40057	0.001719	0.010157	0.002644	0.000702	300	600	0	3360	6720	-16%	149.996
1400	2	22.400514	0.001782	0.009993	0.003114	0.001094	300	600	0	3360	6720	18%	149.997
1400	3	22.399682	0.003073	0.014717	0.004684	0.001243	300	900	0	3360	10080	50%	150.002
1400	3	22.400249	0.002554	0.018129	0.004042	0.001197	300	900	0	3360	10080	-14%	149.998
1400	3	22.40024	0.002804	0.010309	0.004479	0.001188	300	900	0	3360	10080	11%	149.998
1400	4	22.39976	0.004294	0.00984	0.005565	0.001062	300	1200	0	3360	13440	24%	150.002
1400	4	22.400215	0.003754	0.028455	0.006352	0.00196	300	1200	0	3360	13440	14%	149.999
1400	4	22.400494	0.003417	0.011303	0.00439	0.000963	300	1200	0	3360	13440	-31%	149.997
1400	5	22.400057	0.005825	0.014765	0.00846	0.001638	300	1500	0	3360	16800	93%	150
1400	5	22.400579	0.004376	0.019494	0.007372	0.001775	300	1500	0	3360	16800	-13%	149.996
1400	5	22.40018	0.004075	0.025499	0.00863	0.002207	300	1500	0	3360	16800	17%	149.999
1400	6	22.399738	0.004998	0.016827	0.009739	0.002596	300	1800	0	3360	20160	13%	150.002
1400	6	22.400354	0.00686	0.0241	0.010435	0.002072	300	1800	0	3360	20160	7%	149.998
1400	6	22.4002	0.006818	0.017986	0.009419	0.001994	300	1800	0	3360	20160	-10%	149.999
1400	7	22.39999	0.005958	0.015997	0.009212	0.001231	300	2100	0	3360	23520	-2%	150
1400	7	22.400026	0.00622	0.012608	0.00764	0.00102	300	2100	0	3360	23520	-17%	150
1400	7	22.400707	0.007159	0.017037	0.009307	0.001156	300	2100	0	3360	23520	22%	149.995
1400	8	22.399991	0.006463	0.025187	0.009014	0.001465	300	2400	0	3360	26880	-3%	150
1400	8	22.400142	0.007568	0.023983	0.009314	0.002099	300	2400	0	3360	26880	3%	149.999
1400	8	22.400562	0.009309	0.020557	0.01532	0.001931	300	2400	0	3360	26880	64%	149.996
1400	9	22.40013	0.009987	0.033649	0.012938	0.002747	300	2700	0	3360	30240	-16%	149.999
1400	9	22.400146	0.008239	0.025736	0.014057	0.001398	300	2700	0	3360	30240	9%	149.999
1400	9	22.400089	0.010653	0.025961	0.013207	0.001433	300	2700	0	3360	30240	-6%	149.999
1400	10	22.399376	0.0121	0.030791	0.01675	0.002713	300	3000	0	3360	33600	27%	150.004
1400	10	22.400346	0.008936	0.040953	0.011783	0.002937	300	3000	0	3360	33600	-30%	149.998
1400	10	22.400384	0.008776	0.03326	0.015171	0.003872	300	3000	0	3360	33600	29%	149.997
1400	11	22.399801	0.011837	0.029221	0.016807	0.002678	300	3300	0	3360	36960	11%	150.001
1400	11	22.400286	0.009499	0.029708	0.011299	0.001678	300	3300	0	3360	36960	-33%	149.998
1400	11	22.400307	0.009693	0.018843	0.011259	0.001268	300	3300	0	3360	36960	0%	149.998
1400	12	22.39976	0.012765	0.03196	0.015754	0.002184	300	3600	0	3360	40320	40%	150.002
1400	12	22.400857	0.01326	0.040122	0.017322	0.002853	300	3600	0	3360	40320	10%	149.994
1400	12	22.400519	0.010649	0.036846	0.013288	0.003018	300	3600	0	3360	40320	-23%	149.997
1400	13	22.400112	0.011838	0.035258	0.019487	0.00434	300	3900	0	3360	43680	47%	149.999
1400	13	22.400216	0.013133	0.056409	0.019133	0.004328	300	3900	0	3360	43680	-2%	149.999
1400	13	22.400114	0.014373	0.030406	0.017578	0.002133	300	3900	0	3360	43680	-8%	149.999
1400	14	22.400126	0.012522	0.033831	0.020037	0.003391	300	4200	0	3360	47040	14%	149.999
1400	14	22.400376	0.013133	0.02688	0.015223	0.001636	300	4200	0	3360	47040	-24%	149.997
1400	14	22.400468	0.015909	0.055912	0.019466	0.003297	300	4200	0	3360	47040	28%	149.997
1400	15	22.400328	0.011456	0.036355	0.01676	0.003123	300	4500	0	3360	50400	-14%	149.998
1400	15	22.400077	0.017162	0.034448	0.022485	0.002817	300	4500	0	3360	50400	34%	149.999
1400	15	22.4002	0.0188	0.034235	0.023225	0.002255	300	4500	0	3360	50400	3%	149.999

1400	16	22.400285	0.017274	0.049815	0.021449	0.004298	300	4800	0	3360	53760	-8%	149.998
1400	16	22.40038	0.012733	0.027686	0.015544	0.001722	300	4800	0	3360	53760	-28%	149.997
1400	16	22.400141	0.017838	0.028648	0.021481	0.001865	300	4800	0	3360	53760	38%	149.999
1400	17	22.400764	0.020101	0.053912	0.024739	0.002699	300	5100	0	3360	57120	15%	149.995
1400	17	22.400275	0.014798	0.041645	0.02151	0.003364	300	5100	0	3360	57120	-13%	149.998
1400	17	22.399803	0.018964	0.030999	0.023131	0.002253	300	5100	0	3360	57120	8%	150.001
1400	18	22.399578	0.014973	0.044334	0.021219	0.004416	300	5400	0	3360	60480	-8%	150.003
1400	18	22.399991	0.015136	0.041333	0.017306	0.002223	300	5400	0	3360	60480	-18%	150
1400	18	22.39972	0.016365	0.029889	0.018355	0.001448	300	5400	0	3360	60480	6%	150.002
1400	19	22.400193	0.023391	0.054291	0.026669	0.002211	300	5700	0	3360	63840	45%	149.999
1400	19	22.400452	0.018484	0.035099	0.024751	0.002151	300	5700	0	3360	63840	-7%	149.997
1400	19	22.400587	0.019661	0.04165	0.024074	0.003135	300	5700	0	3360	63840	-3%	149.996

Table 16 : Extended results for user space SFU at 0.15 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	22.39895	0.006369	0.115196	0.017813	0.013591	300	300	0	3360	3360		150.007
1400	1	22.40008	0.006625	1.28837	0.129964	0.291336	300	300	0	3360	3360	630%	149.999
1400	1	22.3992	0.005123	0.125861	0.016994	0.012024	300	300	0	3360	3360	-87%	150.005
1400	2	22.3999	0.023196	0.084656	0.038583	0.007834	300	600	0	3360	6720	127%	150.001
1400	2	22.40018	0.024368	0.13127	0.041267	0.014045	300	600	0	3360	6720	7%	149.999
1400	2	22.40004	0.024231	1.509373	0.492517	0.468573	300	600	0	3360	6720	1093%	150
1400	3	22.39849	0.040572	0.099385	0.059301	0.009889	300	900	0	3360	10080	-88%	150.01
1400	3	22.40019	0.034977	2.024044	0.624032	0.646067	300	858	14	3360	9609.6	952%	149.999
1400	3	22.39976	0.060372	1.470605	0.866771	0.332394	300	873	9	3360	9777.6	39%	150.002
1400	4	22.10087	0.09177	4.668077	2.899843	1.328683	300	624	144	3360	6988.8	235%	152.03
1400	4	22.32499	0.371676	10.07809	5.300483	2.786777	300	424	194	3360	4748.8	83%	150.504
1400	4	22.40009	0.065941	1.351698	0.880172	0.291353	300	1132	17	3360	12678.4	-83%	149.999
1400	5	22.25033	0.204385	12.10951	6.589234	3.493264	300	445	211	3360	4984	649%	151.009
1400	5	22.32449	0.140442	3.544215	2.761273	0.768513	300	745	151	3360	8344	-58%	150.507
1400	5	22.39985	0.106272	4.013508	2.599424	1.07944	300	790	142	3360	8848	-6%	150.001

Table 17 : Extended results for P4-based SFU at 0.076 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	44.21117	0.00093	0.023949	0.002139	0.001757	300	300	0	3360	3360		75.999
1400	1	44.21076	0.000954	0.012079	0.001812	0.001053	300	300	0	3360	3360	-15%	76
1400	1	44.21137	0.000947	0.005219	0.001479	0.00047	300	300	0	3360	3360	-18%	75.999
1400	2	44.21078	0.001754	0.013946	0.003696	0.001878	300	600	0	3360	6720	150%	76
1400	2	44.21107	0.00182	0.021029	0.002865	0.001259	300	600	0	3360	6720	-22%	75.999
1400	2	44.21101	0.001972	0.010475	0.003139	0.000772	300	600	0	3360	6720	10%	75.999

1400	3	44.21059	0.002545	0.018789	0.003614	0.001663	300	900	0	3360	10080	15%	76
1400	3	44.21077	0.002636	0.026238	0.004238	0.001949	300	900	0	3360	10080	17%	76
1400	3	44.2114	0.002622	0.0098	0.004118	0.001082	300	900	0	3360	10080	-3%	75.999
1400	4	44.2108	0.003376	0.010722	0.005629	0.001602	300	1200	0	3360	13440	37%	76
1400	4	44.21179	0.003406	0.023085	0.0068	0.002222	300	1200	0	3360	13440	21%	75.998
1400	4	44.21337	0.003671	0.01672	0.006081	0.001542	300	1200	0	3360	13440	-11%	75.995
1400	5	44.21089	0.004462	0.013618	0.006111	0.001142	300	1500	0	3360	16800	0%	75.999
1400	5	44.21139	0.004112	0.011649	0.005707	0.00089	300	1500	0	3360	16800	-7%	75.999
1400	5	44.21113	0.005651	0.033264	0.007497	0.002113	300	1500	0	3360	16800	31%	75.999
1400	6	44.21111	0.005673	0.019904	0.009105	0.002109	300	1800	0	3360	20160	21%	75.999
1400	6	44.21101	0.005401	0.020137	0.007864	0.002281	300	1800	0	3360	20160	-14%	75.999
1400	6	44.21076	0.005485	0.018895	0.006786	0.000947	300	1800	0	3360	20160	-14%	76
1400	7	44.21114	0.006105	0.016446	0.00872	0.002478	300	2100	0	3360	23520	28%	75.999
1400	7	44.2108	0.005733	0.027613	0.00856	0.002888	300	2100	0	3360	23520	-2%	76
1400	7	44.21094	0.006044	0.014063	0.008518	0.001469	300	2100	0	3360	23520	0%	75.999

Table 18 : Extended results for user space SFU at 0.076 Mbps

Pkt Size	Count	Duration	Min Dly	Max Dly	Avg Dly	Std Dly	In Pkts	Out Pkts	Lost	Vol In (KB)	Vol Out (KB)	ΔAvg Dly vs Prev	True BR
1400	1	44.21075	0.005578	0.156753	0.024095	0.018905	300	300	0	3360	3360		76
1400	1	44.21151	0.007197	1.041137	0.136729	0.209705	300	300	0	3360	3360	467%	75.998
1400	1	44.21134	0.006911	0.178627	0.02254	0.01828	300	300	0	3360	3360	-84%	75.999
1400	2	44.21073	0.025192	2.071009	0.319533	0.534744	300	600	0	3360	6720	1318%	76
1400	2	44.21118	0.024309	0.673604	0.08331	0.077961	300	600	0	3360	6720	-74%	75.999
1400	2	44.21186	0.024322	0.815612	0.061015	0.085271	300	600	0	3360	6720	-27%	75.998
1400	3	44.21073	0.037442	0.237537	0.063082	0.02078	300	900	0	3360	10080	3%	76
1400	3	44.21194	0.039187	4.173761	0.806754	1.058233	300	858	14	3360	9609.6	1179%	75.998
1400	3	44.21086	0.040904	0.720326	0.092436	0.109044	300	900	0	3360	10080	-89%	75.999
1400	4	44.21109	0.074786	7.401644	1.211373	1.904699	300	1032	42	3360	11558.4	1210%	75.999
1400	4	44.21053	0.060191	3.725779	0.839468	0.946395	300	1148	13	3360	12857.6	-31%	76
1400	4	44.21083	0.122301	5.67269	2.165471	1.163101	300	1032	42	3360	11558.4	158%	75.999
1400	5	44.06366	0.382107	4.772319	2.53085	0.884308	300	1115	77	3360	12488	17%	76.253
1400	5	44.21306	0.072576	2.029597	0.386494	0.520847	300	1490	2	3360	16688	-85%	75.996
1400	5	44.21177	0.08645	2.099591	0.569434	0.677838	300	1460	8	3360	16352	47%	75.998
1400	6	44.21115	0.107365	1.127771	0.835818	0.274181	300	1800	0	3360	20160	47%	75.999
1400	6	44.06373	0.140572	10.86721	3.931793	2.944413	300	1008	132	3360	11289.6	370%	76.253
1400	6	44.2108	0.180738	14.35274	6.682551	3.978296	300	816	164	3360	9139.2	70%	76
1400	7	44.06281	0.159478	5.412468	2.598091	1.471099	300	1372	104	3360	15366.4	-61%	76.255
1400	7	44.21106	0.113907	2.561131	0.75908	0.710777	300	1967	19	3360	22030.4	-71%	75.999
1400	7	44.06381	0.238525	6.157235	2.634861	1.16702	300	1316	112	3360	14739.2	247%	76.253

7. References

[1] ITU-T TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, G.114 One-way transmission time, Geneva: INTERNATIONAL TELECOMMUNICATION UNION, 2003.

[2] E. Andre, N. Le Breton, A. Lemesle, L. Roux and A. Gouaillard, "Comparative study of webrtc open source sfus for video conferencing," *2018 Principles, Systems and Applications of IP Telecommunications (IPTComm),* pp. 1-8, 2018.

[3] B. GROZEV, *Efficient and Scalable Video Conferences With Selective Forwarding Units and WebRTC, Ph.D. dissertation,* Strasbourg: Université de Strasbourg, 2019.

[4] J. Wei and S. Bojja Venkatakrishnan, "DecVi: Adaptive Video Conferencing on open peer-to-peer networks," in *Proceedings of the 24th International Conference on Distributed Computing and Networking*, 2023.

[5] M. Willebeek-LeMair, D. Kandlur and Z.-Y. Shae, "On multipoint control units for videoconferencing," in *Proceedings of 19th Conference on Local Computer Networks*, Minneapolis, MN, USA, 1994.

[6] A. Eleftheriadis, M. R. Civanlar and O. Shapiro, "Multipoint videoconferencing with Scalable Video coding," *Journal of Zhejiang University-SCIENCE A,* vol. 7, no. 5, pp. 696-705, 2006.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese and D. Walker, "P4: Programming Protocol-Independent," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 3, pp. 87-95, 2014.

[8] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank and M. Menth, "A survey on Data Plane Programming with P4: Fundamentals, advances, and Applied Research," *Journal of Network and Computer Applications,* vol. 212, 2023.

[9] The P4.org Architecture Working Group, "P416 Portable Switch Architecture (PSA) version 1.2," 22 December 2022. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/PSA-v1.2.pdf. [Accessed 14 December 2025].

[10] P4.org API Working Group, "P4Runtime Specification, Version 1.2.0," P4.org, 6 July 2020. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/p4-spec/p4runtime/v1.2.0/P4Runtime-Spec.pdf. [Accessed 13 December 2025].

[11] The P4 Language Consortium, "P416 Language Specification version 1.2.5," 11 10 2024. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.pdf. [Accessed 17 12 2025].

[12] D. Merling, S. Lindner and M. Menth, "P4-based implementation of bier and bier-FRR for Scalable and Resilient Multicast," *Journal of Network and Computer Applications,* vol. 169, p. 102764, 2020.

[13] P. Tsikrikas and G. Xylomenos, "A Selective Forwarding Unit Implementation in P4," *2024 IEEE Conference on Standards for Communications and Networking (CSCN),* 2024.

[14] A. Bas and A. Fingerhut, "The BMv2 Simple Switch target," 19 January 2021. [Online]. Available: https://github.com/p4lang/behavioral-model/blob/d52ac6257bb3a58606383d03b31ed89671504791/docs/simple_switch.md. [Accessed 17 December 2025].

[15] P. Biondi, "Introduction," The Scapy community, [Online]. Available: https://scapy.readthedocs.io/en/latest/introduction.html. [Accessed 12 December 2015].

[16] A. Fingerhut, "p4 tutorials Git Hub," [Online]. Available: https://github.com/p4lang/tutorials. [Accessed 29 11 2025].

[17] B. Lantz, B. Heller and N. McKeown, "ANetwork in a Laptop: Rapid Prototyping for Software-Defined Networks," *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks,* pp. 1-6, 2010.

[18] F. Klassen, "Tcpreplay - Pcap editing and replaying utilities," appneta.com, [Online]. Available: https://tcpreplay.appneta.com. [Accessed 18 Desember 2025].

[19] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon and M. Casado, "The Design and Implementation of Open vSwitch," in *12th USENIX Symposium on Networked Systems*, Oakland, 2015.

[20] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, *RTP: A transport protocol for real-time applications,* The Internet Society, 2003.

[21] "Prepare your organization's network for Teams - Microsoft Teams | Microsoft Learn," [Online]. Available: https://learn.microsoft.com/en-us/microsoftteams/prepare-network#bandwidth-requirements. [Accessed 29 11 2025].

[22] "Accessing meeting and phone statistics," Zoom, [Online]. Available: https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0070504. [Accessed 1 December 2025].

[23] O. Michel, S. Sengupta, H. Kim, R. Netravali and J. Rexford, "Scalable Video Conferencing using SDN principles," *Proceedings of the ACM SIGCOMM 2025 Conference,* pp. 1213-1231, 2025.