Bridging the Gap Between Real-time Video and Backlogged Traffic Congestion Control

by

Pantea Karimi

B.Sc., Sharif University of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Pantea Karimi
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Mohammad Alizadeh
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Bridging the Gap Between Real-time Video and Backlogged Traffic Congestion Control

by

Pantea Karimi

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Real-time video applications, such as video conferencing, have become essential to our daily lives, and ensuring reliable and high-quality video delivery in the face of network fluctuation and resource constraints is critical. However, video congestion control algorithms have been criticized for their sub-optimal performance in managing network congestion and maintaining satisfactory video quality and latency. At the same time, state-of-the-art congestion control algorithms have demonstrated remarkable performance improvements, effectively addressing network congestion challenges and enhancing the overall quality of data transmission. In this work, we first demonstrate why there is such a gap between the performance of congestion control schemes on backlogged flows compared to real-time video streams. Second, we present Dumbo, a design for reshaping the video traffic to look like backlogged traffic, thus enabling state-of-the-art delay-sensitive congestion control algorithms for real-time video. We implemented Dumbo atop WebRTC and evaluated it on emulated network conditions using real-world cellular network traces. Our results show that Dumbo in comparison with GCC achieves a 1.5 dB improvement in PSNR, 1.6 dB improvement in SSIM, 100 ms lower frame latency, 35x faster convergence time, 16% increase in the video bitrate, 32% increase in network utilization, and 4x reduction in the network queueing delay.

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor, Massachusetts Institute of Technology

Acknowledgments

I want to express my heartfelt gratitude to my advisor and mentor, Mohammad Alizadeh, for his unwavering guidance, invaluable expertise, and continuous support throughout my research journey. His mentorship and encouragement have shaped my academic growth and fostered my passion for knowledge. I am profoundly in awe of his dedication, patience, enthusiasm, and calm but firm manner toward research.

I am truly fortunate to have had the guidance and mentorship of Vibhaalakshmi Sivaraman and Sadjad Fouladi. Vibhaa has diligently guided me through the initial two years of my Ph.D., demonstrating remarkable patience and excellent mentorship. I cannot express enough gratitude to Sadjad for his tremendous assistance, guidance, and support, providing invaluable advice and insights into my career.

I want to appreciate the NMS group at MIT, particularly expressing my thanks to Lei, Venkat, Prateesh, and Mehrdad, along with my profound gratitude to Sudarsanan Rajasekaran for his immense support and invaluable friendship. I am also grateful to my colleague and friend Frank Wang for his encouragement and advice.

I would like to extend my gratitude to my other friends, namely Itai, Rachel, Edward, Lily, Nicholas, Mohamed, and James, for making my life's journey more enjoyable and meaningful. Additionally, I want to express my thanks to my invaluable friend Sarah Gurev and her beautiful family, Elizabeth, Michael, Grandma Joan, Jack, JoJo, and Pika, who have embraced me as part of their own family.

Finally, I would like to seize this opportunity to express my most profound appreciation to my beloved family. Words fail to adequately convey my immense gratitude for all the sacrifices they have made, enduring the pain of separation for the sake of my education. From the bottom of my heart, I thank them for being the pillars of love, understanding, and support. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time video applications have become integral to our lives, from cloud video gaming to video conferencing for remote work and online education. In recent years, the use of real-time video systems has seen a noticeable increase across various domains. This upward trend can be attributed to several factors. Firstly, technological advancements and the widespread availability of high-speed internet connections have significantly improved the quality and reliability of real-time video systems, making them more accessible and user-friendly [29]. Secondly, the rise of social media platforms and the increasing demand for live-streaming content have contributed to the surge in real-time video usage [47]. Platforms like Facebook Live, Instagram Live, and YouTube Live have become popular avenues for individuals and organizations to reach and interact with audiences across the world.

Additionally, integrating real-time video systems in various industries such as healthcare, education, and entertainment has fueled their increased adoption [13]. For instance, telemedicine services utilize real-time video systems to enable remote doctor-patient consultations, while educational institutions employ them for virtual classrooms and online learning experiences. The growing use of real-time video systems highlights their versatility and impact in facilitating instant, interactive communication and content sharing.

Several studies and user feedback indicate that people are often unsatisfied with the quality of real-time video systems. Research conducted by Smith and Anderson

(2020) found that a significant number of video conferencing users expressed frustration over issues such as poor audio and video quality, frequent lags, and dropped connections [45]. Furthermore, a survey conducted by Tech Republic revealed that 65% of respondents experienced difficulties with video quality during real-time video applications, leading to reduced engagement and productivity [48]. These challenges can be attributed to various factors, including inconsistent network connectivity, limited internet bandwidth, and hardware limitations. As the demand for real-time video systems continues to grow, providers must address these quality concerns and invest in improving the technical infrastructure to enhance user experience. In this work, we focus on real-time video systems, which are essentially different from other video systems, such as video streaming. Real-time video systems - examples include cloud video gaming, video conferencing, teleoperation, and telemedicine - have a two-way interaction between the users and are highly sensitive to latency.

One primary reason real-time video systems may experience low quality is the lack of effective video congestion control mechanisms. Video congestion control refers to the ability of a system to adapt video transmission parameters in response to varying network conditions and congestion levels. Inadequate or inefficient video congestion control can lead to network congestion, packet loss, and increased latency, ultimately resulting in degraded video quality. Research by Zhai et al. (2020) highlights the importance of adaptive congestion control mechanisms to maintain optimal video quality during real-time video communication [56]. Furthermore, Li et al. (2018) emphasize the need for intelligent congestion control algorithms that dynamically adjust video transmission rates based on network conditions to ensure smooth and low-latency real-time video [32].

Research has shown that many existing video congestion control algorithms are often insufficient in handling network congestion and maintaining satisfactory video quality. A study conducted by Sarker et al. (2019) evaluated several popular video congestion control algorithms and found that they often struggled to adapt to varying network conditions, leading to frequent video freezes, buffering, and degraded user experience [41]. These findings highlight the need for more robust and adaptive video congestion control algorithms to respond dynamically to network conditions and optimize video transmission parameters accordingly while keeping the latency low. Further research and development in this area are crucial to address existing algorithms' insufficiencies and improve the quality of real-time video systems.

On the other hand, state-of-the-art congestion control algorithms have significantly improved data transmission performance in recent year. Delay-sensitive protocols such as BBR [9] (Bottleneck Bandwidth and Round-trip propagation time), PCC [15], and Copa [7] dynamically adjusts the sending rate based on the end-to-end signals like round-trip delay and receive rate, leading to reduced bufferbloat, faster recovery from congestion, and higher throughput [9]. However, these congestion control algorithms are typically evaluated on *backlogged flows.* A backlogged flow is a data flow that always has a packet to send upon request from the congestion control layer. Backlogged flows are a convenient traffic source for congestion control, since the congestion control algorithm can determine precisely when to send or not send a packet. State-of-the-art congestion control algorithms applied to backlogged flows can achieve excellent network utilization while controlling delays, far exceeding the performance of existing congestion controllers designed for real-time video.

In this work, we strived to find out why there is such a gap between the performance of backlogged congestion control compared to video congestion control. There are multiple challenges to sending a real-time video that makes video congestion control algorithms different from conventional congestion control algorithms. First, the video stream is not a backlogged flow of data to send. Video frames are generated periodically and behave as a train of packet bursts with periods of inactivity. Second, the video packets have stringent latency requirements; if they arrive too late, it introduces a high latency in the video, which directly affects the interactivity of the system. Third, in real-time video applications, unlike on-demand video streaming, there is minimal buffering at the video player. Ideally, each frame must be delivered as soon as it arrives. Furthermore, since the video itself is generated in real time, there is pre-encoded version of the frames available. Thus, the size of the frame after encoding is not known beforehand. The implication is that a mismatch between the video bitrate (produced after encoding) and the network rate determined by congestion control can occur rather easily. Specifically, the encoder may produce more packets than congestion control wishes to send, or it may produce fewer packets than congestion control is ready to send. Both situations lead to sub-optimal performance compared to operating congestion control with backlogged traffic.

To close the gap between the video congestion control algorithms and the backlogged congestion control algorithms, we propose Dumbo, a simple framework for making a video traffic source appear like a backlogged flow to congestion control. Dumbo enables state-of-the-art window-based delay-sensitive backlogged congestion control algorithm to be used for real-time video traffic without modification, and it achieves nearly identical performance to using the algorithm for a backlogged flow. In Dumbo, the congestion control algorithm always sees a backlog of data to send. If there are unsent video packets, it first sends those. But if the congestion controller senses available bandwidth but there are not video packets, Dumbo generates "dummy" packets and provides them for transmission. The key observation here is that the dummy traffic allows the congestion control to maintain its ACK-clocked feedback loop, and thus helps it maintain an updated video of the network state. As bandwidth opens up, Dumbo increases the target bitrate of the video encoder, improving video quality. But in the interim before the video bitrate increases, it uses dummy traffic to enable efficient congestion control.

We implemented Dumbo atop WebRTC [51] and evaluated it on emulated network conditions using real-world cellular network traces. We applied Dumbo to two delay-sensitive backlogged congestion control algorithms: Copa [7], and RoCC [1]. Our results show that, on average, Dumbo in comparison with GCC [10], WebRTC's default video congestion control achieves achieves a 1.5 dB improvement in PSNR, 1.6 dB improvement in SSIM, 100 ms lower frame latency, 35x faster convergence time, 16% increase in the video bitrate, 32% increase in network utilization, and 4x reduction in the network queueing delay.

The rest of this thesis is organized into these chapters: In the Related Work chapter, we discuss the works done in the backlogged and video congestion control and the video platforms that use the video congestion control algorithms. In the Motivation chapter, we study the behavior of a video congestion control algorithm and break down the problem we are trying to solve. In the Design chapter, we propose our solution to the problems we found in Motivation. In the Implementation chapter, we describe the implementation details of our Design ideas and the technical details of our proposal. Lastly, in the Evaluation chapter, we compare the end-to-end performance of the current video congestion control algorithm with our proposal.

Chapter 2

Related Work

2.1 Congestion Control

Congestion control algorithms operate as the backbone for adjusting the video rate in video applications and are designed to regulate the flow of data in order to prevent network congestion. The video congestion control algorithms aim to reduce the delay in delivering the packets and utilize the bandwidth as much as possible.

Most end-to-end congestion control approaches can be broadly categorized into delay-based [8, 7, 31, 34, 28, 54, 10, 9] or buffer-filling schemes [15, 49].

2.1.1 Delay-based Congestion Control

Real-time applications, such as video applications, are susceptible to delay. Traditional loss-based congestion control algorithms, such as TCP [27], are unsuitable for real-time video applications since they probe the network by inducing long queues and draining them. These queue oscillations cause a time-varying delay component that adds to the propagation time and make delay-sensitive communications problematic. Thus, using some sort of delay-based congestion control algorithm is preferable. There are several delay-based congestion control algorithms to infer congestion. We describe some of them in this section.

Delay-based protocols are optimized to minimize queueing while achieving a target rate. They either use the delay through the queue [43, 28, 8], or delay-gradients [10, 54, 34] to infer how to adjust the sending rate. Some algorithms like Copa [7] and BBR [9] explicitly drain the queue in order to accurately estimate minimum round-trip time (RTT), which in turn helps get a more accurate estimate of the link rate. A few of these algorithms, such as Timely [34] and DX [31], have also been deployed in datacenter settings.

Congestion Control Algorithms using round-trip-time (RTT):

These algorithms aim to prevent congestion by adjusting the transmission rate based on the observed round-trip time of the network. RTT is the time it takes for a packet to travel from the sender to the receiver and back again. These algorithms work by continuously monitoring the RTT of the network and dynamically adjusting the transmission rate based on this value.

One standard RTT-based congestion control algorithm is TCP Vegas [8], which uses a congestion avoidance mechanism that adjusts the transmission rate based on the observed RTT of the network. TCP Vegas uses a feedback mechanism that compares the expected RTT with the actual RTT and adjusts the transmission rate accordingly.

Another commonly used RTT-based congestion control algorithm is TCP Reno [37], which uses a combination of slow start and congestion avoidance mechanisms to adjust the transmission rate based on the observed RTT of the network. TCP Reno gradually increases the transmission rate until congestion is detected, at which point it reduces the transmission rate and gradually increases it again until congestion is detected once more.

TCP Fast algorithm [28] is based on reducing the time required to open a new TCP connection, which is a critical factor in reducing latency and improving overall network performance. The TCP Fast algorithm achieves this by increasing the initial congestion window (ICW) size, which determines the amount of data a sender can send before receiving an acknowledgment from the receiver. By increasing the ICW size, TCP Fast allows more data to be sent at the start of a connection, reducing the time required to establish the connection and improving the overall performance of

TCP. The main issues with these algorithms are low channel utilization in the presence of reverse traffic or when competing with loss-based flows [21] (video conferencing has reverse traffic).

Congestion Control Algorithms using one-way delay (OWD):

These algorithms estimate network congestion by using the delay between the sender and receiver. OWD-based algorithms measure the time it takes for a packet to travel from the sender to the receiver but do not wait for a response from the receiver. Instead, they estimate the OWD by measuring the difference between the time the packet was sent and the time the receiver reported receiving it. OWD-based algorithms assume that the one-way delay is asymmetrical, meaning that the time it takes for a packet to travel from the sender to the receiver may differ from the time it takes for the response to travel from the receiver to the sender. OWD-based algorithms adjust the transmission rate based on the estimated network congestion level. Examples of these algorithms are LEDBAT (over UDP) [43], and TCP-LP [30]. These algorithms are shown to suffer from the "latecomer effect": when two flows share the same bottleneck, the second flow typically starves the first one [12].

Congestion Control Algorithms using delay gradient:

These algorithms use the gradient of the delay between consecutive packets that are sent from the sender to the receiver. In other words, they look at inter-packet arrival times at the sender and the receiver. They were employed to overcome the aforementioned "latecomer effect." Some examples are CDG [23], Verus [55], and Google Congestion Control (GCC) [11]. Recently, SQP [38] has also been introduced for interactive video streaming applications that need to stream high-bitrate compressed video with very low end-to-end frame delay (e.g., AR streaming, cloud gaming). The video range that SQP performs in is much more than the conventional real-time video applications (e.g., video conferencing) we have in mind.

2.1.2 Buffer-filling Congestion Control

In contrast, buffer-filling algorithms [15, 22, 17] aim to send as much traffic as possible until loss or congestion is detected. At that point, these algorithms signal to the sender to reduce its sending rate and then back off at that point. Such algorithms typically do not compete fairly with delay-based algorithms that tend to reduce sending rate at the slightest increase in queueing delays or RTTs. Active queue management techniques such as RED [40], PI [24] and DCTCP [14], and approaches [20] that can switch between delay-based and buffer-filling modes attempt to ameliorate some of these issues. However, limited attention has been paid to buffer-filling and delay-based algorithms for application-limited flows [6, 5] because bulk-flow settings where traffic is generated will allow researchers to reason through steady state behavior easily. However, video traffic is uniquely application-limited in that frames are generated at a fixed framerate (typically 30 fps) and, thus, produces data in an on-off pattern only every 30 ms. This non-backlogged traffic limits the amount of feedback the sender receives and, consequently, its ability to quickly increase and decrease its sending rate. Dumbo combats this problem by making video traffic appear like a long-running flow and is designed to work in conjunction with any delay-based algorithm.

2.2 Video Systems

Many video applications, including Google Meet, Meta Messenger, Discord, and Amazon Chime [4] use Web Real-time Communication (WebRTC) [51] to deliver real-time, high-quality video and audio content to users. WebRTC is a free, open-source project that provides a collection of standard APIs and protocols for building real-time communication applications directly in web browsers. WebRTC handles connection establishment, stream management, end-to-end encryption, and rate control for video applications. Additionally, WebRTC uses peer-to-peer connections, which can help reduce network latency and improve video quality, especially in scenarios where traditional server-based approaches can result in slower connections or higher costs. WebRTC provides simple JavaScript APIs which enable application developers to easily span the connections between two browsers with a few lines of code. WebRTC is built on top of standard web technologies, including HTML, JavaScript, and the Session Initiation Protocol (SIP), and can be integrated with various other communication technologies, such as voice over IP (VoIP) and instant messaging. WebRTC is also designed with security in mind, using end-to-end encryption and other security features to protect user privacy and prevent unauthorized access. As a result, WebRTC has become a popular choice for building video and audio applications across a wide range of industries, including healthcare, education, and entertainment. With the growing popularity of WebRTC, it is expected that more and more video applications will adopt this technology in the future. As a result, the need for a reliable and efficient video congestion control algorithm is becoming more pronounced.

Google Congestion Control (GCC) [11] is the rate control mechanism within WebRTC. It uses the variations in the inter-packet arrival time of the packets at the receiver to signal to the sender whether to increase, hold, or decrease the rates. WebRTC also uses Transport Wide Congestion Control [25] at the sender to estimate bitrate based on intra-packet delays. However, in the absence of continuous feedback due to the on-off patterns of video traffic, GCC is sluggish in increasing and decreasing its rates. This issue, in turn, affects the bitrate of the video delivered; when there is unused bandwidth, it renders poor-quality video, and when it overestimates the bandwidth, it leads to stalls and frames with glitches. This control loop is worsened by the fact that feedback depends on the actual data sent, which in turn is controlled by the encoder, which is both slow to respond and is operating on faulty estimates in the first place. Dumbo instead decouples the congestion control loop from the encoder's slow responses by padding the encoder's output with additional dummy data to match the desired rate or window at any given moment. This technique provides faster and more accurate feedback, allowing the video application to respond more quickly to bandwidth variations. A recent proposal called SQP [38] achieves low end-to-end frame delay for interactive video streaming applications that stream very high bitrates (*e.g.,* AR streaming, cloud gaming). However, SQP operates in much higher bitrate regimes than what Dumbo is designed for.

Adaptive bitrate algorithms [33, 26, 53, 46, 52] are another category of video rate control solutions. These algorithms run at the application layer and use information about available bandwidth, buffer size, and current bitrate to determine the next bitrate that the video should be encoded at. While these algorithms help influence the encoder's compression, the output produced may not match the network's capacity due to hysteresis from its past outputs. Salsify [19] combats this by producing two options for the next encoded frame and using the one that matches the instantaneous capacity better. Dumbo goes a step further in decoupling the transport and the encoder by allowing the transport to pad additional data if the encoder fails to match the instantaneous capacity and to slow down the encoder's output if it overshoots the available capacity.

Chapter 3

Motivation

3.1 Status Quo for Video Rate Control

To understand how the rate control for real-time video works today, we run Google Congestion Control (GCC) [10], the rate control mechanism inside WebRTC, on a time-varying periodic link. The link starts with 3 Mbps of bandwidth for 40 seconds, then drops to 500 Kbps for the next 40 seconds before jumping back up to 3 Mbps again. The minimum network one-way delay is 25 ms with a minimum round-trip time (RTT) of 50 ms, and the buffer size at the bottleneck is large enough that there are no packet drops. We run the experiment for 160 seconds. To illustrate network measurements, we use the link's utilization or throughput, which is the rate at the egress of the link, and the network delay, which is the per-packet queuing delay in the network. To quantify the video quality, we use per-frame PSNR (Peak Signal-to-Noise Ratio) [44] and SSIM [50] (structural similarity index measure), which is a method for predicting the perceived quality by humans of digital cinematic pictures. We also measure end-to-end frame latency, indicating how laggy the received video is at the receiving point.

We show GCC's network utilization in Fig. 3-1 and network delay in Fig. 3-2. The resulting end-to-end video quality in Fig. 3-4 and Fig. 3-5, and its end-to-end frame latency in Fig. 3-6. We observe that GCC is very slow to increase the rate when the link starts, and its capacity is restored to 3 Mbps in the 80s. Specifically, GCC takes

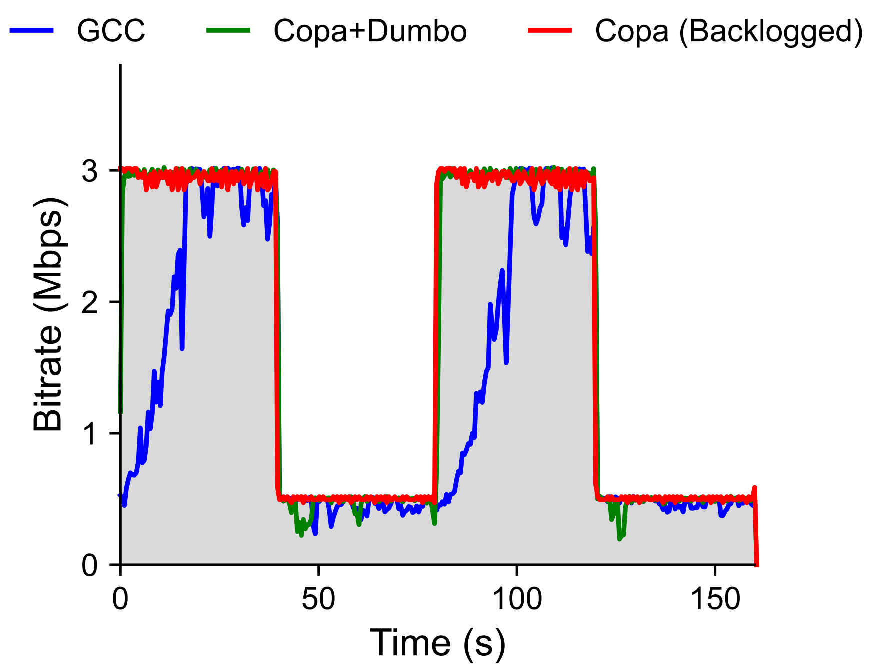

Figure 3-1: Link's utilization time series.

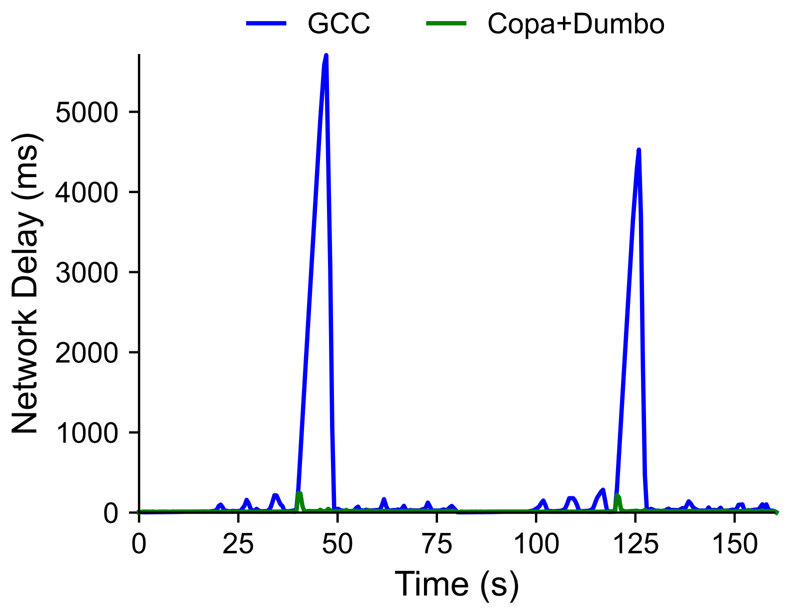

Figure 3-2: In-network queuing delay time series.

Figure 3-3: Network measurements of Copa on a backlogged flow, GCC on a video flow, and Copa + Dumbo on a video flow. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps with a minimum of RTT of 50 ms.

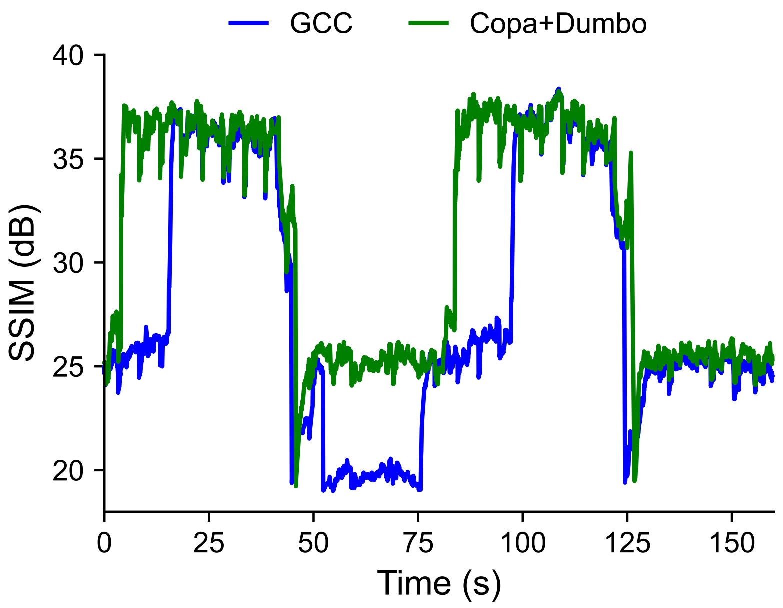

Figure 3-4: Frame SSIM for the received video time series.

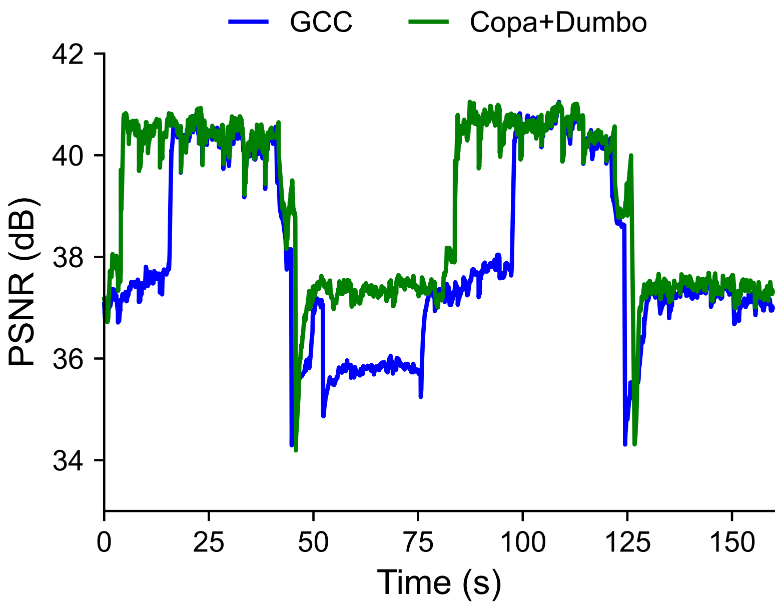

Figure 3-5: Frame PSNR for the received video time series.

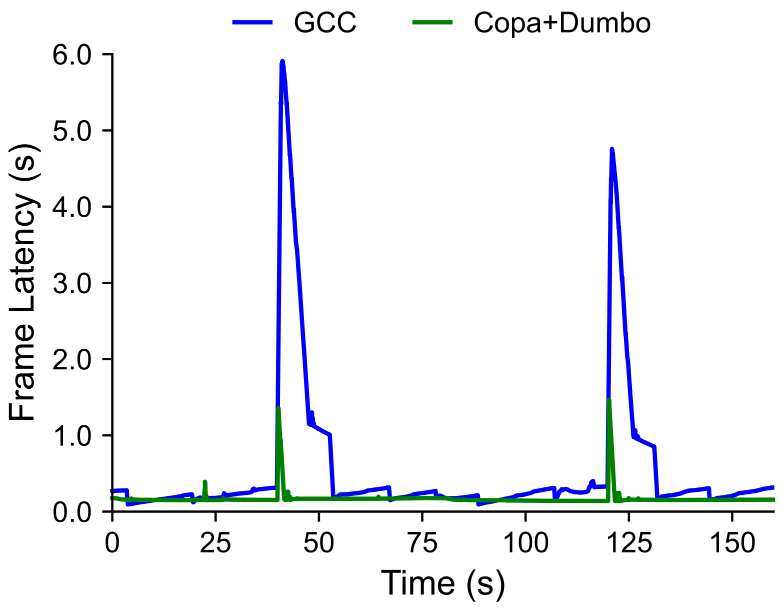

Figure 3-6: Frame Latency time series.

Figure 3-7: End-to-end frame measurements of Copa on a backlogged flow, GCC on a video flow, and Copa + Dumbo on a video flow. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps with a minimum of RTT of 50 ms.

20 seconds to go from 500 Kbps to 3 Mbps.

This sluggish response manifests itself as lower and slowly increasing visual quality (Fig. 3-4 and Fig. 3-5). Further, even once in a steady state, GCC under-utilizes the link, achieving 90% of the link. We also observe that GCC is slow to react when the link rate drops suddenly to 500 Kbps at the 40s. This is best observed in Fig. 3-6, where the latency through GCC's queue spikes to a few seconds at that very moment. This is because GCC continues to send at a higher rate than the network can support, causing queue buildup and added delay. The queue buildup can be observed easily in Fig. 3-2, where GCC cannot back off its rate fast enough and causes a massive queue in the network. This spike takes 15s seconds to settle down. If such a queue buildup exceeds the packet buffer, it can cause drops and stalled video due to decoding errors, which often take a few seconds to recover from at the application level. In contrast to GCC's lengthy response to changes in network capacity, traditional congestion control algorithms [7, 1, 8, 9] operating on backlogged flows, respond much faster, typically on the order of few RTTs or hundreds of milliseconds as opposed to 10s of seconds that it takes for GCC. For instance, the "Backlogged Copa" line in Fig. 3-1 shows that when we run Copa [7], a recent delay-based congestion controller with a backlogged flow on the same time-varying link, its behavior reflects a much faster response. Specifically, Copa matches the link rate of 3 Mbps within a few round-trip-times (milliseconds), after which its steady-state utilization stays at 3 Mbps. Even when the link rate drops to 500 Kbps, Copa reacts almost immediately, keeping the delay through the queue relatively small. This issue manifests as a small blip in Fig. 3-6 for "Copa" around the 20s that drops down in less than 2s. Copa catches up within a second when the link rate returns to 3 Mbps. This wide disparity between GCC on real-time video traffic and Copa on backlogged traffic begs the question: Why does the state-of-the-art for video rate control lag so far behind the state-of-the-art for congestion control?

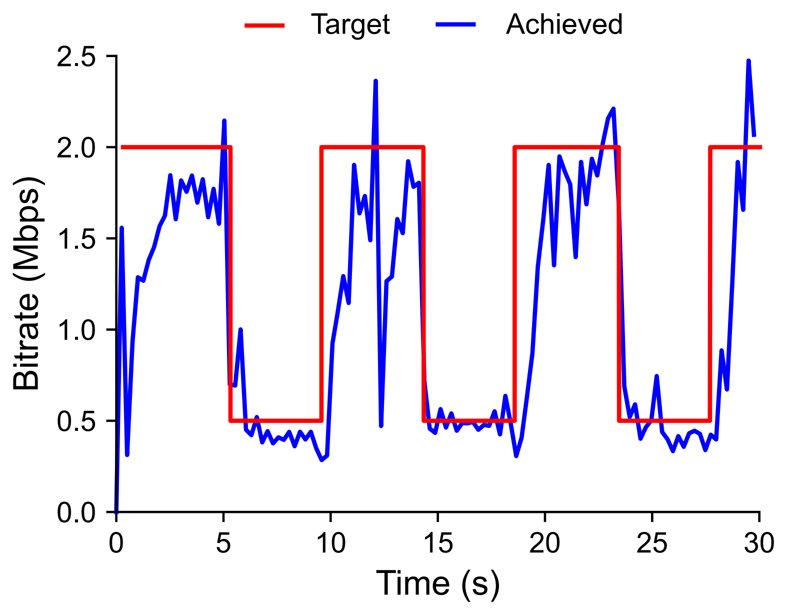

Figure 3-8: Encoder's reaction time: the Target shows the input rate given directly to the encoder, and the Achieved rate is the encoder's output rate. The experiment has been done on WebRTC's default video encoder (VP8).

3.2 Encoder-driven Rate Control Loop

One natural response to the disparity between GCC and Copa in Fig. 3-1 and Fig. 3-6 would be to claim that GCC is simply a lousy rate control algorithm. However, we believe otherwise. GCC has been carefully designed with the tight latency bounds of interactive video applications in mind. Its rate control responds to increases or decreases in delay gradients over RTT timescales. It is also relatively conservative in link utilization to not overwhelm the network and invariably cause packet drops and glitchy frames. The real issue that limits GCC is that, in video applications, *the rate at which data is transmitted on the wire is dictated by the rate that the encoder outputs*. Unfortunately, the encoder is extremely unreliable: it rarely matches the target bitrate exactly [19] and cannot adapt to target bitrate changes immediately. We illustrate this behavior in Fig. 3-8 where we supply a target bitrate that switches between 2 Mbps in periods of 5 seconds to the encoder and observes its achieved bitrate in response to it. Every time the bitrate goes up from 500 Kbps to 2 Mbps, the encoder takes nearly 3 seconds to catch up. This lag is not surprising: video encoders use delta encoding, which encodes the differences between adjacent frames, and it is difficult to abruptly change the quality or bitrate when frames are so closely intertwined. However, the effect of this encoder lag is that GCC, which uses the delay observed for the video packets sent on the wire to infer bandwidth availability, is very slow in its rate increase. Whenever GCC senses that bandwidth is opening up, it slightly increases the rate supplied to the encoder, which response over many seconds. When the feedback from this increased bitrate data trickles in, GCC once again sets the rate a little higher for the encoder, which takes many more seconds to match it, and this process continues. Unsurprisingly, this cycle ends up taking 15–20 seconds end-to-end. Moreover, since the encoder cannot immediately reduce the bitrate (recall Fig. 3-6), GCC has to be conservative and not only increase the bitrate carefully but also leave some bandwidth headroom in a steady state so that it reduces the risk of being unable to react to delay increases quickly. Essentially, GCC's bandwidth discovery and steady-state behavior are limited by the encoder output. This is what we call "encoder-driven rate control".

It is worth noting that GCC tries to mitigate some of the issues arising from variations in the encoder output with a "pacer queue" [10]. The pacer queue operates packet-by-packet and paces packets according to a small multiple of GCC's target rate (e.g., $pacing\_rate = 1.5 \times target\_rate$). While pacing helps smoothen the occasional spike in sending rate (*e.g.,* due to a keyframe), it is not helpful in the context of rate control decisions that cover many RTTs, which are instead dictated by the encoder's output rate over several frames.

3.3 Decoupling the Encoder from Rate Control

The above discussion suggests that there is not much one can do to change the encoder itself, and that relying on the slow encoder for rate control is very ineffective. In other words, we need to decouple the congestion control loop from the encoder's output on small timescales. The goal of this decoupling is to give complete control to the congestion controller to directly set the data rate on the wire. To achieve this, we need two mechanisms: (1) when the encoder overshoots the available capacity, we need to limit what is sent on the wire avoiding sharp increases in the video delay, and (2) when the encoder undershoots the available capacity, we need additional data on the wire to compensate for the encoder's slow response.

Our design, Dumbo, includes mechanisms for both of these scenarios. To account for encoder overshooting, Dumbo repurposes the "pacer queue" in GCC to match the actual rate signaled by the congestion controller. We also bound how big this queue can grow before we pause encoding of further frames. The intuition here is that there is no point encoding a frame that cannot be sent with low delay over the network. Regardless of whether we send the packets of such a frame or hold them in the pacer queue, the end-to-end delay of the frame will be high, and more importantly, the congestion created will increase delay for subsequent frames. Instead, Dumbo simply skips a frame in such situations.

When the encoder undershoots the available capacity, Dumbo sends "dummy packets" to match the rate requested by the congestion control mechanism. We simultaneously signal the latest target rate from the congestion controller to the encoder. This allows the congestion controller to independently discover bandwidth fast, while also allowing the encoder to respond at its own pace to the latest bandwidth estimate. This dummy traffic could also be repurposed for useful information such as forward error correction (FEC) packets [39, 35] or keyframes for faster recovery from loss. However, we leave such enhancements to future work and focus solely on the impact of dummy traffic on video congestion control.

Dumbo effectively makes video traffic look like a backlogged flow: either the encoder produces data continuously, or the encoder's output is padded to produce traffic at the congestion controller's will. As a result, a congestion controller operating with Dumbo has full freedom to adjust the sending rate as it wishes and is not subject to the vagaries of the video encoder. The net impact of adding these mechanisms is visible in Dumbo's lines in Fig. 3-1, 3-5, and 3-6. Dumbo's response is very similar to that of Copa on backlogged traffic. Specifically, it discovers bandwidth a lot faster than GCC when it opens up. Unlike GCC, Dumbo does not face the brunt of the

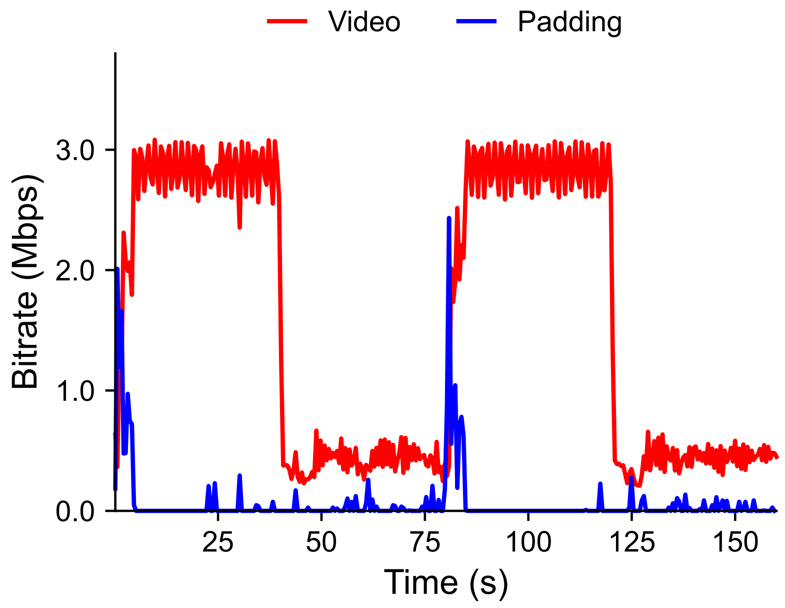

Figure 3-9: Breakdown of the link's utilization: Video rate vs Padding (Dummy) rate for Copa with Dumbo. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps with a minimum RTT of 50 ms.

encoder's sluggishness over many cycles of small rate increases followed by encoder responses.

Instead, the dummy traffic allows this bandwidth discovery to happen independently of the encoder. Fig. 3-5 shows that this dummy traffic does not eat away at useful video traffic; it translates to higher PSNR and SSIM for received frames than GCC. Even when the bandwidth drops, Dumbo keeps the queueing delay slightly like Copa on backlogged traffic. Lastly, Dumbo's steady state link utilization is also very high (almost 100%).

Fig. 3-9 shows the breakdown of the link's utilization of video traffic and the dummy traffic. Note that the sum of the video rate and the padding (dummy) rate equals the "Copa+Dumbo" line in Fig. 3-1. As you can see, the dummy traffic dummy is only high during transients of the bandwidth where the video encoder cannot reach the rate as fast as the congestion controller decides and when the output bitrate of the encoder is transiently lower than the link. The average video bitrate for this experiment is 1573 Kbps, and the average padding bitrate is only 61 Kbps. This means that by only sending 3.8% dummy traffic, we will have both lower end-to-end frame latency (Fig. 3-6) and higher visual quality (Fig. 3-4).

What about probing mechanisms?

A natural question at this point would be if probing mechanisms, specifically those already supported within GCC [2], would achieve the same net effect as what we describe above. While periodic bandwidth probing has been shown to be very effective for certain CCAs [9], we observe that the mechanism within GCC is relatively ad-hoc. It uses a periodic timer that fires every few seconds (this parameter can be adjusted) and sends a little more traffic than the current sending rate in response. Such an infrequent timer does not help on the finer RTT-level timescales required for precise rate control. A timer is, in practice, very similar to a sluggish encoder that responds to the target bitrate over a few seconds. In contrast, we view our approach as a more systematic way of introducing the right amount of "probing" or dummy traffic when necessary such that it does not overwhelm the network while still providing useful feedback for bandwidth discovery.

Fig. 3-10 shows the breakdown of the link's utilization to video and dummy traffic. The timing of the probings is off, and their periodic nature is independent of the underlying changing bandwidth. The probing fires slightly around the 50s (10s later than the link has changed) and the 130s.

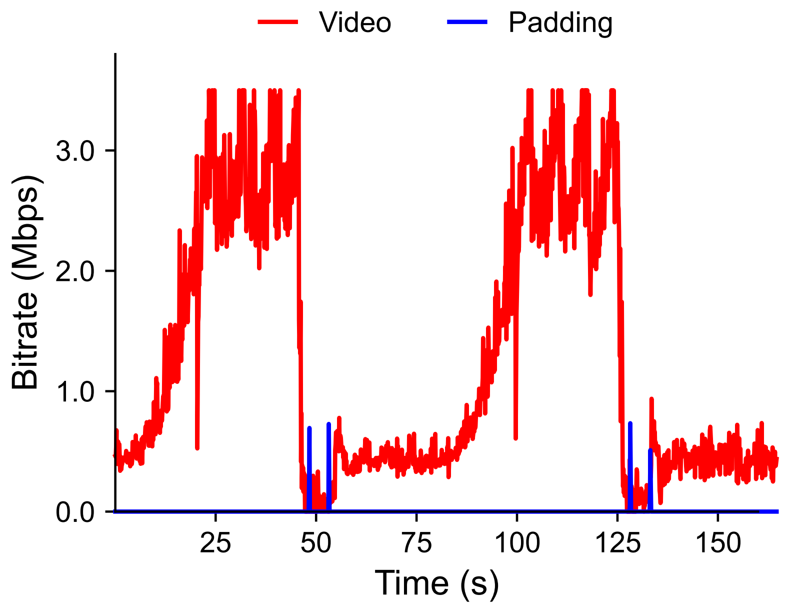

Figure 3-10: Breakdown of the link's utilization: Video rate vs Padding (Dummy) rate for GCC with probing enabled. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps with a minimum RTT of 50 ms.

Chapter 4

Design

The issue with real-time video applications is that the produced traffic is not a backlogged flow. Our idea is to change the shape of the produced traffic such that from the congestion control algorithm's perspective, it looks like a backlogged flow. This allows us to plug in any state-of-the-art congestion control algorithmdesigned to work with a backlogged flow. Fig. 4-1, shows the overall structure of Dumbo.

Congestion Controller. This module is a window-based congestion control algorithmthat is responsible for finding the *cwnd* (number of allowed bytes in flight) at each point in time. When bytes are sent to the receiver, the number of bytes in flight increases at the sender. These bytes could belong to either video or dummy packets and the congestion controller does not differentiate between them. When packets are received, an acknowledgmentis sent back to the sender and the congestion control algorithmuses the ACK to compute the RTT (round-trip time of the packets), and update the *cwnd* . The congestion controller also computes the sending rate for all the streams (video, audio) as the *cwnd* divided by smoothed RTT (using an exponentially weighted moving average of the RTT samples). The sending rate is used to set both the target bitrate of the encoder and to configure the Pacer.

Dumbo can be used with any delay-controlling congestion control algorithm. In our experiments, we use Copa [7] and RoCC [1], two recently proposed congestion controllers. These congestion controllers can be tuned to navigate the throughput and latency tradeoff according to the needs of the application. For example, a video-

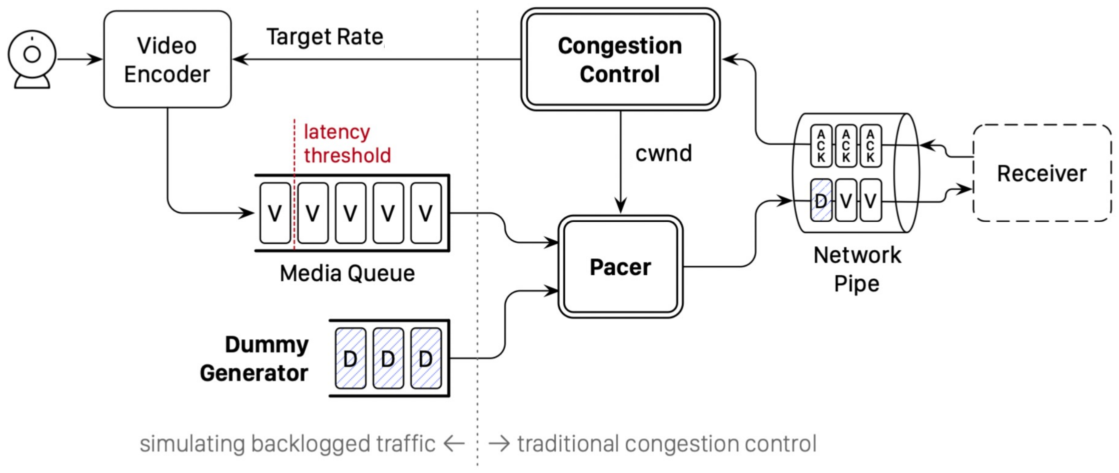

Figure 4-1: Dumbo's Design: Dumbo is trying to re-shape the video traffic to a backlogged flow for state-of-the-art window-based backlogged congestion control algorithms.

conferencing application would want a congestion controller that minimizes latency (potentially at the expense of throughput) while a live video stream may tolerate moderate increases in latency for better throughput and video quality. In our evaluation, we experiment with two CCAs: RoCC[1] and Copa [7]. RoCC has a parameter γ which adjusts how much network queuing delay RoCC causes. In Copa, the parameter δ is responsible for adjusting the throughput and delay tradeoff.

Video Encoder. The encoder receives the video from the application and strives to produce an average bitrate equal to its target bitrate. The encoded packets are put in a media queue available to the pacer. In real-time video encoding, the output size of the encoded frame is not known and the encoder tries its best to output the congestion control algorithm's estimate but it has fluctuations. In traditional video congestion control algorithms, if the encoder produces less data than what the congestion control algorithmis asking for, the sender has no choice but to send that much data and disrespect the congestion control algorithm. This means that congestion control algorithmwill not be able to receive enough acknowledgments it requires to ramp up its window as fast as the algorithm wants. We propose that in such scenarios, the transport fill the surplus with dummy packets to respect the congestion control algorithmrule. This is not a waste of bandwidth, but in fact, it allows the congestion control algorithmto ramp up as fast as it can to estimate the correct link's bandwidth. This correct higher estimate is fed into the encoder, enabling it to generate higher quality video in comparison to when the estimate was incorrect and lower.

Pacer. The Pacer receives *cwnd* and the target bitrate from the congestion controller. The instantaneous bitrate produced by the encoder may be larger than the target bitrate as discussed, so the pacer is responsible for controlling sudden bursts of packets. It does this by spacing packets according to the target rate (WebRTC already implements this pacing function). In addition to spacing packets, the Pacer also ensures that the *cwnd* bound on inflight data is enforced. If the number of unacknowledged bytes is less than *cwnd* and the target rate allows a packet to be sent, the pacer reads a packet from the media queue and sends it. The Pacer stops sending if the amount of data in flight exceeds *cwnd* .

Finally, the pacer also computes the amount of time it takes for the media queue to be drained by dividing the size of the media queue by the target bitrate from the congestion controller. If this time is bigger than a threshold, the pacer signals the encoder to stop encoding because there is no point in generating the data if the network is congested. We choose to stop the encoding instead of dropping the video packets in the media queue after encoding because if we drop an encoded frame, the encoder needs to reset its state and produce a keyframe which in a congested state could result in a cascading generation of large frames. This feature already exists in WebRTC and we incorporate it into our system.

Dummy Generator. To make the video flow mimic the behavior of a backlogged flow, we use a Dummy Generator to produce dummy packets if the media queue is empty and the congestion control algorithmis ready to send a packet. Specifically, if *cwnd* is greater than the amount of data in-flight, for the congestion control algorithmto have proper feedback from the state of the network we need to send a packet. Whenever this occurs and the media queue is non-empty, the pacer pulls packets from the media queue and sends them; but if the media queue is empty, the pacer pulls a packet from the Dummy Generator and sends it. In our implementation, the dummy generator produces empty packets, but for future work, dummy packets could be used to carry useful data like FEC packets.

Latency Control To prevent an unacceptable increase in latency, we have two mechanisms. First, the media queue has a latency threshold beyond which the video encoder is paused and skips frames. This feature already existed in WebRTC, but we chose a different threshold. We choose the latency threshold around 100-200 ms based on human's sensitivity to the lag time between two consecutive frames.

The second mechanism is adjusting the encoder's target rate based on the pacer's standing queue. The pacer's standing queue, denoted by $Standing\_PQ$, is computed as the minimum size of the aggregated media and dummy queue over the past T seconds. We choose T with the same logic as the latency threshold around 100-200 ms and we want that the standing pacer queue is drained before the receiver notices the increment in the latency. The congestion controller declares that in the time span of RTT, $cwnd$ bytes can be sent over the network. This could be translated to a target rate of $\frac{cwnd}{RTT}$ for the whole system, but this rate alone does not have proper control over the pacer queue size. With just declaring the sending rate as $\frac{cwnd}{RTT}$, the only knob controlling the pacer queue is the latency threshold which is risky because it starts skipping frames. Therefore, we incorporate a new term in the Target rate to show the effects of the standing pacer queue. Note that the congestion control determines the total bitrate for all the streams of the system, including video, audio, and control stream. Let's denote the sum of the audio and the control streams as Non_Video_Rate which is typically a fixed small bitrate. Target rate for the video encoder is determined as:

$$Target\_Rate = \frac{cwnd}{RTT} - \frac{Standing\_PQ}{T} - Non\_Video\_Rate$$

Chapter 5

Implementation

To realize our design, we've implemented our system on top of Google's WebRTC [51]. This work has taken several months with careful studying of Google's implementation of WebRTC.

5.1 WebRTC's Structure

Call. This component refers to the high-level process that establishes and manages real-time audio or video streams and communication sessions between two or more web browsers. The Call component is responsible for creating the sender and receiver peers.

Signaling. The signaling component facilitates communication between the peers to exchange metadata necessary for establishing the call. This can include information such as session descriptions, which describe the media streams, network configuration, and encryption settings, and ICE (Interactive Connectivity Establishment) [3] candidates, which are used for NAT traversal [18].

Media Streaming. Once the peer connection is established, the call component handles the real-time audio and video stream transmission between the browsers. This includes encoding the media streams into codecs supported by both parties, packetizing the media, and sending them over the network. The video encoder in WebRTC is responsible for converting raw video frames captured from the user's camera or a video file into a compressed format that can be transmitted over the network efficiently. Video compression is achieved using a variety of encoding techniques, such as transform coding, motion compensation, and entropy coding. The encoder uses algorithms to analyze the video frames and identify redundancies in the spatial and temporal domains, exploiting them to reduce the data needed to represent the video. This compression process results in a lower bitrate, allowing for faster transmission and reduced network congestion. WebRTC supports several video codecs, including VP8, VP9, and H.264, which are widely used for real-time communication. The choice of video codec in WebRTC is modular and depends on factors such as codec efficiency, hardware support, and interoperability among different platforms and browsers. The knob to change how much data the video encoder should generate is a target bitrate that is set by the congestion control.

RTP/RTCP. The data generated by the encoder is packetized and converted into RTP packets. Real-Time Transport Protocol (RTP) [42] is a widely used protocol for transmitting real-time multimedia data, such as audio and video, over IP networks. RTP provides mechanisms for the timely and reliable delivery of media streams, making it essential for applications that require low-latency and interactive communication, such as video conferencing, online gaming, and live streaming. RTP includes features such as sequence numbering, timestamping, and payload type identification, which help ensure that media streams are delivered in the correct order and synchronized at the receiver's end. Additionally, RTP can work in conjunction with other protocols, such as the Real-Time Control Protocol (RTCP) [42], which provides feedback on the quality of service and monitoring of RTP streams. During the call, the call component may also involve the use of the RTCP for monitoring the quality of service (QoS) and gathering feedback on metrics such as packet loss, latency, and jitter. This information can adapt the call parameters in real-time, such as adjusting the codec bitrate or changing the network transport settings, to optimize the call quality.

Transport. This module is responsible for sending and receiving all the packets for an endpoint, and it keeps track of the data in the flight, *inflight*, and the acknowledgments. Transport has two modules essential for video congestion control: a pacing controller and a congestion controller. Transport passes the acknowledgments to the congestion controller, and the congestion controller signals back its update to the Transport. The update can change the congestion window and the sending rate. Transport also divides the estimated available bandwidth for audio, video, and control streams and updates the target bitrate of the encoder. The update from the congestion controller also contains pacer configurations such as the *cwnd*, *inflight*, and the sending rate. Transport compares the amount of data in flight with the congestion window and sets the pacing controller to the "congested" state if there are more data in flight than the congestion window. The congestion control estimates the total available bandwidth. The update could also contain pacer configurations, and the Transport could change the new config to the pacing controller.

Congestion Controller This module estimates the bandwidth to send using the signals it receives from the Transport. The congestion controller determines the congestion window and the sending rate all RTP streams should generate. The congestion controller is flexible to implement both the window-based or rate-based algorithms based on what the underlying application wants.

Pacing Controller. The Pacing Controller aims to maintain a smooth and steady transmission rate and avoid bursts of packets. Since the video is generated periodically bursty, the pacing controller is responsible for pacing the packets in time to achieve the same average sending rate as the congestion control requires. If *cwnd* assigned by the congestion controller is smaller than *inflight*, the pacing controller stops sending media packets; otherwise, it schedules the packets such that the average rate of the sending packets is no more than the bitrate assigned by the congestion controller. The pacing controller allows occasional bursts of a few milliseconds (variable to tune) and waits for the queue it has built up in the network to drain. When the pacer is scheduled to send a packet, it prioritizes the media queue and pulls a packet from it if it is not empty. However, if the media queue is empty, the pacer calls the dummy generator module and gets a dummy packet to send.

Dummy Generator. WebRTC has a padding-generating module. We repurpose this module for creating padding and sending it as the dummy traffic. Each dummy packet is capped at a minimum of 200 bytes and *cwnd* and contains filler bytes. Dummy packets have *padding* type, and the receiver acknowledges them, but the data is not used.

All the packets reach the pacing controller to be sent. The congestion controller updates the pacing rate and its congestion state. If the pacing controller is in the "congested" state, the pacer does not send any packets but keepalive packets in some intervals. Otherwise, the pacer sends packets at scheduled times (*next-send-time*). We have modified the pacing controller to schedule the *next-send-time* to respect the pacing rate while occasionally allowing bursts of maximum duration (*burst-interval*). If a burst happens, the incoming *next-send-time* is adjusted to let the built-up queue be drained from the network. When the *next-send-time* has arrived, the pacer sends a packet from the pacer queue. If there is no packet in the pacer queue, the pacer generates a dummy packet and sends it to the network. The dummy packet is an RTP packet of Padding type. Note that stamping the sequence number on the packets happens after the pacing controller; thus, the acknowledgments will treat the dummy packets the same as any other application packets as desired.

Adaptive Resolution. WebRTC has a rather complex and sluggish mechanism for switching the sending resolution based on network conditions. This mechanism prevents the video bitrate from adapting as fast as possible to the network changes based on our fast window-based congestion control algorithms. This scheme is a limitation because we can not make the congestion control faster without the traffic's source being as fast. Thus, we disabled the native resolution adaptation of WebRTC and carefully implemented our resolution logic to fully follow the network's available bandwidth while preventing abrupt and fluctuating resolution changes. Our scheme computes a moving average of the congestion controller's target rate (*smoothed_rate*) over time when receiving the update from the congestion controller. Based on that, the sending resolution is decided. We update the ith sample of *smoothed_rate* when receiving the ith bitrate sample, b_i, at time t_i according to:

Switch Low Threshold	Switch High Threshold	Resolution
0 Kbps	400 Kbps	320×180
300 Kbps	1500 Kbps	640×360
1000 Kbps	3000 Kbps	1280×720
2500 Kbps	12 000 Kbps	1920×1080

Table 5.1: Mapping between encoder target bitrate and chosen resolution in Dumbo
.

$$smoothed\_rate_i = (1 - e^{-\frac{t_i - t_{i-1}}{\tau}})b_i + e^{-\frac{t_i - t_{i-1}}{\tau}} smoothed\_rate_{i-1}$$

Where τ shows how much memory our average has for keeping the samples from the past. This value could be changed based on the application, but we observed that a value of 500 ms is usually sufficient.

Table 5.1 shows the resolutions and their corresponding thresholds. The threshold bounds overlap to ensure smooth and stable transitions. That is why the switch low threshold of each resolution is smaller than the switch high of the previous resolution. If *smoothed_rate* is lower than the switch low threshold of the current resolution, the resolution goes down one step, and if it is higher than the switch high threshold of the current resolution, the resolution goes up one step.

5.2 Implementing the Congestion Control

The congestion control module is responsible for estimating the available bandwidth and feedbacking this estimation to the encoders for generating data with the correct bitrate. Congestion control in WebRTC provides an update with these fields: time of the update, measured round-trip-time, measured loss ratio, the congestion window, and minimum and maximum pacing rates.

In WebRTC, congestion control is updated from the statistics of the RTP streams from Transport, specifically the acknowledgments. Based on the algorithm, congestion control can also receive other information, such as the size of the pacer queue, by slightly modifying the Transport. Congestion control also has specific APIs required by other layers to implement. These APIs are periodic or event-based, and the congestion control needs to compute and send its update to the Transport. These events include sending a packet and, upon receiving an acknowledgment, where the size of the congestion window and the data in the flight can change.

For the congestion control to perform, it should know about the statistics of sent and received data and the acknowledgments from the receiver. WebRTC has already provided APIs from the transport module to provide feedback on this information to the congestion control as mentioned in Sec. 5.1.

Modifying the Application Congestion Control. WebRTC enables implementation of custom congestion control algorithmas long as it provides the necessary APIs. To modify the congestion control component, we should change the name of the used congestion control algorithmin the call, Transport, and API components of WebRTC and provide the code to factor the congestion control. We have implemented two congestion control algorithms on top of WebRTC: Copa [7] and RoCC [1]. These algorithms are window-based congestion control algorithms, and they control the amount of data sent by a congestion window.

5.2.1 RoCC

RoCC sets the congestion window to the number of received bytes in the past time interval of $(1+\gamma)min\_rtt$ plus a constant window size where min_rtt is the minimum round-trip time observed in a long period of time. The round-trip time and the number of received bytes are computed using the acknowledgments received from Transport. Since in WebRTC, acknowledgments are sent periodically; congestion control receives aggregated packet feedback. To compute min_rtt, we subtract the wait time at the receiver for the acknowledgment to be sent. RoCC computes the window at time t using:

$$cwnd_t = \#\ \textit{acked bytes in [t - } (1+\gamma)min\_rtt\ \textit{, t] + C} \tag{5.1}$$

Where C is a constant positive window size. The value of C helps RoCC to achieve eventual fairness. The round-trip time for each packet is not necessarily equal to *min_rtt* because of the possible queue built up in the network. RoCC measures *propagation_time* as the smoothed (EWMA) round-trip-time measured upon receiving each acknowledgment. Upon receiving an ith acknowledgment, RoCC measures the average round-trip time for those packets, denoted by rtt_i. Then, RoCC updates *propagation_time* as the following where α shows how sensitive we are to the latest updates:

$$propagation\_time_i = \alpha \cdot rtt_i + (1 - \alpha) \cdot propagation\_time_{i-1} \tag{5.2}$$

RoCC updates the sending rate for the total RTP streams as:

$$sending\_rate = cwnd/propagation\_time \tag{5.3}$$

This sending rate is notified to the Transport and eventually is allocated between the audio, video, and control streams. RoCC sets the pacing rate at *sending_rate*, and the pacer will try to drain the pacer queue at this rate.

The intuition behind this algorithm is that it tries to build a queueing delay of γmin_rtt in the network. γ is a variable that the application can set based on how delay sensitive the application is. Having some queue built up in the network ensures high link utilization while application-specific γ, RoCC controls the in-network delay it introduces.

In an algorithm variation, the application can set a desired queueing delay, denoted by *desired_qdelay*, and γ is updated as $\gamma = desired\_qdelay/min\_rtt$. Though this algorithm achieves less delay than the fixed *gamma*, it is less robust and more prone to oscillations due to late acknowledgments.

5.2.2 Copa

Copa incorporates three ideas: first, a target window to aim for that is inversely proportional to the measured queueing delay; second, a window update rule that depends moves the sender toward the target rate; and third, a TCP-competitive strategy to compete well with buffer-filling flows [7].

Copa keeps track of the smoothed RTT values in *progapation_time* similar to 5.2.1. For each congestion control update, Copa computes *rtt_standing*, which is the RTT corresponding to a "standing" queue, and it is the minimum observed RTT in a recent time window of *progapation_time*. Copa calculates the queueing delay, denoted by d_q, in the network by

$$d_q = rtt\_standing - min\_rtt \tag{5.4}$$

where *min_rtt* is the smallest RTT observed over a long period of time. The reason for using the smallest RTT in the recent *progapation_time* duration, rather than the latest RTT sample, is for robustness in the face of acknowledgment compression [57] and network jitter, which increases the RTT and can confuse the sender into believing that a longer RTT is due to queueing on the forward data path. Note that WebRTC uses acknowledgment compression when sending the packet feedback.

Copa estimates the target congestion window, denoted by *target_cwnd*, using this function:

$$target\_cwnd = p \cdot progapation\_time/(q_d \cdot \delta) \tag{5.5}$$

where p is the average packet sizes and δ is a parameter between 0 and 1 to control network utilization and network queueing delay tradeoff. δ can be tuned and is application-specific; higher values of δ mean lower queueing delay. Note that if $q_d = 0$, there is no queue build-up in the network, and it is safe to set $target\_cwnd = +\infty$.

If the current congestion window *cwnd*, exceeds the target, the sender reduces *cwnd*; otherwise, it increases *cwnd*. Let us denote the number of bytes acked since the last update of *cwnd* by B. The update rule for the window is:

$$cwnd = \begin{cases} cwnd + p \cdot v \cdot B/(\delta \cdot cwnd) & increase \\ cwnd - p \cdot v \cdot B/(\delta \cdot cwnd) & decrease \end{cases} \tag{5.6}$$

where v is called the ”velocity parameter”. The velocity parameter, v, speeds up the convergence. It is initialized to 1. Once per update of *cwnd*, the sender compares the current *cwnd* to the *cwnd* value when the latest acknowledged packet was sent (i.e., *cwnd* at the start of the current window). If the current *cwnd* is larger than *target_cwnd*, then set the direction to "up"; if it is smaller than *target_cwnd*, then set the direction to "down." If the direction is the same as in the window, then double v. If not, then reset v to 1. However, start doubling v only after the direction has remained the same for three RTTs [7].

Chapter 6

Evaluation

We evaluate Dumbo in a simulation environment and atop a WebRTC-based implementation. We describe our setup in §6.1 and use it to compare existing baselines in §6.3. In §6.2, we motivate our system design.

6.1 Setup

Testbed. Inspired by [16], we built a testbed in C++ on top of the latest version of WebRTC that enables a peer-to-peer video call between two endpoints in a headless setting. Each endpoint could have a sender peer and a receiver peer. The sender reads the input video from a file instead of a webcam. Similarly, the receiver records the incoming video as a file. In order to track video frames from the sender to the receiver, we put a unique 2D barcode on each frame, similar to [19]. The barcode enables matching sent and received video frames to compute metrics such as frame latency and frame quality. We emulate different network conditions between the peers by putting the receiver behind a Mahimahi [36] shell.

Configuration. Each peer has specific configuration parameters that are passed by JSON files that indicate this information: if the peer is a sender or receiver, the peer's assigned IP and port, the duration of the call, video source (which could be from a file, webcam, or no video), audio source (which could be from a file, microphone, or no audio), video information (such as fps, width, and height), where to save the final video, the format to record the received video, and where to save logs related to the peers.

Metrics. We quantify the performance of a real-time video system using two primary metrics: frame quality and frame latency. The testbed evaluates frame quality as average Structural Similarity (SSIM [50]) to compare the received frames with the corresponding frames in the source video. The frame latency is determined by measuring the duration between the instant a frame is read at the sender and when it is ready for display at the receiver. For frames not received at the receiver, a virtual arrival time is assigned equal to the presentation time of the subsequently displayed frame [19].

For any congestion control algorithm, measuring its performance by metrics such as link utilization, queuing delay it introduces in the network, and round-trip time (RTT) is crucial. To measure the performance of the congestion controlalgorithm, we also log all the packet sizes and their types when sending or receiving them to compute the bitrate associated with each stream. We must identify what proportion of the link is filled with dummy data; thus, we find the media stream bitrate and the dummy bitrate and their ratio. Furthermore, in the emulation network, we process the mahimahi log and compute the arrival rate, departure rate, capacity, and the per-packet queuing delay of the experiment.

Traces. We evaluate each scheme on 16 cellular traces bundled with Mahimahi [36]. We also have created our custom pulse traces to show simple concepts of convergence in 3. In a trace file, each line signifies a packet delivery opportunity, indicating the time when an MTU-sized packet can be delivered in the emulation [36]. Byte-level accounting is used, where each delivery opportunity corresponds to the possible delivery of 1500 bytes. As such, a single line in the trace file can represent the delivery of multiple smaller packets whose cumulative sizes add up to 1500 bytes. We also used the pre-recorded cellular traces that are available online. We connected to the links in our real-life experiments and recorded the time series [36].

Videos. We use a YUV video dataset that we generated from youtube. All the videos have 1920×1080 resolution and 10-minute length at 30 fps. Generating the dataset is

	Videos	
Youtuber	Total Len.	Avg. Bitrate
Adam Neely	10 min	1082 kbps
Xiran Jay Zhao	10 min	2815 kbps
The Needle Drop	10 min	2013 kbps
fancy fueko	10 min	4064 kbps
Kayleigh McEnany	10 min	2521 kbps

Table 6.1: Details of our dataset. All videos are at 1920×1080.

fully scripted and can be replicated by anyone. The video URLs, start time, and end time are available, and we have provided the scripts to download, pre-process, crop them to the desired resolution, and re-encode at 30 fps. Prior to the experiments, each video is barcoded using the technique described in **Metrics** section offline. Table 6.1 displays the details of our dataset.

Baselines. We evaluated WebRTC's default congestion control algorithm, GCC, in emulation networks. We also implemented Copa and Rocc in WebRTC, discussed in 5.2, and evaluated their performance. WebRTC's main code-base provides the implementation of PCC [15] as well that we discuss in Appendix A.

6.2 Understanding Dumbo's Design

6.2.1 Convergence Time

This section studies the transient and steady-state behavior of different schemes in variable-bandwidth environments. We run all the schemes on a pulse-shaped link. The link starts with 3 Mbps of bandwidth for 40 seconds, then drops to 500 Kbps for the next 40 seconds before jumping back up to 3 Mbps again. The minimum network one-way delay is 25 ms with a round-trip time (RTT) of 50 ms, and the buffer size at the bottleneck is large enough that there are no packet drops. We run the experiment for 160 seconds.

We define the convergence time of an algorithm after a link transition to up as the time it takes to achieve 90% utilization of the max utilization it achieves after the transition for the first time. We also define the convergence time of an algorithm after a link transition to down as the time it takes for it to achieve 110% frame latency of the steady-state latency it had before the transition. We also define the steady-state metrics of an algorithm after a link transition as the average of that metric after reaching a steady behavior.

Rocc. Fig. 6-3 shows the results of the convergence experiment for RoCC. For RoCC without Dumbo, on a link transition to up, it has a convergence time of 21.1 s, the max utilization of 2.1 Mbps, and a steady-state utilization of 1.5 Mbps. On a link transition to down, RoCC without Dumbo has a convergence time of 1.1 s, a steady-state frame latency of 173 ms, a max frame latency of 750 ms, and a steady-state utilization of 0.47 Mbps. For RoCC with Dumbo, on a link transition to up, it has a convergence time of 700 ms, the max utilization of 3 Mbps, and a steady-state utilization of 3 Mbps. On a link transition to down, RoCC with Dumbo has a convergence time of 1.7 s, a steady-state frame latency of 204 ms, a max frame latency of 1067 ms, and a steady-state utilization of 0.5 Mbps.

On a link transition to up, Dumbo has a 30x faster convergence time and 2x steady-state link utilization for RoCC while having comparable results for a link transition to down.

Copa. Fig. 6-6 shows the results of the convergence experiment for Copa. For Copa without Dumbo, on a link transition to up, it has a convergence time of 2.8 s, the max utilization of 3 Mbps, and a steady-state utilization of 2.5 Mbps. On a link transition to down, Copa without Dumbo has a convergence time of 2 s, a steady-state frame latency of 169 ms, a max frame latency of 1378 ms, and a steady-state utilization of 0.41 Mbps. For Copa with Dumbo, on a link transition to up, it has a convergence time of 250 ms, the max utilization of 3 Mbps, and a steady-state utilization of 3 Mbps. On a link transition to down, Copa with Dumbo has a convergence time of 2 s, a steady-state frame latency of 178 ms, a max frame latency of 1463 ms, and a steady-state utilization of 0.5 Kbps.

On a link transition to up, Dumbo has an 11x faster convergence time for Copa while having comparable results for a link transition to down.

GCC. Fig. 6-9 shows the results of the convergence experiment for GCC. On a link

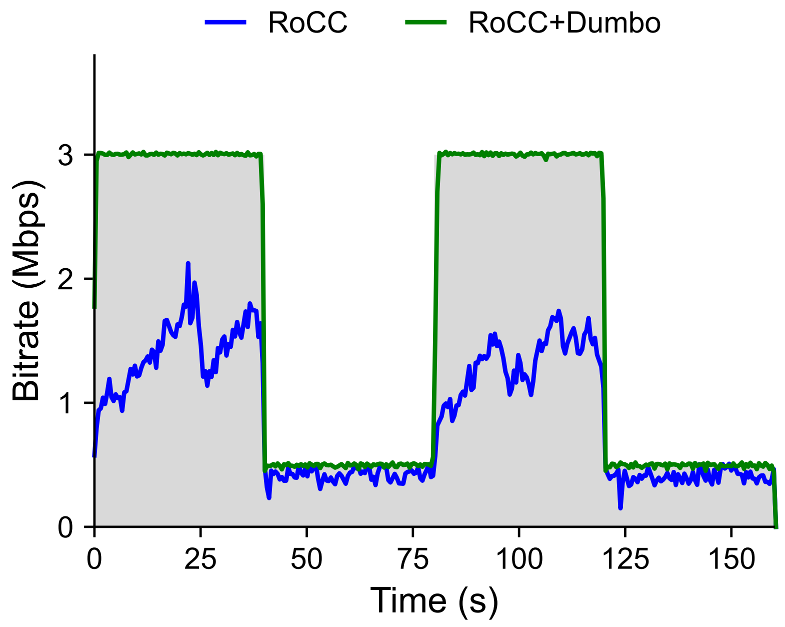

Figure 6-1: Link's utilization time series.

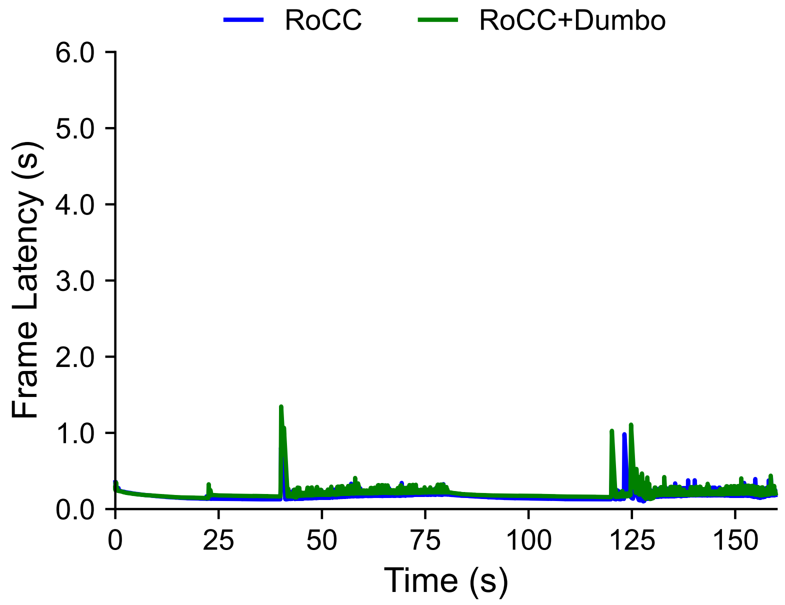

Figure 6-2: Frame latency time series.

Figure 6-3: Convergence measurement for RoCC. The experiment was run for 160 s with a one-way minimum delay of 25 ms. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps.

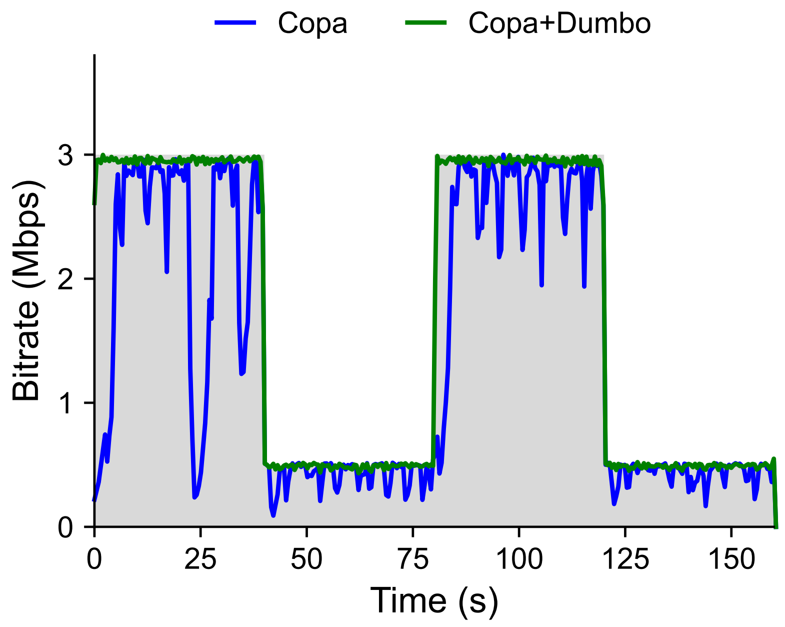

Figure 6-4: Link's utilization time series.

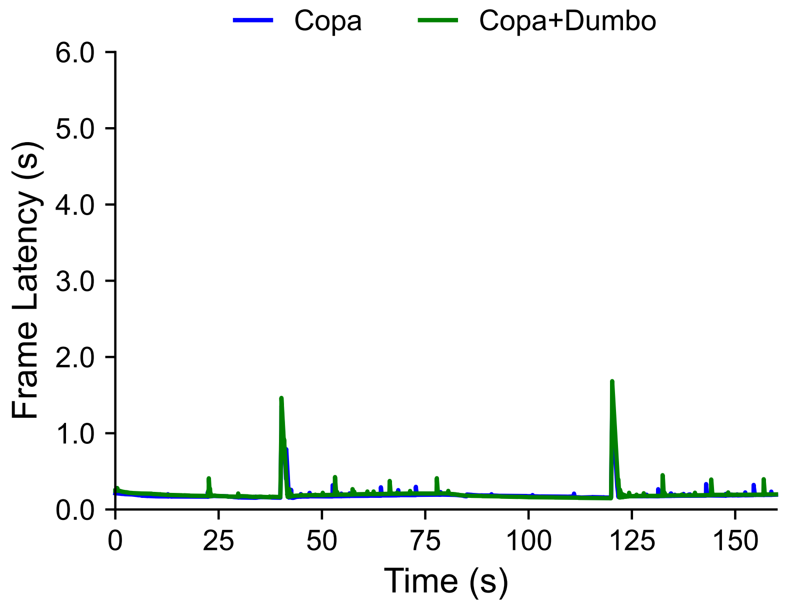

Figure 6-5: Frame latency time series.

Figure 6-6: Convergence measurement for Copa. The experiment was run for 160 s with a one-way minimum delay of 25 ms. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps.

transition to up, GCC has a convergence time of 16.6 s, the max utilization of 3 Mbps, and a steady-state utilization of 2.8 Mbps. On a link transition to down, GCC has a convergence time of 10.5 s, a steady-state frame latency of 260 ms, a max frame latency of 5.1 s, and a steady-state utilization of 0.46 Mbps.

On a link transition to up, Dumbo has a 35x faster convergence time and higher steady-state utilization. On a link transition to down, Dumbo has a 5.6x faster convergence time, a 4x lower max frame latency, and comparable steady-state frame latency and utilization.

6.2.2 Impact of Dummy Traffic

To evaluate the effect of the dummy stream on the window-based congestion control algorithms, we compared the performance of ROCC and Copa with and without the dummy stream on all 16 cellular traces. All experiments are run for 2 minutes with a link with a one-way minimum delay of 25 ms and no packet drops.

RoCC

Fig. 6-13 shows the comparison of end-to-end frame metrics of RoCC with and without the dummy traffic on all the cellular traces. On average, RoCC without a dummy has a PSNR of 39.90, SSIM of 13.90, and frame latency of 557 ms. RoCC+Dumbo has a PSNR of 41.1, SSIM of 14.79, and frame latency of 687 ms. **Dumbo shows an average PSNR improvement of around 1.2 dB and SSIM improvement of 1.07 dB while increasing 23% (130 ms) in the frame latency in RoCC**. Fig. 6-17 compares network metrics on all the cellular traces. On average, RoCC without a dummy has a network delay of 37 ms, a network utilization of 40.62%, and a video bitrate of 1804 Kbps, respectively. RoCC+Dumbo's are 49 ms, 69.88 %, and 3197 Kbps, respectively. **Dumbo achieves on average a 77% increase in the video bitrate and a 72% increase in network utilization while slightly increasing the network delay by 12 ms compared with RoCC**.

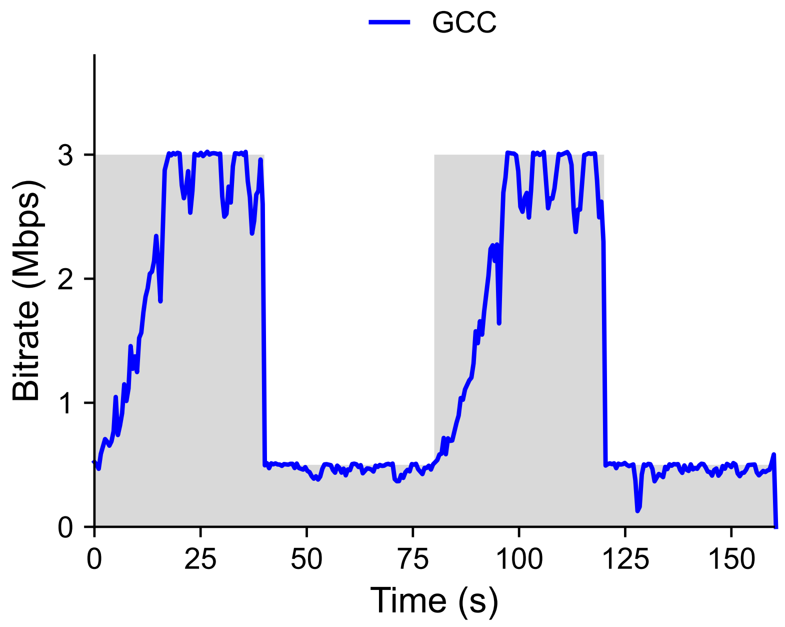

Figure 6-7: Link's utilization time series.

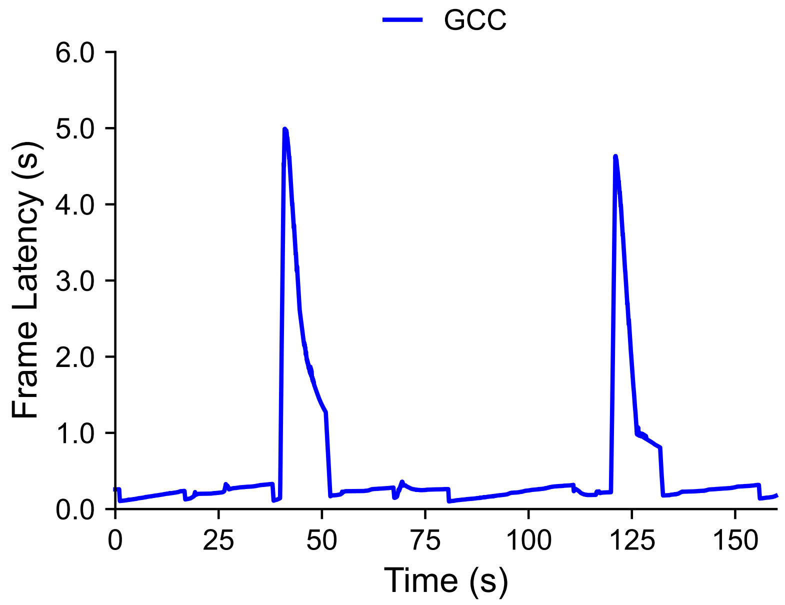

Figure 6-8: Frame latency time series.

Figure 6-9: Convergence measurement for GCC. The experiment was run for 160 s with a one-way minimum delay of 25 ms. The link is a periodic on-off link with a maximum of 3 Mbps and a minimum of 500 Kbps.

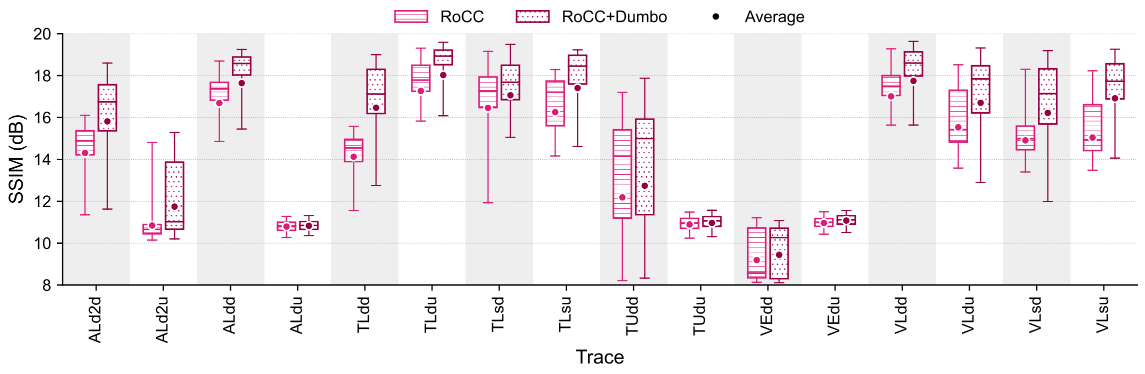

Figure 6-10: Per-frame SSIM statistics of the received video vs. the original video

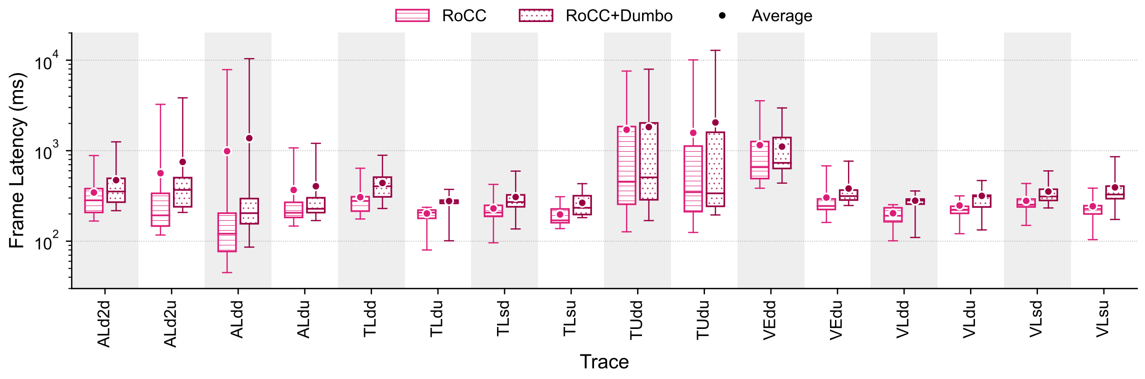

Figure 6-11: Per-frame latency statistics of the received video vs. the original video

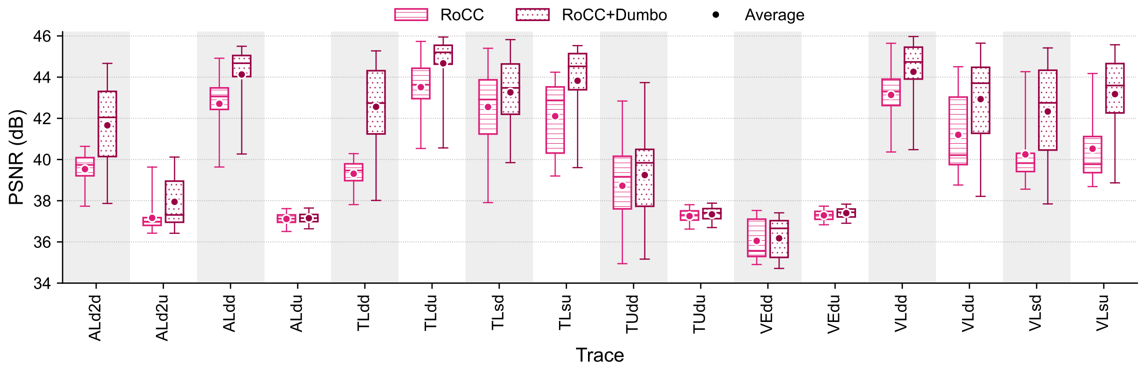

Figure 6-12: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-13: End-to-end statistics of quality of experience metrics of RoCC without the dummy traffic and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

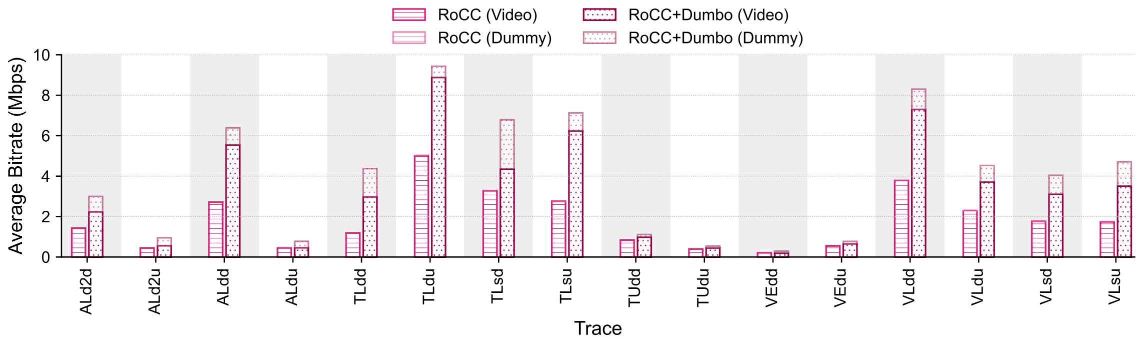

Figure 6-14: Average throughput of Video traffic vs Dummy traffic for different schemes

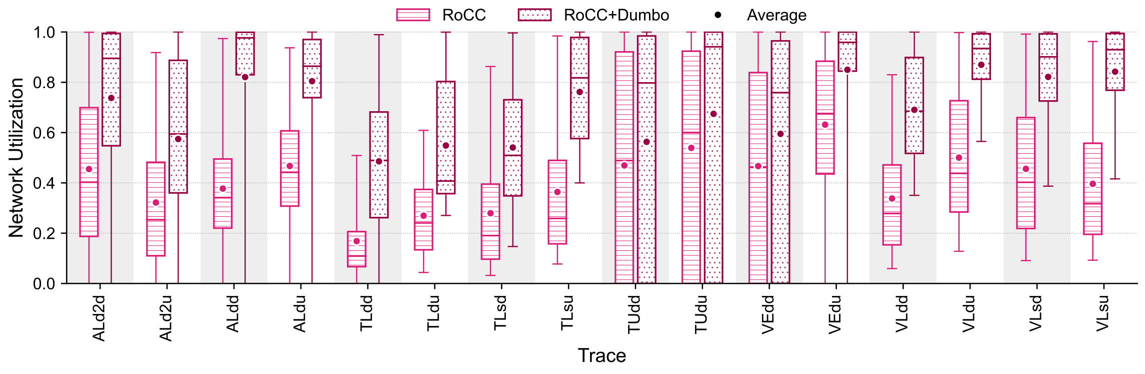

Figure 6-15: Statistics of link's utilization per sample of network bitrate.

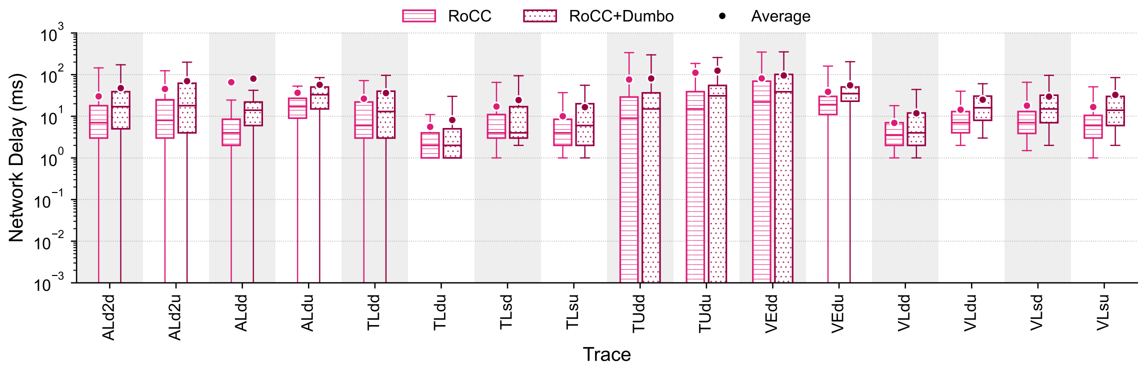

Figure 6-16: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-17: Network statistics of RoCC without the dummy traffic and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with one-way minimum delay of 25 ms.

Copa

Fig. 6-21 shows the comparison of end-to-end frame metrics of RoCC with and without the dummy traffic on all the cellular traces. On average, Copa without a dummy has a PSNR of 37.19, SSIM of 0.83, and frame latency of 725 ms. Copa+Dumbo has a PSNR of 40.64, SSIM of 14.05, and frame latency of 874 ms.**Dumbo shows an average PSNR improvement of around 3.45 dB and SSIM improvement of 3.22 dB while having an increase of 20% (149 ms) in the frame latency in RoCC**. Fig. 6-25 compares network metrics on all the cellular traces. On average, Copa without a dummy has a network delay of 36 ms, a network utilization of 22.15%, and a video bitrate of 458 Kbps, respectively. Copa+Dumbo's measurements are 46 ms, 54.67%, and 3129 Kbps. **Dumbo achieves on average a 5.8x increase in the video bitrate and a 1.47x increase in network utilization while slightly increasing the network delay by 10 ms compared with Copa**.

6.3 Overall Comparison with GCC

To compare GCC with Dumbo, we evaluated Copa+Dumbo and RoCC+Dumbo on all 16 cellular traces. All experiments are run for 2 minutes with a link with a one-way minimum delay of 25 ms and no packet drops. Fig. 6-29 shows the comparison of end-to-end frame metrics of GCC, Copa+Dumbo, and RoCC+Dumbo on all the cellular traces. On average, GCC has a PSNR of 39.41, SSIM of 12.82, and frame latency of 873 ms. **Dumbo schemes show an average PSNR improvement of around 1.5 dB and SSIM improvement of 1.6 dB while having a 100 ms lower frame latency compared with GCC**. Fig. 6-33 compares network metrics on all the cellular traces. On average, GCC has a network delay of 214 ms, a network utilization of 47.05%, and a video bitrate of 2706 Kbps, respectively. **Dumbo schemes achieve on average a 16% increase in the video bitrate and a 32% increase in network utilization while reducing the network delay by 4x compared with GCC**.

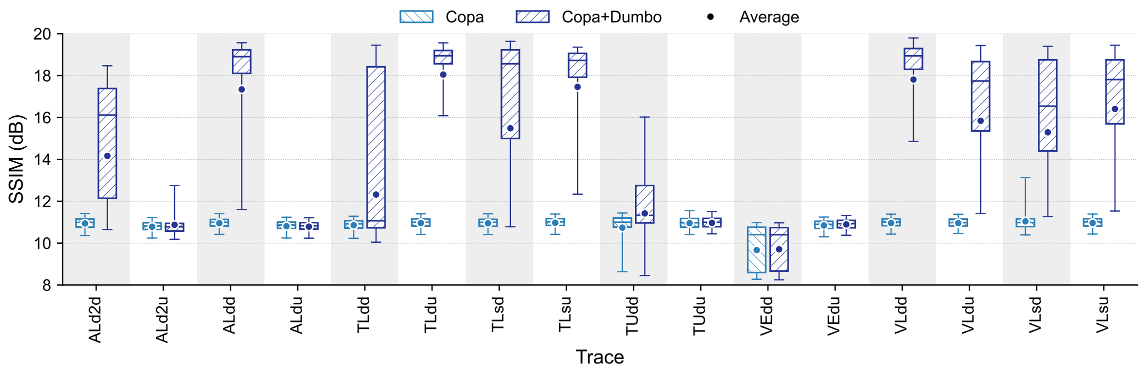

Figure 6-18: Per-frame SSIM statistics of the received video vs. the original video

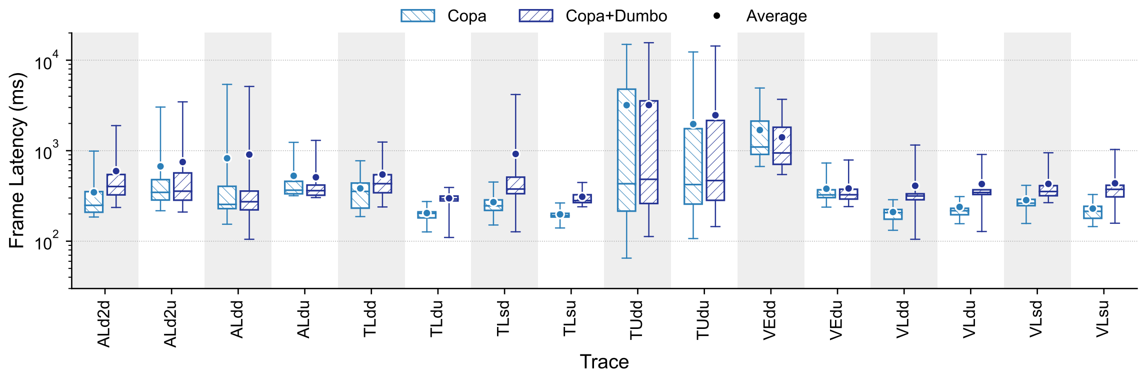

Figure 6-19: Per-frame latency statistics of the received video vs. the original video

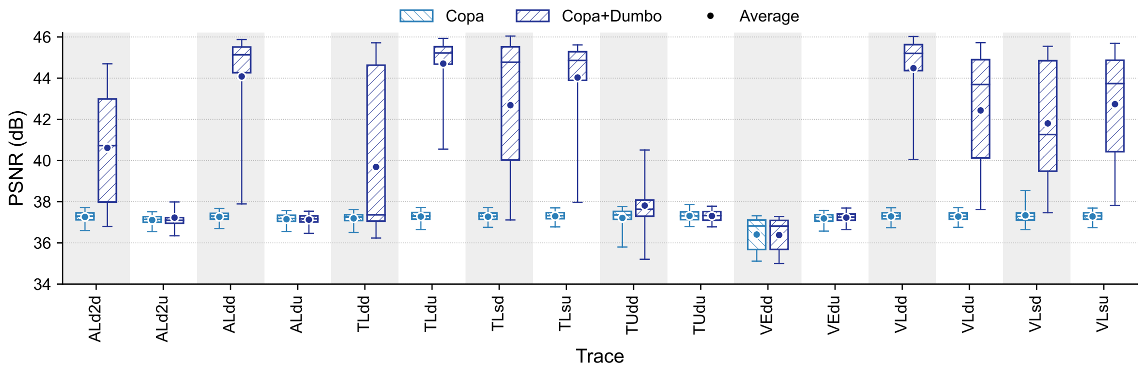

Figure 6-20: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-21: End-to-end statistics of quality of experience metrics of Copa without the dummy traffic and Copa+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

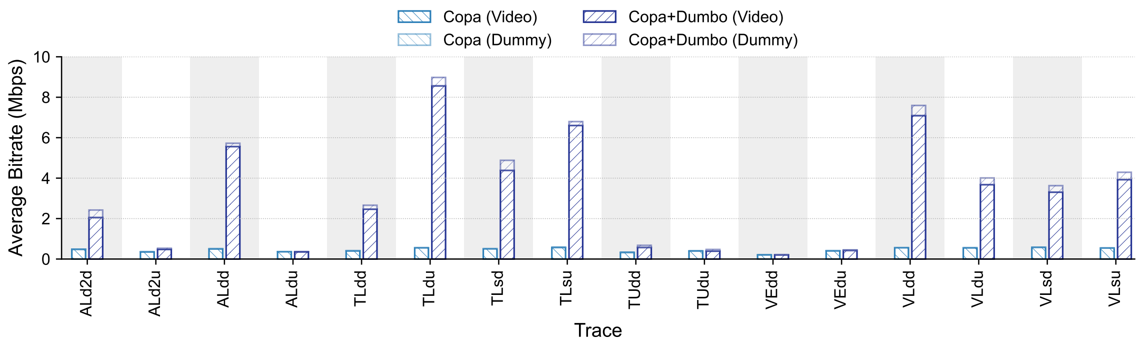

Figure 6-22: Average throughput of Video traffic vs Dummy traffic for different schemes

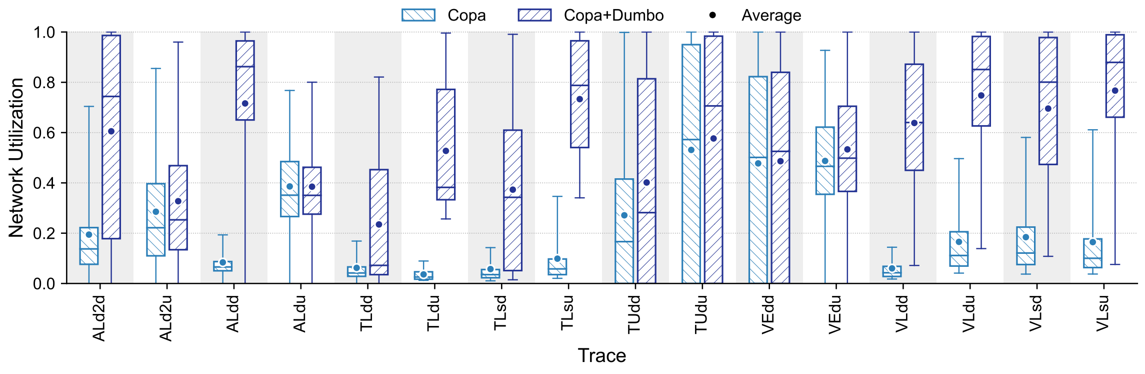

Figure 6-23: Statistics of link's utilization per sample of network bitrate.

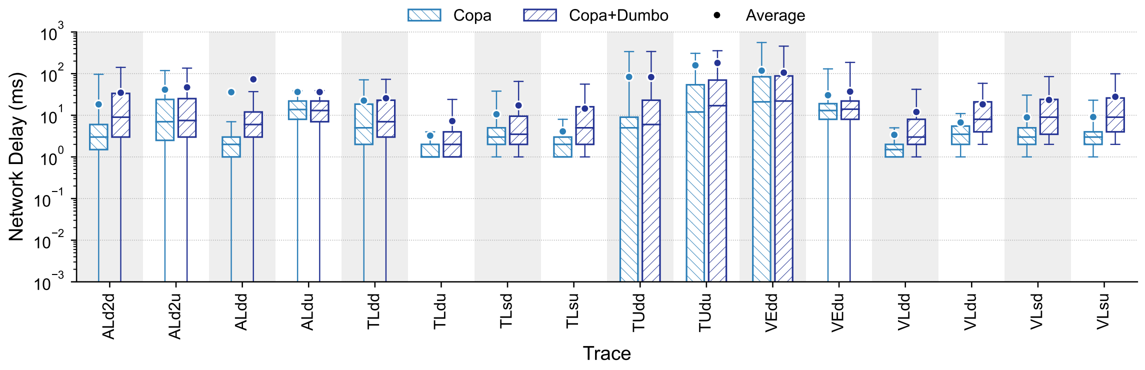

Figure 6-24: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-25: Network statistics of Copa without the dummy traffic and Copa+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with one-way minimum delay of 25 ms.

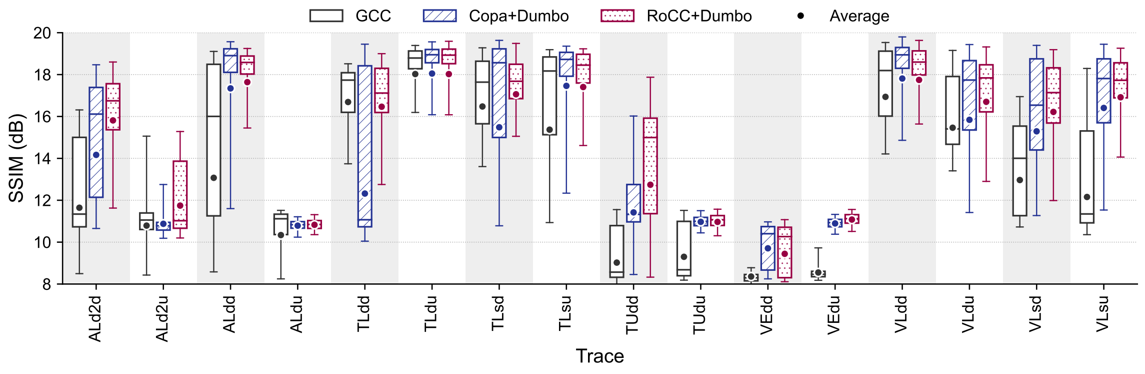

Figure 6-26: Per-frame SSIM statistics of the received video vs. the original video

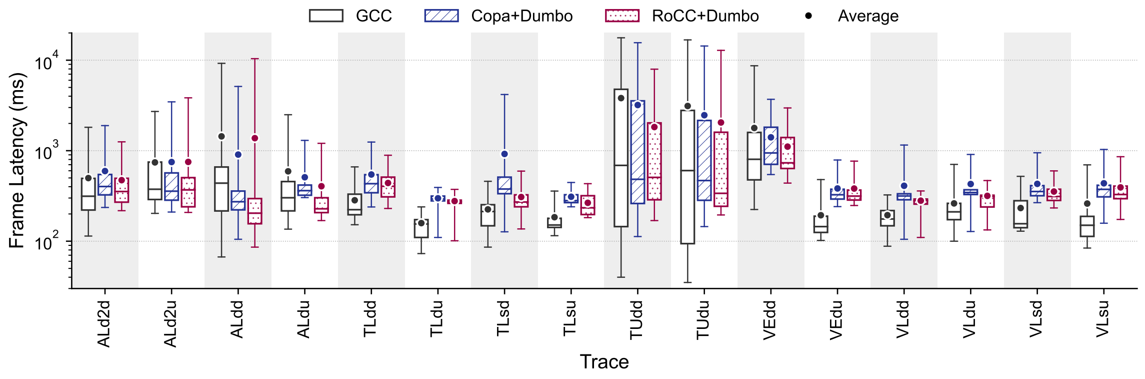

Figure 6-27: Per-frame latency statistics of the received video vs. the original video

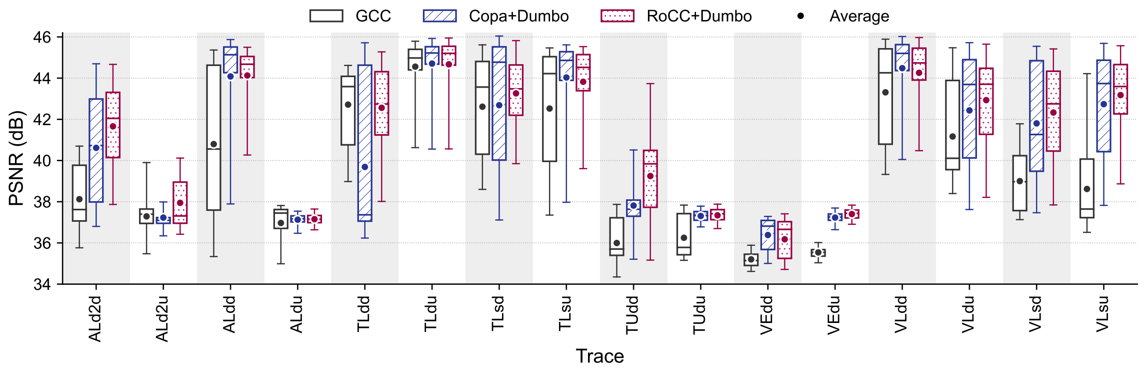

Figure 6-28: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-29: End-to-end statistics of quality of experience metrics of GCC, Copa+Dumbo, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

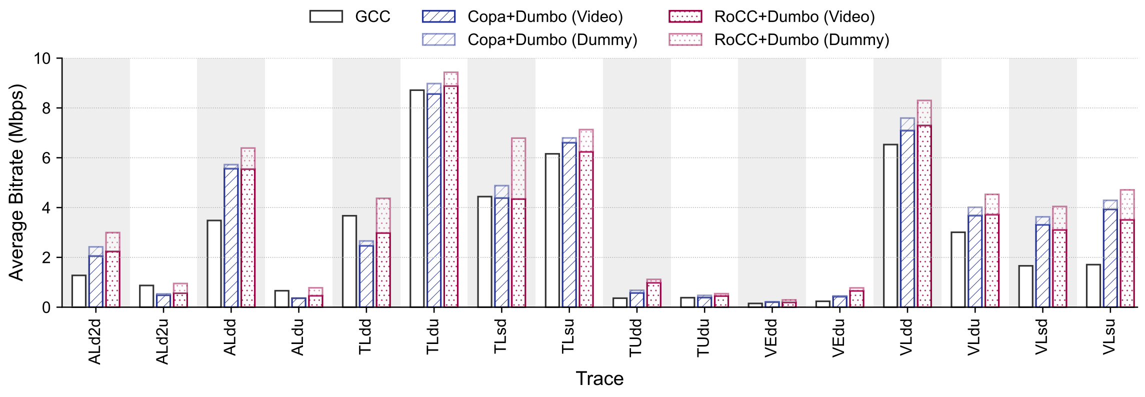

Figure 6-30: Average throughput of Video traffic vs Dummy traffic for different schemes

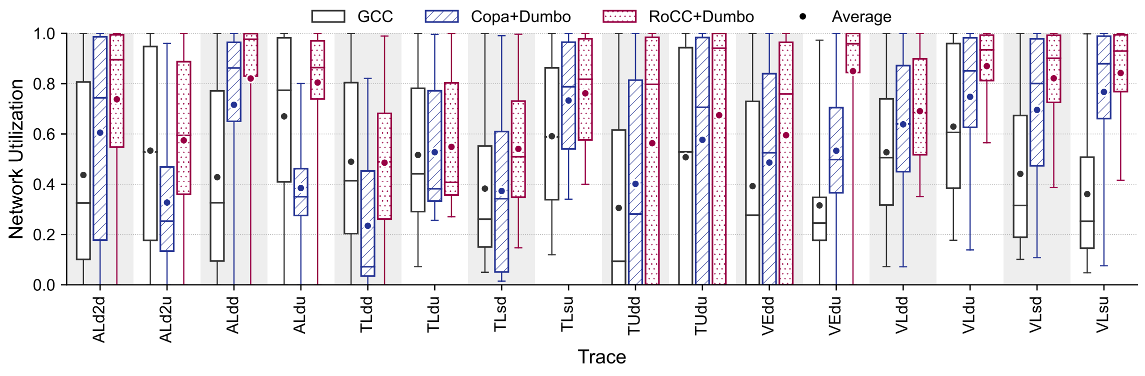

Figure 6-31: Statistics of link's utilization per sample of network bitrate.

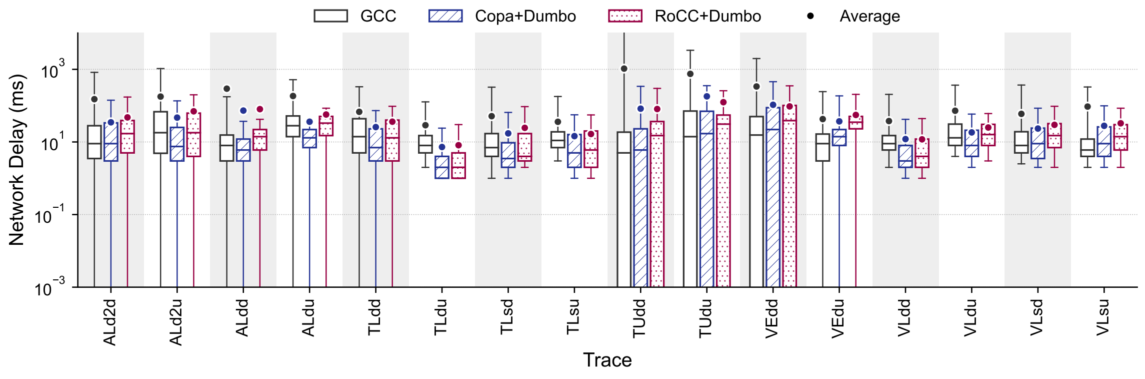

Figure 6-32: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-33: Network statistics of GCC, Copa+Dumbo, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with one-way minimum delay of 25 ms.

6.4 Probing

To compare Dumbo schemes with the probing mechanism in WebRTC, we enabled GCC with probing in WebRTC and ran our video application for all 16 cellular traces. All experiments are run for 2 minutes with a link with a one-way minimum delay of 25 ms and no packet drops. Fig. 6-37 shows the comparison of end-to-end frame metrics of GCC with probing enabled, Copa+Dumbo, and RoCC+Dumbo on all the cellular traces. On average, GCC with probing has a PSNR of 40.41, SSIM of 14.04, and a frame latency of 853 ms. **Dumbo schemes show an average PSNR improvement of around 0.47 dB and SSIM improvement of 0.41 dB while having a 72 ms lower frame latency compared with GCC with probing**. Fig. 6-41 compares network metrics on all the cellular traces. On average, GCC with probing has a network delay of 248 ms, a network utilization of 48.59%, and a video bitrate of 2428 Kbps, respectively. **Dumbo schemes achieve on average a 30% increase in the video bitrate and a 28% increase in network utilization while reducing the network delay by 5x compared with GCC with probing**.

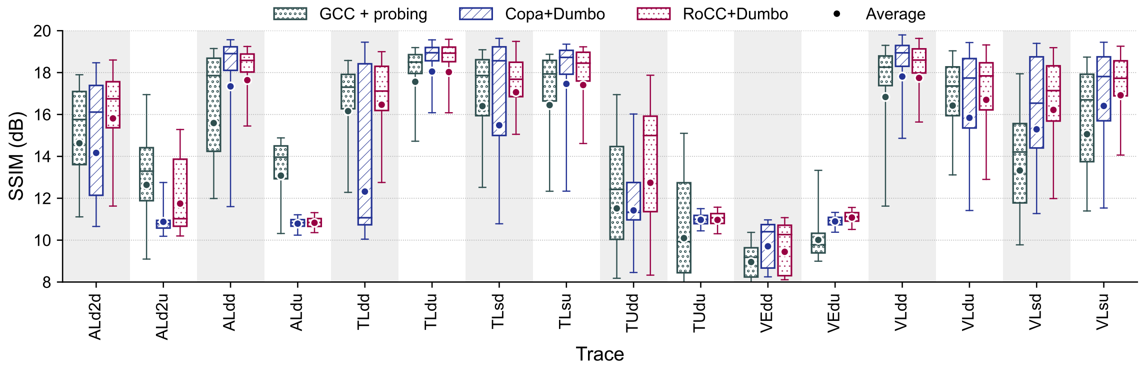

Figure 6-34: Per-frame SSIM statistics of the received video vs. the original video

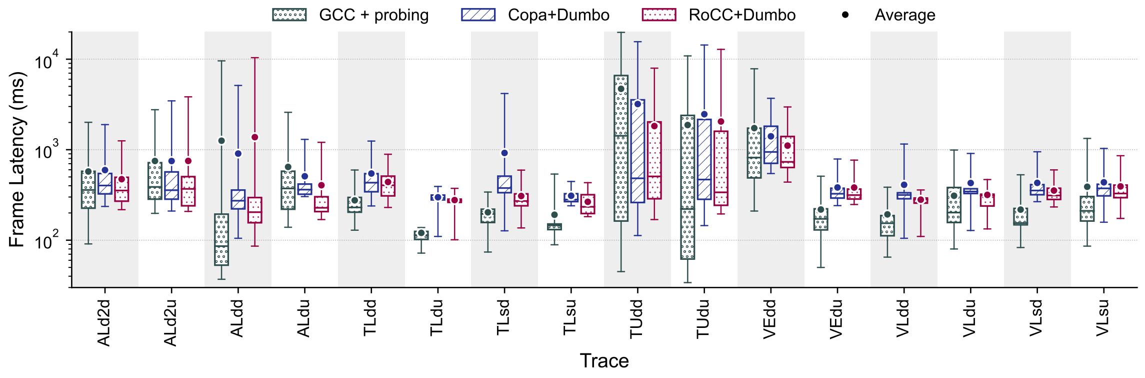

Figure 6-35: Per-frame latency statistics of the received video vs. the original video

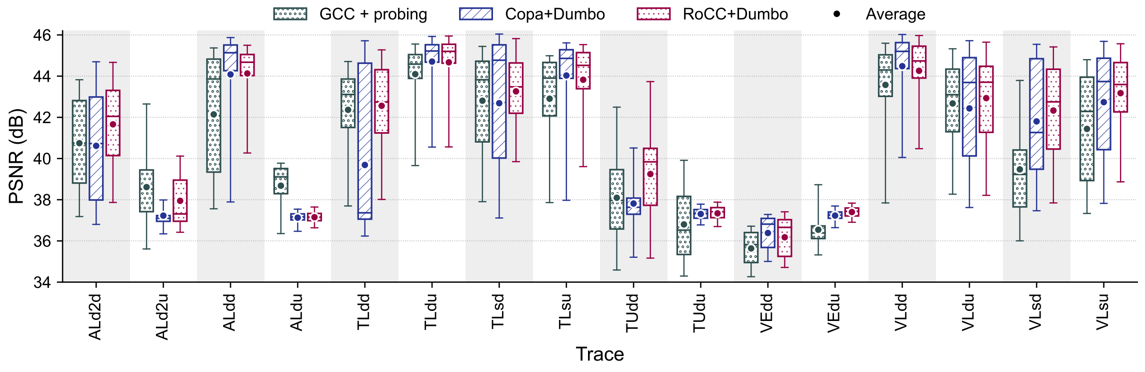

Figure 6-36: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-37: End-to-end statistics of quality of experience metrics of GCC+Probing, Copa+Dumbo, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

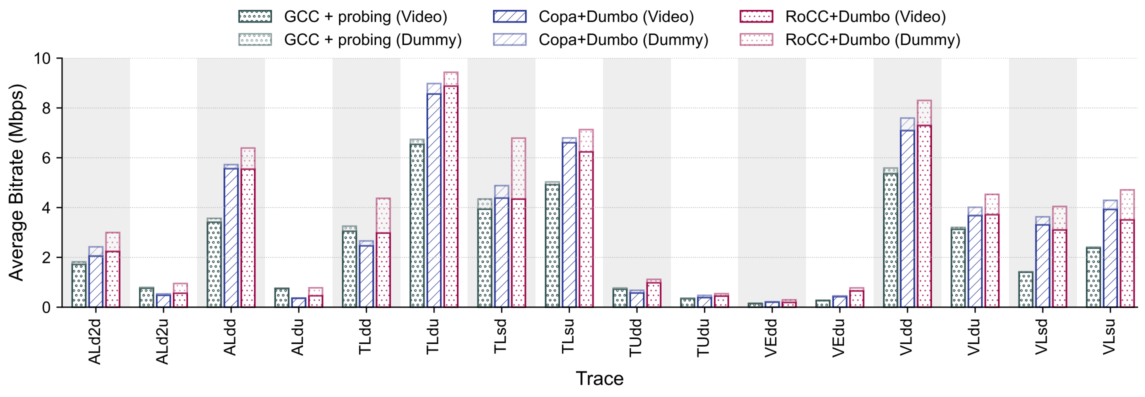

Figure 6-38: Average throughput of Video traffic vs Dummy traffic for different schemes

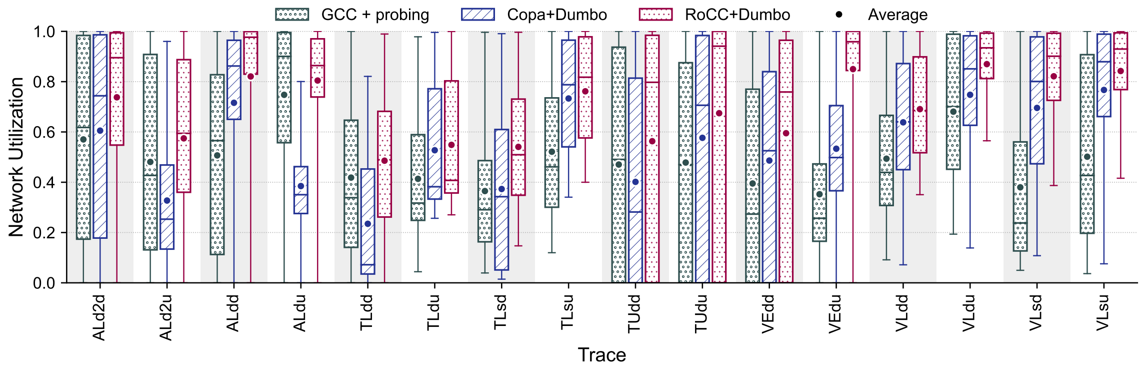

Figure 6-39: Statistics of link's utilization per sample of network bitrate.

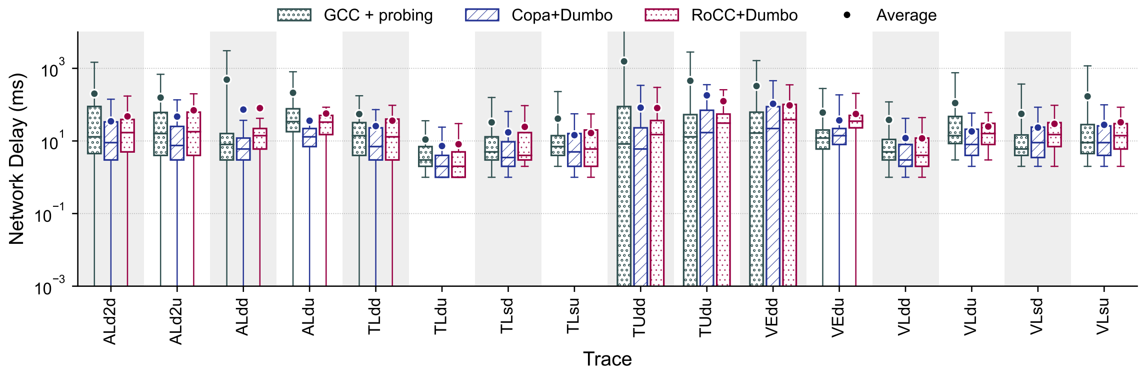

Figure 6-40: Per-frame PSNR statistics of the received video vs. the original video

Figure 6-41: Network statistics of GCC+Probing, Copa+Dumbo, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

Chapter 7

Conclusion

To bridge the gap between video congestion control algorithms and backlogged congestion control algorithms, we introduce our system called Dumbo. Dumbo effectively adapts non-backlogged video traffic to work with window-based, delay-sensitive backlogged congestion control algorithms, taking into consideration the unique characteristics of video traffic. The key insight behind Dumbo is that by generating dummy traffic when the congestion control algorithm expects data but no media packets are available, the congestion control system can maintain its feedback loop without disruption.

Dumbo schemes can adapt the real-time video stream to any delay-sensitive window-based backlogged congestion control algorithm. We implemented Dumbo on top of Google's implementation of WebRTC and demonstrated the performance of Dumbo for two algorithms: RoCC and Copa. We compared the resulting performance with current video congestion control algorithms. The Dumbo schemes demonstrate noteworthy improvements, such as an average PSNR enhancement of approximately 1.5 dB and a 1.6 dB improvement in SSIM, all while maintaining a 100 ms lower frame latency to GCC. Furthermore, the Dumbo schemes achieve significant benefits compared to GCC. On average, they yield a 16% increase in video bitrate, a 35x faster convergence time, and a 32% improvement in network utilization. Additionally, they reduce network delay by a factor of 4 when compared to GCC.

Appendix A

Supplementary Results

Comparison of Dumbo Schemes with PCC

To compare Dumbo schemes with the PCC [15] in WebRTC, we enabled PCC and ran our video application for all 16 cellular traces. All experiments are run for 2 minutes with a link with a one-way minimum delay of 25 ms and no packet drops. Fig. A-4 shows the comparison of end-to-end frame metrics of PCC, Copa+Dumbo, and RoCC+Dumbo on all the cellular traces.

PCC has a PSNR of 36.78, SSIM of 9.95, and a frame latency of 705 ms on average. **Dumbo schemes show an average PSNR improvement of around 4.1 dB and SSIM improvement of 4.5 dB while having a comparable frame latency compared with PCC**. Fig. A-8 compares network metrics on all the cellular traces. On average, PCC has a network delay of 69 ms, a network utilization of 9.85%, and a video bitrate of 980 Kbps, respectively. **Dumbo schemes achieve, on average, a 2.2x increase in the video bitrate and a 7x increase in network utilization while having a 20 ms smaller queuing delay.**

Adaptive Resolution Scheme

As mentioned in Sec. 5.1, we implemented a new resolution scheme to enable the video encoder to respond faster than WebRTC's original resolution scheme to our

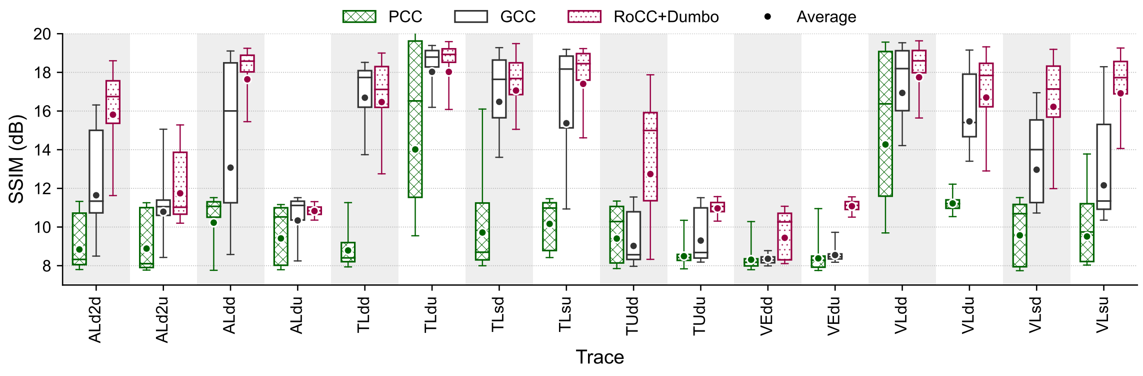

Figure A-1: Per-frame SSIM statistics of the received video vs. the original video

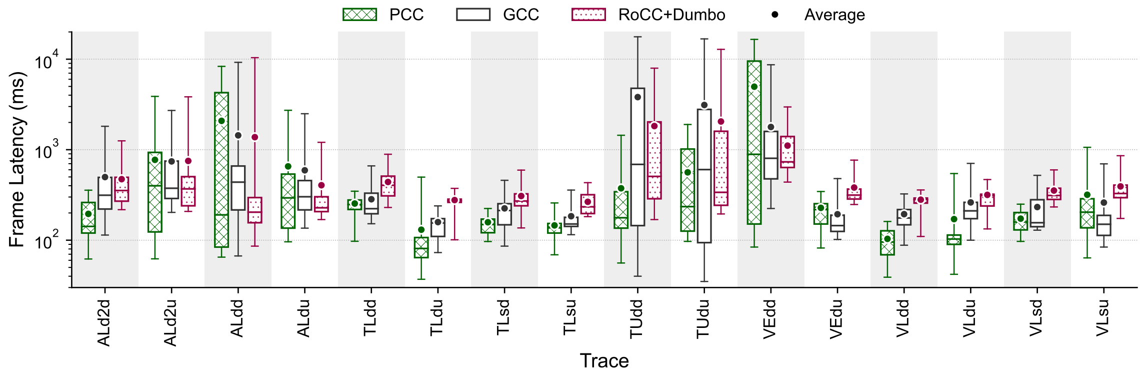

Figure A-2: Per-frame latency statistics of the received video vs. the original video

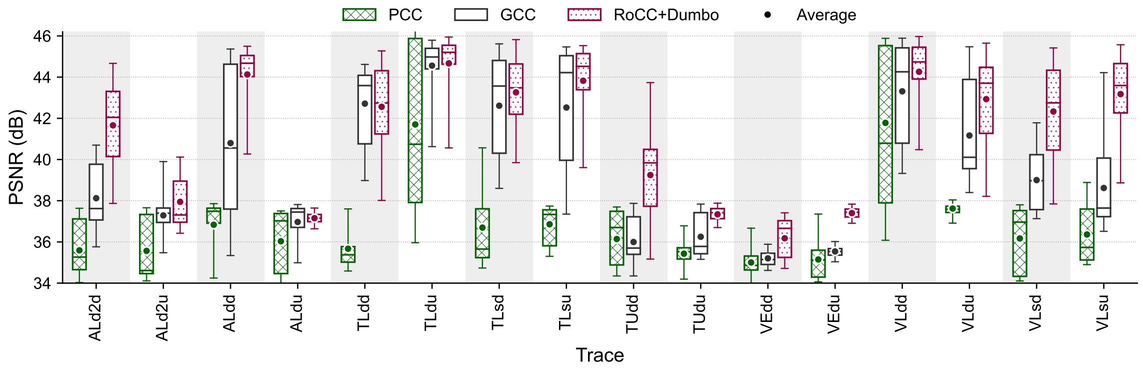

Figure A-3: Per-frame PSNR statistics of the received video vs. the original video

Figure A-4: End-to-end statistics of quality of experience metrics of PCC, GCC, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

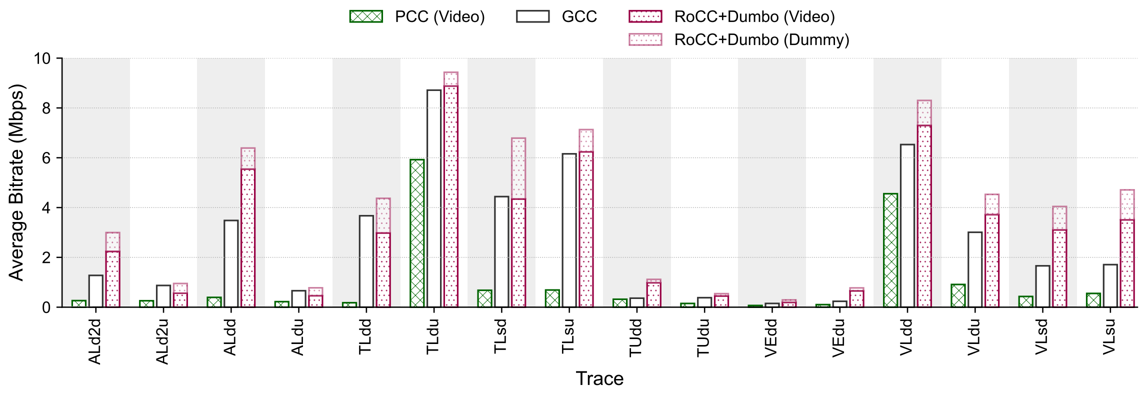

Figure A-5: Average throughput of Video traffic vs Dummy traffic for different schemes

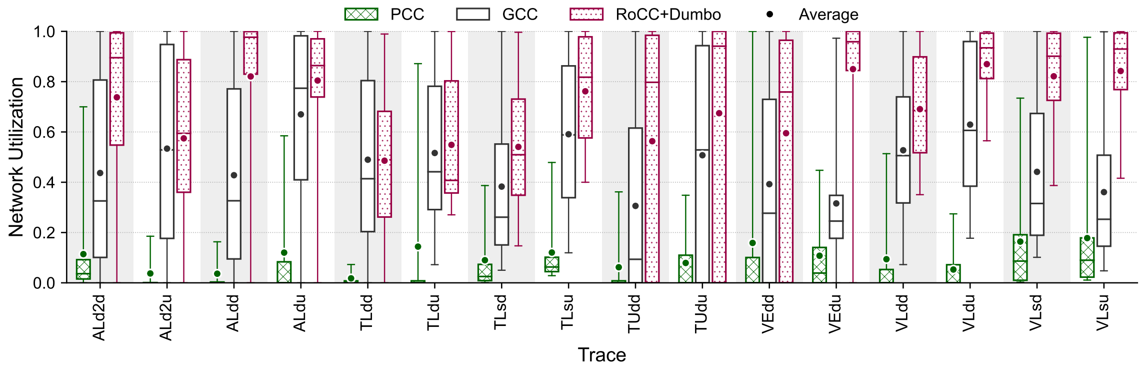

Figure A-6: Statistics of link's utilization per sample of network bitrate.

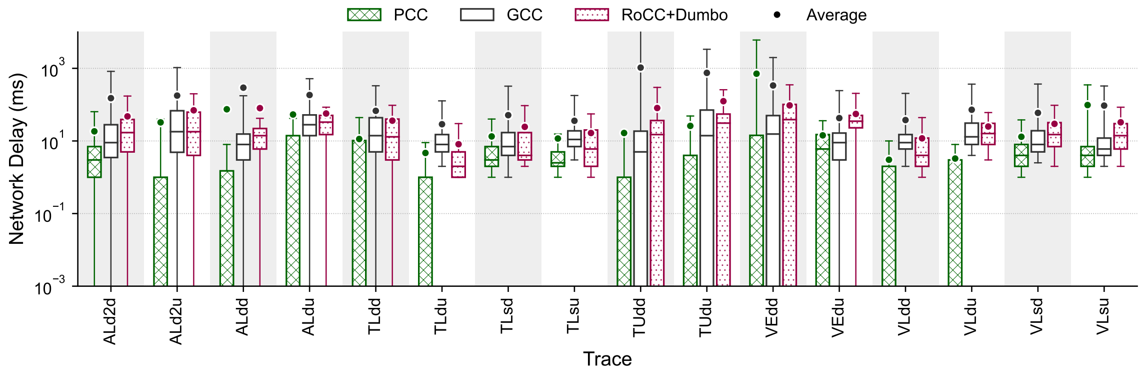

Figure A-7: Per-frame PSNR statistics of the received video vs. the original video

Figure A-8: Network statistics of PCC, GCC, and RoCC+Dumbo on all the Mahimahi cellular traces. All experiments are run for 2 minutes with one-way minimum delay of 25 ms.

Scheme	**Hyperparameter**	**Value**
RoCC	γ	0.1
	C	1500 bytes
	α	0.95
	Latency threshold	100 ms
Copa	δ	0.5
	p	1500 bytes
	α	0.95
	Latency threshold	100 ms
Setting	One-way delay	25 ms
	τ for adaptive resolution	500 ms
	Wait for resolution switch	4 s

Table A.1: Hyperparameter values for experiments.

fast congestion control algorithm. We ran GCC using WebRTC's original codebase (Original GCC) and GCC with our new adaptive resolution scheme to ensure the new changes have not hurt GCC's original performance. In our experiments, we ran our video application for all 16 cellular traces. All experiments are run for 2 minutes with a link with a one-way minimum delay of 25 ms and no packet drops. As seen in Fig. A-12 and Fig. A-16, the visual qualities, video bitrate, and network utilization have been improved for our adaptive resolution while maintaining the frame latency and the network queueing delay.

Hyperparameters

Table A.1 shows the hyperparameters we chose while running the experiments.

Visual Comparison

In this section, we show the visuals for the same frame for one of the cellular traces (ALd2u). The link has a minimum one-way delay of 25 ms, and the frame number is

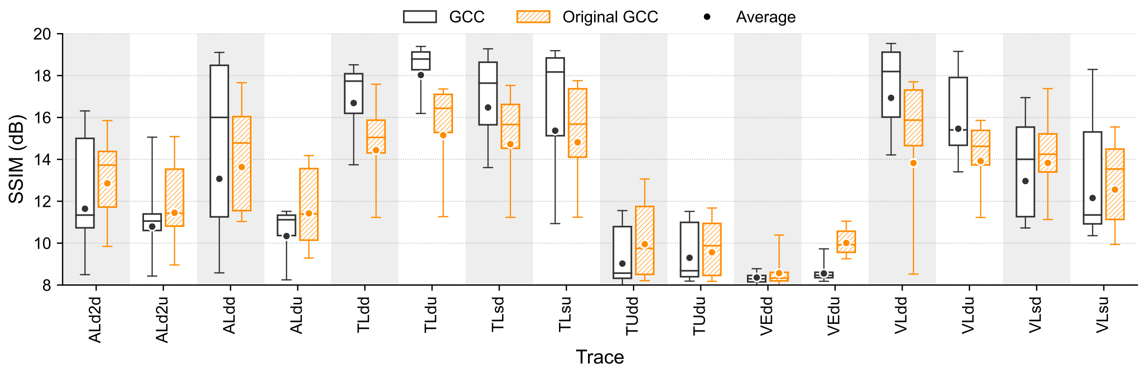

Figure A-9: Per-frame SSIM statistics of the received video vs. the original video

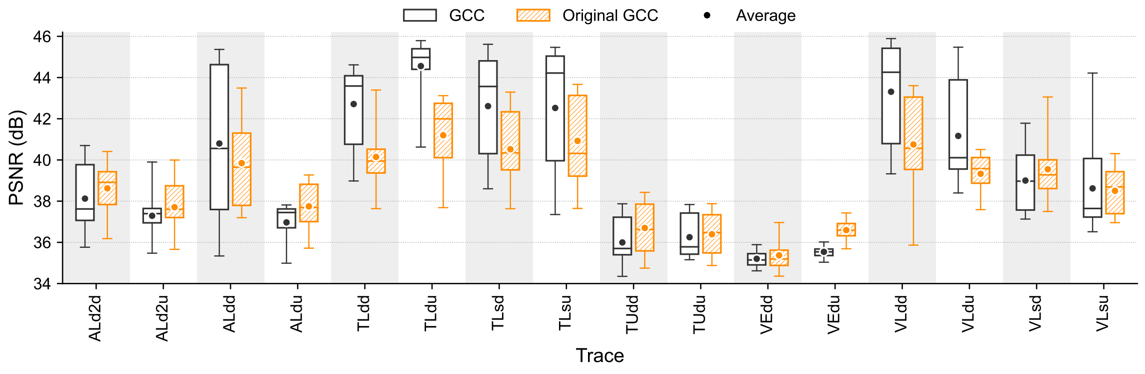

Figure A-10: Per-frame PSNR statistics of the received video vs. the original video

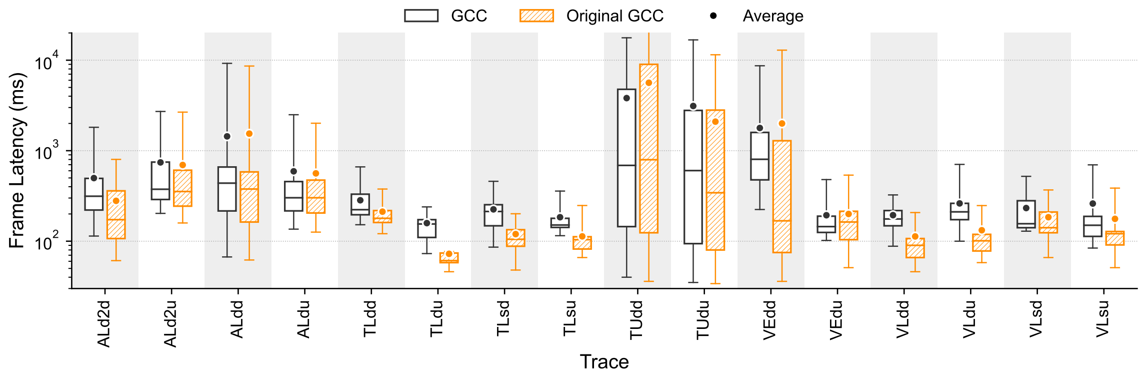

Figure A-11: Per-frame latency statistics of the received video vs. the original video

Figure A-12: End-to-end statistics of quality of experience metrics of original GCC and GCC with our adaptive resolution scheme on all the Mahimahi cellular traces. All experiments are run for 2 minutes with a one-way minimum delay of 25 ms.

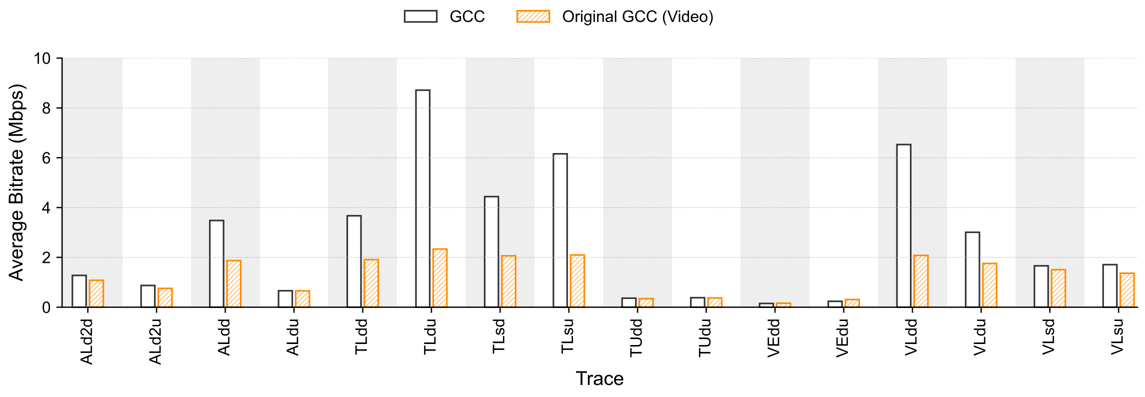

Figure A-13: Average throughput of Video traffic vs Dummy traffic for different schemes

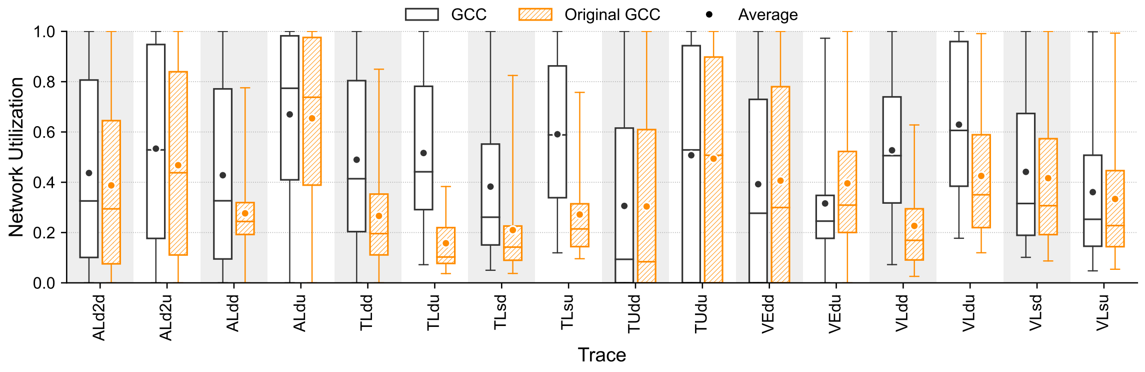

Figure A-14: Statistics of link's utilization per sample of network bitrate.

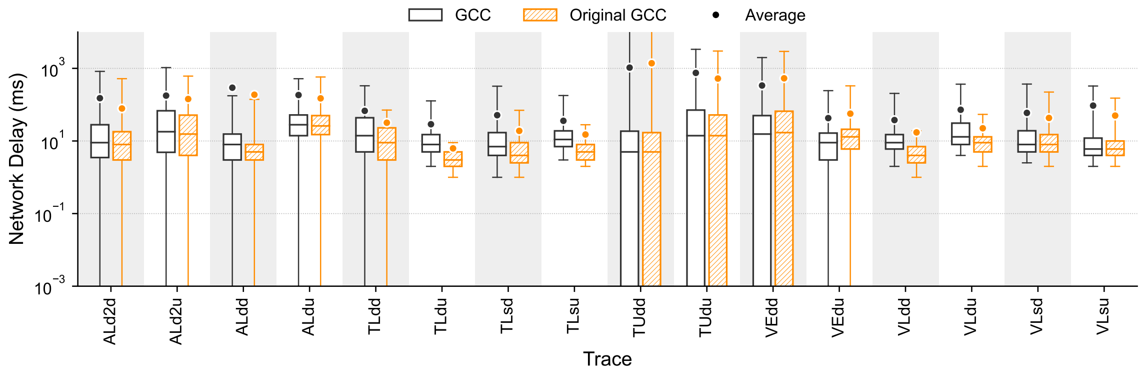

Figure A-15: Per-frame PSNR statistics of the received video vs. the original video

Figure A-16: Network statistics of original GCC and GCC with our adaptive resolution scheme on all the Mahimahi cellular traces. All experiments are run for 2 minutes with one-way minimum delay of 25 ms.

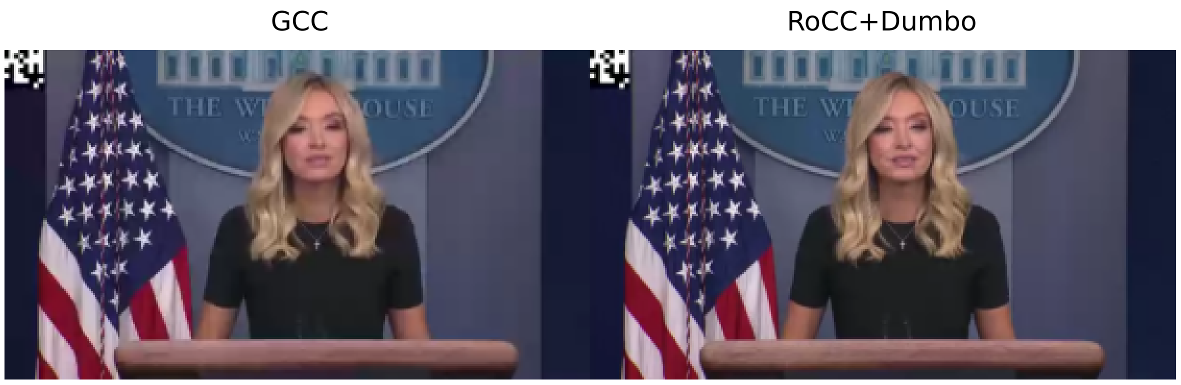

Figure A-17: Visual comparison of the same frame for GCC and RoCC+Dumbo

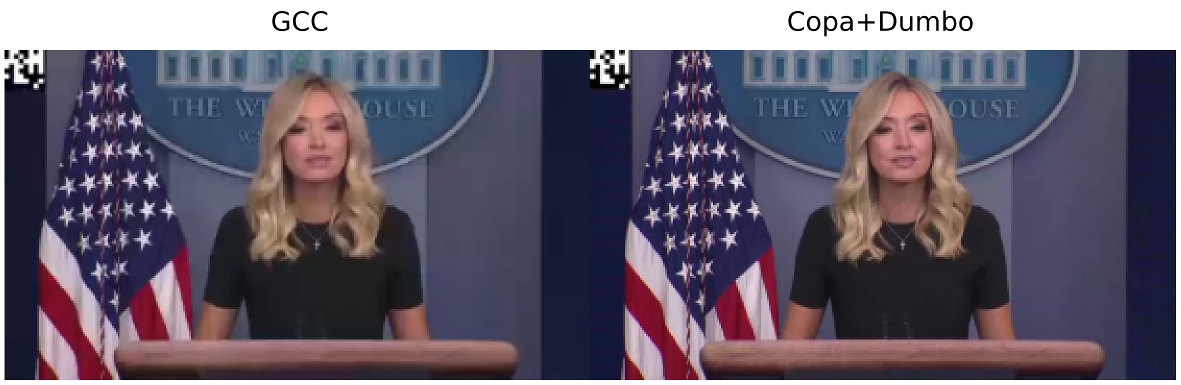

Figure A-18: Visual comparison of the same frame for GCC and Copa+Dumbo

691. We compare Dumbo schemes with GCC in Fig. A-17 and Fig. A-18, and PCC in Fig. A-19, Fig. A-20. For Dumbo, in comparison with GCC and PCC, the details of the face (around the eyes, the mouth, and the smile line) are clearer, the text in the background is sharper, and the necklace and the microphone are more visible. Fig. A-21 and Fig. A-22 compare the congestion control algorithms with and without Dumbo. Dumbo schemes have sharper images with more high-frequency details than their vanilla version. In all these examples, Dumbo frames have at least 1 dB higher PSNR values.

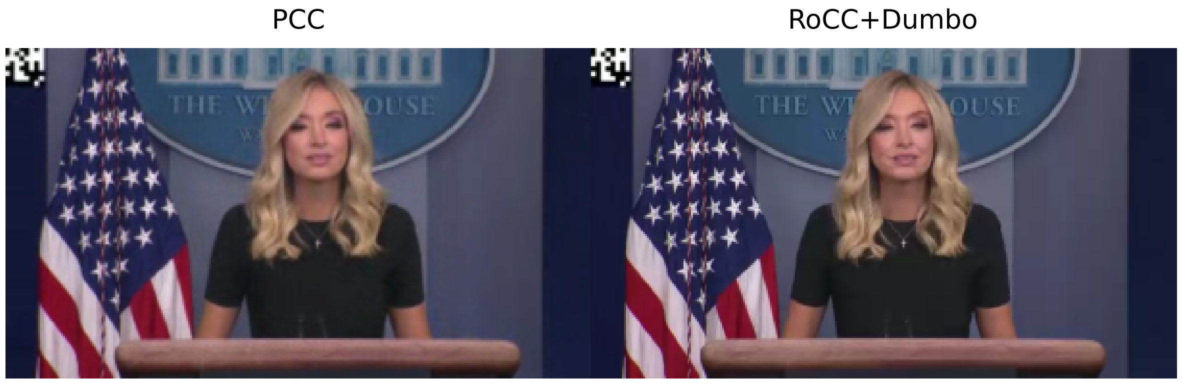

Figure A-19: Visual comparison of the same frame for PCC and RoCC+Dumbo

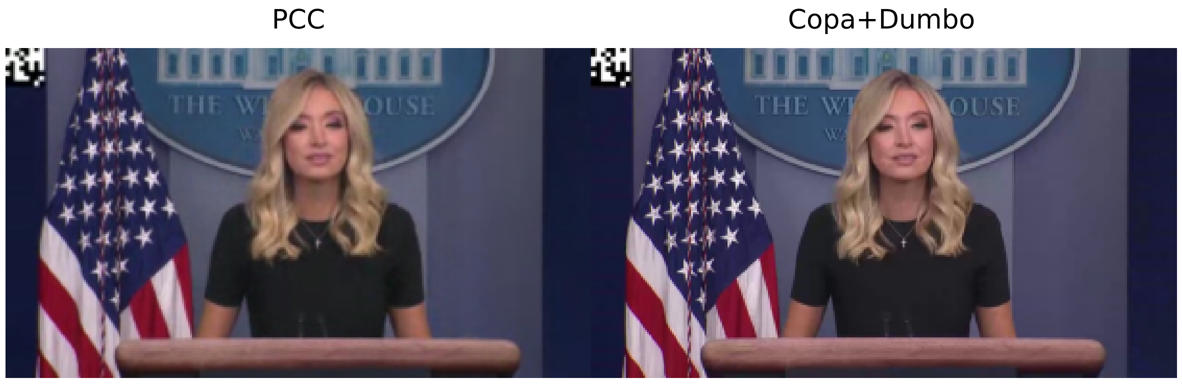

Figure A-20: Visual comparison of the same frame for PCC and Copa+Dumbo

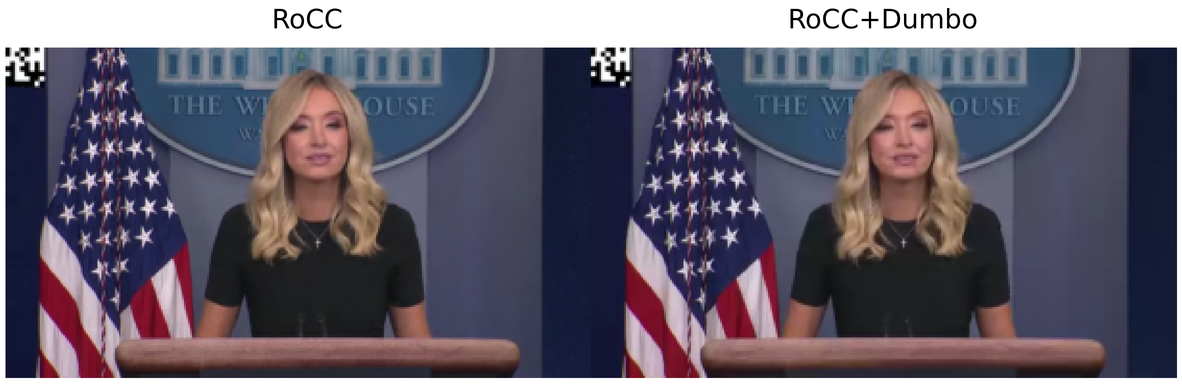

Figure A-21: Visual comparison of the same frame for RoCC and RoCC+Dumbo

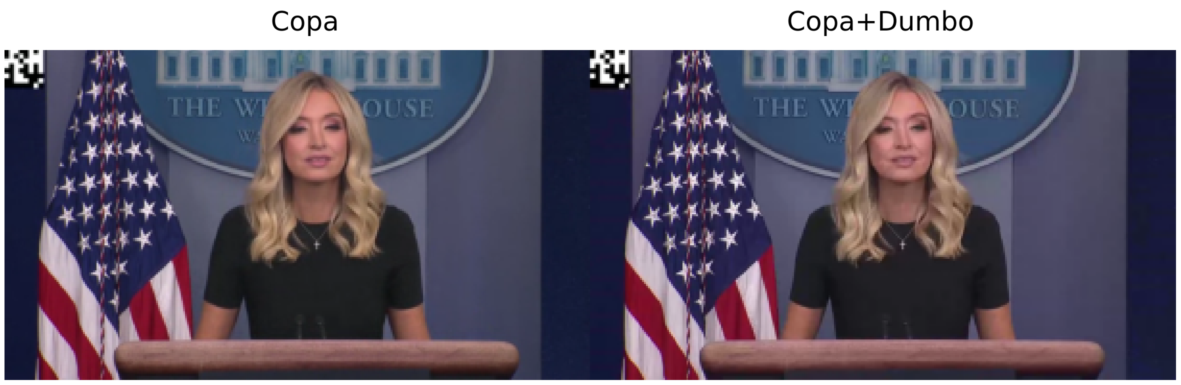

Figure A-22: Visual comparison of the same frame for Copa and Copa+Dumbo

Bibliography

[1] https://108anup.github.io/assets/papers/CCmatic-Hotnets22.pdf.

[2] https://chromium.googlesource.com/external/webrtc/modules/congestion_controller/probe_controller.cc.

[3] https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity.

[4] 8 Powerful Applications Built Using WebRTC. https://www.unitedworldtelecom.com/learn/webrtc-applications/.

[5] Cubic Quiescence: Not So Inactive. https://www.ietf.org/proceedings/94/slides/slides-94-tcpm-8.pdf.

[6] Updating TCP to Support Rate-Limited Traffic. https://www.rfc-editor.org/rfc/rfc7661.html.

[7] 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.

[8] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[9] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.

[10] 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.

[11] 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.

[12] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, and Silvio Valenti. The quest for ledbat fairness. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–6. IEEE, 2010.

[13] Hsuan-Yi Chou and Dana Edge. "they are happier and having better lives than i am": The impact of using facebook on perceptions of others' lives. *Cyberpsychology, Behavior, and Social Networking*, 15(2):117–121, 2012.

[14] DCTCP in Linux 3.18. http://kernelnewbies.org/Linux_3.18.

[15] 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.

[16] Jeongyoon Eo, Zhixiong Niu, Wenxue Cheng, Francis Y Yan, Rui Gao, Jorina Kardhashi, Scott Inglis, Michael Revow, Byung-Gon Chun, Peng Cheng, et al. Opennetlab: Open platform for rl-based congestion control for real-time communications. *Proc. of APNet*, 2022.

[17] Sally Floyd, Tom Henderson, and Andrei Gurtov. Rfc3782: The newreno modification to tcp's fast recovery algorithm, 2004.

[18] B. Ford, P. Srisuresh, and D. Kegel. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats). In *RFC 5389*, 2008.

[19] 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)*, pages 267–282, 2018.

[20] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 158–176, 2022.

[21] Luigi A Grieco and Saverio Mascolo. Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 34(2):25–38, 2004.

[22] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, jul 2008.

[23] David A Hayes and Grenville Armitage. Revisiting tcp congestion control using delay gradients. In *10th IFIP Networking Conference (NETWORKING)*, number Part II, pages 328–341. Springer, 2011.

[24] Chris V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1726–1734. IEEE, 2001.

[25] Stefan Holmer, Magnus Flodman, and Erik Sprang. RTP Extensions for Transport-wide Congestion Control. Internet-Draft draft-holmer-rmcat-transport-wide-cc-extensions-01, Internet Engineering Task Force, October 2015. Work in Progress.

[26] 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*, pages 187–198, 2014.

[27] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[28] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

[29] Saurabh Kapoor and Anubhav Bansal. Real-time video streaming: Techniques, challenges, and opportunities. *Journal of Network and Computer Applications*, 184:103067, 2021.

[30] Aleksandar Kuzmanovic and Edward W Knightly. Tcp-lp: low-priority service via end-point congestion control. *IEEE/ACM Transactions on Networking*, 14(4):739–752, 2006.

[31] Changhyun Lee, Chunjong Park, Keon Jang, Sue B Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *USENIX Annual Technical Conference*, pages 403–415, 2015.

[32] Zhipeng Li, Xiaohua Jiang, Zhi Gao, and Hui Zhang. Adaptive congestion control for multipath video streaming. *Multimedia Tools and Applications*, 77(3):3779–3798, 2018.

[33] 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*, pages 197–210, 2017.

[34] 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.

[35] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using fec for conversational multimedia communication. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 191–202, 2014.

[36] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *Usenix annual technical conference*, pages 417–429, 2015.

[37] 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.

[38] Devdeep Ray, Connor Smith, Teng Wei, David Chu, and Srinivasan Seshan. Sqp: Congestion control for low-latency interactive video streaming. *arXiv preprint arXiv:2207.11857*, 2022.

[39] Michael Rudow, Francis Y Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and KV Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 953–971, 2023.

[40] FLOYD Sally. Random early detection for congestion avoidance. *IEEE/ACM Transactions on Networking.*

[41] Zulfikar Hossain Sarker, Souvik De, Subhayan Nandy, and Anupam Roy. Evaluating the performance of congestion control algorithms for real-time video streaming. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE, 2019.

[42] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. Rtp: A transport protocol for real-time applications, 1996.

[43] Sea Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (ledbat). Technical report, 2012.

[44] A. Smith and B. Johnson. A study on peak signal-to-noise ratio (psnr) in image processing. *Journal of Image Processing.*

[45] Aaron Smith and Monica Anderson. Americans and digital knowledge. https://www.pewresearch.org/internet/2020/10/28/americans-and-digital-knowledge/, 2020.

[46] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions On Networking*, 28(4):1698–1711, 2020.

[47] Statista. Social media live streaming – statistics & facts. https://www.statista.com/topics/5471/social-media-live-streaming/, 2022.

[48] Sarah Stelzer. Zoom burnout is real: Here's how you can avoid it. https://www.techrepublic.com/article/zoom-burnout-is-real-heres-how-you-can-avoid-it/, 2020.

[49] TCP. "https://datatracker.ietf.org/doc/html/rfc793.

[50] 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.

[51] WebRTC. https://webrtc.org/.

[52] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Alexander Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *NSDI*, volume 20, pages 495–511, 2020.

[53] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[54] 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.

[55] 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.

[56] Yu Zhai, Xiaoxi Liu, Yu Gao, and Xiaoming Wang. Adaptive video congestion control for web real-time communication. *IEEE Access*, 8:230876–230885, 2020.

[57] Lixia Zhang, Scott Shenker, and Daivd D Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of the conference on Communications architecture & protocols*, pages 133–147, 1991.