A Selective Forwarding Unit Implementation in P4

Pavlos Tsikrikas and George Xylomenos
Mobile Multimedia Laboratory
Department of Informatics, School of Information Sciences and Technology
Athens University of Economics and Business, Greece
E-mail: tsikrikaspavlos@gmail.com, xgeorge@aueb.gr

***Abstract*—Multi-party conferencing systems rely on a Selective Forwarding Unit (SFU) to replicate and forward media packets between participants, so as to reduce end-to-end delay and improve interactivity. While an SFU does not process these media packets, it still needs to move packets from kernel to user space, replicate them, and push the replicas to the kernel, at the cost of multiple system calls and context switches. To avoid these costs, and even the need for an SFU server, we designed and implemented an SFU based on P4, an open source programming language for network devices. This allows a P4-capable network switch, which is optimized for packet processing, to act as an SFU. We present a simple P4-based SFU implementation and compare its performance against an equally simple, server-based SFU. Even with a virtual (software) switch, the P4-based approach shows dramatic gains in latency and throughout.**

***Index Terms*—P4, Selective Forwarding Unit, Ultra Low Latency.**

I. Introduction

Latency is a crucial issue in conferencing and telepresence applications, as human participants cannot tolerate more than 100-150 ms of end-to-end delay in interpersonal communications. Applications such as *Network Music Performance* (NMP) have even more stringent latency requirements, tolerating mouth-to-ear delays of no more than 30-40 ms [1], making them *Ultra Low Latency* (ULL) applications.

In multiparty conferencing, to avoid having each participant separately exchange media streams with all others, which is clearly not scalable, a server is commonly employed to mediate between participants. In the past, this server was a *Multipoint Conferencing Unit* (MCU), and its task was to receive media from all senders, decode them, mix them into a single media stream, re-encode that stream and duplicate it for each receiver. The decoding, mixing and re-encoding process, however, introduced significant latency.

For this reason, modern conferencing systems rely instead on a *Selective Forwarding Unit* (SFU) [2], which duplicates and forwards only the appropriate media streams to each participant, without any media processing. This is often combined with layered coding, where media (for example, video) are first encoded as a low resolution/quality base layer, with successive enhancement layers offering higher resolution/quality. With layered coding, each participant can generate a set of media layer streams and send them to the SFU; then, each recipient can independently choose which of these streams to receive. For example, one recipient may choose to receive all the video layers from the current speaker and present them in a large window, while another may choose to receive only the base video layers from all participants and compose them into a gallery. Participants only need to inform the SFU of their choices, and the SFU will duplicate and forward only the appropriate packets towards them.

Even though an SFU only looks at packet headers in order to decide how to treat them, without ever touching their content, the SFU process at the server must receive all media packets from the kernel and either drop or duplicate and forward them by sending them back to the kernel, according to what the various recipients have chosen. This means that the latency of an SFU mainly depends on the mechanism that moves packets to/from the network hardware and user space.

In this paper, we present the design and implementation of an SFU using a network switch programmed in P4[1]. P4, short for *Programming Protocol-independent Packet Processors* [3] is an open source programming language for network devices. Using a network switch instead of a server as an SFU has many advantages, including lower capital and operating costs. More importantly though, the switch is optimized for high-speed packet processing: it can exploit hardware capabilities for packet duplication, including multicast, without the costs of system calls and kernel-user space crossings of a server.

While P4 can target diverse network devices and exploit their various hardware capabilities, our prototype implementation uses the *Behavioral Model version 2* (BMv2) reference software switch[2], which runs as a regular process in a server. Despite that, our experimental results indicate that the use of P4 can greatly reduce the processing cost that takes place inside a SFU and, thus, dramatically reduce its latency.

The outline of the remainder of this paper is as follows. In Section II we discuss related work. In Section III, we describe the design of our P4-based SFU, as well as a server-based SFU that we built for comparison purposes. In Section IV we describe our experimental setup, while in Section V we present the results of our experiments. We present our conclusions and discuss future work in Section VI.

II. Related Work

While most commercial conferencing systems use an SFU for multiparty conferencing, their SFU implementations are proprietary. There are however some open-source SFU implementations available. The Ion SFU[3] is a minimal SFU

[1]https://p4.org/
[2]https://github.com/p4lang/behavioral-model
[3]https://github.com/ionorg/ion-sfu

implemented in Go using WebRTC for signaling; it is part of the open-source Ion real time communication platform. The WebRTCSFU[4] is part of a multi-party communication system that also allows the exchange of volumetric video. It is also written in Go and uses WebRTC for signaling. None of these SFUs however attempt to minimize SFU latency, as they are straightforward server-based implementations.

The only work on optimizing SFU latency that we are aware of is our previous SFU prototype that exploits netmap [4], a framework for fast packet I/O, which allows packets to be processed by user-based code without leaving kernel memory [5], [6], so as to reduce the system calls needed to send packet duplicates. While this SFU had a lower latency than a traditional server-based SFU (by 89%) and a lower processing overhead (by 76%), it is still a server-based solution, that requires a dedicated machine to serve as the SFU. An FPGA-based approach has been proposed to reduce the delay at the endpoints, rather than at the SFU [7]. In this work, a dedicated audio processor was created, running on an FPGA chip, whose architecture and instruction set were specifically designed to minimize the latency due to audio sampling.

Another issue with SFUs is their placement in the network: ideally, the SFU should be at the topological center of the participants, to reduce delay. As this is hard to do with many participants spread around the world, an alternative is to use a specially created multicast tree that connects multiple SFUs [8], approximating a single optimally placed SFU.

Since latency depends on both the processing elements (endpoints and SFU) and the networking elements, one way to improve performance is to jointly design the two [9]. In this work, a *Software Defined Network* (SDN) works jointly with the endpoints, first trying to reduce latency by adapting the routes taken, and then reducing the bitrates produced by adapting the coding at the endpoints.

Finally, *Bit Index Explicit Replication* (BIER) is a scheme to implement multicasting inside a network of BIER-aware routers. The BIER concept relies on each packet carrying a bitmap in its header that indicates all the BIER-aware endpoints that the packet should reach. Using unicast routing tables, each router may decide to copy the packet to multiple interfaces, with each packet retaining in its header only the bits for the routers reachable via the corresponding interface. BIER can be simply implemented in P4 by recirculating a packet after making each copy; if the packet needs to be forwarded out of n ports, it is recirculated $n-1$ times. A faster approach is to use the multicast groups in P4-based switches, as shown in [10] for the BMv2. We follow a similar approach for our P4-based SFU, using multicast to avoid packet recirculation, but unlike BIER, our SFU does not need to support arbitrary multicast groups, only the ones configured for a conference.

III. Design & Implementation

A. Design assumptions

To assess the benefits of P4, we decided to design and implement in parallel two very simple SFUs, offering the same feature set: one is a standard, user-space, SFU written in Python and running on a Linux server, while the other is written in P4 and runs on any P4-capable switch; we tested it with the reference software P4 switch, BMv2. The user-space SFU relies on standard socket calls to receive and send packets, with user-level copying of packets when duplication is needed. The P4-based SFU exploits the features provided by P4 to examine, drop, duplicate and forward packets inside a P4-capable network switch; although we tested it with a software switch, the same program can exploit any hardware optimizations present in a real switch.

[4]https://github.com/jvdrhoof/WebRTCSFU

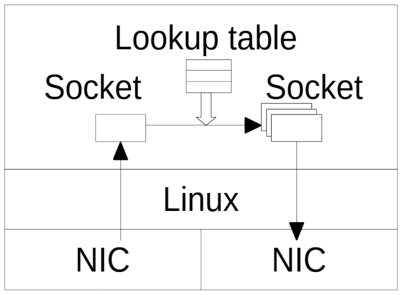

Fig. 1: Outline of our server-based SFU.

To simulate an actual multi-party conferencing application, we created a simple client, written in Python, which transmits and receives one or more streams of UDP packets with dummy content. To simulate the various media streams transmitted by real conferencing clients (audio, base video layer, enhancement video layer, etc.), we used a different UDP destination port for each stream, with a pre-arranged mapping of streams to ports. We also assumed a pre-arranged configuration of hosts, that is, the SFU knew in advance all the participant hosts and had a table showing which recipient host required which stream; a real SFU would employ a signaling protocol to dynamicaly inform the SFU of client preferences.

With these assumptions, the SFU could decide how to treat each incoming packet based only on its IP source address (the sending host) and its destination port (the media stream). For example, recipients that only desired low quality video from everyone, would get all packets sent to the port for the base video layer, while recipients that desired high quality video from a specific user, would get packets sent to the ports for both the base and enhancement layers from that source.

B. Server-based SFU

The server-based SFU simply receives packets from the network, looks at their headers, replicates them for each recipient, and sends the copies back to the network, as shown in Figure 1. Since packet handling is needed, we used Python's `scapy` library. To avoid creating a bottleneck at the network interface that sends and receives packets, we assumed that our server had multiple network interfaces, with a different participant connected to each interface.

The Python program constantly sniffs all available network interfaces. When a packet is caught, a function that handles its processing is called asynchronously. The function examines the packet and checks where is it headed. If it is headed for

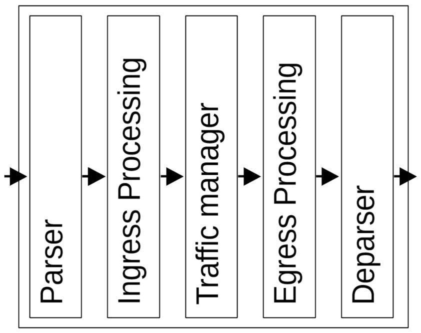

Fig. 2: Outline of a P4-based switch.

the SFU, it means the packet has just arrived and needs to be duplicated and forwarded accordingly. In this case, the UDP destination port is checked, in order to decide on the correct group of recipients. Subsequently, the packet is duplicated as needed and forwarded to the appropriate receivers.

C. P4-based SFU

The P4-based SFU uses the BMv2 framework for the `simple_switch_grpc` target implementation. An outline of a BMv2 switch is shown in Figure 2; note that processing rules can be used at both the ingress and egress processing stages. The P4 variant used is $P4_{16}$, using the `core.p4` and `v1model` modules. The SFU expects media packets to be encapsulated in UDP, therefore the headers of the P4 file include three structures, an `ethernet`, an `ipv4` and a `udp` structure. The SFU parser identifies and extracts these three headers and passes the packet to ingress processing.

To exploit the hardware multicast capabilities of P4-capable switches, we assumed that each participant was connected to a different switch port. We created a multicast group for each media stream, containing the switch ports leading to all the recipients that needed to receive that stream, and let the switch handle multicast processing. To simplify the implementation, we only used two multicast groups, one for the recipients that only needed the base video layer, and another for the recipients that also needed the enhancement video layer; note that the latter were part of both groups.

Inside the ingress processing block, four actions are specified. Action `drop` marks a packet for dropping, if there are no recipients for it, while action `ipv4_forward` forwards the packet according to the control plane rules. Finally, actions `multicastLQ` and `multicastHQ` are used to mark a packet according to the group it needs to be multicasted towards. This is achieved by setting the appropriate value in the `standard_metadata.mcast_grp` field provided by `v1model`. The multicast group entries (the ports for the appropriate recipients), are provided in the `s1-runtime` file. The action taken relies on the packet's UDP destination port.

After the ingress processing block, naturally, comes the egress processing block. There, the packets' egress switch port is compared with the ingress switch port to avoid sending the packet back to the sender; this allows using the same multicast group, regardless of the sender. Following the `v1model` architecture, the packet has its headers reassembled and gets serialized in order to be sent towards the appropriate recipients.

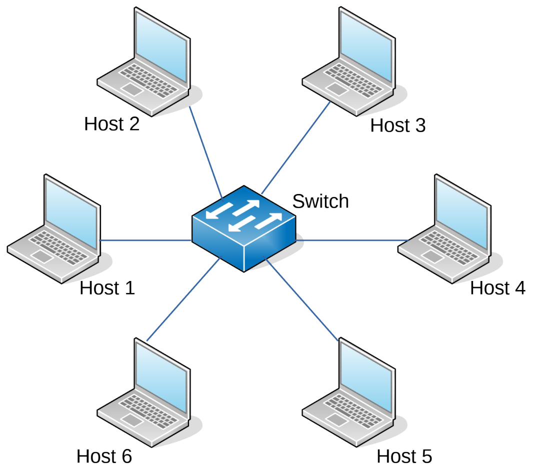

Fig. 3: The topology used in the P4-based SFU experiments.

IV. Experimental Setup

A. Environment configuration

For the implementation of the P4-based SFU, a virtual machine was used with a freshly installed copy of Ubuntu 20.04.6 LTS. In order to install all the required P4 development tools, we executed the `install-p4dev-v5.sh` script. The experiments were ran on the folders of the P4 tutorial exercises, so the main `Makefile` could be used to automate the network generation process. The project folders include a secondary `Makefile` that configures the architecture of the P4-based switch and the topology.

We used Mininet [11] to create and emulate a virtual network for the experiments with both SFUs. The traffic consists of streams of dummy UDP packets created through Python's `scapy` library. Mininet's `xterm` function was used to run these Python scripts on the client nodes. We also ran Wireshark on the SFU-hosting switch or server, to monitor and later analyze the traffic. The code for our experiments, including the client, the two SFU implementations and the analysis scripts, is available in github[5].

B. Network and application settings

The experiments were structured to mimic a conferencing application. Firstly, a network topology was established. Figure 3 shows N participating hosts communicating with each other through an SFU, in a star configuration with the P4-based switch as the hub and each host connected to a different switch port. For the server-based SFU, the SFU was a server, connected to each client host on a separate network interface, again forming a star topology, as shown in Figure 4; as we explain in Section V, the server-based SFU could only handle a small number of clients.

[5]https://github.com/Praiven/P4-SFU

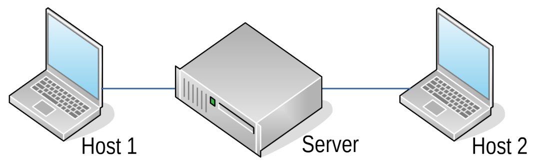

Fig. 4: The topology used in the server-based SFU experiments.

With the intention of more closely approximating a conferencing application environment, we ran two sets of experiments. In the first set, each host sent a single media stream, and all other hosts received it. This single stream configuration allows a straightforward interpretation of the results, showing clearly the delay between the sender and the recipients of each packet. To examine SFU performance with different loads, we started with only two hosts, and added hosts one by one.

In the second set of experiments, each host generated two video streams, a base layer and an enhancement layer, with all hosts receiving the base layer, but only half of them also receiving the enhancement layer. In this case, we also started with two hosts, but then added two hosts at a time, to ensure that half would receive only the base layer and the other half would receive both the base and the enhancement layer.

We initially set each host to send one packet every 36-41 ms; due to the use of Python-based clients, the intervals were not exact, but on average, each host sent roughly 1500 packets in a span of 60 seconds. In the two stream experiments, the same stream of packets was sent, but the odd-numbered ones were for the base layer and the even-numbered ones for the enhancement layer. The packet size was between 352 and 376 bits apiece (44 to 47 bytes); the only payload that the packets had, was a participant number and a sequence number, to simplify analysis (see below).

However, after looking closely at the results, we found that the server-based SFU lost a non-negligible number of packets, even with the minimum number of two hosts. To provide a fair comparison between the two SFUs, we experimented with the packet sending interval, finding that with a 60-62 ms interval, the server-based SFU did not face any loss with two hosts and one packet stream; this translated to a total of around 1000 packets in a span of 60 seconds.

It should be noted that a real conferencing application would produce significantly larger packets, especially for video; audio packets are usually small, to limit packetization delays, but video packets can reach the maximum size that Ethernet allows. By using very small packets, the dominant cost is cloning/copying packet headers; with larger packets, cloning/copying the packet payloads would dominate. However, due to the use of emulated software devices for the server and P4 switch, which do not have the processing power and memory bandwidth of real machines, we tried to reduce the load placed on the SFUs by only using nominal payloads.

C. Measurement method

The main goal of our experiments was to compare the packet processing latency between the two SFUs. To determine this latency, we measured the difference between the time the packet entered the SFU and the time the packet left the SFU. The time difference was extracted with the help of Wireshark: every packet that traveled through the virtual network was captured by Wireshark, along with its timestamp.

To correlate incoming with outgoing packets in the face of multicast/duplication, we needed to mark each packet generated by the clients, to allow us to keep track of its copies. To achieve this, each host marked its outgoing packets with a source identifier and a sequence number. Specifically, packets from host 1 would have the letter “a” in their payload, packets from host 2 would have the letter “b”, and so on, followed by an increasing sequence number; this is why the packet payload ranges from 44 to 47 bytes: packet numbers were 1 to 4 characters. The payload was kept the same when the packet was multicasted or duplicated, allowing the copies to be mapped to the original packet.

For the P4-based switch, we had two separate Wireshark captures for each switch port, one for the incoming packets towards it and the other for the outgoing packets. A Python script first read every “in” capture from Wireshark, creating a dictionary of key-value pairs. The packets’ payload (e.g., “a1”) served as the key of the pair, whilst the arrival time of the packet made up the value. Then, the “out” captures of Wireshark were read by the script. By comparing the key of every entry, we could append every instance of an outgoing packet after the instance of the original packet. Since the “in” captures were read first, lists of key-value pairs were formed with the following form: the first element of the list was the original packet which traveled from the sender to the switch, in order to be duplicated and forwarded. The following elements of the list were the duplicates of the original packet that were created from the switch’s P4 program, and sent to the recipients. We determined the processing time needed for packet duplication at the P4-based switch, by subtracting the arrival time of each packet from the arrival times of its copies.

For the server-based SFU we started a single Wireshark capture process that continuously listened on every network interface of the SFU (server). As with the first scenario, we wanted to create similar key-value pairs with the aim of extracting the latency of the SFU needed when replicating packets. This was accomplished by first iterating through the packets that had the IP address of the SFU host as their destination. These packets were sent towards the server with the purpose of being replicated and forwarded towards the proper recipients. Following the steps above, we read the remaining packets, appended them to the appropriate key-value pair, and then calculated the latency for each packet copy.

V. Evaluation Results

In this section, we present and discuss the results from our experiments with the P4-based and server-based SFUs running in Mininet. Table I shows the minimum, maximum and average packet delay of the P4-based SFU with one media

TABLE I: P4-based SFU statistics with one stream.

Hosts	Min	Max	Avg	Increase	Packets	Traffic volume
2	0.000505	0.002907	0.000778	0	1456	1071 Kbits
3	0.000471	0.004113	0.000937	20%	1484	3276 Kbits
4	0.000467	0.007507	0.001134	21%	1525	6734 Kbits
5	0.000458	0.017137	0.001675	48%	1558	11466 Kbits
6	0.000453	0.015145	0.001978	18%	1590	17553 Kbits
7	0.000456	0.030355	0.003095	56%	1610	24884 Kbits
8	0.000462	0.052992	0.004736	53%	1608	33137 Kbits
9	0.000460	0.106720	0.009122	93%	1601	42420 Kbits
10	0.000484	0.328883	0.031889	250%	1591	52693 Kbits
11	0.000514	0.989098	0.267746	740%	1585	64160 Kbits
12	0.000523	5.958905	3.986195	1389%	1569	68594 Kbits

TABLE II: P4-based SFU statistics with two streams.

Hosts	Min	Max	Avg	Increase	Packets	Traffic volume
2	0.000515	0.002859	0.000834	0	1457	804 Kbits
4	0.000484	0.014854	0.001210	45%	1548	5126 Kbits
6	0.000476	0.018257	0.001987	64%	1613	13358 Kbits
8	0.000457	0.051833	0.003915	97%	1633	25244 Kbits
10	0.000486	0.141570	0.015084	285%	1647	40919 Kbits

stream, as the number of hosts grows from 2 to 12. The table also shows the increase in average latency every time we add a host, the average number of packets sent by each host, and the total traffic volume generated by the SFU.

We can calculate the expected traffic volume generated by the SFU as $n * (n-1) * p * l$, where n is the number of hosts, p is the number of packets sent by the host and l is the packet size: $p*l$ is thus the size of each host's stream, n is the number of such streams and $n-1$ is the number of duplicates generated by the SFU for each stream. From the table, we can see that the traffic volume grows as expected with up to 11 hosts; with 12 hosts, there is only a small growth of traffic, indicating significant packet loss. But even with 11 hosts, we can see a significant growth in average latency, indicating that the P4-based SFU is reaching its capacity.

It is interesting to note that the minimum delays do not really depend on the number of recipients, since they reflect the first duplicate of a packet leaving the SFU. As such, they provide an idea of the minimum latency we can expect from the P4-based SFU, which is around 0.5 ms for our experimental setup. The average and maximum latency on the other hand grow with load. With 10 hosts the average latency is less than 32 ms, and the maximum less than 330 ms, or one third of a second, which are impressively low for a software switch running inside Mininet.

Table II shows the results from the experiments with two media streams; here, we add two hosts in each step, one receiving both streams and one receiving only a single stream, with each stream consisting of half of the generated packets. In this case, we can calculate the expected traffic volume generated by the SFU as follows. First, for the base layer that everyone receives, the traffic volume is $n*(n-1)*(p*l/2)$; the reasoning is the same as above, with the difference that only half of the packets belong to the base layer. Second, for the enhancement layer the traffic volume is $n * ((n-1)/2) * (p * l/2)$, since these packets are only sent to half of the possible receivers, or $(n-1)/2$. The table indicates that there is no significant loss with 10 hosts, but at this point the average and maximum latency have grown far more than in the previous step. With 8 hosts the average latency is less than 4 ms, while the maximum latency is around 50 ms, again, impressive for a software switch; the minimum latency is again less than 0.5 ms.

TABLE III: Server-based SFU statistics with one stream.

Hosts	Min	Max	Avg	Increase	Packets
2	0.010961	10.913122	7.860589	0	1557
3	0.015288	32.523685	23.351614	197%	1564

TABLE IV: Server-based SFU statistics with two streams.

Hosts	Min	Max	Avg	Increase	Packets
2	0.013842	6.811303	4.585829	0	1566
4	0.019133	53.473191	34.797006	659%	1614

As mentioned previously, the server-based SFU is not capable of handling these loads, so we only show results with 2 or 3 hosts in one stream mode in Table III and with 2 or 4 hosts in two stream mode in Table III; even at these levels of traffic, there is significant loss when we try to send the same packet streams as with the P4-based SFU experiments. We can see that the minimum delay through the server-based SFU is 10-20 ms, and that the average delays are on the order of seconds, as opposed to milliseconds with the P4-based SFU.

By increasing the packet interval to 60-62 ms, meaning that we only generate around 1000 packets in 60 seconds, we can run the one stream experiment on both the P4-based and the server-based SFU, without significant loss, albeit with only two hosts. Table V shows the corresponding results, which are directly comparable. The P4-based SFU has three orders of magnitude advantage over the server-based SFU in average and maximum latency, and nearly two orders of magnitude in minimum latency. Recall that these results are with minimal packets, which minimize payload copying; with large packets,

TABLE V: Statistics comparison with one stream and longer intervals.

SFU	Min	Max	Avg	Packets
P4-based	0.000576	0.002957	0.000859	993
Server-based	0.014827	5.081318	2.573470	965

the gaps would probably be far higher.

To interpret these results however, we must bear in mind several factors. First, the server-based implementation is a simple program written in Python; a real SFU would be carefully coded in C or C++ for performance. Second, we are using Mininet, which is an emulated environment, where everything (hosts, switch and servers) runs in the same computing environment; of course, this is true for both the P4-based and the server-based SFU. Third, the P4-based switch is a reference software implementation, not real hardware, which cannot exploit (for example) hardware multicast. Fourth, the network topology favors the P4-based switch, since, even though it is a software switch, it is aware that multicast is taking place and can perform some optimizations; the server-based SFU on the other hand must copy the IP packets for each network interface. Finally, the groups are statically preconfigured; dynamic host groups may be trickier to implement in P4 than in the server-based SFU.

VI. Conclusion

We have presented the design and implementation of a P4-based SFU that exploits the multicasting features of switches to reduce copying overhead and, in general, the ability of P4 to process packets closer to the hardware. Even with a software switch that cannot take advantage of such optimizations, the comparison with a similar, albeit simple, Python-based server, shows huge speedups, indicating the very large potential gains of a P4-based implementation running on real hardware.

While our approach so far relies on manual group setup, it is trivial to extend it with a signaling protocol to setup the streams desired by each sender. This would require replacing the preconfigured tables for packet replication currently used, with a controller that would receive these signaling messages and use the P4Runtime API[6] to modify the multicast groups of the P4-based switch in real time.

A natural next step for experimentation is to replace multicast via multiple ports (or NICs) with a single SFU port (or NIC) via which all clients are reached, as is common in current SFU servers; this would eliminate the advantages of multicast from the P4-based SFU. Although this would require packet recirculation inside the P4-based switch for each packet copy, in a real hardware switch it would still prevent the crossings between kernel and user space of a server-based SFU.

For experimentation, the next step is to use separate machines for the SFU and the clients, as opposed to emulating them via Mininet and, of course, using a hardware P4-capable switch, to explore the real-world performance of our approach. Especially for NMP with its ULL requirements, it would be interesting to test our P4-based SFU against a server-based SFU running at a mobile edge server, in conjunction with 5G connections that can achieve ULL-level delays.

Acknowledgment

The work reported in this paper has been partly funded by the EU's Horizon 2020 Programme through the subgrant Telepresence-Enhanced Network Music Performance (TEN-eMP, SPIRIT-OC1) of project SPIRIT (grant agreement No 101070672).

[6]https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

References

[1] K. Tsioutas, G. Xylomenos, and I. Doumanis, "An empirical evaluation of QoME for NMP," in *Proceedings of the IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2021, pp. 1–5.

[2] A. Eleftheriadis, R. M. Civanlar, and O. Shapiro, "Multipoint videoconferencing with scalable video coding," *Journal of Shejiang University SCIENCE A*, vol. 7, pp. 696–705, 2006.

[3] 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 packet processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, p. 87–95, jul 2014.

[4] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[5] G. Baltas and G. Xylomenos, "Ultra low delay switching for networked music performance," in *Proceedings of the International Conference on Information, Intelligence, Systems and Applications (IISA)*, 2014.

[6] ——, "Evaluating the impact of network I/O on ultra-low delay packet switching," in *Proceedings of the IEEE International Symposium on Computers and Communications (ISCC)*, 2015.

[7] D. Bert, N. Domini, R. Peloso, L. Severi, M. Sacchetto, A. Bianco, and C. Rottondi, "FPGA-based low-latency audio coprocessor for networked music performance," in *Proceedings of the International Symposium on the Internet of Sounds*, 2023, pp. 1–8.

[8] J. Wei and S. Bojja Venkatakrishnan, "Decvi: adaptive video conferencing on open peer-to-peer networks," in *Proceedings of the International Conference on Distributed Computing and Networking (ICDCS)*, 2023, pp. 336–341.

[9] E. Lakiotakis, C. Liaskos, and X. Dimitropoulos, "Improving networked music performance systems using application-network collaboration," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 24, p. e4730, 2019, e4730 cpe.4730.

[10] 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.

[11] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010, pp. 1–6.