Integrating Video Codec Design and Network Transport for Emerging Internet Video Streaming Applications.

Devdeep Ray

CMU-CS-22-143
August 2022

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Srinivasan Seshan, Chair (Carnegie Mellon University)
Justine Sherry Martins (Carnegie Mellon University)
Anthony Rowe (Carnegie Mellon University)
David Chu (Magic Leap)

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Copyright © 2022 **Devdeep Ray**

This research was sponsored by Vanguard Charitable and the National Science Foundation under award numbers CNS-1345305, CNS-1564009, CNS-1565343, CNS-1700521, and CNS-1956095, and through Google research grants and the use of Cloud Lab resources.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Video, Networks, Bandwidth, Latency, Packet loss, Streaming, Cloud, Live, Low-latency, Real-time, Gaming, Virtual Reality, Augmented Reality, Compression, Codec, Congestion Control

This thesis is dedicated to everyone who has played an instrumental role in my life in getting me where I am at today.

Abstract

Video streaming applications on the Internet are diverse, and have very distinct notions of the quality of experience (QoE). These distinctions require carefully designed systems and protocols in order to balance factors like video quality, delay, bandwidth utilization, and video coding performance in a manner that is appropriate for each specific application. While these trade-offs are clear cut and simple to implement for traditional video streaming applications (eg. conferencing, live TV broadcasts, video-on-demand), emerging video streaming applications like social live video streaming, cloud gaming and remote-rendered AR/VR have unique properties and demanding QoE requirements that make choosing the right trade-offs and achieving the desired QoE challenging.

Conventional video streaming systems largely treat the two key aspects of video streaming, video encoding and data transmission, as separate entities, and rely on naïvely combining techniques that have been developed independently in the field of video coding and network transport. This approach severely limits the capability to navigate the complex trade-offs required to achieve good QoE for emerging video streaming applications. In this thesis, we explore techniques that use encoder and network co-design techniques that expand the breadth of the trade-offs that can be achieved by a video streaming system, and thus, enable designs that are tailored to the demands of specific video streaming applications. Our work shows that integration of video encoding and network transport at various levels is crucial in achieving good QoE for emerging video streaming applications like social live streaming, cloud gaming and cloud AR/VR.

We first explore the space of social live video streaming (SLVS), where the key distinction from traditional video streaming applications is the presence of viewers who view the video stream at different delays. Our system, Vantage [1], dynamically optimizes bandwidth allocation across low latency video frames and selective quality-enhancing retransmissions. In the presence of bandwidth variations, Vantage enables low-latency interaction for real-time viewers, and achieves high video quality for time-shifted viewers.

Second, we explore the application space of cloud-rendered video games. Cloud gaming demand extremely low interaction latency, and very high video quality in order to achieve parity with locally-rendered applications - a challenging task when streaming over the Internet. We developed a new end-to-end video streaming architecture, called Prism, in order to improve the frame delay and video quality in the presence of transient packet loss. When a video stream is affected by transient packet loss, Prism carefully splits the available bandwidth between a low latency stream, and a quality-preserving secondary stream, where the different sub-streams address different stages of loss recovery. Prism accounts for the complex relationships between video compression bitrate and the resulting video quality in order to achieve higher video quality and lower video frame delay. Optimizing the allocation of bandwidth between the streams enables the use of aggressive loss prediction techniques, rapid loss recovery, and high quality post-recovery, with zero computational and bandwidth overhead during normal operation - avoiding the pitfalls of existing approaches.

Third, we show that existing approaches for performing congestion control in the context of emerging Internet video streaming applications like cloud gaming and cloud AR/VR severely limit the QoE. We demonstrate that the design choices made by existing congestion control algorithms severely limit their suitability for the demanding requirements of cloud streaming applications, and discuss how the complex interactions between the congestion control algorithm and the video encoder rate control mechanism have a significant impact on the video frame delay and video quality. We also discuss the challenges with testing and deploying new congestion control algorithms designed for emerging applications. We propose a tool called CC-Fuzz for automatically stress testing a congestion control algorithm in order to identify problems with the design and implementation of the congestion control algorithm, with the goal of inspiring confidence in the design of the algorithm and catching implementation bugs.

This work shows that rethinking traditional designs for video streaming with a focus on integrated video codec and network transport design enables novel QoE trade-off modalities that are able to push the QoE envelope of video streaming systems and achieve the demanding QoE goals of emerging Internet video streaming applications.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Srinivasan Seshan, who supported my ideas and lent his technical expertise on various topics. I am extremely grateful to the members of my thesis committee, Justine Sherry-Martins, Anthony Rowe, and David Chu. I would like to thank the National Science Foundation for supporting my research through various grants, Vanguard Charitable, Cloudlab, and Google for various internship opportunities and funding.

Special thanks to everyone who collaborated with me during my PhD journey. Thanks to Matthew K. Mukerjee for guiding me in the initial parts of my PhD and for the opportunity to collaborate on VDX [2]. I would like to thank my co-authors on the Vantage [1] project, Rashmi KV, and Jack Kosaian, for their contributions. I am also thankful to Vicente Bobadilla Riquelme and Deklin Caban for being awesome summer interns. Vicente's work on neural network loss prediction was instrumental for Prism. Deklin did a great job on understanding CCAs used for cloud gaming. I would also like to thank everyone who attended the Tuesday Lunch seminar for excellent presentations on networking topics, and Srini's research group, including Sagar, Anup, Ray, Shawn, Chris, Mallesh, and Daehyeok. Sincere thanks to all of my mentors at Google (Jason Glasgow, Tarun Bansal, Seungjoon Lee, Connor Smith, Teng Wei, David Chu, and Neal Cardwell), and Pierre Alliez, who mentored my first research experience at Inria.

I sincerely thank my parents, Ujjal Kumar Ray and Maytreyi Ray, and my sister, Dipanwita Ray for being the best sibling. I would not be where I am without your constant love and support. A shout out to Pratik Fegade, my forever roommate for helping me debug weird bugs, and to my other roommates, neighbors, and friends who have been my family away from home - Arijit, Siddhant, Nishanth, Abhijeet, Aditi, Akshay, Rohit, Maahin, Shivani, Shreyanshi, Meghna, and Sheona.

Finally, I would not have made it through the isolation of COVID'19 without aviation - for that, I would like to thank my flight instructor, Tim Depaolis, and everyone else at High Flight Academy. In addition, I would like to thank a friend whose first solo post inspired me to turn my childhood dream of being a pilot into reality - you constantly amaze me, and have always inspired me to push myself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Video traffic over the Internet has grown significantly over the past decade and comprises of more than 80% of the total traffic [6] on the Internet today. Video streaming applications are diverse, ranging from applications like video-on-demand (VOD), live broadcasts, video conferencing, social live video streaming (SLVS), cloud gaming, and remote-rendered augmented reality (AR) and virtual reality (VR). While applications like VOD, live broadcasts and conferencing over the Internet have been around for a while, recent technological advances have made more demanding applications like social live video streaming (SLVS), cloud gaming and and remote-rendered AR/VR streaming more ubiquitous.

While emerging applications like SLVS, cloud gaming and AR/VR streaming are able to leverage existing infrastructure (eg. CDNs, edge-computing) and algorithms (eg. low latency congestion control algorithms and real-time video streaming techniques developed for video conferencing), the end-to-end quality of experience (QoE) achieved by existing techniques falls short of the level of performance desired for these applications. One reason is that existing systems are tuned to satisfy the needs of existing applications like live broadcasts and video conferencing, whereas emerging applications have unique requirements that require a different set of trade-offs. Even if these existing approaches are tuned in order to make them work better for emerging video streaming applications, the maximum QoE achievable by these approaches is limited. This is because the designs of existing techniques that are used for achieving the desired QoE and for addressing the various challenges of streaming video on the Internet are usually limited in scope to a small part of the entire video streaming stack, and do not consider interactions between various sub-systems. For example, video quality improvements are usually from improvements in the bandwidth obtained by the congestion control algorithm, or due to better video compression algorithms. Similarly, typical packet loss recovery techniques are either done at the codec layer (IDR-frames), or at the network layer (FEC, retransmissions). This restricts the QoE envelope achievable by these techniques in the context of emerging applications. For example, designing a congestion control algorithm that achieves low queuing delay may have bursty packet transmission patterns, which can make it harder to translate the low queuing delay of a congestion control algorithm to low end-to-end frame (since bursty transmissions at the network layer requires buffering at the sender and receiver side in order to achieve smooth video playback). This requires careful design to ensure that a CCA's transmission patters are compatible with the traffic pattern of real-time video.

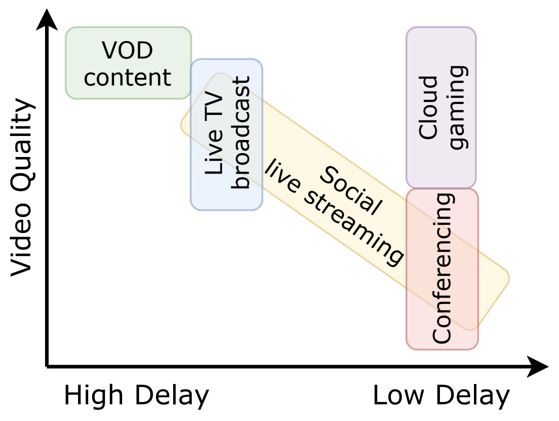
Compared to traditional video streaming applications, emerging video streaming applications like social live video streaming (SLVS), cloud gaming, and remote-rendered AR/VR have unique video quality and delay requirements:

1. **Social Live Video Streaming (SLVS):** SLVS is an evolution of personalized live video broadcast, enabling viewers to interact with the viewers in real-time in the form of comments, reactions and monetary donations [7], while also allowing viewers to view the video in a time-shifted manner (eg. with some delay, or VOD-style viewing). Thus, SLVS applications must achieve low end-to-end interaction latency for the real-time viewers, and must simultaneously achieve high video quality and smooth video playback for time-shifted viewers.
2. **Cloud Gaming and remote-rendered AR/VR:** These services offload compute-heavy applications like video games and AR/VR rendering to the cloud or a networked machine, and stream the rendered view-port to the end-user as a video stream. These services aim to achieve parity with or exceed the QoE of running applications on a local device that is physically connected to the display device using high bandwidth links (e.g. HDMI, displayport). Thus, the transmitted video frames must be of very high quality with minimal compression-induced artifacts, and must have very low end-to-end frame delay (few tens of milliseconds [8]).

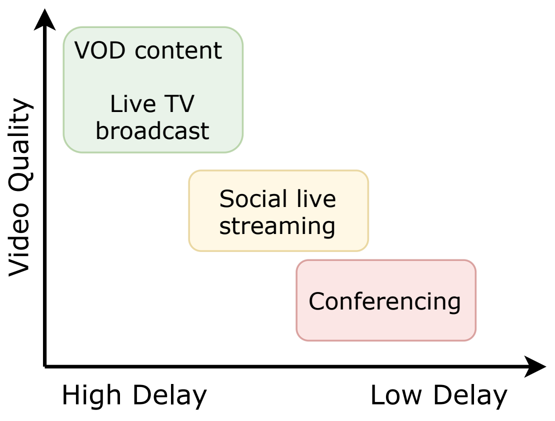
In this thesis, we study the benefits of a targeted, cross-layer approach for designing techniques for the different aspects of video streaming, like video compression, loss recovery, and congestion control. We show that designing video coding techniques and network algorithms that are tailored to the needs of specific applications plays a big role in achieving the unique and demanding QoE requirements of emerging video streaming applications. For example, existing systems for streaming video do not account for the existence of time-shifted viewers for SLVS applications - in this case, designing a tailored solution that explicitly accounts for the diverse viewing delays can result in a higher aggregate QoE across all the viewers. In addition, we show that there are significant benefits of carefully considering the interactions between the video codec layer and the network layer when designing systems and techniques for emerging Internet video streaming applications (e.g. unique QoE requirements, adverse network conditions like loss and variable bandwidth), and this can push the performance boundaries of existing techniques. For example, designing congestion control algorithms that are explicitly designed to work with the traffic pattern of video streaming can result in higher quality video and lower end-to-end video frame delay, instead of integrating video agnostic congestion control algorithms with the video compression layer.

1.1 Quality of Experience for Video Streaming Applications

The end-to-end QoE of a video streaming system is generally determined by the video quality and the end-to-end video frame delay. The QoE requirements of some common video streaming applications are shown in Figure 1.1a. Video-on-demand (or VOD) applications (e.g. YouTube, Netflix) stream pre-recorded content. VOD applications require high video quality and smooth playback, since common use cases include watching high quality movies for example. For VOD applications, there is no need for interaction between the server hosting the video and the viewer



(a) Desired video-quality and delay trade-off for various applications.



(b) Video-quality and delay trade-off achieved by current systems.

Figure 1.1: Application requirements and the quality-delay trade-off achieved by traditional video streaming systems.

- the server simply needs to deliver video data at a rate that is faster than the rate at which it is being played back. Large scale live broadcasts like live TV and webinars stream content in real time, but there are no interaction between the viewers and the streamers. These applications have relaxed end-to-end delay (5-10 seconds) requirements. The video quality requirements are also relaxed compared to VOD - live broadcasts do not need to be as high quality as movies that are streamed on VOD. Video conferencing applications, on the other hand, are interactive applications - this imposes a real-time requirement on these applications. Video conferencing applications require a maximum end-to-end delay of 100-200 milliseconds. On the other hand, these applications are generally more tolerant to low quality video, stutters, and fluctuations in video quality.

Compared to the traditional video streaming applications described above, emerging applications have unique and demanding QoE requirements. Social live video streaming (SLVS) applications enable interactive broadcasting, with viewers interacting with the streamer in real-time. In addition, the video is also typically archived in order to make it available for viewing at a later time. In SLVS, the real-time viewers require low delay and are tolerant to low video quality. The time-shifted viewers view the streamed content as VOD content, and thus, require higher video quality, but do not require interactivity. In contrast to traditional video streaming applications, SLVS applications require both, low delay and high video quality, but for different sets of viewers.

Cloud gaming and remote-rendered AR/VR applications aim to replace local machines that perform computationally demanding tasks like rendering. Thus, these applications require extremely low latency and very high video quality at the same time. In contrast to traditional video streaming applications, where there is usually an acceptable trade-off between video frame delay and video quality, cloud gaming and remote-rendered AR/VR applications have extremely high QoE demands.

When designing a system for streaming video on the Internet, there are three important challenges that need to be solved:

1. **Video Compression / Utilization of Available Bandwidth.** Bandwidth on the Internet is limited, and hence, each system must use the appropriate video compression scheme and video frame transport mechanism in order to utilize the available bandwidth effectively in order to achieve the QoE goals of the application.
2. **Mitigating Packet Loss.** Since the Internet is a best-effort service, packets sent over the Internet are invariably subject to packet loss. Compressed video is highly sensitive to packet loss, since lost parts of a frame prevent the frame and subsequent frames from being decoded.
3. **Congestion Control.** Internet bandwidth can be highly variable, network links can exhibit delay variations and jitter, and video traffic must compete with competing queue-building flows. The congestion control algorithm used in a video streaming system determine how much bandwidth the application can use for transmitting video data.

The design decisions made in a video streaming system with respect to each of the challenges mentioned above affects the QoE of the application. For example, when applications require high video quality but lower delay, the video can be compressed using more efficient techniques like B-frames 2.2.3, with the trade-off being higher video frame delay. The particular packet loss recovery scheme used in a video streaming system must account for the application's QoE needs - for example, using packet retransmissions are fine for applications that are not delay-tolerant, whereas using techniques like FEC are better for latency sensitive applications. The properties of the congestion control algorithm used affects the QoE of the video streaming application - low latency congestion control algorithms reduce the end-to-end delay, but can suffer from low throughput in the presence of queue-building competing traffic and links with high delay jitter. On the other hand, loss-based congestion control algorithms like TCP-Cubic can effectively utilize the network bandwidth, but have much higher queuing delay, and hence, the end-to-end frame delay will be higher. The delay-quality performance achieved using traditional video streaming techniques for various applications are shown in Figure 1.1b

Most VOD and live TV broadcast services today use video streaming protocols like MPEG-DASH [9] and HLS [10]. Since VOD applications do not have low-latency requirements, video content is pre-encoded into multi-second video segments, which are downloaded by the client into a local playback buffer before they are played. The chunk-based approach can leverage the efficiency of traditional video compression techniques developed for stored video content (for example, B-frames). The use of pre-encoded video segments also enables efficient dissemination at a large scale using traditional content delivery networks (CDNs). The video segments are transmitted using reliable transport protocols like TCP, and use traditional congestion control algorithms that prioritize throughput (eg. BBR [11], Cubic [12]). The use of TCP retransmissions for loss recovery, and queue-building high-throughput CCAs like BBR and Cubic are appropriate for VOD applications, since they do not require low latency, and the video quality has a significant impact on the QoE.

Video conferencing applications are interactive in nature, and thus, require low end-to-end video frame delay (Figure 1.1a), but are more tolerant to low quality video. These applications use low-latency video streaming protocols like WebRTC [13], RTMP, and Salsify [14], that prioritize end-to-end video latency over video quality. Systems designed for video conferencing use traditional video compression techniques, with specific configurations that reduce the end-to-end video delay (e.g. avoiding B-frames) at the cost of reduced video quality. These systems use low-latency congestion control algorithms (eg. Sprout [15], GoogCC), conservative video encoding bitrates, and minimal client- and sender-side buffering in order to reduce the latency of the video transmission pipeline. Various loss recovery schemes are used in these systems, like retransmissions, I-frames, and FEC. Some systems like WebRTC dynamically select between the three based on factors like bandwidth, loss duration, and the application delay requirements.

Emerging video streaming applications often borrow existing video streaming system designs with few modifications. For example, SLVS applications use video streaming systems like WebRTC and RTMP, which enables interactivity with the real-time viewers, but severely hampers the QoE of time-shifted viewers. This is because video quality variations and video stutters caused by packet loss, network outages, and bandwidth variations not only affect the real-time viewers, but also affect the time-shifted viewers since the same version of the video stream is archived for consumption by the time-shifted viewers. Cloud gaming applications also use techniques like WebRTC, with some modifications like more aggressive congestion control algorithms and smaller buffers at the sender and the receiver. While this approach can work when the network conditions are excellent, they often fail to achieve the desired QoE level due to loss, competing traffic and network bandwidth variations.

The diverse and dynamic nature of the Internet makes achieving a good balance between the various factors that influence the QoE challenging. While there has been significant research on navigating the video quality and frame delay trade-off for applications like VOD and conferencing, they are not sufficient to satisfy the unique and demanding QoE requirements of emerging applications like SLVS, cloud gaming and AR/VR streaming.

The limitations of existing approaches in achieving the unique QoE requirements of emerging video streaming applications are discussed in additional detail in the section below.

1.2 Limitations of Existing Approaches

The design of existing video streaming systems follow a “one size fits all” philosophy. The use of generic, video agnostic congestion control algorithms, standard video codecs, and traditional loss recovery mechanisms in video streaming systems severely limits the QoE of emerging video streaming applications.

First, traditional video streaming techniques approach the problem of streaming video over the Internet as multiple distinct sub-problems and the solutions are narrow and focus on a specific component of the video streaming system - Network congestion control deals with avoiding congestion and queuing delays in the network, video coding techniques deal with efficient utilization of the effective network throughput for optimizing video quality, and packet losses are handled using traditional network-layer loss recovery schemes, or video codec level loss recovery schemes. For example, in the case of VOD applications, they use generic congestion control algorithms like TCP-Cubic and TCP-BBR, use network-layer loss recovery (via retransmissions), and use adaptive bitrate (ABR) algorithms in order to select appropriate bitrates for each video chunk based on the state of the video playback buffer in order to achieve the best possible video quality without affecting playback smoothness. This is also true in the case of video conferencing applications. For example, loss recovery techniques used for video conferencing either operate at the codec layer (using I-frames), network layer (using retransmissions), or at the application layer (using FEC). Similarly, congestion control algorithms used in conferencing applications are often video-agnostic and work independently, and the video bitrate is determined using a separate rate-control algorithm. This disjoint approach used by existing video streaming systems limits the maximum achievable QoE when used in the context of emerging video streaming applications. In reality, the choices made with respect to congestion control algorithms, video coding techniques, and network transport design do not affect the QoE in isolation - these sub-systems interact with each other and have an impact on the end-user QoE in terms of video quality and the end-to-end delay. For example, the impact on video quality when using I-frames for loss recovery is affected by the bandwidth availability, and the increase in frame delay when recovering from loss using retransmissions is affected by the video decoder's performance. Applications like cloud gaming and AR/VR streaming simultaneously require low latency and high video quality, in addition to stable, predictable performance since they aim to reproduce the experience of running an application on local hardware. While these goals can be achieved to some extent under an extremely narrow set of ideal network conditions by tuning real-time video streaming techniques to work well, achieving a high QoE is a challenging task over real networks in the presence of competing traffic, packet loss, and delay and bandwidth variations.

Second, systems for emerging video streaming applications are designed by tuning the solutions for traditional video streaming application, instead of designing a tailored system that is suited to the needs of the new application. Recall that SLVS applications have viewers in real-time and viewers who view the video with some delay, and that these groups of viewers have different video quality and delay requirements. SLVS platforms today commonly use protocols like WebRTC and RTMP to transmit the video from the broadcaster to an ingestion point, after which it is processed for efficient delivery over a content delivery network. This approach that is borrowed from existing techniques used for video conferencing applications, silos both the real-time viewers and the delayed viewers into receiving the same video quality. This is because techniques like WebRTC and RMP operate at a specific point on the video quality - delay trade-off curve. Video streaming techniques that prioritize the real-time aspect of the video stream force time-shifted viewers to view a low quality video stream that is affected by short-term network issues like packet loss and bandwidth fluctuations. On the other hand, techniques that prioritize video quality and smooth playback increase the viewing delay of the "live" feed, resulting in poor interaction between broadcasters and real-time viewers. Thus, designing a system that is specifically designed to cater to the diverse QoE requirements of the viewers with different viewing delays can result in higher overall QoE across the entire set of viewers. A key factor that differentiates applications like cloud gamign from traditional low-latency applications like conferencing is that there is some level of control over the set of network conditions where these applications are run, but this is much broader than the set of network conditions where existing techniques can achieve the desired level of QoE. For example, cloud gaming platforms like Stadia [16], GeForce Now [17] and Amazon Luna [18], each specify a minimum bandwidth requirement for using their services, and will automatically terminate a streaming session when heavy packet loss, significant network delay and jitter, or extremely low bandwidth is detected, but they still need to address the challenges associated with competing queue-building cross traffic, packet loss, delay, and bandwidth variations. Such relaxations can enable new designs that are tailored to the types of networks these applications are run on, enabling optimizations that are not feasible using traditional approaches.

In this thesis, we propose various techniques for addressing the different aspects of streaming video over the Internet for emerging video streaming applications. The designs of these techniques are tailored to suit the specific needs of each application, and the designs carefully consider the interactions between multiple components of the video streaming stack. We show that such an approach can satisfy the unique QoE requirements of emerging video streaming applications, and significantly improve the QoE over what can be achieved by tuning existing techniques used in traditional video streaming systems.

.

1.3 Thesis Statement

Application-specific video streaming systems that leverage holistic cross- layer optimizations are needed to better satisfy the unique and demanding QoE requirements of emerging video streaming services like SLVS, cloud gaming and AR/VR streaming.

Video streaming systems need to make some key choices with respect to networking design and video compression algorithms, and these choices play a major role in determining the end-to-end QoE by affecting the video quality and the end-to-end video frame delay. These decisions can differ significantly based on the type of application (eg. Real-time versus pre-recorded content). These decisions include *how much*, *what*, and *when* to transmit, which cover the core technical components of a video streaming system.

Designing video streaming techniques for addressing the three key aspects mentioned above boils down to deciding the following:

- **How much to transmit.** When designing a video streaming system, an important design decision is what bitrate the video should be encoded at. The video bitrate plays a significant role when making design choices for addressing the challenges of streaming video on the Internet, like video compression, packet loss recovery, and congestion control. For example, if efficient video compression techniques are used, the video bitrate can be lowered in order to transmit additional data like FEC in order to make the video stream loss resilient. Similarly, the manner in which the video bitrate interacts with the congestion control's transmission rate affects the end-to-end video delay. In the presence of bandwidth variations, the choice of the video bitrate affects the interactivity and the video quality - if the video bitrate adapts rapidly to network variations, the delay is lower but can lead to fluctuations in video quality and video stutters, whereas if the video bitrate is based on the average network bandwidth, the video quality is higher but the frame delay is also much higher due to buffering requirements.
- **What to transmit.** A video streaming system must determine what is being transmitted in order to effectively utilize the available bandwidth, and deciding what to transmit affects the performance of the video streaming system in the presence of packet loss, and adverse network conditions. For example, a video streaming system can encode the video in a less

efficient manner (e.g. I-frames), or can split the available bandwidth across video data and FEC in order to achieve loss resilience.

- **When to transmit.** When video frames are transmitted can have a big impact on the end-to-end QoE for emerging video streaming applications. For example, using buffers and delaying transmissions of frames can result in smoother video with lower fluctuations in the video quality, but will result in higher delays. When data is transmitted also affects the performance in the presence of packet loss - for example, FEC transmits additional data before loss occurs in order to recover from packet loss, resulting in lower delays, but sacrificing video quality in the process due to a bandwidth overhead. On the other hand, retransmissions transmit lost data after loss occurs, which results in higher delays. The interactions between when the packets of a frame are transmitted and the congestion control algorithm's transmission patterns can affect the end-to-end video frame delay.

In this thesis, we present 3 systems that take a holistic approach at answering these questions in order to address the three specific aspects of video streaming - addressing the unique QoE requirements, mitigating packet loss, and performing congestion control.

We first discuss a system called Vantage for SLVS applications. Vantage uses the available bandwidth for transmitting quality-enhancing retransmissions in addition to a real-time video stream (what to transmit) in order to improve the video quality for time-shifted viewers. Vantage uses an optimization formulation in order to determine how the available bandwidth must be split between the real-time video stream and the retransmissions (how much to transmit), and which segments must be repaired using quality enhancing retransmissions (when to transmit). Vantage takes a holistic approach at designing a novel video streaming system in order to provide a tailored solution that meets the unique QoE needs of SLVS applications.

Second, we discuss a system called Prism for mitigating packet loss for cloud gaming and remote-rendered AR/VR. When packet loss occurs, Prism splits the available bandwidth across a low-latency video stream at a lower quality, and a quality-preserving video stream (what to transmit). The available bandwidth is carefully allocated across the two streams in order to maximize the video quality during loss (how much to transmit). Prism's design also includes a loss prediction mechanism in order to trigger early loss recovery (when to transmit). Prism's design choices focus on the low latency and high video quality requirements of cloud gaming and remote-rendered AR/VR applications, and it's design is optimized across layers by taking into account video compression properties, loss prediction performance and the available bandwidth.

Third, we discuss an alternate strategy for mitigating packet loss for wireless VR applications, called ViXNN. ViXNN uses a neural network-based video compression technique (what to transmit) in order to achieve error-resilient video coding. ViXNN's robustness to packet losses enables one-shot frame transmissions without relying on link-layer retransmissions (when to transmit). ViXNN is also robust to bit errors enables the use of faster (but noisier) wireless modulation schemes (how much to transmit). ViXNN is designed to work under specific environments that have high bandwidth and high loss, and integrates the design of video compression and loss resilience.

Fourth, we discuss the pitfalls of existing congestion control algorithms for cloud gaming and remote-rendered AR/VR applications that have demanding QoE requirements. We present a novel congestion control algorithm called SQP, that leverages video frame transmission patterns (when to transmit) in order to probe the network for more bandwidth. SQP uses a novel bandwidth estimation mechanism that enables it to achieve low delay when running in isolation, achieve competitive throughput shares when competing with queue-building flows, and is tightly integrated with video bitrate control (how much to transmit). SQP's design does not require unnecessary probing data in order to probe the network when enough video data is not being generated by the encoder (what to transmit). SQP's design tightly integrates the video codec and the network congestion control in order to minimize encoder bitrate variations and reduce the end-to-end video frame delay, and leverages application-specific properties (e.g. basic network quality requirements of cloud gaming applications) in order to achieve good throughput and delay performance. We also present a system called CC-Fuzz for stress testing congestion control algorithms. CC-Fuzz uses a genetic algorithm in order to synthesize network traces that trigger poor behavior in congestion control algorithms. CC-Fuzz can automatically find issues with the design and catch implementation bugs, and can inspire confidence in a congestion control algorithm before it is deployed in the wild.

Our contributions are summarized below, and detailed descriptions are discussed in later chapters.

1.4 Summary of Contributions

1.4.1 Vantage

In chapter 3, we explore the application class of social live video streaming (SLVS). SLVS has unique QoE requirements that are not addressed by existing live video streaming systems. In SLVS, viewers view a live stream with different viewing delays - some viewers watch the video stream in real time, whereas others may watch the video with some delay (time-shifted viewing) or even after the live stream session has terminated. These distinct sets of viewers have different expectations of video quality and interaction delay. For instance, viewers that interact with the broadcaster using mechanisms like live chat and monetary donations, and co-broadcasters in a multi-party live stream require low interaction delay, but can tolerate occasional stream interruptions and drops in video quality. On the other hand, time-shifted viewers have a higher expectation of video quality, and do not have any latency requirements since there is no real-time interaction (similar to Video-on-Demand content). This unique QoE requirement presents an opportunity for redesigning the live video upload mechanism in order to satisfy the different QoE requirements of different viewers who consume the content differently.

Our first contribution, Vantage [1], is a system that proposes a novel bandwidth allocation and video frame transmission scheduling mechanism (bullet no. 1) in order to address the unique QoE requirements of SLVS applications - real-time viewers require low latency whereas time-shifted viewers expect higher video quality. A key challenge with streaming real-time video over the Internet, especially from mobile devices on cellular networks, is that network bandwidth is highly variable. In order to achieve low latency, it is desirable to adapt the video bitrate such that it closely follows the instantaneous available bandwidth, and use minimal buffering on both, the sender- and the receiver-sides. The downside of this approach is that there can be undesirable fluctuations in video quality, and network interruptions can cause stuttering in the video stream. While this is acceptable for real-time viewers, existing SLVS platforms simply archive the same video stream for consumption by time-shifted viewers. This results in a less than optimal experience for time-shifted viewers. On the other hand, if video is encoded at a bitrate that matches the average bandwidth, and sufficiently large transmit and playback buffers are used at the sender side and receiver side respectively, the fluctuations in video quality are significantly reduced and much smoother video playback can be achieved. The downside of this approach is that it adds a significant amount of delay between the broadcaster and the real-time viewers.

The key idea in Vantage is to split the available bandwidth across two streams - a real-time stream that enables low-latency interaction between the broadcaster and the real-time viewers, and a secondary quality-enhancing video stream that improves the video quality for delayed viewers. Vantage leverages a key property of lossy video compression algorithms - *increasing the video bitrate has decreasing returns in terms of video quality.* Vantage reduces the bitrate of the real-time video stream slightly, and carefully allocates the residual bandwidth to a secondary "quality enhancement" video stream. The purpose of the quality enhancement video stream is to repair past segments that were affected by network variations and transient outages. In order to optimizing the video quality for different viewing delays, Vantage uses an online optimization formulation in order to allocate the total available bandwidth across the low latency stream and the quality enhancing retransmissions, carefully taking into account the relationship between video bitrate and quality, the distribution of viewing delays and the available bandwidth. Vantage's design enables it to operate at multiple points on the quality-delay trade-off curve simultaneously, *allowing the viewers to choose* any point on the video quality and interaction delay trade-off curve. Viewers who care about low delay are able to do so, albeit by sacrificing some video quality, and time-shifted viewers are able to view higher quality version of the live stream.

This work was done in collaboration with my advisor (Srinivasan Seshan), Jack Kosaian, and Rashmi Korlakai Vinayak. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant Nos. DGE-1745016 and DGE-1252522, the National Science Foundation Computer and Network Systems under Grant No. CNS-1565343, and the Facebook Communications and Networking Research award. CloudLab [19] resources were used in running experiments. We would also like to thank Keith Winstein for providing valuable feedback during the shepherding process.

1.4.2 Prism

In Chapter 4, we explore the problem space of handling packet loss for low-latency interactive applications like cloud gaming and AR/VR streaming. Cloud gaming and cloud AR/VR applications aim to replace local versions of applications like video games and AR/VR applications by cloud-rendered versions, where the heavy computational aspects like rendering is offloaded to powerful machines in the cloud, and the rendered view-port is streamed as compressed video to the end user. Since cloud rendering aims to replace the local counterparts, the experience must be as close as possible to the local versions in terms of visual quality and the interaction latency. Thus, cloud-rendered applications require extremely low latency and very high visual quality. Since cloud-rendered applications are streamed over the internet, they invariably experience packet loss. Packet loss has a significant impact on video quality and video frame delay due to the complex interactions between video compression techniques and loss recovery mechanisms. This is extremely detrimental to the QoE of low-latency video streaming applications like cloud gaming and cloud AR/VR.

Our second contribution, Prism, is a system that proposes a novel packet loss recovery mechanism (bullet no. 2) in order to achieve the stringent quality and latency demands of cloud-rendered applications. Video compression techniques leverage temporal redundancies in video data in order to achieve high compression ratios, utilizing the available bandwidth to provide the best possible video quality. This results in temporal dependencies between video frames (predicted- or P-frames), and this makes low-latency compressed video extremely sensitive to packet loss. If a single frame is lost, the subsequent video frames cannot be decoded until the lost frame has been recovered, or the video codec state is reset using an independently encoded frame (intra- or I-frame). Using retransmissions to recover the lost data adds significant latency and causes video stutters, whereas independently encoded frames that can reset the decoder state have much lower video quality for the same bitrate since they do not leverage temporal redundancies in video data. In addition, due to the nature of video compression algorithms, the quality of a frame depends on the quality of past frames as well - thus, a single independently encoded frame affects the quality of multiple subsequent frames. Video skips, freezes, and quality drops are detrimental to the QoE of cloud-rendered applications - users of cloud rendered applications expect the experience to be as close as possible to running the application locally.

Prism's design uses a hybrid dual-stream video encoding and transmission scheme, leveraging various properties of video coding like the difference in coding efficiency of I-frames and P-frames and the temporal dependency of the quality of P-frames. When packet loss occurs, Prism transmits two separate video streams - a primary stream that leverages the compression efficiency of P-frames to preserve high quality of the video stream, and a secondary low-latency stream that reduces the end-to-end frame delay during periods of transient loss. The available bandwidth is carefully allocated across the two sub-streams (bullet no. 1) in order to optimize the overall video quality. The goal of the optimization is to improve the video quality over simply using I-frames for recovery, while Prism's dual stream architecture ensures that the latency is comparable to that of using I-frames. In order to make this approach feasible in real-time, Prism's bandwidth allocations are based on the results of an offline analysis pipeline that analyzes the video encoding properties of the specific game style or scene. This aims to optimize the allocation for particular scene types, since the type of content significantly affects the quality-bitrate trade-offs for I-frames and P-frames, and thus, the optimal bandwidth allocation across the two streams during periods of loss. Our experiments show that Prism can achieve good performance over lossy networks, and has zero run-time overhead when good network conditions prevail.

A second benefit of Prism is that the video quality penalty when loss recovery mechanism is falsely triggered is significantly minimized compared to using I-frames for recovery. This enables aggressive loss prediction mechanisms, leading to further reductions in the end-to-end video frame delay during periods of loss. We implemented a simple neural-network based packet loss prediction mechanism based on real-world network traces, and use this in conjunction with Prism in order to demonstrate it's benefits when used for packet loss recovery in cloud-rendered applications (e.g. cloud gaming and cloud AR/VR).

This work was done under the guidance of my advisor, Srinivasan Seshan, and funded by National Science Foundation Grant No. CNS-1956095.

1.4.3 ViXNN

We present an alternative approach for handling packet loss for extremely delay sensitive applications like wireless VR. Wireless VR operates over local networks, with a last-hop wireless link between the router and the VR headset. We propose an end-to-end loss-resilient video compression technique called ViXNN. In ViXNN, we design a novel neural network architecture for compressing video data in a manner that is resilient to packet losses, instead of the traditional approach of tacking on loss-recovery mechanisms onto existing video codecs.

ViXNN's neural network architecture uses a convolutional autoencoder in order to compress video, but includes a novel design of the narrow waist that outputs the compressed representation of the data. ViXNN's narrow waist has multiple outputs called descriptors, where each descriptor represents a portion of the compressed data. During training, we simulate the loss of descriptors, which forces the neural network to learn a loss-resilient compression scheme. As a result, the quality of the decompressed frame degrades gracefully as the number of descriptors are lost. In addition, due to the inherent resilience of neural-network-based representations to bit errors, the compressed descriptors are also resilient to bit errors. Additionally, compressed data can be treated as analog data, which allows the use of approaches like SoftCast [20], which provides further reduction in the end-to-end frame delay by eliminating link-layer retransmissions.

In our approach, we focus on compressing each individual frame separately, which is unable to leverage the temporal redundancies in video data. With recent advances in end-to-end neural video compression, ViXNN's core design can be leveraged in order to achieve highly efficient, practical end-to-end loss resilient video compression for extremely low latency applications.

This idea originated from a class project in the course Visual Computing Systems, taught by Kayvon Fatahalian. This work was done in collaboration with my advisor (Srinivasan Seshan), Pratik Fegade, and Kayvon.

1.4.4 SQP

We explore the design of congestion control algorithms for low latency applications like cloud gaming and cloud AR/VR. Cloud gaming applications require extremely high video quality and very low end-to-end frame delay. Thus, congestion control algorithms designed for cloud gaming applications must have low queuing delay, must achive high throughput under variable links and in the presence of queue-building cross traffic, and must be compatible with the video frame traffic pattern in order to minimize the end-to-end frame delay and maximize the utilization of the available bandwidth.

We present SQP, a congestion control algorithm developed in collaboration with Google for Google's AR streaming platform. SQP is designed to work in harmony with the traffic pattern of low latency video, and takes a different approach to solving the problem of conflicting goals of minimizing delay and maximizing throughput in the presence of queue-building flows. SQP couples the transmission of video frames and network probing, and provides bandwidth estimates that can be directly used to set the video bitrate. SQP's frame coupled design enables it to achieve low end-to-end frame delay in addition to low packet delay. SQP uses frame pacing and a unique one-way-delay based formulation for estimating the bandwidth, which addresses the challenges with using a packet-train-based bandwidth measurement mechanism. SQP's design includes mechanisms to improve convergence to fairness. SQP makes key application-specific trade-offs that enable SQP to achieve good throughput when competing with queue-building flows, which enables higher video quality - this is critical for cloud gaming and cloud AR/VR applications. SQP demonstrates fairness in competition with homogeneous traffic, and works well on shallow buffers. We theoretically analyze SQP's behavior, and validate the analysis through various emulated experiments. We evaluated SQP on real-world wireless networks via Google's AR streaming platform. Under A/B testing of SQP and Copa [21] on Google's AR streaming platform, SQP achieves significantly more sessions that have acceptable video bitrate and end-to-end frame delay. We also evaluated SQP under a variety of emulated network scenarios. Across emulated wireless traces, SQP's throughput is 11% higher than Copa (without mode switching) with comparable P90 frame delays, while Copa (with mode switching), Sprout [22], and BBR [11] incur a 140-290% increase in the end-to-end frame delay relative to SQP. In addition, SQP achieves high and stable throughput when competing with buffer-filling cross traffic. Compared to Copa (with mode switching), SQP achieves 70% higher P10 bitrate when competing with Cubic [12], and 36% higher link share when competing with BBR.

This work was done in collaboration with Google under the guidance of my advisor, Srinivasan Seshan, and includes contributions from Connor Smith (Google), David Chu (Google), and Teng Wei (Google).

1.4.5 CC-Fuzz

We explore the challenges of implementing congestion control and integrating it with video bitrate selection (bullets 1 and 3) for low-latency interactive applications like cloud gaming and AR/VR streaming. Cloud-rendered applications simultaneously require high video quality and low end-to-end frame delay, and we show that existing congestion control algorithms are unable to achieve these dual goals. Traditionally, congestion control algorithms have been designed to achieve maximum throughput and fairness, while preventing congestion collapse. These algorithms often exhibit queue-building behavior, which limits their applicability for cloud-rendered applications due to excessive network queuing delays. Recently, many low-latency congestion control algorithms have been proposed. We show that these video-agnostic approaches are not suitable for cloud-rendered applications. Specifically, we show that these algorithms exhibit poor performance when competing with queue-building flows (which are ubiquitous in the wild), and, at frame-level timescales, exhibit packet transmission patterns that make it challenging to integrate congestion control and video frame encoding and transmission.

Many of these limitations stem from the fact that most congestion control algorithms are designed to be as generic as possible, and we discuss the potential benefits of designing application-specific congestion control algorithms that account for the scope of environments in which applications like cloud gaming and cloud AR/VR are expected to operate. For example, if a network link is severely bandwidth limited, or has very high latency/packet loss, cloud-rendered applications are not expected to operate under such environments. The potential to incorporate such non-traditional trade-offs may allow congestion control algorithms to achieve the desired performance goals of latency and bandwidth by limiting the scope of operation.

A challenge with developing custom congestion control algorithms for specific applications has it's own challenges. Traditional congestion control algorithms have the benefit of massive world-scale testing across multiple years of usage, which helps in slowly ironing out bugs and other issues over time. On the other hand, designing, testing and deploying a new congestion control algorithm on a small-to-medium scale is very challenging, since there are limited opportunities for testing in the real-world. Our third contribution, CC-Fuzz, aims to address this challenge. CC-Fuzz is a congestion control testing framework that uses a genetic algorithm in order to automatically test congestion control algorithms by generating adversarial network traces that trigger poor behavior in a congestion control algorithm. CC-Fuzz enables the designer of a congestion control algorithm to automatically find situations where the congestion control algorithm is unable to meet the performance goals. For example, CC-Fuzz can automatically generate network traces where a congestion control algorithm cannot maintain low delay, or is unable to achieve high throughput, or results in catastrophic packet loss. Such testing can inspire confidence in the design and implementation of a congestion control algorithm, and catch many bugs early on without warranting a large scale deployment.

Chapter 2

Background

In this section, we will briefly describe the designs of some traditional video streaming systems and discuss their limitations. We will also briefly discuss some background on video compression algorithms and the trade-offs between different video encoding parameters, video quality, bitrate and video encode/decode performance.

2.1 Video Streaming Systems

Video streaming systems can be categorized into three broad categories:

1. **Stored Video or Video-on-Demand (VOD).** VOD streaming systems are designed to disseminate stored, pre-recorded video to viewers for media consumption (eg. YouTube, Netflix). VOD systems prioritize high-quality, smooth playback of stored media and aim to simplify large scale distribution of video content. Some examples of VOD systems include MPEG-DASH and HLS.

   Like any video streaming system, VOD systems must make decisions regarding how much to transmit, what to transmit and when to transmit (Section 1.3). The workflow of a typical VOD system is outlined below:

   - Video is first encoded into short chunks, typically 2-4 seconds long. Each chunk is encoded at different bitrates at varying qualities. The advantage of this approach is that no real-time video encoding is required, and the chunks can be efficiently disseminated using content delivery networks (CDNs).
   - When streaming a video, the bitrate of the chunk is chosen based on network performance using adaptive bitrate algorithms (ABR).
   - The video chunks are transmitted using TCP [23] using a congestion control algorithm like Cubic or BBR. Packet loss is handled at the TCP layer using retransmissions, since there is no requirement for very low latency, and the clients typically maintain a multi-second long buffer for playback.
   - The ABR algorithm also makes decisions regarding pre-fetching of chunks, and how much buffer should be maintained on the client side for a smooth, stutter-free playback.

- The receiver decodes entire chunks, and queues the decoded frames in a buffer for displaying on screen.

This document focuses on applications that have a real-time streaming component as opposed to purely VOD style applications. These application categories are discussed below.

2. **Real-time Video Streaming.** Real-time video streaming applications are designed for interactive applications like cloud gaming and video conferencing. These systems prioritize low latency over video quality in order to enable real-time interaction. Some examples of real-time streaming systems include WebRTC, RTMP, and proprietary streaming systems used by cloud gaming services like Stadia, Luna and GeForce Now.

   Real-time streaming systems must also make the same key decisions we talked about in Section 1.3, but make very different decisions in order to achieve their application QoE goals. A typical frame in a real-time streaming system goes through the following steps:

   - Frames are generated by an application or device (eg. Video game or a rendered application for cloud gaming and AR/VR streaming, camera device in the case of conferencing).
   - Based on a bandwidth estimate from the encoder rate control mechanism and the congestion controller (eg. GoogCC for WebRTC [13]), the video frame is encoded by the encoder and packetized for transmission.
   - The packetized video frames are queued for transmission. Some applications may make use of techniques like forward error correction (FEC) to enable recovery when packet loss occurs.
   - The packets are transmitted by the transport layer, where the congestion control algorithm determines how the packets are transmitted over time (eg. pacing, ACK clocking).
   - The receiver receives the packets and re-assembles them into encoded frames, which are then decoded and immediately displayed on the screen in order to minimize latency.
   - When packets are lost and the decoder is unable to decode the video, it signals the sender to start frame-loss recovery. This may take the shape of packet retransmissions to retransmit the lost segments, or the transmission of an IDR frame, which skips forward in time to the most up-to-date frame.
   - Feedback from the receiver is used by the congestion control algorithm and the encoder rate control algorithm to determine the video encoder bitrate.

   While applications like conferencing can tolerate delays of the order of a few hundred milliseconds, cloud gaming and AR/VR streaming applications require the delay to be of the order of a few tens of milliseconds. In addition, cloud gaming and AR/VR streaming also operate at much higher bandwidths since they require much higher video quality. In this thesis proposal, we aim to design new congestion control and packet loss recovery mechanisms in order to achieve the desired QoE for emerging applications as opposed to using existing video streaming systems that are used for traditional real-time applications like video conferencing.

3. **Real-time and VOD Hybrid Video Streaming.** Hybrid video streaming systems are designed for applications that have a real-time interactive component as well as time-shifted viewers. An example is social live video streaming, where some viewers view the video stream in real-time and interact with the broadcaster, while a second set of viewers view an archived version of the video stream. Traditional video streaming systems designed for hybrid video streaming typically use the following workflow:
   - Applications use a real-time streaming protocol (eg. WebRTC) between the video streamer and an initial ingestion point.
   - The video at the ingestion point is transcoded in real-time for the real-time viewers. The transcoded video is distributed using low-latency live streaming protocols like LLDASH [24], LLHLS [25], and RTP [26].
   - An archival version is also simultaneously generated where the video is encoded into chunks at varying bitrates (similar to VOD-style video streaming), which can be viewed by viewers who watch the live stream with larger delays in order to achieve efficient content distribution.

   Existing real-time and VOD hybrid video streaming systems use traditional real-time video streaming techniques on the upload path. This thesis aims to develop novel video upload mechanisms that account for the diversity in viewing delays in order to optimize the video quality across the time-shifted viewers.

2.2 Video Compression

Video compression techniques are at the heart of video streaming systems and applications. The primary goal of a video compression scheme is to reduce the amount of bytes required to transmit the video without significantly affecting the perceived quality of the video frames. In addition to the bandwidth requirement and the video quality, video compression schemes also differ in terms of their sensitivity to packet loss, the video encoding-decoding speed, and compute and memory requirements.

In this section, we discuss some common techniques that are used for video compression and how these impact various factors like video quality, bandwidth requirements, video encode-decode performance, and sensitivity to packet loss in the context of video streaming applications.

2.2.1 Overview of Compression Schemes

Video compression is made possible due to two key factors - (1) video data has a significant amount of spatial and temporal redundancy, and (2) humans are less sensitive to higher frequency spatial data. Most video compression schemes combine the use of lossless compression and lossy compression. The use of lossless compression leverages the spatio-temporal redundancy in video data, whereas the lossy step leverages the human factor in order to further improve the compression ratio. A common way to compare the compression performance of a particular scheme is to compare the relationship between the video picture quality and the bitrate of the compressed video. Video quality is typically measured using metrics like SSIM [27] (Structural

SIMilarity) and PSNR (Peak Signal-to-Noise Ratio), where each compressed video frame to the original video frame and aggregated across a video. In this section, we will first discuss a scheme called Motion JPEG [28] (MJPEG) that only leverages spatial redundancy for the lossless compression component, and then discuss the high-level video compression architecture used by most modern video codecs.

2.2.2 Intra-frame Schemes

Motion JPEG (M-JPEG) is an intra-frame only compression scheme: each frame is compressed as an image using JPEG. Due to the wide-spread availability of optimized JPEG compression libraries and hardware, this was commonly used in PC multimedia applications, and in certain older digital cameras and security cameras.

Since M-JPEG compresses each frame individually, it is unable to leverage the temporal redundancy in a video stream that arises from most adjacent frames of a video being very similar. Thus, it requires more bits to encode video at a given quality compared to schemes that leverage the temporal redundancy in video data.

The compression mechanism of M-JPEG is identical to JPEG. JPEG splits the frame into 8x8 blocks. These blocks undergo a DCT transform, following by a lossy quantization step, where the high frequency components are scaled down and quantized. These values are then entropy-coded using techniques like Huffman coding and arithmetic coding, which serves the purpose of leveraging spatial redundancies for additional compression.

There are a few benefits of using an intra-frame only compression scheme. The first benefit is that the quality of a video frame for a given bitrate only depends on the static contents of the frame, and not on the relative motion between frames. It does not matter if two consecutive frames are very different, since each frame is independently compressed. The second benefit is that if some packets are lost during streaming, that frame can simply be skipped and the next frame can be decoded and displayed. In addition, this approach inherently supports temporal scaling: frames can be dynamically dropped if the receiver cannot keep up with the decoding of frames.

Modern video codecs use other techniques for intra-frame compression, but they are very similar to JPEG compression in principle. For example, intra-frame compression in H.264 [29] involves representing blocks that are similar to other blocks as translations, and compressing the residuals in a manner that is similar to JPEG. This improves the compression ratio for frames where large areas have a similar color and repeating patterns.

2.2.3 Inter-frame Schemes

Inter-frame schemes use a technique called motion compensation to leverage the temporal redundancies in video data. Motion compensation significantly improves the compression ratio that can be achieved when compressing video data. Most modern codecs like H.264 [29], H.265 [30], VP8 [31], VP9 [32] and AV1 [33] leverage inter-frame compression in addition to the intra-frame compression techniques discussed in § 2.2.2.

The key idea in motion compensation is to search for similar blocks in adjacent frames that come before and after the current frame. The blocks of the frame being encoded are represented as motion vectors from the similar blocks in the adjacent frames. Since the reference blocks are not identical, but only similar to the blocks being encoded, the residual error is encoded using block compression techniques discussed in § 2.2.2.

2.2.4 Motion Compensation and Streaming Video

While inter-frame compression has significant benefits in terms of compression ratios, there are a few important considerations when motion compensation-based techniques are used for streaming video. To understand these limitations, we must first understand how intra-frame and inter-frame techniques are combined for video compression.

The first frame in a video stream is always an intra-only compressed frame (also called an IDR frame, or independent data refresh frame). The role of an IDR frame is to initialize the state of the video decoder. For improving the compression efficiency, the subsequent frames use motion compensation. There are two kinds of motion-compensated frames, P-frames (predicted, only use past frames as reference) and B-frames (bidirectional, use past and future frames as reference). While B-frames achieve better compression efficiency and are useful for VOD style content where the video is pre-encoded into 2-4 second chunks, they are not typically used for real-time streaming video. This is because the encoder would have to wait for future frames to encode a particular frame, which increases the end-to-end video frame delay. Real-time streaming applications encode new frames as P-frames, where blocks in the frame are approximated from past frames.

The first issue with motion compensation is the additional computational overhead required for searching motion vectors. While compression efficiency can be improved by performing a more comprehensive search for motion vectors, this can significantly increase the time required to encode a frame, resulting in higher frame delays. The second issue arises when packet loss occurs, causing the partial or total loss of a P-frame. Since subsequent frames depend on the P-frame, the decoder cannot proceed until it can decode the P-frame or a new IDR frame is sent, which can cause higher delays or lower video quality respectively.

Recovering from packet loss using retransmissions and FEC can have a significant overhead in terms of delay and bandwidth requirements, and hence, the inherent loss resiliency of a video compression scheme is an important consideration when designing real-time video streaming applications that have stringent delay requirements and depend on the real-time bandwidth availability.

While some compression schemes are able to deal with packet loss during real-time streaming, some schemes may require extraneous mechanisms like retransmissions or special codec-level recovery procedures that may result in worse end-to-end delay, require more bandwidth, or suffer from poor video quality. Many available video codecs have features that can improve the loss resiliency of the video stream, but they can result in lower compression efficiency, may require higher computational overhead, and may not meet the requirements for some applications.

For example, Intra-refresh is a common technique for improving the loss resilience of a video stream, where a different part of each frame is transmitted as an I-frame, covering the entire display area across multiple frames. Thus, if some packets are lost, the video impairment lasts for a few frames until the I-frame segments span the entire video area. The limitation of this approach is that each frame requires additional bits for the same video quality, and a video impairment will persist for the duration of a few frames, which is unacceptable for applications like cloud gaming and AR/VR streaming.

Other techniques like SVC (Scalable video coding) and MDC (Multiple Descriptor Coding) also improve the loss resilience, but have a similar impact on compression efficiency and also require significantly higher computational overhead.

Chapter 3

Vantage: Optimizing video upload for time-shifted viewing of social live streams

Social live video streaming (SLVS) applications are becoming increasingly popular with the rise of platforms such as Facebook-Live, YouTube-Live, Twitch and Periscope. A key characteristic that differentiates this new class of applications from traditional live streaming is that these live streams are watched by viewers at *different delays*; while some viewers watch a live stream in real-time, others view the content in a time-shifted manner at different delays. In the presence of variability in the upload bandwidth, which is typical in mobile environments, existing solutions silo viewers into either receiving low latency video at a lower quality or a higher quality video with a significant delay penalty, without accounting for the presence of diverse time-shifted viewers.

In this chapter, we present Vantage, a live-streaming upload solution that improves the overall quality of experience for diverse time-shifted viewers by using selective quality-enhancing retransmissions in addition to real-time frames, optimizing the encoding schedules to balance the allocation of the available bandwidth between the two.

3.1 Social Live Video Streaming (SLVS)

Mobile live video traffic has grown significantly over the last decade [6]. This growth has been propelled by improvements in mobile camera technology, computing power, and wireless technology, which enables the capture, encoding, and transmission of high-quality video in real-time from mobile devices. Applications for video-conferencing and live broadcasting have become ubiquitous on mobile devices today. Social live video streaming (SLVS) applications like Facebook Live [34], YouTube Live [35], and Periscope [36], are a new and increasingly popular class of applications that bring the power of live streaming to individuals.

A unique feature that differentiates SLVS platforms from traditional live video applications is the ability to view real-time, time-shifted, and archival versions of a single stream. SLVS applications enable viewers to interact with broadcasters via comments and reactions in real-time [37, 38], and also archive the video to enable viewing after the live streaming session has ended. Furthermore, platforms like Hangouts-on-air [39] and Facebook Live [40] allow multiple users to broadcast simultaneously on a single live-stream. For viewers using interactive features such as real-time comments, as well as broadcasters taking part in collaborative broadcasting [39, 40], it is critical to deliver the video stream at a low-latency, whereas a higher streaming latency is acceptable for the other viewers.

In contrast, traditional live video streaming applications target a *single* viewing delay. In video conferencing applications, participants require low-latency for interactivity, while viewers of a broadcast event can typically tolerate tens of seconds of delay. Existing approaches for handling network bandwidth variations are *tailored for one particular viewing delay*. In low-latency applications like videoconferencing, video bitrate is chosen to closely follow the available bandwidth in order to ensure that frames are received before their real-time playback deadline, at the expense of lower quality during periods of low bandwidth. On the other hand, when higher delays are acceptable, applications use buffers to absorb network variations and the video bitrate is chosen to match the average bandwidth. This results in higher video quality and smoother playback at the cost of higher latency.

Due to lack of better alternatives, current SLVS platforms make the same tradeoffs between latency and quality as traditional live streaming, despite the diversity of viewing delays. Operating at a single point on the latency-quality tradeoff spectrum is inadequate for providing high quality-of-experience (QoE) for all the viewers of SLVS streams. The problem is further exasperated by the fact that SLVS streams are commonly initiated from mobile devices, which have particularly unpredictable network behavior [22].

3.2 Overview of Contributions

Vantage is a live video upload solution explicitly designed to address the time-shifted viewing characteristic of SLVS in the face of bandwidth variations. Vantage exploits the variability of the upload path to it's advantage: periods with high bandwidth can be used to correct for a loss in quality due to previous periods of network impairment. Vantage optimizes the video upload process across different time-shifted viewing delays by using *quality-enhancing retransmissions* in conjunction with a low-latency video stream. Vantage formulates bitrate selection and transport scheduling as a joint optimization problem that maximizes the video quality across the diverse viewing delays.

Several challenges need to be addressed to make the use of quality-enhancing retransmissions practical and effective: (1) allocating bandwidth and scheduling transmissions for the real-time and retransmitted frames such that the QoE is optimized for all users, (2) handling the computational overheads and latencies associated with complex optimization decisions and video compression, and (3) dealing with the network unpredictability and its impact on scheduling decisions. Vantage incorporates several system design choices like approximations, pipelining, and fallback mechanisms to handle the challenges related to optimization and unpredictability.

We have implemented Vantage and evaluated it on a wide variety of mobile network traces [3] and videos [4]. Our evaluation shows that Vantage achieves high quality for low-latency and time-shifted viewing *simultaneously*. Specifically, for delayed viewing, Vantage achieves an average improvement of 19.9% over real-time optimized streaming techniques across all the network traces and test videos, with observed gains of up to 42.9%. The quality achieved by Vantage

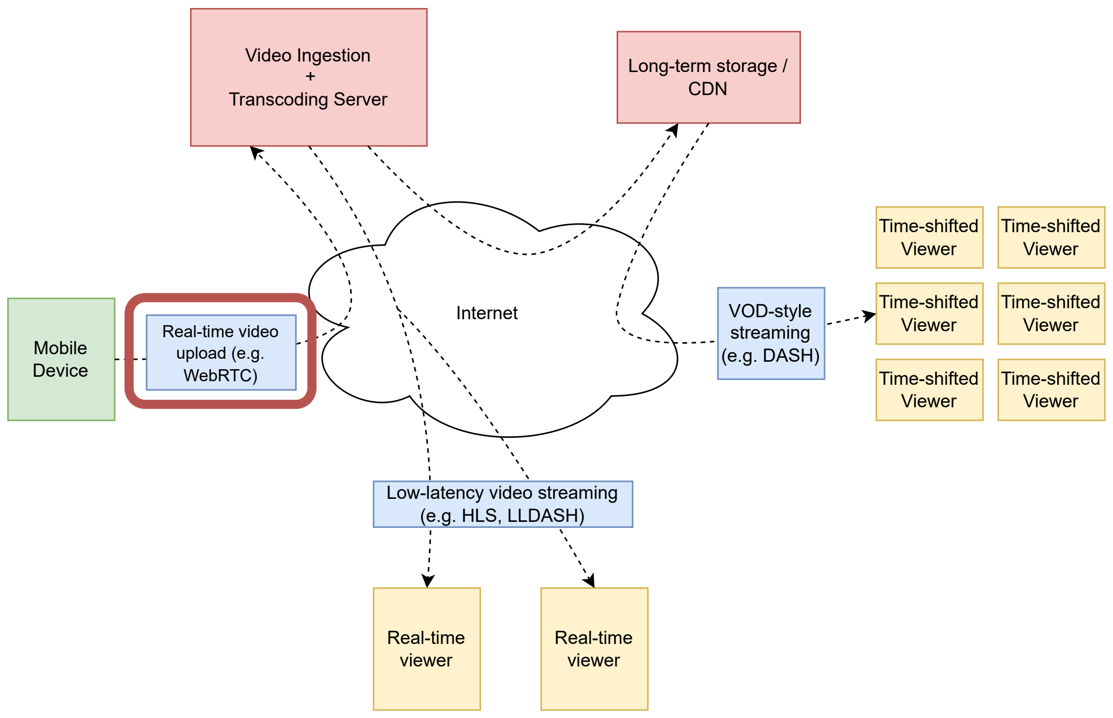


Figure 3.1: High-level architecture of social live video streaming systems. The upload path (focus of this chapter) is highlighted with a red box.

for delayed viewing is within 7.7% on average of the maximum quality achievable by delay tolerant techniques. These benefits come at the cost of an average drop 3.3% in the real-time quality, with a maximum drop of 7.1%. These results demonstrate the significant performance benefits of using Vantage over current techniques used for SLVS applications, which primarily optimize the video upload for real-time viewing.

3.3 Background and Opportunity

In this section, we discuss SLVS architectures and the network variability observed on mobile upload paths used in SLVS.

3.3.1 SLVS Architectures

We describe common designs and practices employed by four of the most widely used SLVS platforms as of 2019: Facebook Live, YouTube Live, Twitch, and Periscope. Our descriptions are informed by recent studies [37] and industry engineering material [41, 42].

The high level architecture typically used by social live video streaming platforms is shown in Figure 3.1. Live video is captured and encoded by a broadcaster's device (e.g., a mobile phone) and uploaded via RTMP [43] or WebRTC [44] to an ingestion point (a point-of-presence or a data center server), where the upload path connection is terminated. We refer to this ingestion point as the *upload endpoint*. The ingested video is re-encoded at the upload endpoint. The re-encoding serves two purposes - first, for real-time viewers, the video is re-encoded into a format that supports large-scale distribution with low delay (e.g. MPEG-LLDASH, HLS), and for the delayed viewers, the video is re-encoded to support efficient VOD-style delivery (e.g. MPEG-DASH) using CDNs. A variety of techniques are used for distributing the video after ingestion, and these techniques have been widely studied in the past [45, 46, 47].

The key limitation of this approach is that the video is only uploaded once, using a real-time streaming protocol on the upload path (e.g. WebRTC [48] or RTMP [43]) - this means that the video re-encoding for both, the real-time viewers and the time-shifted viewers use the same source material. The quality of the source material is affected by network fluctuations on the upload path. Thus, network disruptions during video upload also affects the time-shifted viewers.

This paper focuses on improvements for the *upload path* of SLVS applications (highlighted using a red box in Figure 3.1) in order to support the different QoE requirements of the real-time and time-shifted viewers.

3.3.2 Time-shifted viewing in SLVS

SLVS differs from traditional live-streaming in that it enables viewing of the same video stream at different delays. Traditional live streaming applications are tailored either for interactive, low-latency streaming (e.g., Skype and Hangouts) or for high quality viewing at larger delays (e.g., ESPN and CNN). On the other hand, SLVS platforms enable both real-time and delayed viewing of the same stream. We term this characteristic “time-shifted viewing.”

Time-shifted viewing takes a number of forms within an SLVS stream. Some viewers interact with broadcasters via comments and reactions, and thus require real-time latencies [37, 38]. SLVS platforms also archive video streams to allow for viewing after the live stream has ended, and also allow viewers to seek back to older segments during the live stream and watch the video with a time-shifted delay. Moreover, collaborative broadcasting platforms like Hangouts-on-air [39] and Facebook Live [40] allow collaborative streaming where the co-broadcasters have stronger low latency requirements compared to the viewers.

In summary, SLVS streams have audiences with a wide variety of viewing delays, and thus have varying degrees of latency tolerance. This presents a new and important dimension for improving the quality of experience of SLVS platforms.

3.3.3 Variability in the upload path

Many SLVS broadcasts are initiated from mobile devices over cellular networks like LTE, which experience frequent bandwidth fluctuations [22]. One such cellular LTE trace is shown in Figure 3.2. This forces the broadcaster’s device to either adapt the bitrate of encoded video (i.e., alter quality), or use large sender side buffers (i.e., alter the delay of transmission).

To illustrate the variability of bandwidth in mobile uplinks, we analyzed the network traces from the Mahimahi [3] project. Across the eight upload traces, we made the following observations:

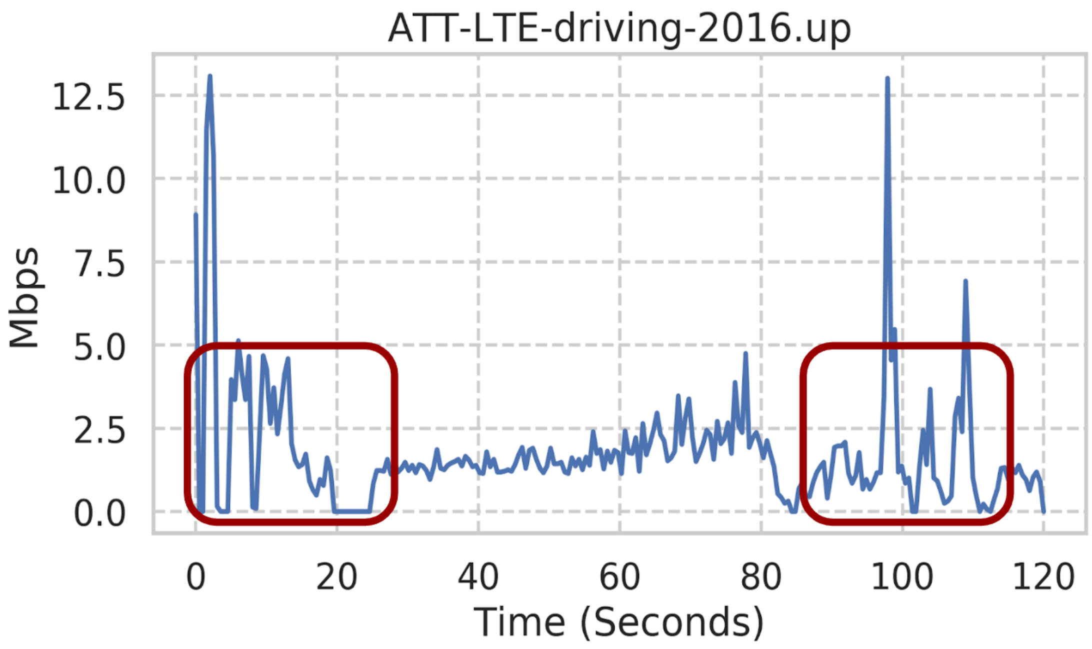


Figure 3.2: An example bandwidth trace from MahiMahi [3] for an LTE cellular link.

1. **Periods of low/high bandwidth are common:** 17.1% and 17.4% of the time, upload bandwidth is 50% or less and 150% or more than the average for a particular trace, respectively.
2. **Periods of low/high bandwidth are short-lived:** Periods with less than 50% and more than 150% of the average bandwidth last, on average, 789 ms and 809 ms, respectively.
3. **More bandwidth is gained during high periods than is lost during low periods:** In five out of the eight traces, we find that there is at least $1.25\times$ additional bandwidth when the bandwidth is above 150% of the average than the amount lost when bandwidth drops below 50% of the average.

These observations suggest that periods of high bandwidth can be exploited to improve the quality of frames that were previously affected by periods of low bandwidth. In Figure 3.2, some regions where the bandwidth is highly variable are highlighted using red boxes. In the next section, we show how an SLVS upload solution can make use of these properties to improve the QoE for all time-shifted viewers.

3.4 Supporting Time-shifted Viewing

In this section, we show that conventional live-streaming upload techniques are inadequate for providing high QoE for applications that support time-shifted viewing. We then describe an approach for providing high QoE for viewers at different time-shifted delays.

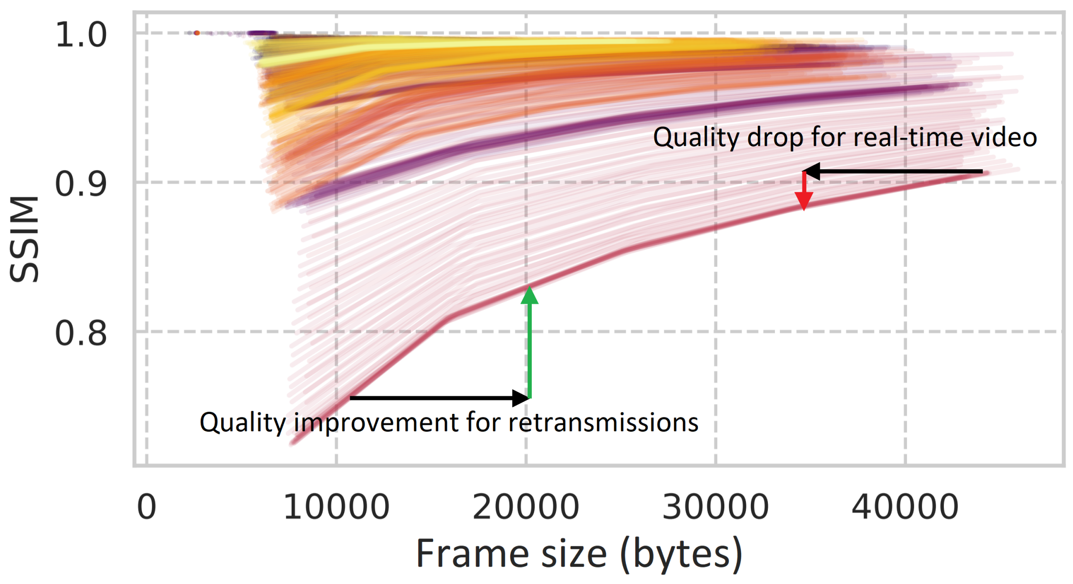


Figure 3.3: SSIM as a function of the frame size for the Sintel trailer [4, 5]. Each line corresponds to a single frame. The video was encoded multiple times with different target bitrates to generate the data.

3.4.1 Inadequacy of existing techniques

We use a simple example to demonstrate the inadequacy of existing live streaming upload techniques at providing high QoE for diverse time-shifted viewers. Consider a broadcaster capturing and uploading an SLVS stream to an upload endpoint. Consider a $20\,\mathrm{second}$ period where the upload bandwidth between the broadcaster and the upload endpoint is $0.5\,\mathrm{Mbps}$ during the first 10 seconds and $3\,\mathrm{Mbps}$ during the final 10 seconds, as depicted in Figure 3.4a.

The structural similarity (SSIM) index is a common metric used to measure the perceptual quality of video frames. The relationship between video bitrate and SSIM is typically observed to be a concave, non-decreasing function [49]. Figure 3.3 shows the relationship between the frame size and the SSIM for each frame of the Sintel [4, 5] trailer. To keep the discussion in this section simple, we use a hypothetical model that reflects this relationship between the video bitrate (b) and the SSIM (Q). Specifically, we assume $Q = 1 - \frac{1}{2b+1}$ for each video frame. In Sections 3.4.1.1 and 3.4.1.2, we analyze the SSIM of the video received at the upload endpoint when transmitted using conventional live upload techniques.

3.4.1.1 Uploading for delayed viewing

We first consider uploading a live stream using techniques commonly used for applications like ESPN and CNN, which typically deliver live content to viewers with up to few tens of seconds of delay. It is common for such techniques to adapt the video bitrate slowly in response to changes in the network bandwidth, and to buffer frames when there is insufficient bandwidth for transmission at the target bitrate. There has been a significant amount of work [50, 51, 52] in the space of bitrate adaptation and buffer allocation for video streaming applications that do not have strict low latency requirements.

For the sake of this example, assume that the uploading client has knowledge of the average bandwidth during the next $20\,\text{seconds}$, which in this case is $1.75\,\text{Mbps}$. The client thus encodes each frame at $1.75\,\text{Mbps}$. Figure 3.4b (purple dotted line) depicts the video quality using this strategy for delayed viewing. Between $0\,\text{s}$ and $10\,\text{s}$, the available bandwidth is lower than the target bitrate. The uploading client consequently buffers frames during this period and begins draining the buffer when the available bandwidth increases at $10\,\text{s}$. This results in an average SSIM of 0.777 for the entire video when viewed at delays of $20\,\text{seconds}$ or more.

3.4.1.2 Uploading for real-time viewing

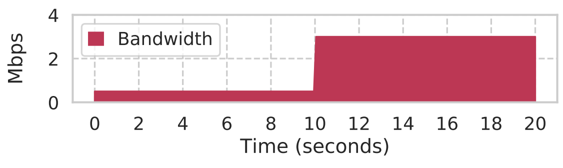
We next consider videoconferencing applications like Skype and Hangouts, which use live-streaming techniques that are tailored for real-time viewing. The technique of buffering frames in the face of bandwidth drops described in Section 3.4.1.1 is inadequate for this setting because buffering delays the transmission of frames beyond the real-time deadline required for interactivity. Real-time video streaming solutions like WebRTC [44] significantly underutilize bandwidth [53] to ensure timely delivery of video frames. Salsify [14] explores video encoding techniques that can accurately match the network bandwidth to reduce buffering for low latency playback.

Assuming that the uploading client has accurate instantaneous upload bandwidth estimates, Figure 3.4b (yellow crossed line) depicts the real-time video quality when using streaming techniques optimized for low latency. Between times $0\,\text{s}$ and $10\,\text{s}$, frames are encoded at $0.5\,\text{Mbps}$ to minimize buffering delay. When the bandwidth increases at $10\,\text{s}$, frames are encoded at $3\,\text{Mbps}$. This results in an average SSIM of 0.679 across the entire received video.

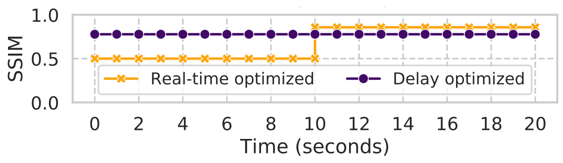
3.4.1.3 Uploading for more than one viewing delay?

While the techniques for live video upload described in Sections 3.4.1.1 and 3.4.1.2 achieve high QoE for a single viewing delay, they are inadequate for applications where the video is viewed at different delays. Tailoring upload for delayed playback (i.e., Section 3.4.1.1) results in high QoE for viewing beyond a certain delay, but renders the video unplayable at smaller time-shifts. On the other hand, while real-time streaming techniques (i.e., Section 3.4.1.2) are suitable for low-latency viewing, this limits the QoE for viewers at larger time-shifts: in the example above, upload optimized for real-time viewers results in an average SSIM of 0.679, whereas uploading for delayed viewing results in an average SSIM of 0.777.

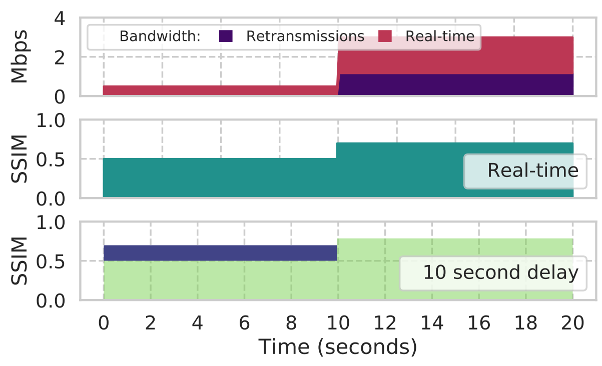
As described in Section 3.3.2, SLVS applications enable time-shifted viewing in which viewers can watch the same video at different delays. Thus, optimizing live video upload for a single viewing delay is inadequate for SLVS applications, necessitating changes in the architecture of the live video upload process to *improve the quality of experience for both real-time and time-shifted viewers*.



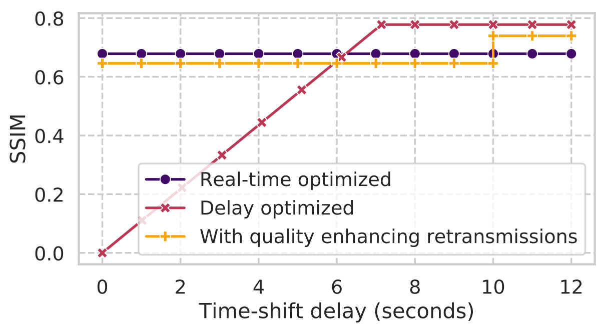
(a) Example bandwidth trace.



(b) SSIM for conventional real-time and delayed streaming techniques.



(c) Bandwidth utilization and improvement in video quality with quality-enhancing retransmissions.



(d) Comparison of average SSIM for different viewing delays for conventional techniques and the proposed technique.

Figure 3.4: Toy example demonstrating the benefits of quality-enhancing retransmissions over conventional techniques.

3.4.1.4 Current SLVS platforms do not cater to time-shifted viewing

In this section, we demonstrate that common SLVS platforms (Facebook Live, YouTube Live, Twitch, and Periscope) today do not explicitly account for time-shifted viewing, thereby presenting significant opportunities for improving the overall QoE. All the platforms we consider support archival of the live stream for viewing after the termination of the live streaming session. Hence, these platforms support at least two distinct “time-shifted viewing modes”: real-time viewing during the live stream, and video-on-demand (VoD) style viewing after the live stream has ended.

Opportunities to improve QoE for time-shifted viewers come into play after the network has recovered from an impairment, where the video was encoded at a lower bitrate to prevent network saturation. To determine whether a platform takes time-shifted QoE into consideration, we initiate a live stream and let it run on an unimpaired network for $40\,\mathrm{seconds}$ and then momentarily throttle the bandwidth to $0\,\mathrm{Mbps}$ (no data can be transmitted) for 3 seconds. This is followed by $30\,\mathrm{seconds}$ of transmission over an unimpaired network. We emulate an impaired network using the `dummynet` [54] network emulator. Note that there is no impairment on the link between the live streaming platform’s upload endpoint and the viewer. While our experiments may also contain variations that occurs in the upload path outside of the impairment we impose, we found the upload path to be fairly stable during all experiments. We test the RTMP protocol using OBS [55] with settings recommended by the platform under test, and the WebRTC protocol using Google Chrome’s WebRTC implementation. We compare the quality of the real-time and the archival versions of the video by comparing the video at the receiver to the prerecorded reference video that was uploaded.

The impairment introduced in this experiment mirrors that described in the toy-example from Section 3.4.1: there is ample opportunity after the network impairment for improving the video quality for the archived version. Yet, in all experiments, we find that the video frames in the real-time and the archived videos are identical. Both videos experience identical frame drops, resulting in pauses in the received video, and the received frames have identical SSIM values.

These experiments suggest that current SLVS platforms do not exploit opportunities to improve the QoE for more than one viewing delay, leaving considerable opportunity for improving QoE for viewers across diverse viewing delays.

3.4.2 Proposed approach

We now describe how a live upload solution can provide high QoE for multiple viewing delays.

3.4.2.1 Naïve approach: re-uploading an entire stream.

A simple strategy to improve the QoE for archived viewing is to store a high quality version of the captured video on the broadcaster’s device and re-upload the video after the stream has ended. However, this approach only improves the archival quality; no improvements are achieved for time-shifted viewers during the live stream. Further, this approach consumes a large amount of storage on the broadcaster’s device and consumes at least twice as much bandwidth as one would use without re-uploading the stream. These downsides are especially of concern for longer SLVS sessions which may last for multiple hours [38, 56], and for video streams initiated from mobile devices, which often have limited, metered bandwidth and insufficient internal storage.

3.4.2.2 Quality-enhancing retransmissions.

As described in Section 3.3.3, mobile networks commonly experience significant bandwidth fluctuations, *both* low and high. Thus, low quality real-time frames transmitted during periods of low bandwidth can be retransmitted *at a higher bitrate* during a later period of high bandwidth *while the live streaming session is ongoing*, thus improving the video quality for delayed, time-shifted viewers. We call these retransmitted frames "quality-enhancing retransmissions" because they are retransmitted for the purpose of improving the quality of a frame compared to the initial version of the frame transmitted for real-time viewing. Unlike typical retransmissions, quality-enhancing retransmissions are not in response to packet loss.

Sending quality-enhancing retransmissions requires reducing the bitrate of real-time frames for allocating sufficient bandwidth for high-quality retransmissions. Due to the concave nature of bitrate-SSIM curves of common video encoding techniques [49], the bitrate of real-time frames can be reduced without causing a significant drop in the video quality. This is evident from the bitrate-SSIM data shown in Figure 3.3. The drop in quality when reducing the frame size of a high quality frame (solid red arrow) is significantly smaller than the corresponding gain in quality when increasing the frame size of a low quality frame (solid green arrow) by the same amount.

Sending quality-enhancing retransmissions comes at the expense of sending redundant bits over the network and introducing additional computation for frames that have already been sent at a lower quality. The use of scalable video coding (SVC)[57] techniques may result in better performance in some cases. We discuss the implications of using an SVC based design in Section 3.5.6.

Example of improvements for time-shifted viewing. Consider the example discussed in Section 3.4.1. Under the approach of using quality-enhancing retransmissions, when the bandwidth is low ($t = 0\,\mathrm{s}$ to $t = 10\,\mathrm{s}$), the video is encoded at $0.5\,\mathrm{Mbps}$. This behavior is identical to the real-time upload approach. When bandwidth is higher ($t = 10\,\mathrm{s}$ to $t = 20\,\mathrm{s}$), older frames can be retransmitted at a higher quality as quality-enhancing retransmissions along with the real-time frames. The bitrate of the real-time frames needs to be lower than the available bandwidth ($3\,\mathrm{Mbps}$) to accommodate the extra network traffic from the quality-enhancing retransmissions.

The choice of bandwidth allocation between real-time frames and quality-enhancing retransmissions can be driven by high-level goals such as maximizing the viewer-count weighted SSIM across the time-shifted delays. Choosing the optimal tradeoff between the drop in real-time quality and the improvement in delayed video quality can be envisioned as a quality (SSIM) maximization problem across the time-shifted delays. Assuming equal weights for real-time quality and the time-shifted viewing quality at a viewing delay of 10 seconds, the bandwidth allocation which maximizes the average quality across the two delays is $1.84\,\mathrm{Mbps}$ for real-time frames and $1.16\,\mathrm{Mbps}$ for retransmitted frames. This bandwidth split is depicted in the top plot (bandwidth) in Figure 3.4c. This results in an average SSIM of 0.643 and 0.742 for the real-time and the delayed viewers, respectively. This is demonstrated in the bottom two plots in Figure 3.4c. The dark green region in the middle plot (labeled "Real-Time") represents the real-time video quality. The dark blue region in the bottom plot (labeled "10 second delay") shows the improvement in video quality for delayed viewing over the real-time video quality (shown in light green for reference). Figure 3.4d shows the average quality for time-shifted viewing between $t = 0\,\mathrm{s}$ (i.e., real-time) and $t = 12\,\mathrm{s}$ for video uploaded by real-time optimized, delay-optimized, and time-shift-aware upload strategies. Compared to real-time optimized streaming, using Vantage improves the SSIM for delayed viewing by 9.4%, while causing a drop of only 5.2% in the SSIM of the real-time stream. Note that the drop in SSIM only occurs for the high-quality frames, which still results in good real-time quality as opposed to delay-optimized streaming, where real-time viewing is not feasible. We note here that although we plot the average SSIM in this case to demonstrate the benefits of time-shift optimized streaming, for our evaluation, we use a unified metric (described in Section 3.6.1) that combines the SSIM of the video frames and the stalling events into a single metric.

Thus, by making use of quality-enhancing retransmissions and allocating bandwidth between diverse time-shifted viewing delays, a SLVS upload solution can deliver high QoE to viewers across a spectrum of viewing delays.

3.4.2.3 Key challenges

While the idea of sending quality-enhancing retransmissions by exploiting periods of high bandwidth during the live streaming session is simple and promising, there are several critical challenges in designing a live streaming upload solution based on this idea.

Optimal, efficient bandwidth allocation in real time. As described in Section 3.4.2, allocating bandwidth between real-time frames and quality-enhancing retransmissions can be formulated and solved as an optimization problem. However, finding an optimal solution to this problem may take a non-trivial amount of time, making it challenging to design an optimization-based system involving real-time streaming.

Bitrate-SSIM curve estimation. The bitrate-SSIM curves of frames inform the allocation of bandwidth between real-time frames and quality-enhancing retransmissions. However, the bitrate-SSIM curve of a video frame is not available before the frame is encoded, and thus must be estimated for performing bandwidth allocation. Estimating bitrate-SSIM curves is challenging as they depend on a variety of properties of each frame and the preceding frames.

Mitigating error in bandwidth estimation. Allocating bandwidth between real-time frames and quality-enhancing retransmissions requires having an estimate of the future bandwidth. Accurately estimating future bandwidth is challenging in its own right, and thus a system allocating bandwidth between real-time frames and quality-enhancing retransmissions must be robust in it's abilities to adapt to inaccuracies in bandwidth estimation.

In section 3.7.7, we show that addressing these challenges are critical for improving the QoE across all viewers for time-shifted viewing of live video streams, and in section 3.7.6, we show that Vantage is able to robustly handle bandwidth mis-estimation due to our design choices made in section 3.5.3.

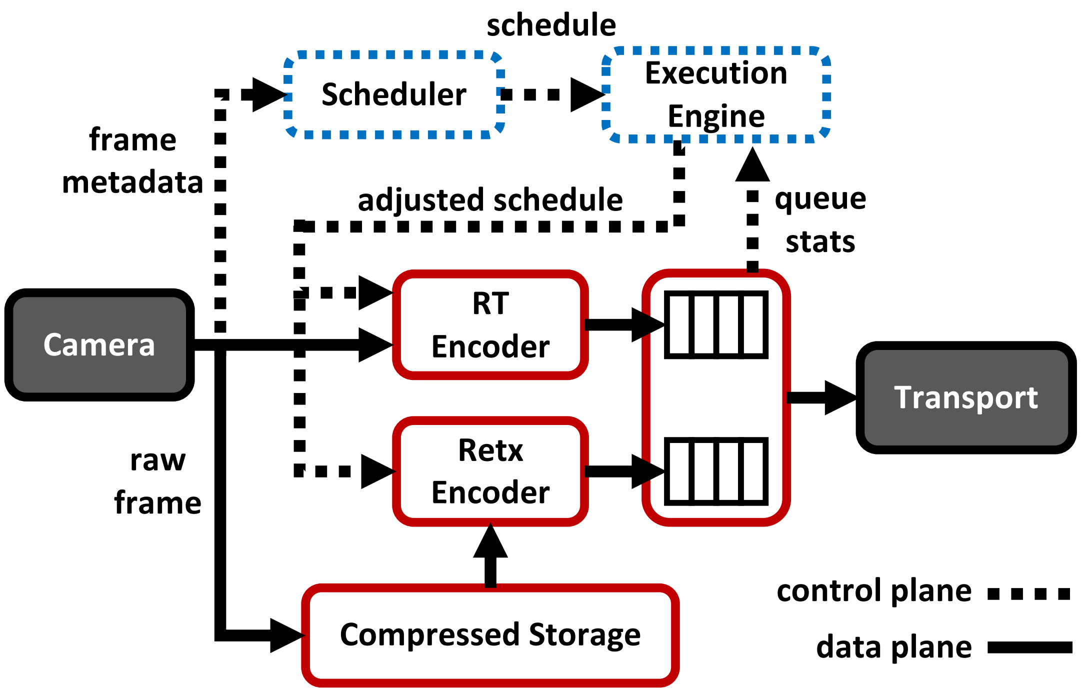


Figure 3.5: Vantage's architecture. Solid lines indicate data-plane components. Dotted lines indicate control-plane components. Components that are unchanged by Vantage are shown in a darker shade.

3.5 Design of Vantage

In this section, we describe the design of Vantage and how Vantage overcomes the challenges outlined in Section 3.4.2.3 in order to deliver high QoE to diverse time-shifted viewers.

3.5.1 Overview

We first describe the architectures of current live video upload systems, and then describe Vantage's high-level operation.

3.5.1.1 Current live video upload systems

In current live video upload systems, the uploading client first captures raw video frames from the camera on a mobile device. Frames are then compressed by an encoder and transmitted to the upload endpoint. The system's network transport mechanism estimates the available upload bandwidth, which informs the level of compression used for encoding the video frames.

3.5.1.2 Architecture of Vantage

Figure 3.5 depicts Vantage's high-level architecture. Vantage modifies the upload architecture described in Section 3.5.1.1 to enable support for quality-enhancing retransmissions. Raw frames from the camera are encoded and enqueued for real-time transmission. Vantage simultaneously compresses these frames at a high quality and places them in memory for potential quality-enhancing retransmissions in the future. Vantage's scheduler generates a bandwidth allocation schedule for the new frames captured by the camera as well as for quality-enhancing retransmissions. The execution engine coordinates the encoding of scheduled frames, and adjusts for inaccuracies in the allocation determined by the scheduler (discussed in detail below in Section 3.5.3). Frames that have been scheduled for transmission are enqueued for transmission by a generic transport protocol that is unmodified by Vantage. We assume that the transport layer provides network bandwidth estimates and drains packets from Vantage's queues.

In the remainder of this section, we describe each of Vantage's components in detail as well as how Vantage overcomes the challenges presented in Section 3.4.2.3.

3.5.2 Scheduler design

Vantage's scheduler takes as input a set of real-time frames and potential candidate frames for retransmissions, and an estimate of the upload bandwidth in the near future in order to choose (a) which frames to schedule for transmission and (b) the bitrate each scheduled frame should be encoded at.

We formulate schedule generation as a *mixed-integer optimization problem* that generates a transmission schedule that optimizes the quality for the viewers across time-shifted delays. The precise formulation of the optimization problem is described in Section 3.5.2.1.

An important consideration with this approach is that the time taken to solve a mixed-integer problem may be non-trivial and highly variable. To address this, Vantage's scheduler is run every P seconds and generates schedules for the next P seconds. When the scheduler is run at time $T = t$, it receives a snapshot of the state (i.e., candidate frames and bandwidth estimation) at time t, and generates a schedule for the period between $T = t + P$ and $T = t + 2P$. If the optimization takes longer than P seconds, the scheduler is interrupted and the current, potentially sub-optimal solution of the optimization is used as the schedule.

3.5.2.1 Optimization formulation

Next, we discuss the formulation of the optimization problem that is performed by the scheduler every P seconds. Consider a single optimization iteration that starts at time $T = t$. We first list the inputs to the optimizer and then subsequently discuss how these inputs are obtained. The optimizer takes the following information as inputs:

1. An estimate of the number of bytes B that can be transmitted between $T = t + P$ and $T = t + 2P$.
2. A set of future real-time frame ids (F) that will be sent between $T = t + P$ and $T = t + 2P$.
3. A set of past frame IDs (G) that are to be considered for retransmission. A retransmission chosen in this iteration would happen at some time between $T = t + P$ and $T = t + 2P$. The difference between $T = t + 2P$ and the time at which a frame g was captured is the delay of the frame, which we denote as d_g.

4. The quality of past frames that have been received by the upload endpoint. For each frame $g \in G$, R_g denotes the SSIM of the version of frame g currently available at the upload endpoint. We do not schedule queued or in-flight frames for retransmission.
5. The bitrate-SSIM curve for each frame $g \in G$. For each frame $g \in G$, $\mathrm{Q}_g : size \rightarrow ssim$ represents the mapping from encoded frame size (in bytes) to the SSIM.
6. The predicted bitrate-SSIM curves for each of the real-time frames that will be sent between $T = t + P$ and $T = t + 2P$. We denote this by $\mathrm{Q}_f : size \rightarrow ssim$.
7. The distribution of the viewing delays of the current set of viewers. For each viewing delay d, $\mathrm{N}(d)$ represents the count of viewers watching the live stream at a viewing delay of d seconds.
8. We also define a set of weights $w_g \ \forall g \in G$ and a weight w_0 for the real-time frames. We discuss how these weights are computed from the delay distribution N in the subsequent paragraphs.

The scheduler returns the target sizes $s_f \ \forall f \in F$ for the real-time frames and a set of frames $G' \subset G$ and the corresponding target size $s_g \ \forall g \in G'$ for the quality-enhancing retransmissions. We formulate the optimization problem as a maximization of the weighted viewing quality subject to bandwidth constraints.

The role of the bandwidth constraint is to ensure that the total amount of data scheduled for transmission (including both the real-time frames and past frames) does not exceed the estimated bandwidth. Thus, the bandwidth constraint is :

$$\sum_{\forall f \in F} s_f + \sum_{\forall g \in G} s_g \leq B \tag{3.1}$$

The objective function includes contributions from the real-time frames and the past frames. For a real-time frame $f \in F$, the contribution to the objective function is

$$w_0 \cdot \mathrm{Q}_f(s_f) \tag{3.2}$$

For a past frame $g \in G$, the contribution to the objective function is

$$w_g \cdot \max(\mathrm{Q}_g(s_g), R_g) \tag{3.3}$$

Note that the weights w_g are different for each frame $g \in G$.

The net objective sums up the contribution from each of the real-time frames and the past frames:

$$obj = \sum_{\forall f \in F} w_0 \cdot \mathrm{Q}_f(s_f) + \sum_{\forall g \in G} w_g \cdot \max(\mathrm{Q}_g(s_g), R_g) \tag{3.4}$$

Since the real-time frames serve as a base for delayed playback as well, the transmission of a real-time frame benefits all delays. Similarly, a quality-enhancing retransmissions at a delay d is useful for all viewing delays that are greater than d. Thus, we set the weights for the real-time frames (w_0) and the past frames (w_g) in the objective function as follows.

$$w_0 = \sum_{d} \mathrm{N}(d), w_g = \sum_{d > d_g} \mathrm{N}(d) \tag{3.5}$$

The functions Q_i that maps size to SSIM for a frame i is typically a non-linear curve (e.g., Figure 3.3) and can vary significantly across frames. We approximate these curves as piece-wise linear functions in the formulation of the mixed-integer program. Since the number of frames that can be encoded in P seconds is limited, we additionally limit the number of retransmissions to $|F|$. This ensures that the computational requirements of encoding the quality-enhancing retransmissions does not exceed that of the real-time frames.

3.5.2.2 Bitrate-SSIM curve estimation

Recall from Section 3.5.2.1 that the optimization problem uses bitrate-SSIM curves of all frames that are candidates for scheduling (i.e., Q_g for retransmissions and Q_f for real-time frames). This is required so that the optimization can make informed choices when reducing the real-time quality to improve the quality of past frames.

Vantage uses a regression heuristic to estimate these curves from previous encoding data. We use a function of the form $\mathrm{Q}(s) = 1 - \frac{1}{a \cdot s + b}$ since it captures the concave non-decreasing behavior (for $a > 0, s > \frac{-b}{a}$) typical of bitrate-SSIM curves (e.g., Figure 3.3). Parameters a and b are computed separately for each frame based on its observed size and SSIM when the frame has already been previously encoded (i.e., for real-time or for quality-enhancing retransmissions). These parameters are updated each time a frame is re-encoded for retransmission. However, bitrate-SSIM information is not available for future real-time frames because they have not yet been captured. Hence, for the future frames, we use the EWMA values of the past parameters for computing the parameters of $\mathrm{Q}_f \;\; \forall f \in F$ because frames that are temporally local have similar content, and thus similar bitrate-SSIM curves.

3.5.2.3 Optimizer performance

A preliminary evaluation of Vantage with a scheduling period $P = 2\,\mathrm{s}$ indicated that the mixed-integer solver often fails in finding an optimal solution within 2 seconds when $|G|$ is large. One alternative is to increase the scheduler period P, but this results in worse performance due to the scheduler receiving stale bandwidth estimates. This is further discussed in Section 3.7.5. Instead, Vantage filters G using a heuristic and only generates the variables in the mixed-integer program for the 200 frames with the worst SSIM. Furthermore, we do not restrict the frame sizes to be integers, and instead use an integer approximation for the continuous solution. We find that using $P = 2\,\mathrm{seconds}$ along with these approximations leads to the optimizer generating high-quality schedules: 98.5% of optimization windows in our evaluation result in a schedule that is within 1% of the optimal solution.

3.5.3 Mitigating bandwidth estimation error

As described in Section 3.5.2, Vantage's scheduler generates an encoding and transmission schedule for P seconds in the future based on an estimate of the future network bandwidth.

Since the optimizer uses a bandwidth estimate measured P seconds before the scheduled frames are transmitted, the true available bandwidth may differ at the time when the scheduled frames will be transmitted. Left uncorrected, the use of a schedule generated from a mispredicted bandwidth estimate will lead to sub-optimal use of the network.

To mitigate the effects of bandwidth misestimation, Vantage's execution engine makes adjustments to the generated schedule prior to transmitting the frames. When the bandwidth estimate used to generate the schedule under-estimated the amount of bandwidth available at transmission time, Vantage's execution engine keeps the network saturated by increasing the bitrate of the real-time video compared to the optimizer's schedule, but only if the retransmissions scheduled in that iteration have been completed.

On the other hand, when the bandwidth estimate used to generate the schedule over-estimated the amount of bandwidth available at transmission time, the execution engine prioritizes transmission of real-time frames: frames scheduled for retransmission at that time are discarded and real-time frames are encoded at a bitrate lower than that specified by the scheduler so as to avoid over-saturating the network. Prioritizing real-time transmission in the event of bandwidth over-estimation ensures high QoE for all time-shifted viewing delays, as real-time frames would be available for viewing at all delays, whereas retransmitted frames only benefit viewers watching with a time-shifted delay.

3.5.4 Encoding retransmissions

Frames that have been scheduled for retransmission at a particular time may not be temporally close to the real-time frames scheduled at the same point in time. This presents a challenge for encoding Vantage's output stream because video encoding algorithms rely heavily on the similarity between successive frames to achieve good compression ratios. Using the same encoder for transmitting both the real-time video and the retransmissions would result in poor compression, as the content in the retransmitted frames may differ significantly from real-time frames.

To address this challenge, Vantage uses two separate encoders for compressing real-time and retransmitted frames. Real-time frames are encoded in the order in which they were captured. Quality-enhancing retransmissions are encoded by a separate encoder based on the schedule determined by the optimizer. Though retransmissions could be temporally far from one another, we note that network impairment events commonly affect groups of neighboring frames. Thus, if a particular frame is a good candidate for retransmission, it is more likely that its neighboring frames are also good candidates for retransmission. We, therefore, add an additional regularization objective to the optimization formulation to favor scheduling consecutive sequences of frames among the retransmission candidates for quality-enhancing retransmissions.

3.5.5 Reducing memory overhead

Vantage keeps previously transmitted frames in memory so that they can be re-encoded as quality-enhancing retransmissions at a later time. Naïvely storing raw video frames in memory is impractical for uploads initiated from a mobile device; the size of a raw frame can be as large as 1.5 MB, thus requiring more than a gigabyte of memory for 30 seconds of video.

To address the high cost of storing raw video frames, Vantage compresses raw frames as lossless I-frames using a tertiary encoder prior to storing in memory. This allows Vantage to maintain a low memory footprint, but incurs additional computational cost. We believe that this is an appropriate trade-off as hardware accelerated encoding and decoding solutions are commonly available today, though we note that this design choice is not required by Vantage's framework.

3.5.6 Discussion

In this section, we briefly discuss required changes to the upload endpoint to support Vantage and how Vantage would differ with the availability of an SVC codec.

Upload endpoint modifications. Recall from Section 3.3.1 that an SLVS upload stream is terminated at an upload endpoint, which decodes the stream and re-encodes it into small video segments for efficient delivery to the viewers over content delivery networks (CDNs) [37].

While the changes proposed in Vantage allow backward compatibility with real-time streaming systems, the upload endpoint must be able to handle Vantage's quality-enhancing retransmissions. This requires the upload endpoint to re-transcode past video segments whenever quality-enhancing retransmissions for that segment are received, and to disseminate these higher quality video segments to the CDN. As described in Section 3.5.4, Vantage's scheduler penalizes retransmissions that are spread apart, which helps limit the rate at which past video segments need to be updated. Requiring only these minor changes to the upload endpoint makes Vantage well-suited for current SLVS architectures.

Scalable video coding. Vantage's approach of sending quality-enhancing retransmissions bears similarity to the enhancement layers used in scalable video coding (SVC) techniques. Indeed, Vantage's design could be simplified by using an SVC codec, since the video would only need to be encoded once, and storage of high-quality frames would not be necessary. Despite these benefits, we have chosen not to design Vantage specifically for SVC codecs because

1. SVC codecs are not widely adopted, limiting hardware-accelerated encoding support, and
2. while simple SVC schemes with coarse grained scalability do not have significant overhead, fine grained SVC schemes have poor compression efficiency and are significantly more compute intensive compared to non-layered codecs.

Even in the absence of the aforementioned downsides of SVC, the use of SVC alone cannot overcome the challenges involved with optimizing the video upload for multiple time-shifted delays. An SVC based live video upload mechanism would still need to make bandwidth allocation decisions between the real-time base video stream and the enhancement streams, and also choose which frames to retransmit. While the ability to encode the video only once and not having to store high quality frames is an advantage of using SVC, the use of non-layered codecs is better in some situations. When retransmitting a sequence of contiguous frames, encoding a P-frame with a high quality past frame as the reference is often more efficient than encoding the frame with a low quality version of itself as the reference frame.

While SVC provides some clear benefits, we believe compatibility with standardized codecs and hardware is more important for adoption in the real world today. We would only need to make minor tweaks to the optimization formulation used in Vantage's scheduler for generating optimized schedules for SVC codecs.

Overhead of two encoders. Vantage's approach of using two separate encoders to compress real-time frames and quality-enhancing retransmissions is computationally expensive. We believe this overhead is well-justified: Vantage gains significant improvements in the QoE across multiple viewing delays for SLVS applications, and the trend of increasing hardware acceleration support further justifies this tradeoff.

3.5.7 Implementation details

We have implemented Vantage in C++, with the scheduler using the Gurobi [58] library to solve the optimization problem. To reduce computational requirements, we limit Gurobi to run on a single core and the execution engine to run on a single thread. Vantage uses the VP8 encoder from Salsify [14] because it provides a convenient API for controlling the size of each frame. For performance reasons, Vantage compresses the high-quality frames and encodes real-time frames and the quality-enhancing retransmissions in parallel.

3.6 Evaluation Methodology

We evaluate Vantage and compare its performance to conventional live video upload techniques for SLVS applications.

3.6.1 Metrics

Vantage is designed to improve the quality of video playback across the various time-shifted viewing delays by replacing low quality frames with high-quality versions and filling in the gaps caused due to skipped frames. While Vantage's scheduler is designed to optimize the SSIM [59] metric, Vantage can support other frame level reference metrics (like PSNR, etc.). We use the SSIM metric when measuring the quality of a single frame and use the SQI-SSIM [60] metric to compute an overall video quality score from the SSIM values of the individual frames. Most objective video quality metrics do not consider the effect of stalling when calculating video quality [61]. SQI-SSIM is a unified metric that takes into account the full reference quality of each frame and also the duration and frequency of video stalls. SQI-SSIM uses an exponential decay function instead of zeros to fill in the SSIM of missing frames. Thus, shorter stalls have a smaller effect on the overall video quality. When video playback resumes after a stall, the SSIM of the subsequent frames are penalized according to an exponential decay function, thus accounting for the frequency of stalls. We note that Vantage can support other video quality assessment metrics (such as PSNR).

3.6.2 Baselines

While there is significant prior work on optimizing video streaming for the individual cases of live streaming and VOD-style video streaming, we are not aware of any prior research on optimizing video quality simultaneously for real-time streaming and time-shifted viewing of the streams. Vantage is designed to work with existing congestion control and video coding systems and enhance the performance of these systems for scenarios involving time-shifted viewing of live streams, and is not meant to be a standalone end-to-end solution for SLVS video upload.

We compare Vantage to the best case performance for low latency streaming and VOD-style streaming using an idealized model for bandwidth estimation and congestion control:

Low latency streaming (Base-RT). Existing low latency optimized streaming systems like WebRTC and Skype maintain low latency by conservatively utilizing the network bandwidth to prevent network saturation. Recent approaches like Salsify [14] utilize the network better by matching the instantaneous network estimate through tight coupling of the video encoder and the transport protocol. Base-RT models these systems by encoding individual video frames at a bitrate that closely follows the real-time network estimate. This results in optimal video quality performance for low latency streaming.

Buffered streaming (Base-Delay). The use of larger buffers at the sender enables a streaming application to encode video at the average network bandwidth. This is similar to conventional ABR based video streaming solutions like HLS [10] and MPEG-DASH [9], which split the video into small segments where each segment is encoded at a specific bitrate. To model the characteristics of streaming techniques that use buffers and slower rate adaptation, we use a window of $30\,\mathrm{seconds}$ to compute the average bandwidth and encode the video at this bitrate. This results in optimal video quality for cases where a delay of $30\,\mathrm{seconds}$ is acceptable.

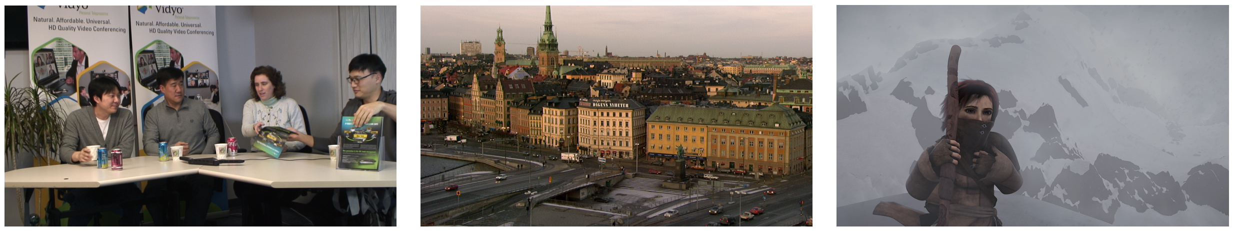
		Talking Heads			City Panning			Animation		
Trace	Delay	Base-RT	Base-Delay	Vantage	Base-RT	Base-Delay	Vantage	Base-RT	Base-Delay	Vantage
Verizon LTE	Real-time	0.8885	0.5854	0.8750	0.8003	0.6479	0.7792	0.9279	0.7504	0.9225
	30s delay	0.8896	0.9552	0.9438	0.8012	0.8472	0.8306	0.9290	0.9818	0.9834
		(0.8890)	(0.7703)	**(0.9094)**	(0.8008)	(0.7475)	**(0.8049)**	(0.9284)	(0.8661)	**(0.9529)**
AT&T LTE	Real-time	0.5638	0.2236	0.5538	0.5224	0.4008	0.4880	0.6511	0.4370	0.6648
	30s delay	0.5705	0.9098	0.8155	0.5285	0.7198	0.6760	0.6576	0.9600	0.9327
		(0.5672)	(0.5667)	**(0.6846)**	(0.5254)	(0.5603)	**(0.5820)**	(0.6543)	(0.6985)	**(0.7987)**
TMobile UMTS	Real-time	0.4957	0.1942	0.4604	0.3371	0.0965	0.3199	0.5055	0.1390	0.4833
	30s delay	0.5054	0.6774	0.5840	0.3451	0.4834	0.4011	0.5169	0.7322	0.6143
		(0.5005)	(0.4358)	**(0.5222)**	(0.3411)	(0.2899)	**(0.3605)**	(0.5112)	(0.4356)	**(0.5488)**

Table 3.1: SQI-SSIM achieved by the baselines and Vantage for each combination of the videos and the network traces. In each case, the average SQI-SSIM across delays (indicated within parentheses) is the highest for Vantage (bolded).

3.6.3 Evaluation setup

We evaluate Vantage with a combination of videos and network traces with different characteristics. Our experiments were run on a machine with Intel(R) Xeon(R) processors, limiting the Gurobi [58] solver to a single core for emulating computational limits in mobile environments. Unless otherwise specified, we evaluate Vantage for a uniform time-shifted viewing delay distribution for the optimization described in Section 3.5.2.1 and use a $2\,\mathrm{second}$ period for the optimizer. This choice is further discussed in Section 3.7.5. We evaluate the effects of different distributions of the time-shifted viewing delays in Section 3.7.3. We run the live streams for $150\,\mathrm{seconds}$ and ignore the data for the last 30 seconds, since the measurements for the last 30 seconds may be affected by the early termination of the program. We repeat videos and traces that are shorter than $150\,\mathrm{seconds}$ until the entire simulation is complete.

Videos. We chose three videos spanning three distinct video styles from the Xiph.org Test Media repository [4] for our evaluation.



(a) Talking Heads: Talking head video with a static background.

(b) City Panning: Panning motion with high detail.

(c) Animation: Animated sequence with varying amounts of motion.

Figure 3.6: Screenshots from the videos used in evaluation. Videos are drawn from the Xiph.org Test Media repository [4].

Figure 3.6 shows screenshots of the three different videos considered for evaluation. “Talking Heads” contains four people talking in front of a static background. This style of video is the most common among SLVS streams [62] and is typically easier to encode. “City Panning” pans across the city of Stockholm. This video is much harder to encode due to a higher amount of moving content and very fine details. “Animation” is an animated video sequence with varying degrees of motion over the duration of the video, which makes some segments easy to encode, while other parts are harder to encode.

Bandwidth traces. We chose a diverse set of network traces from the Mahimahi [3] repository: a high bandwidth LTE trace, a highly variable LTE trace, and a low bandwidth UMTS trace. We find these traces to be representative of Vantage’s performance across all traces in the repository.

Transport layer emulation. We use a bandwidth averaging window of $100\,\mathrm{ms}$ for Base-RT and the real-time stream in Vantage. We use the average bandwidth in the past 1 second as the bandwidth estimate to Vantage’s scheduler. For Base-Delay, we use the average bandwidth of the previous 30 seconds. We run Vantage and the receiver on the same machine and emulate packet transmissions according to the provided bandwidth trace.

Many live-streaming systems use FEC [63] or packet-level retransmission for dealing with network losses. These techniques can be incorporated into the network model by reducing the bandwidth estimates provided to Vantage and using the excess bandwidth for loss recovery mechanisms (e.g., FEC). Since we evaluate baselines using the same model, this provides a fair comparison between Vantage and existing techniques for live video upload.

Encoder performance. Salsify’s encoder is a software-based VP8 encoder written in C++. Software encoders are much slower than hardware based encoders. We observed that even with parallel encoding of the frames, the encoder was not able to achieve a rate of $30\,\mathrm{FPS}$ while encoding 1280×720 (HD) videos. Hence, we run Vantage with time dilation to allow the encoder to run at $30\,\mathrm{FPS}$ in virtual time, but limit the optimization to P seconds of *wall clock time*. This allows us to evaluate Vantage in a manner that is agnostic to the encoding speed of the specific encoder we chose.

Ethics. This work does not raise any ethical issues. We use test video sequences [4] and anonymized bandwidth traces [3] that are publicly available.

3.7 Results

The highlights of our evaluation are as follows:

1. Across a variety of upload bandwidth traces and videos, Vantage improves the SQI-SSIM for time-shifted viewing over Base-RT by 19.9% on average (Section 3.7.1).
2. Vantage *simultaneously* achieves high real-time video quality (within 3.3% of the quality achieved by real-time optimized streaming techniques on average) and high quality for delayed viewing (within 7.7% of the optimal quality achievable for delayed viewing on average), demonstrating the effectiveness of using Vantage for applications that involve time-shifted viewing such as SLVS applications (Section 3.7.1).
3. Vantage is able to adapt and achieve high QoE for skews in the distribution of viewing delays (Section 3.7.3).
4. Vantage can also improve the QoE across different viewing delays for videos with highly dynamic (i.e., harder to encode) content, even when there are no bandwidth variations (Section 3.7.4).
5. Vantage is robust to bandwidth misestimation (Section 3.7.6).

3.7.1 Overall improvements

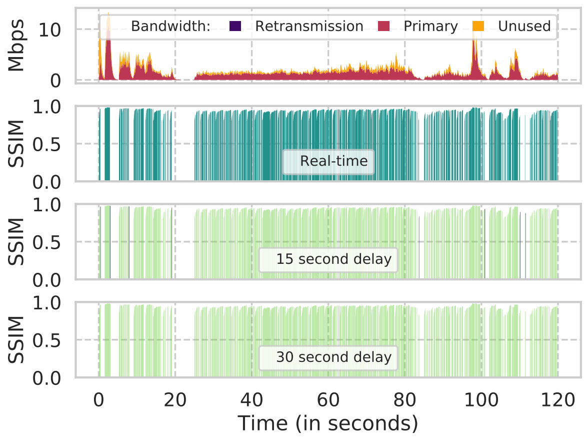
Table 3.1 contains the SQI-SSIM of the received video at both real-time and a delay of 30 seconds for all the traces and videos. The average SQI-SSIM over the two delays (shown in brackets) captures how a particular upload technique caters to both viewing delays.

We observe that in all cases, Vantage achieves significantly better quality than Base-RT (up to 42.9%) for delayed viewing. The gains are biggest for the AT&T-LTE network trace; this can be attributed to the significant variations in the bandwidth. Even for the TMobile-UMTS network trace, which has very low bandwidth and significant periods of zero bandwidth, we observe modest improvements in the delayed viewing quality over Base-RT. This demonstrates that Vantage can *simultaneously* deliver high QoE for both real-time and time-shifted viewing. We also find that, across all traces and videos, Vantage (bolded entries) significantly outperforms both Base-RT and Base-Delay in average SQI-SSIM across the viewing delays.

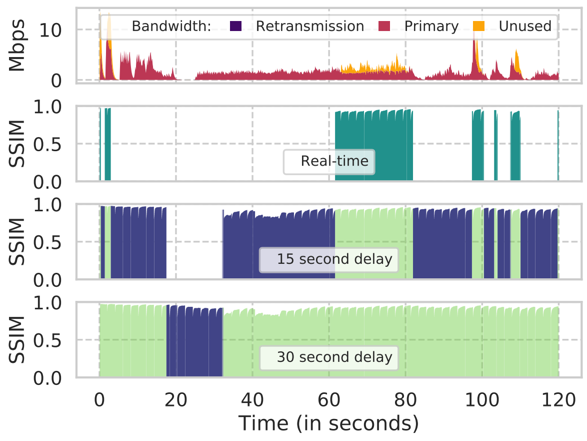
As discussed in Section 3.4.2, the cost paid by Vantage is slightly worse quality for real-time-only and delayed-only viewing as compared to upload transmission techniques optimized individually for either of these settings. Across all traces and videos, we find that the SQI-SSIM of real-time video resulting from Vantage is no more than 7.1% worse than that resulting from Base-RT, with the improvement in the SQI-SSIM for delayed viewing being higher than the drop in the real-time SQI-SSIM in all cases.

We note that while Base-RT and Base-Delay are specialized for real-time and delayed viewing, they are *not necessarily* optimal for these targets. Vantage can occasionally outperform the baselines for these targets (as is seen in Table 3.1) for the following reasons: (1) Base-RT occasionally sends real-time frames that are larger than the available bandwidth, which causes frames to be dropped. In contrast, Vantage encodes real-time frames more conservatively because it also performs retransmissions. (2) Base-Delay can build large buffers of frames that take longer than $30\,\mathrm{seconds}$ to drain, while Vantage only schedules retransmissions that are expected to go through in P seconds.

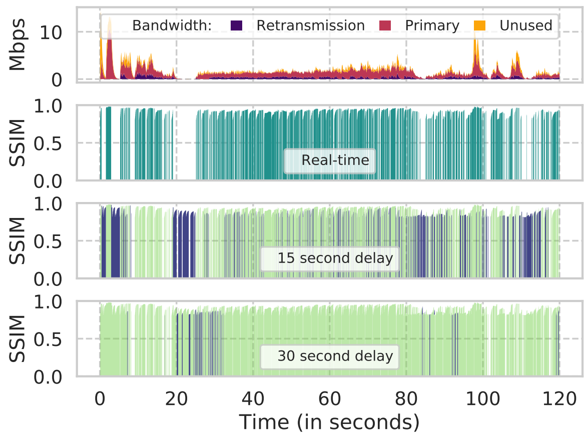
3.7.2 Inspecting Vantage's improvements



(a) Base-RT



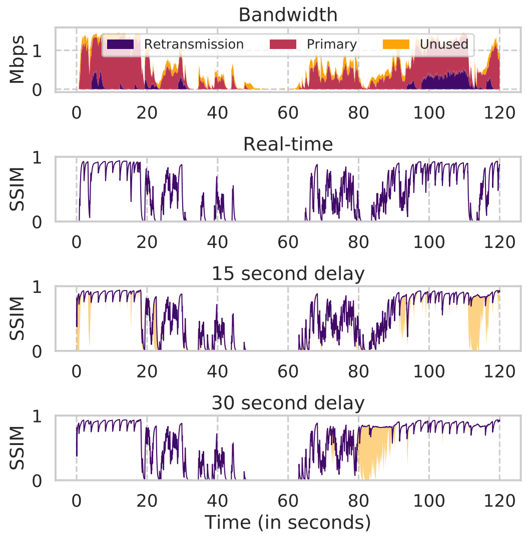
(b) Base-Delay



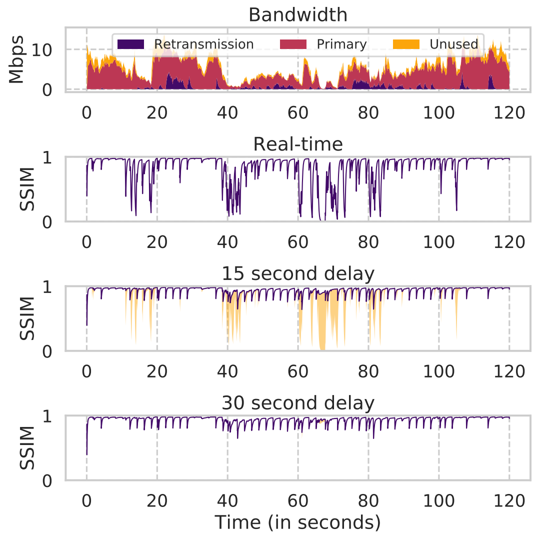
(c) Vantage

Figure 3.7: SSIM of frames at various viewing delays for the Talking Heads video and the AT&T-LTE trace. The top row shows the utilization of the upload bandwidth. The bottom three rows show the quality of the received video at real-time, 15 seconds, and 30 seconds of delay. The dark green region represents the instantaneous quality for the real-time timeseries. For the 15-second delay and the 30-second delay timeseries, the dark blue region represents the improvement in quality compared to the real-time quality and the 15-second delay quality respectively. The light green shaded regions in the 15-second and 30-second timeseries represent the baseline video quality for the previous delay bucket (real-time and 15-second respectively).

In this section, we analyze Vantage's improvements over Base-RT and Base-Delay in detail for a single video and bandwidth trace combination (AT&T-LTE trace and Talking Heads video).



(a) Vantage's performance on Verizon-LTE



(b) Vantage's performance on TMobile-UMTS

Figure 3.8: Timeseries examples showing Vantage's improvement on the two other traces. Here, the yellow shaded regions in the 15-second and 30-second plots show the improvement over real-time quality and the quality for 15-seconds of viewing delay respectively.

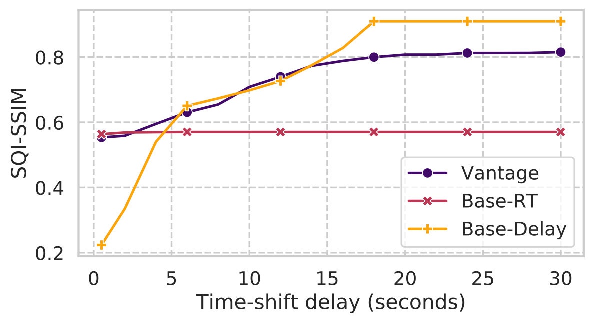
Figure 3.7 compares the baseline approaches to Vantage by plotting the raw SSIM of each frame of the video for various viewing delays. Frames not received in time for a particular viewing delay are shown as zero.

Base-RT (Figure 3.7a) achieves reasonable quality for real-time viewing, skipping frames when the network bandwidth is zero, but does not improve quality for higher viewing delays. Whether a viewer views the stream in real-time or at a delay of 30 seconds, they would both experience a 5-second pause in the video starting at approximately $t \approx 20\,\mathrm{s}$.

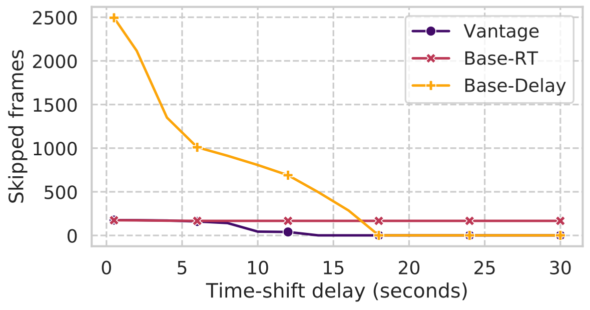
Base-Delay (Figure 3.7b) on the other hand achieves smooth, high-quality video playback for viewing delays of more than 30 seconds, but is unplayable at smaller time-shifts since a majority of the video does not get delivered in time for real-time playback. Even at a delay of $15\,\mathrm{seconds}$, there is a large pause in the video playback between times $t \approx 18\,\mathrm{s}$ and $t \approx 33\,\mathrm{s}$.

Vantage (Figure 3.7c) has no noticeable drops in the real-time video quality as compared to Base-RT. The added benefit of using Vantage can be seen for higher viewing delays where the video quality is dramatically improved. One example is the $5\,\mathrm{s}$ period starting at $t \approx 20\,\mathrm{s}$, where the network bandwidth is zero. We observe in Figure 3.7c that for viewing delay of $30\,\mathrm{s}$ and higher, Vantage repairs this entire segment using quality-enhancing retransmissions, thus resulting in significantly improved video quality over Base-RT. Vantage achieves this by reducing the amount of bandwidth used for real-time frames slightly, and using the excess bandwidth for sending the high-quality retransmissions, as shown in the bandwidth usage graph in the top subplot in Figure 3.7c. With Vantage, the delayed viewing quality comes very close to that of Base-Delay, while *simultaneously* achieving real-time playback quality that is comparable to Base-RT.

Improvements at fine-grained time shifts.



(a) SQI-SSIM achieved by Vantage and the two baselines for various time-shifted delays.



(b) Frames skipped in groups of 10 or more as a function of the viewing delay.

Figure 3.9: Fine-grained time-shift results for the Talking Heads video and AT&T-LTE trace.

Figure 3.9a plots the SQI-SSIM of the video for each viewing delay. For real-time viewing, Base-RT outperforms both Base-Delay and Vantage. At viewing delays beyond 15 seconds, Base-Delay outperforms both Base-RT and Vantage. While the performance of Base-RT and Base-Delay is unsatisfactory for the viewing delays that they are not optimized for, Vantage provides a smooth increase in quality as time-shift delay increases. Vantage's performance is competitive simultaneously at the delays for whith the two baselines are separately optimized for.

Figure 3.9b shows the number of frames of the received video that are skipped in groups of 10 or more frames for different delays. This quantifies the smoothness of the resulting video. Both Base-RT and Base-Delay suffer from a large number of skipped frames for time-shifts that they are not optimized for. In contrast, Vantage has nearly the same number of skipped frames as the baselines at their respective optimal delays and significantly reduces the number of skipped frames for intermediate delays.

3.7.3 Adapting to viewer-delay distributions

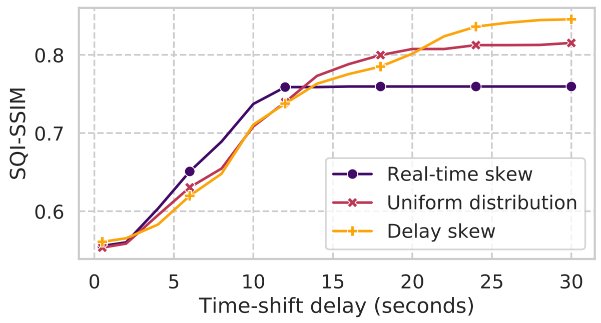


Figure 3.10: Performance of Vantage for various viewing delay distributions.

An important aspect of Vantage is the ability to optimize the video upload process for different distributions of the viewing delays. We evaluated the performance of Vantage for three different viewing delay distributions, one skewed towards real-time viewing, one skewed towards delayed viewing and one with uniform weights for low latency and delayed viewing. Figure 3.10 shows the effect of these weights on the quality of video at different viewing delays between 0 and 30 seconds. For low viewing delays between 0 and 10 seconds, Vantage with a real-time skewed delay distribution achieves the highest quality, whereas for highr viewing delays between 22 and 30 seconds, Vantage with a delay skewed distribution achieves the highest quality. Vantage with a uniform delay distribution strikes a balance between the two across all delays, achieving the highest quality for viewing delays between 10 and 22 seconds. This demonstrates that Vantage can not only support multiple time-shifted viewing delays, but also be tuned to cater to the exact distribution of the viewing delays for optimized QoE across the different delays.

3.7.4 Quality improvements for dynamic videos

In addition to improving QoE in the face of bandwidth variations, Vantage can also be used to compensate for lower video quality for harder to encode segments of a video, even when there is no bandwidth variation. Video content with varying compression difficulty is common in video game streaming applications like Twitch, where the video is significantly harder to encode during highly dynamic segments compared to more static segments like in-game menus. To emulate this setting, we run Vantage and Base-RT for the Animation video, which is an animated sequence with highly dynamic scene content, with a constant bandwidth of $1.5\,\mathrm{Mbps}$.

Base-RT causes 7 frames to be dropped. These drops not only affect real-time viewing, but are also present during delayed viewing. On the other hand, Vantage drops 8 frames in real-time, but retransmits these later *during the live-stream*, resulting in *no lost frames for delayed playback* and a corresponding increase in the SQI-SSIM from 0.960 (for Base-RT) to 0.963.

3.7.5 Optimizer period.

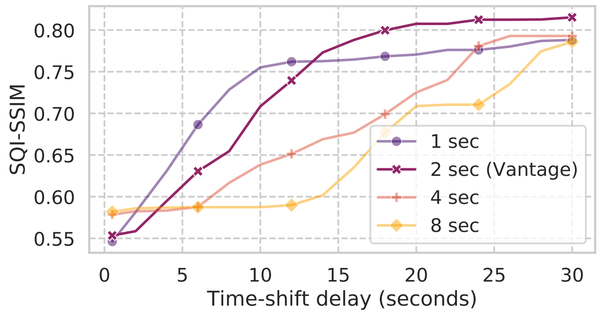
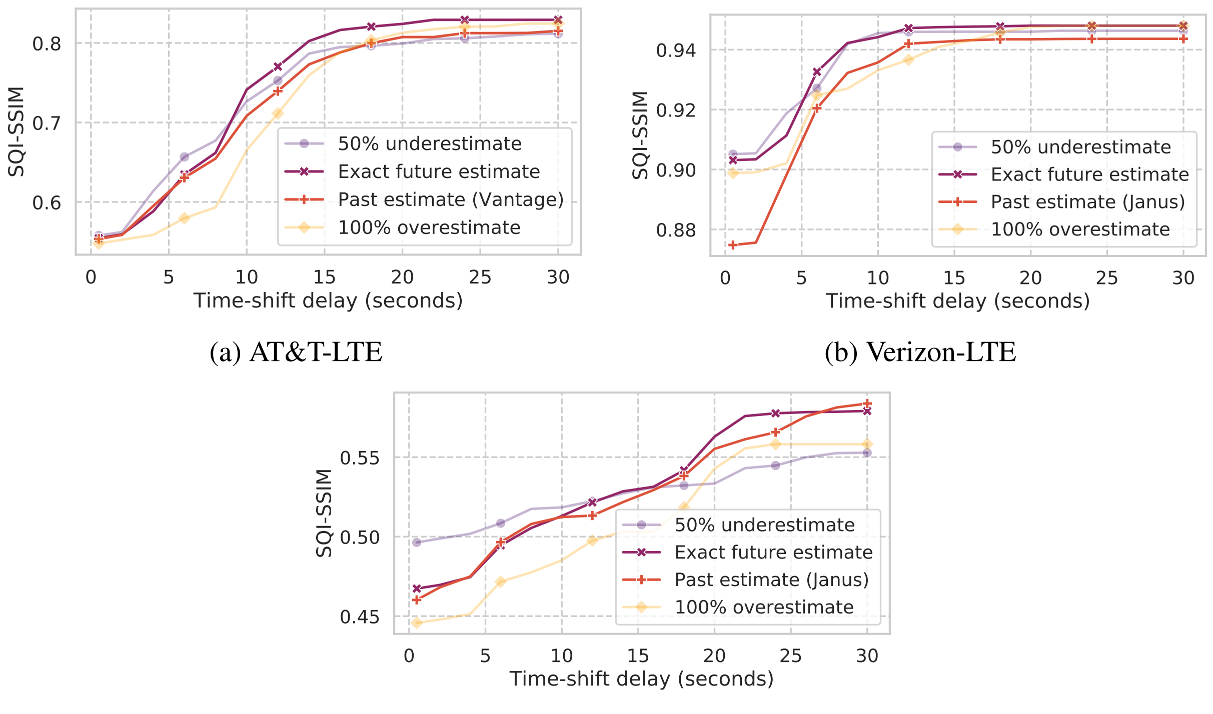


Figure 3.11: Effects of choosing different optimizer periods on the video quality. Vantage uses $P = 2\,\mathrm{sec}$

We ran Vantage with different values of P ranging from $1\,\mathrm{second}$ to $8\,\mathrm{seconds}$, and the results are shown in Figure 3.11. Choosing a large time period allows the optimizer to spread the retransmissions over a longer duration, resulting in better real-time quality, but smaller improvements for delayed viewing since the bandwidth estimates are stale. Smaller time periods result in more accurate bandwidth estimates, but choosing a time period that is too small results in a bigger drop in the real-time quality and smaller improvements for delayed viewing due to retransmissions being squeezed into shorter periods.

3.7.6 Errors in bandwidth estimation.



(a) AT&T-LTE (b) Verizon-LTE

(c) TMobile-UMTS

Figure 3.12: Effects of bandwidth mis-estimation, and performance of Vantage compared to using perfect bandwidth estimates.

Recall from Section 3.5.3 that Vantage's scheduler uses the average bandwidth from the previous $1\,\mathrm{second}$ to schedule frame transmissions for a time period that is $2\,\mathrm{seconds}$ in the future. To evaluate the effect of bandwidth misestimation, we analyze how Vantage performs when the bandwidth estimates provided to Vantage's scheduler are incorrect. To do this, we test Vantage's scheduler with 4 different types of bandwidth estimates - (1) Perfectly accurate bandwidth estimate using an oracle that provides the exact future bandwidth, (2) Bandwidth estimate based on the past, which is practical to implement, (3) a bandwidth estimate that consistently overshoots the oracle-based estimate by 100%, and (4) a bandwidth estimate that consistently undershoots the oracle-basd estimate by 50%.

Figure 3.12 shows the SQI-SSIM for different time-shifted delays when Vantage is faced with varying degrees of bandwidth estimation error for the three network traces using Talking Heads.

Vantage in its normal operation mode (i.e., using the average bandwidth of the past) is listed as "Past estimate (Vantage)."

We see that Vantage achieves only slightly lower SQI-SSIM values across times-shift delays compared to what it would achieve with knowledge of the exact future bandwidth. Across all video-trace combinations, we find that Vantage results in drops in quality of at most 1.7%, 3.2%, and 2.4% for real-time, 15 second, and 30 second viewing delays, as compared to what it would achieve with knowledge of the exact future bandwidth. This suggests that Vantage's approach of using the average bandwidth from the previous $1\,\mathrm{s}$ is satisfactory for generating high-quality bandwidth allocation schedules and Vantage is resilient to errors in bandwidth estimation.

Figure 3.12 further indicates that Vantage is highly resilient to even large bandwidth estimation errors due to effective fallback mechanisms: Vantage achieves high SQI-SSIM even when the bandwidth is severely mis-estimated. The real-time video is largely unaffected by bandwidth misestimation in the scheduler since the execution engine adapts to rapid variations in the real-time bandwidth (described in Section 3.5.3). When the generated retransmissions are too large to be transmitted, the scheduler overwrites the schedule in the next iteration and does not send the retransmissions, and instead allocates the available bandwidth to the real-time video stream. On the other hand, when the generated retransmissions are small, they get transmitted faster and the excess bandwidth is again allocated to real-time frames in order to utilize all of the available bandwidth.

3.7.7 Ablation studies.

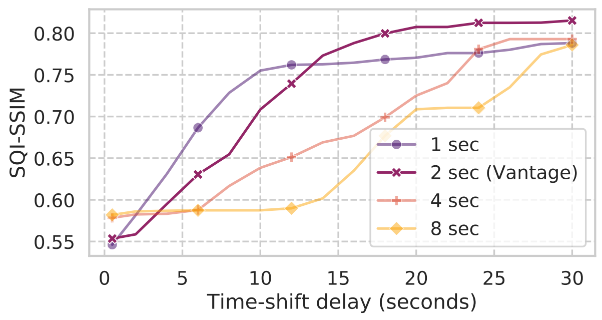


Figure 3.13: Comparison of Vantage with simpler retransmission schemes based on restricted versions of Vantage.

The use of a quality enhancing retransmissions to improve the video quality for higher viewing delays can be implemented in multiple ways and it is important to understand the additional benefits Vantage's design provides over simpler mechanisms such as reserving a fixed amount of bandwidth for enhancing the quality of the video frames affected by network impairments. The effectiveness of using Vantage can be attributed to three core design ideas: (1) optimizing bandwidth allocation across the real-time and retransmitted frames, (2) optimizing the choice of which frames to retransmit based on the population distribution, (3) and predicting the SSIM of the real-time and quality-enhancing frames for the optimization formulation. To understand the benefits of using Vantage over simpler designs, we conducted ablation studies comparing Vantage to restrained versions of Vantage.

Worst frame heuristic only considers the worst 30 frames for scheduling in each iteration, but lets Vantage perform bandwidth allocation based on the bitrate-SSIM curves to optimize the quality across the time-shifted delays. This is similar to a naïve solution that performs optimal bandwidth allocation, but selects frames for quality-enhancing retransmissions every 2 seconds according to a heuristic that picks a few of the worst frames. This approach works well for the two extreme viewing delays (real-time and 30 seconds of delay), but performs poorly for intermediate delays.

Unweighted optimization runs Vantage with equal weights for each frame, giving equal importance to real-time frames and retransmissions. This is similar to a slightly smarter heuristic that prefers contiguous segments for retransmission (since we penalize the selection of isolated frames in the optimization).

No SSIM prediction runs Vantage with a fixed bitrate-SSIM curve. As a comparison point for Vantage, this ablation demonstrates the importance of performing smart bandwidth allocation across individual frames based on the actual video quality instead of a simple heuristic based bandwidth allocation strategy.

The results of running Vantage and the restrained versions on the AT&T LTE trace and the talking heads video are shown in Figure 3.13. We observe that Vantage performs significantly better than the restrained versions, demonstrating the benefits of each design feature Vantage's design.

3.8 Related Work

Previous work related to video streaming including WebRTC [44], RTMP [43], MPEG-DASH [9], HLS [10] and SVC [57] are discussed in earlier sections. We discuss additional related work in this section.

Joint bitrate selection and transport. Salsify [14] couples encoding and transport in order to match available bandwidth for real-time communication. This improves the quality and delay for interactive video but does not consider the variety of delays at which viewers interact with a single video, which Vantage targets.

Downstream path. There has been a considerable amount of work in improving content delivery on the downstream path for various network conditions (e.g., [47, 64, 65, 66]). In contrast, Vantage focuses on improving quality of experience by mitigating network variability on the *upload path*. Targeting the upload path is critical since techniques for improving the download quality are rendered ineffective if the quality of the uploaded video was poor to begin with. This is especially relevant for SLVS platforms due to the high degree of variability of mobile networks over which streams are commonly broadcasted.

Prioritization in multimedia. Several previous works have developed transport protocols [67] and application-specific optimizations (e.g., 360-video [68, 69, 70, 71]) which prioritize multimedia to improve quality. Vantage uses a similar technique (of selective prioritization) to prioritize real-time frames over retransmissions.

3.9 Conclusion and Key Takeaways

In this chapter, we presented the design of our system, Vantage. Vantage, a live video upload system that explicitly accounts for the diverse time-shifted viewing delays common in social live video streaming platforms. Vantage balances available upload bandwidth between real-time frames and quality-enhancing retransmissions of previously impaired frames, resulting in high QoE for real-time and time-shifted viewing *simultaneously*. Compared to existing live video upload solutions tailored for either real-time or delayed viewing, Vantage improves the SQI-SSIM for time-shifted viewing by 19.9% on average, with only minor reductions in real-time or delayed quality.

The key takeaways from this chapter are

1. Traditional notions of QoE for video streaming applications are often not applicable for emerging video streaming applications, or consider a severely limited perspective of the QoE for applications like SLVS.
2. A carefully designed system that targets the specific QoE needs of a video streaming application can significantly improve the end-user experience.
3. The use of cross-layer designs that leverage insights regarding the properties of video compression algorithms in order to perform tasks like bitrate selection and frame scheduling is a very useful paradigm in order to address the QoE requirements of emerging video streaming applications.

Chapter 4

Prism: Handling packet loss for ultra-low latency video.

Steady improvements in networking algorithms, hardware, and video coding techniques have enabled the Internet to support a wide range of video applications (e.g. live conferencing, large-scale broadcasts, video-on-demand). Recently, a new class of video streaming applications has emerged: *ultra-low latency* interactive video. These applications offload compute and rendering to the cloud, and stream the view-port to the end user. Examples of these applications include virtual game consoles (Stadia [16], GeForce NOW [17], XCloud [72]), cloud AR [73], and virtual desktop (Chrome Remote Desktop [74], Azure Virtual Desktop [75]). These applications push the limits of existing streaming techniques and network infrastructure, and have extremely demanding QoE requirements (video stream quality and end-to-end frame delay).

The Internet's design fundamentally follows a best-effort philosophy - there are no guarantees for reliable delivery of packets. As a result, applications must deal with issues such as packet loss [76], which is especially harmful for low-latency video streaming applications due to the interdependence between frames in compressed video (e.g. P- and B-frames in MPEG). Loss of video data as a consequence of packet loss stalls the video decoder pipeline, since past frames need to be decoded in order to decode a P-frame.

In this chapter, we present a system called Prism. Prism incorporates a novel frame transport protocol, optimizations at the video codec level, and deep-learning based packet loss prediction to significantly improve the QoE by achieving smoother video playback without significantly sacrificing video quality.

4.1 Packet Loss Mitigation for Real-Time Video

Packet loss mitigation strategies fall into four broad categories:

1. **Preventing data loss:** Techniques like FEC [77, 78] transmit redundant packets to avoid data loss even if some packets are lost.
2. **Concealing lost data:** Techniques like partial P-slice decoding [79, 80], error concealment [81, 82], joint source-channel coding [20], and scalable coding [57, 83] (SVC) are able to continue decoding the video at a reduced quality when packet losses occur.

3. **Recovering lost data:** Packet- or frame-level retransmissions [84] can be used to recover lost video data, and has the benefit zero bandwidth or performance overhead when no losses occur.
4. **Recovering from data loss:** IDR-frames and slice-based intra-refresh [85] result in lower delays compared to (3) by eliminating the need for the video decoding pipeline to catch up to the current frame.

Techniques that prevent or conceal data loss (FEC, partial decoding, error concealment, scalable coding) enable packet loss mitigation without requiring receiver feedback, and thus, do not incur a round-trip penalty for loss recovery. On the other hand, these techniques break down when packet losses exceed a certain threshold, resulting in non-recoverable video data loss. Non-recoverable video data loss when using FEC has the same implications as a packet loss when not using FEC, whereas with techniques like loss concealment, packet losses manifest as undesirable artifacts that significantly degrade the QoE for immersive applications like cloud gaming, AR, and VR.

There are two techniques used for recovering from video data loss: packet retransmissions, and IDR frames. These solutions force applications to make difficult tradeoffs between accepting significantly degraded picture quality, or significantly higher frame delay when affected by packet loss [79, 86]. For instance, during bursty loss events, transmitting IDR frames reduces frame delay since they can be independently decoded. On the other hand, since IDR frames do not leverage temporal redundancies in video data, they require more bits per frame compared to P-frames for equivalent picture quality. In the case of retransmission based recovery, high picture quality is maintained since the frames are still encoded as P-frames. Unfortunately, since the receiver accumulates a backlog of undecoded frames until the lost data has been recovered, the end-to-end frame delay remains high until the decoder catches up.

In order to recover from non-recoverable video data loss, the sender must first be notified, after which the sender can retransmit packets or send IDR frames to recover from the data loss event. This process incurs a minimum delay of 1 RTT before the sender can trigger loss recovery. In order to reduce this delay, one could use speculative loss prediction mechanisms [87, 88], but this approach has the risk of false positives, which can affect the video quality significantly (eg. when using IDR frames).

4.2 Overview of Contributions

In this chapter, we present the design of our system called Prism. Prism is a hybrid predictive-reactive packet loss recovery scheme that uses a split-stream video coding technique to meet the needs of ultra-low latency video streaming applications. Prism's approach enables aggressive loss prediction, rapid loss recovery, and high video quality post-recovery, with zero overhead during normal operation - avoiding the pitfalls of existing approaches.

Prism leverages the insights that (1) IDR-frames enable rapid recovery of a video stream after a loss event, since they are immediately decodable and reset the decoder state, and (2) A stream of P-frames can sustain high quality for some time even after a reduction in bitrate, and thus enable rapid recovery post-recovery. When Prism identifies a potential loss event, it splits the video stream into two substreams - a low-latency, unreliable IDR-frame stream, and a high quality, reliable P-frame stream. The IDR-frames enable the application to continue displaying frames with low delay, albeit at a lower quality. When the lost P-frame data is retransmitted and the pending frames are decoded, the application can quickly switch back to the higher quality P-frame stream. Prism carefully allocates the total available bandwidth between the two streams by using results from a novel video analysis and optimization pipeline that runs offline, and thus avoids real-time computational overhead.

Prism's approach reduces the impact of falsely triggering loss recovery, enabling the use of deep-learning based loss prediction. This allows Prism to take action before potential losses occur, reducing the time to recovery. When Prism's recovery mechanism is falsely triggered, the receiver can simply decode and display the high quality P-frame stream. Prism eliminates the hard tradeoffs present in traditional approaches and provides a more flexible trade-off between delay and video quality.

Prism's key contributions include:

1. **Split Stream Video Coding:** During packet loss, Prism splits the available bandwidth across a low latency IDR-frame stream and a high quality P-frame stream. This enables low frame delay during loss events, and ensures high video quality post recovery. In addition, the impact of falsely triggering Prism's loss recovery is low; this enables Prism to use loss prediction mechanisms to respond early to potential packet loss.
2. **Deep-learning Based Loss Prediction:** Prism incorporates deep-learning based packet loss prediction, allowing Prism to react to losses earlier, reducing the network round-trip penalty for recovery from packet loss.
3. **Video Analysis Pipeline:** Prism analyzes video scenes offline to optimize the bandwidth allocation across the two video streams. Prism's novel black box encoder modeling technique provides fast approximations of the video quality for given bitrate configurations, speeding up the optimization pipeline by multiple orders of magnitude.

Our evaluation of Prism's real-time implementation using a diverse set of video game footage and on a variety of emulated network conditions shows that Prism reduces the quality penalty of using IDR-frames for loss recovery, while preserving the low delay benefits of using I-frames for recovery. Simulation of Prism's algorithm using discrete event simulation techniques on real world network traces from M-Labs [89] shows that Prism can outperform IDR-frame based recovery and P-frame transmissions by handling packet loss better in the presence of significant delay and bandwidth variation, reducing the quality penalty of I-frames by 81 % on average. Additionally, Prism's aggressive loss prediction mechanism reduces delay by proactively triggering loss recovery, and in conjunction with Prism's split-stream approach, achieves much higher video quality compared to using I-frames for recovery with loss prediction.

4.3 Video Compression Background

Video frames are comprised of slices, where each slice can be one of three types: (1) Intra (or I), (2) Predicted (or P), and (3) Bidirectional (or B). An I-slice is encoded and decoded independently, whereas P-slices depend on past frames and B-slices may depend on both, past and future frames. P- and B-slices leverage inter-frame redundancy by using motion compensation [90] to approximate the slice by using localized motion vector references to past and future frames, significantly reducing the bitrate requirement to achieve the desired picture quality.

Low latency streaming applications rely heavily on the use of IDR-frames (independent data refresh) and P-frames. These applications encode video as P-frames during normal operation due to their compression efficiency - thus, a single missing packet stalls prevents the decoding of subsequent frames. IDR-frames are special I-frames (contains only I-slices) that are commonly used to recover from video data loss, since they reset the decoder state which can then discard previous undecoded frames and decode the most recent frames with minimal delay. In the subsequent text, we will use the terms IDR-frame and I-frame interchangeably since we are talking about loss recovery. It is possible to have an I-frame that does not reset the decoder state. In this case, the P-frames that come after this frame can still use other past frames as reference. These frames are often used in VOD content in order to support efficient seeking operations during playback.

Prism leverages some key properties of video compression - (1) The quality of an I-frame depends only on the bitrate and the content of a particular frame, (2) I- and P-frames demonstrate diminishing returns with respect to the video quality as the bitrate increases, and (3) In addition to motion, frame content, and the bitrate, the quality of a P-frame also depends on the quality of previous frames since they are approximated from previous frames using motion vectors.

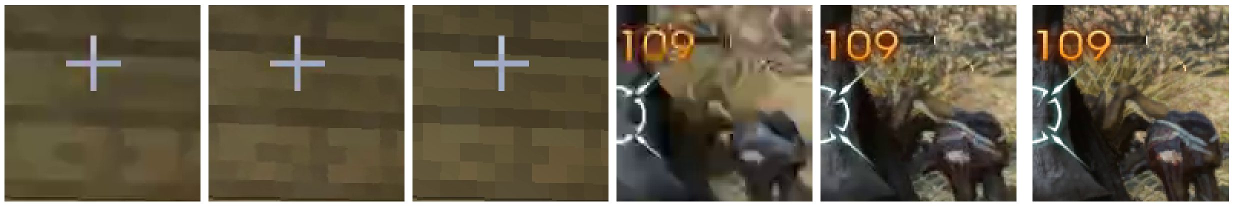
4.3.1 Understanding Video Quality - SSIM

SSIM (Structural Similarity [27]) is an objective metric that is used to evaluate the visual quality of a video frame. SSIM is computed by comparing small windows extracted from the source frame (reference frame) and the compressed frame, and uses the average, variance, and covariance of the pixel values in these windows. We use FFMPEG (*lavfi* [91]) to compute the SSIM values (the parameters used for SSIM like window size can vary across implementations). It's value typically ranges from 0 (worst quality) to 1 (best quality) (**Note:** The raw SSIM value can be negative, if the covariance between the two frames being compared is negative. Most tools will adjust the value to be in a range of 0-1 in a linear manner). In this chapter, we use this metric in order to evaluate the quality of a frame, and also as the metric to optimize for in our system. Note that we could use any other quality metric, like the peak signal-to-noise ratio (PSNR) or VMAF [92]. It is useful to get a high level understanding of what different SSIM values mean, and how video content influences the numerical SSIM values.

The visual quality difference between I-frames and P-frames for Minecraft (video with the lowest difference) and for FF XV (video with the highest difference) is shown in Figure 4.1.

In the case of Minecraft, while the difference in SSIM between I-frames and P-frames for Minecraft is small, the visual impact is significant in terms of what Minecraft should look like. Since Minecraft uses macro-pixel like textures, if the pixel edges are blurry, this looks quite different from what Minecraft is expected to look like. On the other hand, the SSIM value does not change much, because in the SSIM computation window, the edge of a macro-pixel plays a much smaller role. This is because the edges of the pixel textures in Minecraft comprise a smaller portion of the overall frame - the blurry edges only reduce the SSIM by a small amount.

Thus, even if the difference in SSIM values is small, there can be a large difference in visual quality for some videos. This makes it harder to interpret the visual quality based on absolute



(a) Minecraft I-frame (0.91) (b) Minecraft P-frame (0.96) (c) Minecraft Raw (d) FF XV I-frame (0.62) (e) FF XV P-frame (0.90) (f) FF XV Raw

Figure 4.1: Visualization of the difference between I-frames vs P-frames at 10 Mbps, for the videos with the highest and lowest difference in average quality. The SSIM is denoted inside parentheses.

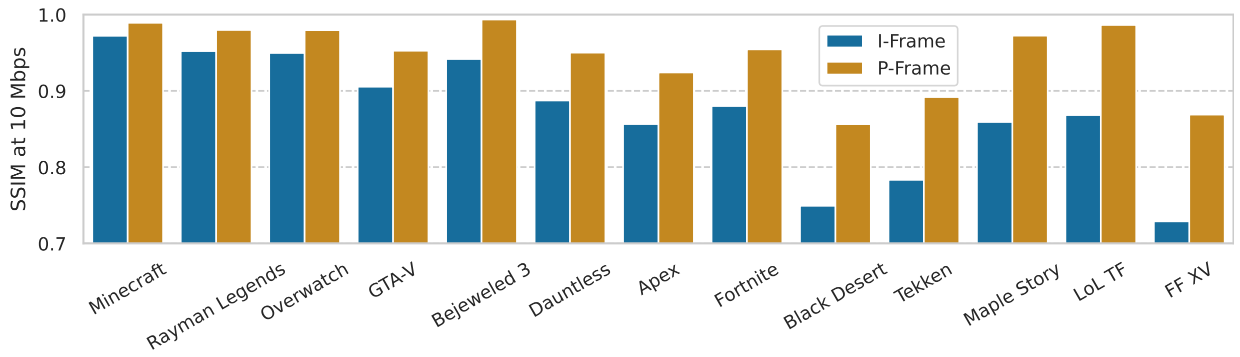


Figure 4.2: Comparison of average video quality using just I-frames and just P-frames at 10 Mbps, sorted in the order of increasing difference between the average quality.

values of SSIM. This is not a significant factor for our use case, since we never compare absolute SSIM values across different videos. Later, in our evaluation (Section 4.6), instead of the absolute difference in SSIM values, we use the relative difference between the structural dissimilarity (DSSIM), defined as $(1 - SSIM)$ for SSIM values that have been adjusted to be in the range $[0, 1]$, or $\frac{(1-SSIM)}{2}$ for raw SSIM values in the range $[-1, 1]$. The relative DSSIM values represent how much closer the video frames are to the source video frame instead of a baseline SSIM value of 0 (or -1, in the case of raw SSIM values that lie between -1 and 1).

For example, in the case of Minecraft (shown in Figure 4.1), the DSSIM of (b) is 0.04, and the DSSIM of (a) is 0.09. Thus, (b) represents a 56% reduction in distortion over (a) with respect to the source image, (c). Similarly, in the case of FF-XV, (e) represents a 74% reduction in distortion over (d) with respect to the source image, (f).

4.3.2 I-frames vs. P-frames Compression Efficiency

In Figure 4.2, we compare the average video quality when the video is encoded using only I-frames, and when the video is encoded using P-frames at 10 Mbps. We analyze videos in the CGVDS [93] dataset, and the videos are described in Section 4.7. The videos are sorted in the order of increasing difference in the quality from left to right. The difference in video quality between I-frames and P-frames is a key factor that influences Prism's design, which we discuss

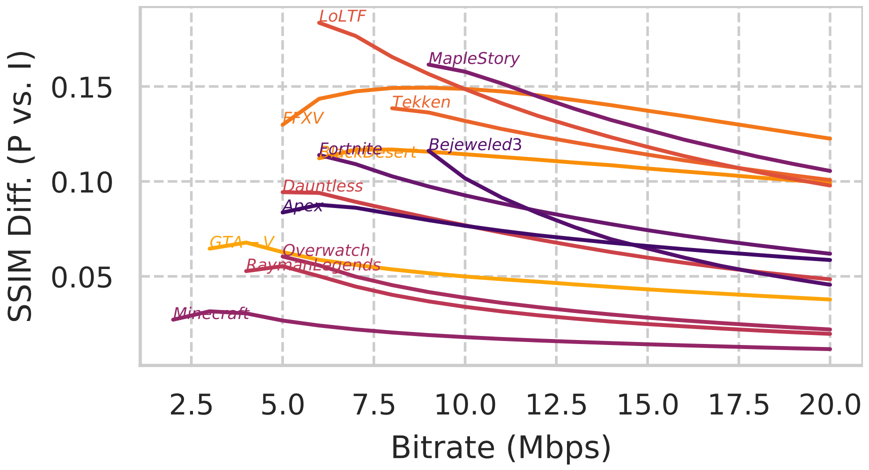
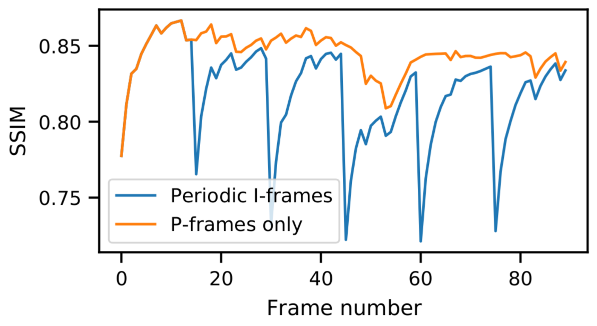


Figure 4.3: Difference between P-frame quality and I-frame quality across a range of bitrates.

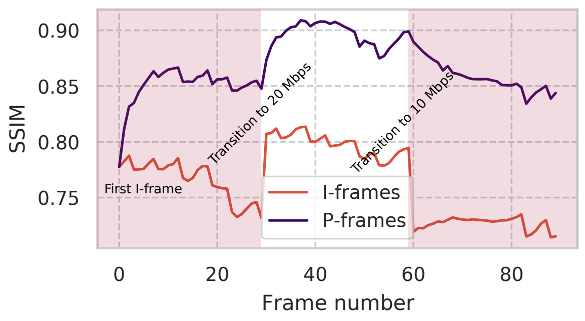
in Section 4.5.2. The difference in quality between I-frames and P-frames is affected by the following factors:

1. **The baseline quality of I-frames**: If a video has simple textures and areas with large, flat colors, intra-compression is very effective and the video quality is already high, which reduces the impact of motion compensation. This is true for videos like Minecraft (large area occupied by sky, areas inside the pixel textures are flat colors), Rayman Legends, Overwatch, and Bejeweled 3 (simplified textures). On the other hand, I-frames are inefficient when videos have complex textures, and the relative difference between I-frame and P-frame coding efficiency can be very large, i.e. P-frames may be highly beneficial in such scenarios.
2. **The efficacy of P-frames**: If a video has complex textures, and the inter-frame motion is not a simple 2D translation or affine transform (eg. 3D first person/ third person games), motion compensation is not very effective. This is true for videos like Minecraft (the edges of the pixelated blocks don't align well after motion compensation) and Overwatch (aliasing in frames is not stable across frames). Motion compensation is very effective for videos like LoL TF (static background) and Maple Story (translating background). In addition, when the video quality achieved using I-frames is low, there is a larger gap in the video quality between using just I-frames versus using P-frames and motion compensation. This is because there is more room for improvement in the video quality (since there are decreasing returns if the baseline quality is already high), and thus using P-frames tends to result in a larger increase in SSIM (eg. Black Desert, FF XV).

The impact of bitrate on the difference between I-frame and P-frame quality is shown in Figure 4.3. The difference in video quality reduces as the bitrate increases. In the case of a local network game streaming setup, where the video bitrate can be much higher, the best approach to mitigate the adverse impact of packet loss can be to simply transmit I-frames all the time, since there is enough bandwidth available that I-frames achieve sufficient video quality. This avoids video decoding stalls caused by the loss of P-frames.



(a) Impact of I-frame insertion on the video quality.



(b) Change in video quality with bitrate changes.

Figure 4.4: Temporal behavior of the video quality of P-frames when encoding parameters are changed (e.g. I-frame insertion, bitrate change).

4.3.3 Bitrate Transitions and P-frame Quality

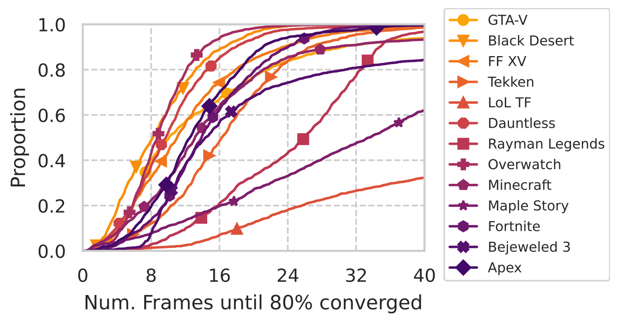
Another video coding property that is relevant for Prism's design is how the video quality of a P-frame stream converges to the steady state video quality. Since P-frames use past frames as reference, for a given bitrate, the video quality of a P-frame will be lower if the reference frame used during video encoding has lower quality. Eventually, the video quality of the P-frames will converge to the steady state. For example, the gradual convergence behavior of P-frames determines the video quality when the bitrate of a P-frame stream is changed, or when an I-frame is inserted in a video stream.

In Figure 4.4a, we show the SSIM of a sample video encoded as P-frames. The Orange line shows the video quality of the P-frame stream at a fixed bitrate (10 Mbps). The Blue line shows the quality of the P-frame stream at the same bitrate, but with I-frames inserted periodically (every 15 frames). The I-frames are encoded at the same bitrate, and thus, have lower quality than the steady state P-frame stream. Note how the insertion of an I-frame not only results in an immediate drop in the video quality, but the low quality is sustained across multiple frames. This is the main challenge with using I-frame based recovery for real-time streaming video - I-frames have a significant impact on video quality, and can result in pixelation and block artifacts - this is highly undesirable for cloud-streaming applications like cloud gaming and cloud AR/VR.

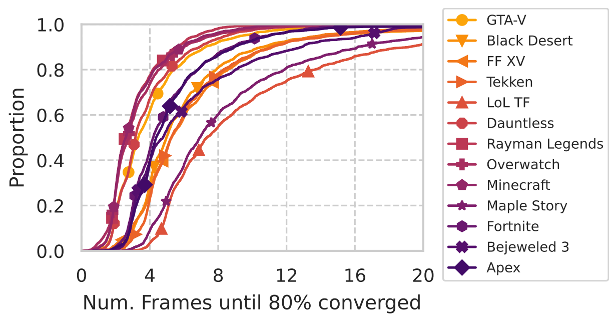
In Figure 4.4b, we compare the temporal behavior of SSIM when the bitrate is changed. We encode the video twice, once purely as I-frames, and a second time as a pure P-frame stream. The bitrate is initially set to 10 Mbps, and is increased to 20 Mbps at frame 30. At frame 60, the bitrate again drops to 10 Mbps. The red line shows the quality of the I-frame stream. In this case, the change in quality is instantaneous - the SSIM of an I-frame only depends on the contents of the frame and the encoding bitrate. The purple line shows the quality of the P-frame stream. In this case, the SSIM increases gradually when the bitrate is increased, and decreases gradually when the bitrate is reduced. This property is a core aspect of Prism's design, and we discuss this in Section 4.4.3.

In Figure 4.5, we plot the distribution of the number of frames until a P-frame stream converges to 80% of the steady state quality when the bitrate changes (increase and decrease).

When motion compensation is very effective, the video quality decays slowly when the bitrate

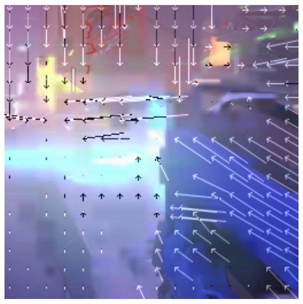


(a) Bitrate Decrease.

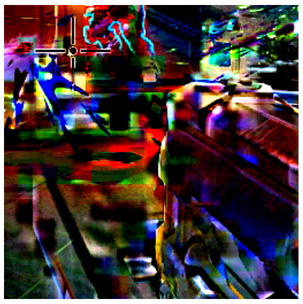


(b) Bitrate Decrease.

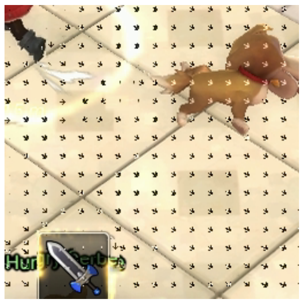
Figure 4.5: Distribution of the time taken by a P-frame stream to converge to 80% of the quality of a steady-state P-frame stream.



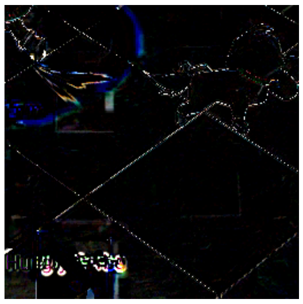
(a) Overwatch motion vectors



(b) Overwatch residuals



(c) MapleStory motion vectors



(d) MapleStory residuals

Figure 4.6: Visual comparison of motion compensation performance for two videos. Motion compensation is more effective at reducing the bitrate requirements for Top-down 3D scrolling games like MapleStory.

is lowered (eg. LoL TF and MapleStory). On the other hand, motion compensation is less effective for videos like Overwatch, and each compressed frame consists of a large amount of encoded residuals, making each frame more like an I-frame. Thus, the video quality converges quickly, and Prism is less effective.

On the other hand, when intra-compression is more effective (eg. Rayman Legends), the quality increases quickly to converge to the steady state. This results in higher video quality for the I-frame baseline, resulting in lower gains for Prism (eg. Rayman Legends in Figure 4a in the main paper).

In Figure 4.6, we show the motion vectors overlaid on a motion compensated frame for Overwatch and MapleStory, and also the residual difference between the motion compensated frame (reconstructed using just the motion vectors) and the uncompressed frame. In the case of Overwatch, the motion compensated frame differs significantly from the uncompressed frame, whereas in the case of Maple Story, the motion compensated frame is very close to the uncompressed frame, with small residuals present at sharp edges and object boundaries. Thus, P-frames in the case of Overwatch behave closer to I-frames, since a large component of the compressed video frame data is the residual. Thus, the SSIM transition times for the P-frames are much lower. On the other hand, the P-frames of Maple Story are largely comprised of motion vectors, with very small residuals that need to be encoded. Hence, the SSIM transition times are much longer.

4.4 Loss Detection and Recovery

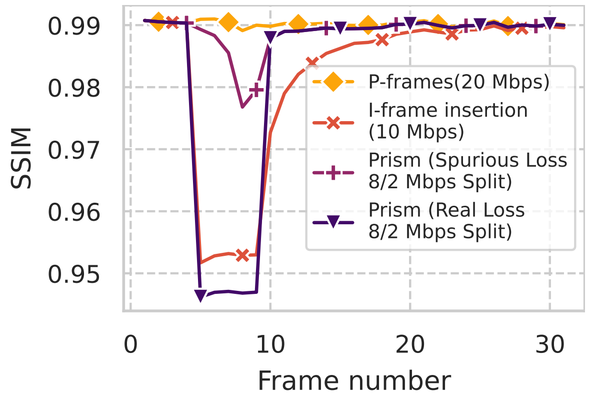
The goal of this paper is to design a video data loss recovery mechanism that *simultaneously* achieves (1) low delay and smooth video playback, (2) high picture quality, (3) minimal quality impact under false packet loss triggers, enabling aggressive loss prediction without significant penalties, and (4) zero overhead under normal operation. While loss prevention techniques (e.g. error coding, FEC) can reduce the frequency of video data loss with some bandwidth and compute overhead, they provide no guarantees. Transmitting IDR-frames on video data loss achieves (1) and (4), and retransmission of lost P-frame packets achieves (2), (3) and (4). In the following subsections, we discuss: (1) the impact of loss detection mechanisms, (2) the two reactive recovery mechanisms, and (3) an example in Section 4.4.3 to show how Prism's split stream approach can achieve all of the above properties simultaneously.

4.4.1 Loss Detection

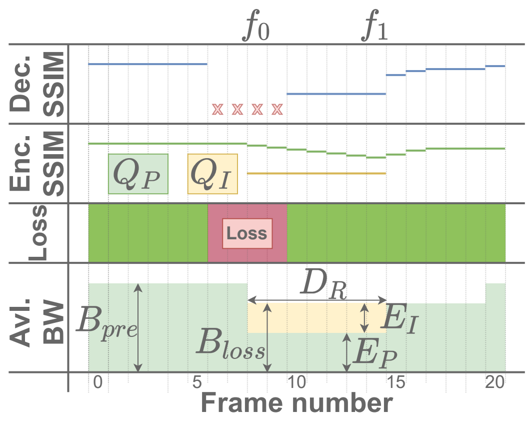
The sender getting notified of video data loss is the first step in loss recovery, which then triggers the loss recovery mechanism. Since packet losses are often accompanied by periods where no ACKS are received, or an increase in delay (e.g. due to cross traffic running loss-based TCP), speculatively triggering loss recovery using aggressive loss prediction (e.g. sub-RTT timeouts, loss prediction using machine learning) can reduce video stuttering and frame delays [94, 95]. On the other hand, noisy networks, queue-building cross traffic (e.g. BBR [96]), packet reordering and ACK loss [97] may result in frequent false triggers. Excessive false triggering of recovery mechanisms (like IDR-frames) harms video quality (§ 4.3). Delaying recovery until packet loss can be verified results in significant video stutter, which is unacceptable for ultra-low latency video streaming applications. Prism drastically reduces the impact of false loss recovery triggers on video quality, and thus enables the use of aggressive loss prediction mechanisms in order to reduce video stutter under packet loss.

4.4.2 Reactive Loss Recovery

Packet Retransmissions: A video data loss in a P-frame stream can be recovered using packet retransmissions. This approach achieves high picture quality by leveraging the compression efficiency of P-frames due to motion compensation, but results in higher end-to-end frame delays and video stutter. Consider a video stream being sent over a link with an RTT of 60 ms. When a loss occurs, the sender is only notified 60 ms after having sent the original data. During this period, the sender keeps encoding frames as P-frames, and these frames get queued at the decoder without being displayed, until the lost packets are retransmitted. On receipt of the retransmissions, the decoder needs to "catch up" to the current frame, which may take a long time depending on decode performance. For example, for a 60FPS video where each frame takes



(a) SSIM behavior when inserting I-frames or changing P-frame bitrate.



(b) Prism timeline depicting the flow of events when a packet loss occurs.

Figure 4.7

$10\,\mathrm{ms}$ to decode, catching up with $60\,\mathrm{ms}$ of backlogged frames can take up to 100 ms [1]. When streaming in 4K to low power client devices like mobile phones and TV streaming sticks, the decode time can be much closer to the frame time, significantly increasing the recovery delay.

I-frames: When packet loss occurs, transmitting the latest frames as I-frames (IDR) resets the decoder state, and thus, the latest frames can be decoded and displayed immediately. Using I-frames in conjunction with aggressive loss detection can significantly reduce video stutter and improve video smoothness. Unfortunately, (1) IDR frames have much lower video quality compared to P-frames encoded at the same bitrate, and (2) the low quality of an IDR frame also affects the video quality of subsequent P-frames, since their quality depends on the video quality of past frames.

4.4.3 Proposed Hybrid Approach

The two loss recovery mechanisms discussed above have their own benefits and downsides: P-frame retransmissions achieve higher picture quality, but incur higher delays and stutter. On the other hand, IDR-frames improve video smoothness and reduce latency at the cost of video quality. In Prism, we propose a hybrid architecture that combines the two mechanisms to achieve a better trade-off between latency and video quality by leveraging two key properties (§ 4.3): (1) The diminishing returns of video quality with higher bitrate, and (2) the dependence of the quality of a P-frame on the quality of past frames. When packet loss occurs, the quality of a P-frame stream can be preserved for a short duration at a reduced bitrate, while the residual bandwidth can be used for transmitting I-frames in order to maintain low delay until the P-frame stream recovers using packet retransmissions.

In Figure 4.7a, we encode a video segment for "GTA-V" using different encoding configurations. The yellow line is the steady state P-frame SSIM at $20\ \mathrm{Mbps}$ (best case scenario, no packet loss). The line marked "I-frame insertion" demonstrates the instantaneous drop in picture

[1]Let x be the delay after the loss until P-frames recover. Frames decoded: $\frac{x}{10}$, frames sent: $\frac{60+x}{16}$. Solving, we get x=100

quality, and the impact on the quality of subsequent P-frames when I-frames at 10 Mbps are used during loss recovery (between frames 5 and 10). During loss recovery, Prism splits the available bandwidth in order to continue the P-frame stream (at a lower bitrate), while additionally transmitting low delay I-frames. As an example, suppose we allocate 2 Mbps for the P-frame stream and 8 Mbps for the I-frames during loss. The I-frames reduce latency during the ongoing loss event, and the receiver switches back to the P-frame stream after frame 10. The key insight here is that when the bitrate is reduced, the quality of the P-frame stream is preserved for a short duration. The line marked as "Real Loss" denotes Prism's video quality when real loss occurs - while the quality during loss is slightly lower compared to I-frames (at 10 Mbps), the video quality post-recovery is much higher. The line marked as "Spurious Loss" denotes Prism's video quality when loss recovery is falsely triggered (i.e. the quality of the P-frame stream), and thus, Prism's approach works well with aggressive loss-prediction techniques as opposed to I-frame insertion.

4.5 Design

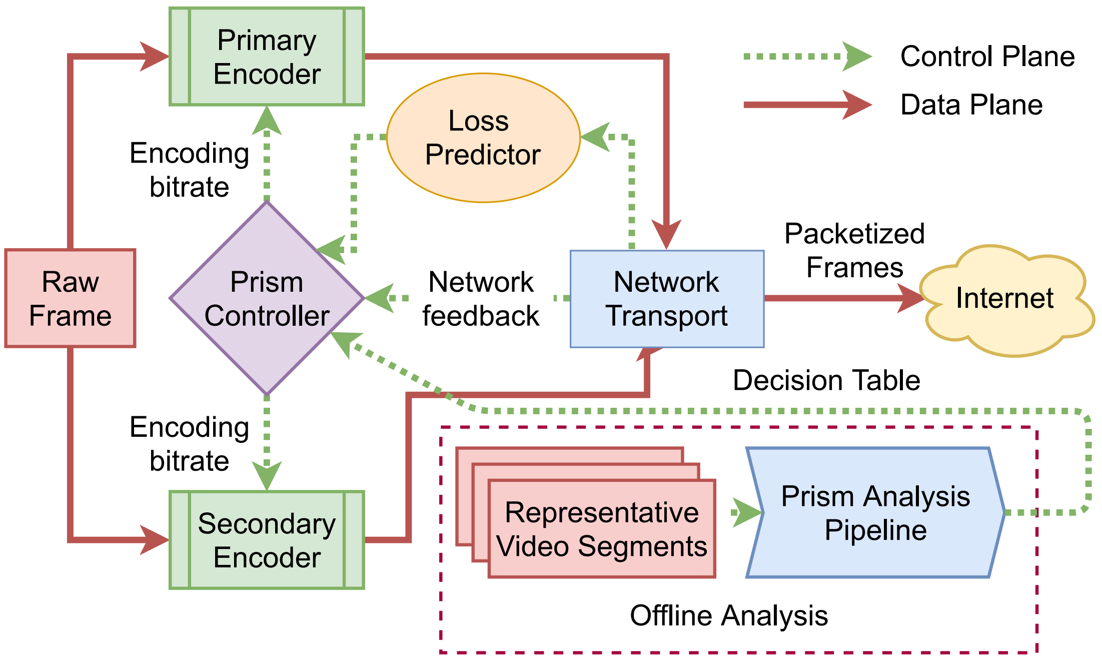
In this section, we discuss Prism's system design, and how it robustly mitigates video data loss for interactive video streaming applications like cloud gaming, and AR/VR streaming.

4.5.1 Overall Architecture

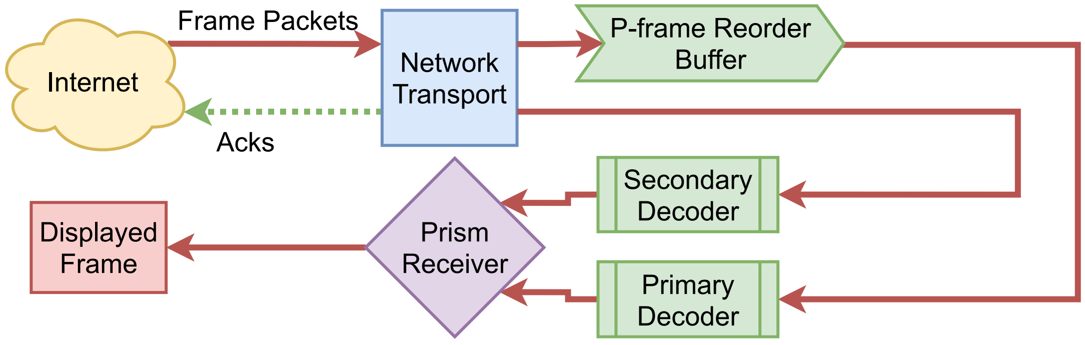
Figures 4.8a and 4.8b show Prism's streamer and receiver architectures respectively. The network transport layers at the streamer and the receiver are responsible for packetization and multiplexing of the two video streams over a single connection. In order to trigger loss recovery, the Prism controller combines signals from a deep-learning based loss predictor and loss signals from the transport layer. The controller also determines the allocation of the available bandwidth across the two video streams.

When no packet loss occurs, frames are encoded as P-frames using all of the available bandwidth. The receiver reassembles the packets, decodes and displays the frame. When the transport layer or the loss predictor indicates packet loss, each frame is encoded as a P-frame (by the primary encoder) and an IDR-frame (by the secondary encoder), and the Prism controller allocates the total bandwidth across the two streams. The P-frame is transmitted reliably using packet retransmissions, and the IDR-frames are transmitted as one shot (no retransmissions). The P-frames after a packet loss are stored in a buffer at the receiver while the transport layer retransmits the lost packets. Meanwhile, the IDR-frames that make it through are immediately decoded and displayed, maintaining low video latency while the P-frame stream recovers from the loss.

Prism needs to optimize the bandwidth allocation in order to maximize the benefit of this approach, and ensure that the quality is better than simply using I-frames. Prism's offline analysis pipeline uses prerecorded training video sequences to compute compact bandwidth allocation tables using a greedy heuristic in order to make bandwidth allocation decisions in real-time (§ 4.5.2). Prism uses a novel black box encoder modeling technique that enables fast and accurate estimation of the video quality for arbitrary video encoding schedules (bitrate, IDR-frame inser-



(a) Streamer architecture.



(b) Receiver architecture.

Figure 4.8: Prism architecture.

tions) without actually encoding the video, which reduces the amount of computation required for generating the bandwidth allocation tables by multiple orders of magnitude (§ 4.5.3).

4.5.2 Optimizing Bandwidth Allocation

Consider the timeline shown in Figure 4.7b, with the X-axis denoting the frame number. In the example, no packet losses occur until frame 5, and frames 6 through 9 are affected by packet loss (shown in the loss timeline). The video bitrate of the P-frame stream before loss recovery is triggered is B_{pre} Mbps, and the available bandwidth during the period of loss recovery is B_{loss} (B_{pre} and B_{loss} are determined by the congestion control and rate control algorithms, assumed to be constant here for simplicity). During the recovery period, let the encoding bitrate of the I-frames and the P-frames be $E_I(f)$ and $E_P(f)$ respectively, where f is the frame number. This gives us the constraint

$$E_I(f) + E_P(f) = B_{loss} \ \forall \ f_0 \leq f \leq f_1 \tag{4.1}$$

Since the duration of loss is not known beforehand, we propose a greedy optimization strategy: when transmitting a frame during loss recovery, Prism splits the bandwidth assuming that it is the last frame in the loss recovery state. Thus, given the bandwidth split until frame f_i, Prism needs to determine the allocation for frame f_{i+1} that balances three things: (1) The video quality during loss (the quality of the I-frames), (2) the video quality after the loss event (quality of the last P-frame sent during loss recovery), and (3) the video quality when the loss is spurious (quality of the P-frames during loss recovery).

We first derive the optimization objective for a known video segment, where the loss recovery begins at f_0 and ends at f_1. The mappings between video bitrate and video quality are defined as:

1. $Q_I(x, f)$: The quality of frame f when encoded at a bitrate x as an I-frame.
2. $Q_P([x_1...x_f], f)$: The quality of frame f when frames $1...f$ are encoded as P-frames, and the bitrate of frame i is x_i.

For a frame f transmitted during loss recovery, we define two objective functions, O_1 and O_2, to denote the video quality during real loss, and when loss recovery was falsely triggered respectively. When real loss occurs, the video quality is determined by the quality of the I-frames (Q_I in Figure 4.7b) and the quality of the P-frames after recovery (Q_P after f_1 in Figure 4.7b). If the I-frame allocation is x,

$$O_1(x) = Q_I(x, f) + \sum_{i=f+1}^{f+K} Q_P([x_{f_0}...x_{f-1}, B_{loss} - x, B_{loss}...B_{loss}], i) \tag{4.2}$$

where K is a fixed horizon of future frames that are assumed to be encoded at B_{loss} to account for the quality convergence of the P-frames, and $x_{f_0}...x_{f-1}$ are fixed according to the allocations for past frames during the current loss event (greedy approximation).

When spurious loss occurs, the video quality is determined solely by the quality of the P-frame stream. Thus,

$$O_2(x) = \sum_{i=f}^{f+K+1} Q_P([x_{f_0}...x_{f-1}, B_{loss} - x, B_{loss}...B_{loss}], i) \tag{4.3}$$

This allows us to define objective function for determining the bandwidth split for a frame f:

$$x_f = \arg\max_x w \cdot O_1 + (1 - w) \cdot O_2 \tag{4.4}$$

Here, w is the weight for real loss, and $1 - w$ is the weight for false loss recovery triggers. w is set based on the accuracy of the loss prediction mechanism (eg. a smaller value for w is better if false loss recovery triggers are frequent).

In order to find the best split, it is sufficient to sweep different values of x, and pick the value that maximizes the objective function. Unfortunately, this optimization cannot be solved in real-time since objective function requires the quality of future frames, which are not available for real-time applications. Prism's key insight is that while video properties can vary across individual frames, the result of the optimization is "stable" for a given video style. In 4.5.3, we describe an offline approach to compute bandwidth allocation tables for a particular video style using prerecorded training segments, enabling Prism to run in real-time without any runtime computational overhead.

4.5.3 Offline Analysis Pipeline

Solving the optimization problem in real-time when a loss event occurs is impractical not only from a delay or overhead perspective, but also not possible since the objective function includes the quality of future frames. To address this issue, Prism analyzes pre-recorded video segments that are representative of particular scene types, and runs the optimization for a wide range of bandwidth values and loss durations, for various starting points in the pre-recorded video. The results are aggregated across the starting points in the video into a decision table that maps $(B_{pre}, B_{loss}) \rightarrow [x_f ... x_{f+D}]$, where D is a maximum limit on loss recovery duration before Prism falls back onto traditional techniques. Prism uses the appropriate decision tables for the particular video style in order to determine the bandwidth split in real-time during loss recovery.

Consider the optimization run for a particular section of a pre-recorded video segment. If we limit the maximum loss duration to 10 frames, and limit B_{pre}, B_{loss}, and x_f to $1 - 20$ Mbps in steps of 1 Mbps, the brute force approach requires $\approx 20 \times 20 \times 10 \times 10$, ie. 40000 different encoding schedules to be evaluated. In addition, for each evaluation, we must consider around 1 second of video before and after loss recovery for allowing the quality of the P-frames to converge. Thus, analyzing a single location in a training video requires 80000 seconds of video encoding (22 hours). This scales up linearly as we sample more locations in the training video in order to generalize the decision table for a particular video style.

4.5.3.1 Black box encoder modeling

In Prism, we propose a unique approach to reduce the complexity of the offline analysis step, speeding it up by multiple orders of magnitude. Prism analyzes data from a small set of carefully designed video encoding runs, which enables Prism to accurately predict video quality for the given video sample when (1) a frame is encoded as an I-frame, (2) the encoding bitrate of a P-frame stream is changed, or (3) an I-frame is inserted in a P-frame stream. This enables Prism to avoid encoding the actual video to compute the objective function when performing a brute force sweep of the search space for determining the optimal bandwidth splits. To our knowledge, this method is unique to Prism and has not been published in past work.

Suppose we are given a long sample video of duration N seconds. First, we encode the video only using I-frames for all bitrates ranging from $1-20$ Mbps ($20 \times N$ seconds of video encoded), enabling us to compute $Q_I(x, f)$. Second, we sample K segments (each segment starts at frame f, duration of 1 second) from the video, and encode each segment in the following manner for each bitrate E value between 1 Mbps and 20 Mbps:

1. Encode the first frame at a very low bitrate, and the subsequent frames at the chosen bitrate as P-frames ($20 \times K$ seconds of video encoded). Let's denote this family of functions as

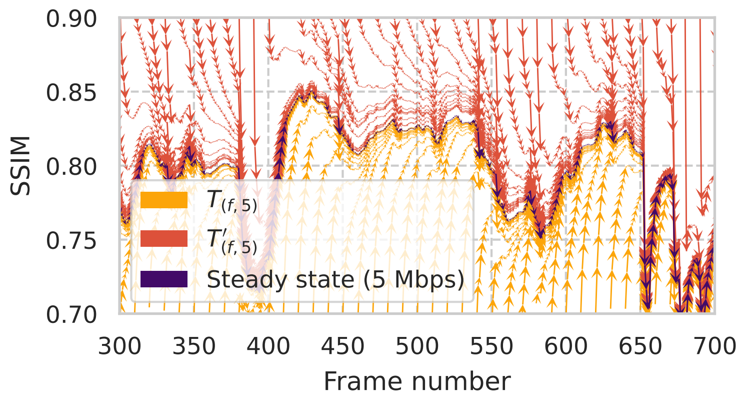


Figure 4.9: $T_{(f,5)}$, $T'_{(f,5)}$ for various starting points (f) (Tekken)

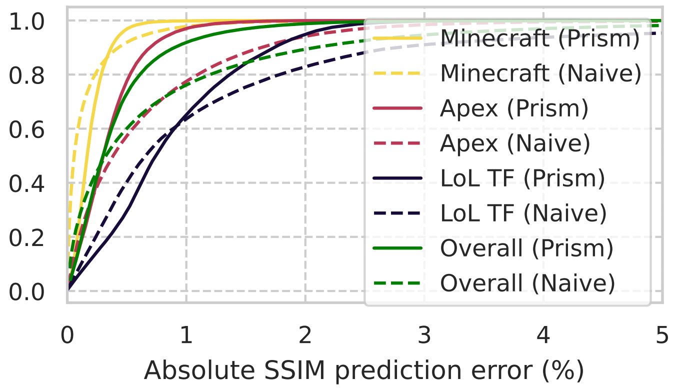


Figure 4.10: SSIM prediction error - Prism vs. naïve exponential decay.

$T_{(f,E)}$, shown as the red flow lines in Figure 4.9.

2. Encode the first frame at a very high bitrate, and the subsequent frames at the chosen bitrate as P-frames ($20 \times K$ seconds of video encoded). Let's denote this family of functions as $T'_{(f,E)}$, shown as the red flow lines in Figure 4.9.

$T_{(f,E)}$ and $T'_{(f,E)}$ can be combined and interpolated as a vector field that denotes the quality convergence of P-frames for a given bitrate E, enabling us to compute the video quality of a P-frame encoded at E when the video quality of the previous frame is known.

If we sample $K = 100$ segments from the original video of duration $N = 20\text{s}$, the total duration of video encoding required is 400s for I-frame qualities, and 4000s for the P-frame transition properties. This enables us to compute the SSIM of the 40000 encoding schedules for each starting point in the video discussed in § 4.5.3 without requiring any additional encoding. If the optimization aggregates across 100 different starting points in the video, this technique provides more than a three orders of magnitude speed up.

To measure the accuracy of our algorithm, we generated video encoding schedules with random changes in the video bitrate and with random I-frame insertions. Each video is encoded with 100 different random schedules. A zoomed in view of one such random schedule is shown in Figure 4.11. The lower subplot shows the bitrate schedule and the I-frame insertions, while

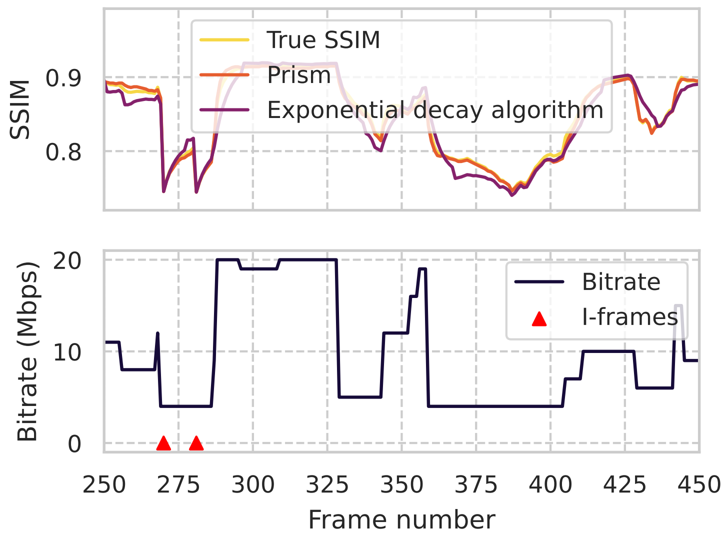


Figure 4.11: Prism's SSIM prediction for a particular encoding schedule.

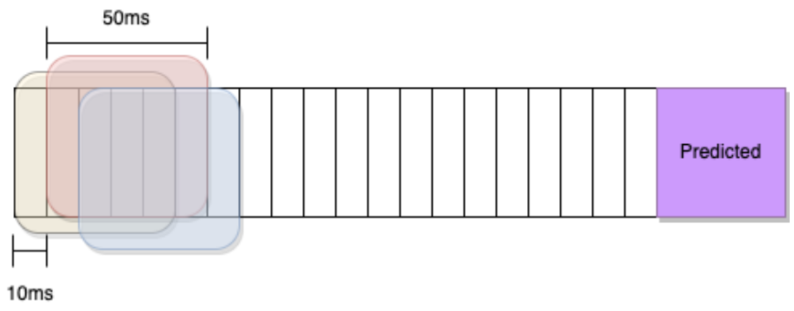


Figure 4.12: Prism's loss prediction input window and output window.

the true SSIM and the predicted SSIM are shown in the top subplot . We also compared the results of our algorithm discussed in Section 4.5.3 to a simpler algorithm where the SSIM of the video converges to the steady state P-frame SSIM using exponential decay (we choose the decay parameter that minimizes the total absolute error for each encoding schedule). The prediction error CDFs for three videos (best, worst and one in-between) are shown in Figure 4.10, along with the overall prediction error CDF across all 13 videos (§ 4.6). Our algorithm for modeling the video SSIM is very accurate, with 90% of the predictions being within 1% of the true SSIM. The naïve approach using exponential decay fails to capture the SSIM variations across frames during transition periods.

4.5.4 Loss Prediction

Prism's design enables the use of aggressive loss prediction which in turn reduces video frame delay, since the decoder can utilize the higher quality P-frame stream if the loss prediction was a false trigger. We designed a simple convolutional neural network that uses network statistics

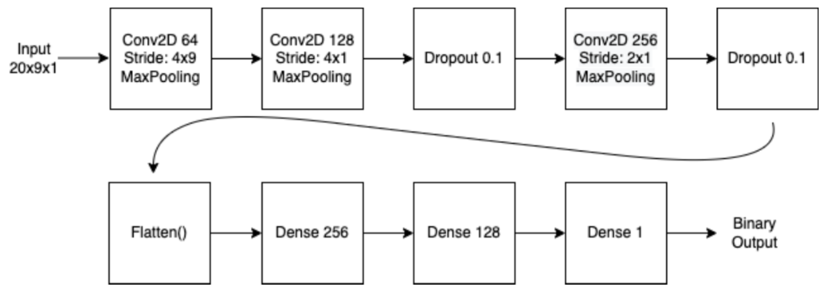

Input 20x9x1
Conv2D 64 Stride: 4x9 MaxPooling
Conv2D 128 Stride: 4x1 MaxPooling
Dropout 0.1
Conv2D 256 Stride: 2x1 MaxPooling
Dropout 0.1
Flatten()
Dense 256
Dense 128
Dense 1
Binary Output

Figure 4.13: Prism's loss prediction input window and output window.

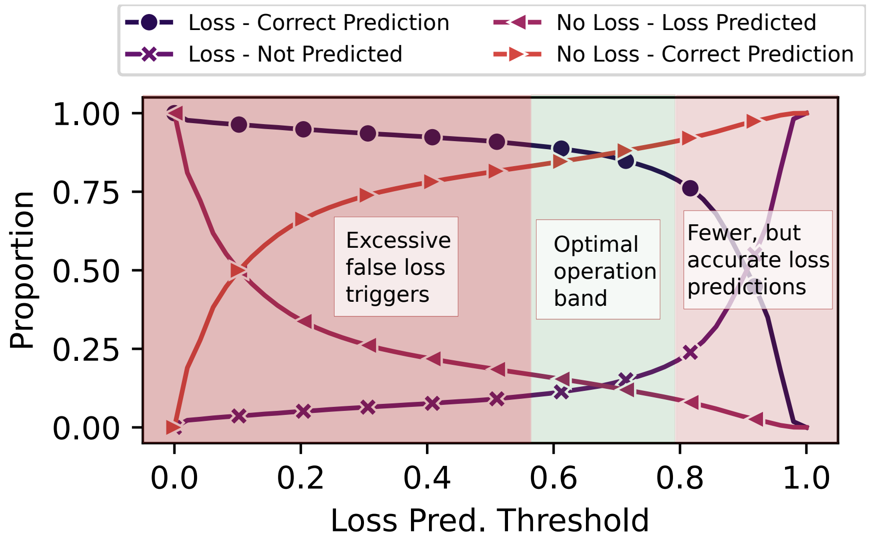


Figure 4.14: Prism neural-net loss prediction accuracy trade-off

from a past window and predicts the occurence of packet loss for a short period in the future. Prism's loss prediction neural network uses data from a 200 ms window in the past in order to predict if there will be a loss in the next 50 ms. The 200 ms window comprises of multiple overlapping 50 ms windows with a stride of 10 ms. This is shown in Figure 4.12. Each 50 ms window includes statistics like minimum RTT, maximum RTT, mean RTT, RTT variance, RTT linear fit, and the number of packets sent and lost. We also include an additional field to mark windows where no data was available, and use 0 for the other fields for such windows.

The architecture of Prism's loss predictor is shown in Figure 4.13. The neural network consists of three convolutional layers and three dense layers, and outputs a value between 0 and 1. Loss recovery can be triggered by comparing the output to a threshold, where the threshold determines the trade-off between mispredicting real losses and falsely triggering loss recovery. The simplicity of this architecture results in low computational overhead - predicting loss for a 50 ms window takes less than 1 ms on an Intel Core i7 CPU on a single thread. We trained this model on 9 days of M-Labs NDT data [89], and tested the packet loss prediction performance for a tenth day.

The neural network outputs a value in the range 0-1, where a value closer to 1 indicates that loss is very likely, and a value close to 0 indicates that loss is unlikely. This design enables Prism to choose a probability threshold for triggering loss recovery, which determines the trade-off between loss prediction accuracy and false triggering of loss recovery. This trade-off between accuracy and false loss recovery triggers is shown in Figure 4.14. While a low threshold for loss prediction is able to identify many more packet loss events, thus enabling faster loss detection and recovery for lower delay and smoother video, it also results in a significant number of false packet loss triggers - Prism is able to adapt it's bandwidth allocation to the accuracy of loss prediction since it includes a tuning parameter w (§ 4.5.2) that balances the quality between real losses and false loss triggers.

4.5.4.1 Training details

We use data from the M-Labs [89] project in order to train Prism's loss predictor. We use network traces from a period of 9 days to train the network, and test the performance on data from the 10th day. We filtered the data to only include traces that contained at least 500 windows and had a maximum RTT less than 200 ms.

Naturally, the training data was heavily skewed towards examples with no loss - less than 20% of files analyzed contained any loss at all, and less than 3% of training examples contained loss events. In order to avoid training bias, we ensured that each training batch had an equal number of loss and no loss examples by oversampling the windows with packet loss.

4.5.4.2 Discussion

Note that since Prism is predicting loss over a 50 ms interval, it forces loss recovery to last for at least 50 ms each time it is triggered - reducing the size of the prediction window may improve video quality when losses are falsely triggered, but we believe it would be a challenging task to train a model that accurately predicts packet losses over smaller windows. We leave this exploration to future work.

4.6 Evaluation

Prism's evaluation has two broad themes: evaluation of the optimization framework, and end-to-end evaluation of the system.

Optimization Framework. Prism uses pre-computed bandwidth allocation tables for different video types, since optimizing the video quality in real-time is not feasible (§ 4.5.2). We evaluate Prism's optimization strategy using the CGVDS [93] dataset, which contains a diverse set of 13 video game captures that have very different video compression properties. These videos are described in Section 4.7. This section of the evaluation provides a better understanding of Prism's efficacy in achieving better quality compared to recovery using I-frames. In § 4.7.1.1, we evaluate the impact of overall bandwidth availability on Prism's performance. In §§ 4.7.1.2 and 4.7.1.3, we quantify the impact of using a single decision table for an entire scene, as opposed to the "ideal" bandwidth split for a particular location in a given video, and quantify the impact of loss duration on video quality.

End-to-End Evaluation. In the second part, we evaluate Prism's performance using two different approaches: (1) we evaluate Prism's implementation that runs in real-time and uses the generated decision tables for allocating bandwidth across the two streams, and (2) we evaluate Prism using a discrete event simulator that uses real-world network traces from M-Labs [89] and Prism's neural-network based loss prediction.

We compare Prism's end-to-end performance to the two key video data loss recovery techniques: IDR-frames, and packet retransmissions (§ 4.4.2). For comparing video quality, we use SQI-SSIM [98], a metric based on SSIM that imposes an exponential decay penalty when video frames are delayed (accounting for the length of video stalls), and penalizes the quality for some time after each stall event (accounting for frequency of stalls). We also compare the end-to-end frame delay (including transmission delay, propagation delay and decoding delay) for frames that end up getting displayed on the screen. SQI-SSIM and the frame delay together provide a holistic view of the QoE for low-latency streaming applications.

4.7 Video Dataset

In our evaluation of Prism, we use videos from CGVDS [93]. This dataset includes a variety of gameplay footage with different gameplay and artistic styles. Screenshots from these videos are shown in Supp. Figure 4.15. In this section, we give a brief description of each video game, and discuss key aspects of each video game that affects it's video compression properties.

1. **Bejeweled 3**: A tile matching game, with mostly static content across frames. The motion is localized (object movements) with the background not moving at all, apart from some light animations.
2. **Black Desert**: A 3rd person online multiplayer role-playing game (RPG). The video sample used is a fast-paced scene inside a forest with fine textures like grass and gravel, and uses a lot of light-streak weapons animations. In addition, the textures are dithered, which means that the codec has to encode the dithering noise across frames in order to preserve the quality of the raw footage.
3. **Dauntless**: A similar setting to Black Desert, but with simplified textures. For example, the grass blades are thick with smooth color gradients, instead of finely textured grass. Weapon animations are simpler compared to Black Desert.
4. **FF XV**: Final Fantasy XV is another RPG and is very similar to Black Desert, with fine-textured elements like grass and gravel, and fast paced action and weapons animations.
5. **Fortnite**: A third-person shooter that is stylistically similar to Dauntless, with simple textures and color gradients instead of fine textures and patterns.
6. **GTA V**: Grand Theft Auto V is a third person action-adventure game. The scene we evaluate has hard-to-compress features like fine textures (roads and grass), but also includes flatter textures like sky, which are easier to compress.
7. **LoL-TF**: League of Legends Team Fight is a special mode where players battle in a static arena, with the camera fixed in a top-down view. Compared to Bejeweled 3, the static part of the scene has more textures, and thus, intra-compression is harder.

(a) Bejeweled 3

(b) Black Desert

(c) Dauntless

(d) FF XV

(e) Fortnite

(f) GTA V

(g) LoL Teamfight

(h) Maple Story

(i) Minecraft

(j) Overwatch

(k) Apex Legends

(l) Rayman Legends

(m) Tekken

Figure 4.15: Screenshots from the raw videos.

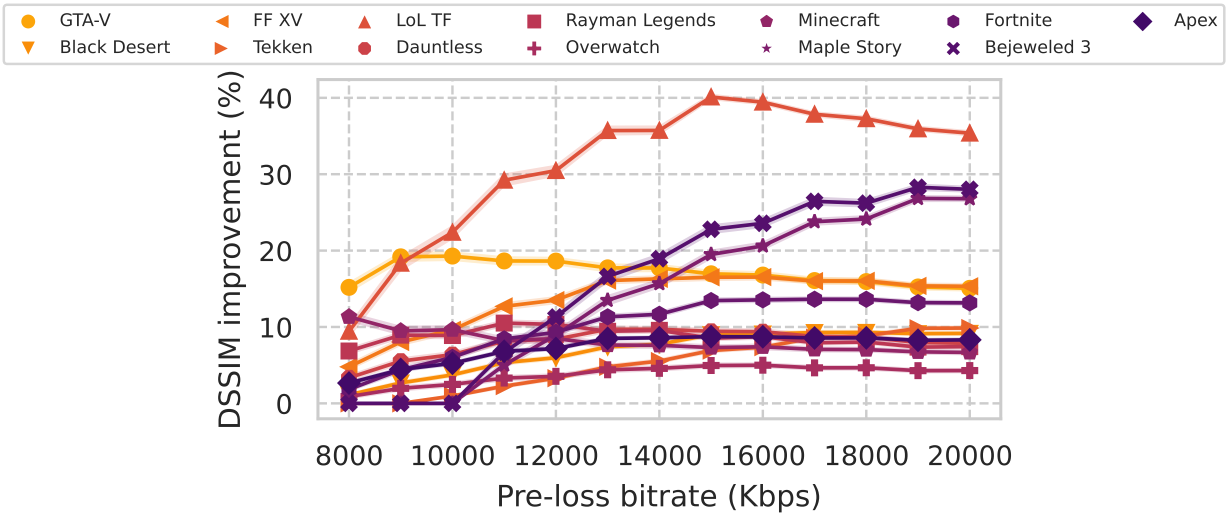
8. **Maple Story**: A top-down scrolling game, where the entire background scrolls, while the main player is always at the center of the screen. This style of video is highly amenable to motion compensation, since most of the scene can be approximated well using motion vectors.
9. **Minecraft**: A sandbox building game, with pixellated textures. The pixellated textures make motion compensation challenging, since the block size used in video compression is typically larger than the pixel textures in the game. Since the game is 3D, movement results in an affine transform to the texture pixels, which means the edges are not faithfully approximated using just motion compensation. On the other hand, the area inside the texture pixels are flat colors, and the scene we evaluate is a night-time scene, with a large portion of the video being occupied by a dark sky, which results in a higher baseline video quality for a given bitrate due to better intra compression.
10. **Overwatch**: A first-person shooter with simplified textures and large areas with flat colors/mild color gradients. This video style benefits significantly from intra-compression, since each frame has a lot of redundancy.
11. **Apex Legends**: A first-person shooter that has a combination of fine textures (on the ground) and flat regions (like walls and the sky). The footage we used in our evaluation has antialiasing disabled, which increases the complexity of the frame and results in temporal aliasing noise during movement, making motion compensation less effective.
12. **Rayman Legends**: A side-scrolling 2D platformer. The scene we evaluate has smooth, simple textures, which makes it easy to encode, and the scrolling nature makes motion compensation very effective.
13. **Tekken**: An arena combat game. Fine textures in the arena environment make intra-compression less effective. The camera movement primarily involves lateral panning movements (no rotation or 3D movement), which makes motion compensation more effective even though there are fine textures in the environment.

4.7.1 Optimization Framework

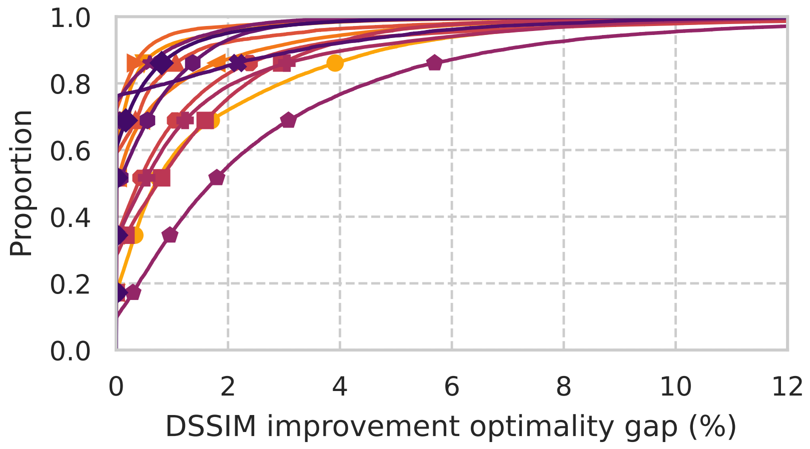
In our plots, we compare structural dissimilarity (DSSIM), defined as $\frac{1-SSIM}{2}$, which quantifies the amount of distortion from the reference image. For consistency, we use decision tables generated using $w = 0.5$ throughout (equal weights for real and spurious loss).

4.7.1.1 Bandwidth Sensitivity

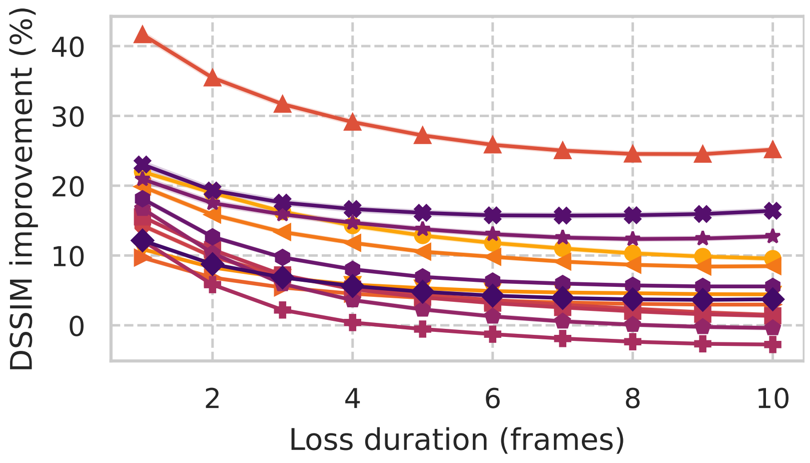
Video quality gains achieved by Prism depend on the overall bandwidth availability. Transmitting two separate streams is inefficient when the available bandwidth is low, since a large portion of this bandwidth would be taken up by redundant data shared across the frames. On the other hand, when a lot of bandwidth is available, the impact of I-frames on video quality is insignificant. Figure 4.16a shows the improvement (reduction) in DSSIM vs. using I-frames, for different values of pre-loss bandwidth (shown on the X-axis). The data points on the Y-axis aggregate the improvements across all the values of loss bandwidth that are greater than half of the pre-loss bandwidth.



(a) Impact of network bandwidth on Prism's performance.



(b) DSSIM gap when optimizing for an entire scene vs. a specific point in the video.



(c) Impact of loss duration on DSSIM improvement.

Figure 4.16: Performance of Prism's bandwidth split optimization algorithm.

Prism's gains are higher for videos like "LoL TF", where P-frames are particularly efficient at encoding the video data. On the other hand, games like "Minecraft" and "Tekken" have smaller gains since intricate textures and complex motion reduce the efficiency of P-frames (see appendix). Further, Prism's gains are lower at very high bitrates, since I-frames have sufficient video quality.

4.7.1.2 Scene Stability

Prism's optimization pipeline generates a single decision table for an entire scene by aggregating the objective function across multiple segments extracted from the sample scene. The underlying assumption behind this approximation is that while the SSIM of individual frames can vary, the transition properties and the relative quality between I-frames and P-frames are stable across a particular scene type. To verify this assumption, we compute the CDF of the percentage difference in the objective function when using the decision table, versus the optimal bandwidth split that is optimized for a specific video section. We generate data points using various pre-loss and loss bitrates, and multiple loss durations.

The results (shown in Figure 4.16b) show that the performance when using a single decision table computed per-video is almost as good as using the optimal split for a specific segment. This approximation enables Prism to make bandwidth allocation decisions in real-time by simply performing a table lookup.

In Figure 4.17, we show the difference in the video quality improvement achieved by Prism's per-video optimization strategy and an alternate strategy that generates a single bandwidth allocation table across all of the videos. The data points in the figure are aggregated across a range of bitrates and loss durations. When using a single bandwidth allocation table, the improvement in video quality can be up to 15 percent lower at the 80th percentile. In addition, the distributions for most videos have a heavy tail, indicating that it can be much worse than optimizing the bandwidth allocation per-video in some cases.

Note that since our algorithm is greedy and not truly optimal, the single bandwidth allocation table gets lucky sometimes and beats the per-video optimized allocation table - hence, there are some data points on the negative side of the X-axis.

4.7.1.3 Loss duration sensitivity

Prism's benefits are realized due to two fundamental video coding properties: P-frames are more efficient at encoding video data, and the quality of a P-frame stream exhibits a gradual transition when the bitrate is changed. The benefits of Prism can be realized even for very long periods of packet loss if the P-frames are extremely efficient (eg. "LoL TF"). When P-frames are not as efficient, Prism still demonstrates quality gains for shorter loss durations (e.g. "Overwatch"). This is shown in Figure 4.16c, which plots the improvement in video quality over simply using I-frames, as a function of the loss duration. Prism's optimization algorithm accounts for scenarios like "Overwatch", switching to sending I-frames only after running in split-stream mode for a few frames. For extended loss durations, this approach is only slightly worse than transmitting I-frames right from the beginning, and stems from Prism's greedy approach for allocating bandwidth across the two streams.

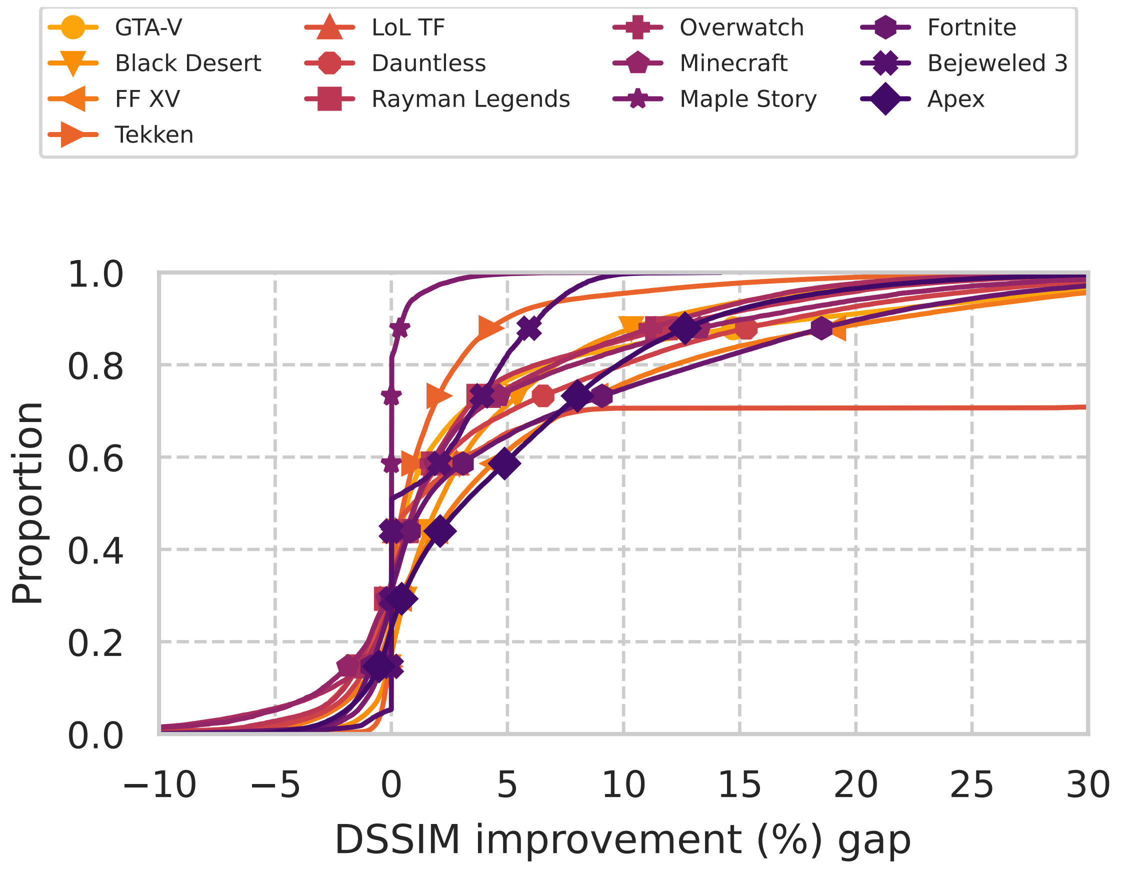


Figure 4.17: Comparison of the DSSIM improvement achieved by a video-specific bandwidth allocation table and a bandwidth allocation table optimized for all the videos as a whole. The X-axis is the difference in the %-improvement in the DSSIM between the two methods, and the Y-axis plots the cumulative distribution of this gap across various bitrates and loss durations.

These results also suggest that game developers designing games for cloud platforms can improve streaming performance by making encoder-friendly choices for artistic styles and gameplay mechanics.

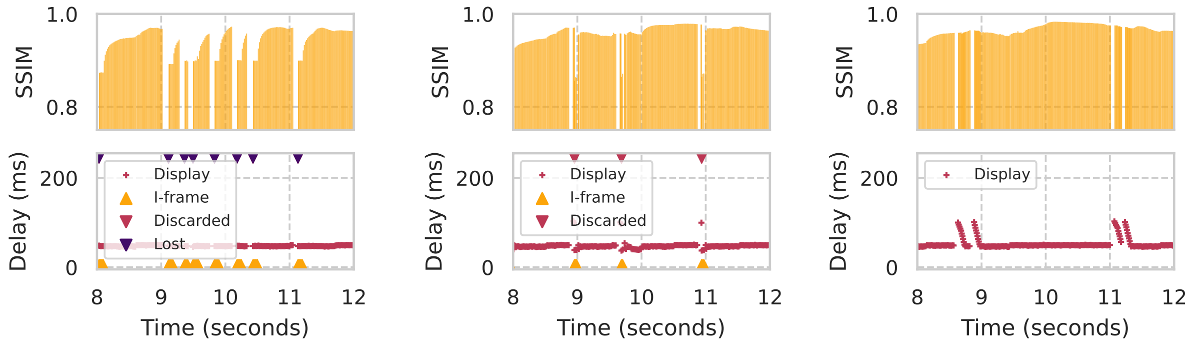
4.7.2 End-to-end Evaluation

We evaluated the end-to-end performance of Prism in two different ways: (1) A real-time implementation of Prism that streams video over an emulated lossy link (§ 4.7.2.1), and (2) a discrete event simulator that evaluates Prism on real-world M-Labs [89] traces, and integrates Prism's neural network based loss prediction mechanism (§ 4.7.3.1).

4.7.2.1 Real-time Streaming Experiments

We implemented a real-time video streaming testbed that enables the comparison of various loss recovery mechanisms and allows us to compare Prism's performance to baselines like I-frame based recovery and packet level retransmissions. This testbed uses the NvEnc [99] encoder and the H.264 [29] video codec[2], and implements a video streaming API that provides direct control over each frame's encoding configuration. For simplicity, we assume that the available bandwidth under normal operation is 20 Mbps, droping down to 15 Mbps during periods of loss. Note that Prism can work with any congestion control algorithm, Prism just needs to know the instantaneous available bandwidth and needs a trigger for loss recovery. In these experiments, we use the Mahimahi [3] network emulator to emulate a 25 Mbps link with a 40 ms RTT. We evaluate two types of packet loss - (1) random packet loss, and (2) loss caused by queue-building flows (e.g. loading a web page with `wget` that uses TCP-Cubic).

4.7.2.2 Emulated Network with Random Loss



(a) IDR-frames on packet loss. (b) Prism. (c) Packet retransmissions.

Figure 4.18: Timeseries comparing Prism and the two baselines (IDR-frames, retransmissions) in the presence of random loss.

[2]Any other codec that is supported by FFMPEG [100] and NvEnc can be used.

In Figure 4.18, we show timelines comparing I-frame based recovery, packet retransmissions, and Prism under random loss. Each loss event results in a large, sustained drop in video quality when using I-frames. While the quality of Prism's I-frames are slightly lower, the video quality recovers rapidly after the loss event. In the case of packet retransmissions, the delay spike that occurs after a loss event takes a significant amount of time to recover since the decoder needs to catch up to the latest frame after receiving the retransmissions for the lost packets. Secondary losses can compound this delay, resulting in very low QoE.

4.7.3 Web Page Load Timeseries

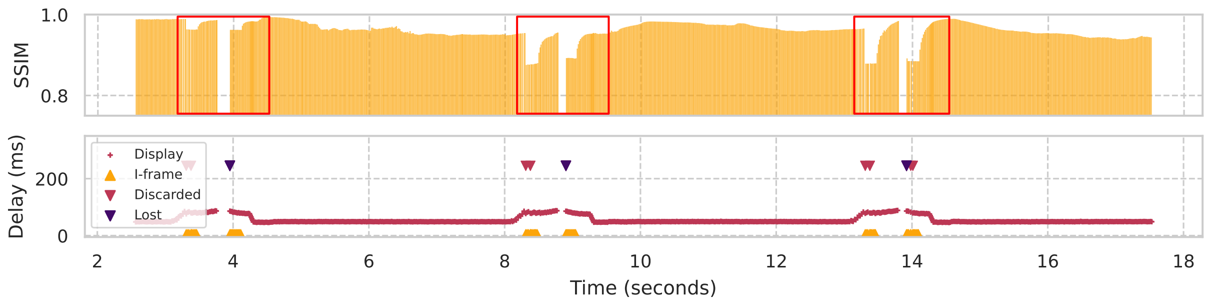


Figure 4.19: I-frame timeseries for loss caused by a competing flow performing a web page load.

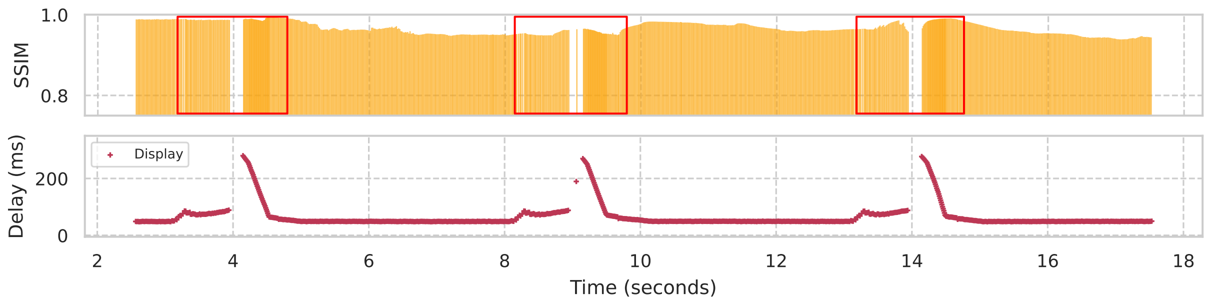


Figure 4.20: P-frame timeseries for loss caused by a competing flow performing a web page load.

We evaluated Prism's real-time implementation over an emulated bottleneck using MahiMahi [3], where we induce packet losses by introducing a competing Cubic flow that downloads a web page. In this experiment, we use a simple RTT threshold filter as a dummy loss prediction technique in order to clearly show how loss prediction impacts the video quality and delay.

The time series depicting the performance of I-frames (SSIM and frame delay) is shown in Supp. Figure 4.19. During each run, the web page is downloaded 3 times, and each page load lasts for a duration of about 1.5 seconds, which can be seen by the regions with increased delay. For each event, there is one false loss trigger at the beginning (due to the increased RTT as Cubic starts filling the buffer), and one true loss event. When I-frames are used for recovery, there is

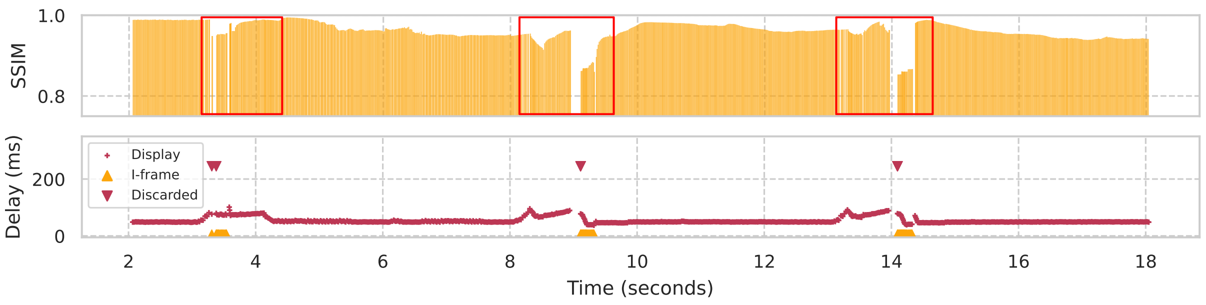


Figure 4.21: Prism timeseries for loss caused by a competing flow performing a web page load.

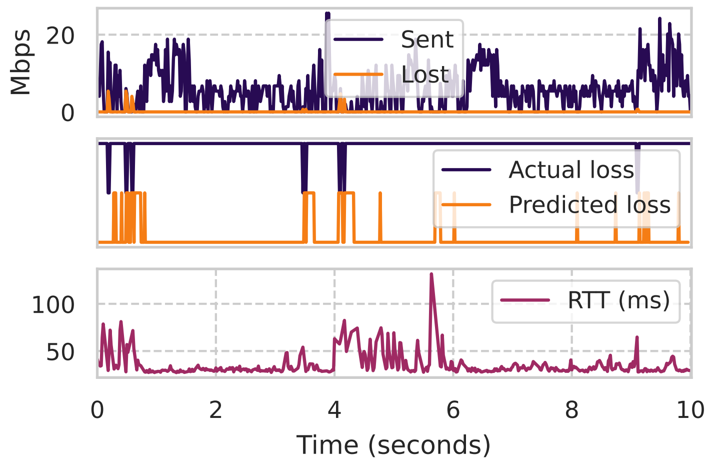


Figure 4.22: Prism loss prediction (MLabs trace).

a significant drop in video quality during both, the false loss trigger and the actual loss event. In addition, the video quality takes a long time to recover back to the steady state quality. The benefit of using I-frames is that the end-to-end frame delay is low, since the receiver is able to decode and display the latest frame always.

Supp. Figure 4.20 shows the video quality and delay when using packet retransmissions for recovering from packet loss. In this case, there is no drop in the video quality, but there is a large video stutter until the missing packet is retransmitted. In addition, the frame delay remains high for around half a second after the loss event while the backlogged P-frames are decoded by the receiver.

Supp. Figure 4.21 shows the timeline for Prism. The three key benefits of Prism are highlighted here - (1) The drop in video quality when a loss is falsely triggered is minimal, (2) Once loss recovery is complete, the video quality recovers to the steady state P-frame quality quickly, and (3) the delay remains low throughout the recovery process.

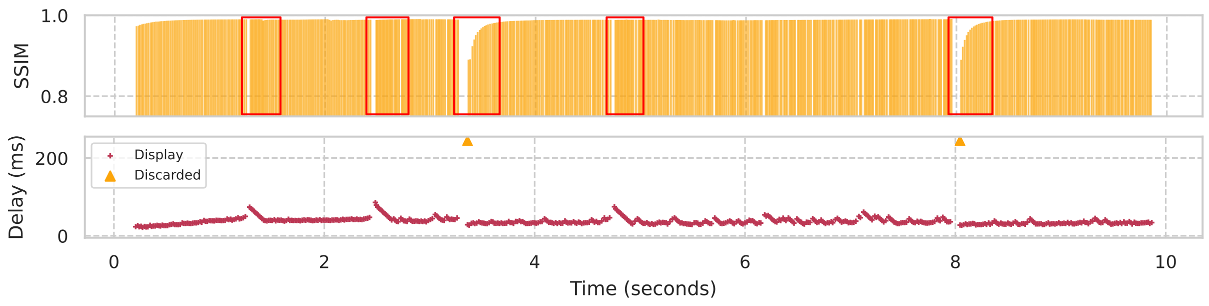


Figure 4.23: I-frame timeseries without loss prediction for an MLabs trace.

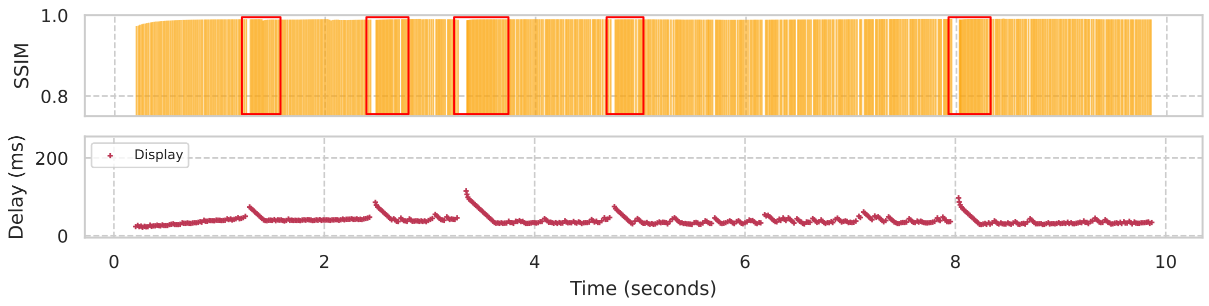


Figure 4.24: P-frame timeseries for an MLabs trace. Note the large spikes in delay when losses occur.

4.7.3.1 Trace-based Discrete Event Simulation

In order to evaluate Prism's performance under real-world network conditions, we designed a discrete event simulator that replicates Prism's protocol and integrates Prism's neural network based loss predictor. We evaluate Prism on ≈ 50 different network traces from M-Labs. These traces exhibit multiple loss events, and delay and bandwidth fluctuations, and were not used in the training of the loss predictor.

The bandwidth and delay of a sample trace is shown in Figure 4.22, along with the output of Prism's loss predictor. Aggressive loss prediction enables Prism to initiate recovery earlier, which reduces video stutter and delay. While Prism's loss predictor has quite a few false positives, recall that Prism's design significantly reduces the impact of falsely triggering loss recovery (§ 4.4.3).

Figures 4.23, 4.24 and 4.25 show the timelines for I-frames, retransmissions and Prism respectively for one trace from the M-Labs dataset. Prism avoids the extended drop in video quality that occurs when using I-frames for recovery, while simultaneously achieving lower delay than packet retransmission based recovery.

Figures 4.26 and 4.27 show the timelines for Prism and I-frames respectively when using loss prediction. Loss prediction reduces the end-to-end frame delay in both cases. In the case of I-frames, the video quality drops significantly for an extended period of time when loss recovery is triggered. Prism avoids this drop in some cases when the loss recovery is falsely triggered (first two events), and thus achieves overall higher video quality than I-frame based recovery.

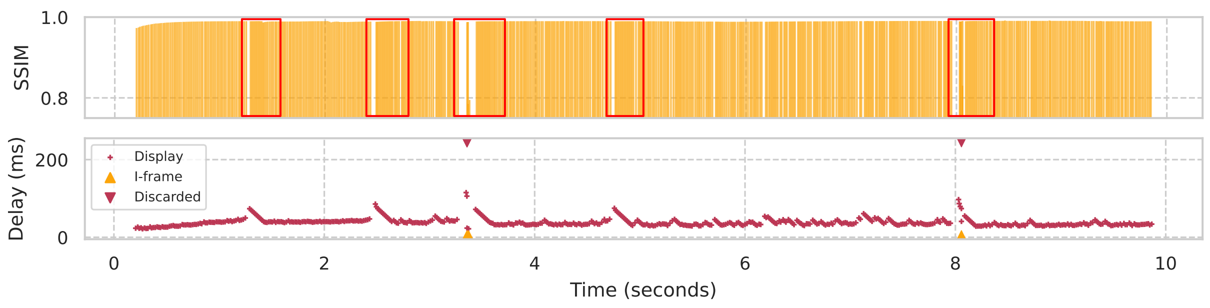


Figure 4.25: Prism timeseries without loss prediction for an MLabs trace. The video quality quickly recovers after a loss event.

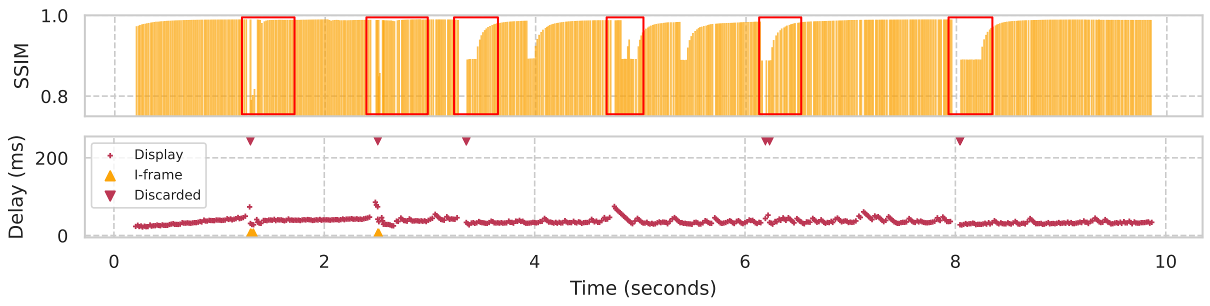


Figure 4.26: Prism timeseries with loss prediction for an MLabs trace. False predictions result in some additional events where the video quality drops, but the delay is much lower and stutters are reduced due to loss prediction.

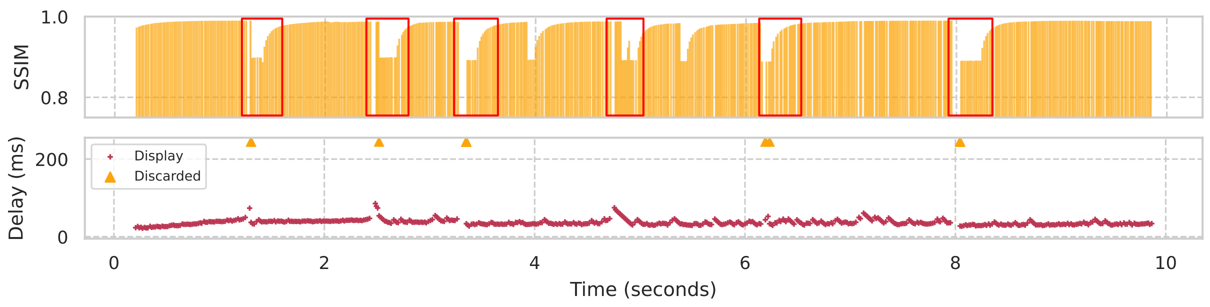


Figure 4.27: I-frame timeseries with loss prediction for an MLabs trace. Note the frequent drops in video quality for events where loss recovery is triggered without an actuall loss.

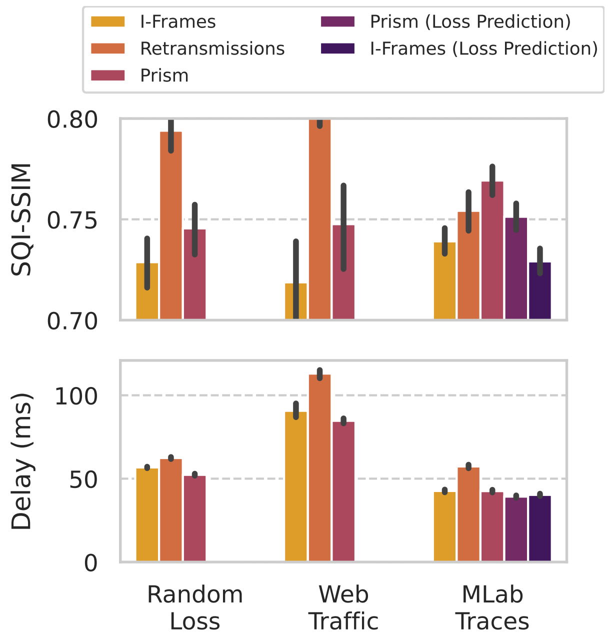


Figure 4.28: Prism's performance across various network types.

4.7.3.2 Results Summary

In Figure 4.28, we compare the performance of Prism over the two emulated network conditions mentioned earlier (random loss, web-page load cross-traffic), and the network traces from M-Labs. The results are aggregated across 5 videos ("LoL TF", "Apex", "GTA-V", "Overwatch", and "FF XV"), and multiple runs for random packet loss and web-page load cross traffic (with staggered page load timing in each run), and across all the M-Labs traces. In our analysis, we use a simple clustering algorithm (see appendix) to extract events of interest (e.g. I-frames, large stutter, frame loss) and compare the results from these periods (as opposed to the quality of an entire run, which may only have a few loss events) to highlight the performance of our system. The filter identifies frames in the video where there were large delay spikes, missing frames, I-frames and retransmissions, and expands these events into windows of interest that include additional frames after the event in order to account for the transition properties of the video quality of P-frames. For example, in Figure 4.23, the red boxes highlight the events automatically detected by our filtering algorithm. The video quality and delay from these windows across various networking environments are aggregated in the results shown in Figure 4.28.

Across all the three scenarios, Prism achieves higher SQI-SSIM than a pure I-frame based approach for recovering from video data loss, since Prism's P-frame stream enables it to achieve higher video quality after recovery from loss (§ 4.7a). Prism's SQI-SSIM is also higher than re-transmission based recovery on M-Labs traces - some of these traces have significant bandwidth and delay variations, and Prism's recovery stream is able to meet more playback deadlines due to it's lower bitrate and since it is composed entirely of I-frames. In all cases, Prism achieves lower delay than I-frame based recovery and packet retransmissions. While it is clear why Prism's delay is lower than packet retransmissions (avoids delays from the decoder needing to catch up), the reasoning behind Prism's lower delay compared to I-frames is subtle - since the bandwidth of

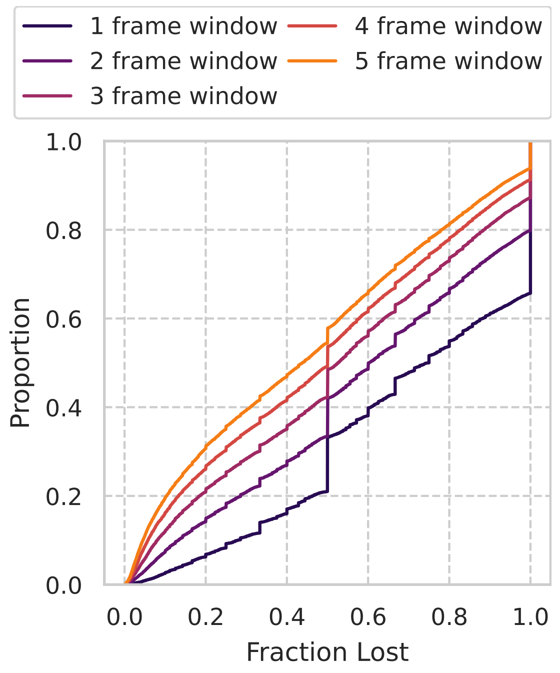


Figure 4.29: Loss rate distribution for various timescales.

Prism's I-frames are lower, they have lower encoding, decoding, transmission and propagation delays.

This figure also shows the trade-offs of using loss prediction - Prism with loss prediction achieves lower end-to-end delay at the cost of lower video quality, but the video quality is much higher than I-frame based recovery with loss prediction since Prism's receiver simply ignores the recovery stream and uses the higher quality P-frame stream during a false loss recovery trigger.

4.8 Related Work

A key body of related work is the field of error correcting codes (forward error correction). These techniques aim to prevent video data loss by transmitting redundant packets in addition to the base video data. A given FEC scheme makes the assumption that over a certain window of time, the loss rate is lower than a predetermined threshold. When packet loss exceeds this threshold, video data loss occurs and the only solution is a reactive loss recovery approach like Prism, I-frames or packet retransmissions. For handling 50% packet loss, the FEC code rate must be at least double of the video bitrate, and needs to be even higher for higher loss rates.

To understand the impact of Internet loss patterns on FEC designs, we plot the distribution of the fraction of data lost for different windows of time (Figure 4.29). Even over durations as long as 5 frames (~80ms for 60FPS video), the 80th percentile loss rate is over 80%, which makes FEC impractical for low latency streaming applications [101]. FEC schemes that operate over larger windows add significant latency for recovery when losses occur, and also add a significant amount of computational overhead. Thus, systems that use FEC still require video data loss recovery mechanisms to handle unrecoverable packet loss. Prism's design is complementary to

FEC - Prism simply needs to know the effective available bandwidth after accounting for the overhead of using FEC. No other changes are required to Prism's core algorithm.

Salsify [14] proposes a joint video codec and network transport design that aims to avoid inducing losses by matching the size of the compressed frames to the estimated network capacity. While this approach is suitable for low bandwidth environments where the video traffic is itself likely to cause queuing and packet losses, Prism targets a different set of network conditions - cloud streaming applications require much higher bandwidth, and losses are often caused by external factors like queue-building cross traffic or noisy wireless links.

In addition, a key contribution of Salsify was a functional video encoder (Alfalfa). Alfalfa supports state saving and restoration in the video encoder. Salsify uses this stateful video encoder interface to encode each frame at two target quantization parameters. Salsify then chooses the encoded version of the frame that can be safely sent under the current network conditions based on the frame size, in an attempt to avoid inducing packet loss altogether. In our experience with using NvEnc, NVIDIA's GPU-based hardware encoder, we found the rate control algorithm to be very accurate with the low latency preset.

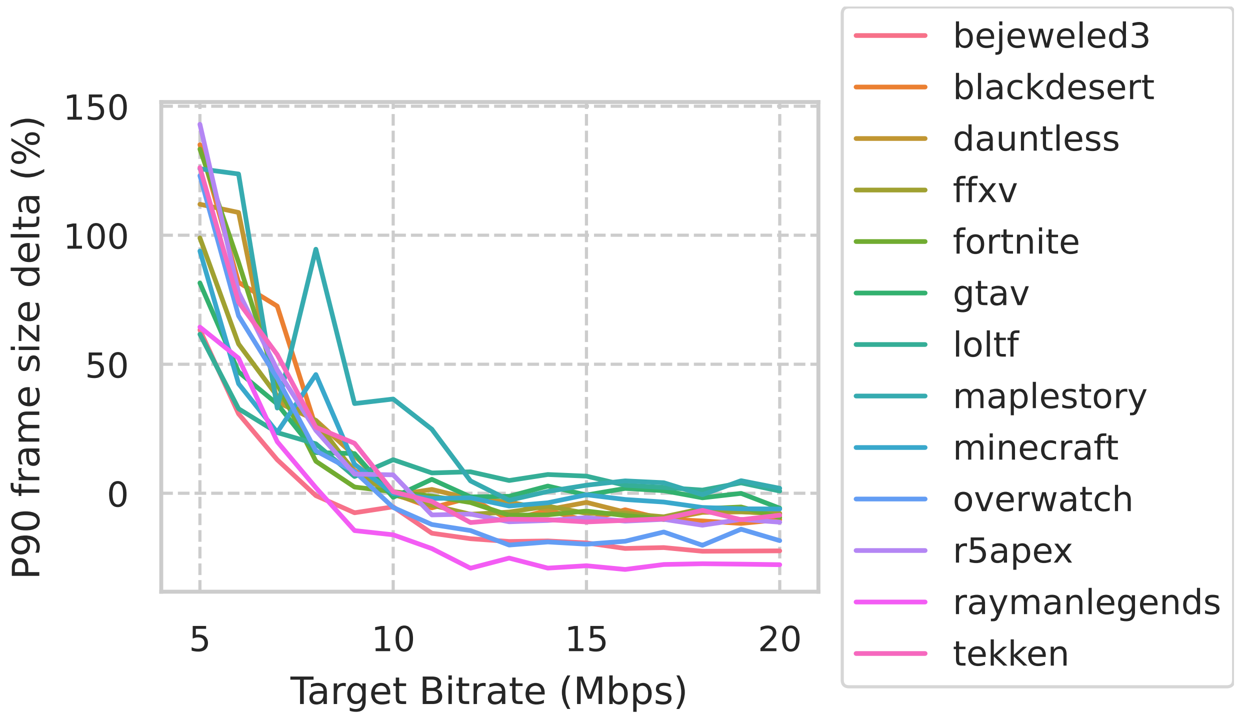
Figures 4.30a and 4.30b compare the rate control performance of Alfalfa (the codec used in Salsify) and the NvEnc codec. We encoded each video 5 times with a random target bitrate for each frame, and compared the requested target size and the size of the encoded frame. For Alfalfa, we used the REALTIME_QUALITY preset with two pass encoding, and for NvEnc we used recommended options by NVIDIA for real-time streaming [102]. Alfalfa overshoots significantly for bitrates lower than 10 Mbps, with some frame-size overshoots present even at higher bitrates. On the other hand, we do not see any overshoots at the 90th percentile for NvEnc. Thus, rate control algorithms of hardware-based video codecs today work extremely well, and there is no need to use a slower software-based codec like Alfalfa.

4.9 Conclusion and Key Takeaways

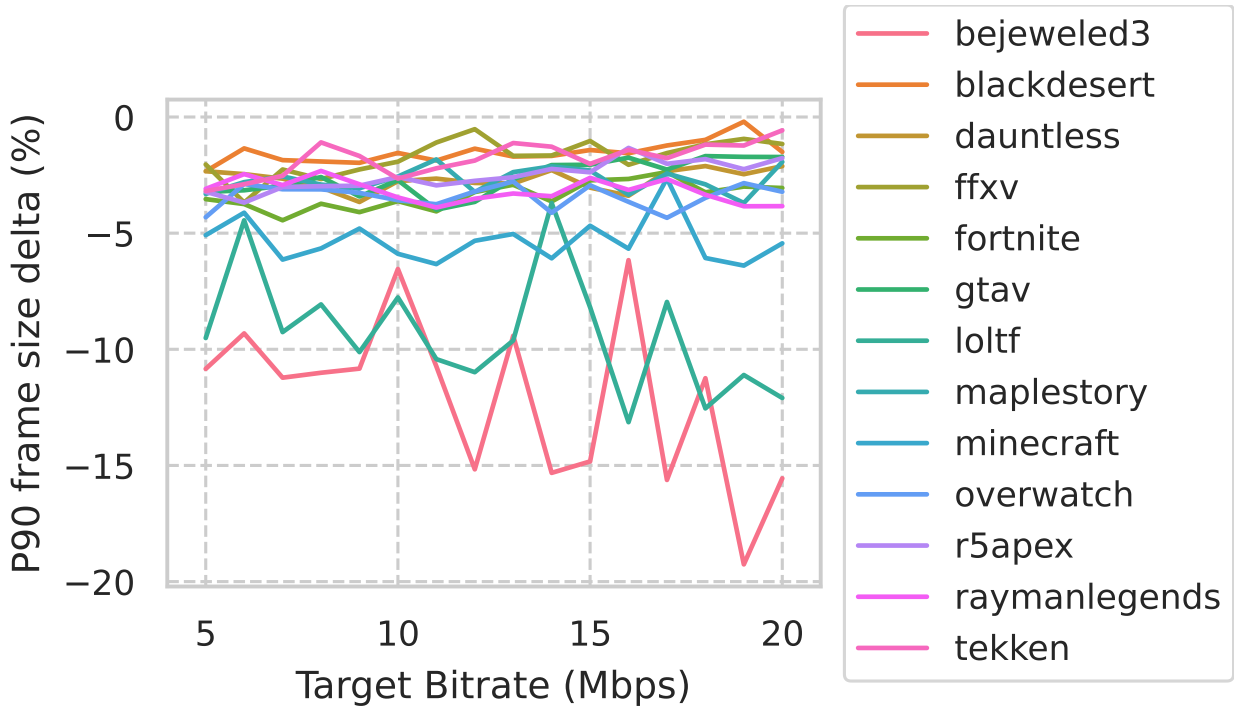
In this paper, we presented the design of Prism, a novel approach for recovering from transient packet loss for ultra-low latency applications like cloud streaming. Prism uses a hybrid approach, splitting the available bandwidth between an I-frame stream and a P-frame stream to balance the video quality during loss recovery and video quality post-recovery while having zero overhead when network conditions are stable. Prism proposes a novel algorithm to model video SSIM which significantly reduces the computational requirements for optimizing the bandwidth allocation. Prism's approach also significantly reduces the impact of spurious losses on video quality, which enables aggressive loss prediction that can further reduce the end-to-end latency, resulting in overall improved QoE for low-latency interactive streaming applications like cloud gaming and AR/VR streaming.

The key takeaways from this chapter are:

1. Emerging video streaming applications like cloud gaming have demanding video quality and frame delay requirements, and addressing these requirements is key to their success, since they aim to replace an existing solution, local rendering, which does not have to deal with network issues when displaying rendered visual contents.
2. Network bandwidth, packet loss, and video compression interact in complex ways, and



(a) Alfalfa (Salsify) video encoder rate control accuracy. Significant overshoot is present, especially at lower bitrates.



(b) NvEnc video encoder rate control accuracy. Most frames respect the target size, with very few overshoots.

Figure 4.30: Rate control accuracy comparison of Alfalfa (Salsify) and NvEnc.

designing solutions based on individual components are not sufficient (e.g. retransmissions/FEC attempt to solve the problem from a networking perspective, and IDR-frames are a solution at the video codec level).

3. A solution like Prism that takes an integrated approach at solving the issue of packet loss, taking into account the various interactions between bandwidth, loss, and video compression can significantly improve the QoE over existing, narrow solutions.

Chapter 5

ViXNN: A deep learning approach to loss resilient image and video transmission.

In recent years, there has been a significant amount of growth in the space of virtual reality applications. Early VR headsets were cumbersome, used thick wires to connect to a powerful PC and and required multiple physical tracking stations to be installed (outside-in tracking). Today, the most popular VR headset is the Oculus Quest [103]. The Oculus Quest is a stand-alone android based device with it's own compute capabilities, and inside-out tracking using multiple cameras on the head-mounted display (HMD). The biggest benefit of the Oculus Quest is that it can run VR applications without being tethered to a machine using thick wires. This design has it's downsides:

1. Since the compute on the HMD is limited, it is limited to running applications with simple graphics, and is unable to run applications with highly immersive graphics.
2. Even if the amount of compute on the HMD can be increased in future versions, power consumption can be an issue, since the HMD needs to be lightweight and comfortable.
3. In comparison to tethered headsets, VR headsets with stand-alone compute capability get obsolete much more quickly.

Since it's launch, the Oculus Quest received software updates that allows PC users to use the Oculus Quest as purely an HMD to display applications running on a PC. Initially, it was supported via a USB cable connected to the headset, and recently, wireless streaming to the HMD was enabled by a software update [104]. Streaming rendered content over a local wireless network has many benefits - desktop-class VR applications can be experienced without wires connecting the HMD to a physical machine, the HMD uses much less power since it does not need to perform any rendering, and the compute capabilities can be updated without the need to buy a completely new headset.

Unfortunately, this approach also has it's downsides. Since the rendered content would need to be compressed before it can be transmitted wirelessly to the HMD over conventional Wi-Fi, noisy wireless environments and wireless packet loss can significantly harm the QoE. Conventional video coding techniques are designed to primarily work with reliable storage and transport mechanisms.

Deep learning based techniques have made great strides in the compression of visual data.

Research has shown that neural networks can outperform conventional, state of the art, image and video compression techniques [105, 106]. Machine learning based techniques enable the optimization of the compression pipeline for specific data distributions and quality metrics, which gives them a leading edge in compression performance over conventional video codecs that are more general purpose.

Motivated by these observations, we propose ViXNN, an end-to-end neural network architecture for loss resilient video compression. ViXNN can adapt to different loss patterns and application specific requirements. ViXNN's design incorporates a novel multi-path neural network architecture which simulates data loss during the training phase which can be used to learn different schemes for loss resilience. When using conventional video codecs for streaming, packet losses and bit errors cause frame skips, delays, and stutters when frames are not decodable, and visible artifacts if the frames are decoded with error concealment techniques. On the other hand, when packet losses and bit errors affect video data compressed using ViXNN, it results in a graceful degradation of the video quality instead of visible artifacts or frame stutters.

We also show that using structured loss emulation in the narrow waist of the neural network architecture, we can train the neural network to learn highly customizable scalable video codecs.

5.1 Introduction

VR applications require extremely low end-to-end latency in order to maintain immersion and avoid motion sickness. When streaming VR applications wirelessly as compressed video, packet loss can have a significant impact on the QoE. In the context of VR applications, apart from latency, the frame rate also has a big impact on the QoE.

Traditional video codecs use intra- and inter-frame compression in order to significantly reduce the bandwidth requirement. When packet losses lead loss of compressed video data, the video frames cannot be decoded. When the video is streamed as a series of P-frames, the loss of a single frame can cause the video decode pipeline to stall until the lost data is retransmitted or an I-frame is sent. In the case of VR streaming from a computer on the local network, the latency is very low, and hence, the lost data can be retransmitted fairly quickly. Unfortunately, this would still cause atleast 1 frame to get dropped, since it would not be displayed on time. This problem exists even if P-frames are not used, and every frame is transmitted as an I-frame.

Table 5.1 shows a comparison of 5 common applications. The video coding and transmission pipeline must be flexible enough to achieve the desired trade-off between compression, loss resilience, latency, bandwidth adaptability and content specialization.

Application	**Acceptable latency**	**Sensitivity to loss**	**Bandwidth**	**Network variability**	**Content specialization**
Social live streaming	Few seconds	Moderate	Variable	Yes	Low
Gaming on demand	50 ms	Very high	High	No	Moderate
Virtual Reality	10 ms	Very high	High	No	Moderate
Video telephony	150 ms	Moderate	Variable	Yes	Moderate
Surveillance	10s of ms to a few seconds	Moderate	Low	Yes	High

Figure 5.1: A comparison of the requirements of various applications

In this paper, we propose ViXNN, an end-to-end neural network architecture for video encoding. ViXNN's design is motivated by the need for a video encoding and transmission pipeline that can deal with network losses, heterogeneous multi-cast, variable transmission bandwidth, while being able to adapt to application specific quality metrics and leverage specialized content for a high QoE. ViXNN's design incorporates a multi-path neural network design, enabling the neural network to learn a compression scheme that achieves the desired trade-off between loss resilience, compression and bandwidth adaptability. Our evaluation shows that ViXNN can learn highly tailored compression schemes that are competitive with, and often better than conventional techniques for a wide array of scenarios. Finally, we discuss the computational requirements and the performance of ViXNN for encoding and decoding videos in real time, and some important considerations for deploying ViXNN in the real world.

5.2 Background

H.264/AVC, H.265/HEVC, VP8 and VP9 are the most commonly deployed standards today. These codecs perform intra-frame and inter-frame redundancy elimination to achieve compression of video data.

5.2.1 Video coding

5.2.1.1 Intra-frame coding

In intra-frame coding, individual blocks of the frame are predicted from the values of the pixels on the edge of the block. The prediction error is quantized and entropy coded based on the bitrate and quality settings.

If the video stream is encoded as a sequence of intra-frames, the effects of a packet loss do not propagate across frames, since each frame is independently encoded. This is similar to transmitting each frame as an image (e.g. Motion JPEG). The flip side of this approach is that the corruption of a single pixel can corrupt large regions of the frame.

5.2.1.2 Inter-frame coding

Inter-frame coding leverages the temporal redundancy in video data. Inter-frame compression schemes rely on the observation that a patch in a frame can be approximated by a neighboring patch in the previous frame, since successive frames differ due to camera panning or object motion. This technique is also known as motion compensation. The generated video stream has three types of frames: I (independent), P(forward predicted) and B (bidirectional predicted). The residual error is quantized and entropy coded, just like intra-frame coding. In real-time systems, B-frames are not used, since this adds encoding and decoding latency, since future frames are required in order to encode and decode video when B-frames are used. Real-time systems use P-frames in order to achieve better compression efficiency. The downside of using P-frames is that an error in a single frame can propagate to all the other frames that depend on it. A minimal set of frames which only depend on each other is called a group of pictures (GOP).

Figure 5.2 illustrates the impact that losses can have on these codecs. Codecs like JPEG and H.264 introduce very noticeable block artifacts under high compression settings (highlighted in red in the figure). The H.264 artifacts persist across multiple frames, i.e., until an I frame

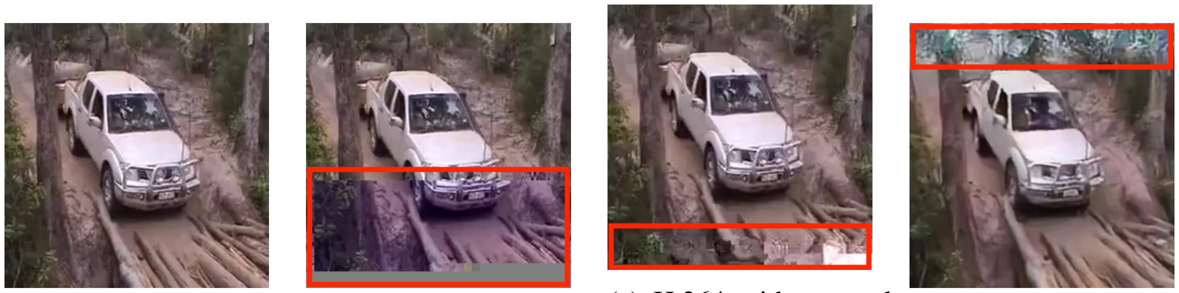

(a) Original frame (b) JPEG (c) H.264 with concealment (d) Neural network

Figure 5.2: Visual effects of packet loss for various compression techniques

refreshes that portion of the image. These artifacts result in poor performance on video quality metrics such as PSNR and SSIM.

Error Concealment. Video codecs like H.264/AVC provide some tools that help make the data stream resilient to losses. H.264/AVC provides a data partitioning scheme that separates the more important syntax elements from the less important data elements. This allows the usage of unequal error protection schemes. Motion vectors and quantization parameters are usually small in size and can be given higher protection than block coefficients. Uncorrupted syntax elements can be used for better error concealment at the decoder. Figure 5.2(c) is an example where the errors at the bottom of the frame were concealed to some extent by H.264/AVC. Data interleaving spreads the effect of lost packets throughout the frame as opposed to losing a large patch. This allows better concealment of errors because it is more likely that a corrupt pixel has a valid neighborhood. The GOP size used for encoding the video also has a profound effect on the loss resiliency of the data stream. Using only I-frames results in a more robust data stream because it stops the temporal propagation of errors, but at the cost of reduced compression efficiency.

Handling packet loss in traditional video compression techniques: If only I-frames are being used, and packet loss occurs, there are two choices -

1. **Decode the frame with error concealment**: This causes visible artifacts that appear temporarily for the duration of the frame that is affected. Such artifacts are not acceptable for immersive applications like VR.
2. **Skip the frame**: Since every frame is an I-frame, the decoding can resume when the next frame that has not been affected by packet loss arrives. This can cause stuttering, especially in the case of burst losses that last across frame boundaries, where all of the affected frames would have to be skipped.

When using P-frames, there are three choices when packet loss occurs -

1. **Decode the frame with error concealment**: In the case of P-frames, this causes visible artifacts that persist across multiple frames, until an I-frame is transmitted in order to reset the decoder state.
2. **Wait until lost packets are retransmitted**: In this case, the decoder stops decoding frames until the lost packets have been recovered. Once the lost packets are recovered, the decoder needs to catch up to the latest frame - this can cause stuttering, which can be severe if packets from multiple frames are lost.

3. **Wait for the next I-frame**: The decoder stops decoding frames until the next I-frame is received. In this case, the stutter performance is similar to skipping frames when using only I-frames for streaming the video.

5.2.2 Scalable video coding

Scalable coding is a general class of techniques which attempt to solve the problem of packet loss and multi-rate transmission. While the use of techniques like error correcting codes and reliable transport assume that the data needs to be transmitted without any loss, scalable coding takes advantage of the fact that video data is analog in nature and graceful degradation of video quality in the form of noise, reduced frame rate or reduced resolution is acceptable when losses occur. The general approach of scalable codecs involve generating multiple streams of data where each stream has partial information. These streams can be aggregated for higher quality decoding. There are two broad categories of scalable codecs: SVC (Scalable Video Coding) and MDC (Multiple Description Coding).

SVC. Scalable video coding [57] is a hierarchical layered approach to video coding. The video is encoded into a base stream and multiple hierarchical enhancement streams. The base stream is required while the enhancement streams are optional. When video is encoded using SVC, the loss of the base layer makes the data undecodable, whereas if the enhancement layers are lost, the video quality degrades gracefully. This enables the use of differential error protection for the base layer and the enhancement layers. For example, the application of high levels of FEC to the base layer would reduce the chance of data loss in the base layer, and the video will be decodable even if the enhancement layers are lost.

MDC. Multiple description coding [107] is a non-hierarchical approach. Every stream can be decoded individually and higher quality decoding can be achieved by using a larger number of streams. This implies that every stream must have some basic information which is common across all streams. This reduces the efficiency of MDC coding, but improves loss resiliency. This is because unlike SVC, any subset of streams can be lost, and the video would still be decodable at a lower quality. Thus, there is no need to have additional loss protection mechanisms like differential error protection. Unless all of the streams are lost, the video decoding can proceed, resulting in smooth, stutter free video.

Both SVC and MDC techniques solve the problem of multi-rate video transmission. The source can perform one-shot encoding and send as many streams as possible based on the current uplink speed.

MDC is better than SVC at handling packet losses at the cost of worse coding efficiency. This is a result of the stream priorities in SVC. If the base stream is lost, the entire video stream is lost. SVC can still be useful with prioritized drops and unequal error protection of streams. This idea is used in SoftCast [20], which is a cross layer solution for wireless video transmission. The more important data is scaled up compared to the less important data and then transmitted over raw OFDM. This provides greater protection to the base data stream from bit errors due to noise. The results in graceful degradation when the wireless signal-to-noise ratio drops.

5.3 Using Neural Networks

Artificial neural networks have recently gained considerable popularity for a variety of learning and prediction tasks ranging from natural language processing [108] and computer vision, to finding approximate solutions to the traveling salesman problem [109]. This can partially be attributed to the lower training and inference times enabled by the development of general purpose GPUs and CUDA. Even mobile devices today are now capable of running complex neural networks [110]. The second reason for the skyrocketing interest in the use of neural networks is the recent advancements in deep learning techniques and architectures, which have been shown to surpass the performance of many hand crafted computer vision algorithms.

5.3.1 Compression using neural networks

Data compression using neural networks is a widely researched topic, that has experienced renewed interest among researchers in recent times. Neural networks can learn patterns in the data and encode them as high level representations, with the potential of achieving higher compression ratios than conventional compression techniques. In [105], the authors propose a neural network architecture that can learn a compression scheme for generic images. Their architecture can outperform JPEG at higher compression ratios. The authors use an architecture based on recurrent neural networks to achieve variable bit rate compression. They use an iterative process to generate encoded bits, where the input to each iteration is the residual error of the output of the previous iteration. In [106], the authors show that real-time image compression can be performed using neural networks. They use an adversarial model to make the reconstructed images more visually pleasing. The authors also show that using neural networks enables the learning of context adaptive compression schemes, which can outperform traditional approaches that are meant to work on generic datasets. In [111], the authors explore the use of generative adversarial networks (GANs) for image and video compression. GANs have recently gained tremendous popularity owing to their ability to generate visually meaningful outputs. To extend their approach to video, the authors show that they can drop intermediate frames altogether, and predict these frames by interpolating the compressed representations.

5.3.2 Compression for lossy networks

Neural network based approaches have already been shown to have high compression and the ability to adapt to the context, surpassing traditional approaches for compression. The goal of our work is not to improve the quality or compression achieved by the past literature, but rather look at how the flexibility of neural networks can be used to compress video data for a growing range of video transmission scenarios. Specifically, we seek to answer the question of whether neural networks can learn to generate representations that are resilient to network loss, amenable to multirate streaming, and tunable to operate at the desired point of trade-off between compression, loss resilience and bitrate adaptability.

An important aspect of compression and loss resilience is the manifestation of artifacts under high compression and loss rates. While the impact on traditional codecs is perhaps better known, Figure 5.2 also illustrates the impact that losses can have on neural network codecs. In this case, the data was serialized in a row-wise manner, which is why, the loss impacts a particular section of the frame. On the other hand, if the data is packetized in a different way (e.g. according to a reversible pseudorandom pattern), the impact will be much more spread out spatially and harder to notice. Note that his type of distortion under error conditions may not apply to all neural network codecs. For example, in [111], the reconstruction output under high compression ratios may produce a completely different image (e.g. a Ford car may be replaced by a Chevy). This is a result of using GANs, which trains the neural network to generate realistic looking images rather than a noisy approximation of the actual input. While this may be acceptable in some applications, we leave understanding the trade-offs of GAN-based designs to future work. In this paper, we focus on whether we can design a neural network based scheme that can be tuned to achieve alternate forms of degradation under compression and loss.

5.3.3 Neural networks background

The general architecture used to compress data using artificial neural networks is called an autoencoder. Autoencoders are neural networks trained to replicate the input layer at the output. Autoencoders usually have a hidden layer with a size smaller than the input, which forces the network to learn high level representations in a fewer number of bits. Training an autoencoder is an unsupervised process, i.e. there is no need for human intervention to augment the training data, since the neural network just needs learn to match the input to the output.

Neural networks for visual applications typically use convolutional neural networks[112]. A convolution layer in a neural network is a group of 3D filters which perform 2D convolution over the inputs. Each convolution filter reduces the input to a single feature with the same spatial resolution. Each convolutional layer has many such filters.

In the absence of any other layer, convolutional layers generally lead to an explosion in the size of the intermediate data. A pooling layer fixes this by spatially localized aggregation of the output of a convolutional layer. Another approach is to skip pixels in the spatial domain leading to strided convolutions. Strided convolutions result in significant savings in memory and compute because there is no need to calculate and store values which will be discarded after a pooling step. Strided convolutions are also better because they learn the pooling function rather than choosing a fixed pooling function.

Neural networks are trained by back-propagating the loss gradient from the output layer towards the input layer. The loss function of a neural network must be a differentiable function whose value is low if the output is ‘closer’ to the desired output. Mean squared error is an example of a loss function that can be used to train an autoencoder.

5.4 ViXNN Design

In this section, we describe the architecture of ViXNN. We first introduce the high level design ideas that allow ViXNN to learn loss resilient compression schemes. This is followed by a description of ViXNN’s compression architecture that allows it to leverage various redundancies in video data for efficient compression. Finally, we discuss ViXNN’s training procedure, which includes details regarding the loss function to be optimized and the generation of training data.

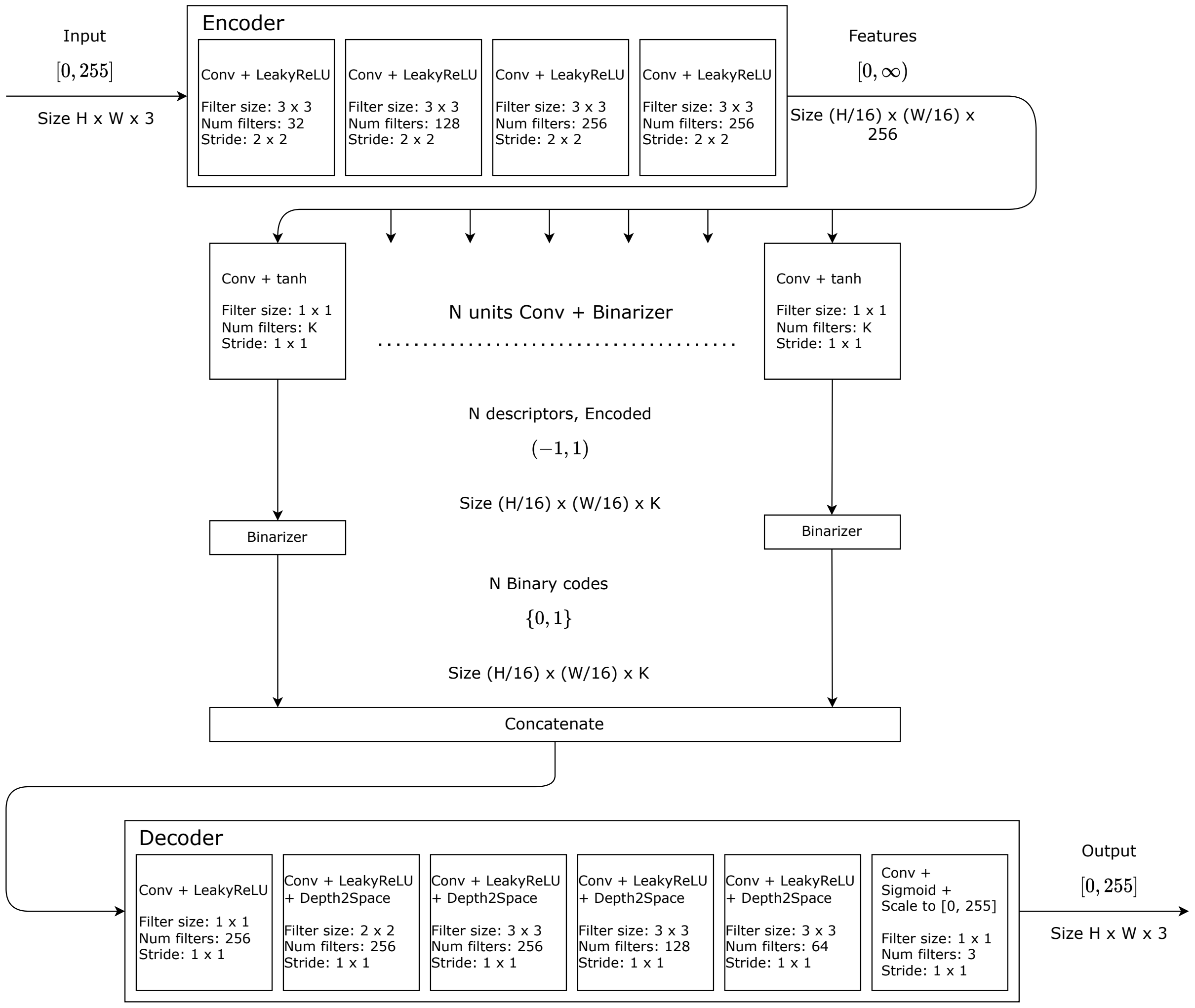

Input
$[0, 255]$
Size H x W x 3
Encoder
Conv + LeakyReLU Filter size: 3 x 3 Num filters: 32 Stride: 2 x 2
Conv + LeakyReLU Filter size: 3 x 3 Num filters: 128 Stride: 2 x 2
Conv + LeakyReLU Filter size: 3 x 3 Num filters: 256 Stride: 2 x 2
Conv + LeakyReLU Filter size: 3 x 3 Num filters: 256 Stride: 2 x 2
Features
$[0, \infty)$
Size (H/16) x (W/16) x 256
Conv + tanh Filter size: 1 x 1 Num filters: K Stride: 1 x 1
N units Conv + Binarizer
Conv + tanh Filter size: 1 x 1 Num filters: K Stride: 1 x 1
N descriptors, Encoded
$(-1, 1)$
Size (H/16) x (W/16) x K
Binarizer
Binarizer
N Binary codes
$\{0, 1\}$
Size (H/16) x (W/16) x K
Concatenate
Decoder
Conv + LeakyReLU Filter size: 1 x 1 Num filters: 256 Stride: 1 x 1
Conv + LeakyReLU + Depth2Space Filter size: 2 x 2 Num filters: 256 Stride: 1 x 1
Conv + LeakyReLU + Depth2Space Filter size: 3 x 3 Num filters: 256 Stride: 1 x 1
Conv + LeakyReLU + Depth2Space Filter size: 3 x 3 Num filters: 128 Stride: 1 x 1
Conv + LeakyReLU + Depth2Space Filter size: 3 x 3 Num filters: 64 Stride: 1 x 1
Conv + Sigmoid + Scale to [0, 255] Filter size: 1 x 1 Num filters: 3 Stride: 1 x 1
Output
$[0, 255]$
Size H x W x 3

Figure 5.3: ViXNN end-to-end architecture

5.4.1 Overview

Any compression architecture has two primary components: the encoder and the decoder. The encoder takes as input raw frame data and generates a compressed representation of the input data. The compressed data is transmitted over the network and sent to the receiver, where the decoder is used to recover uncompressed data from the compressed bits.

ViXNN achieves loss resilience by generating multiple streams of compressed video that constructively interact to generate higher quality video at the receiver. This is stylistically similar to SVC or MDC coding (as described in Section 5.2.2). ViXNN applies multiple non-linear transforms to the input frame, which are then compressed and transmitted over the network. The decoder expands the compressed data to recover the feature maps and then combines them to generate the final frame.

ViXNN uses an auto-encoder architecture based on [105]. The autoencoder has a narrow waist, whose output represents the compressed data. In ViXNN, we modify the architecture to have multiple narrow waists. These "descriptors" represent portions of data, that when combined, give higher quality. This property is imposed on the neural network by simulating loss in the neural network during the training phase. During training, the whole architecture behaves as a single neural network, and we simulate packet losses in the narrow waist by dropping descriptors at random. The descriptor drop pattern is based on how much loss resilience is required, and if MDC or SVC style compression is desired.

A key advantage ViXNN has over traditional SVC or MDC coding techniques is that the neural network can implicitly learn how the video data is semantically split across the multiple descriptors or streams, in order to optimize the performance for different loss scenarios.

5.4.2 Compression Architecture

ViXNN's end-to-end architecture is shown in Figure 5.3. In our design of ViXNN, we focus on compressing individual frames - while this does not leverage temporal redundancies in video data, we believe ViXNN's design can be adapted to more modern video compression architectures that leverage temporal redundancy. A discussion is presented in Section 5.4.5.

The architecture has three major componenents: first, the encoder transforms the input data into feature space using a series of strided convolutions. The features are then converted into multiple descriptors using multiple parallel convolutional layers. The role of these convolutional layers is to combine the features in different ways in order to generate multiple descriptors that constructively interact and result in higher quality compression. At the decoder, the multiple descriptors are concatenated, and passed through a series of convolutional filters and spatial expansion layers that restore the size of the original frame and reconstruct the pixel values.

The encoder first performs a feature transform on an input RGB frame using 4 fully convolutional layers. Each convolution has a stride of 2, which reduces each spatial dimension by a factor of 2 after every convolutional layer. Assuming the input is a color image of size $H \times W \times 3$, the spatial dimensions of the output of the first layer is $\frac{H}{2} \times \frac{W}{2}$. Hence, The final output of the encoder has a size of $\frac{H}{16} \times \frac{W}{16}$.

The output of every convolutional layer in the encoder is gated using a non-linear activation function. Specifically, in our implementation, we use the Leaky ReLU [113] activation function.

After the feature transform, the output is fed to N different convolutional layers simultaneously in order to generate multiple descriptors. These convolutional layers have K filters each, and have a filter size of 1x1 and a stride of 1x1. Each convolutional layer selects and combines the features differently in order to generate multiple descriptors that interact constructively to give higher video quality. The output of these convolutions are gated using a `tanh` activation layer and passed through a binarizer layer, that digitizes the values by quantizing the output of the `tanh` activation into 0-1 bits.

Thus, after the entire encoding process, there are N descriptors, each of size $\frac{H}{16} \times \frac{W}{16} \times K$ bits. The descriptors can then be packetized for transmission. The packetization scheme is designed such that a single packet contains data from at most one of the N descriptors, such that a packet loss corresponds to losing a particular descriptor.

At the decoder, the received descriptors are concatenated, with the missing descriptors replaced by zeros. In order to keep the average of the compressed data the same before decoding, the values of the descriptors are scaled by $\frac{N}{N_d}$, where N_d is the number of descriptors being decoded, and N is the total number of descriptors.

The decoder involves a series of convolutions, that are interspersed with (2×2) depth-to-space operations. A (2×2) depth-to-space operation on an input with height H, width W and C channels produces an output that has a height $2 \times H$, width $2 \times W$, and $\frac{C}{4}$ channels. This step essentially reverses the strided convolutions in the encoder, increasing the spatial dimension to the original image resolution. The intermediate layers use leaky ReLU as the activation function. The final convolution generates an image with three channels. We use a sigmoid activation in the final convolution, which has an output range $(0, 1)$, and scale up the values by 255 in order to generate pixel values in the range of the original input.

5.4.3 Adding loss resilience and variable rate coding

Scalable video coding techniques achieve two goals: loss resilience and variable rate coding. We follow a similar design philosophy to add loss resilience and variable rate coding to ViXNN. ViXNN's compressed output generates multiple descriptors, which are concatenated at the receiver and decoded. If the end-to-end neural architecture is trained without any additional steps, the multiple descriptors simply behave like a single large layer. In this case, the neural network does not learn any loss resilience at the descriptor scale, and only has the inherent bit-error resilience of using neural networks to compress data. The video quality of the decoded image degrades rapidly as more descriptors are lost.

The key observation we make here is that we can simulate the dropping of the descriptors during training, and the neural network will learn representations of the input frame that are resilient to losses at the descriptor level. In order to do this, we add a custom loss simulation layer after the binarizer. During training, the loss simulation layer will randomly drop descriptors by multiplying them by zero.

Different patterns, probabilities and structures can be used when randomly dropping descriptors. This enables ViXNN to learn MDC- and SVC-like loss resilience. Traditional MDC descriptors are designed so that the output can be decoded from any subset of descriptors. In order to force the neural network to do so, we can simply drop or keep each descriptor according to a fixed random probability. The value of this probability affects the video quality and the loss resilience. When descriptors are dropped with high probability, the neural network will learn a video encoding that achieves good video quality when many descriptors are dropped, but the peak video quality with all the descriptors will be lower than what would be achieved without simulating loss during training. This is because the encoding learned by the neural network will share more redundant data across all the descriptors, since during the training phase, only a few descriptors will be on during each iteration. This leaves less space for representing finer details in the descriptors. If the descriptors are dropped with lower probability, the neural network will share less redundant data across all the descriptors, since it is less likely to have seen situations with only a few descriptors that haven't been dropped during the training phase. This will increase the rate at which the video quality falls off as descriptors are lost, but will result in higher peak quality when no descriptors are dropped. This is because there are more bits available for representing the finer details.

In SVC coding, the descriptors have a hierarchical structure, and the enhancement layers that represent finer details require the base layers to be decoded first. If the base layer is lost, then the frame cannot be decoded. To simulate this in ViXNN, we simulate the loss of descriptors such that descriptor i is dropped, everything after that descriptor is also dropped. In order to do this, we sample an index i between 1 and N, where N is the number of descriptors, and drop everything after descriptor i during training. SVC-like approaches are better when packet loss is not a major concern, but bitrate adaptability is important. For example, in the case of wireless broadcast, an SVC like approach enables each receiver to perform their own rate control, and requires only a single encoding of the video data.

ViXNN's design also enables variable bitrate encoding. If the sender is bandwidth constrained, it can simply fewer descriptors. Compared to [105], ViXNN is able to achieve variable bitrate using a single-shot encoding, although the limitation is that the maximum bitrate, and hence, the maximum quality is limited. In [105], the authors propose iteratively encoding the residuals, which does not impose a static maximum limit on the bitrate.

5.4.4 Choosing a loss function

One of the biggest advantages of using neural networks for compression is the ability to use arbitrary loss functions to optimize the behavior for the desired goals. The loss function is the prediction error between the original input data and the reconstructed output after going through compression. The simplest option is to use the $L2$ distance as the error metric. This implies that the the neural network will get trained to maximize the PSNR value. A second option is to use the dissimilarity index (DSSIM), which is essentially the inverse of SSIM as the loss function. This will optimize the neural network for SSIM.

There are other potential benefits of using an architecture like ViXNN for general video compression tasks. For instance, the loss function can be different based on how many descriptors were lost - this enables learning end-to-end compression schemes where the quality degrades in a desired manner when descriptors are lost. For example, when all descriptors are available, ViXNN could be trained to reconstruct the original image, but when few descriptors are available, ViXNN could be trained to optimize for generating a low color-depth image instead. We demonstrate using a simple example in Section 5.5.4.

5.4.5 Discussion on Leveraging Temporal Redundancy

Video data has a lot more redundancy than image data. This arises due to the temporal nature of video and the fact that successive frames are very similar, differing due to object motion and camera panning movements. The focus of this chapter is to show the power of neural networks in learning end-to-end loss resilient compression schemes, and thus, focuses on a simple frame-by-frame compression approach. Below, we discuss some possible extensions and alternative approaches that can leverage temporal redundancy, and thus, achieve higher compression ratios.

The first approach is to stack multiple frames together at the input and compress them simultaneously. This approach does have a latency trade-off - before compression, the encoder needs to wait for all of the frames that are being compressed together to be available before they are compressed and transmitted. Thus, this solution is not well suited for low latency applications like VR streaming. There are alternate applications where this approach would be useful - for example, this approach could be used for multi-rate wireless broadcasts, similar to Softcast [20].

This is similar to the approach used in Softcast, with the key difference that the core compression methodology is learned by the neural network instead of a human designed compression scheme.

A potential second approach is to temporally compress the outputs of the binarizer layer. Suppose the binarizer layer had 24 channels. The output of the binarizer layer has width $\frac{W}{4}$ and height $\frac{H}{4}$. The 24 bits in the channel dimension can be interpreted as three 8 bit color channels, and then compressed using a traditional video compression scheme. Unfortunately, in our experience, this does not seem to improve the compression ratio. The primary reason is that we cannot leverage the benefits of lossy compression at this stage. Traditional video compression schemes quantize the data for lossy compression, which causes the loss of bit values. While this makes sense if the 8 bits representing each color was derived from real pixel data - in this case, the least significant bits are less important. In the case of ViXNN's output at the binarizer layer, each bit is equally important, which causes significant reconstruction errors if 24 bits in the channel dimension are interpreted as the color value of a pixel and quantized.

The third approach, which is most promising, is to incorporate the core idea in ViXNN - training the neural network for loss resilience - into more recent end-to-end neural network architectures that have been proposed for video compression. These include works like DVC [114] and [115]. These modern neural video compression techniques are also able to leverage the temporal redundancy in video data - this can significantly improve the compression efficiency, resulting in higher video quality for a given bitrate.

5.5 Evaluation

We first compare ViXNN and JPEG to demonstrate ViXNN's compression abilities. We then show how ViXNN performs under packet losses and bit errors and compare it to a few configurations of H.264/AVC encoding. We then show how ViXNN can be tuned to achieve the desired loss behavior for the needs of specific applications.

Training data One of the motivations for using neural networks for compression was their ability to learn the characteristics of the data and achieve better performance than codecs designed for general use. In our evaluation, we do not train the neural network for specialized data. We use the LabelMe-12-50k [116] dataset, which is a collection of 50,000 natural images. To improve the performance of ViXNN across multiple scales of images, we extracted multi-resolution patches from the dataset and resized them to 32×32 patches to generate our training set. This results in a good mix of high and low frequency patches for training.

Training for loss resilience In our evaluation, we use a 128 bit binarizer with 8 descriptors. Each descriptor thus generates 16 bits for each 16x16 patch in the input data. This architecture can achieve a maximum of 0.5 bpp. We trained four versions of ViXNN, each with different loss behavior. We trained three models with MDC like dropout. In each model, the descriptors are independently randomly dropped with a probability of 0.1, 0.3 and 0.5 respectively. We will refer to these models as ViXNN 0.1, ViXNN 0.3 and ViXNN 0.5. The fourth model was trained using SVC like dropout. We refer to this model as ViXNN SVC. In each training iteration, a random prefix of the descriptors are dropped. The length of the prefix is decided by sampling a uniform random variable. To drop a descriptor, the output of a descriptor is set to zero. In practice, lost descriptors can be detected by adding a small amount of metadata to each descriptor which includes the frame number and the sequence number of the descriptor.

Loss function A typical choice for the loss function for training an autoencoder is the mean squared error. Optimizing for the mean squared error directly optimizes the PSNR. This often results in blurry images. The SSIM [59] index, or more accurately, the structural dissimilarity index (DSSIM) can also be used as the objective function. We use a 50-50 weighted combination of the mean squared error and the DSSIM as the loss function.

Comparison with H.264/AVC We use the H.264/AVC codec as a baseline for comparing the performance of ViXNN under packet loss and bit errors. We use `FFmpeg` [117] as the front end to the `libx264` [118] implementation on Linux. We use three different sets of configuration options for `libx264` in our comparison. In the first configuration, which we will refer to as "GOP1" for simplicity, we set the GOP size to 1. In this mode, each frame is independently coded, checking the propagation of errors to successive frames. In the second configuration, we set the GOP size to 8, but disable B frames since B frames increase the latency significantly and increases the susceptibility to losses. We refer to this configuration as "GOP8". In the third configuration, we enable the intra-refresh feature of H.264. We refer to this as "Intra". Intra refresh uses a wave of I blocks instead of using a single I frame in every GOP. This guarantees that some part of the picture is refreshed every frame, reducing the propagation of errors to successive frames.

To ensure a fair comparison, we use a bit-rate that is comparable to the bit-rate generated by ViXNN. It is important to note that the goal of the paper is not comparing the compression performance of ViXNN when no losses occur. H.264/AVC is a very advanced codec and easily surpasses ViXNN when there are no losses. Another point to keep in mind while reading the evaluation is that our models were trained for a limited time on limited data. The performance of neural networks can be vastly improved by fine tuning various parameters like the learning rate and batch size, and choosing the best instance out of multiple training sessions.

Test data It is important to choose a diverse set of videos for testing. We chose a set of 4 videos (akiyo, foreman, coastguard, bus) from the Xiph.org Video test media [119] collection for our evaluation. These videos are uncompressed videos without chroma sub-sampling. We use the test images on `http://imagecompression.info/test_images/` to evaluate the baseline compression performance on different resolutions.

Implementation details We implemented the neural network with Keras [120] using the Tensorflow [121] backend. We used the Adam [122] optimizer for training the models. We switched between using an EC2 GPU instance and an NVIDIA 1080 Ti on a local machine for the training and evaluation.

5.5.1 Compression using ViXNN

We used ViXNN SVC to compress the test images from `http://imagecompression.info/test_imag` at different bpp values. We compare the performance of ViXNN and JPEG on two sets of resolutions. In Figure 5.4, HR (high resolution) refers to the original images. To get the second set of plots, we resized the images to fit in a 256×256 box. A bpp value of 0.5 is obtained when all the descriptors are used for decoding. We deterministically drop the descriptors to get lower bitrate coding.

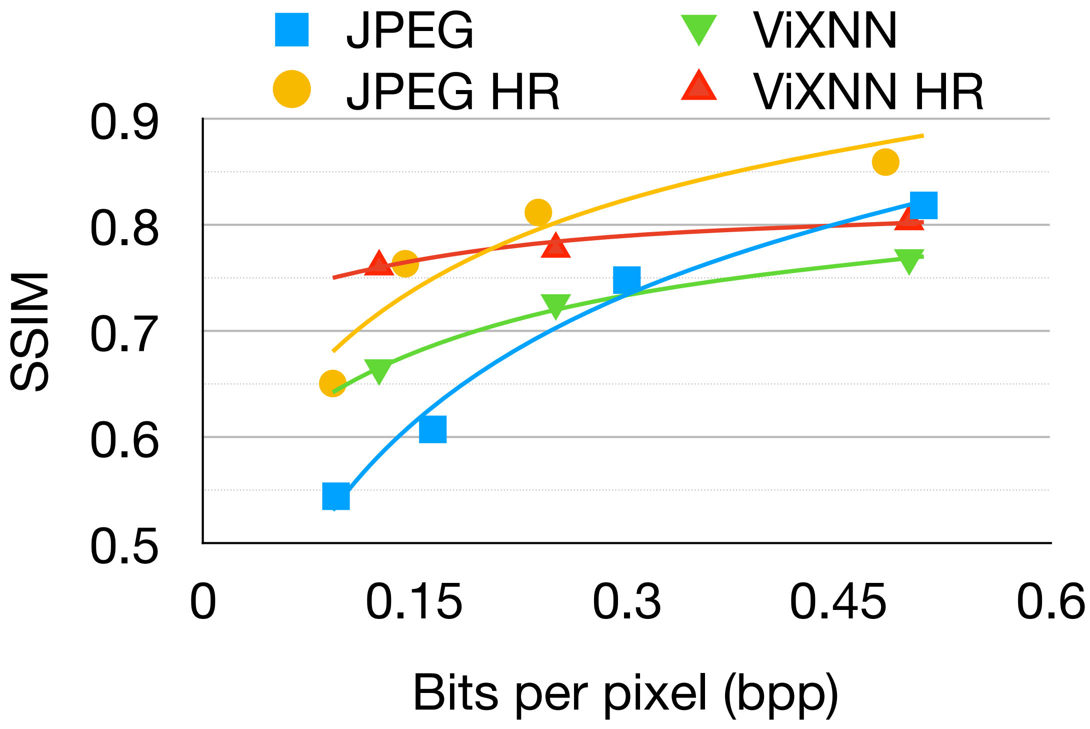


Figure 5.4: Compression performance of ViXNN without losses

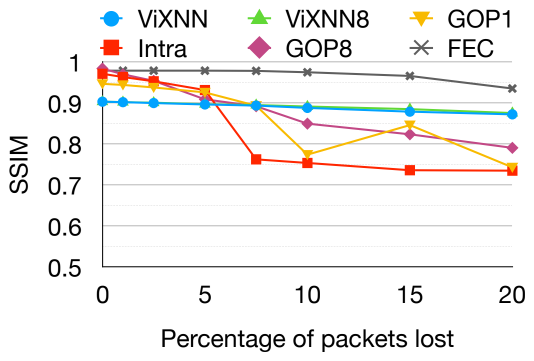
ViXNN outperforms JPEG for low resolution images and lower bitrates. ViXNN successfully achieves variable bitrate encoding without retraining, and while using one-shot compression instead of iterative residual compression [105]. We believe that the reason for poor performance on higher resolution images is the inability to take advantage of large scale spatial redundancies in the data. In Section 5.4.5, we discuss potential approaches in order to leverage temporal redundancy in video data for additional compression. One of the primary reasons why algorithms like JPEG can achieve high compression is quantization, but the bit vector generated by ViXNN cannot be quantized. The reason is that a small difference in numerical value can result in a large hamming distance in the bit vector. We think that intelligently mapping the bit vector will allow lossy compression of the compressed data, but we leave that for future work.

To keep the comparison fair, the JPEG bitrates were calculated by subtracting the size of the metadata, including the huffman coding tables and the quantization tables.

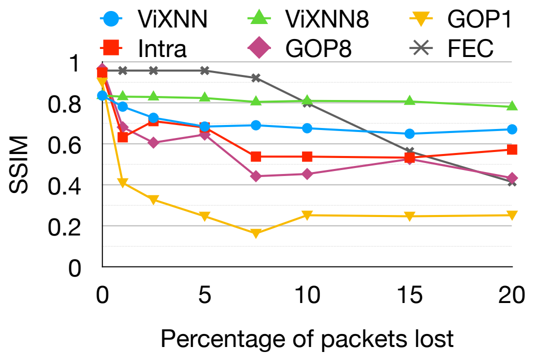
5.5.2 Resilience to packet loss

We evaluate the performance of ViXNN 0.3 and compare it to H.264/AVC's behavior under packet loss ranging from 1% to 20%. 20% is a very high packet loss rate, but is not uncommon in transient conditions on wireless networks with mobile endpoints.

A typical system that uses H.264/AVC over a lossy network most likely uses some form of forward error correction to recover from packet losses. To simulate the performance of H.264/AVC with FEC, we assume a linear FEC block code. We assume that the coding parameters are $(n = 10, k = 7)$, which implies that all the data can be recovered using $k = 7$ out

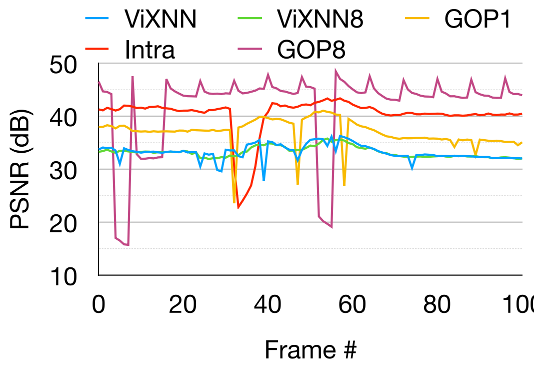


(a) Average SSIM vs packet loss



(b) Minimum SSIM vs packet loss

Figure 5.5: ViXNN vs. H.264/AVC under packet loss



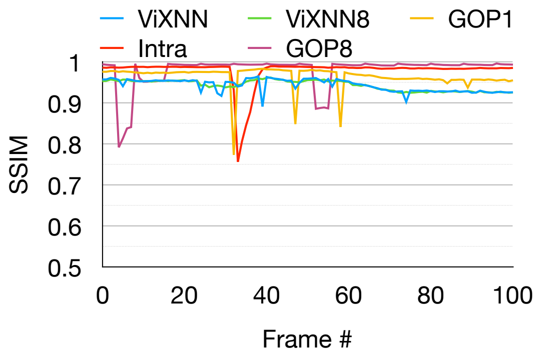


Figure 5.6: Timeseries plots of PSNR and SSIM under 5% packet loss

of $n = 10$ blocks. For a given packet loss rate p, we calculate the effective group loss rate using the following formula:

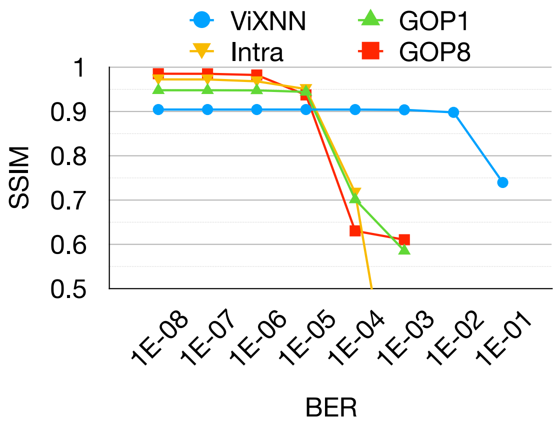
$$p_{group} = 1 - (q^{10} +^{10} C_1 q^9 p +^{10} C_2 q^8 p^2 +^{10} C_3 q^7 p^3) \qquad (5.1)$$

where $q = 1 - p$.

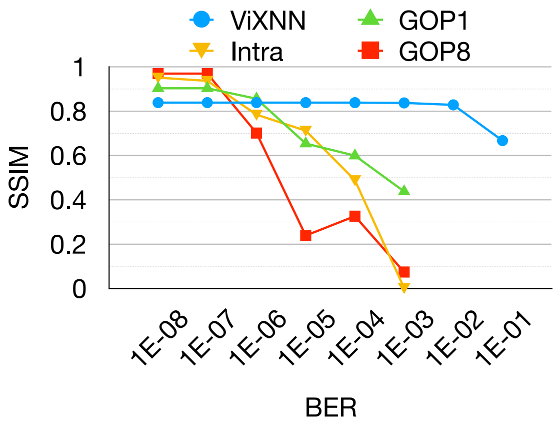
We also evaluate an alternate byte ordering technique for ViXNN. When a burst loss occurs, it can wipe out all the descriptors generated by ViXNN for a single frame. We reorder the data so that the corresponding descriptors of a sequence of 8 frames are contiguous. This reduces the chance that a single packet loss can result in the loss of many descriptors. The caveat is that it introduces a delay of 8 frames, but achieves better resilience to packet loss. We refer to this as ViXNN8.

Figure 5.5 shows the results of simulating packet loss on H.264/AVC and ViXNN. For loss rates below 10%, H.264/AVC outperforms ViXNN in terms of the average SSIM, but the minimum values across the frames are much worse than ViXNN for even small amounts of packet loss. H.264/AVC with FEC performs better than ViXNN for loss rates below 10%, but degrades sharply beyond that. It is also evident from the minimum SSIM values that ViXNN8 is more robust to packet losses than ViXNN.

Figure 5.6 is a frame-by-frame plot of the PSNR and SSIM for ViXNN, ViXNN8 and the three configurations of H.264/AVC. GOP1 is relatively stable compared to GOP8 and Intra, but



(a) Average SSIM vs BER



(b) Minimum SSIM vs BER

Figure 5.7: ViXNN vs. H.264/AVC under bit errors

exhibits sharp drops in quality on packet loss events. ViXNN also exhibits sharp drops on packet loss, but they are less pronounced as compared to GOP1. On the other hand, ViXNN8 has no sharp drops in quality. The effect of a packet loss is spread over 8 frames and the quality of each frame drops slightly.

The time series figures also give important insight into the behavior of Intra and GOP8. When a packet loss occurs in GOP8, the quality remains low until an I-frame is received. At this point, the effects of the packet loss are nullified. On the other hand, the quality of Intra ramps up smoothly. This is because each frame has a few new I blocks which gradually nullify the effects of the packet loss event.

5.5.3 Resilience to bit errors

Using neural networks for compression has the added bonus of resilience to bit errors [123]. We evaluate the performance of ViXNN under bit errors and compare it to H.264/AVC in Figure 5.7. ViXNN outperforms H.264/AVC for BER greater than 10^{-7} in the worst case. The average PSNR and SSIM values suffer a sharp drop for BER greater than 10^{-5} in the case of H.264/AVC. ViXNN is highly robust to bit errors for BER values as high as 10^{-2}. Individual frames are not perceivably affected by bit errors in the case of ViXNN.

5.5.4 Tunability of ViXNN

The architecture of ViXNN provides immense flexibility in terms of the behavior under loss. ViXNN can be trained to behave as desired by using different dropout strategies during the training phase.

Figure 5.8 compares the behavior of ViXNN 0.1, 0.3, 0.5 and SVC when descriptors are dropped. Since ViXNN 0.5 was trained to be robust to higher drop rates, the PSNR and SSIM values drop at a slower rate than ViXNN 0.1 and 0.3. The trade-off is the maximum quality achieved when there is no loss. ViXNN 0.5 is the worst. ViXNN SVC trades off resilience to

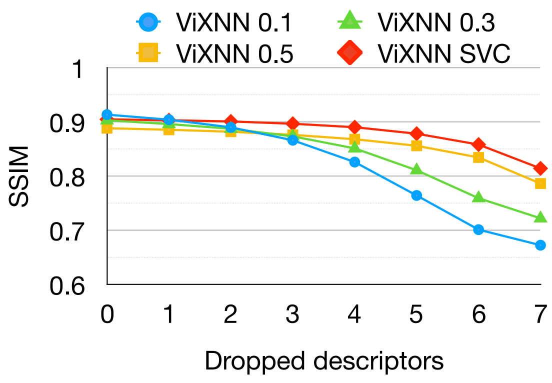
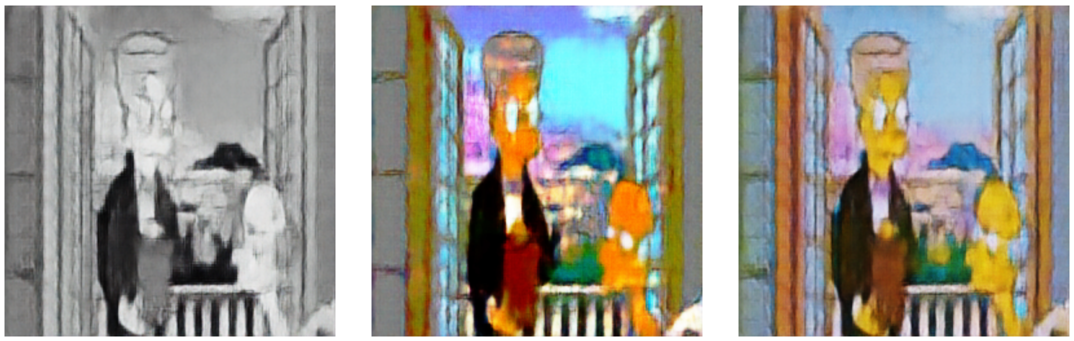


Figure 5.8: Tuning ViXNN for different loss behaviors



(a) Decoding without color descriptors (b) Decoding with low color depth descriptors only (c) Decoding with all the descriptors

Figure 5.9: ViXNN trained for alternate reconstructions under loss

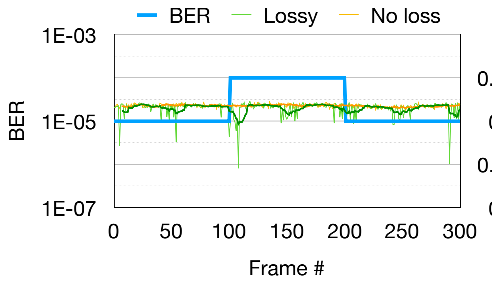
loss of arbitrary descriptors with higher overall quality. It is interesting to note that the plots for ViXNN 0.5 and ViXNN SVC are almost parallel. We think it is due to the fact that the expected number of descriptors dropped during each training iteration is the same for both.

We also showed earlier that ViXNN can trade off latency for better resilience to packet losses by rearranging the bits of the compressed data.

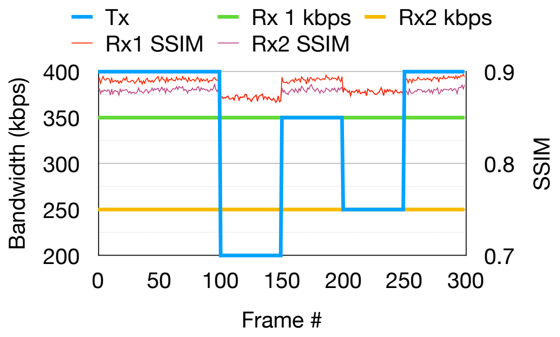
ViXNN can also be trained to have alternate low fidelity reconstructions under loss. We trained ViXNN with a dynamically weighted loss function based on the number of active descriptors. Figure 5.9(a) shows the output when ViXNN was trained to generate gray scale reconstructions under loss. Figure 5.9(b) was obtained from training ViXNN to reconstruct a low bit depth image when losses occurred.

This demonstrates the flexibility of ViXNN and it's ability to be adapted to the requirements of the application at hand. Consider two example applications, wireless VR and live mobile video.

Wireless VR VR applications have high bandwidth requirements and cannot tolerate drops in quality. An important aspect of wireless transmission is the ability to perform rate adaptation based on the SNR to reduce the bit error rate, but VR applications cannot tolerate these latencies. Obstacles can also cause errors and packet drops, especially when using high bandwidth



(a) ViXNN for VR



(b) ViXNN for live video

Figure 5.10: Tuning ViXNN for applications

millimeter wave WiFi [124].

Live mobile video Live mobile video is multicast in nature, and thus requires rate adaptation to cater to diverse set of receiver bandwidths. The data is typically transmitted over the Internet, where it is more prone to packet losses than bit errors. These applications can often tolerate a few hundred milliseconds of latency.

A setup similar to ViXNN 0.1 can be used for wireless VR. The encoded data can be transmitted at a constant data rate without worrying about bit errors or rate adaptation as ViXNN is robust to bit errors. All error correction mechanisms would have to be disabled. Packet losses may occur due to obstructions, but they are less frequent since bit errors do not cause packet drops. Using ViXNN 0.1 will give better quality as it is trained for lower packet loss rates.

On the other hand, ViXNN 0.5 can be used for live mobile video as it caters to a wider range of bitrates. The transmitter can encode the video and transmit all the descriptors as multicast datagrams. The receivers receive data based on their available bandwidth and the quality gracefully scales with the number of descriptors received. ViXNN 0.1 would result in slightly better quality but result in poor performance for receivers with low bandwidth.

Figure 5.10 shows the performance of ViXNN for VR and live video. In the case of VR, we simulated 5% packet loss for a period of 20 frames after every 50 frame interval. The bit error rate was set to 10^{-5} for the first and last 100 frames, going up to 10^{-4} for the 100 frames in the middle. We simulated time varying bandwidth for the transmitter and heterogeneous receivers with constant bandwidth for live video.

5.5.5 Computational and storage requirements

Neural networks are computationally intensive and require powerful GPUs for evaluation. We tested the encoding and decoding performance of ViXNN on an NVIDIA 1080 Ti GPU. The results are shown in Table 5.11.

Although these numbers are low, various techniques can be used to reduce the computational requirements, hence speeding up the encoding and decoding process. The design space of the encoder and decoder architectures can be explored to create light weight architectures. There has been a lot of work on approximate evaluation by leveraging the intrinsic redundancy present in

Resolution	**Encode fps**	**Decode fps**
SD (640×480)	72	49
FHD (1920×1080)	12	7
VR (3840×1080)	6	4
4k (3840×2160)	3	2

Figure 5.11: Encoding and decoding performance of ViXNN on an NVIDIA 1080 Ti GPU

neural networks [125]. In [126], the authors compare neural network evaluation speed on CPUs, GPUs and ASICs. ASICs can be up to $100\times$ faster than GPUs, while consuming less power.

The size of the trained models is also an important factor, as these models need to be sent to the end points and stored. Models tailored to the data may need to be retrained and redeployed from time to time when the data characteristics change. This requires expensive bandwidth and storage space on remote devices. ViXNN's encoder and decoder are each approximately 40MB in size. While this is not a lot, it is beneficial to have smaller models that can be deployed cheaply. Various model compression techniques can be used to reduce the sizes of the trained models. These techniques often have the added benefit of reducing computational and memory requirements[127]. [127] [128] proposes the joint training of encoders and decoders with different capacities, which allows the deployment of different models based on the performance limitations of the devices.

5.6 Conclusion

In this chapter, we discussed a technique to compress image and video data using neural networks in a loss resilient manner by incorporating loss simulation during the training of the neural network. This shows a promising way to achieve end-to-end loss resilient video compression that is highly tailored to the particular network loss patterns, video properties, and bandwidth regime. While ViXNN's evaluation is based on frame-by-frame compression, recent advances in neural video compression show promise and incorporating ViXNN into these new video compression architectures can achieve superior loss resilient video compression for low latency applications.

Chapter 6

Congestion Control Design for Emerging Video Streaming Applications

Cloud gaming and AR/VR applications aim to achieve parity or perform better than the locally rendered versions. Thus, systems designed for these applications must consistently achieve high video frame quality with minimal artifacts, and low end-to-end frame delay. Video quality and delay heavily depends on the performance of the congestion control algorithm (CCA). Congestion control algorithms for ultra-low latency applications must be able to achieve high throughput and low queuing delay, and must work well with the traffic pattern of streaming video in order to achieve low end-to-end frame delay. There are two key challenges that are not addressed by congestion control algorithms today:

1. In the presence of queue-building cross traffic, existing low-latency congestion control algorithms are not able to sustain high throughput in a reliable manner.
2. Most congestion control algorithms are video agnostic, and are not tightly integrated with the traffic pattern of streaming video - this can result in higher end-to-end frame delay even if the algorithm on its own can achieve low end-to-end packet delay.

In this chapter, we first list the key requirements for designing a congestion control algorithm for ultra-low latency video streaming applications like cloud gaming and AR/VR streaming. We then discuss some low-latency congestion control algorithms and their pitfalls when used for low-latency video streaming applications like cloud gaming and cloud AR/VR. We then show how the interactions between the video encoder, the congestion control algorithm, and the network affect the end-to-end frame delay, and present a case for designing tailored congestion control algorithms that tightly integrates network measurements, congestion control, and video encoding in order to achieve the demanding QoE requirements of emerging applications like cloud gaming and cloud AR/VR.

Following the general discussion on congestion control algorithms for emerging applications, we present a congestion control algorithm for cloud gaming and cloud AR/VR applications called SQP. SQP uses frame-coupled, paced packet trains to sample the network bandwidth, and uses an adaptive one-way delay measurement to recover from queuing, for low, bounded queuing delay. SQP rapidly adapts to changes in the link bandwidth, ensuring high utilization and low frame delay, and also achieves competitive bandwidth shares when competing with queue-building flows within an acceptable delay envelope. SQP has good fairness properties, and works well on links with shallow buffers.

6.1 Congestion Control Background for Low Latency Video Streaming

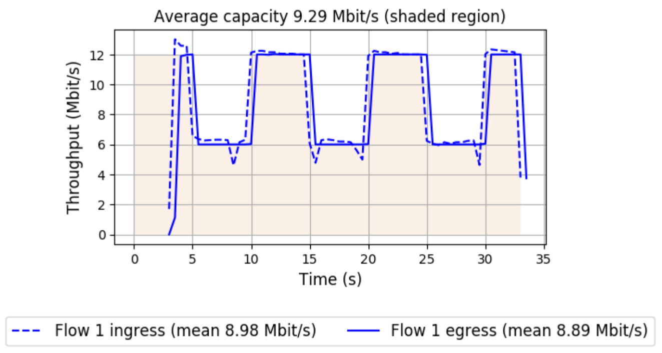
6.1.1 Traditional Congestion Control Algorithms

Traditionally, congestion control algorithms (CCAs) have been designed with the following key goals in mind:

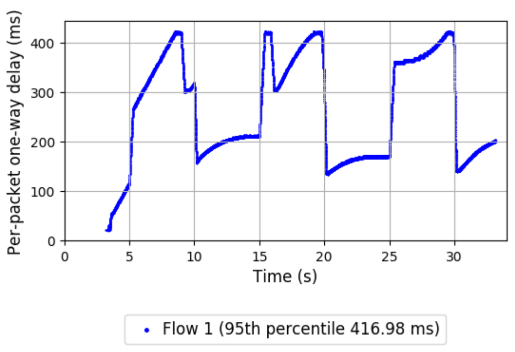
1. **Prevention of Congestion Collapse**: This is the primary goal of any congestion control algorithm. When multiple flows share a bottleneck, each CCA must take appropriate actions in order to avoid falling into a state of high offered load and low throughput due to catastrophic packet losses. For example, in the case of traditional algorithms like TCP-Reno [129] and TCP-Cubic [12], this involves reducing the congestion window to a safe level when packet losses occur.
2. **High Throughput**: The CCA must be able to utilize the available bandwidth in an effective manner in order to maximize the application layer throughput. Traditional loss-based CCAs use mechanisms like slow-start and additive increase in order to probe the network for more bandwidth, and maintain occupancy in the bottleneck queues in order to fully utilize the network bandwidth.
3. **Fairness**: Multiple flows sharing a bottleneck must get a fair share of the bottleneck link, and must avoid starvation. Traditional CCAs like TCP-Reno and TCP-Cubic achieve fairness without co-ordination due to their congestion window increase and decrease mechanism. For instance, TCP-Reno fairness mechanism is it's Additive Increase - Multiplicative Decrease (AIMD) mechanism of changing the congestion window.

Loss-based CCAs like TCP-Reno and TCP-Cubic largely satisfy these goals. Unfortunately, loss-based CCAs fall short in terms of meeting the performance requirements for many emerging applications and network environments. Emerging applications like cloud gaming, cloud AR/VR, and other real-time video streaming applications require very low delays. Loss-based CCAs use packet loss as a signal of congestion, which only happens when the in-network queues are filled up. This queue-building behavior results in very high end-to-end delays, and this is not acceptable for low-latency video streaming applications. We ran a simple test using the Pantheon [130] test-bed (using MahiMahi [3] for emulating the bottleneck). The throughput and delay graphs are shown in Figure 6.1 - while TCP-Cubic is able to fully utilize the link, the cost of doing that is very high queuing delays. In practice, these delays can be much higher depending on the size of the in-network queues and the available bandwidth.

Various Active Queue Management (AQM) schemes have been proposed in order to reduce the in-network queuing delays of loss-based CCAs, like RED and CoDel. These schemes work by dropping packets early *before the router buffers fill up*, which avoids the excessive delays caused by deep router buffers. While these approaches work well in controlled environments like data centers and home networks, deploying AQM schemes widely across the internet is an extremely challenging task.



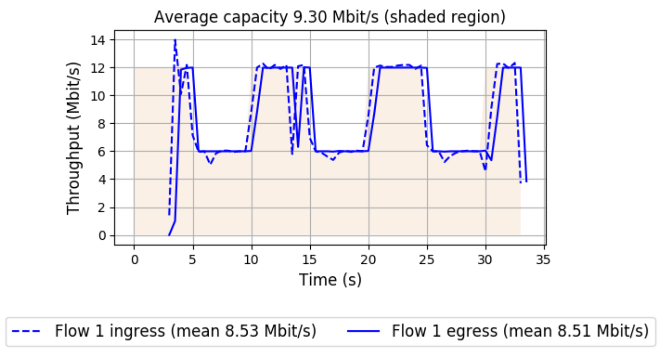
(a) Throughput.



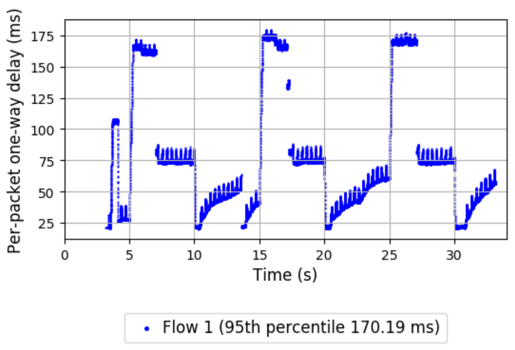
(b) Delay.

Figure 6.1: Throughput achieved by TCP-Cubic and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.

Some algorithms like TCP-BBR prioritize high throughput, but have design elements that aim to reduce the delay compared to loss-based CCAs like TCP-Cubic and TCP-Reno. For instance, TCP-BBR does not use loss as a signal, and instead, attempts to create a model of the network by using probes. TCP-BBR uses two types of probes - a bandwidth probe, and a round-trip time (RTT) probe. Every 8 RTTs, BBR increases it's send rate by a factor of 1.25 for 1 RTT, followed by a drain phase when it sends at a rate that is 0.75 $\times$ it's bandwidth estimate. For the next 6 RTTs, BBR transmits at it's bandwidth estimate. In addition, BBR measures the minimum RTT by significantly reducing the congestion window every 10 seconds. This serves the purpose of draining any built-up network queues, and allows BBR to get an estimate of the baseline link delay, which it uses for capping the maximum in-flight data to twice the bandwidth-delay product (BDP).



(a) Throughput.

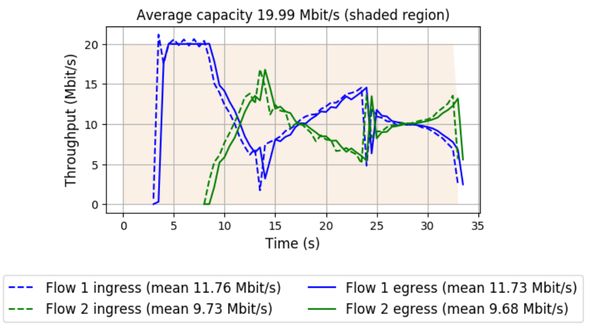


(b) Delay.

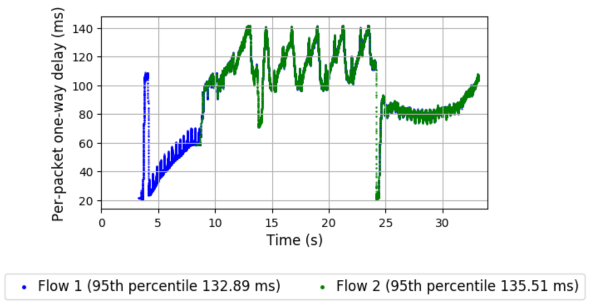
Figure 6.2: Throughput achieved by TCP-BBR and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.

Under normal operation on a very stable link, BBR can maintain low delays, but this is not true in the presence of delay jitter, bandwidth variations, and other network phenomenon like

ACK-aggregation and packet aggregation. In such scenarios, BBR fills the in-network queues until the in-flight data hits the $2 \times BDP$ cap. This is shown in Figure 6.2 - after each min-RTT probe and whenever the link rate goes up to 12 Mbps (e.g., at 10s, 14s, 20s, and 30s), the delay is initially low, but BBR slowly fills up the queue until it hits the 2 BDP cap. This is due to slight over-estimation of the link bandwidth in each probing cycle (MahiMahi emulates throughput at the millisecond-level, causing packet aggregation at the millisecond scale). The queue also fills up temporarily when the link bandwidth drops. BBR's rate estimate is based on the maximum bandwidth sample in a past window, and this causes high delays when the network bandwidth is variable at RTT-level timescales.



(a) Throughput.



(b) Delay.

Figure 6.3: Throughput achieved by TCP-BBR and the corresponding queuing delay on a 20 Mbps link where TCP-BBR shares the link with TCP-Cubic. The blue line is the TCP-BBR flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the TCP-BBR flow.

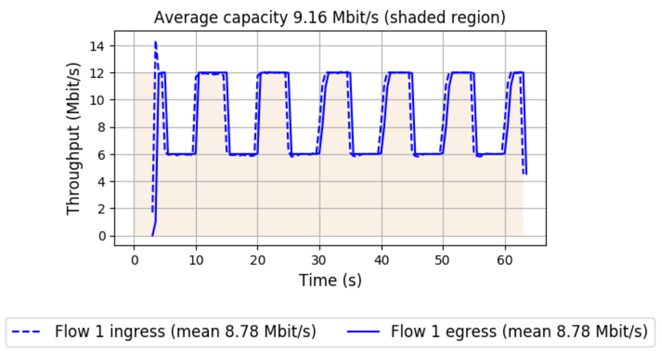
When BBR shares the bottleneck link with queue-building flows like TCP-Cubic or TCP-Reno, the min-RTT probe may not entirely drain the queue since the queue now also contains packets from the other flow. This causes BBR's delay floor to go up, and subsequently, BBR's maximum in-flight cap increases. This is an intentional design feature that enables BBR to compete with loss-based, queue-building flows like TCP-Cubic and TCP-Reno. This is shown in Figure 6.3.

The design of BBR makes it unsuitable for emerging applications like cloud gaming and cloud AR/VR because of the following reasons:

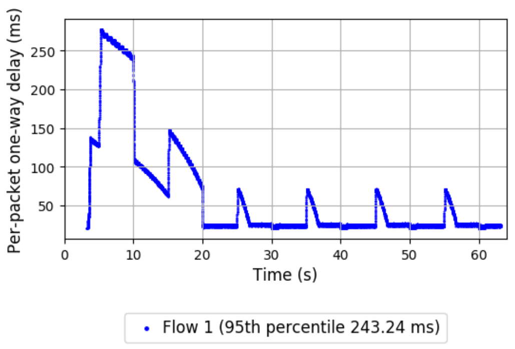
- BBR can build up to 1 BDP of in-network queues, causing higher delays that are not acceptable for latency sensitive real-time video streaming applications.
- BBR's minRTT probing mechanism means that no video frames can be sent during the probing period ($\tilde{2}$00ms) - this would result in the video stream stuttering for 200ms every 10 seconds. This is unacceptable for immersive applications like cloud gaming and cloud AR/VR.

6.1.2 Low Latency Congestion Control

In recent times, there has been increasing interest in designing new congestion control algorithms for achieving low delay by reducing the in-network queuing. Achieving the three goals mentioned in Section 6.1.1 in addition to the requirement of low delay is not an easy task. Network traffic on the Internet needs to navigate a diverse range of network conditions, like variable link bandwidth, delay jitter, lossy networks, and must share the bottleneck with other flows that may exhibit queue-building flows and use other congestion control algorithms. Thus, designing a CCA with low delay invariably comes with trade-offs, like low throughput in the presence of delay jitter, or poor fairness when competing with other queue-building flows.

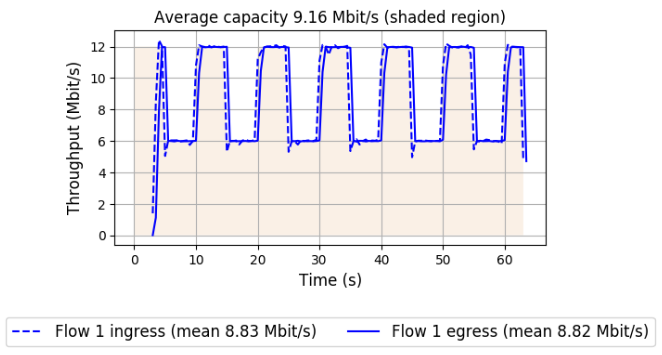


(a) Throughput.

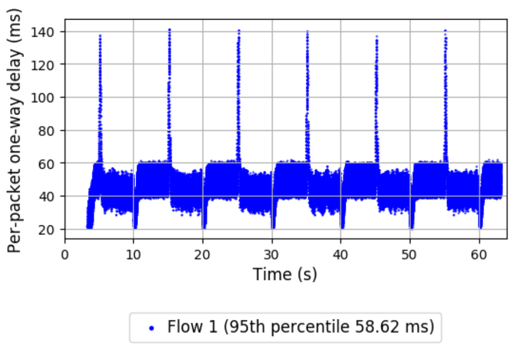


(b) Delay.

Figure 6.4: Throughput achieved by TCP-Vegas and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.



(a) Throughput.



(b) Delay.

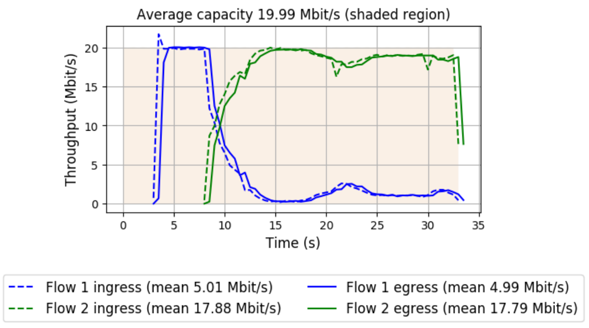
Figure 6.5: Throughput achieved by Sprout and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.

6.1.2.1 CCAs that Prioritize Low Delay

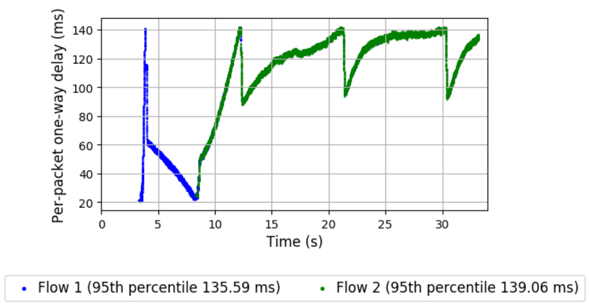
Congestion control algorithms like Sprout [22], TCP-Vegas [131] and TCP-Lola [132] prioritize low delay in their design. These algorithms work well in isolation and maintain low delay while achieving good utilization.

Figure 6.4 shows the throughput and delay achieved by TCP-Vegas on a link that oscillates between 12 Mbps and 6 Mbps. In the initial part of the trace, TCP-Vegas induces a large amount of queuing, but this is eventually drained and Vegas is able to maintain low delay. There are some delay spikes when the link rate drops, but Vegas recovers quickly from these self-induced queues by reducing the send rate.

Sprout behaves in a similar manner to TCP-Vegas. Figure 6.5 shows the throughput and delay achieved by Sprout on a link that oscillates between 12 Mbps and 6 Mbps. While Sprout's delay is slightly higher than the delay caused by TCP-Vegas, it is more consistent and the delay spikes when the network bandwidth drops are much shorter in duration. The peak delay when the link drops is higher than TCP-Vegas - this can be attributed to the fact that Sprout does not use a congestion window mechanism and is not ACK-clocked. Instead, Sprout transmits at a rate that is determined based on packet receive timestamps. This causes Sprout to transmit at the previous rate (i.e. 12 Mbps) when the link drops to 6 Mbps for one round trip, until the feedback is received by the sender.



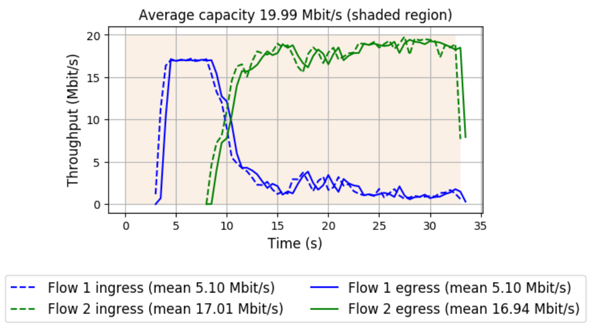
(a) Throughput.



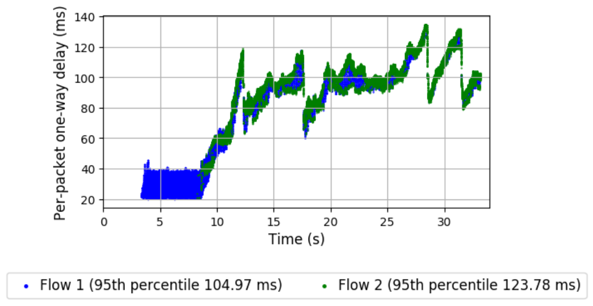
(b) Delay.

Figure 6.6: Throughput achieved by TCP-Vegas and the corresponding queuing delay on a 20 Mbps link where TCP-Vegas shares the link with TCP-Cubic. The blue line is the TCP-Vegas flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the TCP-Vegas flow.

While these algorithms are able to maintain low delay, their performance suffers when competing with queue-building flows, and in networks with significant delay jitter. When high delays are caused by competing flows, these CCAs have no way to know whether or not the delay was self-induced or caused by competing flows, and hence, they reduce the throughput in response to any observed increase in delay. Figures 6.6 and 6.7 show the performance of TCP-Vegas and Sprout, respectively, when they shares the link with a TCP-Cubic flow. As soon as the Cubic flow enters the link, the throughputs of the both, the Vegas flow, and the Sprout flow, drop drastically and these flows are unable to attain their fair share of the bottleneck link.



(a) Throughput.



(b) Delay.

Figure 6.7: Throughput achieved by Sprout and the corresponding queuing delay on a 20 Mbps link where Sprout shares the link with TCP-Cubic. The blue line is the Sprout flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the Sprout flow.

The inability to achieve a fair share of the throughput when competing with queue-building flows makes CCAs that prioritize low delay unsuitable for emerging video streaming applications like cloud gaming and cloud AR/VR. This is because cloud-rendered applications require high, stable bandwidth in order to ensure high video frame quality and minimal video compression artifacts.

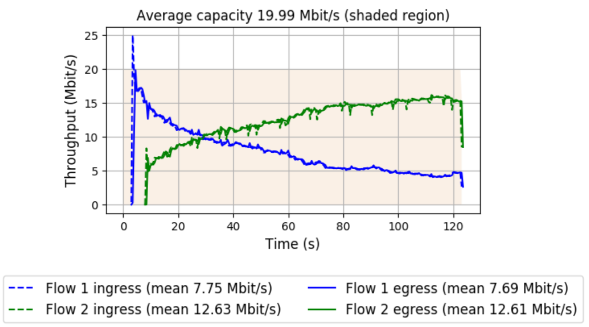
6.1.2.2 Utility-based CCAs

Congestion control algorithms like PCC [133, 134] have been proposed in order to solve the challenges associated with CCAs that prioritize low delay. PCC's congestion control mechanism involves conducting randomized experiments in real time, where the sending rate is changed for short probing durations in order to probe the network for more throughput or to measure the amount of network queuing. The performance achieved as a result of running these experiments are fed into a utility function based on delay and throughput, which is used to determine whether the bandwidth estimate should increase or decrease.

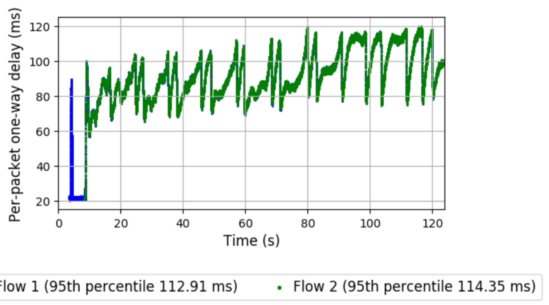
Compared to CCAs that prioritize low delay, PCC achieves higher throughput when competing with a queue-building flow like TCP-Cubic. This is shown in Figure 6.8. Note that while PCC is stable and achieves high throughput in the short term, it's throughput eventually drops in the presence of a long running Cubic flow.

PCC's stable throughput when competing with queue-building flows has a flip-side - on variable links, PCC converges extremely slowly. This results in high delays when the link bandwidth drops. This is shown in Figure 6.9 - PCC barely responds to the drop in the link rate to 6 Mbps. In addition, PCC does not utilize the available bandwidth fully when the link rate goes back up to 12 Mbps.

PCC-Vivace [134] is a more reactive version of PCC. It's performance on the oscillating link is shown in Figure 6.10. While PCC-Vivace is faster when probing for more bandwidth when the link rate increases, it also suffers from slow convergence when the link rate drops, causing high delays.

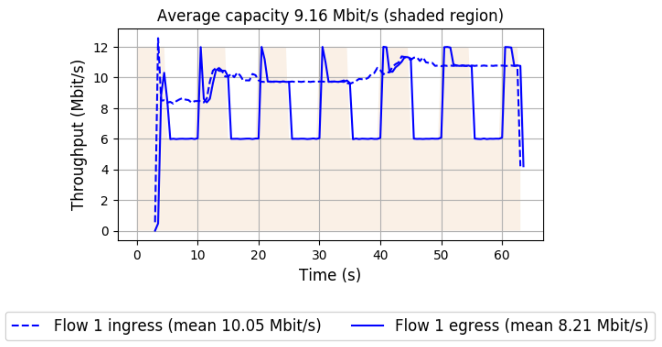


(a) Throughput.

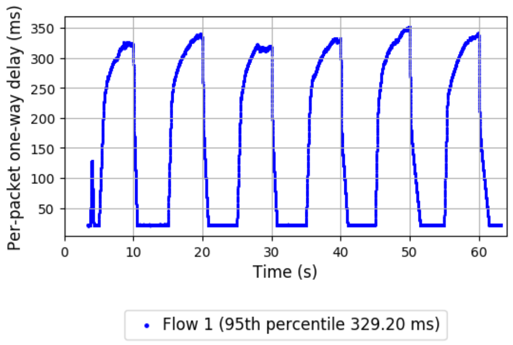


(b) Delay.

Figure 6.8: Throughput achieved by PCC and the corresponding queuing delay on a 20 Mbps link where PCC shares the link with TCP-Cubic. The blue line is the PCC flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the PCC flow.

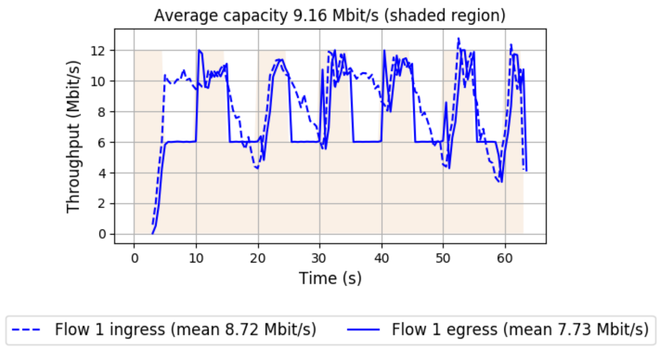


(a) Throughput.

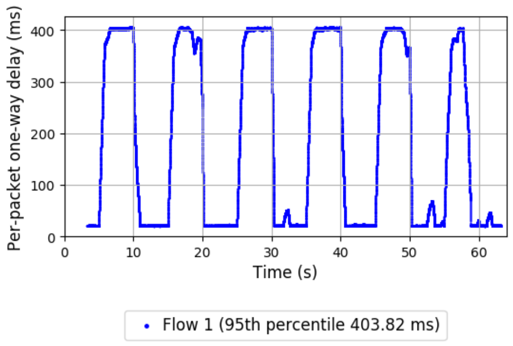


(b) Delay.

Figure 6.9: Throughput achieved by PCC and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.

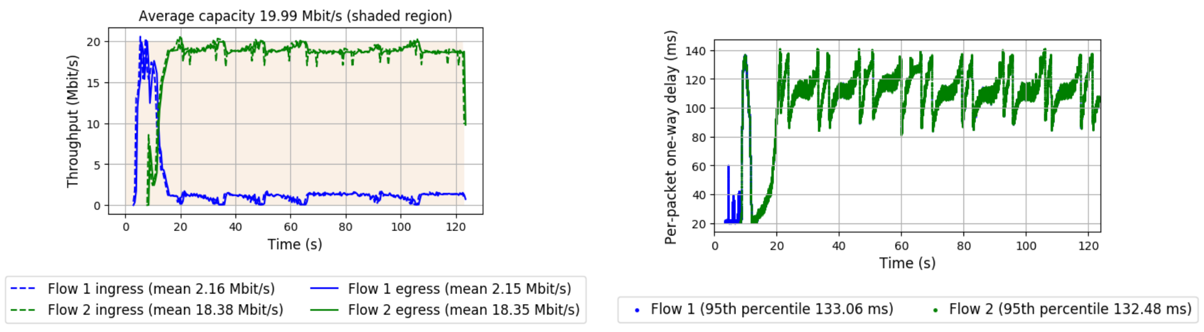


(a) Throughput.



(b) Delay.

Figure 6.10: Throughput achieved by PCC-Vivace and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.



(a) Throughput. (b) Delay.

Figure 6.11: Throughput achieved by PCC-Vivace and the corresponding queuing delay on a 20 Mbps link where PCC-Vivace shares the link with TCP-Cubic. The blue line is the PCC-Vivace flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the PCC-Vivace flow.

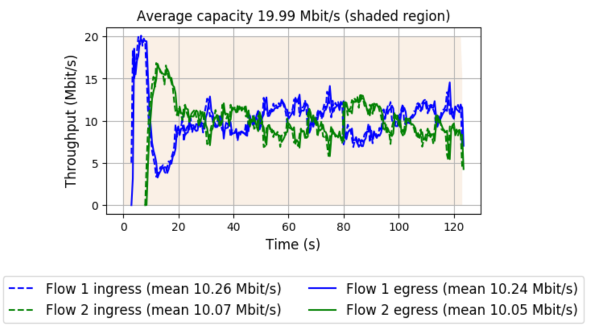
PCC-Vivace's faster reaction times come with the downside that it achieves low throughput when competing with queue-building flows like TCP-Cubic. Figure 6.11 shows the performance of PCC-Vivace when competing with TCP-Cubic. As soon as the Cubic flow starts, the throughput of PCC-Vivace drops drastically, and it is unable to probe for more bandwidth for the rest of the duration.

Utility-based algorithms like PCC and PCC-Vivace have slower convergence times, which causes high delays in situations where the network bandwidth drops and bandwidth underutilization when the network bandwidth increases. In addition, these algorithms are not robust, and can suffer from starvation (e.g. PCC-Vivace). Both of these properties make these algorithms not suitable for cloud-rendered applications, since these emerging applications require consistent, high throughput and extremely low delay.

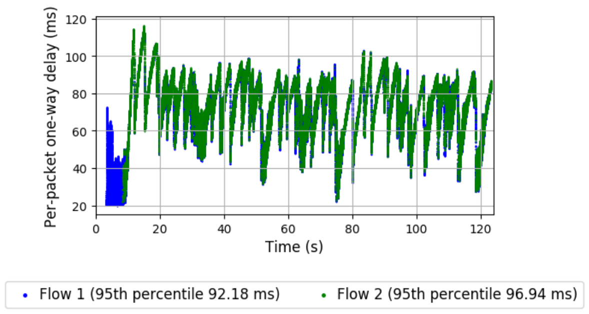
6.1.3 Mode-switching Low Latency CCAs

Congestion control algorithms like Copa [21], and Nimbus [135] use a different approach in order to achieve the dual goals of low delay when running in isolation, and high throughput when sharing the bottleneck with queue-building flows. Copa is a low-delay algorithm that uses an explicit mode-switching mechanism when it detects the presence of other queue-building flows.

Copa's core mechanism is a 5-RTT cycle, where it injects packets into the queue and subsequently drains the queue during the 5-RTT period. Copa's target rate calculation uses a parameter δ, which represents the algorithm's bias towards delay or throughput. A lower value of δ makes Copa more aggressive in terms of throughput, whereas a higher value of δ makes Copa more sensitive to delay. When Copa detects that the queue has not drained after a 5-RTT period, it switches to competitive mode, where δ is chosen dynamically in a manner that mimics the AIMD mechanism of traditional loss-based congestion control algorithms. This is shown in Figure 6.12. In the first 5 seconds (before the TCP-Cubic flow starts), Copa is able to utilize the



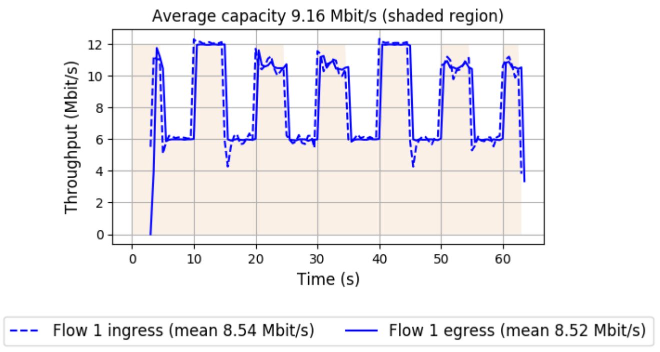
(a) Throughput.



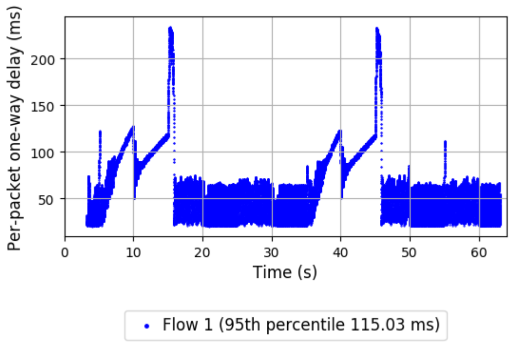
(b) Delay.

Figure 6.12: Throughput achieved by Copa and the corresponding queuing delay on a 20 Mbps link where Copa shares the link with TCP-Cubic. The blue line is the Copa flow, and the green line is the TCP-Cubic flow. The TCP-Cubic flow starts 5 seconds after the Copa flow.

entire link bandwidth while maintaining low delay. When the TCP-Cubic flow starts, Copa is able to attain it's fair share of the link since it switches to competitive mode.



(a) Throughput.



(b) Delay.

Figure 6.13: Throughput achieved by Copa and the corresponding queuing delay on a link that oscillates between 12 Mbps and 6 Mbps.

This enables Copa to achieve low delay when running in isolation, and to achieve it's fair share when competing with queue-building flows. Unfortunately, this mechanism is fragile, causing Copa to operate in the wrong mode [135] - this can cause occasional periods of very high delay when running in isolation. In Figure 6.13, we see two instances where Copa incorrectly switched modes to competitive mode, causing buffer bloat and high delays.

Nimbus [135] proposes an active measurement technique of the elasticity of the cross traffic by transmitting traffic in the form of sinusoidal pulses. In the presence of elastic cross-traffic (cross-traffic that responds to changes in the available link capacity), Nimbus uses a loss-based CCA like TCP-Cubic. When the cross-traffic is inelastic, Nimbus uses a delay-based congestion control mechanism. This enables Nimbus to achieve low delay when competing with inelastic flows (or in isolation), and high throughput when competing with elastic flows.

Unfortunately, these techniques are either very fragile, or work at longer timescales. In the case of cloud gaming, consistent performance is desirable, and thus, the CCA must be able to sustain stable throughput in the presence of short queue-building flows, and must achieve low delay when running in isolation over noisy, variable links.

6.1.4 Congestion Control for Cloud Gaming/AR/VR.

Congestion control algorithms designed for low-latency interactive streaming applications must be carefully designed in order to achieve the right performance trade-off. These are listed below in roughly the order of decreasing priority:

1. **Low end-to-end Frame Delay**: Minimize the video streaming delay by reducing in-network queuing and additional delays due to buffering at the sender and the receiver.
2. **High, stable throughput**: Achieve high throughput in the presence of queue-building flows and over links with time-varying throughput in order to achieve good video quality, without sacrificing frame delay.
3. **Video Awareness**: The congestion control algorithm must work well with the traffic pattern of streaming video by handling temporary frame-size overshoots and undershoots due to encoder variations. The algorithm must not transmit packets in a bursty manner, since this can cause video frames to get delayed at the sender due to the congestion control algorithm's flow control mechanism.

6.1.5 Congestion Control and Encoder Integration

As discussed in the sections above, a congestion control algorithm designed for emerging low-latency video streaming applications like cloud gaming and cloud AR/VR must at least have low in-network queuing delays and high bandwidth utilization. In addition to these properties, there are additional factors to consider regarding how the congestion control algorithm fits into the broader picture of video bitrate and frame delay.

In this section, we argue that low in-network queuing delay and high throughput are not sufficient in order to guarantee good QoE for cloud streaming applications. Understanding how the congestion control behavior is translated to the video bitrate, and how this integration affects the video quality and video delay is a key step in designing effective congestion control algorithms for cloud streaming applications.

Adaptive bitrate (ABR) algorithms are used in traditional Video-on-Demand (VOD) streaming systems in order to match the amount of video data sent to the network capacity. In traditional VOD systems, the video is pre-encoded into chunks at multiple bitrates. ABR algorithms can range from purely using buffer occupancy statistics (e.g. how much data has already been buffered at the client), to using a combination of buffer occupancy and network congestion control statistics like bandwidth and delay. These algorithms work at a chunk-level granularity, determining what the bitrate of the next chunk should be in order to ensure sufficient occupancy of the playback buffer for smooth video playback.

Real-time streaming applications encode video on-the-fly in real time, and hence, enable fine-grained control of the video bitrate. In the case of real-time streaming applications, the use of a playback buffer is detrimental to the video delay. While real-time streaming systems designed for latency tolerant applications like video conferencing and live broadcast use some degree of buffering in order to reduce frame jitter, this approach is not suitable for cloud streaming applications, since they require extremely low end-to-end frame delay. Thus, buffer-based ABR algorithms are not used in real-time video streaming systems that require low latency. Instead, these systems use real-time video bitrate control algorithms that translate the congestion control signals into the video encoder bitrate.

For latency-tolerant real-time video streaming systems like video conferencing and live broadcast, it is sufficient to have some kind of loose coupling between the congestion control algorithm and the video encoder bitrate. For example, one could use the average bandwidth in some small past window to set the video encoder bitrate. If the video bitrate is set according to the average bandwidth, the generated video data may occasionally be higher than the congestion control algorithm's instantaneous transmission rate. This means that frames can get delayed at the sender, since the congestion control algorithm is transmitting at a rate that is lower than the video bitrate. In order to achieve smooth playback, buffering can be used at the client side in order to absorb such delay variations. The benefit of this approach is that it allows the use of any video-agnostic congestion control algorithm that has a mostly steady transmission over windows of time on the scale of the acceptable video delay (e.g. 200 ms in the case of video conferencing).

Unfortunately, such loosely coupled encoder bitrate control mechanisms in conjunction with video-agnostic congestion control algorithms limit the QoE of ultra-low latency cloud streaming applications. One example is the case of BBR. BBR uses a minRTT probing mechanism in order to measure a baseline network round-trip value. During this phase, BBR significantly reduces the congestion window for a period of 200 ms in order to drain any built-up queues. This implies that none of the video frames generated during this 200 ms period can be transmitted, and instead, get buffered at the sender. In general, consider a scenario where the transmission rate of the congestion control temporarily drops for a short period of time for reasons other than the actual network bandwidth going down. In the case of BBR, this can happen because of BBR's min RTT probe. In the case of Copa, the transmission rate oscillates over a period of 5 RTTs. This can result in one of three things:

1. If the video is encoded at the average bitrate, and if no buffering is used at the client side, this would result in a progressive increase in the frame delay until the transmission rate goes back up, at which point the delay will slowly recover. In the case of BBR, where the transmission rate drops to almost zero, the video playback would stall every 10 seconds, which is unacceptable for cloud gaming or cloud AR/VR and is extremely detrimental to the QoE.
2. If the video is encoded at the average bitrate, and if sufficient buffering is used at the client side, the video playback would be smooth, but this would result in a much higher steady state video frame delay (equal to the size of the buffer). Cloud gaming and cloud AR/VR applications have strict low latency requirements, and the high video frame delay would harm the user's QoE.
3. If the video is encoded according to the instantaneous transmission rate, it would result in a drop in the video quality. These fluctuations in the video quality are harmful for cloud streaming applications, and must be avoided if possible (e.g. if the network bandwidth is

stable).

6.2 SQP overview.

For the end user experience of cloud gaming and cloud-rendered AR/VR to be comparable to running the applications locally, the video must be encoded at the highest bitrate that still allows the frames to be transmitted and received with minimal delay. A CCA for low-latency interactive video streaming must have the following properties:

1. *Low Queuing Delay:* The CCA must be able to probe for more bandwidth without causing excessive queuing, and must quickly back off when the available bandwidth decreases in order to reduce in-network queuing. CCAs like Cubic [12] fill up network queues until packet loss occurs, and some CCAs, like PCC [133], are slow to react to drops in bandwidth, resulting in very high delays that are unacceptable for low-latency interactive streaming.
2. *Link Utilization:* The CCA must achieve high, stable bandwidth when competing with queue-building flows (e.g., Cubic, BBR [11]), while achieving low delay when running in isolation. Some low-delay CCAs have explicit mechanisms to prioritize throughput over delay when queue-building cross traffic is detected, but they can be inherently unstable (e.g., Copa [21] can misdetect self-induced queuing as competing traffic, resulting in additional self-induced queuing [136]), while others are slow to converge (e.g., Nimbus [136] operates over 10s of seconds).
3. *Fairness:* Multiple homogeneous flows should converge to fairness quickly, and must be fair at frame-level timescales.
4. *Friendliness:* The CCA must be friendly to other CCAs and must avoid starving them. The maximum bandwidth for cloud gaming applications is typically capped to a maximum value - this causes the bottleneck to typically be near the last mile. Thus, perfect fairness with other CCAs is not necessary, and is not possible in general [137].
5. *Video Awareness:* The CCA must accommodate encoder frame size variations, and achieve bandwidth probing in application-limited scenarios without the need for frame padding. The CCA must use a rate-based congestion control mechanism to minimize the end-to-end frame delay - the bursty nature and time-varying throughput of window-based mechanisms necessitate an undesirable trade-off between bandwidth utilization, encoder rate-control updates, and the end-to-end frame delay.

While most CCAs aim to achieve high throughput, low delay, and competitive performance when competing with queue-building flows, simultaneously achieving these requirements is challenging in an environment as diverse as the Internet. Choosing the right trade-offs and correctly prioritizing the design requirements (listed above in decreasing order of priority) enables a design that is highly optimized for the specific application class. Existing CCAs make different trade-offs based on their particular design goals, and some of these design choices make them unsuitable for low-latency interactive streaming applications.

SQP is a novel congestion control algorithm that was developed in conjunction with Google's AR streaming platform. SQP's key features are listed below:

1. *Prioritizing Delay over Link Utilization:* Since delay is more critical for the QoE of low-latency interactive video streaming applications, SQP sacrifices peak bandwidth utilization when running in isolation in order to achieve low delay and delay stability. For example, on a 20 Mbps link where SQP is the only flow, it is acceptable to utilize 18 Mbps if this trade-off reduces delays across a wider range of scenarios.
2. *Application-specific Trade-offs :* SQP is designed for low-latency interactive streaming applications, which have specific requirements in terms of minimum bandwidth and maximum delay. If these parameters are outside the acceptable range due to external factors (e.g., poor link conditions, very high delays due to queue-building cross traffic), it is acceptable to end the streaming session. In contrast to traditional algorithms, SQP restricts its operating environment, which enables SQP to achieve acceptable throughput and delay performance across a wider range of relevant scenarios.
3. *Frame-focused Operation :* In-network queuing is a key mechanism that allows CCAs to detect the network capacity. CCAs that probe infrequently (e.g., PCC, GoogCC) have lower average delay, but suffer from link underutilization on variable links. SQP piggybacks bandwidth measurements onto each frame's transmission by sending each frame as a short (paced) burst, and updates its bandwidth estimate after receiving feedback for each frame. For low-latency interactive streaming applications, the QoE is determined by the end-to-end frame delay, and not just the in-network queuing delay. SQP network probing relies on queuing at the sub-frame level without increasing the end-to-end frame delay, and is able to adapt to changes in network bandwidth much faster than protocols like PCC [133, 134] and GoogCC [138].
4. *Direct Video Bitrate Control :* SQP uses frame-level bitrate changes in order to respond to congestion, and drains self-induced queues by reducing the video bitrate. SQP's rate-based congestion control minimizes the end-to-end frame delay compared to protocols that are window-based (Copa), or throttle transmissions for network RTT measurements (BBR).
5. *Competitive Throughput :* SQP's bandwidth probing and sampling mechanism is competitive by default, and achieves high, stable throughput share when competing with queue-building flows that cause delays within an acceptable range. SQP avoids high queuing delays and starving other flows using mechanisms like adaptive one-way delay measurements, a bandwidth target multiplier, and frame pacing. SQP's design avoids the pitfalls of delay-based CCAs that use explicit mode switching (e.g., Copa [21, 136]).

SQP's evaluation on real-world wireless networks for Google's AR streaming platform, and across a variety of emulated scenarios, including real-world Wi-Fi and LTE traces show that:

1. Under A/B testing of SQP and Copa[1] on Google's AR streaming platform across ≈ 8000 individual streaming sessions, 71% of SQP sessions on Wi-Fi have P50 bitrate > 3 Mbps and P90 frame RTT < 100 ms, compared to 56% for Copa. On cellular links, 36% of SQP sessions meet the criteria versus only 9% for Copa.
2. Across emulated wireless traces, SQP's throughput is 11% higher than Copa (without mode switching) with comparable P90 frame delays, while Copa (with mode switching), Sprout [22], and BBR incur a 140-290% increase in the end-to-end frame delay relative to SQP.

[1]Adapted from mvfst [139], does not implement Copa's mode switching.

3. SQP achieves high and stable throughput when competing with buffer-filling cross traffic. Compared to Copa (with mode switching), SQP achieves 70% higher P10 bitrate when competing with Cubic, and 36% higher link share when competing with BBR.

This work was done in collaboration with Google. My contributions include the theoretical analysis of the impact of pacing and target multipliers on SQP, video encoder undershoot correction, SQP's adaptive min-oneway delay measurement window, analysis of SQP's update rule and fairness, and extensive evaluation of SQP and other CCAs.

6.3 Related Work

Protocol Category	**Congestion Detection**	**Competing with Queue-building Flows**	**Congestion Control Mechanism**	**Comments**
Explicit signaling DCTCP, ABC, XCP	Explicit signals from network to detect congestion	Compete with homogeneous flows	Various	Lack of support, traffic heterogeneity - unsuitable for Internet-based interactive video streaming
Low Delay TCP-Lola, TCP-Vegas, Sprout (Salsify), TIMELY, Swift	Packet delay/delay-gradient, stochastic throughput forecast (Sprout)	Queue-building flows cause low throughput	Window-based, Rate-based (TIMELY), hybrid (Swift)	High, stable throughput required - not achieved with queue-building cross-traffic, custom encoder for handling bursty CCA (Salsify)
Mode Switching Copa, Nimbus, GoogCC (WebRTC),	Packet delay/delay-gradient as congestion signal	More aggressive when competing flow detected	Window-based, Rate-based (GoogCC)	Mode-switching is unstable (Copa, GoogCC), can be slow to converge (Nimbus, GoogCC)
Model-Based BBR	minRTT probe, pacing gain for bandwidth	Designed to be competitive with Cubic	Rate- and window-based	200ms minRTT probe throttles transmissions, 2 BDP in-flight under ACK aggregation/competition
Utility-based PCC, PCC-Vivace	Explicit probing, delay, packet loss	Measure network response to rate change	Rate-based	Inconsistent performance with queue-building flows, slow convergence on dynamic links

Table 6.1: Various CCAs that exist today, and their properties.

A suitable congestion control algorithm for low-latency interactive video streaming must satisfy the four key properties discussed in § 6.2. Various CCAs are summarized in Table 6.1.

Low-latency CCAs like TCP-Lola [140], TCP-Vegas [141], and Sprout (Salsify[2]) [14, 22] that use packet delay as a signal have a key drawback: they are unable to achieve high throughput when competing with queue-building cross-traffic. Some mode switching algorithms (e.g., Copa [21]) can misinterpret self-induced queues as competing flows, resulting in high delays, whereas other CCAs like Nimbus [136], and GoogCC (WebRTC) [138, 142] converge slowly, and can have unstable throughput when competing with queue-building flows.

BBR [11] periodically throttles transmissions to measure a baseline delay, which is problematic for interactive video streaming since frames cannot be transmitted during its minRTT probe. Window-based protocols have a similar problem - they transmit packets in bursts, and a mismatch between packet transmissions and frame generation require sender-side buffering, and increase the end-to-end frame delay.

Utility-based algorithms like PCC [133, 134] explicitly probe the network and aim to maximize a utility function based on throughput, delay, and packet loss. These CCAs converge slowly on dynamic links, and have inconsistent performance when competing with queue-building flows.

CCAs use rate-based or window-based mechanisms in order to control the transmission rate under congestion. Rate-based CCAs are better suited for video streaming due to the burst-free nature of packet transmissions, whereas the bursty window-based mechanisms can block frame transmissions at the sender and make encoder rate-control challenging (e.g., Salsify-Sprout). The other benefit of rate-based CCAs is that their internal bandwidth estimate can be used to directly set the video bitrate, whereas window-based mechanisms require additional mechanisms for setting the video bitrate.

Some CCAs require explicit signals from the network and specialized hardware, or work under tightly controlled environments (e.g. DCTCP [143], ABC [144], XCP [145], TIMELY [146], Swift [147]) - these are not applicable for Internet video streaming applications since explicit signaling mechanisms are not currently widely deployed on the Internet.

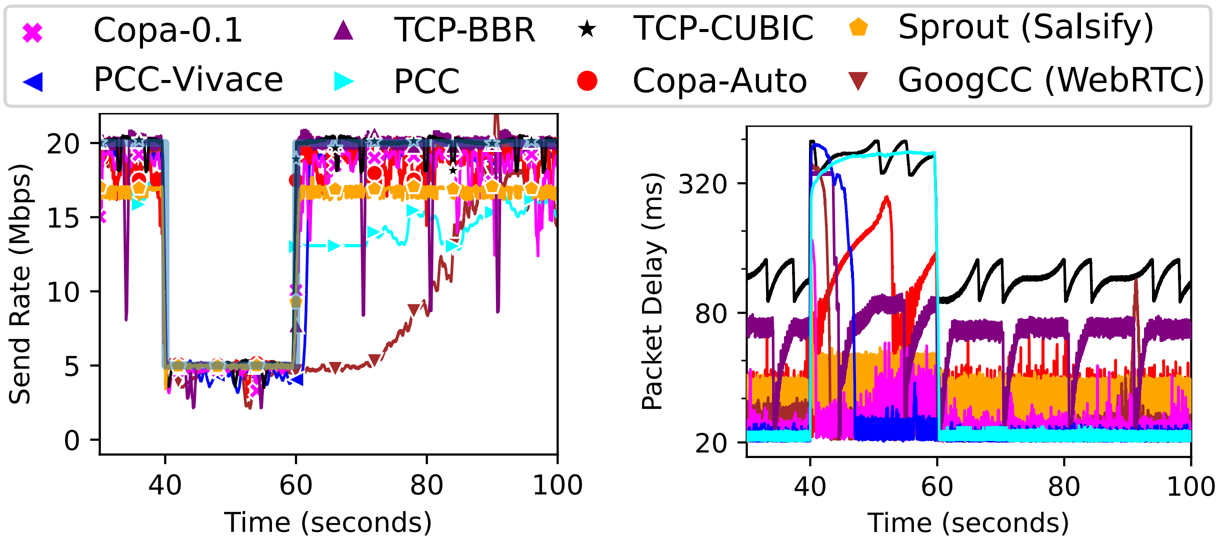
6.4 Preliminary Study

To illustrate the shortcomings of existing congestion control algorithms in the context of low-latency interactive streaming, we present some preliminary experimental results using the Pantheon [3, 130] testbed. Details regarding the specific CCA implementations are provided in § 6.7.1.

6.4.1 Variable Bandwidth Link

We ran a single flow for 120 seconds over a 40 ms RTT link, where the bandwidth temporarily drops down to $5\,\mathrm{Mbps}$ from $20\,\mathrm{Mbps}$. The goal of this experiment is to see if the algorithm can (1) quickly discover additional bandwidth when the available bandwidth increases, and (2) maintain low delay when the available bandwidth decreases.

[2]Salsify streamer uses Sprout as the CCA (used interchangeably)



(a) Send rate timeseries.

(b) Packet delay timeseries.

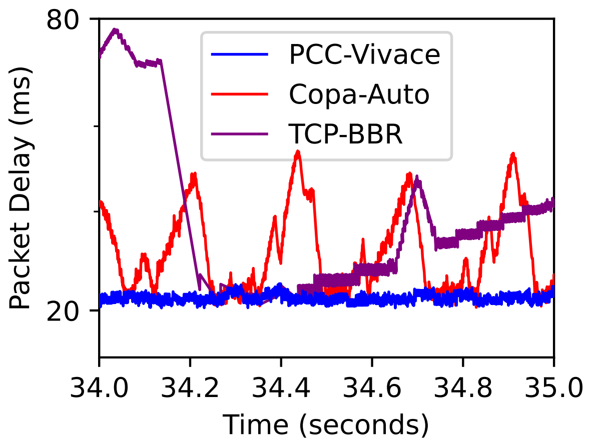
Figure 6.14: Congestion control performance on a variable link (link bandwidth shown as a shaded light blue line).

Figure 6.14 shows the throughput and delay timeseries between $t = 30s$ and $t = 100s$.

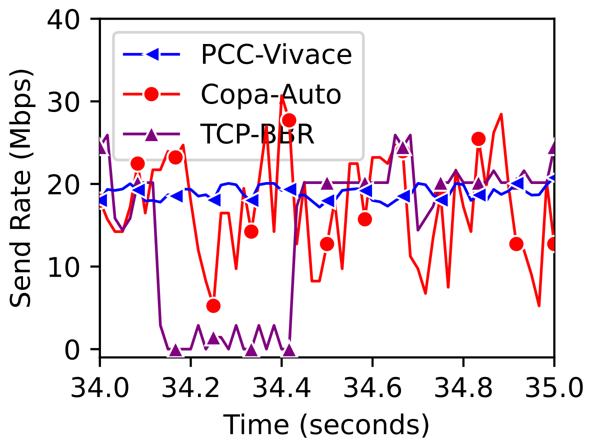
Throughout the entire trace, Cubic operates with the queues completely full, resulting in high link utilization at the cost of high queuing delays. Among the low-delay algorithms, the delay performance differs greatly across specific algorithms. When the link rate goes down to $5\,\mathrm{Mbps}$, Sprout and Copa-0.1 (Copa without mode switching, $\delta = 0.1^3$) are able to adapt rapidly without causing a delay spike. PCC-Vivace [134], GoogCC and BBR are slower to adapt, causing 3-8 seconds long delay spike. Throughout the low bandwidth period, PCC maintains persistent, high queuing delay, whereas Copa-Auto (Copa with mode-switching, adaptive δ) incorrectly switches to competitive mode, significantly increasing queuing delay.

In addition to the delay that occurs when the link rate drops, some algorithms have inherently more queuing than others. BBR can maintain up to 2 BDP in-flight, causing up to 1 BDP of in-network queuing. Both, Copa-0.1 and Copa-Auto demonstrate significant short-term delay variations due to Copa's 5-RTT probing cycle, which serves the role of probing the network for additional capacity. The peak delay is inversely proportional to the value of δ, and is worse in the case of Copa-Auto, since it periodically misinterprets its own delay as delay caused due to a competing queue-building flow, and consequently reduces the value of δ in response. There is significant variation in Sprout's packet delay due to the bursty nature of its packet transmissions, even though it is significantly underutilizing the link. GoogCC also demonstrates a delay spike around $t = 90s$, when its send rate hits the link limit after an extended ramp-up period. PCC-Vivace, Copa-Auto, Copa-0.1, and BBR are able to rapidly probe for more bandwidth when the link rate increases. On the other hand, PCC and GoogCC are the slowest to converge, taking more than 20-30 seconds to ramp up after the link rate increases, resulting in severe underutilization.

[3] A lower delta makes Copa more aggressive, sacrificing low delay for higher throughput. The original paper proposes using 0.5, whereas the version on Pantheon uses 0.1. Facebook's testing of Copa [148] used 0.04.



(a) Short-term delay variation.



(b) Short-term send rate variation.

Figure 6.15: A closer look at the short term delay and send rate variation on a constant 20 Mbps link.

In order to achieve high link utilization and low delays for low-latency interactive video streaming, the CCA must quickly identify the link capacity without causing queuing delays, and quickly back off when the delay is self-induced. SQP is able to achieve these requirements, as shown in Section 6.7.3.

6.4.2 Short Timescale Variations

In this section, we examine the short timescale behavior of existing protocols to see if they can provide the low packet delay and stable throughput [149] needed to support the requirements of low-latency streaming applications. We present three algorithms that demonstrate distinct short-term behavior: Copa-Auto, BBR and Vivace (additional results in Section 6.7.7). Copa-0.1 and Sprout behave similar to Copa-Auto, and PCC behaves similar to Vivace in these experiments. We ran each algorithm on a fixed 20 Mbps link with 20 ms of delay in each direction. Figure 6.15a shows the one-way packet delays and Figure 6.15b shows the packet transmission rate for each frame period (16.66 ms at 60FPS).

Copa-Auto's one-way packet delay oscillates between 20 ms and 60 ms over a 12-frame period, with large variations in the send rate at frame-level timescales. If a smooth video bitrate is determined using the average send rate to maximize utilization, the frames at $T = 34.3,\ 34.5$ would get delayed at the sender. To lower the sender-side queuing delay, the encoder rate selection mechanism must either: (1) choose a conservative video bitrate, resulting in underutilization, or (2) have frequent rate control updates.

While BBR is not particularly suitable for interactive streaming because of its higher queuing delay, BBR's RTT probing mechanism is especially problematic. Every 10 seconds, BBR throttles transmissions (transmitting at most 4 packets per round trip) for 200 ms to measure changes in the link RTT (between $T = 34.2$ and $T = 34.4$). During this period the generated video frames will be queued at the sender, resulting in 200 ms of video stutter every 10 seconds.

Rate-based algorithms like PCC and Vivace are better suited for streaming applications, since the internal rate-tracking mechanism can be used to set the video bitrate, and frames are not delayed at the sender if the encoded frames do not overshoot the requested target bitrate. While Salsify [14] attempts to solve this problem using a custom encoder that can match the instantaneous transmission rate of a bursty CCA like Sprout [22], rearchitecting the CCA is a more universal and practical solution that can leverage advances in hardware video codecs that have good rate control mechanisms.

To minimize the end-to-end frame delay, the CCA must transmit encoded frames immediately, and pace faster than the rate at which the network can deliver the packets. SQP directly controls the video bitrate using smooth bandwidth estimates, and the transmissions are synchronized with the frames, which reduces the end-to-end frame delay (Section 6.7.7).

6.5 Design

Low-latency interactive streaming applications generate raw frames at a fixed frame-rate. The video bitrate is determined by an adaptive bitrate (ABR) algorithm using signals from the CCA in order to manage frame delay, network congestion, and bandwidth utilization. The compressed frames are transmitted over the network, and eventually decoded and displayed at the client device.

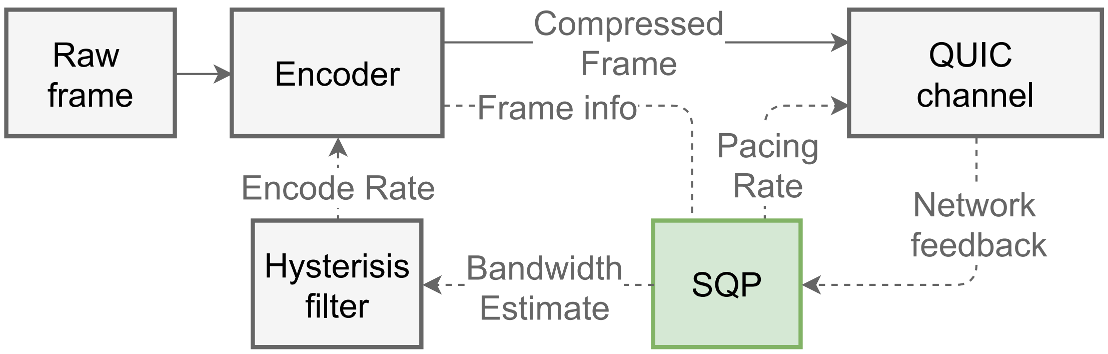
SQP is a rate- and delay-based CCA for low-latency interactive video streaming, and aims to (1) provide real-time bandwidth estimates that ensure high utilization and low end-to-end frame delay on highly variable links, and (2) achieve competitive throughput in the presence of queue-building cross traffic. SQP's congestion control mechanism must be purely rate-based in order to avoid the undesirable trade-offs between bandwidth utilization, encoder bitrate changes, and the end-to-end frame delay (§ 6.2).

6.5.1 Architecture Overview

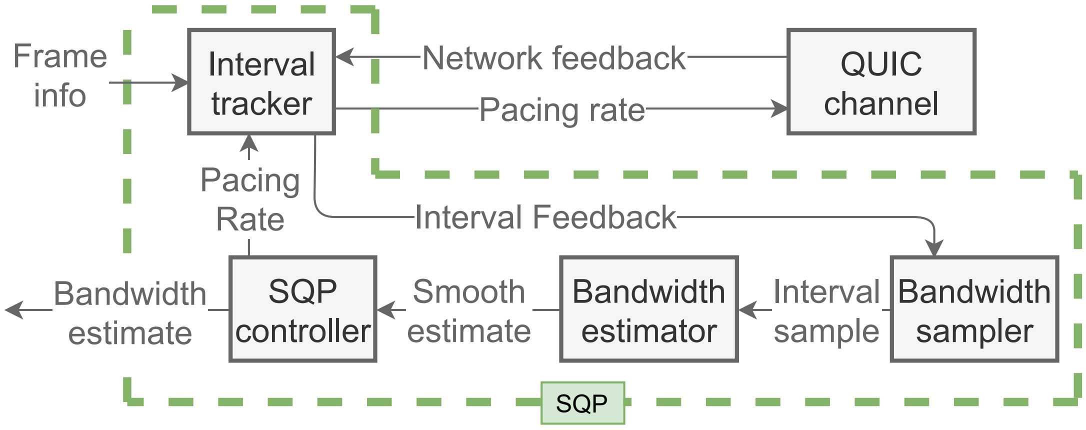
SQP's role in the end-to-end streaming architecture and its key components are shown in Figures 6.16a and 6.16b. SQP relies on QUIC [150] to reliably transmit video frames, perform frame pacing, and provide packet timestamps for estimating the network bandwidth. SQP directly controls the video bitrate, and a simple hysteresis filter serves as a bridge between SQP and the encoder to reduce the frequency of bitrate changes.

Internally, SQP's components work together in order to achieve the key design goals:

1. *Bandwidth Probing:* SQP transmits each frame as a short, paced burst, and the bandwidth sampler uses frame-level packet dispersion statistics from the interval tracker for discovering additional bandwidth.
2. *Recovery from Transient Queues:* SQP's bandwidth samples are penalized when the delay increases over a short period (§ 6.5.2), enabling it to recover from transient self-induced queuing.
3. *Recovery from Standing Queues:* SQP uses a target multiplier mechanism (§ 6.5.5) to maintain some slack in the link utilization, enabling recovery from self-induced standing queues. SQP remains competitive when competing flows cause standing queues (within

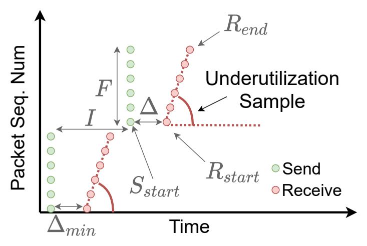


(a) Video streaming architecture.

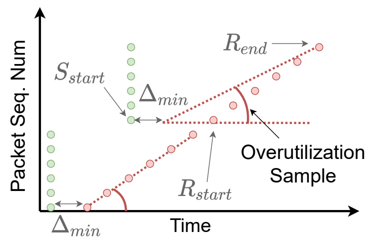


(b) SQP internal architecture.

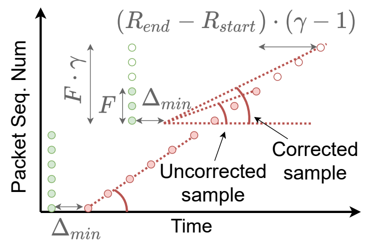
Figure 6.16: Low-latency video streaming and SQP architectures.



(a) SQP's bandwidth samples are higher than the video bitrate when the link is underutilized, indicating that the video bitrate should be increased.



(b) SQP's bandwidth samples are lower than the link rate when the link is overutilized, indicating that the video bitrate should be reduced.



(c) Bandwidth samples from frames that are smaller than the target bitrate are more sensitive to transient queuing. SQP's corrected bandwidth sample is closer to the link rate than the uncorrected sample.

Figure 6.17: SQP's bandwidth samples converge towards the link rate and aid in draining self-inflicted queues. The slope of the dotted red line represents the bandwidth sample in each case.

acceptable delay limits) since it uses a small, dynamic window to track the transient delay (§ 6.5.3).

4. *Rate-based congestion control:* SQP aims to carefully pace each frame faster than the rate at which the network can deliver the packets (§ 6.5.5), and responds to congestion by smoothly changing the bandwidth estimate (and consequently, the video bitrate) using gradient-based updates (§ 6.5.4). As opposed to using ACK-clocking and window-based mechanisms, rate-based congestion control simplifies integration with the video encoder and reduces the end-to-end frame delay (§ 6.4.2).
5. *Fairness and Interoperability:* SQP's bandwidth estimator (§ 6.5.4) is based on maximizing a logarithmic utility function, which improves dynamic fairness due to its AIMD-style updates (§ 6.6.2). SQP's frame pacing and the bandwidth target multiplier mechanisms ensure dynamic fairness across multiple SQP flows (§ 6.6.2) and provide a theoretical upper bound on SQP's share when competing with elastic flows (§ 6.6.1).

For the initial part of the discussion, we will assume the existence of a 'perfect' encoder with the following properties: (1) the target bitrate can be changed on a per-frame basis without any negative consequences, as long as the target bitrate does not change significantly from frame to frame, and (2) the encoder does not overshoot or undershoot the specified target rate. In § 6.7.10, we discuss how SQP works in a practical setting when encoders do not satisfy some of these assumptions.

6.5.2 Bandwidth Sampling

The goal of SQP's bandwidth sampling algorithm is to measure the end-to-end frame transport rate that achieves high link utilization while avoiding self-induced queuing and packet transmission pauses (e.g., Copa and BBR in § 6.4.2). SQP transmits each frame as a short burst that is faster than the network delivery rate, which causes a small amount of queuing. This queue is drained by the time the next frame arrives at the bottleneck if the average video bitrate is lower than the available bottleneck link capacity. SQP uses the dispersion of the frame-based packet train [151] to measure link capacity, with some key differences that aid in congestion control compared to basic packet-train techniques. SQP's probing works at sub-frame timescales ($< 16.66\ \mathrm{ms}$ @ 60FPS), in contrast to CCAs that probe for bandwidth over longer timescales (PCC:2RTT, BBR:1 min-RTT, and Copa:2.5 RTT).

Consider an application generating frames at a fixed frame rate. As shown in Figure 6.17, a frame of size F is transmitted every inter-frame time interval, I (e.g., 16.66 ms at 60FPS), and the average bitrate is $\frac{F}{I}$. Each frame is paced at a rate that is higher than $\frac{F}{I}$ (shown by the steep slope of the green dots). If there are no competing flows (competition scenario discussed in § 6.6.1), and the link bandwidth is lower than the pacing rate, the packets will get spaced out according to the bandwidth of the bottleneck link (slope of the red dots). SQP computes the end-to-end frame transport bandwidth sample as:

$$S = \frac{F}{R_{end} - S_{start} - \Delta_{min}} \tag{6.1}$$

This is the slope of the red dotted line in Figure 6.17. S_{start} and R_{end} are the send and receive times of the first and last packets of a frame, respectively, and Δ_{min} is the minimum one-way delay (delta between send and receive timestamps) for packets sent during a small window in the past (§ 6.5.3). Δ_{min} and $R_{end} - S_{start}$ have the same clock synchronization error (sender-side vs. receiver-side timestamps) and cancel each other out.

Underutilization Sample: When the network is underutilized or 100% utilized, no additional queuing occurs across multiple frames ($\Delta = R_{start} - S_{start} = \Delta_{min}$ remains constant). Thus, as shown in Figure 6.17a, the sample is equal to the packet receive rate for a frame (ie. the bottleneck link bandwidth). The samples during link underutilization are higher than the video bitrate ($\frac{F}{I}$), and SQP increases its bandwidth estimate. When the link is 100% utilized, the samples are equal to the video bitrate, indicating good link utilization.

Overutilization Sample: Transient overutilization due to frame size overshoots, bandwidth overestimation (link aggregation, token bucket policing), or a drop in network bandwidth can cause queuing that builds up across multiple frames. This results in an increase in $\Delta - \Delta_{min}$ for subsequent frames, which lowers the bandwidth samples for subsequent frames (Figure 6.17b, slope of dotted red line for the second frame). Thus, SQP lowers the video bitrate below the link rate and recovers from transient queuing. When packets are lost, SQP scales down its samples by the fraction of lost packets, primarily responding to sustained loss events (e.g., shallow buffers, § 6.7.6).

Video Encoder Undershoot: While SQP is also able to discover the link bandwidth quickly in application-limited scenarios since it relies on the pacing burst rate, and not the average video bitrate, bandwidth samples from small frames are unfairly penalized due to delay variations. SQP often has to deal with application-limited scenarios where the bitrate of the encoded video is less than the bandwidth estimate. This can be due to conservative rate control mechanisms that serve as a bridge between the bandwidth estimate and the encoder bitrate to reduce the frequency of encoder bitrate updates, or due to encoder undershoot during low complexity scenes that do not warrant encoding frames at the full requested target bitrate (eg. low-motion scenes like menus). When SQP is application-limited, queuing delay from past frames can unfairly penalize the bandwidth sample (Figure 6.17c). While padding bytes can be used to bring up the video bitrate to SQP's bandwidth estimate, this results in wastage of bandwidth. To improve SQP's robustness under application-limited scenarios, we modify the bandwidth sampling equation to account for undershoot:

$$S = \frac{F \cdot \gamma}{R_{end} - S_{start} - \Delta_{min} + (R_{end} - R_{start}) \cdot (\gamma - 1)} \tag{6.2}$$

where $\gamma = \frac{F_{max}}{F}$ is the undershoot correction factor, F_{max} is the hypothetical frame size without undershoot, and $(R_{end} - R_{start}) \cdot (\gamma - 1)$ is the predicted additional time required for delivering the hypothetical full-sized frame. This computes the bandwidth sample by extrapolating the delivery of a small frame to the full frame size that is derived from SQP's current estimate. In Figure 6.17c, the solid dots are actual packets for a frame, and the hollow dots show the extrapolated transmission and delivery of the packets.

6.5.3 Tracking Minimum One-way Delay

The minimum packet transmission delay, Δ_{min}, serves as a baseline for detecting self-inflicted network queuing. The window size for tracking Δ_{min} represents the duration of SQP's memory of Δ_{min}, which affects SQP's self-induced queuing and throughput when competing with queue-building cross traffic. If a small, fixed window were used (e.g., 0.1-0.5 s), when self-induced queuing occurs, Δ_{min} could expire before SQP can recover. While a larger, fixed window (e.g., 10-30 s) would aid recovery from self-induced queues by anchoring SQP to the lowest one-way delay observed over the window, SQP's bandwidth samples would be more sensitive to delay variations caused by the queue-building cross traffic, lowering SQP's throughput share.

To balance these trade-offs, SQP uses an adaptive window size of $2 \times \text{sRTT}$ [152]. This has two advantages. First, for self-induced queuing, SQP's adaptive window grows quickly, and in conjunction with SQP's bandwidth target multiplier mechanism (§ 6.5.5), enables SQP to drain self-induced queues. Second, when competing queue-building flows build standing queues, Δ_{min} quickly increases in response, so that SQP doesn't react to the competitor's standing queue. Since SQP paces frames into the network faster than SQP's current share, it can probe for more bandwidth when competing with queue-building flows, even if the combined link utilization of SQP and the cross traffic is near 100%. Together, this enables SQP to obtain a high throughput share when those queue-building flows (1) do not cause very high delays, and (2) have low queuing delay variation over periods of $2 \times \text{sRTT}$ (§ 6.7.5).

While the role of SQP's Δ_{min} mechanism is similar to BBR's minRTT mechanism, SQP does not need an explicit probing mechanism for Δ_{min} since it (1) increases the window size when self-induced queuing occurs, and (2) reduces the video bitrate to drain the self-induced queue, which provides organic stability. While BBR's explicit probing of the baseline network RTT is more accurate, the need to significantly limit packet transmissions for 200 ms makes this approach unsuitable for real-time interactive streaming media. We evaluate the impact of the window size scaling parameter in § 6.6.3.

6.5.4 Bandwidth Estimate Update Rule

SQP's bandwidth estimator processes noisy bandwidth samples measured by SQP's bandwidth sampler to provide a smooth bandwidth, which is used to set the video encoder bitrate. SQP's update rule is inspired from past work on network utility optimization [153], and is derived by optimizing a logarithmic reward for higher bandwidth estimates and a quadratic penalty for overestimating the bandwidth:

$$\max \log(1 + \alpha \cdot B) - \beta \cdot (B - e)^2 \tag{6.3}$$

where B is SQP's bandwidth estimate, α is the reward weight for a higher bandwidth estimate, β is the penalty for overestimating the bandwidth, and e is a parameter derived from the bandwidth sample S, such that the function is maximized when $B = S$. Taking the derivative of this expression and evaluating the expression with B set to the current estimate provides a gradient step towards the maxima. Simplifying the derivative of the expression 6.3, and the constant expressions involving α and β, the update rule can be rewritten as

$$B' = B + \delta \left(r \left(\frac{S}{B} - 1 \right) - \left(\frac{B}{S} - 1 \right) \right) \tag{6.4}$$

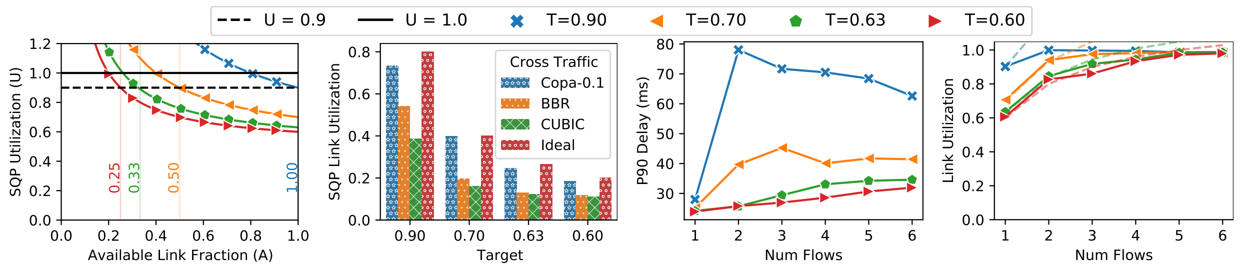
B' and B are the updated and current estimates, r is the reward weight for bandwidth utilization and δ is the step size and represents a trade-off between the smoothness of the bandwidth estimate and the convergence time under dynamic network conditions. SQP empirically sets $\delta = 320$ kbps, and $r = 0.25$.

SQP's target and pacing multiplier mechanisms (§ 6.6.1) work in conjunction with the update rule to improve SQP's convergence to fairness (§ 6.20).

6.5.5 Pacing and Target Multipliers

SQP's design includes two key mechanisms for ensuring friendliness with other flows - instead of transmitting each frame as an uncontrolled burst at line rate, SQP paces each frame at a multiple of the bandwidth estimate, and targets a slightly lower video bitrate than the samples (determined by a target multiplier). Suppose SQP is sharing a bottleneck link with a hypothetical elastic CCA [136] that perfectly saturates the bottleneck link without inducing any queuing delay. If SQP transmitted frames as uncontrolled bursts, the elastic flow might not be able to insert any packets between SQP's packets. Thus, the bandwidth samples would match the link rate, and SQP would starve the elastic flow by utilizing the entire link bandwidth.

SQP paces each frame at a rate P, which is a multiple of the current bandwidth estimate, ie. $P = m \cdot B$ ($m > 1.0$). Thus, each frame is transmitted over $\frac{I}{m}$, where I is the frame interval. While pacing enables competing traffic to disperse SQP's packets, SQP's bandwidth samples would still be higher than the average rate it is sending at, and SQP would eventually starve the other flow. To avoid this problem, SQP combines frame pacing with a bandwidth target multiplier mechanism. SQP multiplies bandwidth samples with a target multiplier $T < 1$ before calculating the bandwidth estimate. SQP's target multiplier serves three key roles: (1) it allows SQP to drain any self-inflicted standing queues over time, (2) in conjunction with the pacing



(a) Theoretical utilization of available capacity.

(b) Single SQP flow throughput when competing with cross traffic.

(c) P90 packet delay for SQP flows sharing a bottleneck.

(d) Total link utilization for SQP flows sharing a bottleneck.

Figure 6.18: Impact of the target multiplier on delay, link utilization and link share obtained under cross traffic for a pacing multiplier $m = 2$. Experimental results validate the theoretical analysis. In each case, the bottleneck link rate was 20 Mbps, the one-way delay in each direction was 20 ms and the bottleneck buffer size was 120 ms.

multiplier, it prevents SQP from starving competing flows, and (3) enables multiple SQP flows to converge to fairness. We empirically set $m = 2$ and $T = 0.9$, and analyze the impact of other values of T in Section § 6.6.1.

6.6 Analysis of SQP Dynamics

6.6.1 Competing Flows

SQP's pacing multiplier (m) and bandwidth target multiplier (T) mechanisms provide important guarantees that prevent SQP from starving other flows, and enable SQP to achieve fairness when competing with other SQP flows. In this section, we derive SQP's theoretical maximum share when competing with elastic flows, and the conditions under which SQP achieves queue-free operation when competing with inelastic flows. This analysis provides valuable insight into how SQP's parameters can be tuned for application-specific performance requirements.

SQP adds $B \cdot I$ bytes (i.e., the frame size, equal to the bandwidth estimate times the frame interval) to the bottleneck queue over a period $\frac{I}{m}$ (§ 6.5.5), during which a competing flow transmitting at a rate R adds $\frac{R \cdot I}{m}$ bytes to the queue. Thus, the time to drain the queue (T_d) is

$$T_d = \frac{B \cdot I + \frac{R \cdot I}{m}}{C} \tag{6.5}$$

where C is the link capacity. If the link is not being overutilized (Δ_{min} remains constant),

$$T_d = R_{end} - R_{start} = R_{end} - S_{start} - \Delta_{min} \tag{6.6}$$

From Eq. 6.6 and Eq. 6.1, SQP's bandwidth sample (S) can be written as $S = \frac{B \cdot I}{T_d}$. After substituting the value of T_d from Eq. 6.5 and simplifying the equation, we get:

$$S = \frac{C}{1 + \frac{R}{m \cdot B}} \tag{6.7}$$

Note that we assume $m \cdot B + R > C$ (link is not severely underutilized), otherwise no queue will build up during the SQP's pacing burst, and the bandwidth sample would simply be $m \cdot B$. SQP multiplies the bandwidth sample (S) with a target multiplier (T) before it is used to update the current bandwidth estimate using Eq. 6.4. Steady state occurs when the bandwidth estimate (B) is equal to the bandwidth target, ie. $B_T = S \cdot T$. Substituting S from Eq. (6.7), we get:

$$B = C \cdot T - \frac{R}{m} \tag{6.8}$$

If $A = \frac{C-R}{C}$ is the fraction of the link capacity available for SQP, and $U = \frac{B}{C-R}$ is SQP's utilization of the of the available link capacity, Eq. 6.8 can be re-written as

$$U = \frac{m \cdot T + A - 1}{m \cdot A} \tag{6.9}$$

This equation predicts SQP's behavior in a variety of scenarios. Figure 6.18a plots U on the Y-axis as a function of A on the X-axis for various target multipliers and for a pacing multiplier of 2.

Recovery From Self-Induced Queuing When SQP is the only flow on a bottleneck link, the available link share $A = 1$ (right edge of Figure 6.18a). This implies that SQP will always underutilize the link slightly (specifically, it will use fraction T of the total link capacity), which will result in standing queues getting drained over time. The value of T caps SQP's maximum link utilization in the steady state, and determines how quickly SQP will recover from self-induced standing queues. In our evaluation and for SQP's deployment in Google's AR streaming service (§ 6.8), we use a target multiplier of 0.9, which achieves good link utilization and is able to drain standing queues reasonably well.

Inelastic Cross Traffic. When SQP competes with an inelastic flow (transmitting at a fixed rate), the available link fraction (A) is fixed. For SQP to operate without any queuing, U (SQP's utilization of the available capacity) must be less than 1. Thus, the available bandwidth must be greater than the value at which the utilization curve crosses $U = 1$ in Figure 6.18a.

For example, with a pacing multiplier of 2 and a target multiplier of 0.9, SQP requires at least 80% of the link to be available so that it can consistently maintain a slight underutilization of the link. When less than 80% of the link is available, SQP will tend to overutilize its share and cause queuing. While SQP's initial window size for tracking Δ_{min} (6.5.3) may not be large enough for SQP to completely drain the self-induced queue, the increase in the RTT due to queuing will eventually cause the window to grow to a size that is large enough to stabilize SQP's queuing. While a smaller target value would enable SQP to operate without queuing for lower values of A, it would sacrifice link utilization when there are no competing flows.

Elastic Cross Traffic. The minimum value of A for queue-free operation of SQP when competing with inelastic flows is also the maximum bound for SQP's share of the throughput when it is competing with elastic traffic. When SQP is not using its entire share ($U < 1$), the elastic flow will increase its own share since the link is underutilized. This reduces the available link share for SQP, moving the operating point to the left in Figure 6.18a until the entire link is utilized ($U = 1$). If SQP is over-utilizing its share, the elastic flow will decrease its own share and move the operating point to the right until the link is no longer being over-utilized.

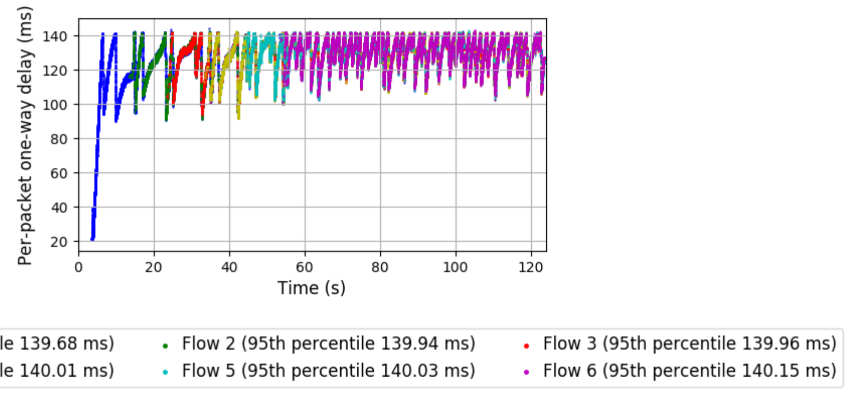


Figure 6.19: Cubic delay variation increases with more cubic flows.

This is an upper-bound of SQP's share. Non-ideal elastic flows can cause queuing delays that will cause SQP to increase its one-way delay tracking window, whch in turn will make the bandwidth samples more sensitive to delay variations caused by the cross traffic. Figure 6.18b shows the share of a single SQP flow competing with various elastic flows, for different target multipliers. Copa-0.1 closely resembles an ideal elastic flow which does not cause queuing and has low delay variation. Hence, SQP's share (shown in blue) is close to the theoretical maximum (shown in red). With BBR and Cubic, SQP's share is less than the theoretical maximum since the higher delays induced by BBR and Cubic make SQP more reactive to queuing delay variations.

Heterogeneous fairness In general, it is not possible for heterogeneous CCAs to achieve fairness when competing with each other. Let's consider the fairness behavior of BBR. We ran a simple experiment where BBR flows compete with Cubic flows. When one BBR flow competes with one Cubic flow on a 20 Mbps link, each flow achieves approximately 50% of the link capacity. When 4 BBR flows compete with a single Cubic flow, each flow achieves approximately 20% of the link, which is also fair. The third case, which is interesting, is the case when a single BBR flow competes with 4 Cubic flows. In this case, BBR achieves approximately 40% of the link capacity, and each Cubic flow only achieves 15% of the link capacity [154]. The key issue here is that BBR does not account for loss rate, which is the key signal used by Cubic.

In contrast BBR's fairness behavior, SQP's throughput also depends on the queuing and delay variations. In the particular case of SQP competing with Cubic, where one SQP flow competes with multiple Cubic flows, the delay variation caused by Cubic increases with the number of Cubic flows. This is shown in Figure 6.19, where 6 cubic flows start with an interval of 10 seconds between them. The excess delay variation reduces SQP's share as the number of Cubic flows increase, which results in better heterogeneous fairness compared to BBR.

6.6.2 Intra-protocol Dynamics and Fairness

From the analysis in § 6.6.1, we can also infer the number of SQP flows that can operate without queuing on a shared bottleneck, with some caveats. The underlying assumption in § 6.6.1 is that packet arrivals at the bottleneck are evenly spaced. The analysis also holds in the case of Poisson arrivals since the bandwidth samples are smoothed out by the update rule (§ 6.5.4). Multiple SQP flows transmit frames as regularly spaced bursts, and thus, the packet dispersion observed by one SQP flow depends on how its frames align with the frames of the other flows. If the two flows that are sharing the bottleneck have perfectly aligned frame intervals, each flow will observe exactly half of the link rate, and they will operate without queuing since $T < 1$. If the frame intervals are offset exactly by $I/2$ (when pacing at 2X), each flow will see the full link bandwidth until link overutilization triggers SQP's transient delay recovery mechanism. When the intervals are offset by $I/4$, the packet dispersion is the same as the dispersion caused by a uniform flow. Note that this is only a concern if there are very few SQP flows, and the applications have perfectly timed frames. As the number of SQP flows increase, the aggregate traffic pattern gets smoothed out.

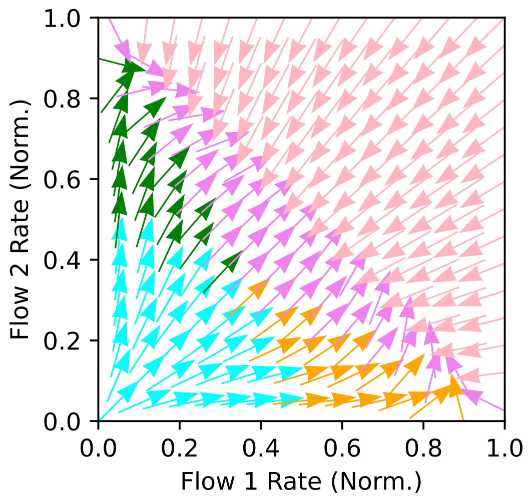
Figures 6.18c and 6.18d show the 90th percentile delay and the total link utilization respectively on the Y-axis as a function of the number of flows for various target multipliers. Figure 6.18d plots the theoretical link utilization of multiple SQP flows using dashed lines. The pacing multiplier was set to 2.0 for all runs. To avoid the impact of frame alignment in our experiment, we incorporate 1 ms of jitter into the frame generation timing[4]. When $T = 0.9$, a single SQP flow in isolation maintains low delay and utilizes 90% of the link; two or more SQP flows fully utilize the link and stabilize at a slightly higher delay (similar to SQP's behavior with inelastic cross-traffic, § 6.6.1). Reducing the target value reduces the steady state queuing delay, with the trade-off that an isolated SQP flow will have lower link utilization (Figure 6.18d) and will obtain less throughput share when competing with elastic flows (§ 6.6.1).

While SQP can be adapted to use more sophisticated mechanisms like dynamic frame timing alignment across SQP flows and dynamically lowering the target and pacing multipliers when the presence of multiple SQP flows is detected, we defer this to future work and only evaluate the base SQP algorithm with a fixed target multiplier $T = 0.9$ and a fixed pacing multiplier $m = 2$ in § 6.7.

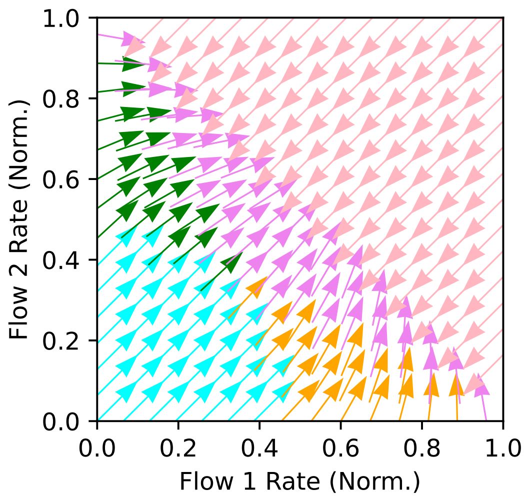
Fairness. When competing with other SQP flows, there are two key mechanisms that enable SQP to converge to fairness: SQP's pacing-based bandwidth probing (§ 6.5.2), and SQP's logarithmic utility-based bandwidth smoothing (§ 6.5.4).

Let's consider a scenario where SQP is not using bandwidth smoothing, and directly updates its bandwidth estimate according to the sample. When overutilization occurs, each flow observes a common delay signal, and hence the bandwidth is reduced by a multiplicative factor. For various values of each flow's initial rate, we compute the update step as $S \times T - B$, where S is computed using Eq. 6.7 (average case behavior with randomized frame alignment, § 6.6.1). When the link is severely underutilized by a flow (the pacing burst of SQP does not cause queuing - see § 6.6.1), the update step is $2 \times B - B = B$. These update steps are shown in Figure 6.20a as arrows, where the tail of the arrow is anchored at the initial condition, and the length of the arrow is proportional to the step size. In the cyan region, neither of the flows cause queuing due to their pacing burst, and hence, the rates undergo a multiplicative increase. In the purple region, both flows cause queuing due to their pacing burst, and the slower flow increases its rate more than the faster flow (whose increase is sublinear). The green and orange regions depict a region of transition, where only one of the flows observes pacing-induced queuing. Thus, while SQP will undergo multiplicative increase when the link is severely underutilized, as the link utilization

[4]Incorporating 1ms of sub-frame jitter into an application's frame rendering will have minimal impact on video smoothness



(a) Raw sample update



(b) Using update rule

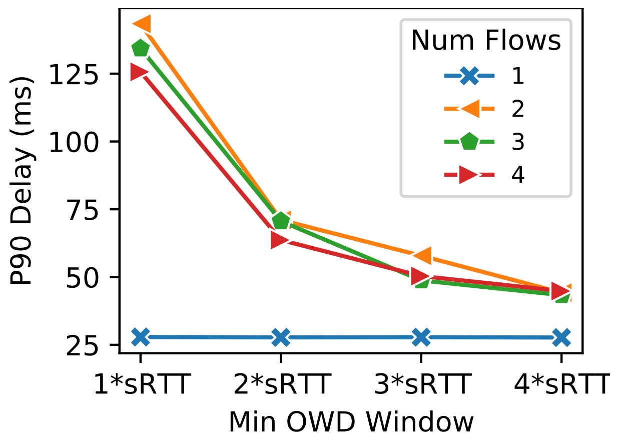
Figure 6.20: Vector field showing bandwidth update steps for different starting states for two competing flows. SQP's update rule significantly speeds up convergence to fairness.

increases, the increases become sublinear.

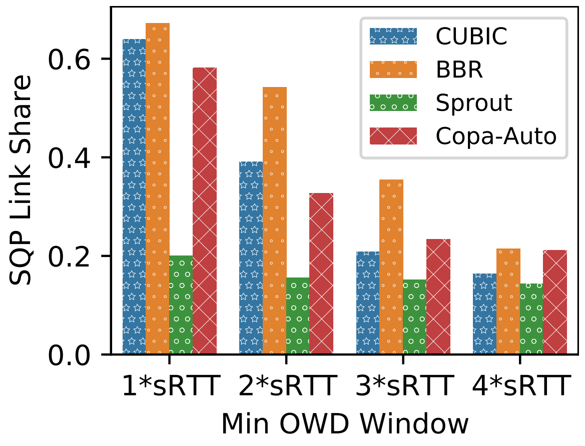
In Figure 6.20b, we compute the update steps by incorporating SQP's logarithmic utility-based bandwidth update rule. In this case, SQP undergoes sublinear increase when the link is underutilized, and linear decrease when the link is overutilized, which still converges to fairness (similar to AIMD). The linear increase speeds up convergence for multiple SQP flows from an under-utilized state, whereas the additive decrease makes SQP's throughput stable when competing with queue-building flows. SQP's bandwidth update rule also ensures that the updates are proportional to the difference relative to the current estimate, as opposed to fixed-size steps (e.g., additive increase in Cubic) or velocity-based mechanisms (e.g., Copa). We evaluate SQP's fairness in § 6.7.9.

6.6.3 Adaptive Min One-way Delay Tracking

SQP's adaptive min one-way delay window is a key mechanism that enables SQP to recover from network overutilization. Recall that SQP's window scales with the currently observed sRTT (§ 6.5.3). A larger window speeds up recovery from queuing caused by overutilization, but results in poor performance when competing with queue-building cross traffic. Different multipliers are evaluated in Figure 6.21. With $T = 0.9$ and $m = 2$, more than one SQP flows sharing a bottleneck require a larger Δ_{min} window to stabilize. A multiplier of 2 results in acceptable level of steady state queuing (nearly as low as $3\times$ and $4\times$), while achieving reasonable throughput in the presence of queue-building cross traffic like Cubic and BBR. SQP competing with Sprout is also shown as a worst case example; Sprout causes significant delay variation due to its bursty traffic pattern, causing SQP to achieve low throughput.



(a) P90 packet delay for multiple SQP flows.

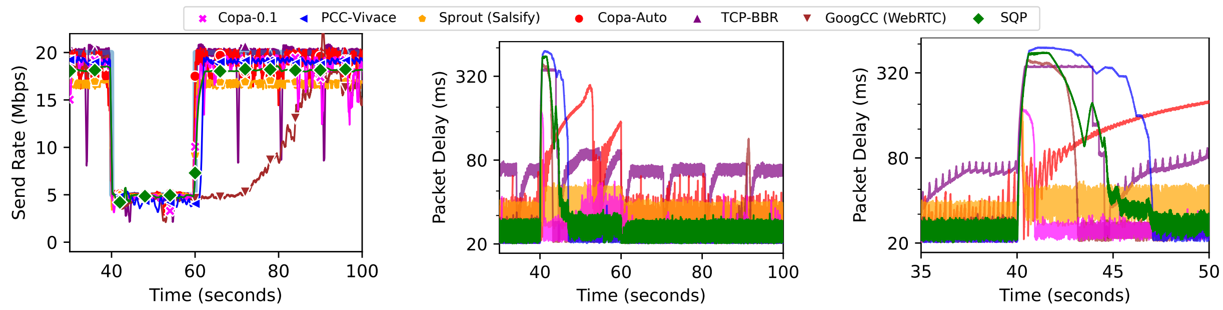


(b) Throughput of 1 SQP flow in the presence of cross traffic.

Figure 6.21: Impact of the min one-way delay multiplier on frame delay and throughput when competing with other flows. The bottleneck setup is the same as Figure 6.18, and $T = 0.9, m = 2$.

6.7 Evaluation

SQP's evaluation has three broad themes. § 6.7.4 evaluates SQP's performance on a large set of calibrated emulated links modeled after real-world network traces obtained from Google's game streaming service. §§ 6.7.5-6.7.10 evaluate SQP's throughput when competing with cross traffic, impact of shallow buffers, fairness, and bandwidth probing in application-limited scenarios. In § 6.8, we compare SQP and Copa (without mode switching) in the real world on Google's AR streaming service. In this section we compare SQP's performance to recently proposed high performance low latency algorithms like PCC [133], Copa [21] (with and without mode switching), Vivace [134] and Sprout [22], traditional queue-building algorithms like TCP-Cubic [12] and TCP-BBR [11], and WebRTC (using GoogCC as CCA), an end-to-end low-latency streaming



(a) Send rate timeseries. (b) Packet delay timeseries. (c) Packet delay timeseries (zoomed in).

Figure 6.22: Congestion control performance on a variable link (link bandwidth shown as a shaded light blue line).

solution.

6.7.1 Emulation Setup

We use the Pantheon [130] congestion control testbed, which works well for links under $100\,\mathrm{Mbps}$. For the baselines, we use the implementations available on Pantheon. These include kernel-space (Linux) implementations of Cubic and BBR-v1 [155] (iperf3 [156]), user-space implementations of PCC [157], Vivace [158], Copa [159], and Sprout, and Chromium's version of WebRTC (with GoogCC, max bitrate changed to 50 Mbps from 2 Mbps). Additionally, we evaluate the Copa algorithm with a fixed delta ($\delta = 0.1$). We implement additional functionality in Pantheon, including flow-specific RTTs, start and stop times, and testing of heterogeneous CCAs sharing a link. For experiments with fixed bandwidth links, we choose a queue size of 10 packets / Mbps ($\approx 120\mathrm{ms}$ for 20 Mbps) and the drop-tail queuing discipline. We fix $T = 0.9$ and $m = 2.0$ for SQP.

6.7.2 Metrics

While metrics like average throughput and packet delay are sufficient for evaluating a general purpose congestion control algorithm, they do not accurately reflect the impact on quality-of-experience (QoE) of a low-latency streaming application that is using a particular congestion control algorithm [160]. To evaluate how a CCA affects the QoE of low-latency streaming, we need metrics that quantify properties like video bitrate and frame delay.

After an experiment is run, Pantheon generates detailed packet traces with the timestamps of packets entering and leaving the bottleneck. We compute a windowed rate from the ingress packet traces, which serves as a baseline for the video bitrate. For a time slot t, the frame size $F(t)$ is:

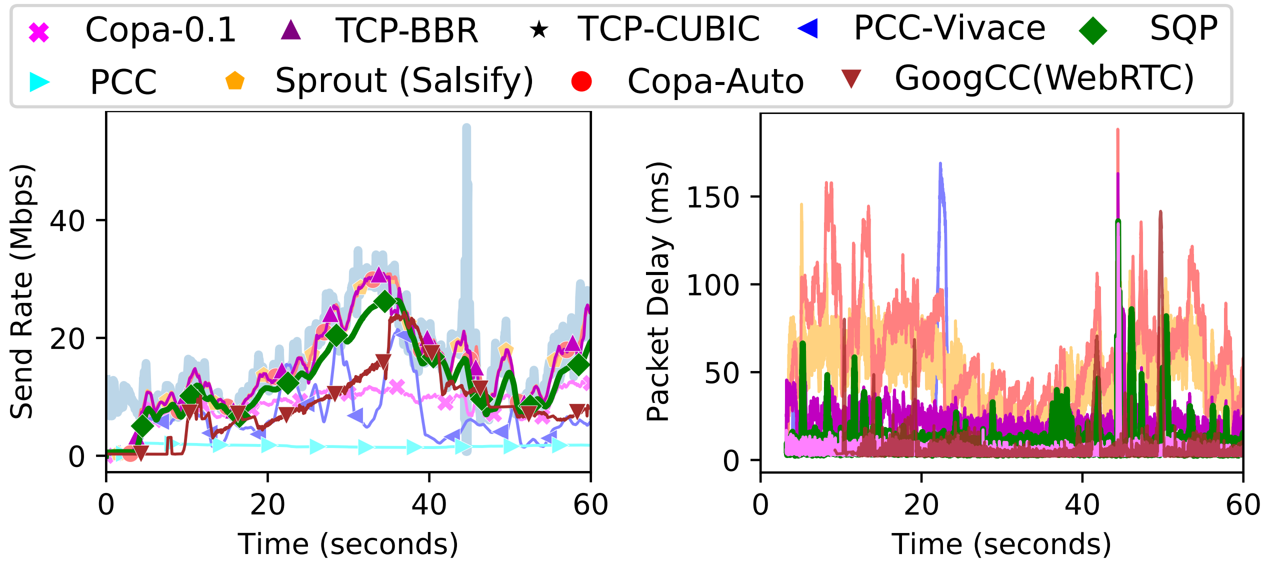
$$F(t) = max\left(\frac{S(t, t + n \cdot I) - p}{n},\ S(t, t + I) - p,\ 0\right) \tag{6.10}$$

where p denotes the pending unsent bytes from previous frames, I is the frame interval, $S(t_1, t_2)$ is the number of bytes sent by an algorithm between t_1 and t_2 and n is the window size in number of frames used for smoothing. This ensures that none of the bytes the algorithm sent in a particular interval are wasted (maximum utilization).

To quantify video frame delay, we simulate the transmission of the frames to measure the end-to-end frame delay. For zero size frames, we assume that the delay of the frame is the time until the next frame. The choice of n limits the worst case sender-side queuing delay to n frames, which can occur when an algorithm sends a burst of packets during the n^{th} frame slot after a quiescence period of $n - 1$ frames.

6.7.3 Simple Variable Bandwidth Link

We evaluated SQP on a link that runs at 20 Mbps for 40 seconds, drops to 5 Mbps for 20 seconds, and then recovers back to 20 Mbps (same as the experiment described in § 6.4.1). The throughput



(a) Throughput timeseries for a sample Wi-Fi trace.

(b) Packet delay timeseries for the trace shown in 6.23a.

Figure 6.23: Performance of various CCAs on a sample Wi-Fi network trace, with the bottleneck buffer size set to 200 packets. SQP rapidly adapts to the variations in the link bandwidth, and achieves low packet queueing delay.

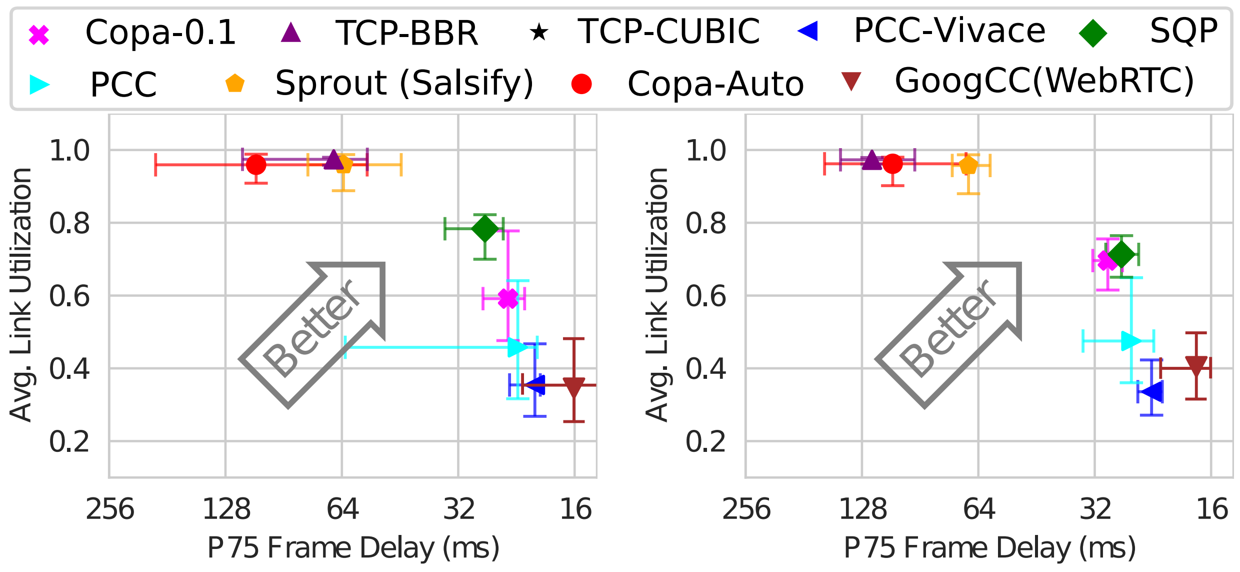
is shown in Figure 6.22a, and the delay is shown in Figure 6.22b, with a zoomed version of the delay in Figure 6.22c. SQP quickly probes for bandwidth after the link rate increases ($T = 60$), and is able to maintain consistent, low delay when the link conditions are stable. When the link rate drops, SQP's recovery is faster than PCC-Vivace, and as fast as BBR. While GoogCC's recovery is slightly faster, it takes a very long time compared to SQP in order to ramp up once the link rate increases back to 20.

6.7.4 Real-world Wireless Traces

To evaluate SQP's performance on links with variable bandwidth, delay jitter and packet aggregation, we obtained 100 LTE and 100 Wi-Fi throughput and delay traces from a cloud gaming service. Each network trace was converted to a MahiMahi trace using packet aggregation to emulate the delay variation, and the link delay was set to the minimum RTT for each trace.

Figures 6.23a and 6.23b show the throughput and delay of a single flow operating on a representative Wi-Fi trace. The thick gray line represents the link bandwidth. SQP achieves high link utilization and can effectively track the changes in the link bandwidth while maintaining low delay. While Copa-Auto, Sprout, and BBR achieve high link utilization, they incur a high delay penalty. WebRTC, PCC and Vivace are unable to adapt to rapid changes in the link bandwidth, resulting in severe link underutilization and occasional delay spikes (e.g., Vivace at T=22s).

The aggregated results for the Wi-Fi traces are shown in Figure 6.24a. Across all Wi-Fi traces, SQP achieves 78% average link utilization compared to 46%, 35% and 59% for PCC, Vivace and Copa-0.1 respectively while only incurring 4-8 ms higher delay. While Cubic, BBR,



(a) Performance across 100 real-world Wi-Fi traces.

(b) Performance across 100 real-world LTE traces.

Figure 6.24: SQP's performance over emulated real-world wireless network traces. The bottleneck buffer size was set to 200 packets. In Figures 6.24a and 6.24b, the markers depict the median across traces and the whiskers depict the 25th and 75th percentiles.

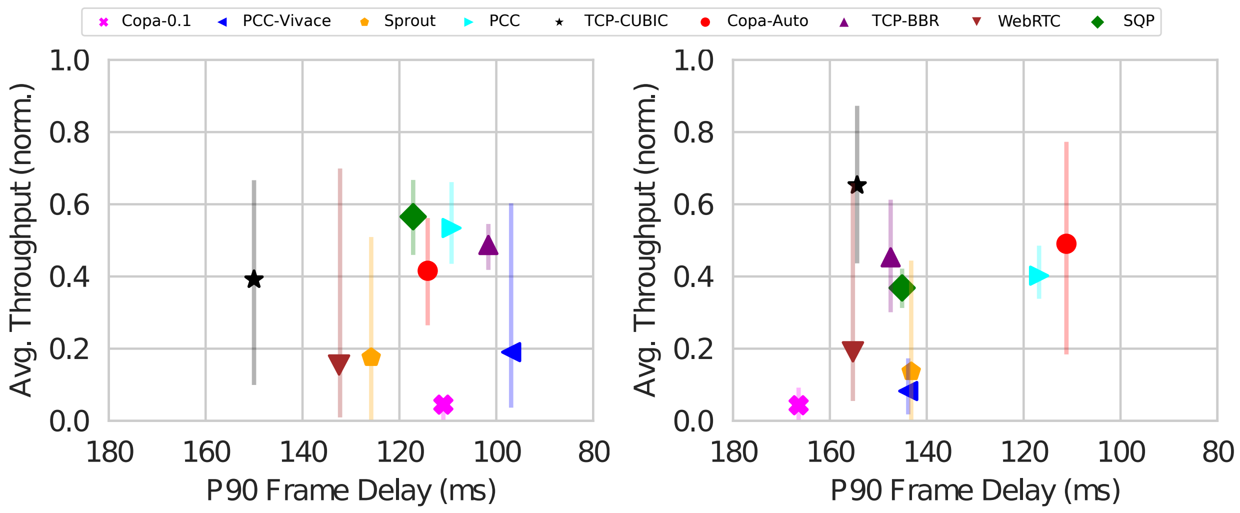
Sprout, and Copa-Auto achieve higher link utilization, this is at a cost of significantly higher delay (130-342% higher).

Figure 6.24b shows the performance various CCAs across 100 real-world LTE traces. SQP and Copa-0.1 have good throughput and delay characteristics, whereas other CCAs either have very high delays or insufficient throughput.

6.7.5 Competing with Queue-building Flows

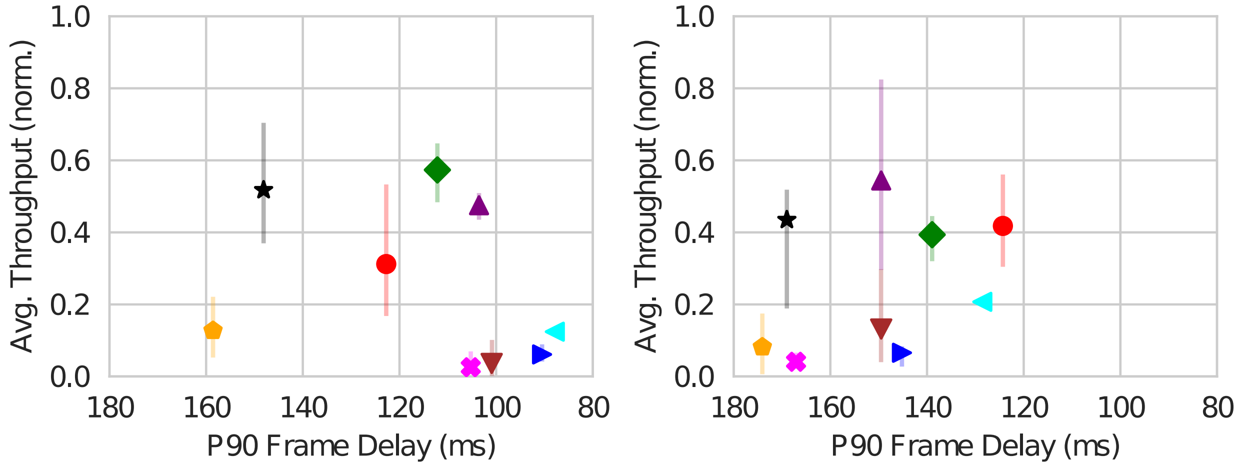
Next, we evaluate the ability of various congestion control algorithms to support stable video bitrates in the presence of queue-building cross traffic. We ran the experiment for 60 seconds on a 20 Mbps bottleneck link with 120 ms of packet buffer, and a baseline RTT of 40 ms, where each algorithm is run for 10 seconds before the cross traffic is introduced. Figure 6.25a shows the average normalized throughput and P10-P90 spread of the windowed bitrate for each algorithm versus the P90 simulated frame delay after a BBR flow is introduced. Figure 6.25c shows the average normalized throughput versus the P90 simulated frame delay when the CCA being tested starts 10 seconds after a BBR flow is already running on the link. We ran similar experiments with Cubic as the cross traffic, and the results are shown in Figures 6.25b and 6.25d.

SQP is able to achieve high and stable throughput due to SQP's bandwidth sampling mechanism (§ 6.5.2) and the use of a dynamic min-oneway delay window size (§ 6.5.3). While PCC performs well when it starts before the competing traffic is introduced on the link, PCC's normalized throughput is less than 0.2 when it starts on a link that already has a BBR or Cubic flow running on it. GoogCC's slower start affects its throughput, with things improving slightly if



(a) Streaming performance when a BBR flow starts after the primary flow has reached steady state.

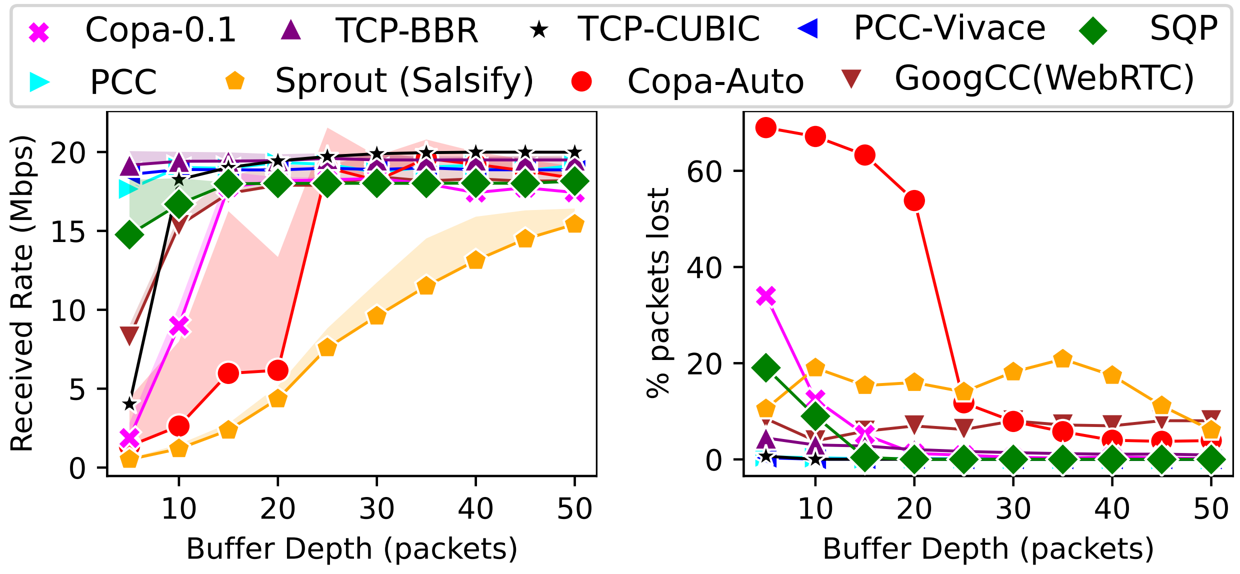
(b) Streaming performance when a Cubic flow starts after the primary flow has reached steady state.



(c) Streaming performance when a CCA starts after a BBR flow has reached steady state.

(d) Streaming performance when a CCA starts after a Cubic flow has reached steady state.

Figure 6.25: CCA performance when competing with queue-building cross traffic. The error bars mark the P10 and P90 simulated frame bitrates (§ 6.7.2).



(a) Ingress and Egress rate vs. Buffer Depth. (b) Packet Loss rate vs. Buffer Depth.

Figure 6.26: Performance impact of shallow buffers on a 20 Mbps, 40ms RTT link.

the Cubic flow starts after 20s, and it is also suffers from the latecomer effect. Vivace, Copa-0.1 and Sprout are unable to maintain high throughput in all the cases. While Copa-Auto has good average throughput, its performance is unstable at the frame timescale, which is evident by the spread between the P10 and P90 bitrate.

6.7.6 Shallow Buffers

For the target workload of interactive video with $I = 16.66$ ms (60FPS) and a pacing multiplier $m = 2$, SQP's pacing-based bandwidth probing only requires approximately 8 ms of packet buffer at the bottleneck link to be able to handle the burst for each frame. The lines in Figure 6.26a show the link egress rate for various CCAs for different buffer sizes, and the shaded regions denote the rate of loss (i.e., the top of the shaded region is the link ingress rate). The loss rate is also shown in Figure 6.26b. SQP achieves its maximum throughput with a buffer of 15 packets or more, which corresponds to 8 ms of queuing on a 20 Mbps link. If the buffer size is smaller than 15 packets, SQP transmits at 18 Mbps ($T = 0.9$ fraction of the link capacity), but the packets that correspond to the tail end of each frame are lost. Copa-Auto, Sprout and GoogCC require larger buffers, whereas BBR (˜4% loss with a 5 packet buffer), and both PCC versions (¡1% loss with a 5 packet buffer) excel at handling shallow buffers. Typical last-mile network links like DOCSIS, cellular, and Wi-Fi links have much larger packet buffers [161]. When SQP competes with other flows (vs. SQP, inelastic flows), SQP may require a higher level of queuing to stabilize. Dynamic pacing and target mechanisms are required to handle such scenarios, and we leave that for future work.

Discussion: While SQP causes sub-frame queuing since it paces each frame at 2X of the bandwidth estimate, this queuing is limited to a maximum of 8 ms (14 packets for 20 Mbps). Hence, for buffer sizes of 15 packets and above, SQP has exactly zero loss. Sprout on the other hand has 10-20 % loss for buffer sizes all the way up to 50 packets, and more than 60% of packets sent by Copa-Auto are lost for buffer sizes lower than 20 packets. GoogCC (WebRTC) has around 10% packet loss for the entire range of buffer sizes evaluated here, which may be due to WebRTC sending FEC packets in response to the loss observed.

6.7.7 Short Timescale Variations

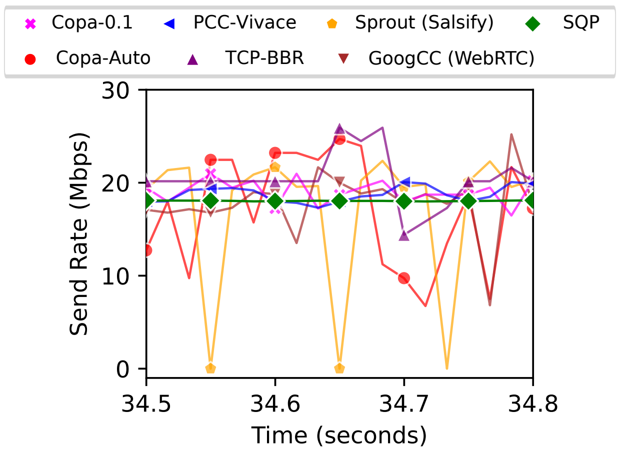
In Figure 6.27, we show the short-term throughput and delay behavior of SQP and other various CCAs, over a period of 0.5 seconds (see § 6.4.2). Figure 6.27a shows the transmission rate for each frame period (16.66 ms at 60 FPS). SQP's transmission rate is very stable, and does not vary at all across multiple frames. Figure 6.27b shows the packet delay for various CCAs over 0.5 seconds. Since SQP transmits each packet as a short (paced) burst, it causes queuing at sub-frame timescales, but since SQP does not use more than 90% of the link (due to $T = 0.9$, § 6.5.5), there is no queue buildup that occurs across frames. While sprout's probing looks similar, the queuing caused by Sprout is much higher. We note that Sprout's dips in throughput may be caused by the fact that the burst frequency of the Sprout sender used in our test is 50 cycles per second. This may not be a factor for video streaming if the burst frequency matches the video frame rate. Sprout's inadequacy for low-latency interactive streaming applications primarily stems from its inability to achieve sufficient bandwidth when competing with other queue-building flows.

6.7.8 Impact of Feedback Delay

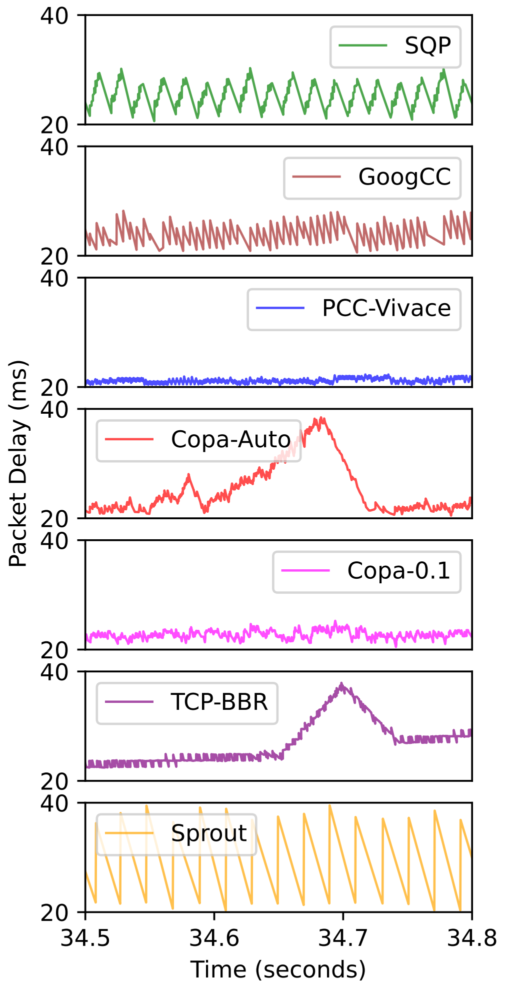
Since SQP receives the frame delivery statistics at the sender after 1-RTT, it is important to evaluate the impact of delayed feedback on SQP's dynamics. Figure 6.28a shows the average delay, and Figure 6.28b shows the average throughput after a 20 Mbps link steps down to 5 Mbps, for various baseline network RTT values. The link is run at 20 Mbps for 40 seconds, following which the link is run at 5 Mbps for an additional 20 seconds. Figure 6.28a shows the average delay for the last 20 seconds, when the link is operating at 5 Mbps. SQP's performance is consistent across the entire range, even though the feedback is delayed, and can be attributed to the fact that SQP uses a larger window for Δ_{min} on higher RTT links. Sprout and Copa-Auto have lower delay on higher RTT links, but for different reasons: Sprout's link utilization drops sharply as the link RTT increases from 40 to 80 ms (Figure 6.28b), and hence, it's delay is lower, whereas Copa-Auto incorrectly switches to competitive mode on low RTT links, causing very high delays (Figure 6.28c shows the delay timeseries for a 10ms RTT link). PCC-Vivace can only maintain low delay across a 10ms RTT link, and PCC is unable to drain the queuing caused after the link rate drops in all cases.

6.7.9 Fairness

The first experiment evaluates the performance of 10 homogeneous flows sharing a $60\,\mathrm{Mbps}$ bottleneck link, with a link RTT of $40\,\mathrm{ms}$. Figure 6.29a compares the average throughput and the P90 one-way packet delay for each flow. The ideal behavior is that each flow achieves exactly 6 Mbps and low delay, ie. the points should be clustered at the 6 Mbps line and be towards the right

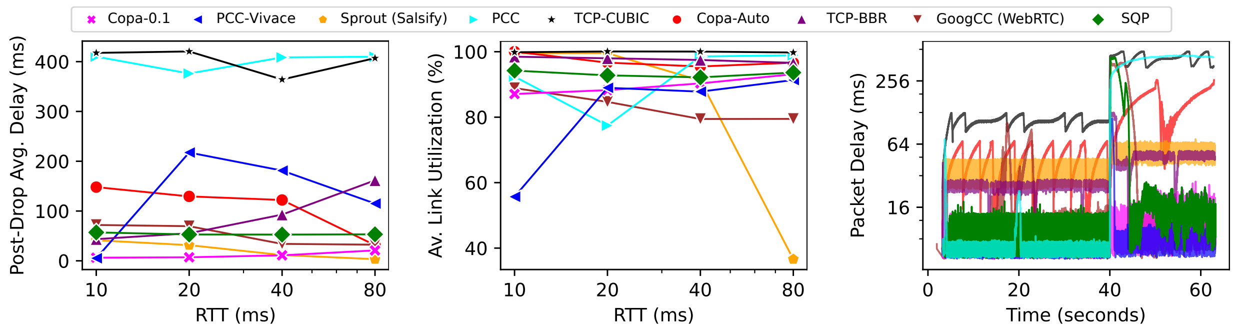


(a) Send rate timeseries.



(b) Packet delay timeseries.

Figure 6.27: Short-timescale throughput and delay behavior on a 20 Mbps link (link bandwidth shown as a shaded light blue line).

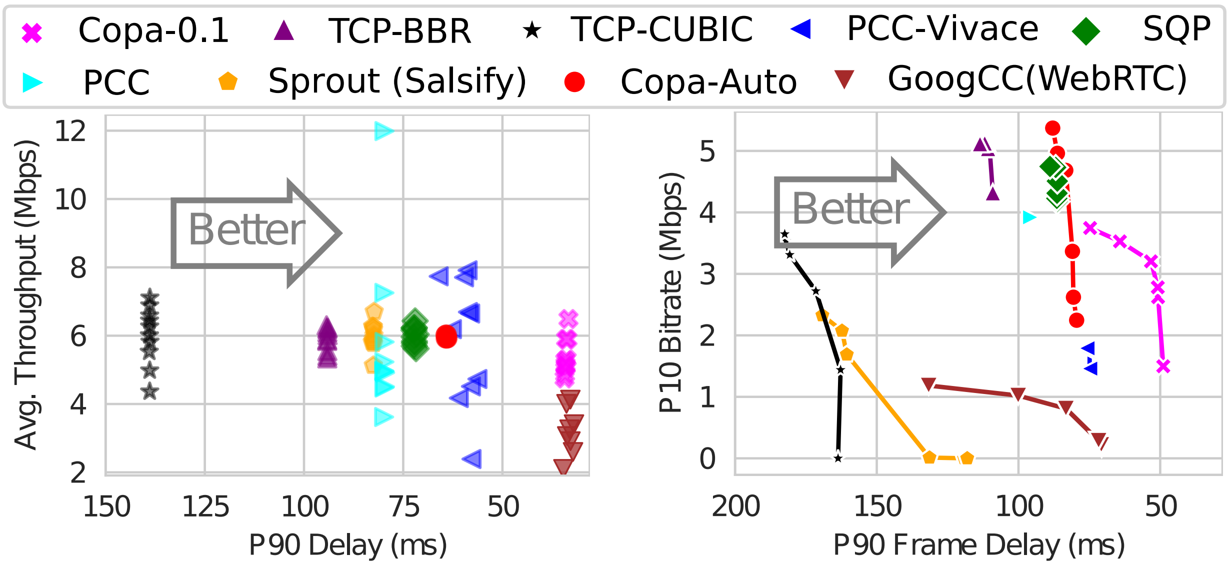


(a) Average delay after the link rate drops for various RTTs.

(b) Average link utilization for different RTTs. Sprout has low utilization on higher RTT links, and thus, lower delay.

(c) Packet delay timeseries for 10ms RTT link. Copa-Auto runs in competitive mode on low RTT links (10ms).

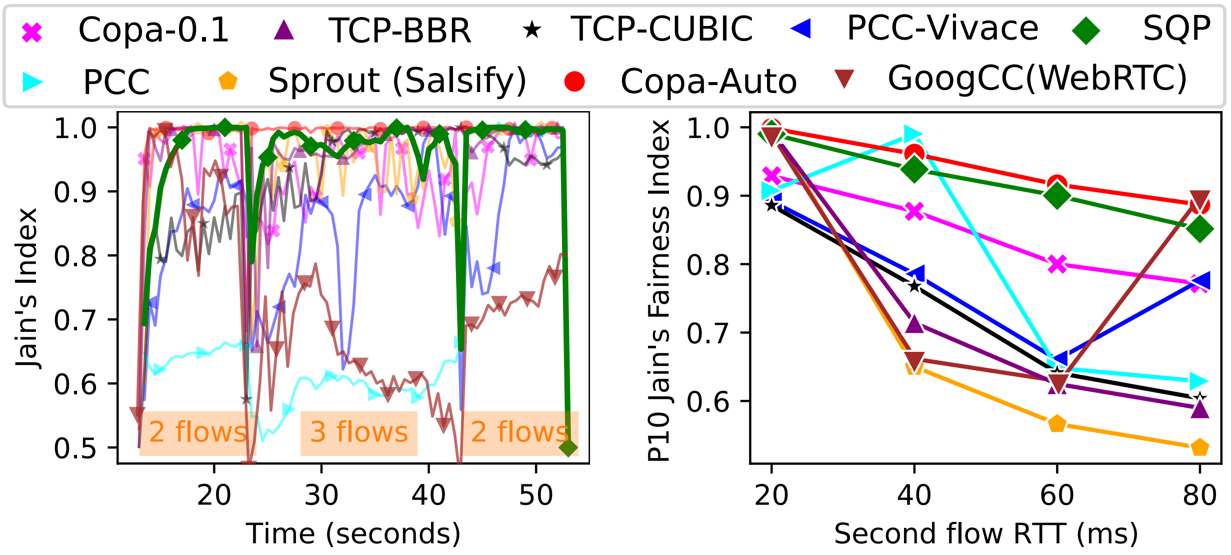
Figure 6.28: Impact of link RTT on throughput and delay, where link changes from 20 Mbps to 5 Mbps at T=40s.



(a) Throughput-delay performance of 10 flows sharing a 60 Mbps bottleneck link.

(b) Bitrate-frame delay performance of 10 flows sharing a 60 Mbps bottleneck link.

Figure 6.29: Fairness results with 10 flows sharing a bottleneck. SQP achieves a fair share of throughput on average, and at smaller time-scales. CCAs like Sprout, WebRTC, and Copa become excessively bursty at smaller time-scales.



(a) Jain's fairness index over time for homogeneous flows entering and exiting the bottleneck link.

(b) P10 fairness (500 ms windows) for flows with different RTTs. First flow RTT = 20 ms.

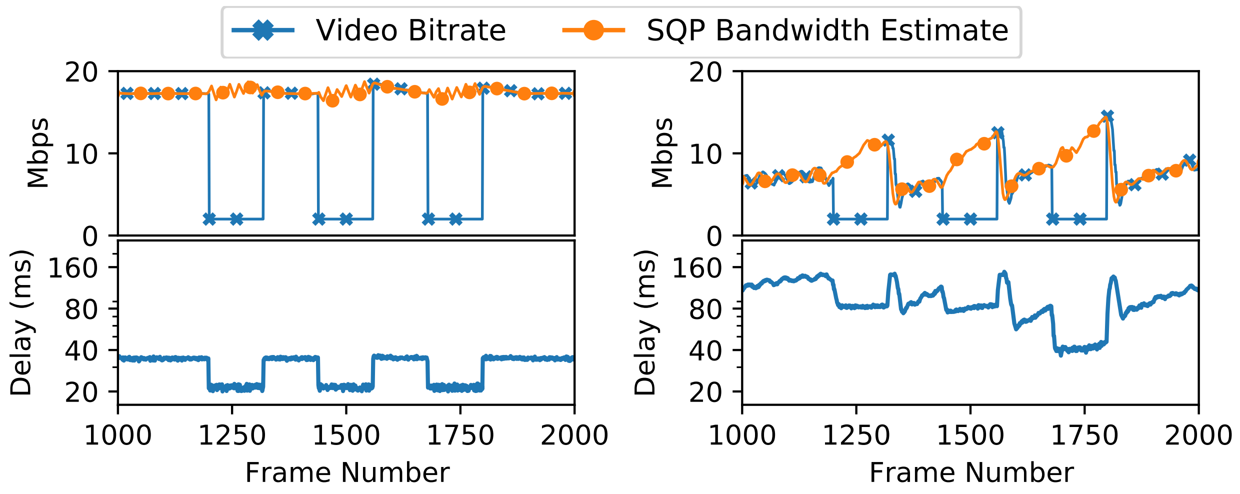
Figure 6.30: Dynamic fairness and RTT fairness comparison. SQP quickly converges to fairness, and has good RTT fairness.

in the plot. SQP flows[5] achieve equal share of the link, with lower P90 one-way packet delay compared to Sprout, PCC, BBR and Cubic (75 ms). While BBR, and both versions of Copa achieve good fairness with respect to the average throughput for the full experiment duration, neither version of PCC is able to do so. While WebRTC has very low P90 packet delay, the flows cumulatively underutilize the link and do not achieve fairness. Figure 6.29b compares the streaming performance of the algorithms by plotting the P10 bitrate and the P90 frame delay for different bitrate estimation windows ranging from 1 frame to 32 frames in multiplicative steps of 2 (§ 6.7.2). The streaming performance of Copa-0.1, Sprout, Cubic and WebRTC are significantly worse than their average throughput and packet delay due to bursty transmissions when multiple flows share the bottleneck link.

In the second experiment, we evaluate dynamic fairness as flows join and leave the network. Flows 2 and 3 start 10 s and 20 s after the first flow respectively, and stop at 40 s and 50 s respectively. Figure 6.30a plots the Jain fairness index [162] computed over 500 ms windows versus time. SQP converges rapidly to the fair share, whereas both versions of PCC, Copa-0.1 and WebRTC cannot reliably achieve fairness at these time scales.

SQP also demonstrates good fairness across flows with different RTTs. We evaluated steady-state fairness among flows that share the same bottleneck, but have different network RTTs. In Figure 6.30b, we plot the P10 fairness (using Jain's fairness index) across windowed 500 ms intervals. When two flows have the same RTT, SQP, Copa-Auto, TCP-BBR and Sprout achieve perfect fairness. As the RTT of the second flow increases, SQP and Copa-Auto are able to maintain reasonable throughput fairness but the fairness degrades rapidly in the case of TCP-BBR, Cubic and Sprout as the RTT of one flow increases. The slight drop in fairness in the case

[5]Inter-frame timing jitter enabled (§ 6.6.1)



(a) SQP in isolation. (b) Competing with Cubic.

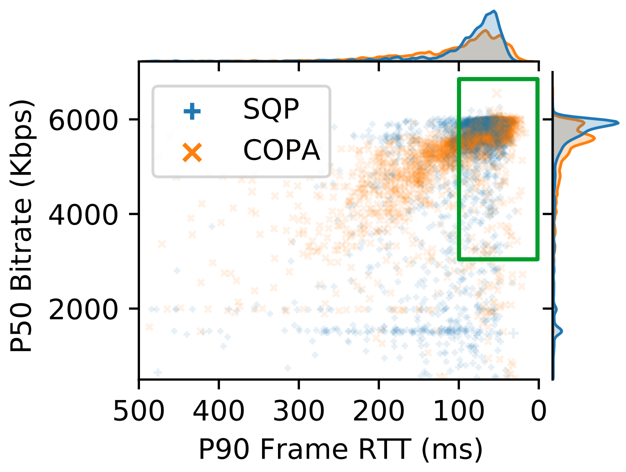
Figure 6.31: SQP's performance when application-limited.

of SQP is because the flow with the higher RTT achieves lower throughput since its minimum one way delay window size is larger. PCC, Vivace, and WebRTC also achieve low fairness for flows with different RTTs and do not demonstrate any particular pattern as the RTT difference between the flows increases. In order to achieve additional fairness when competing with other loss-based CCAs, SQP could incorporate loss signals into it's bandwidth estimate in a manner similar to TCP-friendly rate control (TFRC [149]).

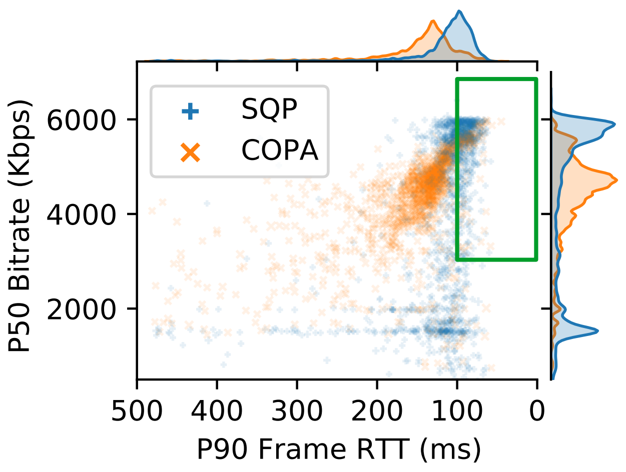
6.7.10 SQP Video Codec Integration

We evaluated SQP's bandwidth estimation in a scenario where the video bitrate is significantly lower than the bandwidth estimate. We tested SQP by artificially limiting the video bitrate on a 20 Mbps, 40 ms RTT link with 120 ms of bottleneck buffer. The encoder bitrate is artifically capped to 2 Mbps for three 2-second intervals. In Figure 6.31a, SQP maintains a high bandwidth estimate, which is appropriate since SQP is the only flow on the link. SQP also obtains a reasonable estimate of the link bandwidth under application-limited scenarios when competing with other flows. Figure 6.31b shows SQP's bandwidth estimate when the video bitrate is lower than the target bitrate and SQP is competing with a Cubic flow. When the video bitrate is lower than the target, SQP is able to maintain a high bandwidth estimate, which demonstrates that SQP is able to maintain a high bandwidth estimate without requiring additional padding data. This allows SQP to quickly start utilizing its share when the video bitrate is no longer limited (matches the target bitrate), instead of acquiring its throughput share from scratch, which would take much longer. These experiments demonstrate that padding bits are not necessary for SQP to achieve good link utilization.

The generated video bitrate can also overshoot the requested target bitrate. In such scenarios, it is typically the encoder's responsibility to make sure that the average video bitrate matches the requested target bitrate, although SQP can handle and recover from occasional frames size overshoots since they would cause subsequent bandwidth samples to be lower. Persistent overshoot



(a) Wi-Fi performance.



(b) LTE performance.

Figure 6.32: Real world A/B testing of SQP and Copa-0.1.

can occur in very complex scenes when the target bitrate is low. In such cases, the application must take corrective actions that include reducing the frame rate or changing the video resolution. Salsify [14] proposes encoding each frame at two distinct bitrates, choosing the most appropriate size just before transmission. In Chapter 4, we show that standard modern hardware encoders like NvENC have very accurate rate control mechanisms, and thus, there is no need to rely on highly custom encoders like the one used in Salsify. Note that SQP can serve as a viable replacement for Sprout in Salsify.

6.8 Real-World Performance

To evaluate SQP's performance in the real world, we deployed SQP in Google's AR streaming platform. We also deployed Copa-0.1 (without mode switching) on the same platform by adapting the MVFST implementation of Copa [139] and performed A/B testing, comparing the performance of the two algorithms. We chose Copa-0.1 since it consistently maintained low delay compared to other CCAs (e.g. Sprout (Salsify) has very high delays) on emulated tests, and has been demonstrated to work well for low-latency live video in a production environment [148] [6]. For Copa, we use $\frac{\text{CWND}}{\text{sRTT}}$ to set the encoder bitrate, and reduce the bitrate by $\frac{Q_{sender}}{D}$ when sender-side queuing occurs (Q_{sender} = pending bytes from previous frames, $D = 200\text{ms}$ is a smoothing factor), gradually reducing the sender-side queue over a period of 200 ms. We ran the experiment for 2 weeks and obtained data for approximately 2400 Wi-Fi sessions and 1600 LTE sessions for each algorithm. Figure 6.32 shows the scatter plots of the median bitrate and the P90 frame RTT (fRTT; send start to notification of delivery) in addition to the separate distributions for each metric. 64 SQP and 105 Copa sessions over LTE, and 36 SQP and 52 Copa sessions over Wi-Fi had a P90 fRTT higher than 500 ms, and these are not shown in the figure.

[6]In addition, Salsify's custom software encoder cannot sustain the frame rates required for low-latency interactive streaming applications

71% of SQP sessions over Wi-Fi had good performance (bitrate $>$ 3 Mbps, fRTT $<$ 100 ms), compared to 56% of Copa-0.1 sessions. On LTE links, 36% of SQP sessions had good performance, compared to 9% of Copa sessions. Across all the sessions, fRTT was less than 100 ms for 64% of SQP sessions and only 39% of Copa sessions. These regions are highlighted with green boxes in Figure 6.32.

SQP achieves lower frame delay compared to Copa across both Wi-Fi and LTE. SQP on Wi-Fi also achieves higher bitrate compared to Copa. On LTE connections, SQP demonstrates a bi-modal distribution of the bitrate, with a significant number of sessions being stuck at a low bitrate despite having a low RTT. We believe SQP gets stuck at a low bandwidth estimate due to a combination of noisy links, a low bandwidth estimate and encoder undershoot, although this needs to be investigated further (Eq. 6.2 was not used). On the other hand, the bitrates for Copa sessions over LTE are more evenly distributed, but also incur higher delays compared to SQP.

Our results from emulation and real-world experiments demonstrate that SQP can efficiently utilize wireless links with time-varying bandwidth and simultaneously maintain low end-to-end frame delay, making it suitable for wireless AR streaming and cloud gaming applications.

6.9 Conclusion

In this chapter, we have presented the design, evaluation, and results from real-world deployment of SQP, a congestion control algorithm designed for low-latency interactive streaming applications. SQP is designed specifically for low-latency interactive video streaming, and makes key application-specific trade-offs in order to achieve its performance goals. SQP's novel approach for congestion control enables it to maintain low queuing delay and high utilization on dynamic links, and also achieve high throughput in the presence of queue-building cross traffic like Cubic and BBR, without the caveats of explicit mode-switching techniques. SQP's video-aware design is an important design aspect that enables SQP to achieve low end-to-end frame delay, as opposed to video-agnostic congestion control algorithms.

The key takeaways from this chapter are:

1. Emerging video streaming applications like cloud gaming, and remote-rendered AR/VR have demanding QoE requirements, which include high video quality and extremely low end-to-end video frame delay. Existing congestion control algorithms are not adequate.
2. Adverse interactions between the video frame traffic pattern and a video-agnostic congestion control algorithm's transmission pattern can lead to sub-optimal QoE in terms of video frame delay and video quality.
3. A congestion control algorithm that makes application-specific trade-offs and takes an integrated approach that cuts across the video and network layer can significantly improve the QoE of emerging video streaming applications.

Chapter 7

CC-Fuzz: Genetic algorithm-based fuzzing for stress testing congestion control algorithms.

Traditional congestion control algorithms (CCAs) were designed with the core goals of high throughput and fairness, while preventing congestion collapse. Unfortunately, traditional loss-based CCAs fall short in terms of meeting the performance requirements for many emerging applications and network environments. Recent research has shown a significant interest in designing new congestion control algorithms for achieving specific performance goals, or improving general CCA performance. For example, SCReAM [163], GoogCC [138], and Sprout [22] are designed for low latency video streaming, with the key goal of achieving low end-to-end delay. Some CCAs are designed to extract maximum performance from special networking infrastructure avaiable in data center settings, such as programmable NICs (Swift, TIMELY), switches supporting AQM (DCTCP), and hybrid optical-packet networks [164]. Other CCAs like Copa [21], Nimbus [135], and TCP-BBR [96] use complex network modeling techniques in order to achieve dual goals of (1) low delay, and (2) high throughput when competing with other flows.

A significant fraction of new CCAs are developed by the academic community, where the opportunities for large scale testing in the real-world are limited. In order to deploy a new CCA on the Internet or in large-scale data centers, where the packets traverse across a variety of network conditions and encounter diverse cross traffic patterns, it is important to evaluate the robustness of a CCA and it's implementation across a wide range of scenarios. This is a challenging task in an academic setting - many newly proposed CCAs are evaluated using a combination of small scale deployment [130], and local scenario-based emulation and simulation [165]. These new CCAs are much more complex compared to traditional loss-based CCAs like TCP-CUBIC and TCP-Reno, and the tests typically performed at an academic scale can easily miss situations where the algorithm fails to achieve it's goals (like high utilization, fairness, or low delay [135, 137]), or corner cases where bugs in the CCA implementation are triggered.

In this chapter, we present CC-Fuzz, an automated congestion control testing framework that uses a genetic search algorithm in order to stress test congestion control algorithms by generating adversarial network traces and traffic patterns. Initial results using this approach are promising

- CC-Fuzz automatically found a bug in BBR that causes it to stall permanently, and is able to automatically discover the well-known low-rate TCP attack, among other things.

7.1 Introduction

In this paper, we describe the design of our testing framework called "CC-Fuzz[1]", and demonstrate the value of using genetic search algorithms for exploring the search space of link behavior and cross traffic patterns in order to identify issues with CCAs and their implementations, or inspire confidence in a CCA before it is deployed in the real-world. A genetic algorithm is a search heuristic that is inspired from the Darwinian theory of biological evaluation - the algorithm maintains a pool of traces and on every iteration, each entity in the population is assigned a fitness score. The fitness scores are used to generate the next population generation in a manner that is similar to natural selection. In our case, the population entities are network traces, the fitness scores are based on the performance of a CCA for each trace, and evolution involves modifying the traces in the population in a manner such that eventually we find traces that trigger poor behavior in the CCA being tested (convergence).

CC-Fuzz has two modes - (1) Link mode aims to identify bottleneck packet transmission patterns, and (2) Traffic mode aims to identify cross traffic patterns, that that result in poor performance for the particular CCA being tested. We believe these two approaches can trigger different behaviors, since variations in the link rate model arbitrary delay jitter, whereas the delay is bounded when injecting cross traffic. In order to generate realistic link behavior and cross traffic, CC-Fuzz uses (1) heuristics during trace generation, and (2) leverages the generality of genetic algorithms by using carefully designed fitness scores for imposing implicit constraints that are hard to model using heuristics. In Section 7.5, we propose an alternate way to impose realism on network traces as part of future work.

We present a version of CC-Fuzz that uses NS3-based [167] simulation in order to evaluate CCAs and assign fitness scores for the traces, and we evaluate the pre-defined CCAs in NS3. We discuss the challenges of using emulation-based testing in Section 7.3.6. Our findings (§ 7.4) include:

1. **BBR**: CC-Fuzz is able to generate network traces that cause BBR to permanently stall due to the way ACKs and spurious retransmissions interact with each other during a retransmission timeout. CC-Fuzz is also able to generate traffic patterns that trigger BBR to cause high queuing delays.
2. **CUBIC**: CC-Fuzz is able to generate traffic patterns that trigger a NS3-specific implementation bug in CUBIC regarding CWND updates.
3. **Reno**: CC-Fuzz is able to generate traffic patterns that are similar to the TCP low-rate attack [168].

In the remainder of the paper, we discuss the design of CC-Fuzz (§ 7.3) and present directions for future work (§ 7.5).

[1]CC-Fuzz is a pun on Sisyphus. Wikipedia notes, "tasks that are both laborious and futile are therefore described as Sisyphean" [166]

7.2 Motivation and Related Work

Newly proposed CCAs are often evaluated using metrics such as throughput, delay and fairness across a range of simple scenarios like testing a CCAs ability to track available bandwidth at macroscopic time-scales, and coexist with other flows (e.g. by introducing competing flows using various CCAs).

Past work has shown that commonly evaluated scenarios often fail to catch surprising failure modes - In [137], the authors use mathematical modeling to show that multiple BBR flows are unfair towards loss-based CCAs. In CCAC [169], the authors argue that basic evaluation techniques are not sufficient for capturing every scenario that causes undesirable behavior, and propose a formal verification technique that can generate network behavior that satisfy queries about CCA performance. Formal approaches are limited since they analyze "theoretical models" of CCAs. This step can hide key bugs and issues present in real implementations. In addition, formal techniques become intractable when the fidelity of the model is increased or when verifying properties over long time periods (e.g. BBR's minRTT probing behavior).

Fuzzing [170] is a widely used technique for discovering vulnerabilities in code. TCPwn [171] uses model-based fuzzing in order to identify manipulation attacks (e.g. dup ACK injection, ACK storm, sequence desynchronization) on CCAs. TCP-Fuzz [172] tests TCP stack implementations for bugs, and ACT [173] uses state space exploration to generate particular numerical values of the state variables in the CCA implementation that triggers buggy behavior.

Packetdrill [174] uses scripted tests to detect bugs in the networking stack and for regression testing of CCAs. Packetdrill has proven successful in catching many issues over time, but it requires clearly laid out networking scenarios that must be developed by hand - this can miss many situations.

The goal of CC-Fuzz is to find realistic situations where CCA performance suffers due to packet delivery timing and losses *automatically*. We do not aim to find protocol-level bugs that are triggered by injecting spoofed packets - the tools mentioned above can be used for low-level bug finding. We are not aware of any existing system that automatically generates network environments for stress-testing CCAs for high level throughput and other performance objectives.

7.3 Design

CC-Fuzz explores the search space of network behavior and cross traffic by using a genetic algorithm for generating network traces in order to identifying network behaviors that cause the CCA to perform poorly. CC-Fuzz's high level loop is described in Figure 7.1. CC-Fuzz's core components include the following:

1. **Trace Generator**: Generates initial traces, and defines functions for performing cross-overs between pairs of traces, and mutating individual traces.
2. **Scoring Function**: Simulates or emulates the CCA's performance for a link or traffic trace, and assigns a score based on the property being evaluated (e.g. throughput, delay, loss, or a combination).
3. **Selection Algorithm**: Selects trace pairs for generating cross-over traces for the next generation, and selects traces that will be mutated before being added to the next generation

Algorithm Genetic Algorithm Loop

```
procedure CC-FUZZ
    TRACES ← Initial pool of traces
    kElite ← Number of traces that live on unmodified.
    kCrossover ← Count of new traces generated by
                 combining traces with high scores.
    repeat
        for trace ∈ TRACES do
            SCORE($i$) ← Score when CCA run with
                         TRACES($i$)
        ELITE ← Top kElite traces
        CROSSOVER ← kCrossover traces that are
                    generated by combining traces
        MUTATED ← len(TRACES) - kElite - kCrossover
                  traces generated by modifiying traces
        TRACES ← ELITE + CROSSOVER + MUTATED
    until convergence
```

Figure 7.1

trace pool.

These components are discussed in further detail below.

7.3.1 Network Model

CC-Fuzz uses a simple network topology with two sources (one source uses the CCA being tested, and the other source generates cross traffic) that are connected to a gateway with high speed links. The gateway is connected to a sink via a bottleneck link with a fixed propagation delay. The gateway consists of a fixed-size drop-tail FIFO queue.

CC-Fuzz has two distinct modes -

1. **Link Fuzzing:** We generate bottleneck service curves that define the rate at which packets are drained from the bottleneck queue and transmitted over the link.
2. **Traffic Fuzzing:** We generate traffic traces that determine the injection of cross traffic into the bottleneck queue, and the bottleneck transmission rate is fixed.

We explore two different modes due to the following reasons:

1. A fixed rate link with variable cross traffic cannot model unbounded delays that can be caused by variable links with a fixed bottleneck buffer size.
2. Variable links with a fixed bottleneck buffer size cannot model loss with bounded delay, which can be caused by queue-building cross traffic.
3. A realistic link may exhibit aggregation and delay jitter, and some degree of long-term temporal rate variation. On the other hand, realistic cross-traffic can be highly adversarial.

Algorithm Packet Distribution Algorithm

```
procedure DistPackets(num, start, end)
    if $num == 0$ then return [ ]
    if $num == 1$ then return $[\frac{start+end}{2}]$
    $rate \leftarrow \frac{num}{end - start}$
    loop
        $tsplit \leftarrow \mathbf{U}(start, end)$
        $numleft \leftarrow \mathbf{U}(0, num)$
                                                    ▷ U is uniform random sampling.
        if $end - start < kAgg$ then break
        $lrate \leftarrow \frac{numleft}{tsplit - start}$, $rrate \leftarrow \frac{num - numleft}{end - tsplit}$
        if $lrate > 2 \times rate$ or $rrate > 2 \times rate$ then
            continue
        if $lrate < 0.5 \times rate$ or $rrate < 0.5 \times rate$ then
            continue
    return DistPackets(numleft, start, tsplit)
            + DistPackets(num - numleft, tsplit, end)
```

Figure 7.2

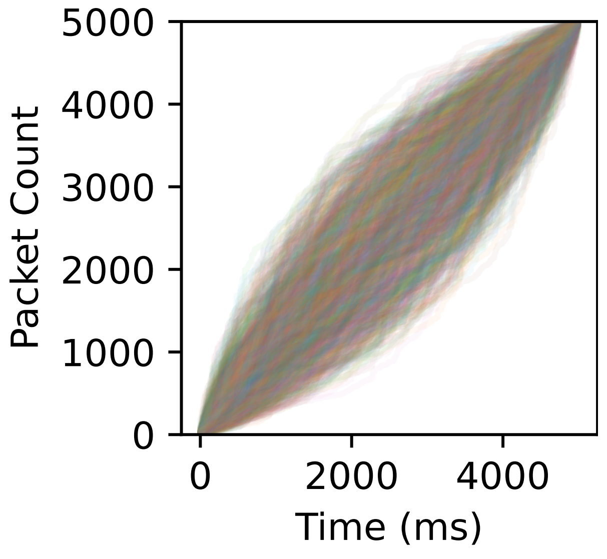
Separating out link and traffic fuzzing enables us to model such constraints and makes the results easier to understand.

These two modes are quite general, but they do not capture certain behaviors like random packet losses. This and other approaches like combining link and traffic fuzzing can be potential directions for future work (§ 7.5).

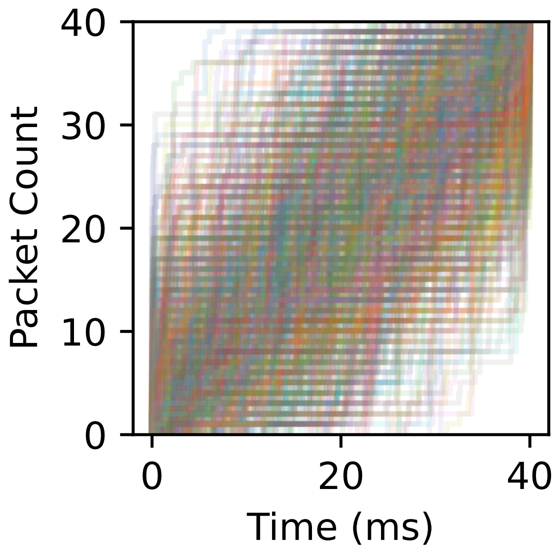
7.3.2 Link Fuzzing

Link fuzzing aims to generate bottleneck service curves that trigger poor performance in the CCA being tested. We represent the service curve as a sequence of packet transmissions (similar to the model used in MahiMahi [165]). This representation lends itself well for modeling jitter, and imposing high-level constraints on the service curve. For link fuzzing, we fix the total number of packets that can be serviced by the link during a run (and thus, the average bandwidth).

Initial Trace Generation. In order to generate realistic traces, but still cover a large portion of the search space, CC-Fuzz distributes the packet transmissions over time using the DistPackets algorithm shown in Figure 7.2. The key idea is to recursively divide packets by splitting the time length and number of packets into two in each step, and ensuring that in each step, the average rate for each division lies within a multiplicative range of the average rate. This mechanism is a heuristic that bounds the long term variation in the bandwidth. Deeper in the recursion, when the time length drops below a threshold (*kAgg*), the bound checks are relaxed in order to allow arbitrary short-term rate variations to model packet aggregation and jitter. In Figure 7.3, we show



(a) 5 second interval.



(b) 50 millisecond interval.

Figure 7.3: Service curves generated using DISTPACKETS, with an average rate of 12 Mbps and $kAgg = 50\,\mathrm{ms}$.

the distribution of service curves generated by this algorithm.

Evolution Mechanism Typical genetic algorithms have two evolution mechanisms: mutations, and crossovers. Mutation mechanisms choose entities that have the desired properties, and modify them in order to generate new entities for populating the next generation. Crossover mechanisms pick two or more traces that have the desired properties, and combine them in some way for generating new entities.

When creating a new generation from a pool of link traces from a previous generation, CC-Fuzz must ensure that the same properties as that of the initial generation hold - otherwise, the constraints we want to impose on the traces can be violated to arbitrary extents after only a few generations. For mutating a link trace, CC-Fuzz selects a random split point in the trace, and redistributes packets (DISTPACKETS) on either the left or the right side of the split point (chosen using a coin toss). This preserves the properties of the trace from the initial generation step. CC-Fuzz does not use crossovers for link fuzzing, since there is no obvious way of combining two independent service curves while maintaining the core properties of the two subtraces(e.g. total packets transmitted, rate variation heuristics). In order to generate easier to understand link traces, CC-Fuzz optionally supports trace annealing. After evaluating a trace and before performing mutations, CC-Fuzz applies Gaussian smoothing to the packet timestamps. Over multiple generations, this reduces the link variation in regions that are not relevant for triggering the poor behavior.

7.3.3 Traffic Fuzzing

CC-Fuzz uses the same algorithm (DISTPACKETS) for generating traffic traces, with some modifications.

1. **Trace generation heuristics:** We eliminate the local rate constraints, allowing bursty cross traffic.
2. **Crossover operation:** By eliminating local rate constraints, we define the crossover operation as follows: randomly choose a split point by packet count, randomly select the left half of one trace and the right half of the other trace around the split point, and combine the two sets of timestamps.

In the case of traffic fuzzing, it is desirable to generate "minimal" traffic injection vectors that induce poor behavior in CCAs. For example, a large burst of cross traffic where many packets of the cross traffic are lost will have the same impact if the lost packets were never sent. In addition, cross traffic arrives and departs the bottleneck queue when the CCA under test is silent (e.g. when TCP is waiting for ACKs after filling the CWND) has no impact.

In order to impose these properties, we allow a variable number of cross traffic packets up to a maximum limit. When regenerating a portion of the trace during a mutation operation, the number of packets in that portion are changed randomly. During a crossover operation, the number of traffic packets naturally change based on whether the trace on the right side has more or less packets. This is combined with a change to the scoring function (§ 7.3.4) in order to implicitly impose constraints like minimizing the number of cross-traffic packets required to trigger poor performance in the CCA being tested.

7.3.4 Scoring Function

The scoring function is a key aspect of a genetic algorithm - it determines which traces were successful in triggering specific performance behavior, and allows implicit modeling of desirable properties in a link or traffic trace. As part of calculating the score for a trace, CC-Fuzz runs the CCA using the link or traffic trace (simulated using NS3. Emulation using tools like MahiMahi can also be used - comparison in Section 7.3.6), and analyzes the queuing behavior. The score assigned to each trace in a generation has two components: performance score and trace score.

Performance Score. The performance score can be designed for specific types of poor behavior like high loss rate, high delay or low utilization. For example, for quantifying low utilization, CC-Fuzz calculates windowed throughput for the run, and takes the average of the lowest 20% of the windows. Compared to using the overall throughput, this prevents the algorithm preferring traces that trigger poor behavior early on, improving trace diversity.

Trace Score. In order to model properties of the traces that are hard to model during trace generation, each trace can be assigned a score based on how well it satisfies the desired properties. For example, CC-Fuzz scores traffic traces using the (negation of) total traffic packets and the total traffic packets dropped in order to make the genetic algorithm gravitate towards generating minimal traffic vectors.

7.3.5 Selection Algorithm

Once the traces in a generation have been assigned scores, we rank the traces from highest score to lowest score. We first pick *kElite* of the highest scored traces that make it to the next generation unchanged. We assign a relative probability of $\frac{1}{rank}$ to each trace and then choose *kCrossover* pairs of traces according to these probabilities, and combine them for generating crossover traces. The same probabilities (based on rank) are used for picking traces that undergo mutation for generating the rest of the traces in order to maintain a constant population size.

7.3.6 Emulation vs. Simulation

Our current implementation of CC-Fuzz uses NS3-based simulation for evaluating a CCA's performance for a link or traffic trace. An alternative is to use a network emulator like MahiMahi. In either case, CC-Fuzz will test a combination of the CCA implementation and the run-time framework, finding failures in either system and their interactions.

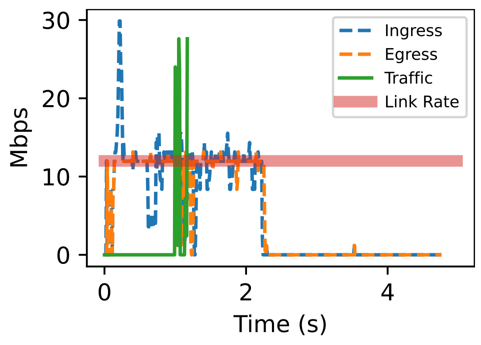
The benefit of emulation is the ability to test a real implementation of a CCA. Unfortunately, emulating multiple traces in parallel in a reproducible manner is challenging. We need to ensure that the performance is not affected due to CPU and memory bottlenecks, and that the start time of a flow is synchronized with the network trace. Otherwise, the CCA behavior can be very different across generations for a given trace, which can delay or even prevent convergence of the genetic algorithm.

Simulation, on the other hand, will generate identical results across repeated runs, resulting in faster convergence. In addition, for link rates in 10s of Mbps, simulation is likely to be faster than real-time emulation, and the results of the simulation are not affected by machine load - this makes it easy to massively parallelize the algorithm on a single machine. The key drawback of simulation is that it does not test the actual implementation, but a re-implementation in the simulation framework (e.g. NS3). Tools like DCE [175] can mitigate this drawback by simulating real network stacks.

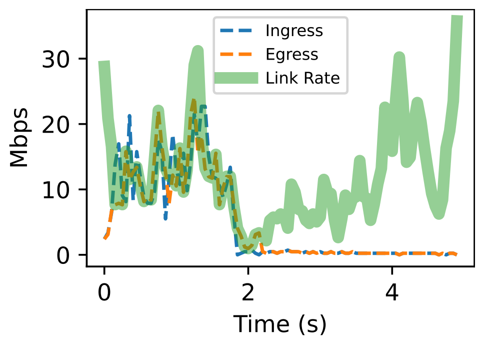
In addition, randomization in a CCA's implementation can also prevent convergence. In such cases, we need to modify the CCA implementation so that the randomization is repeatable (fix the random seed). This is much easier in a simulated setup as opposed to modifying kernel CCA code in an emulated environment. In the future, we plan to explore the use of emulation for CC-Fuzz.

7.4 Findings

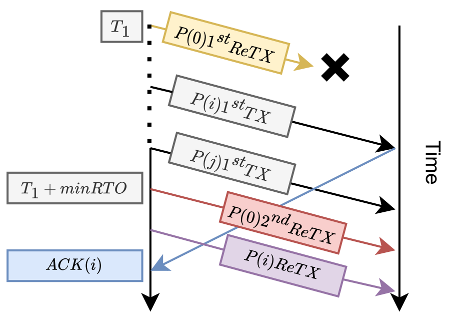
In this section, we will discuss some interesting findings that CC-Fuzz was able to automatically discover. For all of our tests, we set the bottleneck bandwidth to 12 Mbps (average bandwidth in the case of link fuzzing) and set the propagation delay of the bottleneck link to 20 ms. TCP-SACK and delayed ACKs are enabled (Linux defaults), and min-RTO is set to 1 second (as per RFC 6298/2.4, Linux uses 200 ms). We use a population size of 500, and use an island-isolation [176] strategy with 20 islands for solution diversity, where 10% of the traces migrate every 10 generations. Across island generations, the best trace is preserved (*kElite = 1*), 30% of the traces are crossovers, and the rest are mutations.



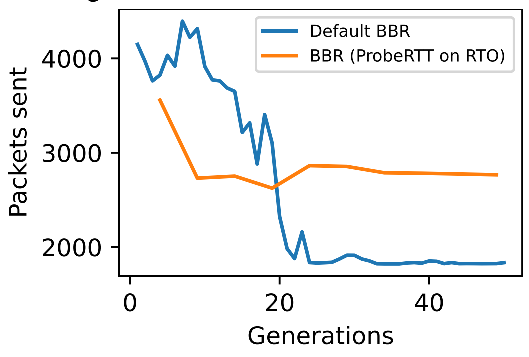
(a) CC-Fuzz traffic trace that causes BBR to get stuck.



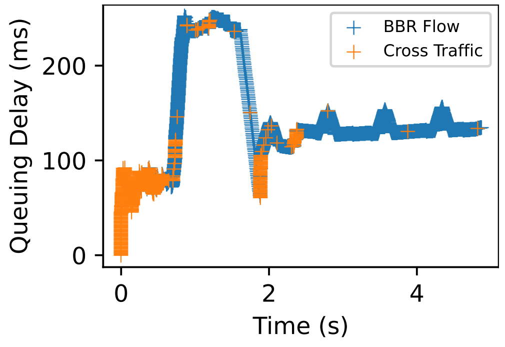
(b) CC-Fuzz link trace that causes BBR to get stuck.



(c) Timeline showing how BBR's bug is triggered.



(d) CC-Fuzz performance with and without BBR patch.



(e) CC-Fuzz triggering high delays in BBR with cross traffic.

Figure 7.4: Analyzing BBR with CC-Fuzz.

7.4.1 BBR - Stuck Throughput

We tested NS3's version of TCP-BBR with CC-Fuzz, and after a few generations, it produced traces that triggered low throughput for BBR where it get's stuck permanently. One such trace is shown in Figure 7.4a. For understanding the root cause, we dug into NS3 code and generated various internal logs from BBR's code and from the NS3 TCP socket code.

BBR uses an 8-RTT gain cycle for estimating bandwidth, where it sends at 1.25X the current bandwidth estimate for the first RTT, 0.75X on the second RTT and at 1X for the rest of the gain cycle. Each RTT is considered as a probing round. The measured rate in each probing round is processed through a windowed max-filter that keeps the estimates from the last 10 rounds of probing.

We found the root cause to be BBR's mechanism for timing it's bandwidth probing cycles in terms of RTT. For each packet, the TCP send buffer tracks the number of bytes delivered when that packet was sent in the SKB. At the beginning of a probing round, BBR records the number of bytes delivered so far. The probe ends when the prior delivered of the packet most recently ACK (i.e. bytes delivered when the ACKed packet was sent) exceeds the bytes delivered at the beginning of the probing round.

Suppose a packet $P(0)$ is transmitted at time T_0, and is lost. Fast retransmit will cause the first retransmission to occur at some time $T_1 > T_0 + RTT$, and an RTO timer will be set for $T_1 + minRTO$. At $T_1 + minRTO$, $P(0)$ is retransmitted for the second time. Suppose $P(i)...P(j)$ were the last few packets sent before the second retransmission for $P(0)$, and the SACKs for these have not arrived yet. After transmitting $P(0)$ for the second time, $P(i)$ will be transmitted again (a spurious retransmission). Here, the prior delivered for $P(i)$ is updated in the SKB for $P(i)$ to the current bytes delivered. If the SACK for the original transmission of $P(i)$ arrives right after the second transmission of $P(i)$, BBR will prematurely end the current probe cycle, since the value of prior delivered for $P(i)$ increased when the spurious retransmission was sent, and now likely exceeds threshold at which the current probing round was supposed to end. This sequence of events is depicted in Figure 7.4c. Thus, BBR's rate sample is now incorrect, as it is using the time and bytes delivered between the ACK for the original packet, and the packet's spurious retransmission, to calculate the rate. This can result in a low value for the bandwidth sample. This can repeat for the other packets $P(i + 1)...P(j)$ that were in-flight when $P(0)$ was transmitted the second time. If this continues for 10 or more packets, the true bandwidth estimates in the bandwidth max-filter expire, and BBR's bandwidth estimate becomes low. With a very low bandwidth estimate, delayed ACKs can cause a positive feedback loop, causing BBR to send slower and slower, stalling BBR indefinitely. It is possible that this is the same issue being referenced in [177].

CC-Fuzz was able to trigger this behavior with both, link fuzzing and traffic fuzzing. Figure 7.4b shows a link trace generated by CC-Fuzz that triggers the same bug. The traffic trace generated by CC-Fuzz is very easy to understand - CC-Fuzz's implicit constraints on traffic traces generate a clean, minimal trace. On the other hand, despite our trace annealing mechanism significantly smoothing out the bandwidth variations, the link trace is harder to reason about. In the future, we plan to implement better heuristics and implicit constraints in order to generate easier to understand link traces that trigger poor behavior.

In order to try and mitigate this behavior, we made BBR trigger a minRTT probe when an

RTO occurs - this slows down BBR momentarily which allows BBR to receive the in-flight ACKs, and thus avoid the spurious retransmissions that cause poor RTT-clocking for BBR's bandwidth probes. Figure 7.4d plots the average of the top 20 traces with the lowest throughput in each generation. Our proposed fix reduces throughput a little bit, but avoids the permanent stalling behavior observed in BBR without the fix.

7.4.2 TCP-CUBIC Incorrect CWND Update

When testing TCP-CUBIC, we discovered a bug in NS3's implementation of CUBIC's window update during slow start. When a packet is lost, and it's retransmission triggered by fast retransmit is also lost, the CCA goes into slow start after RTO. The sender performs a second retransmission for the packet, and when the ACK for this is received, there is a large jump in the cumulative ACK. CUBIC's slow start window-increase function is called with the large number of segments ACKed. At this point, the CWND must only be increased upto the slow-start threshold. In NS3, this check is not performed, and the congestion window is increased by a large value - causing CUBIC to send almost 1-RTO (1 second in the case of NS3) worth of pending data, causing catastrophic losses. This leads to CUBIC going into slow start again. As of commit `60e1e403`, this bug is still present in NS3. This computation is performed correctly in the Linux kernel source code.

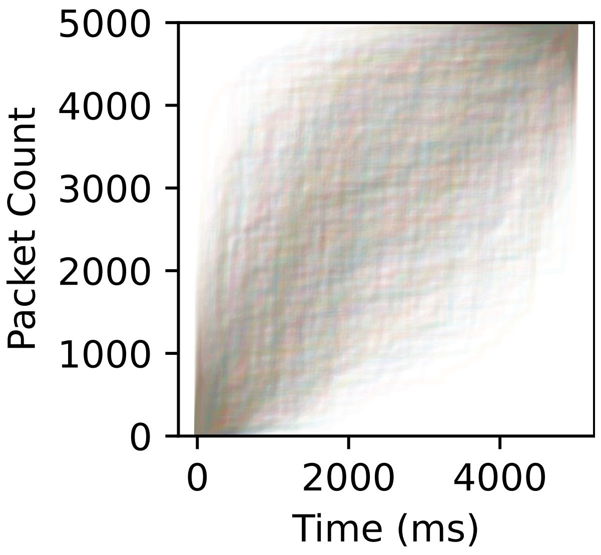
7.4.3 Other Findings

For TCP-Reno, CC-Fuzz was able to find a traffic trace similar to the well-known low-rate TCP attack [168] for a single flow, where traffic bursts cause the same packet sequence to get lost after each retransmission, which triggers exponential RTO back-off. This prevented Reno from ever ramping up after the initial slow start phase. CC-Fuzz can also test CCAs for goals other than low throughput, by just changing the performance component of the scoring function. For example, we ran traffic fuzzing on BBR with the goal of inducing high delays by setting the score function to the 10th percentile delay. This caused CC-Fuzz to generate a traffic vector that (1) fills up the queue just before BBR starts, so that BBR cannot see the true link RTT, and (2) injects traffic right after BBR's slow start phase to accelerate queue-growth caused by BBR. This is shown in Figure 7.4e.

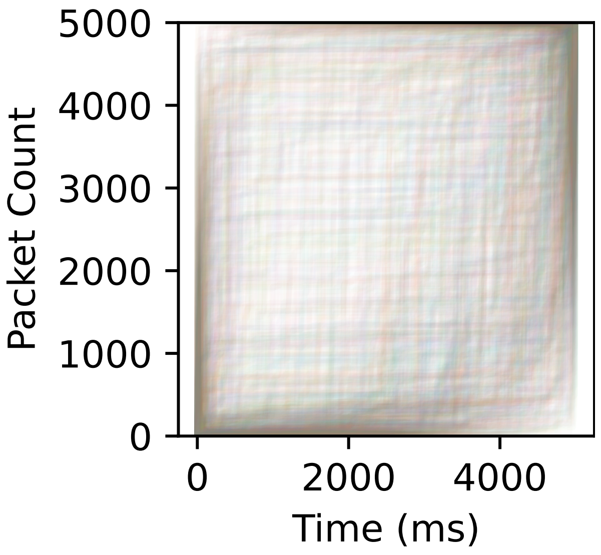
7.5 Future Directions

Realism Scoring. The current version of CC-Fuzz uses a heuristic-based approach for generating realistic traces.

An alternate technique that can be used for generating realistic traces is to use aggregate performance across multiple CCAs as a score function to quantify the realism of a trace, assigning high scores to traces under which at least a few algorithms perform well, and vice versa. Figure 7.5 shows the traces accepted and rejected by this mechanism. Note how traces that have low bandwidth initially and higher bandwidth later are rejected - such traces will naturally cause low throughput in most CCAs.



(a) Valid traces.



(b) Invalid traces.

Figure 7.5: Distribution of service curves according to realism scores assigned by testing on multiple CCAs. The traces were generated with DISTPACKETS, but without the local rate constraints.

In order to reduce the amount of computation required, the realism score can be computed every few iterations instead of every iteration, or can be computed for a single randomly chosen CCA instead of all CCAs in each generation.

Diversity and Semantic Scoring. Currently, CC-Fuzz tends to converge at a point where most traces trigger the easiest to induce performance bug. In order to find other bugs, an iterative process of fixing the bug and retesting, or defining a score function that negatively weights the manifestation of that bug can be used. In order to make CC-Fuzz automatically find a diverse set of bugs, machine learning techniques could be used to classify the different behaviors. This information could be used to drop traces that trigger similar bugs across generations. Another potential direction for future work is to create a framework that translates logical specifications of performance goals into score functions, so that the user does not have to come up with complex score functions themselves in order to make CC-Fuzz work.

Random Losses and Combined Fuzzing. Random packet losses are common on wireless links. CC-Fuzz's two modes, link, and traffic fuzzing, do not cover scenarios where random losses occur without a corresponding queue build up. Loss fuzzing can be added to the set of network models in CC-Fuzz in order to increase the testing coverage. Another potential direction is to combine link, traffic, and loss fuzzing into a single process. Combined fuzzing will result in much more complex network traces that include link variations, cross traffic and loss - these are harder to understand, and thus, it is harder to pin-point the bug.

7.6 Conclusion

In this chapter, we have presented the design of an automated congestion control testing tool, CC-Fuzz. Our results are highly promising with an initial prototype of CC-Fuzz are finding both known and unknown issues with existing well-tested CCAs. We believe that with further development, CC-Fuzz could fill an important gap in the development of new CCAs for emerging applications by providing a simple way to identify environments in which a particular CCA performs poorly.

Chapter 8

Conclusion

Video streaming is one of the most ubiquitous class of applications that run on the Internet today. The Internet presents many challenges for real-time streaming video, like packet loss, bandwidth limitations and variability, and network delay. Systems designed for video streaming applications must navigate these challenges in order to achieve high QoE. Significant past research has focused on optimizing video streaming systems for tasks like VOD streaming and real-time video conferencing, where the QoE requirement is a single point on a video quality-delay trade-off curve. Today, there are various emerging video streaming applications like social live video streaming, cloud gaming, and cloud AR/VR which have unique and demanding QoE requirements, and existing systems are unable to satisfy the needs of emerging applications. For instance, social live video streaming requires operation at multiple points on the video quality-delay trade-off curve in order to satisfy the needs of different time-shifted viewers, and the QoE requirements of cloud gaming and cloud AR/VR applications lie beyond the typical video quality-delay trade-offs that are achieved by existing systems, since they simultaneously require extremely high video quality and extremely low end-to-end video frame delay.

In this thesis, we discuss various projects that have two key underlying design principles:

1. **Tailored designs** We show in this thesis that carefully considering the QoE requirements and designing video systems that make key application-appropriate trade-offs is important in order to meet the unique and demanding needs of various emerging video streaming applications, as opposed to using highly optimized techniques that are more general.

2. **Holistic design** We show in this thesis that carefully considering the interactions between various aspects of video streaming, like the amount of available bandwidth, the properties of video compression mechanisms, loss recovery techniques, and congestion control enables better trade-offs and improves the QoE of emerging video streaming applications.

We show that combining these two principles can lead to highly optimized video streaming systems for specific applications and environments, where the QoE achieved is much higher than what can be achieved using existing video streaming techniques.

8.1 Key Takeaways

In this thesis, we presented various practical designs for addressing the unique and demanding QoE of emerging video streaming applications.

We first explored the application space of social live video streaming, where real-time viewers require low delay and interactivity, whereas delayed viewers demand higher video quality. Our system, Vantage addresses this by leveraging the relationship between bitrate, and video quality, and optimizes video quality accounting for the distribution of the viewing delays. Vantage's design demonstrates that carefully considering the specific requirements of the application (i.e. diverse viewing delays), and using holistic design (i.e. by using codec- and transport-layer optimizations to maximize a QoE metric like video quality) can achieve significantly higher QoE than existing techniques.

We then explored the space of ultra-low latency immersive applications like cloud gaming, cloud AR and VR - these applications have demanding QoE requirements, and require high video quality and low latency simultaneously. We proposed two key designs - Prism and SQP - which deal with loss recovery and congestion control for cloud-rendered applications.

Prism proposes a system design that enables high video quality and low latency during packet loss. Prism's design shows that carefully optimizing the video codec using deep insights into the properties of different types of compressed video, leveraging the application-specific bandwidth regime for cloud gaming, and leveraging network-level optimizations like packet loss prediction can achieve significantly higher QoE in the face of packet loss compared to existing systems, where the various subsystems are more general and designed in isolation.

SQP is a congestion control algorithm for achieving the demanding bandwidth requirements and extremely low end-to-end frame delay for cloud gaming and AR/VR streaming applications. SQP's tightly integrated video coding and network measurements, and key application-specific trade-offs enable SQP to achieve high throughput on time-varying links and in the presence of competing queue-building flows, while simultaneously achieving low end-to-end video frame delay.

In addition to the projects above, we discuss two additional projects. ViXNN is an end-to-end video compression technique using neural networks which learns a loss-resilient video compression scheme by simulating loss during the training phase. ViXNN generates video compression schemes that are optimized for rate-distortion, loss resiliency, and specific types of video content. CC-Fuzz is a tool that uses a genetic algorithm to generate adversarial network traces with the goal of testing the design and implementation of new congestion control algorithms, in order to detect issues with rate adaptation and loss recovery.

8.2 Future Work

As the state of network technologies and video compression techniques evolve with hardware and software advancements, there will be additional opportunities for performing cross-layer optimizations in order to address the QoE demands of the growing space of emerging video streaming applications. For instance, Prism transmits two parallel streams, and these streams share a significant amount of redundancy between them. If hardware video codecs enable support for SVC or MDC codecs, systems like Prism can be adapted to use these new codecs in order to leverage the benefits of reduced redundancy across multiple video streams in SVC and MDC. Similarly, for Vantage, using SVC would eliminate the need for storing high quality frames separately, and will also eliminate the need to use a separate video encoder for the enhancement stream - the scheduler will simply determine which enhancement layers must be transmitted and for which frames. In the space of congestion control, wide scale deployment of AQM schemes [178] can enable additional optimizations that can further improve CCA performance to meet the demanding requirements of cloud streaming applications. In addition, the SSIM modeling techniques we developed can be used for more wide-ranging applications like designing CCAs that account for video quality in order to achieve QoE-based fairness across multiple video streams [179]. With advancements in neural video compression techniques [180] that also leverage temporal redundancy in video data, and with improvements in neural-network performance stemming from hardware advancements, using neural networks for optimized end-to-end video compression can be a viable approach in the future.

Bibliography

[1] Devdeep Ray, Jack Kosaian, K. V. Rashmi, and Srinivasan Seshan. Vantage: Optimizing video upload for time-shifted viewing of social live streams. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 380–393, New York, NY, USA, 2019. Association for Computing Machinery. (document), 1.4.1

[2] Matthew K Mukerjee, Ilker Nadi Bozkurt, Devdeep Ray, Bruce M Maggs, Srinivasan Seshan, and Hui Zhang. Redesigning cdn-broker interactions for improved content delivery. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 68–80, 2017. (document)

[3] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015. (document), 3.2, 3.3.3, 3.2, 3.6.3, 4.7.2.1, 4.7.3, 6.1.1, 6.4

[4] Xiph.org Test Media. `https://media.xiph.org/`. Last accessed 27 January 2019. (document), 3.2, 3.3, 3.4.1, 3.6.3, 3.6, 3.6.3

[5] Colin Levy and Ton Roosendaal. Sintel. In *ACM SIGGRAPH ASIA 2010 Computer Animation Festival*, SA '10, pages 82:1–82:1, New York, NY, USA, 2010. ACM. (document), 3.3, 3.4.1

[6] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2016-2021. Technical Report 1465272001663118, September 2017. 1, 3.1

[7] Katrin Scheibe, Kaja J Fietkiewicz, and Wolfgang G Stock. Information behavior on social live streaming services. *Journal of Information Science Theory and Practice*, 4(2):6–20, 2016. 1

[8] Michael Jarschel, Daniel Schlosser, Sven Scheuring, and Tobias Hoßfeld. An evaluation of qoe in cloud gaming based on subjective tests. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 330–335. IEEE, 2011. 2

[9] Mpeg-dash standard. `https://mpeg.chiariglione.org/standards/mpeg-a/mpeg-dash`. Last accessed 23 June 2019. 1.1, 3.6.2, 3.8

[10] Http live streaming. `https://developer.apple.com/streaming/`. Last accessed 19 June 2018. 1.1, 3.6.2, 3.8

[11] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Ja-

cobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016. 1.1, 1.4.4, 2, 6.3, 6.7

[12] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008. 1.1, 1.4.4, 1, 1, 6.7

[13] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and Design of the Google Congestion Control for Web Real-time Communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems (MMSys 16)*, 2016. 1.1, 2

[14] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018. 1.1, 3.4.1.2, 3.5.7, 3.6.2, 3.8, 4.8, 6.3, 6.4.2, 6.7.10

[15] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 509–522, 2015. 1.1

[16] Stadia. https://stadia.google.com/. 1.2, 4

[17] Geforce now gaming anywhere —& anytime. https://www.nvidia.com/en-us/geforce-now/. 1.2, 4

[18] Amazon luna: Amazon's cloud gaming service. 1.2

[19] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014. 1.4.1

[20] Szymon Jakubczak and Dina Katabi. Softcast: Clean-slate scalable wireless video. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students*, pages 9–12, 2010. 1.4.3, 2, 5.2.2, 5.4.5

[21] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 329–342, 2018. 1.4.4, 6.1.3, 2, 5, 6.3, 6.7, 7

[22] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, 2013. 1.4.4, 3.1, 3.3.3, 6.1.2.1, 2, 6.3, 6.4.2, 6.7, 7

[23] Wright Stevens et al. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. 1997. 1

[24] Nassima Bouzakaria, Cyril Concolato, and Jean Le Feuvre. Overhead and performance of low latency live streaming using mpeg-dash. In *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*, pages 92–97. IEEE, 2014. 3

[25] Kerem Durak, Mehmet N Akcay, Yigit K Erinc, Boran Pekel, and Ali C Begen. Evaluating the performance of apple's low-latency hls. In *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6. IEEE, 2020. 3

[26] Henning Schulzrinne, Stephen Casner, Ron Frederick, Van Jacobson, et al. Rtp: A transport protocol for real-time applications, 1996. 3

[27] Zhou Wang. The ssim index for image quality assessment. *https://ece. uwaterloo. ca/˜ z70wang/research/ssim*, 2003. 2.2.1, 4.3.1

[28] Boon-Lock Yeo and Bede Liu. A unified approach to temporal segmentation of motion jpeg and mpeg compressed video. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 81–88. IEEE, 1995. 2.2.1

[29] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003. 2.2.2, 2.2.3, 4.7.2.1

[30] Dan Grois, Detlev Marpe, Amit Mulayoff, Benaya Itzhaky, and Ofer Hadar. Performance comparison of h. 265/mpeg-hevc, vp9, and h. 264/mpeg-avc encoders. In *2013 Picture Coding Symposium (PCS)*, pages 394–397. IEEE, 2013. 2.2.3

[31] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of vp8, an open source video codec for the web. In *2011 IEEE International Conference on Multimedia and Expo*, pages 1–6. IEEE, 2011. 2.2.3

[32] Debargha Mukherjee, Jim Bankoski, Adrian Grange, Jingning Han, John Koleszar, Paul Wilkins, Yaowu Xu, and Ronald Bultje. The latest open-source video codec vp9-an overview and preliminary results. In *2013 Picture Coding Symposium (PCS)*, pages 390–393. IEEE, 2013. 2.2.3

[33] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. An overview of core coding tools in the av1 video codec. In *2018 Picture Coding Symposium (PCS)*, pages 41–45. IEEE, 2018. 2.2.3

[34] Facebook Live. `https://live.fb.com/`. Last accessed 18 June 2018. 3.1

[35] Youtube-Live. `https://www.youtube.com/channel/UC4R8DWoMoI7CAwX8_LjQHig`. Last accessed 18 June 2018. 3.1

[36] Periscope. `https://www.pscp.tv/`. Last accessed 19 June 2018. 3.1

[37] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y Zhao. Anatomy of a Personalized Livestreaming System. In *Proceedings of the 2016 Internet Measurement Conference (IMC 16)*, 2016. 3.1, 3.3.1, 3.3.2, 3.5.6

[38] Xiaodong Wang, Ye Tian, Rongheng Lan, Wen Yang, and Xinming Zhang. Beyond the Watching: Understanding Viewer Interactions in Crowdsourced Live Video Broadcasting Services. *IEEE Transactions on Circuits and Systems for Video Technology*, 2018. 3.1, 3.3.2, 3.4.2.1

[39] Hangouts On Air with YouTube Live. `https://support.google.com/youtube/answer/7083786?hl=en`. Last accessed 4 July 2018. 3.1, 3.3.2

[40] More Ways To Connect with Friends in Facebook Live. https://newsroom.fb.com/news/2017/05/more-ways-to-connect-with-friends-in-facebook-live/. Last accessed 4 July 2018. 3.1, 3.3.2

[41] Douglas Soo. Twitch Engineering: An Introduction and Overview. https://bit.ly/2JGR5yb, 2015. Last accessed 19 June 2018. 3.3.1

[42] Facebook Live video for News Feed (part 2). https://atscaleconference.com/videos/facebook-live-video-for-news-feed-part-2/, 2017. Last accessed 19 June 2018. 3.3.1

[43] Real-Time Messaging Protocol (RTMP) Specification. https://www.adobe.com/devnet/rtmp.html. Last accessed 19 June 2018. 3.3.1, 3.8

[44] WebRTC. https://webrtc.org/. Last accessed 19 June 2018. 3.3.1, 3.4.1.2, 3.8

[45] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT 12)*, 2012. 3.3.1

[46] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A Practical Prediction System for Video QoE Optimization. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016. 3.3.1

[47] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017. 3.3.1, 3.8

[48] Tsahi Levent-Levi. 10 Massive Applications Using WebRTC. https://bloggeek.me/massive-applications-using-webrtc/, 2017. Last accessed 16 July 2018. 3.3.1

[49] Luis Teixeira. Rate-distortion Analysis for H.264/AVC Video Statistics. In *Recent Advances on Video Coding*. InTech, 2011. 3.4.1, 3.4.2.2

[50] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Trans. Netw.*, 22(1):326–340, February 2014. 3.4.1.1

[51] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 187–198, New York, NY, USA, 2014. ACM. 3.4.1.1

[52] Reza Rejaie, Mark Handley, and Deborah Estrin. Layered quality adaptation for internet video streaming. *IEEE Journal on Selected Areas in Communications*, 18(12):2530–2543, 2000. 3.4.1.1

[53] Luca De Cicco, Gaetano Carlucci, and Saverio Mascolo. Experimental investigation of the google congestion control for real-time flows. In *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, FhMN '13, pages

21–26, New York, NY, USA, 2013. ACM. 3.4.1.2

[54] Luigi Rizzo. Dummynet: a Simple Approach to the Evaluation of Network Protocols. *ACM SIGCOMM Computer Communication Review (CCR)*, 1997. 3.4.1.4

[55] Open Broadcaster Software. `https://obsproject.com/`. Last accessed 19 June 2018. 3.4.1.4

[56] Zhenyu Li, Mohamed Ali Kaafar, Kave Salamatian, and Gaogang Xie. Characterizing and Modeling user Behavior in a Large-scale Mobile Live Streaming System. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(12):2675–2686, 2017. 3.4.2.1

[57] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007. 3.4.2.2, 3.8, 2, 5.2.2

[58] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018. 3.5.7, 3.6.3

[59] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 3.6.1, 5.5

[60] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016. 3.6.1

[61] Stefan Winkler and Praveen Mohandas. The evolution of video quality measurement: from psnr to hybrid metrics. *IEEE Transactions on Broadcasting*, 54(3):660–668, 2008. 3.6.1

[62] John C Tang, Gina Venolia, and Kori M Inkpen. Meerkat and Periscope: I Stream, you Stream, Apps Stream for Live Streams. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI 16)*, 2016. 3.6.3

[63] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling Packet Loss in WebRTC. In *2013 20th IEEE International Conference on Image Processing (ICIP 13)*, 2013. 3.6.3

[64] Jongwon Yoon, Honghai Zhang, Suman Banerjee, and Sampath Rangarajan. MuVi: A Multicast Video Delivery Scheme for 4G Cellular Networks. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (Mobicom 12)*, 2012. 3.8

[65] X Rex Xu, Andrew C Myers, Hui Zhang, and Raj Yavatkar. Resilient Multicast Support for Continuous-Media Applications. In *Proceedings of the IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 97)*, 1997. 3.8

[66] Ming Tang, Lin Gao, Haitian Pang, Jianwei Huang, and Lifeng Sun. Optimizations and Economics of Crowdsourced Mobile Streaming. *IEEE Communications Magazine*, 55(4):21–27, 2017. 3.8

[67] Aiman Erbad and Charles Buck Krasic. Sender-side buffers and the case for multimedia adaptation. *Communications of the ACM*, 55(12):50–58, 2012. 3.8

[68] Hamed Ahmadi, Omar Eltobgy, and Mohamed Hefeeda. Adaptive multicast streaming of virtual reality content to mobile users. In *Proceedings of the on Thematic Workshops of ACM Multimedia 2017*, 2017. 3.8

[69] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. Favor: Fine-Grained Video Rate Adaptation. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys 18)*, 2018. 3.8

[70] Xing Liu, Qingyang Xiao, Vijay Gopalakrishnan, Bo Han, Feng Qian, and Matteo Varvello. 360 Innovations for Panoramic Video Streaming. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets 17)*, 2017. 3.8

[71] Kaixuan Long, Chencheng Ye, Ying Cui, and Zhi Liu. Optimal Multi-Quality Multicast for 360 Virtual Reality Video. *arXiv preprint arXiv:1901.02203*, 2019. 3.8

[72] Cloud gaming (beta) with xbox game pass: Xbox. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming/home. 4

[73] Google cloud streams augmented reality. 4

[74] Chrome remote desktop. 4

[75] Deploy and scale your virtualized windows desktops and apps on azure. 4

[76] Luca De Cicco, Saverio Mascolo, and Vittorio Palmisano. Feedback control for adaptive live video streaming. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 145–156, 2011. 4

[77] Rohit Puri, Kannan Ramchandran, Kang-Won Lee, and Vaduvur Bharghavan. Forward error correction (fec) codes based multiple description coding for internet video streaming and multicast. *Signal Processing: Image Communication*, 16(8):745–762, 2001. 1

[78] Yanlin Liu and Mark Claypool. Using redundancy to repair video damaged by network data loss. In *Multimedia Computing and Networking 2000*, volume 3969, pages 73–84. International Society for Optics and Photonics, 1999. 1

[79] Nick Feamster and Hari Balakrishnan. Packet loss recovery for streaming video. In *12th International Packet Video Workshop*, pages 9–16. PA: Pittsburgh, 2002. 2, 4.1

[80] Mark Claypool and Yali Zhu. Using interleaving to ameliorate the effects of packet loss in a video stream. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 508–513. IEEE, 2003. 2

[81] Yanling Xu and Yuanhua Zhou. H. 264 video communication based refined error concealment schemes. *IEEE Transactions on Consumer Electronics*, 50(4):1135–1141, 2004. 2

[82] Bo Yan and Hamid Gharavi. Efficient error concealment for the whole-frame loss based on h. 264/avc. In *2008 15th IEEE International Conference on Image Processing*, pages 3064–3067. IEEE, 2008. 2

[83] Bernd Girod, Klaus Werner Stuhlmueller, M Link, and Uf Horn. Packet-loss-resilient internet video streaming. In *Visual Communications and Image Processing'99*, volume 3653, pages 833–844. International Society for Optics and Photonics, 1998. 2

[84] Callstats. Error resilience mechanisms for webrtc video communications. 3

[85] ST Worrall, AH Sadka, AM Kondoz, and P Sweeney. Motion adaptive intra refresh for mpeg-4. *Electronics Letters*, 36(23):1924–1925, 2000. 4

[86] Jason Greengrass, John Evans, and Ali C Begen. Not all packets are equal, part 2: The impact of network packet loss on video quality. *IEEE Internet Computing*, 13(2):74–82, 2009. 4.1

[87] Anna Giannakou, Dipankar Dwivedi, and Sean Peisert. A machine learning approach for packet loss prediction in science flows. *Future Generation Computer Systems*, 102:190–197, 2020. 4.1

[88] Lopamudra Roychoudhuri and Ehab S Al-Shaer. Real-time packet loss prediction based on end-to-end delay variation. *IEEE transactions on Network and Service Management*, 2(1):29–38, 2005. 4.1

[89] Measurement Lab NDT 2021 Data. The M-Lab NDT data set. `https://measurementlab.net/tests/ndt`. 4.2, 4.5.4, 4.5.4.1, 4.6, 4.7.2

[90] Didier J Le Gall. The mpeg video compression algorithm. *Signal Processing: Image Communication*, 4(2):129–140, 1992. 4.3

[91] Lavfi. 4.3.1

[92] Zhi Li, Christos Bampis, Julie Novak, Anne Aaron, Kyle Swanson, Anush Moorthy, and JD Cock. Vmaf: The journey continues. *Netflix Technology Blog*, 25, 2018. 4.3.1

[93] 4.3.2, 4.6, 4.7

[94] Andy Backhouse and Irene YH Gu. A bayesian framework-based end-to-end packet loss prediction in ip networks. In *IEEE Sixth International Symposium on Multimedia Software Engineering*, pages 35–42. IEEE, 2004. 4.4.1

[95] Roger Immich, Pedro Borges, Eduardo Cerqueira, and Marilia Curado. Qoe-driven video delivery improvement using packet loss prediction. *International Journal of Parallel, Emergent and Distributed Systems*, 30(6):478–493, 2015. 4.4.1

[96] Neal Cardwell, Yuchung Cheng, S Hassas Yeganeh, and Van Jacobson. Bbr congestion control. *Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00*, 2017. 4.4.1, 7

[97] TV Lakshman, Upamanyu Madhow, and Bernhard Suter. Tcp/ip performance with random loss and bidirectional congestion. *IEEE/ACM transactions on networking*, 8(5):541–555, 2000. 4.4.1

[98] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016. 4.6

[99] Abhijit Patait and Eric Young. High performance video encoding with nvidia gpus. In *2016 GPU Technology Conference (https://goo. gl/Bdjdgm)*, 2016. 4.7.2.1

[100] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006. 2

[101] Ian Swett. Quic fec v1. 4.8

[102] Introduction. 4.8

[103] Oculus vr headsets, games and equipment - meta quest. 5

[104] Introducing oculus air link, a wireless way to play pc vr games on oculus quest 2, plus infinite office updates, support for 120 hz on quest 2, and more. 5

[105] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell. Full Resolution Image Compression with Recurrent Neural Networks. *ArXiv e-prints*, August 2016. 5, 5.3.1, 5.4.1, 5.4.3, 5.5.1

[106] O. Rippel and L. Bourdev. Real-Time Adaptive Image Compression. *ArXiv e-prints*, May 2017. 5, 5.3.1

[107] Vivek K Goyal. Multiple description coding: Compression meets the network. *IEEE Signal processing magazine*, 18(5):74–93, 2001. 5.2.2

[108] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM. 5.3

[109] Jean-Yves Potvin. The traveling salesman problem: A neural network perspective. 5.3

[110] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, April 2016. 5.3

[111] Shibani Santurkar, David Budden, and Nir Shavit. Generative compression. *arXiv preprint arXiv:1703.01467*, 2017. 5.3.1, 5.3.2

[112] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 5.3.3

[113] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015. 5.4.2

[114] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework, 2018. 5.4.5

[115] XiangJi Wu, Ziwen Zhang, Jie Feng, Lei Zhou, and Junmin Wu. End-to-end optimized video compression with mv-residual prediction, 2020. 5.4.5

[116] R. Uetz and S. Behnke. Large-scale object recognition with cuda-accelerated hierarchical neural networks. In *2009 IEEE International Conference on Intelligent Computing and Intelligent Systems*, volume 1, pages 536–541, Nov 2009. 5.5

[117] Ffmpeg. `http://www.ffmpeg.org`. 5.5

[118] Laurent Aimar, Loren Merritt, Eric Petit, Min Chen, Justin Clay, Mns Rullgrd, Christian Heine, and Alex Izvorski. x264-a free h264/avc encoder, 2005. 5.5

[119] C Montgomery et al. Xiph. org video test media (derf's collection), the xiph open source

community, 1994. *Online, https://media. xiph. org/video/derf.* 5.5

[120] François Chollet. keras. `https://github.com/fchollet/keras`, 2015. 5.5

[121] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. 5.5

[122] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 5.5

[123] C. Torres-Huitzil and B. Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017. 5.5.3

[124] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 531–544, Boston, MA, 2017. USENIX Association. 5.5.4

[125] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016. 5.5.5

[126] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, Dec 2016. 5.5.5

[127] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017. 5.5.5

[128] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017. 5.5.5

[129] Jitendra Padhye, Victor Firoiu, Donald F Towsley, and James F Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM transactions on Networking*, 8(2):133–145, 2000. 1

[130] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018. 6.1.1, 6.4, 6.7.1, 7

[131] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994. 6.1.2.1

[132] Mario Hock, Felix Neumeister, Martina Zitterbart, and Roland Bless. Tcp lola: Congestion control for low latencies and high throughput. In *2017 IEEE 42nd Conference on*

Local Computer Networks (LCN), pages 215–218. IEEE, 2017. 6.1.2.1

[133] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 395–408, 2015. 6.1.2.2, 1, 3, 6.3, 6.7

[134] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. {PCC} vivace:{Online-Learning} congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018. 6.1.2.2, 6.1.2.2, 3, 6.3, 6.4.1, 6.7

[135] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. *arXiv preprint arXiv:1802.08730*, 2018. 6.1.3, 6.1.3, 7

[136] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for delay-sensitive congestion control. In *ANRW*, page 75, 2018. 2, 5, 6.3, 6.5.5

[137] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the internet measurement conference*, pages 137–143, 2019. 4, 7, 7.2

[138] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016. 3, 6.3, 7

[139] Facebookincubator. facebookincubator/mvfst. 1, 6.8

[140] Mario Hock, Felix Neumeister, Martina Zitterbart, and Roland Bless. Tcp lola: Congestion control for low latencies and high throughput. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pages 215–218. IEEE, 2017. 6.3

[141] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994. 6.3

[142] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. Performance evaluation of webrtc-based video conferencing. *SIGMETRICS Perform. Eval. Rev.*, 45(3):56–68, mar 2018. 6.3

[143] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010. 6.3

[144] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. {ABC}: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020. 6.3

[145] Yongguang Zhang and Thomas R Henderson. An implementation and experimental study

of the explicit control protocol (xcp). In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1037–1048. IEEE, 2005. 6.3

[146] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015. 6.3

[147] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020. 6.3

[148] Nitin Garg. Copa congestion control for video performance, Mar 2020. 3, 6.8

[149] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 43–56, New York, NY, USA, 2000. Association for Computing Machinery. 6.4.2, 6.7.9

[150] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196, 2017. 6.5.1

[151] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions On Networking*, 12(6):963–977, 2004. 6.5.2

[152] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. Computing tcp's retransmission timer. Technical report, rfc 2988, November, 2000. 6.5.3

[153] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998. 6.5.4

[154] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 137–143, New York, NY, USA, 2019. Association for Computing Machinery. 6.6.1

[155] Bbrv1: Linux kernel. 6.7.1

[156] Ajay Tirumala. Iperf: The tcp/udp bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/*, 1999. 6.7.1

[157] PCC. `https://github.com/modong/pcc`, 2016. 6.7.1

[158] Vivace. `https://github.com/PCCproject/PCC-Uspace/tree/`

NSDI-2018, 2016. 6.7.1

[159] genericCC. https://github.com/venkatarun95/genericCC, 2018. 6.7.1

[160] Yan Liu and Jack YB Lee. Streaming variable bitrate video over mobile networks with predictable performance. In *2016 IEEE Wireless Communications and Networking Conference*, pages 1–7. IEEE, 2016. 6.7.2

[161] Jim Gettys. Bufferbloat: Dark buffers in the internet. *IEEE Internet Computing*, 15(3):96–96, 2011. 6.7.6

[162] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984. 6.7.9

[163] Ingemar Johansson and Zaheduzzaman Sarker. Self-clocked rate adaptation for multimedia. Technical report, 2017. 7

[164] Matthew K Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C Snoeren. Adapting {TCP} for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, 2020. 7

[165] Ravi Netravali, Anirudh Sivaraman, Keith Winstein, Somak Das, Ameesh Goyal, and Hari Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement. *ACM SIGCOMM Computer Communication Review*, 44(4):129–130, 2014. 7, 7.3.2

[166] Wikipedia contributors. Sisyphus — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sisyphus&oldid=1091831787, 2022. [Online; accessed 23-June-2022]. 1

[167] Gustavo Carneiro. Ns-3: Network simulator 3. In *UTM Lab Meeting April*, volume 20, pages 4–5, 2010. 7.1

[168] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86, 2003. 3, 7.4.3

[169] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 1–16, New York, NY, USA, 2021. Association for Computing Machinery. 7.2

[170] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007. 7.2

[171] Samuel Jero, Md Endadul Hoque, David R Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in tcp congestion control using a model-guided approach. In *NDSS*, 2018. 7.2

[172] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

7.2

[173] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. {Model-Agnostic} and efficient exploration of numerical state space of {Real-World}{TCP} congestion control implementations. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 719–734, 2019. 7.2

[174] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 213–218, 2013. 7.2

[175] Hajime Tazaki, Frédéric Urbani, and Thierry Turletti. Dce cradle: Simulate network protocols with real stacks. In *Workshop on NS3 (WNS3)*, 2013. 7.3.6

[176] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7(1):33–47, 1999. 7.4

[177] BBR-Development. Question on strange bbr behavior. `https://groups.google.com/g/bbr-dev/c/XUOKHJiAW80`, 2022. [Online; accessed 23-June-2022]. 7.4.1

[178] Richelle Adams. Active queue management: A survey. *IEEE communications surveys & tutorials*, 15(3):1425–1476, 2012. 8.2

[179] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019. 8.2

[180] Dandan Ding, Zhan Ma, Di Chen, Qingshuang Chen, Zoe Liu, and Fengqing Zhu. Advances in video compression system using deep neural network: A review and case studies. *Proceedings of the IEEE*, 109(9):1494–1520, 2021. 8.2