Scalable Video Conferencing Using SDN Principles

Oliver Michel
Princeton University
omichel@princeton.edu

Satadal Sengupta
Princeton University
satadals@princeton.edu

Hyojoon Kim
University of Virginia
joonkim@virginia.edu

Ravi Netravali
Princeton University
rnetravali@princeton.edu

Jennifer Rexford
Princeton University
jrex@princeton.edu

Abstract

Video-conferencing applications face an unwavering surge in traffic, stressing their underlying infrastructure in unprecedented ways. This paper rethinks the key building block for conferencing infrastructures — selective forwarding units (SFUs). SFUs relay and adapt media streams between participants and, today, run in software on general-purpose servers. Our main insight, discerned from dissecting the operation of production SFU servers, is that SFUs largely mimic traditional packet-processing operations such as dropping and forwarding. Guided by this, we present Scallop, an SDN-inspired SFU that decouples video-conferencing applications into a hardware-based data plane for latency-sensitive and frequent media operations, and a software control plane for the (infrequent) remaining tasks, such as analyzing feedback signals and session management. Scallop is a general design that is suitable for a variety of hardware platforms, including programmable switches and SmartNICs. Our Tofino-based implementation fully supports WebRTC and delivers 7-422× improved scaling over a 32-core commodity server, while reaping performance improvements by cutting forwarding-induced latency by 26×. We also present an implementation of Scallop on the BlueField-3 SmartNIC.

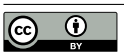
CCS Concepts

• **Networks → Programmable networks**; **Middle boxes / network appliances**; *In-network processing*; *Application layer protocols*.

Keywords

video conferencing, programmable data planes, selective forwarding units, SDN, P4, WebRTC

ACM Reference Format:
Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. 2025. Scalable Video Conferencing Using SDN Principles. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3718958.3750489



SIGCOMM '25, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/2025/09
https://doi.org/10.1145/3718958.3750489

1 Introduction

Video-conferencing applications (VCAs) such as Google Meet and Zoom have become essential for remote work, education, and social interactions. The past decade has seen substantial efforts to improve these applications, e.g., via more efficient codecs [16, 52, 62], rate-adaptation algorithms [3, 11, 14, 63], and measurement studies [2, 4, 36, 41]. Though effective, prior work has primarily focused on end users, with the scaling challenges that VCA *operators* face to support exploding traffic rates [15, 37] being far less explored.

At the core of VCA infrastructure are selective forwarding units (SFUs) [18, 25, 35, 39]. These servers are tasked not only with relaying media streams among meeting participants, but also with monitoring and adapting media signals to match the time-varying network capabilities of users. Unfortunately, their deployment on general-purpose servers – the status quo today – makes scaling very difficult, particularly given several fundamental properties of VCAs (§2):

- The workload of an SFU is *hard to predict and can change rapidly*. Beyond diurnal variation, the number of streams that an SFU must handle grows quadratically with the number of participants in a meeting, i.e., even a single new participant in a meeting introduces substantial load since their media streams must be relayed to all other participants, and they must receive all media streams from all other participants.
- And yet, the replication and forwarding that SFUs perform on media packets is on the *latency-critical path of user interactions*. At hundreds of packets/sec./stream, operating-system delays for software packet processing (e.g., scheduling, context switches, interrupts, socket-buffer copying) can lead to significant user-perceived jitter and latency, especially in the face of under-provisioned resources.

As a result, VCA operators are left with two options today: massively over-provision SFU server infrastructure to handle peak loads, which is costly and wasteful, or (reactively) auto-scale those resources using traditional mechanisms [47, 56] which risks harming QoE for users.

In this paper, we forego ephemeral SFU scaling enhancements (e.g., improved software packet processing or provisioning mechanisms) in favor of a fundamental rethink of VCA infrastructure that can support long-term traffic forecasts. Guided by our detailed study of production SFUs and real campus VCA traces (§3), our key insight is that: despite the large semantic gap, the forwarding and adapting of media signals at SFUs — the most frequent tasks that account for the lion's share of scaling overheads — is *strikingly similar to traditional packet-processing tasks*. Indeed, relaying and

adapting media signals in today's SFUs can be distilled down to primitives such as packet replication and selective dropping that network hardware is optimized for.

Fueled by this insight, we present **Scallop**[1] a new hardware/software co-designed SFU that is built on top of the WebRTC standard. Drawing inspiration from SDN principles, Scallop decouples SFUs into (1) an efficient data plane that adapts and relays high-volume media streams using line-rate network switching hardware, and (2) a two-tier software control plane that handles the remaining infrequent tasks, e.g., session management, periodic feedback handling, and signaling. The potential benefits are significant, with promises of 7-422× improved packet-processing performance over general-purpose servers at similar cost with fixed per-packet delays to eliminate SFU-induced jitter.

Yet, decoupling SFUs in this manner requires a rethink of their design and introduces three challenges, which we describe next.

First, since not all SFU functions are amenable to processing in hardware, we decouple SFUs into a 3-tier architecture that reflects the different time scales and latency requirements of SFU operations as well as the locality of their data. Specifically, (a) session management is complex but occurs at low frequency and benefits from a global view of the system, making it suitable for centralized processing in software; (b) network feedback processing and rate estimation occur at medium frequency and are local to each SFU instance, requiring a responsive—but not ultra-low-latency—control loop in software on each Scallop network device; and (c) media forwarding occurs at hundreds of packets per second per stream and critically depends on low latency in the data plane. Decoupling these tasks in practice, however, is challenging while preserving all application semantics and the correctness of the forwarding and adaptation logic. To address this, we present a detailed analysis of the SFU workload and then place all of the many different SFU functions on one or more of these tiers while maximizing performance. Moreover, we describe a series of methods for filtering, routing, and processing the many feedback messages in VCAs to maintain faithful SFU operation in this redesigned architecture.

Second, Scallop's data plane must implement the media-replication and forwarding tasks of an SFU in hardware. Most packet-processing platforms have packet *mirroring* capabilities; however, unlike mirroring where only one packet replica is created, VCAs require as many replicas as there are receivers. To address this mismatch, we present a general but hardware-amenable solution that can be implemented on a wide range of packet-processing platforms and accelerators, including switches, SmartNICs, or eBPF. From this, we derive two platform-optimized designs for programmable switches (here the Intel Tofino2) and SmartNICs (here the NVIDIA BlueField-3). These designs play to the strengths of the respective platforms while accommodating their constraints.

Third, to build a system that is practical and deployable, we follow WebRTC, a widely adopted standard for real-time communication. To do so, we must ensure that our SFU implementation is transparent to unmodified WebRTC clients running in browsers or on mobile devices. We do this in two ways. First, owing to the peer-to-peer (P2P) design of video-conferencing protocols like WebRTC, SFUs traditionally operate as *split proxies* that terminate and spawn new client connections. However, this design would burden the SFU with tasks typically handled by end hosts and unsuited for network hardware (e.g., packet de/re-encryption), ultimately increasing control-plane overheads and reducing scalability. Instead, we aim to run SFUs as *true proxies* by capitalizing on the observation that most of these functions already run individually at clients and need not be replicated at SFUs. Doing so, while remaining faithful to all SFU semantics, requires a redesign of the way WebRTC establishes sessions and handles feedback signals at each participant. Second, our proxy SFU design must ensure that all traffic (media and control) appears to clients exactly as if it were sent from another client, without intermediate processing. Among other challenges discussed in the paper, this is especially difficult when adjusting the media rate to adapt to network conditions, which, if done naively, creates sequence gaps that WebRTC receivers interpret as network losses, triggering unnecessary retransmissions. To maintain the P2P illusion, the SFU must rewrite sequence numbers to mask intentional gaps in the stream. However, such rewriting is inherently difficult, especially in the presence of network-induced loss and reordering—even software implementations cannot do this perfectly. Our experiments reveal that leaving extra gaps is preferable to masking legitimate ones: missing sequence numbers trigger packet retransmissions, while incorrect rewrites break the decoder's state, leading to a permanent freeze that can only be recovered by a new key frame, causing severe video freezing. Based on this finding, we design a hardware-friendly sequence-number rewriting heuristic that minimizes retransmissions while preserving stream continuity, even under high loss or reordering. We further discuss possible changes to the WebRTC protocols and reference implementation that would enable more efficient in-network processing of media streams to improve overall system scalability.

We implemented Scallop using P4 on a 12.8 Tbit/s Intel Tofino2 ASIC and a NVIDIA BlueField-3 SmartNIC. In experiments replaying campus-scale Zoom traces, Scallop handles 96.5% of all packets and 99.7% of bytes entirely in the hardware-based data plane. On the Tofino, this enables Scallop to support up to 128,000 concurrent meetings on a single switch, a 7-422× improvement over a 32-core commodity server running existing SFUs [39]. Further, Scallop reduces the latency introduced by SFUs by a factor of 26.8, improving QoE for all participants. We will publicly release our Scallop implementation post publication.

Ethics. The campus traces and Zoom API data used in this study were anonymized with a one-way hash and media payloads were removed at source. All data were inspected and sanitized by an authorized network operator to remove all personal data before being accessed by researchers. In the packet traces, the media payloads have been removed as well before researchers gained access. This study was conducted with approval from Princeton University, including its Institutional Review Board (IRB).

2 SFU Scaling Challenges

2.1 Meeting Topologies and SFUs

Need for SFUs. Modern VCAs use SFUs as the de-facto standard to connect participants for two main reasons: First, while P2P connections are possible for meetings with more than two participants, they are impractical due to the need for each participant to encode

[1] Wordplay on "scale up".

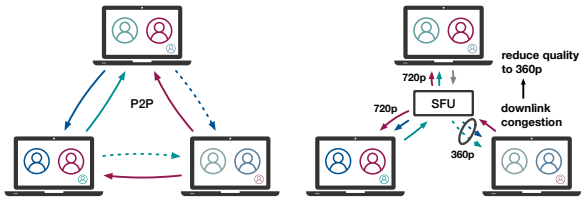


Figure 1: VCA Architectures: P2P vs. SFU

and send media to every other participant (Figure 1, left). This results in significant computational overhead and requires substantial uplink bandwidth which is often unavailable in residential and wireless Internet settings. Second, an intermediate (publicly routable) server solves challenges associated with network address translators (NATs), making SFUs useful even for two-party calls. For example, Google Meet always uses an SFU for two-party calls.

SFU Scaling Properties. While the use of SFUs solves the problem of constrained uplink bandwidth and reduces the required CPU resources at clients, SFUs do not entirely solve the scalability problem in multi-party video conferencing as sometimes suggested [24]. The number of streams required in an SFU scenario still grows quadratically with the number of participants. There are N^2 media streams for N participants (per media type, i.e., video, audio, or screen share) in an SFU topology (see Figure 1) which even grows slightly faster compared to the $N(N-1)/2$ streams in a P2P topology. The key difference is that all these streams are now sent to or received by the SFU, effectively moving the bottleneck from the clients to the SFU. The second important observation is that the amount of work that an SFU needs to perform is determined by the number of media streams and not by the number of participants which therefore leads to quadratic scaling behavior at the SFU. This is due to the fact that every incoming stream needs to be replicated for each of the downstream participants and then sent out. Figure 1 shows an example of this with three participants where P2P connections result in a total of six streams while the SFU handles nine streams.

SFU Load in Campus Trace. Of course, meeting participants do not share both audio and video at all times. To get an understanding of the actual number of streams per meeting in a real-world video-conferencing application, we analyzed Zoom usage data taken from the Zoom Account API [64] on our University campus (more in Appendix A). The data was collected over the course of two weeks in October 2022 and contains 19,704 meetings. More details on the data set can be found in Appendix A. Figure 2 shows the range (gray bars) and median number (blue dots) of media streams at the SFU for each meeting as a function of the maximum number of participants within each meeting on our campus. A media stream was counted when it was active for at least 10% of the meeting's duration. The dashed line indicates the upper bound of streams possible if every participant shares both audio and video which can be exceeded in practice when participants also share their screen, as seen in this figure. Even for meetings with 10 participants, the SFU already handles up to 200 media streams. Meetings with 25 participants, a typical classroom size, generate over 700 streams at the SFU in our data set and can theoretically produce up to 1250 streams.

2.2 Consequences of Under-Provisioning

SFU Performance Implications. In contrast to signaling and rate adaptation, media distribution is latency-sensitive. SFUs must touch each of these media packets, and as such, any delay introduced by the SFU is added to the end-to-end delay a user experiences, directly impacting QoE. Consequently, it is crucial that the forwarding delay and induced jitter are minimal. However, software packet processing is subject to OS-level delay artifacts due to scheduling, context switches, interrupts, etc., adding to the forwarding delay. At hundreds of packets/sec./stream (typically between 800 and 1400 Bytes in size for video and around 200 Bytes in size for audio), SFU operations require copying significant amounts of data among socket buffers for receiving packets and before sending them out, resulting in high CPU load and frequent context switching. These delays are hard to predict and impair session quality o the point where the VCA becomes unusable.

QoE Degradation under High Load. To confirm the suspected quality impacts of under-provisioning SFUs, we conducted an experiment using the Mediasoup open-source SFU [39] which we deployed on a server with a 40-core Intel Xeon Silver 4114 CPU and 96GB of RAM in our testbed (§8). A second emulated clients using headless Chrome and was directly connected to the SFU via a 10Gbit/s Ethernet link. We pinned the Mediasoup server to a single CPU and incrementally built up to 15 meetings with 10 participants each, adding one participant every ten seconds. We measured the quality of the first meeting using the WebRTC statistics API [58] as we added more participants to the SFU. The server reached 100% CPU utilization at around 80 participants. Figure 3 shows the receive jitter. Tail jitter is high throughout the experiment before exceeding 100ms, causing significant mouth-to-ear delay and freezes, as depicted in Figure 3. Figure 4 shows that the video frame rate starts dropping at around 60 participants and making the session effectively unusable beyond 100 participants.

Takeaways. In summary, SFUs share scaling properties (e.g., diurnal usage patterns) with other user-facing services. Additionally, however, they exhibit unique scaling challenges due to the quadratically growing amount of media streams to be forwarded. Dynamics and unpredictability within meetings, for example, due to participants joining or leaving or starting or stopping to share a particular media type (e.g., video, audio, or screen), further exacerbate this problem. At the same time, under-provisioned SFUs can rapidly hit high utilization levels, which have a direct, noticeable, and sometimes prohibitive impact on the session quality. Taken together, VCA operators either vastly over-provision their infrastructure to accommodate such dynamics or they jeopardize QoE.

3 SFUs as Packet Processors

Before presenting Scallop, we provide an overview of SFU operations and the key insights guiding our design.

SFU Design Choices. As opposed to earlier generations of intermediate servers, SFUs do not mix or transcode media streams but instead operate on *packetized* media. WebRTC is the only widely adopted standard and open-source framework for video conferencing, yet it does not provide any guidance on how SFUs should be implemented or how they should handle RTP streams. The simplest way to implement an SFU is to maintain separate P2P connections

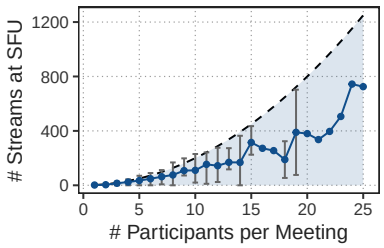


Figure 2: Number of media streams per meeting in campus trace

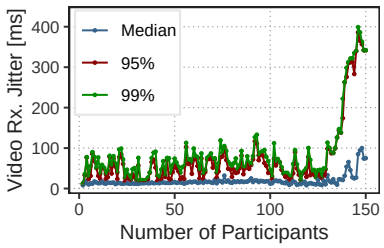


Figure 3: Video jitter while adding participants to the SFU

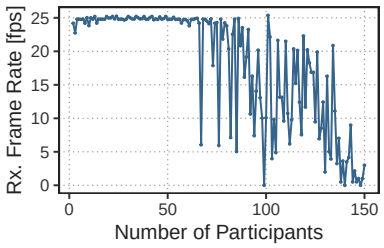


Figure 4: Video frame rate while adding participants to the SFU

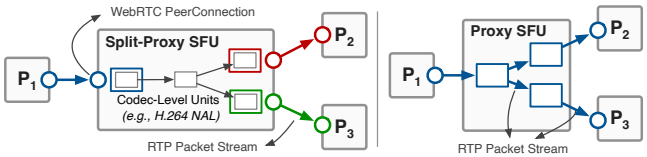


Figure 5: SFU design choices

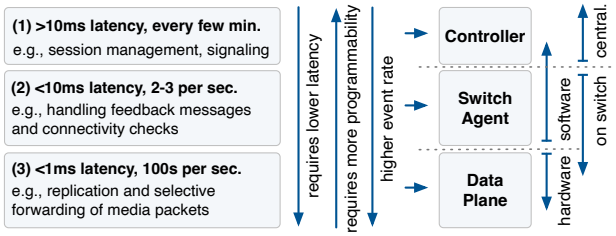


Figure 6: Latency and programmability requirements of key SFU responsibilities and resulting placement

and distinct RTP streams between the SFU and each participant. This approach is similar to a *split-proxy* design, a term we will use going forward and is illustrated on the left side of Figure 5. Existing WebRTC SFUs (e.g., MediaSoup [39]) operate in this way.

Alternative Approaches. In contrast to the aforementioned *split-proxy* design, we design Scallop using a powerful insight that makes the design of SFUs computationally simpler: SFU functionality can be accomplished by replicating packets and rewriting header fields. These tasks are supported by traditional packet processors (e.g., switches) and do not require software processing. We look for evidence of such a design in the real-world by collecting and analyzing packet-level traces of meetings on our campus (see Appendix B). In accordance with previous studies [20, 41], we find that the SFU servers used by Zoom—one of the market leaders in video conferencing—send exact copies (except for rewritten IP addresses and port numbers) of the incoming RTP packets to all downstream participants. While we do not know Zoom's full architecture or how their SFUs are implemented, these findings lend credibility to Scallop's design.

Rate Adaptation in Zoom's SFUs. Rate adaptation becomes necessary when the network conditions of a participant change, e.g., due to network congestion or if it is not necessary to forward high-resolution video to a participant due to device characteristics, such as a smaller screen on a cell phone. Without rate adaptation at the SFU, media senders would all have to reduce their sending rate to relieve congestion, resulting in lower quality for all participants, even those unaffected by congestion (see §5.3). Realizing this functionality (without transcoding media) requires using a scalable media stream, for example using Scalable Video Coding (SVC). In SVC, video is encoded in multiple layers, each with a different bitrate. The media stream is packetized so that a given packet always belongs to exactly one layer. As a result, reducing the media resolution or frame rate can be achieved by dropping a specific subset of packets. This insight is the second foundation of our design. Our analysis of Zoom's SFU reveals that it uses a combination of Simulcast and SVC to achieve rate adaptation, providing further credibility to our design. We explain SVC using a real-world example in Appendix C.

Takeaways. Taken together, the core work of SFUs can be implemented by replicating packets and sending copies out to all receiving participants. Furthermore, if SVC is used, adapting a media stream reduces to forwarding a clearly defined subset of packets to a given participant.

4 Introducing Scallop

Based on the insights from §3, we introduce Scallop, a novel SFU design leveraging SDN principles and programmable networking hardware to improve the scalability and performance of video-conferencing infrastructure. Scallop offloads all media replication, forwarding, and rate-adaptation tasks to high-speed hardware, yet several operations SFUs perform are not amenable for such an implementation. Consequently, we require a split of functionality where we leave as many tasks as possible inside the data plane and only carefully leave operations in software when absolutely necessary.

A Taxonomy of SFU Operations. Scallop's control/data-plane split is driven by the latency requirements, computational complexity, and frequency of the tasks an SFU performs. Along these axes, we classify SFU tasks into 3 categories as shown on the left side of Figure 6: (1) infrequent tasks that are not latency-sensitive, including session management and signaling; (2) latency-sensitive tasks (on the order of 10s of milliseconds), including deciding the correct target sending rate of a media stream based on feedback signals, as well as handling connectivity checks performed by the STUN protocol [38]; (3) ultra-low-latency tasks (sub-millisecond), including the actual handling of media packets, i.e., forwarding and dropping them if necessary.

Three-Tier SFU Design. The resulting architecture (right side of Figure 6) is inspired by the design of SDN systems but goes a step further by introducing a third plane between a centralized controller and the switch data plane for the aforementioned latency-sensitive tasks. Here, the central controller is only involved when (1) a new session is created, (2) a participant joins or leaves a meeting, or (3) a participant starts or stops sharing a particular media type (i.e., audio, video, or screen). The controller is centralized as it is designed to manage multiple distributed Scallop instances to enable more efficient forwarding topologies (§9). Finally, the data plane handles truly latency-sensitive tasks on the critical path for QoE.

Scallop's Switch Agent. All remaining tasks that are either (1) not amenable to implementation in the data plane due to their complexity, or (2) fall in the category of latency-sensitive tasks that not on the critical path for QoE, run in software directly on the switch (i.e., the switch's CPU and operating system). This, for example, includes processing feedback signals that are subsequently used for rate adaptation. We call this intermediate component the *switch agent*. Importantly, the switch agent is not involved in media forwarding, and none of the above-mentioned tasks require any media or feedback to be sent back from the switch agent; rather, the switch agent only receives copies of packets, analyzes them, and reconfigures the data plane if required. Its location on the switch enables a low-latency control loop.

Deployment Setting. Today, production-scale SFUs are deployed on data-center servers of cloud providers. Increasingly, these servers are connected using *programmable* top-of-the-rack switches. Besides, these servers are often equipped with SmartNICs to offload networking tasks. While Scallop is not tied to a data-center setting, data-center switches, and SmartNICs are natural and readily available deployment settings for Scallop. Going further, switches implementing Scallop can entirely replace multiple SFU servers, increasing traffic served per rack at comparable operating costs.

Support on Packet Processors (also see Appendix D). Modern programmable switches have higher capacity than SmartNICs (Tbps vs. hundreds of Gbps) and more advanced capabilities (e.g., scalable replication, register arrays) in the hardware data plane. Scallop requires three main capabilities in the hardware: deep header parsing (Appendix E), scalable replication (§6), and stateful sequence-number rewriting (§7). Consequently, Scallop's potential can be fully realized on switches such as the Intel Tofino2, which is our evaluation platform (§8). We also implement Scallop on the hardware pipeline of a representative SmartNIC (NVIDIA BlueField-3).

5 Control-Plane Prototype

Scallop's control plane must handle two key tasks: (1) establishing and managing WebRTC sessions between participants such that the SFU is inserted as a proxy between them and (2) handling the infrequent but important remaining tasks that cannot run in the data plane. The first set of tasks are handled by a centralized controller (§5.1) while the second set of tasks are handled by the switch agent, a lightweight control program on the switch CPU (§5.2-5.5). Both applications are written in C++ and communicate with each other using remote procedure calls. The switch agent can additionally exchange packets with the data plane via the switch CPU port. Figure 7 shows the resulting architecture.

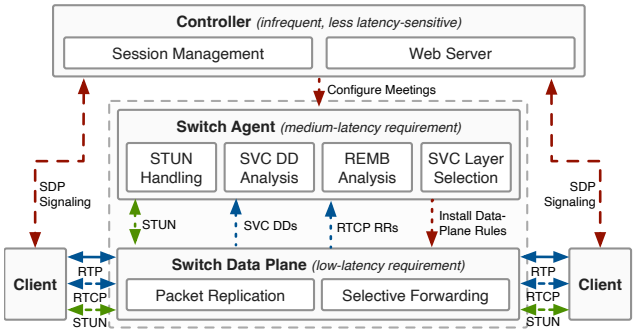


Figure 7: Scallop's 3-tiered architecture

5.1 Session and Connectivity Management

The core task of the central controller is to manage WebRTC sessions and connectivity between participants such that all media traffic is sent to and can be received from Scallop as opposed to another client, realizing our proxy architecture.

WebRTC Signaling. WebRTC uses the *Session Description Protocol (SDP)* to negotiate media-session parameters between participants, including codecs, their parameters, and IP addresses and ports [42, 51]. This negotiation, known as *signaling*, is initiated by participants whenever a new media stream is created. Scallop's controller acts as the signaling server that exchanges SDP messages between participants and maintains state about participants and their media streams to correctly configure the data plane.

Controlling Signaling to Create Proxy Topology. Each SDP message includes a list of *connection candidates*, which convey the IP addresses and ports RTP media is being sent from or can be received at. Using this information, we can *insert* the SFU as an intermediate entity between participants while appearing to meeting participants as their sole peer. Scallop achieves this by intercepting SDP messages and modifying the connection candidates on the fly.

STUN and Connectivity Management. WebRTC uses periodic *Session Traversal Utilities for NAT (STUN)* [38] packets to continuously check reachability between participants. Scallop handles STUN in the switch agent as processing STUN packets is too complex for the data plane due to their header format. This is fine since STUN packets are not classified as *latency critical*. A connection is only deemed interrupted after multiple consecutive connectivity checks fail.

5.2 Bandwidth Estimation

Rate Adaptation in SFU Architectures. Continuous and timely bandwidth estimation and rate adaptation are critical tasks in video conferencing as trying to send media at a higher bitrate than the network can support rapidly leads to high latency and, ultimately, loss, severely degrading QoE. When an SFU is used, this task is split into two parts: (1) each sender sends at the highest rate any of the receivers can receive at (§5.2-§5.3), and (2) for receivers that can only support lower rates than the highest rate, the SFU scales down the stream by selectively dropping packets (§5.4). WebRTC uses Google Congestion Control (GCC) [3] to estimate link capacity, which is then used to adjust media bitrate.

GCC Modes. GCC can operate in two ways. In sender-driven mode, the sender computes available bandwidth based on *frequent feedback* from receivers. Scallop uses receiver-driven mode where each receiver independently estimates bandwidth based on packet arrival-time variation and then *periodically* informs the sender of the new bandwidth estimate using *Receiver-Estimated Bandwidth (REMB)* messages. Although REMB messages are still too complicated for the data plane to process, the frequency of them is proportional to the frequency of the link capacity changing, unlike in sender-driven mode, where the receiver sends a message every 10-20 media packets. Scallop intercepts REMB messages and adapts media bit rates based on the reported estimates.

5.3 Preserving Feedback Semantics

Bandwidth Estimation in a Proxy Architecture. Realizing correct bandwidth estimation in our proxy-based Scallop architecture, however, is challenging. This is because forwarding feedback among participants, rather than using individual control loops (as in a split proxy), mixes signals, incorporating uplink measurements from all senders instead of reflecting a specific downlink. As a result, feedback converges on the lowest-bandwidth receiver, forcing the sender to lower the media bitrate unnecessarily for all participants.

Mixed Feedback Signals. To illustrate the problem, consider a 3-party meeting as depicted in Figure 8. Participant 1 (P_1, blue) and 2 (P_2, red) share video while participant 3 (P_3) only receives. Solid lines depict media streams and dashed lines depict Real-Time Transport Control Protocol (RTCP) feedback messages. Done naively, two problems arise. First, RTCP combines feedback messages concerning multiple media streams into a single RTCP packet which is too complex to parse, analyze, and correctly forward in the data plane. Second, they also mix feedback from different media streams in a way that feedback is not actionable anymore. For example, the bandwidth estimated by P_3 now is based on feedback from both P_1's and P_2's uplinks (to SFU) in addition to P_3's downlink (from SFU). This estimate is irrelevant for the SFU since it cannot do anything about uplink capacity. More importantly, it leads to the problem that when naively forwarding the combination of all feedback to P_1, P_1's media will be encoded at a bitrate adapted to P_2's uplink performance. Put differently, all send rates will converge to the lowest capacity of all uplinks in the meeting which defeats the purpose of even having rate adaptation in the SFU.

Solution Part 1: Split Connections. We solve these problems using two techniques (Figure 8). First, we split each WebRTC UDP stream (the scope for bandwidth estimation) by participant, such that every receiver's RTCP feedback pertains to exactly one sender. This eliminates the need to parse and filter combined RTCP messages carrying reports for multiple streams—Scallop simply forwards each receiver's feedback directly to its corresponding sender.

Solution Part 2: Selectively Forward Feedback. In our example, P_3 now has a dedicated stream to receive media from P_1 and a second stream to receive media from P_2. Consequently, P_3's REMB messages only include information about the path between P_3 and P_1 or P_2, respectively, and not about the combination of all paths. A filter function f at the SFU selects the best-performing downlink per sender and configures the data plane to only forward these messages to the respective sender. The selection is done by computing an EWMA over each receiver's bandwidth estimates and periodically selecting the maximum out of the EWMAs. The rationale for this is that each sender should send at the highest rate allowed by its uplink and the best downlink. As all packets traverse the sender's uplink, its performance is inherently accounted for in the feedback. The SFU then handles adaptation for lower-bandwidth downlinks, explained next.

5.4 SVC Analysis and Layer Selection

Scalable Video Coding (SVC). Scallop leverages *SVC* [52] with the AV1 codec's L_1T_3 profile [12, 54] to adapt media streams to network capacity. This allows the SFU to choose among three temporal layers (frame rates) for video streams. While the data plane handles the actual forwarding, the control plane decides the quality level to send to each participant. The SVC structure can change with each key frame (sent when a stream starts or the resolution changes), requiring the SFU to adapt the data plane's forwarding rules accordingly. Figure 9 shows an L_1T_3 SVC stream's dependencies.

AV1 Extension and Dependency Descriptor. In AV1, each RTP packet contains an RTP AV1 extension header indicating its layer through a unique *template id*. Key frames additionally contain a *dependency descriptor* that carries the semantics of the SVC structure [54]. In our example, template ids 0 and 1 represent the base layer (7.5 fps), id 2 the first enhancement layer (15 fps), and ids 3 and 4 the second enhancement layer (30 fps). Dropping frame ids 3 and 4 would reduce the frame rate from 30 fps to 15 fps. The data plane parses the AV1 extension header (primarily its template id) to decide whether the packet should be dropped or not. Appendix E describes the parsing process and its associated challenges.

Selecting a Quality Layer. Parsing the dependency descriptor is beyond the data plane's capabilities. Thus, Scallop sends key frames to the control plane for analysis. Whenever a bandwidth estimate (i.e., an REMB message) is received, the switch agent invokes a function that can be defined by adopters of Scallop. This function is declared as follows:

`selectDecodeTarget(currDT, estHist, newEst)` → `newDT`.

The function takes as input the current decode target (currDT), a history of past estimates (estHist), and the new estimate (newEst) from the REMB message; it returns the new decode target. If the returned decode target is different from the previous one, the switch agent reconfigures the data plane. Importantly, while we implemented a simple heuristic that switches between quality levels based on fixed capacity-estimate thresholds, using this model, arbitrary rate-adaptation algorithms can be implemented.

5.5 Handling other RTCP Messages

Besides REMB, additional feedback messages are delivered through RTCP. From the receiver side, negative acknowledgments (NACKs) request the retx. of a specific media packet, and picture-loss indication messages (PLI) notify the sender to send an intra-coded video frame. NACK and PLI messages are sent from a receiver experiencing loss to the respective sender (Figure 10). While PLIs, NACKs, and REMB messages are forwarded to their intended destinations through the data plane without delay, the data plane also creates copies of them on-the-fly, sending them to the switch CPU for further analysis (e.g., for the filter function described earlier).

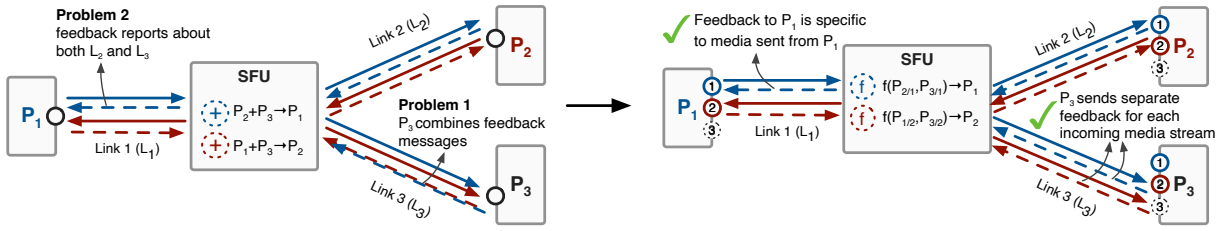


Figure 8: Splitting WebRTC connections per participant and forwarding feedback messages of the best-performing downlink only preserves feedback semantics and ensures effective rate adaptation.

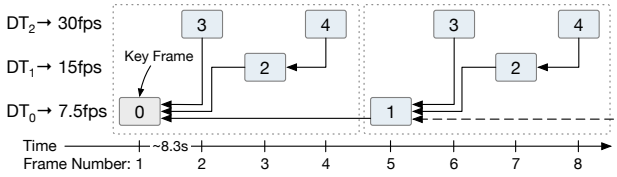


Figure 9: Frame dependencies in AV1 L_1T_3 SVC.

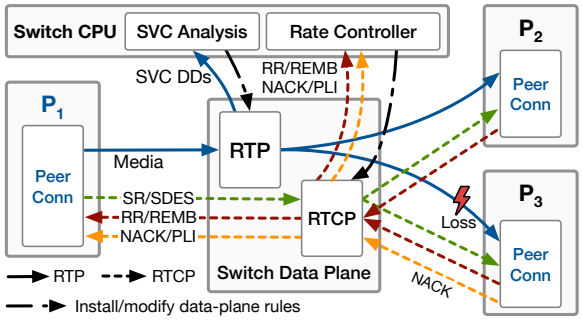


Figure 10: Flow of all types of media and control packets in a Scallop 3-party conference.

Algorithm 1: General Packet Replication in Scallop (RA-driven logic highlighted in green)

```
1  Sender2Meeting: Map{Participant: Meeting}
2  Meeting2Participants: Map{{Meeting, ReplicaID}: Participant}
3  Template2Layer: Map{{Meeting, Template}: Layer}
4  ExcludedReceivers: Map{{Meeting, Layer, Participant}: Bool}
5  ExcludedSenderReceivers: Map{{Meeting, Layer, Participant, Participant}:
   Bool}
6  Function scallop_replicate (Packet pkt):
7      Participant sender := {pkt.ip.src, pkt.udp.src, pkt.rtp.ssrc}
8      Meeting meeting := Sender2Meeting[sender]
9      Layer layer := Template2Layer[{meeting, pkt.av1.template_id}]
10     ReplicaID replica_id := 1
11     while Meeting2Participants[meeting, replica_id].hit do
12         receiver := Meeting2Participants[meeting, replica_id]
13         if receiver != sender then
14             if ExcludedReceivers[{meeting, layer, receiver}].miss &&
               ExcludedSenderReceivers[{meeting, layer, sender,
               receiver}].miss then
15                 Packet pkt2 := replicate(pkt)
16                 send_packet_to(pkt2, receiver)
17         replica_id := replica_id + 1
```

6 Scalable Media Replication

Background and Challenges. Scallop must enable scalable replication of media packets in hardware to support thousands of simultaneous meetings. This entails two core challenges: First, for each meeting, Scallop stores both the list of active participants and each participant's rate-adaptation status. On every incoming media packet, this state determines which downstream recipients receive a replica. Since memory is limited on packet-processing hardware, supporting a large number of concurrent meetings becomes difficult. Second, once the recipients are known, Scallop must generate and forward replicas to them efficiently. Although modern packet-processing platforms offer hardware-assisted replication, their features vary widely, and no prior work demonstrates how to map SFU packet replication onto them. Scallop addresses these challenges in two parts. First, we introduce a general, memory-efficient algorithm for SFU replication on hardware packet-processing platforms. Second, we adapt and optimize this algorithm for two representative platforms—programmable switches (e.g., Intel Tofino2) and SmartNICs (e.g., NVIDIA BlueField-3).

6.1 General, Memory-Efficient Replication

For our platform-agnostic design, we rely on three basic primitives available on all programmable packet-processing platforms: match-action (MA) tables populated by the control plane (in Scallop, the switch agent), a replication (mirroring) function that produces a single copy of an input packet, and a recirculation function that feeds packets back into the ingress pipeline for further iterations. Our replication algorithm then proceeds in two parts: (1) *Participant-driven*, which replicates each packet according to the meeting-to-participants mapping, and (2) *Rate Adaptation (RA)-driven*, which excludes replication for a subset of rate-adapted participants.

Solution Part 1: Participant-Driven Replication. Consider a meeting M with N participants. Scallop must replicate each incoming packet from any participant (sender) to the other $N-1$ participants (receivers). In a naive approach, we could assign a separate *replica ID* (RID) to every sender-receiver pair in M and store them as MA-table records of the form: key={sender, RID}, value={receiver}. When a packet arrives from a sender, we create a replica for its receiver with RID=1, recirculate the packet and create a replica for its receiver with RID=2, and so on. While this technique works, it requires $N(N-1)$ or $O(N^2)$ records per meeting. A more efficient strategy (outlined in Algorithm 1) is to assign a global replica ID

to each participant in M (line 2). This reduces the memory requirement to N records ($O(N)$). However, now when Scallop iterates over the replica IDs (lines 1, 7-8), it would encounter the sender itself. Fortunately, the sender is trivially known from the packet headers (line 7), so we skip that iteration (line 13). Once created, the replica is addressed to the receiver's IP and UDP port, and forwarded (line 16). Since this logic is sufficient for meetings not subject to RA, we also call it the *non-rate-adapted (NRA)* technique.

Solution Part 2: RA-Driven Replication. In some meetings, individual receivers might have lower receive bandwidth than other participants, requiring RA. To support RA, Scallop must replicate only the necessary quality layers for each affected receiver. For example, if a receiver can handle only 7.5 fps instead of the full 30 fps, Scallop should replicate the base layer but omit the two enhancement layers (see §5.4). Scallop identifies each packet's quality layer by looking up its AV1 *template ID* in an MA table, which uses constant memory per meeting (lines 3, 9). Once the layer is known, Scallop must determine which participants should receive replicas. Because RA is often not required, we record only the *excluded* receivers rather than the included ones. Exclusions fall into two categories: (1) *Receiver-driven RA (RA-R)*, where a receiver receives the same reduced quality (e.g., 7.5 fps) from all senders, and (2) *Sender–Receiver-driven RA (RA-SR)*, where a receiver receives different qualities from different senders. RA-SR happens when some senders send media at a higher bitrate than others. Scallop implements exclusions using two disjoint MA tables for RA-R and RA-SR (lines 4-5). In the worst case, RA-R consumes $O(N)$ records and RA-SR consumes $O(N^2)$, respectively, although actual usage is much lower. During each iteration, Scallop checks both tables and skips any excluded receiver before replicating the packet (line 14).

Optimizing Bandwidth and Latency. Our general solution runs entirely in hardware and is memory-efficient—a significant improvement over software SFUs. However, the *while* loop in Algorithm 1 (lines 11–17) has two drawbacks. First, it recirculates the original packet $N-1$ times, consuming recirculation capacity. Second, it produces the final replica only after $N-1$ recirculations, incurring $O(N)$ latency. Fortunately, some packet-processing platforms, such as Intel Tofino and Juniper Trio [60] programmable switches, provide hardware primitives to generate hundreds of packet replicas at once. In §6.2, we describe how we leverage one such platform (Intel Tofino2) to address both drawbacks. On platforms without bulk-replication support (e.g., NVIDIA BlueField-3), we replace the while loop with a recursive, perfect binary-tree routine that produces all replicas within $\lceil \log_2 N \rceil$ iterations and therefore a uniform $O(\log N)$ latency (§6.3).

6.2 Scalable replication on the Tofino2

Background. The Tofino *Packet Replication Engine* (PRE) is a specialized hardware module designed for efficient multicast through a hierarchical, three-level structure called a *multicast group* or a *multicast tree* (Figure 11). The PRE sits in the *Traffic Manager* between the ingress and the egress pipelines of the switch, allowing the ingress pipeline to invoke replication on a packet, the PRE to perform the replication, and the egress pipeline to process each replica before forwarding it. The control plane configures the PRE at runtime in three steps: (1) Allocate level-2 (L2) nodes, each associated with one egress port ; (2) Allocate level-1 (L1) nodes, each identified by a *node ID* (unique across the PRE), a *replica ID* or RID (unique across a multicast tree), and a *set* of allocated L2 nodes; (3) Create multicast trees, each with a unique *multicast group ID* (MGID) and an associated set of allocated L1 nodes.

Replication: The ingress pipeline sets a packet's *mgid* metadata field to map it to an existing multicast tree. When the packet arrives at the *root* of that tree, the PRE stores the packet in a buffer and creates a *pointer*. Then, it replicates this pointer to L1 nodes, and further to L2 nodes. At each L2 node, a replica is created from the pointer, attached to the associated egress port, and forwarded to the egress pipeline.

Pruning: The PRE also supports branch-*pruning*. The control plane configures pruning by assigning an L1 XID to each L1 node, and an L2 XID to each egress port. The data plane's ingress invokes pruning by setting the packet's *l1_xid* (for L1-pruning), and *rid* and *l2_xid* (for L2-pruning) metadata.

Opportunity and Challenges. Due to its specialized design, the PRE neither requires recirculations nor does it pass around packets. Consequently, it can replicate a packet hundreds of times with minimal latency. Scallop can benefit greatly from the PRE, but it must address two challenges. First, mapping VCA entities (meetings, senders, receivers) onto the PRE's tree hierarchy (root, L1 nodes, L2 nodes) is not obvious. No prior work has used the PRE (on Tofino or similar switches) for such purposes, requiring us to explore the design space from the ground up. Second, the PRE comes with resource constraints. While it can support up to 2^{24} L1 nodes, 2^{16} RIDs per tree, and all of the switch's egress ports per L1 node, it can only support $\mathcal{T}$=64K multicast trees. Furthermore, only one L1-XID and L2-XID can be set per packet, limiting pruning flexibility. Scallop must adapt its solution to the PRE and maximize its utilization under these constraints.

Two-Party Meetings. Meetings with only two participants (60% in our campus dataset) do not require replication. Therefore, we do not allocate multicast trees for them thus saving PRE resources.

NRA Design. For meetings with N>2 and not requiring RA, Scallop aggregates all N meeting participants into a single replication tree (Fig. 12a). The root represents the entire meeting, each L1 node represents a participant and each L2 node maps a participant to their egress port. This design supports $\mathcal{T}$ concurrent meetings and consumes $O(N)$ L1 nodes. We use L2-pruning to prevent the sender from receiving its own packet. To further increase efficiency, Scallop aggregates multiple (m) meetings into a single replication tree (Fig. 12b), supporting $m\mathcal{T}$ concurrent meetings. However, in addition to L2-pruning, this approach requires L1-pruning to ensure that packets of one meeting are not received by participants of another. For example, in Figure 12b, replication to $M2$'s participants is suppressed for a packet from $M1$, and vice-versa. The PRE's pruning limitations force m to be 2. Scallop can support 128K concurrent meetings using this design (§8.2).

RA Design. Since both L1- and L2-pruning are consumed in the NRA design, we need alternatives to handle RA. For RA-R, Scallop creates one tree (following NRA design) per SVC layer. When a rate-adapted receiver should not receive a particular layer, it is removed from that layer's tree. With q=3 SVC layers, Scallop can support up to $m\mathcal{T}/q$ concurrent meetings, which evaluates to 42.7K (§8.2). For

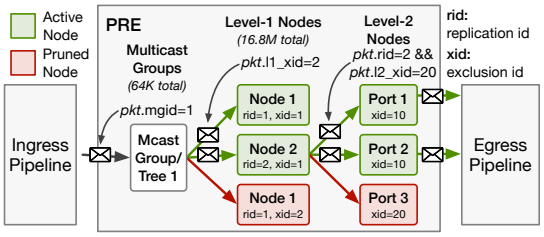


Figure 11: Tofino's Packet Replication Engine (PRE)

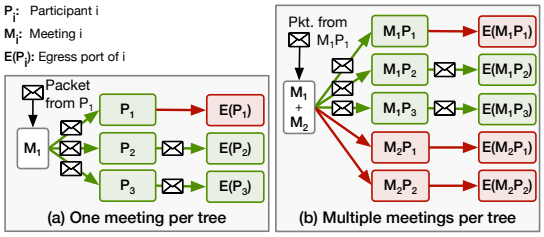


Figure 12: Constructing efficient replication trees by aggregating meetings and using dynamic pruning

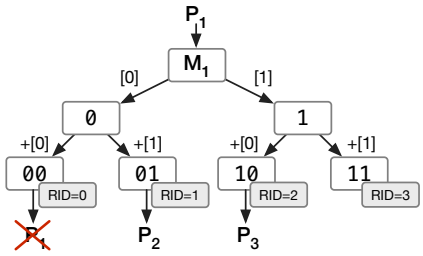


Figure 13: Packet Replication using Perfect Binary Tree

RA-SR, Scallop cannot do better than aggregating two senders (and their receivers) per SVC layer into a single tree. Scallop consumes $2\mathcal{T}$ L1 nodes to support $2\mathcal{T}/qN$ concurrent RA-SR meetings, which evaluates to 4.3K, compared to 192 on a 32-core server (assuming 10-party meetings, all sending video and audio).

Dynamic Migration across Designs. Since RA (especially RA-SR) is not always required for a meeting, Scallop dynamically and seamlessly migrates each meeting across two-party, NRA, RA-R, and RA-SR designs as needed. This maximizes the PRE's utilization, ensuring the bottleneck remains the switch's bandwidth rather than its replication capacity, whenever possible.

6.3 Scalable Replication on the Bluefield-3

Unlike Tofino2, some packet-processing platforms, such as the BlueField-3 SmartNIC, cannot replicate in bulk and require recirculation to create multiple replicas. However, we can improve the replication latency on such platforms using a perfect binary tree design (as shown in Figure 13). During meeting setup, the switch agent populates three MA tables: one mapping each participant to its meeting ID, another mapping every (meeting ID, replica ID) pair back to a participant, and a third recording, for each meeting, the *maximum tree depth* $\lceil \log_2 N \rceil$ (where N is the number of participants). This depth ensures the tree will have at least N leaves.

Figure 13 shows an example meeting with N=3 (and so, maximum depth = 2). When a media packet from sender P_1 arrives in the data plane, we encode two fields in its header: a running replica ID (RID) and the current depth, both initialized to zero. The SmartNIC hardware mirrors the packet, then recirculates both copies. Conceptually, one copy traverses the left subtree and the other the right. Each time a mirrored packet returns, we append the bit value 0 (for left) or 1 (for right) to its RID, increment its depth, mirror it again, and recirculate. This process repeats until the packet's depth equals the precomputed maximum depth for that meeting.

At the end, each packet carries a full-length RID. In our example, we end up with RIDs 0 (00), 1 (01), 2 (10), and 3 (11). RID 0 corresponds to the sender, and RID 3 has no matching participant, so both are dropped. The remaining two replicas with RIDs 1 and 2 map to P_2 and P_3 respectively, and are forwarded accordingly. We then layer our RA schemes (RA-R and RA-SR) on top of this replication mechanism, following the logic described in Algorithm 1. This method can be further optimized by preemptively checking whether a downstream subtree will yield valid replicas and skipping its mirroring if not; we leave this as future work.

7 Transparent Rate Adaptation

Background and Challenges. Under WebRTC's P2P model in a split-proxy architecture, each receiver expects a continuous, unmodified media stream from the SFU. However, in our true-proxy architecture, Scallop performs RA by suppressing packets associated with a specific quality layer to match network conditions. If not handled explicitly, this suppression would create gaps in RTP sequence numbers, which receivers would interpret as packet losses, triggering unnecessary retransmissions (retxs.). To preserve a continuous RTP packet stream, Scallop rewrites sequence numbers after replication. However, rewriting sequence numbers *perfectly*—when intentional suppression coincides with network-induced loss and reordering—is impossible without buffering packets.

Naive Solution. To illustrate, consider the example in Figure 14. Here, the sender sends packets with sequence numbers 1 to 5. Packets 1, 4, and 5 are base-layer packets while packets 2 and 3 are enhancement-layer packets to be suppressed. In a straw-man design (Fig. 14a), Scallop would simply increment a counter every time it sends out a packet to a receiver. This approach would preserve a continuous stream of sequence numbers and the receiver would not trigger negative acknowledgments (NACKs) for missing packets, thus avoiding retx. of intentionally suppressed packets. This design, however, would also mask network-induced packet losses, leading to the receiver's decoder state breaking and the video freezing until a picture-loss indication (PLI) to reset the decoder is triggered (Fig. 14b). This situation, which we refer to as an error of *type 1*, would lead to an unacceptable stall for the affected receiver.

Acceptable Heuristic Error. Instead, we design a hardware-amenable heuristic that uses a combination of per-meeting, per-stream, and per-packet data to infer whether a sequence number skipped due to a loss corresponds to a packet that was supposed to be suppressed

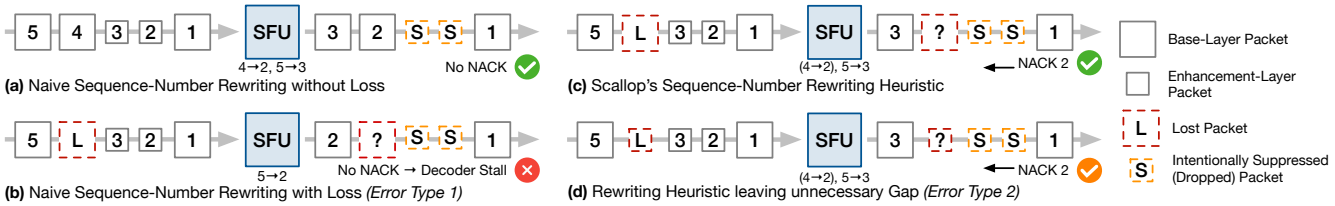


Figure 14: Example of sequence-number rewriting and associated error types in Scallop

anyway. If so, Scallop increments the sequence number by one. If the packet belongs to a base layer (required for the decoder), Scallop will preserve the sequence gap to trigger a NACK and retx. (Fig. 14c). There are, however, cases where *no* online algorithm for this problem can exactly determine which layer the packet with the skipped sequence number belongs to. Consider the last case (Fig. 14d) where packet 4 carries the enhancement layer which is supposed to be suppressed. Assuming Scallop cannot infer that 4 is a candidate for suppression, it preserves the sequence-number gap. This leads to a NACK for packet 2 (originally packet 4), causing its retx. which is unnecessary. Our experiments show that, while this retx. wastes some bandwidth, it does not cause any QoE degradation. Therefore, this error, which we call error *type 2*, is acceptable but slightly expensive. We design our heuristic to *never* commit type-1 errors, and to *minimize* type-2 errors. Figure 27 in Appendix F shows that the overhead of unnecessary retxs. from type-2 errors grows with the loss rate but is low even at significant loss (e.g., under 5% overhead at 10% loss).

Heuristic Design. To rewrite sequence numbers correctly, Scallop must determine an accurate *offset* (difference between original and rewritten sequence numbers) for each replicated packet in each rate-adapted sender-receiver media stream. This is challenging because the rewriting must be done post-replication, i.e., in the egress pipeline, where suppressed packets are never "seen". The task is further complicated by network loss and reordering, as described above. Our main insight is that the offset's accuracy depends on how accurately we reconstruct the state of the media frame the replica belongs to, and the one immediately preceding it. For this, we rely on (1) per-packet state: current frame number, sequence number, frame start and end markers from headers; (2) per-stream state: the offset, max. frame number seen, max. sequence number seen of the max. frame, a flag indicating whether the max. frame's end marker was seen, the *cadence* of frames suppressed by RA (e.g., every second frame) populated by the control plane, and a history of the offsets for the last few unsuppressed frames. For each incoming packet, first, we check whether the offset should be updated according to the following cases, and then we rewrite its sequence number based on the offset:

(1) If current frame equals max. frame, don't update offset;
(2) If current frame succeeds max. frame, add to the offset the best estimate of the no. of suppressed packets in between;
(3) If current frame precedes max. frame (late reordered packet), rewrite using offset in history if found, else drop.

Accuracy vs. Memory Efficiency. If the loss (and reordering) rate is low, a short history of offsets suffices; otherwise, we need a longer history. Thus, there is a trade-off between accuracy (i.e., unnecessary retxs.) and per-stream memory, with the optimal balance dictated by the loss rate. We implement several variants of our heuristic to explore the design space. Below, we describe two:

Seq. Rewriting-Low Memory (S-LM) stores no history of frames beyond the max. frame, minimizing memory usage.

Seq. Rewriting-Low Retx. (S-LR) stores the history of three additional frames to accommodate 150 ms of suppressed (15 fps) video, the max. recommended RTC end-to-end delay [23]. This reduces unnecessary retxs. under higher loss rates.

We discuss the Tofino2-based prototypes of these variants and evaluate their modest overhead in Appendix F.

Proto./Type	Packets	Pct.	Per sec.	KBytes	Pct.
RTP	170,870	94.5	284.30	166,762	99.47
- Audio	29,746	16.46	49.49	3,826	2.28
- Video	141,124	78.09	234.81	162,935	97.19
- AV1 DS*	5	⋘	0.008	6	⋘
RTCP	9,153	5.06	15.22	801	0.48
- SR/SDES	3,456	1.91	5.75	304	0.18
- RR*	240	0.39	0.13	15	0.01
- RR/REMB*	5,457	3.02	9.07	482	0.29
STUN*	695	0.38	1.15	89	0.05
Ctrl. Plane	6397	**3.54**	10.64	593	**0.35**
Data Plane	174,326	**96.46**	290.06	167,066	**99.65**
Total	180,718	100	300.69	167,653	100

Table 1: Packets per participant sent to SFU (10 min.)

8 Evaluation

Experimental Setup. We deploy the data plane of Scallop on an 12.8 Tbit/s Intel Tofino2 hardware switch. The switch agent runs on the CPU of this switch, an 8-core Intel Pentium with 8GB of RAM. The controller is deployed on a 40-core Intel Xeon server with 96 GB memory. We use another server with the same configuration for both Scallop and MediaSoup (where applicable) clients.

8.1 Control Plane

We first analyze the amount of packets and bytes that our controller needs to process compared with the amount of packets that stay entirely in the data plane. This ratio demonstrates the feasibility of Scallop's control/data-plane split. We collect a packet-level trace of a real three-party meeting using Scallop where participants send audio and 720p AV1 SVC video. We then determine the number of packets and bytes that can be processed in the data plane. The experiment ran for ten minutes with a total of 180,718 packets.

Table 1 shows the results. 94.5% of these packets were RTP packets which can be handled in the data plane with the exception of five RTP packets containing an AV1 dependency descriptor. RTCP accounted for 5.06% of all packets and 0.48% of all bytes, out of which

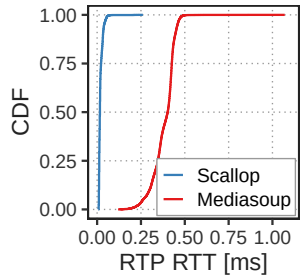
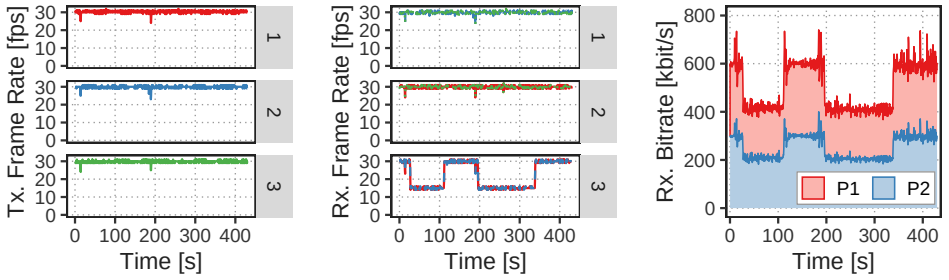


Figure 15: Forwarding Latency in Scallop



(a) Send Frame Rate (b) Receive Frame Rate (c) Receive Bit Rate at P3

Figure 16: Scallop Rate Adaptation: Part. 3's receive bit rate is reduced twice.

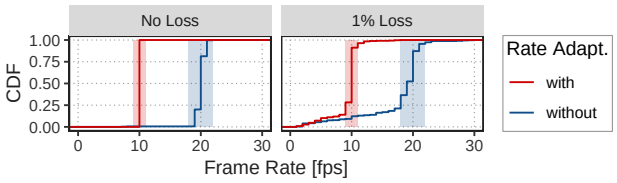


Figure 17: Frame rate under loss and rate adaptation

Resource type	Scaling behavior	Usage under peak campus load (avg.)
Parsing depth	Constant	Ing. 27, Eg. 7
No. of stages	Constant	Ing. 7, Eg. 5
PHV containers	Constant	17.9%
Exact xbars	Constant	5.66%
Ternary xbars	Constant	2.52%
Hash bits	Constant	4.62%
Hash dist. units	Constant	6.94%
VLIW instr.	Constant	7.29%
Logical table ID	Constant	21.87%
SRAM	Linear	6.77%
TCAM	Linear	1.38%
Egress Tput.	Quadratic	1.2 Gb/s

Table 2: Resource usage of the Tofino2 hardware prototype.

our switch agent uses RTCP receiver reports and REMB messages to control the RA logic. These packets accounted for 3.41% of overall packets. Finally, STUN packets accounted for 0.38% of all packets and also need to be processed in software. In summary, 96.46% of all packets and 99.65% of all bytes can be processed in the data plane, showing that this workload is well-suited for a control/data-plane split. More importantly, the remaining packets, except for a few STUN packets at the start of a session, are not blocking.

8.2 Data Plane

Tofino Resource Utilization. We implement Scallop's data plane using ~2000 lines of $P4_{16}$ code on the Tofino2. Table 2 summarizes the resource utilization of the Tofino2. We categorize resource types by how they scale with the number of concurrent participants supported by Scallop—constant, linear, or quadratic. For the non-constant types, we report the average utilization under peak campus traffic load, as observed in our campus dataset. The high ingress parsing depth is evidence of the deep, flexible parsing required to extract the SVC quality layer information, as we describe in Appendix E. The number of stages consumed by Scallop falls squarely within the bounds of the maximum available on the Tofino2. For the rest of the components with constant scaling behavior, we report the average utilization across all stages. This analysis shows that Scallop's resource usage is low enough such that other network applications can be supported simultaneously on the switch.

8.3 Latency and Impact on Session Quality

Latency. We demonstrate that leveraging a hardware-based data plane significantly reduces SFU-induced delay. This is shown by comparing per-packet RTTs of RTP media packets in a two-party call. Two participants are connected either through Scallop's Tofino or the Mediasoup's SFU server. Figure 15 shows that Scallop achieves 26.8× lower median latency while cutting 99%ile latency by 8.5×.

Session Quality during Rate Adaptation. To show that Scallop is faithful to the core SFU functionality, we validate that our SVC-based rate adaptation effectively reduces bit rate without causing freezes or other QoE degradation. We conduct two experiments. First, we start a three-party call, where all participants send and receive video. We collect WebRTC performance statistics [58] from Google Chrome, including receive frame rate, stall time, video resolution, and more; they correspond to the media stream after decoding and, thus, are an accurate representation of actual playback quality. Figure 16 confirms that Scallop successfully reduces the frame rate from 30 to 15 fps for the bandwidth-constrained participant 3 while maintaining a decodable media stream without incurring otherwise lower QoE (e.g., via freezes). Second, we conduct four 5-minute experiments covering all combinations of 1% packet loss and rate adaptation (RA). Figure 17 shows that under loss, even without rate adaptation (and sequence-number rewriting), we see many samples of low frame rates (i.e., short freezes). This is consistent with prior observations and ITU recommendations [17, 22]. Scallop's RA does not change the shape of the distribution but moves the median frame rate, here, from 20 fps to 10 fps as intended. In fact, the fraction of fps readings that fall out of a 10% margin of the target frame rate (shaded areas) is slightly lower under RA (14% vs. 21%), showing that Scallop's RA does not further reduce QoE, even under loss.

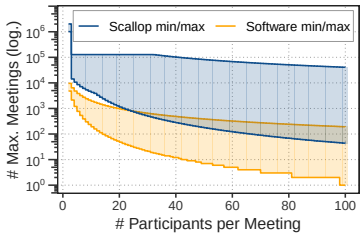


Figure 18: Best-case and worst-case Performance

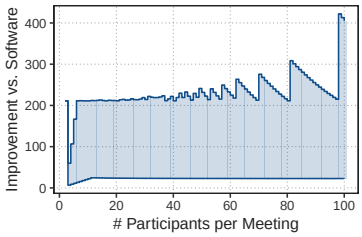


Figure 19: Scallop Scalability Gain over Software

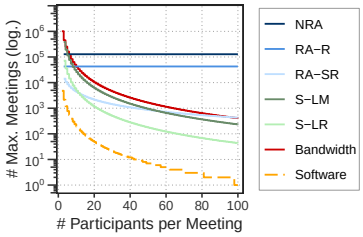


Figure 20: Worst-case Perf. by Meeting Config. & Impl. Choice vs. Software

8.4 Scalability

First, we perform a faithful simulation of Scallop's data-plane components and subject them to a wide range of meeting configurations (described below) and loss rates. Second, we record MediaSoup's performance across experiments on varying workloads (similar to §2.2) and extrapolate it to the same configurations on a 32-core server. Finally, for each configuration, we compare the two systems.

Best-case vs. Worst-case Performance. We exhaustively simulate all possible combinations of loss rates (0 to 5% in 0.1% increments), number of participants (N) per meeting (1 to 100), number of senders per meeting (1 to N), the RA status per sender-receiver pair and its associated media quality level (low@1, medium@1.5, and high@3 Mbps, determined from experiments), replication-tree designs (NRA, RA-R, RA-SR), and the sequence-number rewriting heuristic used (S-LM vs. S-LR). For each configuration, we compute the min. and max. number of meetings supported by Scallop and MediaSoup (Software), respectively. In Figure 18, we show the range between Software/min and Software/max in orange, and that between Scallop/min and Scallop/max in blue. For any given N (x-axis), Scallop/min is always higher than Software/min and Scallop/max is always higher than Software/max. This shows that Scallop always supports many more meetings than software irrespective of the configuration. For N=2, Scallop/max is massive since the PRE is not needed. Thereafter, the numbers are determined by the complex interplay among the simulation configurations. For example, with N=3, without RA, Scallop supports 3× more meetings with just one sender vs. three senders, since the bandwidth used is only 9 Mbps in the former case (3 streams × 3 Mbps) but 27 Mbps (9 streams × 3 Mbps) in the latter. However, if RA results in low-quality media in the latter case, the bandwidth reduces to 9 Mbps. We observe that in most cases, the Scallop/min corresponds to the memory bottleneck encountered when S-LR is used.

Scalability Improvement over Software. For each configuration, we compute the ratio between the number of meetings supported by Scallop and Mediasoup. Figure 19 shows the lowest and highest such ratio per N with the intermediate range shaded. The range shows that Scallop can support between 7-422× more meetings than software. Toward the right of the plot, the highest ratio shows a sawtooth pattern because the denominator is small yet discrete.

Software vs. Scallop Bottlenecks. The different Scallop designs (NRA/RA-R/RA-SR and S-LM/S-LR) have varying trade-offs. Figure 20 shows Scallop's performance assuming the respective design was the bottleneck with all participants sending media. For example, NRA supports more meetings than RA-R, but both are constant, whereas RA-SR supports fewer meetings and decreases with N. The overall performance corresponds to the minimum of all the designs. We plot the performance of MediaSoup (orange) for comparison.

9 Discussion

9.1 Making WebRTC Hardware-Amenable

WebRTC is a widely adopted framework that enables browsers to establish secure and efficient real-time multimedia sessions, even across NATs and firewalls. Originally designed for peer-to-peer communication between two endpoints, its limitations become apparent in multi-party use cases where SFUs become required (§2). These limitations are associated with the way WebRTC enables secure communication and, secondly, with the design of its wire protocols. We explain both aspects next before making recommendations on how to modify WebRTC for hardware-friendliness.

Encryption and message authentication. In WebRTC, RTP headers are HMAC-protected while payloads are also AES-encrypted via SRTP-DTLS [44]. In Scallop, neither mechanism is currently implemented. The main challenge is key distribution: WebRTC's split-proxy design uses a separate key for each P2P connection, exchanged between SFU and client. This is incompatible with our proxy redesign, where a single sender's packets are replicated to many receivers, necessitating one-to-many key distribution, for example, via centrally distributed keys (as previously done in WebRTC via SDES [7] and done today in Zoom [20]). Since the SFU does not need to touch payloads, it would then operate on encrypted packets. Unlike in WebRTC's current encryption mechanism and key-distribution scheme, doing so would also enable the use of end-to-end encryption (E2EE). Prior work demonstrates that ciphers (e.g., AES) and cryptographic hashes (e.g., SipHash) can be computed on programmable data-plane devices [5, 46, 48, 49, 59, 61], and modern SmartNICs provide even more capabilities [27, 29, 53]. Rewriting header fields requires recomputing HMACs over the short RTP header, which is feasible in programmable hardware.

Hardware-amenable wire protocols. The WebRTC framework and its protocols, which most video-conferencing applications follow, are not hardware-friendly. This is because the protocol was initially designed as a P2P protocol, where software with complex algorithms is designed to run on the end hosts. Furthermore, codec and protocol designers emphasize efficiency, making every bit count and aiming to minimize the bandwidth usage between participants. This is a noble goal, but it makes the protocol inherently challenging to parse and process in hardware (e.g., network devices).

WebRTC design considerations for scalability. Designing SFUs as true proxies for WebRTC adds significant overheads (§3) while a more efficient and hardware-friendly use as demonstrated in this paper, comes with some limitations outlined above or requires some added complexity in managing feedback messages (§5.3) or realizing rate adaptation (§7). We find that none of these limitations are fundamental to WebRTC and argue that WebRTC already provides the necessary building blocks to support more hardware-amenable, lightweight SFU designs. With minor modifications WebRTC can support SFUs that are better suited for hardware offload:

- As outlined above, centralized key distribution would free the SFU from having to decrypt and re-encrypt packets, while still allowing for standard encryption and message authentication mechanisms. If end-to-end encryption of payloads is desired, a double encryption scheme with keys directly exchanged among clients can be used [19, 57]. This approach would allow the SFU to operate on encrypted payloads such that the SFU does not need to be trusted with payloads, which can be desirable.
- If RTCP did not use packets that combine reports across multiple streams, the SFU could process and forward RTCP packets in hardware more easily without needing to parse deep into packets or splitting packets in the data plane.
- If the SFU could indicate via the AV1 dependency descriptor which decode target should currently be active for a given stream, the receiver could ignore all packets (even in the presence of sequence skips) not part of the current decode target. This would eliminate the need for sequence rewriting.
- Compression and dynamic variable-length fields can be difficult to handle in hardware. Following the trend of hardware offloading and co-designing software and hardware, it is worth re-exploring the design and implementation of more hardware-amenable protocols (e.g., for AV1 and RTCP), shifting the balance towards offloading computation and less on bandwidth optimization. In fact, the community has seen ideas in the past, for example using fixed-length header fields in BGP routing [28].

9.2 Scallop for Commercial Deployment

Extended VCA Use Cases. Video-conferencing applications often implement additional features that need direct media access; these include live transcription and visual effects (e.g., virtual backgrounds or "funny hats") [8]. In most cases, including Zoom, such functions are implemented at SFUs, and this requires decryption of client's media stream at the SFU. Because Scallop operates on encrypted and encoded media, these features cannot be implemented in Scallop's data plane. We argue, however, that these features should be implemented at clients, not SFUs, to preserve end-to-end encryption. WebRTC provides APIs for this [10, 57].

SFU cascading. SFU cascading is a technique of deploying SFUs in a hierarchical manner to improve the scalability of VCA infrastructure by aggregation of media streams at higher-level SFUs. We argue our system is not an alternative to or a competing solution with SFU cascading. In fact, the approaches are independent and could be combined to improve the scalability further. Our control/data plane split has the potential to simplify deploying many SFU data planes under the management of a single controller. Our current system is already designed in this way and would provide the architectural framework to enable such SFU topologies.

10 Related Work

Studies on Video-Conferencing Applications. Baset and Schulzrinne's analysis of Skype [1] provided first insights about RTC systems. In addition to earlier studies on VCAs [33, 43], recent papers conducted extensive QoE-centric measurement studies of different VCAs such as Zoom, Meet, Teams, and WebEx [4, 36]. In the context of this work, these studies shed light on the infrastructure, geographic location, latency, bit rate and network utilization, and their impact on QoE. Choi et al. did a more longitudinal analysis of Zoom [6] while Michel et al. did an in-depth study of Zoom in a production network, demystifying Zoom's packet format [41].

Handling Video-Conferencing Traffic in the Data Plane. Edwards and Ciarleglio showcased a programmable data plane that can perform "clean" video switching of uncompressed video flows based on RTP timestamps [13]. This demo solely focuses on showing the capability of parsing RTP headers and making forwarding decisions based on RTP timestamp values, rather than offloading any SFU functionality. Other work showed that programmable data planes and eBPF/XDP programs running on servers can help with the NAT traversal functionality in VCAs [26, 34]. Our work goes way beyond this and enables the data plane to also perform packet replication and selective forwarding of actual RTP media traffic. The perhaps closest work [55] builds an SFU in software (bmv2) using P4 but does not actually implement a functional VCA. It generates dummy packets with port numbers distinguishing media layers and the design ignores all challenges related to feedback.

11 Conclusion

Taken together, our SDN-inspired SFU-switch design is driven by the key insight that most SFU tasks are, in fact, replicating and dropping media packets. Unlike traditional infrastructure where an SFU server takes care of everything, our prototype comprises an efficient programmable data plane that processes media packets at line rate and a software control plane that handles infrequent tasks such as signaling, quality monitoring, and rate adaptation. Our prototype is built with a real programmable switch (Intel Tofino2 [21]) and delivers 7-422× improved scalability over a 32-core SFU server.

Acknowledgements

We thank our shepherd, Ilias Marinos, and the anonymous reviewers for their insightful feedback. We are grateful to David Hay, John Sonchack, and Sophia Yoo for their help with Scallop's BlueField-3 implementation and the many engineers at NVIDIA and Juniper Networks for the fruitful discussions on the suitability of Scallop to their hardware platforms. We also thank Nate Foster for his advice regarding alternative packet-processing platforms for Scallop. Finally, we thank Princeton University's Office of Information Technology, Office of Institutional Research, and the Institutional Review Board for enabling us to study campus traffic. This work is supported by DARPA grant HR0011-20-C-0107 and by the NSF under CNS grants 2147909, 2151630, 2140552, 2153449, and 2152313.

References

[1] Salman A. Baset and Henning G. Schulzrinne. 2006. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *IEEE INFOCOM*. IEEE, New York, NY, USA, 1–11. doi:10.1109/INFOCOM.2006.312

[2] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2020. Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience. *SIGMETRICS Perform. Eval. Rev.* 48, 1 (2020), 27–28. doi:10.1145/3410048.3410064

[3] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. 2017. Congestion Control for Web Real-Time Communication. *IEEE/ACM Transactions on Networking* 25, 5 (2017), 2629–2642. doi:10.1109/TNET.2017.2703615

[4] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. 2021. Can You See Me Now? A Measurement Study of Zoom, Webex, and Meet. In *ACM Internet Measurement Conference*. ACM, New York, NY, USA, 216–228. https://doi.org/10.1145/3487552.3487847

[5] Xiaoqi Chen. 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*. 8–14.

[6] Albert Choi, Mehdi Karamollahi, Carey Williamson, and Martin Arlitt. 2022. Zoom Session Quality: A Network-Level View. In *Passive and Active Network Measurement*. Springer, Berlin, Germany, 555–572.

[7] NTT Communications. 2015. A Study of WebRTC Security. Retrieved January 28, 2025, https://webrtc-security.github.io.

[8] The World Wide Web Consortium. 2023. W3C Editor's Draft: WebRTC Extended Use Cases. Retrieved July 24, 2025, from https://w3c.github.io/webrtc-nv-use-cases.

[9] The World Wide Web Consortium. 2024. W3C Working Draft: Scalable Video Coding (SVC) Extension for WebRTC. Retrieved July 24, 2025, from https://www.w3.org/TR/webrtc-svc.

[10] The World Wide Web Consortium. 2025. W3C Editor's Draft: WebRTC Encoded Transform. Retrieved July 24, 2025, from https://w3c.github.io/webrtc-encoded-transform.

[11] Luca De Cicco, Gaetano Carlucci, and Saverio Mascolo. 2017. Congestion Control for WebRTC: Standardization Status and Open Issues. *IEEE Communications Standards Magazine* 1, 2 (2017), 22–27. doi:10.1109/MCOMSTD.2017.1700014

[12] Peter de Rivaz and Jack Haughton. 2019. AV1 Bitstream and Decoding Process Specification. Retrieved July 24, 2025, from https://aomediacodec.github.io/av1-spec/av1-spec.pdf.

[13] Thomas G. Edwards and Nick Ciarleglio. 2017. Timestamp-Aware RTP Video Switching Using Programmable Data Plane. ACM SIGCOMM '17 Industrial Demos, Retrieved April 12, 2023, from https://conferences.sigcomm.org/sigcomm/2017/files/program-industrial-demos/sigcomm17industrialdemos-paper2.pdf.

[14] Mathis Engelbart and Jörg Ott. 2021. Congestion Control for Real-Time Media over QUIC. In *Workshop on Evolution, Performance and Interoperability of QUIC* (Virtual Event, Germany) *(EPIQ '21)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3488660.3493801

[15] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. 2020. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *ACM Internet Measurement Conference* (Virtual Event, USA). ACM, New York, NY, USA, 1–18. doi:10.1145/3419394.3423658

[16] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. 2018. Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol. In *USENIX Networked Systems Design and Implementation* (Renton, WA, USA). USENIX Association, USA, 267–282.

[17] Boni García, Micael Gallego, Francisco Gortázar, and Antonia Bertolino. 2019. Understanding and estimating quality of experience in WebRTC applications. *Computing* 101, 11 (Nov. 2019), 1585–1607. doi:10.1007/s00607-018-0669-7

[18] Alex Gouaillard. 2018. Breaking Point: WebRTC SFU Load Testing. Retrieved July 24, 2025, from https://webrtchacks.com/sfu-load-testing/.

[19] Philipp Hancke. 2020. True End-to-End Encryption with WebRTC Insertable Streams. Retrieved July 22, 2025 from https://webrtchacks.com/true-end-to-end-encryption-with-webrtc-insertable-streams.

[20] Todd Hoff. 2020. A Short On How Zoom Works. Retrieved July 24, 2025, from http://highscalability.com/blog/2020/5/14/a-short-on-how-zoom-works.html.

[21] Intel Corp. 2022. Intel Tofino. Retrieved July 24, 2025, from https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[22] ITU-T. 2001. *End-user multimedia QoS categories*. Recommendation G.1010. International Telecommunication Union, Geneva, Switzerland.

[23] ITU-T. 2003. *One-way transmission time*. Recommendation I.371. International Telecommunication Union, Geneva, Switzerland.

[24] Mukund Iyengar. 2021. WebRTC Architecture Basics: P2P, SFU, MCU, and Hybrid Approaches. Retrieved July 24, 2025, from https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66.

[25] Jitsi 2023. Jitsi Video Bridge - Open Source Video Conferencing for Developers. Retrieved July 24 2025, from https://jitsi.org/jitsi-videobridge/.

[26] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2020. Offloading Media Traffic to Programmable Data Plane Switches. In *IEEE International Conference on Communications (ICC)*. 1–7. doi:10.1109/ICC40277.2020.9149159

[27] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. 2020. A case for SmartNIC-accelerated private communication. In *Asia-Pacific Workshop on Networking (APNet)*. 30–35.

[28] Firat Kiyak, Brent Mochizuki, Eric Keller, and Matthew Caesar. 2009. Better by a HAIR: Hardware-amenable internet routing. In *IEEE International Conference on Network Protocols*. 83–92. doi:10.1109/ICNP.2009.5339694

[29] Shagufta Zuveria Kottur, Krishna Kadiyala, Praveen Tammana, and Rinku Shah. 2022. Implementing ChaCha based crypto primitives on programmable SmartNICs. In *ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures*. 15–23.

[30] Xsight Labs. 2025. X-ISA: SIMPLE IPV4. Diving into the Nitty-Gritty: Implementing a Basic Network Cross-Connect. Retrieved May 5, 2025, from https://xsightlabs.com/wp-content/uploads/2025/04/XISA-Cross-Connect.pdf.

[31] Xsight Labs. 2025. X-ISA: SIMPLE IPV4. Diving into the Nitty-Gritty: Implementing a Trivial IPv4 Switching Program. Retrieved May 5, 2025, from https://xsightlabs.com/wp-content/uploads/2025/04/XISA_Simple_IPv4.pdf.

[32] Xsight Labs. 2025. X2 Programmable Ethernet Switch. Retrieved Jun 27, 2025, from https://xsightlabs.com/products/.

[33] Insoo Lee, Jinsung Lee, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. 2021. Demystifying Commercial Video Conferencing Applications. In *ACM International Conference on Multimedia*. ACM, New York, NY, USA, 3583–3591. https://doi.org/10.1145/3474085.3475523

[34] Tamás Lévai, Balázs Edvárd Kreith, and Gábor Rétvári. 2023. Supercharge WebRTC: Accelerate TURN Services with EBPF/XDP. In *Workshop on EBPF and Kernel Extensions* (New York, NY, USA) *(eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 70–76. doi:10.1145/3609021.3609296

[35] Luis López, Miguel París, Santiago Carot, Boni García, Micael Gallego, Francisco Gortázar, Raul Benítez, Jose A. Santos, David Fernández, Radu Tom Vlad, Iván Gracia, and Francisco Javier López. 2016. Kurento: The WebRTC Modular Media Server. In *ACM International Conference on Multimedia (MM '16)*. ACM, New York, NY, USA, 1187–1191. doi:10.1145/2964284.2973798

[36] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. 2021. Measuring the Performance and Network Utilization of Popular Video Conferencing Applications. In *ACM Internet Measurement Conference*. ACM, New York, NY, USA, 229–244. https://doi.org/10.1145/3487552.3487842

[37] Markets and Markets 2023. Video Conferencing Market Forecast & Statistics. Retrieved July 24, 2025, from https://www.marketsandmarkets.com/Market-Reports/video-conferencing-market-99384414.html.

[38] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. 2008. Session Traversal Utilities for NAT (STUN). RFC 5389. doi:10.17487/RFC5389

[39] Mediasoup 2025. MediaSoup - Cutting Edge WebRTC Video Conferencing. Retrieved July 24, 2025, from https://mediasoup.org.

[40] Rick Merritt. 2019. Broadcom Throws Programmable Switch. Retrieved June 27, 2025 from https://www.eetimes.com/broadcom-throws-programmable-switch.

[41] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. 2022. Enabling Passive Measurement of Zoom Performance in Production Networks. In *ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 244–260. doi:10.1145/3517745.3561414

[42] Mozilla. 2025. WebRTC API: Signaling and video calling. Retrieved January 28, 2025, from https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling.

[43] Antonio Nistico, Dena Markudova, Martino Trevisan, Michela Meo, and Giovanna Carofiglio. 2020. A comparative study of RTC applications. In *IEEE International Symposium on Multimedia*. IEEE, New York, NY, USA, 1–8.

[44] Karl Norrman, David McGrew, Mats Naslund, Elisabetta Carrara, and Mark Baugher. 2004. The Secure Real-time Transport Protocol (SRTP). RFC 3711. doi:10.17487/RFC3711

[45] NVIDIA. 2025. NVIDIA BlueField Networking Platform. Retrieved Jun 27, 2025, from https://www.nvidia.com/en-us/networking/products/data-processing-unit.

[46] Isaac Oliveira, Emídio Neto, Roger Immich, Ramon Fontes, Augusto Neto, Fabrício Rodriguez, and Christian Esteve Rothenberg. 2021. Dh-aes-p4: on-premise encryption and in-band key-exchange in P4 fully programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 148–153.

[47] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *IEEE International Conference on Cloud Computing*. 500–507. doi:10.1109/CLOUD.2011.42

[48] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. 2019. Cryptographic hashing in P4 data planes. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 1–6.

[49] Lars-Christian Schulz, Robin Wehner, and David Hausheer. 2023. Cryptographic Path Validation for SCION in P4. In *European P4 Workshop (EuroP4)*. 17–23.
[50] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC 3550. doi:10.17487/RFC3550
[51] Henning Schulzrinne and Jonathan Rosenberg. 2002. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264. doi:10.17487/RFC3264
[52] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. 2007. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology* 17, 9 (2007), 1103–1120. doi:10.1109/TCSVT.2007.905532
[53] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA–Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *USENIX Annual Technical Conference (ATC)*. 691–704.
[54] The Alliance for Open Media, AV1 Real-Time Communications Subgroup. 2023. RTP Payload Format For AV1. Retrieved October 5, 2023, from https://aomediacodec.github.io/av1-rtp-spec/.
[55] Pavlos Tsikrikas and George Xylomenos. 2024. A Selective Forwarding Unit Implementation in P4. In *2024 IEEE Conference on Standards for Communications and Networking (CSCN)*. 181–186. doi:10.1109/CSCN63874.2024.10849740
[56] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems* (Bordeaux, France). Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. doi:10.1145/2741948.2741964
[57] W3C. 2024. MediaStreamTrack Insertable Media Processing using Streams. Retrieved January 28, 2025, from https://w3c.github.io/mediacapture-transform.
[58] W3C. 2025. Identifiers for WebRTC's Statistics API. Retrieved July 24, 2025, from https://www.w3.org/TR/webrtc-stats.
[59] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. 2023. Raven: Stateless rapid IP address variation for enterprise networks. *Privacy Enhancing Technologies Symposium (PETS)* (2023).
[60] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using Trio: Juniper networks' programmable chipset-for emerging in-network applications. In *ACM SIGCOMM Conference*. 633–648.
[61] Sophia Yoo and Xiaoqi Chen. 2021. Secure keyed hashing on programmable switches. In *ACM SIGCOMM Workshop on Secure Programmable network INfrastructure*. 16–22.
[62] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. 2021. Loki: Improving Long Tail Performance of Learning-Based Real-Time Video Adaptation by Fusing Rule-Based Models. In *ACM MobiCom* (New Orleans, Louisiana). Association for Computing Machinery, New York, NY, USA, 775–788. doi:10.1145/3447993.3483259
[63] Xiaoqing Zhu, Rong Pan, Michael A. Ramalho, and Sergio Mena de la Cruz. 2020. Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media. RFC 8698. doi:10.17487/RFC8698
[64] Zoom. 2025. Zoom Account API. Retrieved January 27, 2025, from https://developers.zoom.us/docs/api/rest/reference/account/methods/.

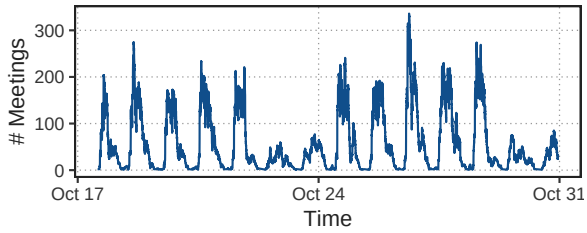


Figure 21: Number of concurrent Zoom meetings hosted by our university's Zoom account over time.

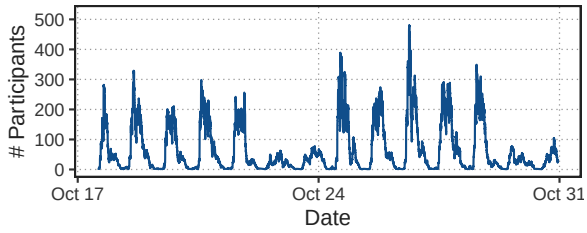


Figure 22: Number of concurrent Zoom participants hosted by our university's Zoom account over time.

Appendices are supporting material that has not been peer-reviewed.

A Zoom API Data Set

The Zoom API data set used in this study was collected from our university campus. Zoom's API [64] is made available to account administrators to access information about meetings, participants, and recordings. We cooperated with our campus IT department to continuously collect this data which was then anonymized and aggregated to protect user privacy (see §1). The data set contains information about 19,704 meetings that took place between October 17 and October 30, 2022. The data only includes meetings that were hosted by our university's Zoom account and does not contain any information about external meetings, for example those hosted by other institutions or individual users. The data set includes information about the number of participants in each meeting, the duration of the meeting, and the duration and composition of media streams that were active during the meeting. The number of concurrent meetings and concurrent participants over time are depicted in Figures 21 and 22, respectively.

B Zoom Packet-Capture Data Set

The packet-level trace we use to analyze the operation of Zoom's SFUs was collected at two border routers on our University campus on May 5th, 2022 over the course of 12 hours. As opposed to the API data set (see Appendix A, this data set includes packet-level data of all Zoom calls that traverse our campus network regardless of who hosted the meeting. The key statistics of the trace are summarized in Table 3. In order to reduce the data rate to be analyzed in software, we wrote a P4 program for an Intel Tofino switch that filters Zoom packets. During the capture, our switch processed an average of 626,069 packets per second, with an average of 43,733 per second being Zoom traffic and, subsequently, filtered out.

Capture duration	12h
Zoom packets	1,846 M (42,733/s)
Zoom flows	583,777
Zoom data	1,203 GB (222.9 Mbit/s)
RTP media streams	59,020

Table 3: Capture Summary

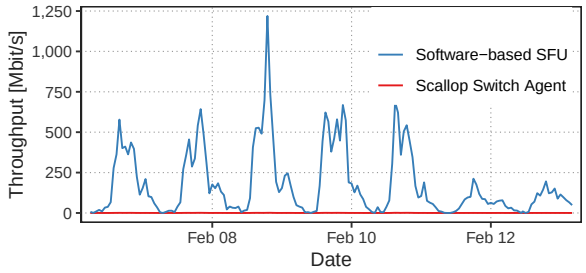


Figure 23: Bytes processed during 2nd week of Feb '23

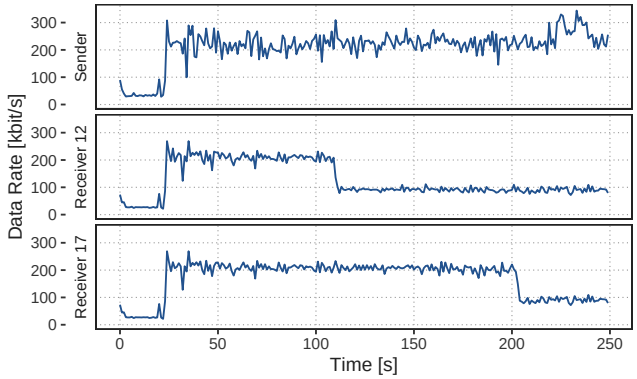


Figure 24: Forwarded bytes of a single video stream to two separate participants in a Zoom meeting.

In Figure 23, the blue curve shows the byte rate a software-based SFU would have to process if it were to handle all of our campus traffic during a week (peaks around 1250 Mbit/s), and the red curve shows the byte rate that the Scallop switch agent would have to process in comparison (peaks around only 4.4 Mbit/s). Even for a powerful 40 Gbit/s server, the peak byte rate from our campus already consumes 3.1% of its total capacity if a software-based SFU is used. For large providers like Zoom, many such servers would, therefore, be required to serve the thousands of campuses and enterprises that use their services. In contrast, with Scallop, only 0.01% of the server's capacity would be used.

C Use of Media Scalability in Zoom Packet Trace

Scalable Video Coding (SVC) enables efficient video transmission by encoding a stream into multiple layers of increasing quality, allowing selective adaptation based on network conditions. An SFU can reduce the frame rate (or resolution) by dropping frames from

Platform	Category	Support for Scallop's Data-Plane Features		
		Deep and flexible parsing	Scalable replication	Sequence-Number rewriting
Intel Tofino2 [21]	Switch	Yes	Yes	Yes
Juniper Trio [60]	Switch	Yes	Yes	Yes
Xsight Labs X2 [32]	Switch	Yes	Yes	Yes
NVIDIA BlueField-3 [45]	SmartNIC	Yes	Yes, but only via mirroring	Yes, with workaround (see E)

Table 4: Comparison of packet-processing products by support for Scallop's key data-plane features.

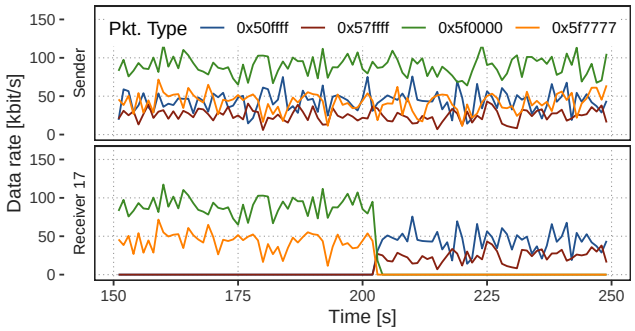


Figure 25: Forwarded bytes per scalability layer of a single video stream to a single participant in a Zoom meeting.

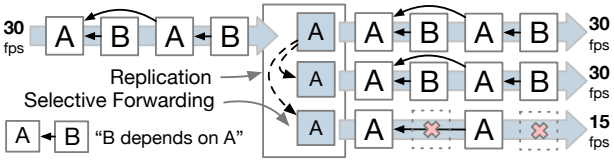


Figure 26: Media forwarding when using SVC.

the enhancement layers, effectively limiting playback to a lower frame rate while maintaining smooth video delivery. Figure 26 shows an example where, by discarding every other frame, the SFU decreases the temporal resolution, reducing bandwidth usage while preventing playback freezes.

To give a concrete example of how Zoom adapts media streams at their SFUs, we analyze a meeting that is part of our Zoom packet trace introduced in Appendix B. Figure 24 shows an example of a participant's outgoing video stream and the corresponding incoming streams at two other participants. Time zero denotes the start of the meeting at which the sending participant starts transmitting a low-bitrate video stream which then increases in bitrate at around 20 seconds into the meeting. Presumably in response to constrained downlink capacity, the SFU then reduces the bitrate forwarded to participant 12 (at around 110s) and the bitrate forwarded to participant 17 (at around 200s), respectively.

Zoom uses the Real-time Transport Protocol (RTP) [50] encapsulated in custom, proprietary headers over UDP to transmit media [41]. The RTP header carries a sequence number and various extension fields in each packet that is not being changed by Zoom's servers during forwarding. As a result, we can exactly specify which packets are forwarded to a given participant by the SFU and which are not. We further observed that Zoom's RTP packets carry various RTP extension headers. One header extension field carries a three-byte value that appears to be a bit mask. The packets in the outgoing media streams typically carry between two and six different values for this field across their packets which we for now denote as the *packet type*. We observe that the SFU only forwards either all or a strict subset of these packet types to a given receiver.

Figure 25 shows the bit rate of the same media stream as previously shown broken down by packet type. We can see that the adaptation of the media stream received by participant 17 around second 200 is achieved by changing the set of packet types forwarded to this client. This is consistent with previous observations that Zoom uses media scalability through Scalable Video Coding (SVC) and Simulcast [9, 20, 41, 52] and leverages this field to indicate to the SFU which layer of a scalable stream the respective packet carries. It is common to encode the type of scalability layer in the RTP header [52] which is transmitted in clear text; similar header extensions exist, for example, for the AV1 codec [12, 54] which we leverage in this work.

D Support for Scallop on Packet-Processing Platforms

As discussed in Section 4, Scallop's data plane requires three main capabilities in the hardware: (1) deep parsing into RTP headers, (2) scalable replication of packets, and (3) stateful memory for sequence-number rewriting. We investigated products from different vendors to understand the support for these capabilities on various packet-processing platforms. Among these products, were the Intel Tofino line of switches [21], the Juniper Trio switch [60], the X2 switch from Xsight Labs [32], and the NVIDIA BlueField-3 SmartNIC or Data Processing Unit (DPU) [45]. Table 4 provides a matrix of these hardware targets versus supported features of Scallop, to the best of our understanding. We proceed to implement Scallop's data plane on the hardware pipeline of two of these platforms—a representative switch (Intel Tofino2), and a representative SmartNIC (NVIDIA BlueField-3). Finally, we note that many other hardware platforms are programmable, even if their vendors do not expose that programmability to third-party programmers. For example, Broadcom chipsets can be programmed using the Network Programming Language (NPL) [40]. Using the insights and designs in our paper, vendors can create support for SFUs as a service of their switch or NIC devices, even if researchers or third-party software developers cannot.

Scallop on programmable hardware switch platforms. Scallop works and is highly performant on the Intel Tofino2 switch, as illustrated in Section 8. Our investigation of the alternative switch platforms revealed the following. The Juniper Trio has a highly

parallelized run-to-completion model (as opposed to the Tofino's pipeline-based approach) where each packet is processed—including parsing and replication—using native microcode by an individual thread. Each thread has access to a common pool of GBs of stateful memory via a cross-bar. Using these features, the Trio should be able to process thousands of video-conferencing packets simultaneously in hardware and can handle Tbps-scale traffic with tens to hundreds of microseconds in packet-processing latency. These capabilities make it a suitable alternative hardware target for Scallop. We have limited understanding of the X2 programmable switch at the time this paper is written. However, based on our interactions with the vendor's engineers, and our review of their open-source examples [30, 31], we believe it should provide features and performance similar to the Tofino2, including support for 12.8 Tbps traffic.

Scallop on SmartNIC platforms. As discussed in Section 4, production-scale SFUs are currently deployed on servers (entirely in software) in data centers. If such a server comes equipped with a SmartNIC, Scallop can be leveraged to offload much of the server's work to the SmartNIC's accelerated hardware pipeline, making the SmartNIC. However, SmartNICs such as the NVIDIA BlueField-3 are limited in many ways compared to the high-speed programmable switches discussed above. First, the BlueField-3's packet- processing capacity is a few hundred Gbps, compared to the Tbps capacity of switches. Second, the BlueField-3's parser is considerably less flexible than the Tofino's, making it challenging to parse deep into the RTP header extensions as required by Scallop. Third, unlike the Tofino's Packet Replication Engine (PRE), BlueField-3 does not provide hardware support for generating multiple replicas of a packet scalably. Fourth, the BlueField-3's hardware pipeline does not allow access to register arrays, making it difficult to implement stateful operations directly in the hardware.

Scallop Hardware Prototype on BlueField-3. Nevertheless, the BlueField-3 does have an assortment of strong features that allow us to work around some of these limitations. First, the BlueField-3 specifies the DOCA Pipeline Language (DPL), which is NVIDIA's target-specific P4 language, to program their hardware switch pipeline. This allows developers familiar with P4 to leverage its hardware capabilities. Second, the BlueField-3 implements a *FlexParser*, which is a flexible parser to allow programmability in the parsing logic. The fixed parser parses until the transport layer, and then allows the FlexParser to take over the rest of the parsing. However, the FlexParser has several limitations that don't allow us to parse RTP packets to the required depth. Fortunately, the BlueField-3 has a *reparse* feature that allows developers to retain the results of partial parsing in one pass, and *re-invoke* parsing from that point on in a new pass. This feature is is different from and more efficient than recirculation—since the BlueField-3 follows a *run-to-completion* model as opposed to the Tofino's fixed pipeline model, it allows the same functions (e.g., parsing) to process the packet multiple times in a single ingress-to-egress pass. Using the FlexParser and the reparse features, we were able to parse sufficiently deep into the RTP header extensions. Third, the BlueField-3 has a *mirror* extern that creates one replica of a packet at a time. Using this extern and packet recirculations, we were able to implement SFU-scale replication, as discussed in Section 6. Finally, unlike the Tofino, it has an *add_entry* extern that allows the data plane to insert records directly into a match-action table. Using a combination of this feature, match-action tables populated from the control plane, and recirculations, we were able to implement our sequence number-rewriting logic directly in the hardware pipeline.

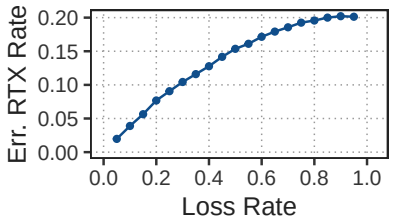


Figure 27: Retransmission Overhead with S-LR.

E Parsing RTP Extension Headers

Below, we describe our parsing logic for the Intel Tofino2 programmable switch. We implement a parsing tree with the same functionality on the BlueField-3 SmartNIC using its FlexParser and reparse features, as described in Appendix D.

Parsing Deep into RTP Headers on Tofino2. After parsing the Ethernet, IP, and UDP headers, we use the destination IP and port numbers to determine whether the packet is destined for the SFU. If so, we *lookahead* into the first 4 bits of the UDP payload to determine whether the packet resembles an RTP or an RTCP packet. If the packet is an RTP media packet or an RTCP sender report, we mark it for replication; otherwise, we forward it. Rate adaptation in SVC works by dropping packets that correspond to the higher quality layers of an AV1 media stream, as needed. Determining what quality layer a packet belongs to requires parsing into the RTP *header extensions* beyond the RTP header, identifying the header extension corresponding to the AV1 protocol (if present), and parsing the AV1 header[2] to extract its *dependency template ID*, which when combined with the control plane's knowledge of the mapping between dependency templates and quality layers, tells us the quality layer of the packet itself. Parsing deep into the RTP header extensions to find the AV1 header requires handling variable-length and variable-position headers which is tricky because the P4 parser on Tofino relies on a largely static parse graph. We solve this by (i) implementing a *depth-aware* parsing tree for the RTP header extensions, where for every depth in the tree, we have a *landing state* that determines—using the *lookahead* function—what type of header element (one-byte header, two-byte header, or padding) comes next, and (2) using the Tofino's *ParserCounter* feature to track whether there are header bytes left to parse.

F Sequence-Number Rewriting in Hardware Data Planes

We implement Scallop's sequence number-rewriting logic on both the Tofino2 and the BlueField-3 platforms. In Appendix D, we describe the features we leverage to make our prototype work on the BlueField-3. Below, we describe aspects of our Tofino2 prototype.

[2]We implement a Wireshark plugin to parse RTP header extensions to identify and display AV1 headers which we open-source.

Collision-free hash tables. Scallop stores per-stream state in a *Stream Tracker* table. The control plane (switch agent) provides a unique, collision-free hash-based index for each new stream via match-action rules in a *Stream Index* table: this helps use the Stream Tracker table's memory maximally.

Per-stream state. The egress pipeline stores four hash tables for the S-LM heuristic or eight hash tables for the S-LR heuristic on the Tofino to enable sequence number rewriting The hash tables are always accessed in order. Each active rate-adapted stream maintains state across all the hash tables in each case. Since the indices are managed by the control plane ensuring zero hash collisions and immediate cleanup when a stream ends, all cells in these tables can be used. Our experiments show that each stage of Tofino2 can support up to 143.5K such cells. Since the S-LM heuristic has a smaller memory footprint, we can fit 16 stages of each of its data structures inside the Tofino2, amounting to 2.3M concurrent streams. In contrast, we can fit 3 stages of each data structure of the S-LR heuristic, amounting to 430.5K concurrent streams.

Overhead due to Sequence Number Rewriting. In Figure 27, we report the overhead of our sequence number rewriting heuristic S-LR, which is designed for a high-loss environment (§7). This overhead is in terms of the fraction of extra retransmissions triggered by the receiver due to a mismatch between the rewritten sequence number and the ideal rewritten sequence number an oracle may have generated. We observe that the overhead is below 5% for up to 10% loss rate, and around 7.5% for a 20% loss rate. Even under really high loss rates (where the meeting itself would start to become unusable), the overhead does not exceed 20%.