

NetFEC: In-network FEC Encoding Acceleration for Latency-sensitive Multimedia Applications

Yi Qiao^{*†}, Han Zhang^{*†}, Jilong Wang^{*†‡}

^{*}Institute for Network Sciences and Cyberspace, Tsinghua University

[†]Zhongguancun Laboratory, Beijing, China

[‡]Quancheng Laboratory, Jinan, China

Abstract—In face of packet loss, latency-sensitive multimedia applications cannot afford re-transmission because loss detection and re-transmission could lead to extra latency or otherwise compromised media quality. Alternatively, forward error correction (FEC) ensures reliability by adding redundancy and it is able to achieve lower latency at the cost of bandwidth and computational overheads. We propose to re-locate FEC encoding to hardware that better suits the computational pattern of FEC encoding than CPUs. In this paper, we present NetFEC, an in-network acceleration system that offloads the entire FEC encoding process on the emergent programmable switching ASICs, eliminating all CPU involvement. We design the ghost packet mechanism so that NetFEC can be compatible with important media transport functionalities, including congestion control, pacing and statistics. We integrate NetFEC with WebRTC and conduct extensive experiments with real hardware. Our evaluations demonstrate that NetFEC is able to eliminate server CPU burden and adds negligible overheads.

Index Terms—Programmable switches; Forward Error Correction; Real-time Communication; In-network Content Replacement

I. INTRODUCTION

Latency-sensitive multimedia applications, such as conversational media conference, high-quality video streaming and cloud gaming, have resulted in a renewed interest in packet loss recovery. These applications pose strict limit on end-to-end latency to ensure Quality of Experience (QoE). For example, 150 ms mouth-to-ear delay is recommended for fluent real-time conversation between participants [1]. For cloud gaming, the round-trip time should be stringently limited to less than 130ms to guarantee smooth user experience [2].

The best-effort nature of the Internet yields communication links with unstable and ever-changing characteristics, such as packet loss due to hardware failure or wireless interference, and unstable bandwidth capacity due to cross traffic and bottleneck links. The traditional re-transmission-based packet recovery solutions will result in extra latency and downgraded media quality [3]. Loss detection and re-transmission can potentially cause a media packet to miss its play-out point, so that the receiver has to either wait for its arrival, causing extra delay, or apply techniques [3] to conceal the errors, sacrificing the quality.

Alternatively, the Forward Error Correction (FEC) solutions achieve error resilience by generating parity FEC packets from

media packets, so that the lost packets can be reconstructed on receivers with received media and FEC packets. Compared with the re-transmission methods, the FEC solutions induce less latency but *more bandwidth and computational overheads*.

FEC encoding can traditionally be performed on either senders or media servers, but they both have practical performance issues: *Firstly*, performing FEC encoding on senders can be wasteful when senders have up to hundreds of viewers. The *fec_ratio*¹ is determined by the downlink that suffers the highest loss rate. The sender will have to add FEC packets in case any of its receivers suffers lossy link. Also, the senders might be pushing multiple streams with different resolutions, so that adding FEC for every stream might easily drain the uplink bandwidth capacity; *Secondly*, performing on-demand FEC encoding on media servers deployed in CDNs might cause scalability problems. Today, most media providers deploy Selective Forwarding Units (SFU) [4] to manage multiparty connections. SFUs act like routers and they multicast the incoming media streams to all receivers. SFUs achieve high scalability by ignoring media payloads. FEC encoding on SFU servers, however, requires per-byte payload inspection, which violates the scalability-oriented design philosophy of SFUs.

We propose to re-locate FEC calculation to an alternate hardware platform that better suits the computational pattern of FEC: it only involves simple exclusive-or operations, but is *throughput-hungry* because every byte has to be inspected during encoding process. GPU (Graphics Processing Units) and FPGA (Field Programmable Gate Array) are viable hardware solutions, but GPU induces processing delay due to batching and FPGA is throughput-bounded (§VI). They also require per-machine upgrading, which will cause high expenditures (§V-F). The emerging programmable switching ASIC, however, is a more suitable option because it is designed for high throughput line-rate packet processing with a less expressive computational model. Although the programmable switches are initially intended to ease network protocol customization [5], they have already been used to accelerate many computational-intensive applications, such as deep learning tensor aggregation [6], [7], storage system erasure coding acceleration [8] and coordination services in distributed systems [9]. A programmable switch resides in the network, which is sometimes a better location to per-

Corresponding authors: Han Zhang, zhhan@tsinghua.edu.cn

¹the ratio of the number of fec packets to media packets

form computation due to its centrality and on-path nature [10]. Its benefits include high throughput, low processing delay and an all-or-nothing programming model (detailed in §II), while its limitation is a less expressive programming model. These characteristics make programmable switches a suitable option to offload computational-intensive applications, including FEC encoding. The throughput demand of FEC encoding is fulfilled, while the less expressive programming model is sufficient for commonly-used exclusive-OR based FEC schemes.

Therefore, we propose NetFEC, an in-network acceleration system that offloads the entire FEC encoding process on the emergent programmable switching ASICs, eliminating all CPU involvement. When the programmable switches receive media packets, they generate FEC packets automatically and send them to receivers. Media servers only run management tasks, such as determining FEC ratio depending on the packet loss rate, grouping consecutive packets into FEC groups and informing NetFEC of necessary encoding information. NetFEC has two novel designs, i.e., an in-network FEC encoder aiming to achieve line-rate FEC encoding on programmable switching ASICs and a ghost packet mechanism aiming to make NetFEC compatible with pacing and congestion control. **In-network FEC Encoder.** We design the in-network FEC encoder that offloads the entire FEC encoding software implementation in WebRTC [11] to the programmable switching ASICs. On receipt of media packets, the programmable switches perform exclusive-OR operations and generate FEC packets. We also use recirculation and checksum recalculation to overcome some technical problems, such as extending parser limits and checksum updating, similar to [8].

Ghost Packet Mechanism. We design the ghost packet mechanism to make NetFEC compatible with pacing and congestion control. Packet bursts will occur if the switches naively send FEC packets as soon as they are generated. To avoid burstiness, we require pacing, which is difficult to implement on programmable switches. Also, FEC can induce extra outbound traffic, which should be notified to the congestion control (CC) modules to compute sending rates and loss rates for CC algorithms such as gcc [12] or bbr [13]. With the ghost packet mechanism, the servers send ghost FEC packets, i.e., FEC packets with correct headers yet empty payloads. The other modules treat these packet as if they are real FEC packet, and function properly. The payloads are replaced when these ghost FEC packets reach the programmable switches. Packet burstiness is avoided and statistics are calculated correctly.

We implement NetFEC on real programmable switches, and integrate it with WebRTC [11], the most widely used real-time communication platform. We compare NetFEC with WebRTC default FEC scheme. We evaluate that NetFEC generates FEC packets correctly and indeed eliminates CPU overheads for different resolution of videos. We also evaluate NetFEC scalability and show that NetFEC can fulfill the throughput requirement of FEC encoding.

This paper makes the following contributions:

- We propose NetFEC, an in-network acceleration system

that offloads FEC encoding process on the emergent programmable switching ASICs, eliminating all CPU involvement.

- We design an in-network FEC encoder to achieve line-rate in-network FEC encoding and a ghost packet mechanism to make NetFEC compatible with other media functionalities.
- We implement NetFEC on programmable switches and integrate it with WebRTC.
- We evaluate that NetFEC can eliminate CPU involvement in FEC encoding and achieve high throughput with low overheads. NetFEC is also compatible with functionalities such as pacing and congestion control.

II. BACKGROUND AND RELATED WORKS

A. Forward Error Correction

Forward Error Correction achieves error-resilience by generating parity packets from the media packets. Lost packets can be reconstructed with received media and parity packets. The cost of FEC is extra bandwidth consumption and computational overhead. Researchers have explored a variety of FEC algorithms and systems for real-time communication and media streaming. WebRTC supports ULPFEC [14] and FLEXFEC [15], which are only based on XOR operation. Reed-Solomon code [16] based FEC is also widely studied, which requires finite field arithmetic.

$$\begin{bmatrix} b_{1,1} & \cdots & b_{1,N} \\ \vdots & \ddots & \vdots \\ b_{rN,1} & \cdots & b_{rN,N} \end{bmatrix} \cdot \begin{bmatrix} m_1 \\ \vdots \\ m_N \end{bmatrix} = \begin{bmatrix} p_1 \\ \vdots \\ p_{rN} \end{bmatrix} \quad (1)$$

We introduce a typical FEC encoding scheme used in WebRTC as followed. We later analyze in §VI that any FEC algorithm that follows this scheme can be easily supported by NetFEC.

The first step is to determine `fec_ratio` based on downlink packet loss rate. We denote `fec_ratio`, i.e., the ratio of the number of fec packets to media packets, as r . Suppose we want to protect N media packets, then rN parity packets are generated. FEC algorithms ensure that out of these $rN + N$ packets, any N packets received (no matter they are media packets or parity packets) can be used to reconstruct the remaining rN packets. Thus, this FEC scheme can tolerate approximately $\frac{rN}{rN+N} = \frac{r}{r+1}$ packet loss. We denote packet loss as L , then we can determine `fec_ratio` based on estimated downlink packet loss rate with $\frac{L}{1-L}$.

The second step is to determine how parity packets are generated. We follow the standard procedure in WebRTC implementation. As shown in (1), an $rN \times N$ bit-matrix is used to generate rN parity packets ($p_1 \dots p_{rN}$) with N media packets ($m_1 \dots m_N$). The i -th row of the matrix is a bit-map associated with m_i , the i -th of the N packets. The j -th bit in the i -th bitmap, i.e., b_{ij} indicates whether the i -th packet should be included in the generation of the j -th parity packet.

Finally, successive N media packets in media streams are automatically grouped into an FEC group and rN FEC packets

are generated from these N packets. The FEC packet can either be sent in a separate stream or in the same stream. The receiver can reconstruct lost media packets if it receives enough media packets and FEC packets.

B. Programmable Switching ASICs

The emergent programmable switching ASICs [5], [17] run match-action forwarding applications at line rates that are orders of magnitude higher than CPU, NPU (Network Processing Unit) and FPGA. Programmable ASICs have specialized architecture that utilizes pipelining and instruction-level parallelism [5] to support terabit-rate data plane functionalities. Intel Tofino [18] and Cavium XPlaint [19] are popular commodity products.

A programmable switch consists of an application-specific integrated circuit (ASIC) and a general purpose CPU hosting an operating system (OS). The switch ASIC is composed of ingress pipelines, a traffic manager and egress pipelines. Each pipeline is composed of a parser, match-action units and a deparser. Domain-specific languages, such as P4 [17] are used to program the parser and match-action units in each pipeline. A P4 program defines how the parser parses the packet headers to extract a set of fields, and how match-action units (each with limited TCAM/SRAM and ALUs [5]) process these fields. A general purpose CPU is connected to the switch ASIC via PCIe. An OS runs on the CPU, serving as a controller and can manipulate table contents on ASICs.

Switch ASICs have orders of magnitude higher packet processing capability than FPGA and NPU, and more importantly, their performance are *non-degradable*, i.e., computation complexity and resource usage of the installed program do not affect the processing delay ($< 1\mu s$), jitter and throughput ($\sim Tb/s$) of the switch pipeline [8], [20]. Programmable switching ASICs have the all-or-nothing characteristic [10]: programs can run at the line rate on the switch as long as they can run. Packets go through a fixed number of stages on switch, with a constant traversal time. No matter how complex the program can be, as long as it can be installed on the switch, it can run at line-rate, with up to 6.4Tb/s throughput and ~ 100 ns processing delay. The cost of the above mentioned benefits is a less expressive computational model than other hardware.

Researchers have leveraged programmable switches to offload many computational-heavy applications. SilkRoad [21] achieves fast and cheap L4 load balancing. NetCache [9] utilizes on-switch storage to achieve in-network caching for key-value store. NetChain [22] accelerates coordination services in distributed systems. *Flow [20], Univmon [23], HashPipe [24] are representative network monitoring systems that leverages switching ASICs. ATP [7], SwitchML [20], and NetEC [8] accelerate distributed computing with in-network aggregation that greatly saves bandwidth and CPU utilization.

C. Transport Optimizations for RTC

In modern real-time communication software, various transport optimizations are introduced to maintain QoE in face of

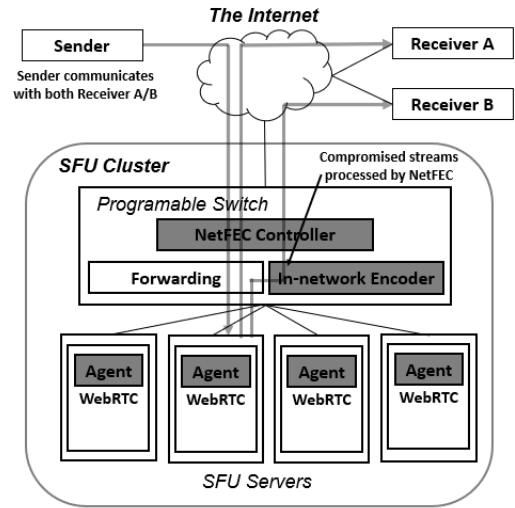


Fig. 1: NetFEC Framework.

varying and unstable network conditions. We briefly introduce some mechanisms that are closely related to FEC.

Congestion Control. Congestion Control (CC) algorithms, such as gcc [12] or bbr [13] are used to estimate downlink available bandwidth determined by downlink network capacity and the presence of cross traffic. Video encoders are informed of the available bandwidth and produce video streams with bounded bitrates because sending video streams with bitrates larger than the available bandwidth will result in buffered packets in bottleneck links, causing longer queuing delay and packet loss. When FEC is adopted, the FEC streams consume extra bandwidth, so that FEC streams should be included during probing process of CC. Also the encoders should produce media streams with even lower bitrates to make room for FEC traffic.

Pacing. If packets are transmitted in bursts, there would be risks of increased jitter and potentially packet loss. Pacing reduces burstiness by ensuring that the time interval between consecutive packets are equal. Pacing is usually achieved by delaying the packet by a certain amount of time determined by current sending rate, packet size and queue length. When FEC is adopted, the newly generated FEC packets (either on senders or on media servers) should also be paced to avoid burstiness.

III. DESIGN

In this section, we first outline the NetFEC framework, and then introduce the in-network FEC encoder and the ghost packet mechanism in detail.

A. NetFEC Components

As shown in Figure 1, multiple SFU servers are connected to a programmable switch, forming an SFU cluster. The programmable switch, deployed with the in-network FEC encoder, intercepts all inbound and outbound traffic of the entire cluster, performing basic forwarding functionalities and

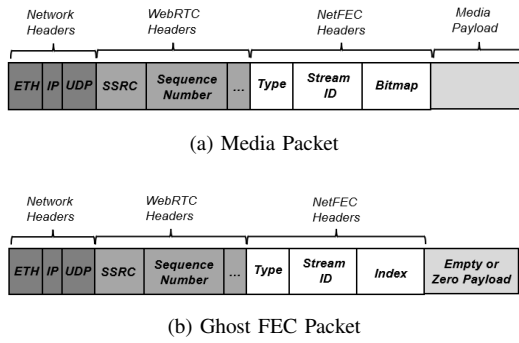


Fig. 2: NetEC Packet Format.

FEC calculations if needed. Suppose a multi-party conference or a video streaming session is hosted on one of the SFU servers in the cluster, then all streams from the participants go through this server and are forwarded to receivers that subscribe the streams. The in-network FEC encoder is only activated when a lossy link is detected. Then all streams forwarded to this receiver will go through the NetFEC module deployed on the switch to generate FEC packets.

NetFEC Controller. The NetFEC controller running on the dedicated switch OS is responsible for on-switch memory management and task management. The NetFEC controller is assigned a dedicated array of SRAM memory. It segments the memory into slots and maintains the status of each slot. The NetFEC controller negotiates with sender-side agents to coordinate the offloading tasks.

In-network FEC Encoder. The in-network FEC encoder is implemented on the programmable switch along side the basic network functionalities, such as forwarding and firewalls. The in-network FEC encoder identifies media packets, constructs and buffers parities, and fills FEC packets (detailed in §III-C).

Sender-side Agents. Sender-side agents performs necessary server-side logics, including the ghost packet mechanisms, packet re-encapsulation with the NetFEC packet format and a signaling module responsible for negotiation with NetFEC controller.

NetFEC Packet Format. As shown in Figure 2, the sender-side agents insert an extra header before payloads to facilitate the on-switch encoding process. The *Type* field is used to differentiate media packets and ghost packets. The *STREAMID* field is used for addressing. The *Bitmap* field and the *Index* field is used for encoding and ghost packet processing. This header only exists on paths between sender and the switch, and is removed when the packet leaves the switch.

B. The Encoding Life-cycle.

The whole FEC encoder process is divided between the servers and the switch. After detecting a lossy link, the server first determines the *fec_ratio* based on downlink loss rate, as described in §II. Then the server generates the encoding bitmaps (or selected from a pre-calculated set) and informs the receiver of the encoding bitmap through the control channel. The switch controller is also informed, and if the switch runs

out of memory, the NetFEC controller will respond with an alert, and the sender uses default FEC solutions. Otherwise, the switch controller allocates a piece of SRAM as the encoding buffer of the stream and returns the address of the encoding buffer back to the sending server. The sender then stops calculating FEC packets and recapsules traffic with packet format shown in Figure 2. When media packets reach the switch, the in-network FEC encoder performs FEC calculation and temporarily buffers results. When the ghost FEC packets (detailed in §III-D) reach the switch, they are filled with these results and forwarded to the receivers. When the media stream terminates, the sender informs the controller, and the allocated memory is freed.

C. In-network FEC Encoder

In this section, we illustrate the in-network FEC encoder. As shown in Figure 3, suppose the size of each FEC group is five and the *fec_ratio* is 0.6, meaning that three FEC packets are generated from five media packets, then we allocated three slots of memory for this media stream (Figure 3, ①). The size of each slot is equal to packet size².

The media packet header has a field containing the bitmap (Figure 2a), indicating which FEC packets this media packet should contribute to. In Figure 3, the bitmap of media packet M1 is 110 (②), meaning that the content of M1 should be XOR-ed with the first and the second slots (③). The bitmap of media packet M2 is 101 (④), meaning that the content of M1 should be XOR-ed with partial XOR sums stored in the first and the third slots (⑤). After all media packets have arrived, the three slots in the encoding buffer contains finished FEC payload (⑥).

The FEC packets are not encapsulated and emitted the moment they are finished in the encoding buffer, but are stored until the ghost FEC packets arrives. This is to tackle pacing and CC problems, as discussed §III-D. Each ghost packet contains an index (Figure 2b), suggesting which slot should this packet obtains its payload from. As shown in Figure 3, when ghost FEC packet G1 arrives (⑦), its payload is filled with content stored in the first slot and the slot is cleared (⑧). When all ghost FEC packets have arrived, all slot is cleared (⑨). Then the first packet of the next FEC group arrives and the encoding process depicted in Figure 3 restarts.

D. Ghost Packet Mechanism

In current video systems, such as WebRTC, FEC is strictly coupled with other functionalities. For example, FEC packets are included when calculating sending rates and packet loss rate. Also, FEC packets are paced to avoid burstiness. Offloading FEC to programmable switches must not affect these functionalities. Otherwise, there will be wrong bandwidth estimation and packet bursts, hurting media quality. However, implementing statistics or pacing logic on programmable

²Media frames usually are segmented to multiple equally-size packets(after padding) for transport. FEC packets have the same size with media packets because FEC packets are generated by XOR-ing media packets.

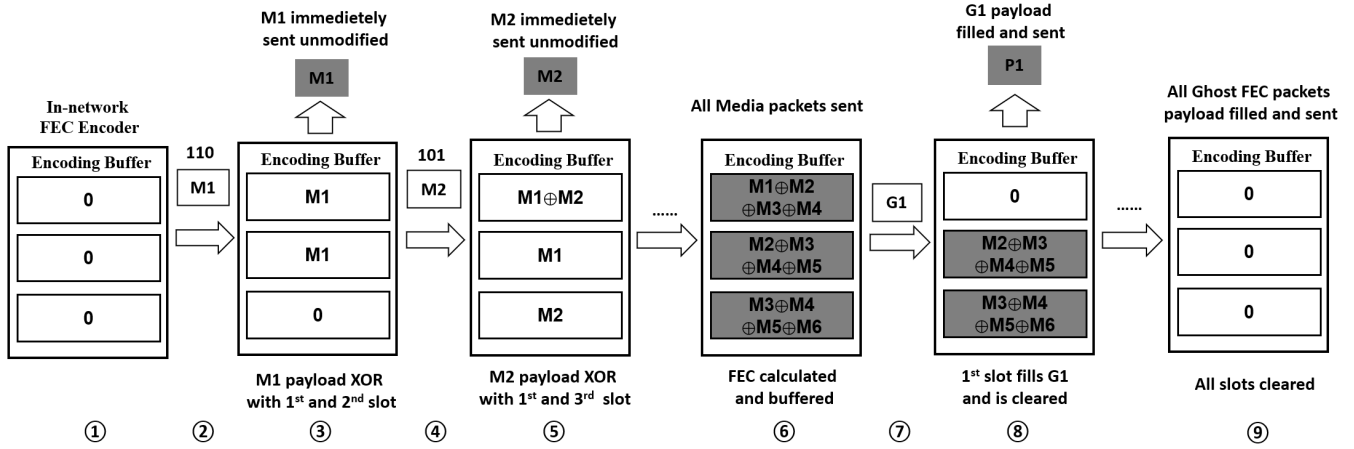


Fig. 3: In-network FEC Encoder.

switches is complicated, if not impossible, and will require too much interaction with servers.

We introduce the ghost FEC packet mechanism to achieve the best division of labour between programmable switches and server software. Servers are responsible for anything other than FEC calculations, while the programmable switches only perform FEC calculations. The server logic chains are largely unmodified, except that FEC calculations are skipped, meaning that the server generates FEC packets with empty payload. We call these ghost FEC packets. These packets have the same headers with the real FEC packets (and the corresponding data structures), except that their payloads are empty. Then the other modules treat these packets as if they are real FEC packets. The CC module computes sending rates correctly, and the pacing modules compute the sending time of the packet correctly. When these ghost packets are sent by the pacer and reach the programmable switches, they are filled with the calculated FEC payloads stored on switch and finally emitted.

Implementing ghost FEC packet is simple in most server software. We identify and delete FEC calculation logic, and replace the result with empty or zero payload. However, we need to preserve the header data structures that contain important packet metadata, such as packet type, packet size, sequence number, etc. The other modules treat ghost packets as if they are real FEC packets because the packet metadata is properly set.

Next, we discuss how other modules function properly with the help of ghost packet mechanism.

Statistics Module. In statistics modules, the sending rate is calculated by accumulating sent bytes in a period of time. The statistics modules will need to access the packet size element in the packet header data structure to calculate sending rate. We ensure that the ghost FEC packets, though their payloads are empty, have the correct packet size so that the sending rate is calculated correctly.

Pacing Module. When pacing is enabled, packets are not sent immediately when they are ready, but are delayed for a

certain amount of time depending on the current queue size, the pacing rate and the packet size. We obtain packet delay time by dividing queue size with the pacing rate, and delay the sending time. Because the ghost packet size is set correctly, the change of queue size is correct, and the sent time of the ghost packet is calculated correctly.

Sequence Number. FEC packets can either be in the same stream with the media packets or in another separate stream. If FEC packets are in the same stream, then sequence number (SN) should be translated so that SNs are consecutive in the view of receivers. If FEC packets are in a separate stream, then we need to maintain another stream with a different SSRC³ and initiating SN. These management tasks are easier to be performed on servers. When a ghost FEC packet is generated, the headers are already processed, containing correct translated SNs and other metadata.

E. Comparing with host-based and strawman solutions

We demonstrate the ghost packet mechanism in Figure 4 and compare it with the traditional host-based solution where the server performs FEC calculation and a strawman solution which naively offloads FEC calculation to the switch without considering other modules. We divide the entire encoding process into four stages.

1) Stage 1: In the host-based solution, the server performs FEC calculation, generating three parity packets out of five media packets. The statistics module watches all eight packets, accumulating packet counts and sent bytes. Finally all eight packets undergo pacing before being sent to the switch.

In the strawman solution, the server does not perform FEC calculation. The three FEC packets are not generated, so that they cannot be calculated by the statistics module, and cannot be paced. The original five media packets are sent to the switch.

In the ghost packet solution, the server does not perform FEC calculation, but three ghost FEC packets are generated

³Synchronization source (SSRC): The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header [25]

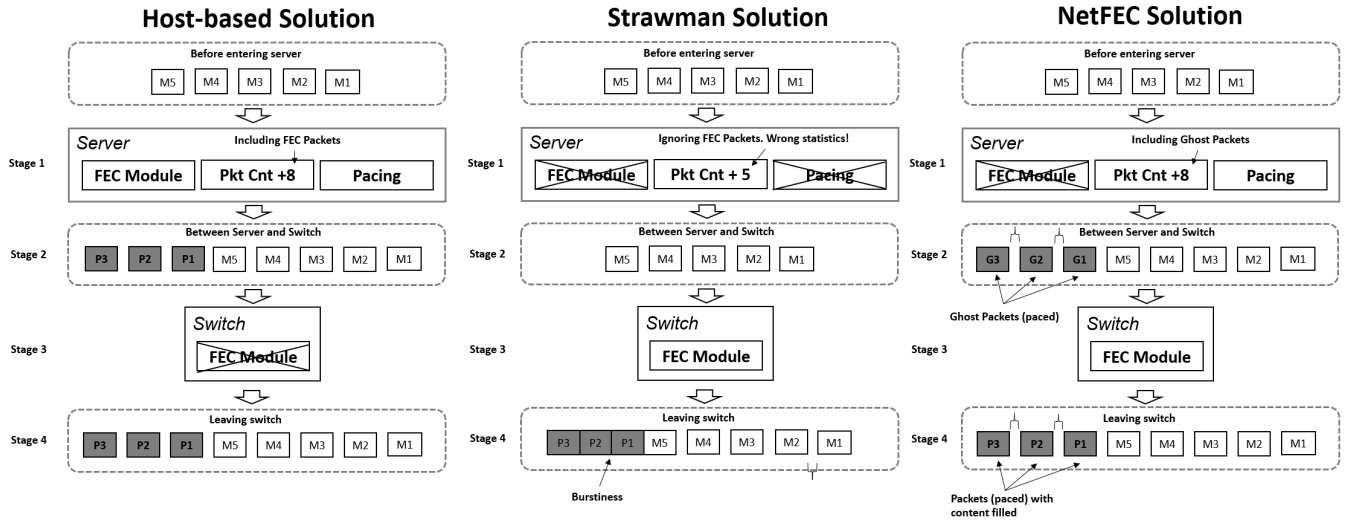


Fig. 4: Comparing with host-based and strawman solutions.

without payload. The statistics module and the pacing module treat these packets as if they are real FEC packets. The statistics are collected correctly and all eight packets are paced.

2) *Stage 2*: In the host-based solution, the five media packets and three FEC packets are transmitted between the server and the switch. In the strawman solution, only the five media packets are transmitted. In NetFEC solution, five media packets together with three ghost FEC packets are transmitted.

3) *Stage 3*: In the host-based solution, the five media packets and three parity packets pass the switch without any modification. In the strawman solution, FEC calculation is performed and three parity packets are generated. Because the pacing logic cannot be performed on switch the three parity packets are transmitted the moment their payloads are ready.

In NetFEC solution, FEC calculation is also performed, but the parity packets are not transmitted immediately. The three pieces of calculated parity payloads are buffered until the corresponding ghost packets arrive. The calculated payloads fill in the ghost packets.

4) *Stage 4*: In host-based solution, three FEC packets are added after leaving the system. All packets including the media packets and parity packets are paced.

In the strawman solution, due to lack of pacing, the three FEC packets are sent back-to-back, leading to burstiness.

In NetFEC solution, because the switch logic in stage 3 adds negligible delay, the arriving intervals between these eight packets remain unchanged, meaning they are paced when leaving the system.

F. Handling Intra-cluster Packet Loss

There exists possibility that packets are lost on the link between SFU servers and the programmable switch. Note that we discuss in §VI that such cases are very rare in our target deployment settings. Also, FEC is activated only when a media stream is already experiencing a heavily lossy link, so the loss rate within the cluster can be neglected compared to the

downlink loss rate in most situations. We still provide the following recovery methods, and let the system fallback to host-based FEC schemes when we observe high intra-cluster loss rate.

Group-wise Re-transmission. If the SFU still supports re-transmission, an intuitive solution is to adopt traditional NACK-based re-transmission. However, by the time the re-transmitted packet arrives, the slot contents are modified by subsequent FEC groups. Therefore, NetFEC adopts group-wise re-transmission. The NetFEC controller informs the SFU of the FEC group that the lost packet belongs to. The SFU waits until the currently processing FEC group is en-queued, and then en-queues the requested FEC group. This method induces extra retransmission traffic yet less switch logic.

Best-effort Recovery. If the SFU does not support re-transmission, which is usually the case because it is already adopting FEC, NetFEC adopts best-effort recovery. We add two extra bitmaps to each slot to track encoding process. The target bitmap is pre-computed and provided by the sender, representing which media packets in the FEC group should contribute to the slot. The result bitmap indicates what media packets have contributed to the slot. The result bitmap is cleared when ghost packet obtains the slot content. Normally, a ghost FEC packet should observe equality between these two bitmaps. Non-equality suggests that some media packets are lost and in this case, can never be recovered. We let the ghost packet carry the unfinished slot content and the result bitmap, which receivers can still leverage during decoding. In this way, the receivers can at least reconstruct part of the media packets with the partial yet redundant information carried in FEC packets. To handle lost ghost FEC packets, we check whether there are finished-yet-unsent slots upon arrival of media packets. We let the media packet replicate itself and let the copies carry the unsent slot contents. In this way, no information is lost but the packets are longer paced.

IV. IMPLEMENTATION

We implement a prototype of NetFEC on commodity switches. The in-network FEC encoder is implemented with P4 [17](441 LoC). The controller of the P4 program is written with python (167 LoC). We modify FEC related functions and add customized functionalities in WebRTC [11] (213 LOC, C++).

A. Integration with WebRTC

Integrating NetFEC with WebRTC requires very few code changes (213 LOC). The first part of the changes is skipping the FEC encoding related function calls, so that the generated FEC packets all have zero payload, while the packet metadata is not changed. We trace the usage of the packet metadata and make sure that other modules such as congestion control, pacing and statistics are properly functioning. The second part of code changes is in the packet parsing/deparsing modules. Before media packets are deparsed, we insert the NetFEC media packet header (Figure 2a) between WebRTC headers and the actual media payloads. Before ghost FEC media packets are deparsed, we insert the NetFEC ghost packet header (Figure 2b) after WebRTC headers and omit the zero FEC payload. In the receiving end, the NetFEC headers are skipped when packets are parsed. The third part is a simple communication module which interacts with the switch controller and performs management tasks as described in §III.

B. Switch Resource Usage

Implementing NetFEC requires SRAM usage on programmable switches because of the FEC encoding buffer (Figure 3). The SRAM consumption of each stream depends on the number of FEC packets within an FEC group. In Figure 3, three slots are occupied to buffer partial encoding results. Each slot has the same size with packet payload size. The packet payload size is upper bounded by MTU (Maximum Transmission Unit), which is usually 1460 bytes in today's Internet, and WebRTC chooses 1200 Bytes as the maximum RTP packet size. In our implementation, we select 500 bytes as the payload size, then the total SRAM consumption of a media stream is 1500 bytes if the number of FEC packets per group is 3. Modern programmable switches have ~100 MB SRAM [10], [21], meaning that approximately 60K of concurrent FEC streams can be supported with one single switch.

C. Overcoming Hardware Limits.

Programmable switches achieve high throughput and low latency at the cost of limited resources and less expressive programming model. To make possible NetFEC, we overcome the following hardware constraints.

Parser Limits. Current programmable switches can only parse no more than 400 bytes payload of the packet. We leverage recirculation to overcome this limit. For example, for a packet with M bytes payload, the switch parses and processes the first M/2 bytes, and does emit them. After deparsing the

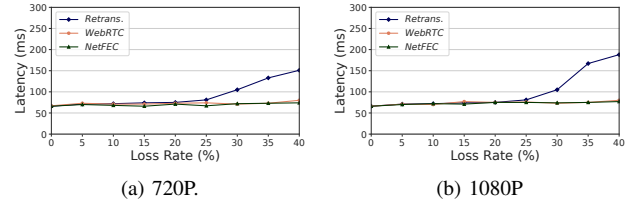


Fig. 5: End-to-end Media Latency.

packet only contains the remaining M/2 bytes. The switch recirculates this "truncated" packet instead of sending it out and processes it again. As shown in [26], recirculation once halves the overall switch throughput, but is tolerable because common commodity programmable switch has at least 3.2 tbps backplane capacity. Recirculation is a common practice in other offloading works [8], [26].

Checksum Calculation In NetFEC, after the FEC packets are filled with calculated FEC payloads, the transport layer checksum needs to be recalculated. Current programmable switches only support checksum update, but not checksum recalculation. We use similar ideas from [8] to recalculate checksums even when recirculation is involved.

V. EVALUATION

A. Setup

Our testbed includes three servers, each with two 12-core Xeon E5-2650v3 CPUs and 64GB of memory. The first server is used as the media server, deployed with Janus [27], an open source general purpose WebRTC platform. The second server is used as the sending sender, on which we use the `ffplay` command in FFmpeg [28] to send pre-recorded time-stamped H.264 videos of different resolutions to the media server. The last server is served as the receiving server. It receives the videos, records them and analyzes the end-to-end latency and PSNR (Peak Signal to Noise Ratio) of the reconstructed videos. On the receiving server, we use Linux `tc` [29] commands to emulate a lossy link. The three servers are connected to a 3.3 Tb/s Intel Tofino programmable switch [18] via 40GbE NICs.

We compare three loss recovery solutions to show correctness and effectiveness of NetFEC. The first solution is the traditional retransmission solution based on NACK packets. The second solution is the default WebRTC ULPFEC, where FEC packets are calculated by media server CPUs. The third solution is NetFEC, where FEC calculations are divided to media servers and the programmable switches.

B. End-to-end Latency and Media Quality

In this section, we show that NetFEC can achieve the same FEC encoding performance with WebRTC default FEC. FEC packets can be successfully parsed and used for reconstruction. End-to-end media latency and media quality (measured with PSNR) are not affected.

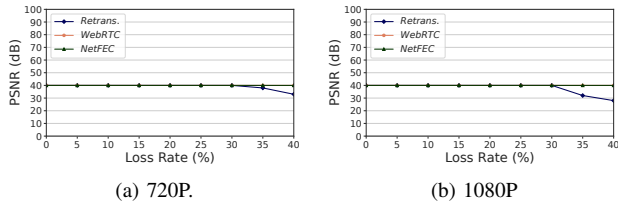


Fig. 6: PSNR of Reconstructed Videos.

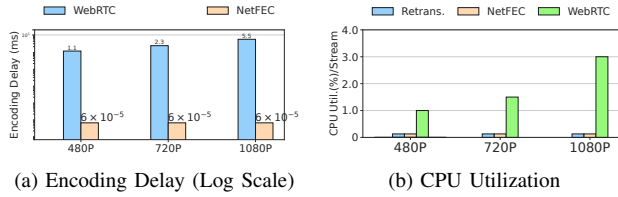


Fig. 7: FEC Encoding Overheads.

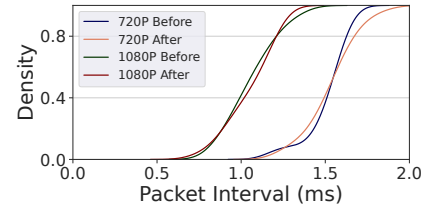
The end-to-end latency is the entire latency between the sending time and the display time of a video. It includes processing and communication, such as media encoding/decoding, FEC encoding/decoding, network transport and buffering. To calculate correct end-to-end latency, we play edited 720P and 1080P videos on the sending server with `ffmpeg`. We edit these videos so that timestamps showing local time are displayed as part of the video. The timestamp is an ever-changing text showing current local time above the video content that can be seen with human eyes. The received videos are recorded on the receiving server and the end-to-end latency are calculated with the displayed timestamp of a capture frame and local time when capturing the frame.

We manually configure the `fec_ratio` to be 1, so that up to 50% packet loss could be tolerated. Figure 5 shows that retransmission solution cannot handle high loss rate, while NetFEC and WebRTC default solution can both maintain low end-to-end latency. We also measure the PSNR of the received videos. Figure 6 shows that retransmission solution leads to compromised media quality when loss rate exceeds 30%. NetFEC and WebRTC both maintain high media quality in face of packet loss.

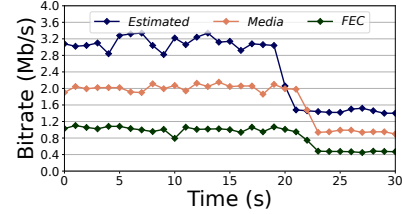
C. FEC Encoding Delay

We measure the FEC encoding delay, both on the media server and on the programmable switch. We record timestamps before and after the FEC calculation functions in WebRTC FEC implementations to calculate the FEC encoding delay on media servers. On the programmable switch, we measure the extra processing time of the in-network FEC encoder with the hardware-supported timestamp mechanism.

We show that the FEC encoding process of 480P, 720P and 1080P videos on servers requires 1.1 ms, 2.3 ms and 5.5 ms. Processing on the switch requires extra 60 ns because of the extra logic of the in-network FEC encoder. The above



(a) Packet Interval CDF.



(b) Congestion Control.

Fig. 8: Effectiveness of Ghost Packet Mechanism.

results show that NetFEC also reduces the FEC encoding delay (though FEC encoding delay is insignificant in the entire end-to-end latency compared to media encoding delay and network transport).

D. Effectiveness of Ghost Packet Mechanism

We capture packets on the outbound interface of the media server (before entering the switch) and the inbound interface of the receiving server (after leaving the switch), and measure the distribution of packet intervals. We show in Figure 8a that the packets leaving the media server are properly paced, with equal packet intervals. Also the packets leaving the switch had the same packet interval distribution, meaning that the switch did not affect packet intervals. §V-C has shown that NetFEC adds negligible encoding delay, meaning that NetFEC will not change packet arriving intervals, thus will not affect pacing.

To show how NetFEC affects congestion control, a single 720P video is played and we configure `fec_ratio` to be 0.5, so that the media bitrate is twice the FEC bitrate. We use Linux `tc` to limit bandwidth on the receiving server. We then record the estimated downlink capacity and the sending rate of both the media stream and the FEC stream. Results in Figure 8b show that after bandwidth is limited, the congestion control module of WebRTC detects congestion in approximately 2 seconds and reduces the estimated downlink capacity. The bitrates of both media stream and FEC stream is reduced proportionately.

E. CPU Utilization

We measure the CPU utilization per stream when `fec_ratio` is set to 1. The WebRTC default solution suffers high CPU overheads because FEC calculation is performed on CPUs. The CPU utilization are higher when processing high resolution videos. NetFEC, on the other hand, achieves the

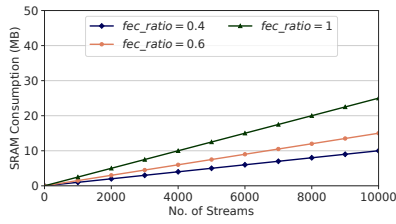


Fig. 9: NetFEC SRAM Usage.

same CPU utilization with retransmission solution, which is order of magnitude lower than WebRTC default solution. This is because NetFEC and retransmission solution only process packet headers, while the WebRTC default solution requires inspecting payloads.

Processing 1080P videos consumes up to 3.0% CPU usage, meaning that only about 30 FEC encoding tasks can be performed concurrently. With NetFEC, the consumption per stream becomes 0.13 % so that about a thousand concurrent FEC streams can be processed on a single server, significantly increasing concurrency.

F. NetFEC Scalability and Cost

As shown in §IV-B, the SRAM consumption of each FEC stream is related to the number of FEC packets per group, regardless of video codecs, resolutions or bitrates. Then the switch SRAM usage grows linearly with the number of concurrent FEC streams, as depicted in Figure 9, where the FEC group size is fixed to 5 and the FEC packets per group is determined by *fec_ratio*.

§V-E shows that each server can support about a thousand concurrent 1080P FEC streams, and §IV-B shows that a single switch can hold ~60K concurrent streams. Then a single programmable switch can theoretically cover up to 60 SFU servers in a NetFEC cluster described in §1. The cost of replacing a normal switch with a programmable switch (~\$15K) is far lower than equipping ~60 servers with other hardware, such as GPUs and FPGAs.

VI. DISCUSSION

Encryption. Encryption is a critical issue in many in-network acceleration systems. In FEC systems, the effect encryption depends on the order of encryption and FEC encoding. If each packet is first encrypted and FEC packets are constructed from these encrypted packets, then NetFEC will function normally. The receiver receives encrypted media packets and FEC packets generated with these encrypted packets. The receiver decrypts the reconstructed packets and obtains the original content.

Supporting Other FEC Algorithms. Any FEC algorithms that follow the procedures introduced in §II can be supported by NetFEC. Some exclusive-OR based FEC algorithms carefully design the encoding bitmatrix to fulfill more requirements, such as layer-aware protection and high performance. Supporting these algorithms only requires updating

the bitmatrix generation function, without changing the NetFEC framework. Another family of FEC algorithms is based on Reed-Solomon code, which is successfully offloaded to programmable switches by another work [8]. NetFEC can potentially borrow its ideas to support RS-based codes.

Comparing With Other Hardware Solutions. GPU and FPGA are popular hardware solutions to offload computations. However, we argue that they are not suitable for FEC encoding. GPU achieves high throughput through batch processing, leading to extra delay. FPGA supports more flexible programming model, but is throughput bounded. More importantly, using these hardware requires upgrading every server in the cluster, which induces significant operational and capital expenditure. In contrast, programmable switches can cover a cluster with multiple servers as shown in §V-F.

Packet Loss and Out-of-order within clusters. We assume packet loss and out-of-order is rare, because of the following two reasons. On the one hand, as shown in Figure 1, most in-network acceleration frameworks, including NetFEC, are deployed on the top-of-the cluster rack switch. The distance between NetFEC and the servers is only one-hop and most in-network schemes (e.g. reference [22], [23], [26], [27], [30]) also take this assumption; on the other hand, in these schemes there is little cross traffic within the cluster since the cluster is dedicated to specific application (e.g., machine learning) to achieve high performance. Therefore, packet loss is rare in the single-hop high-speed link with little cross traffic. Still, we design fallback and recovery mechanisms in §III-F

VII. CONCLUSION

Latency-sensitive media applications are moving to forward error correction from re-transmission because of stringent latency requirement. Loss detection and re-transmission induce intolerable delay, affecting QoE. FEC fulfills the latency requirement at the cost of high bandwidth and computational overheads. We propose to re-locate FEC calculation to hardware that better suits the computational pattern of FEC encoding than CPUs. We propose NetFEC, an in-network acceleration system that offloads FEC encoding from CPUs to the emergent programmable switching ASICs. NetFEC is composed of an in-network FEC encoder that achieves line-rate FEC and a ghost packet mechanism that makes NetFEC compatible with pacing and congestion control. We implement a prototype of NetFEC and integrate it with WebRTC. Evaluations show that NetFEC significantly reduces CPU utilization of FEC encoding, achieves high throughput and is compatible with other functionalities.

ACKNOWLEDGEMENT

The research is supported by the National Natural Science Foundation of China under Grant No. 62394322. This work is part of Future Internet Technology Infrastructure (FITI).

REFERENCES

- [1] "ITU-T. 2003. g.114 one-way transmission time. series g: Transmission systems and media, digital systems and networks international telephone connections and circuits general recommendations on the transmission quality for an entire international telephone connection (2003), 1–20." 2022.
- [2] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 151–165.
- [3] S. Holmer, M. Shemer, and M. Paniconi, "Handling packet loss in webRTC," in *2013 IEEE International Conference on Image Processing*. IEEE, 2013, pp. 1860–1864.
- [4] "Selective Forwarding Unit," 2022, <https://webrtcglossary.com/sfu/>.
- [5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.
- [6] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 150–156.
- [7] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 741–761.
- [8] Y. Qiao, M. Zhang, Y. Zhou, X. Kong, H. Zhang, J. Bi, M. Xu, and J. Wang, "Netec: Accelerating erasure coding reconstruction with in-network aggregation," *IEEE Transactions on Parallel and Distributed Systems*, 2022.
- [9] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 121–136.
- [10] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch code generation using program synthesis," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 44–61.
- [11] "WebRTC," 2022, <https://webrtc.org/>.
- [12] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Congestion control for web real-time communication," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2629–2642, 2017.
- [13] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
- [14] "RFC 5109," 2022, <https://datatracker.ietf.org/doc/html/rfc5109>.
- [15] "Flexible FEC," 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-payload-flexible-fec-scheme>.
- [16] "Reed-Solomon Code," 2022, https://en.wikipedia.org/wiki/Reed-Solomon_error_correction.
- [17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [18] "Intel Tofino," <https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html>, 2018.
- [19] "Cavium Xpliant," <https://www.openswitch.net/cavium/>, 2018.
- [20] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow," in *2018 USENIX Annual Technical Conference (USENIX ATC18)*. USENIX Association, 2018.
- [21] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.
- [22] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.
- [23] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 101–114.
- [24] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.
- [25] "RFC 3550," 2022, <https://www.ietf.org/rfc/rfc3550.txt>.
- [26] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," 2019.
- [27] "Janus," 2022, <https://janus.conf.meetecho.com/>.
- [28] "FFmpeg," 2022, <https://ffmpeg.org/>.
- [29] "Linux TC," 2022, <https://man7.org/linux/man-pages/man8/tc.8.html>.