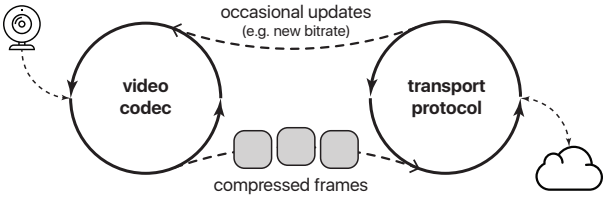
Salsify: low-latency network video through tighter integration between a video codec and a transport protocol

Paper #337 (12+2 pages)

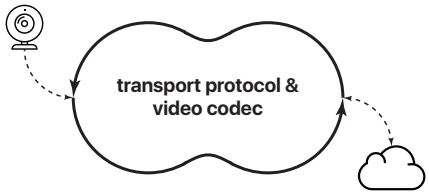
Abstract

We present Salsify, a system for real-time Internet video transmission that achieves 3.9× lower delay and 2.7 dB higher visual quality on average when compared with five existing systems: FaceTime, Hangouts, Skype, and WebRTC with and without scalable video coding.

Salsify achieves these gains through a joint design of the video codec and transport protocol that features a tighter integration between these components. The design includes three major improvements. First, Salsify's transport protocol is video-aware and accounts for the fact that video encoders send data in bursts rather than "full throttle." Second, Salsify's video codec exposes its internal state to the application, allowing it to be saved and restored. Salsify uses this to explore two compression levels for every frame, sending the frame that best matches the network conditions after compression. Third, Salsify combines the video codec's and transport protocol's control loops so that both components run in lockstep, and frames are encoded when the network can accommodate them. This improves responsiveness on variable network paths.



(a) Conventional systems



(b) Salsify

Figure 1: Conventional video applications execute the codec and transport in separate control loops running largely independently: the codec runs at a steady frame rate and quality level, with these parameters adjusted occasionally in response to transport-layer congestion feedback. Salsify explores a more tightly integrated design.

1 Introduction

Real-time video transmission has long been a popular Internet application—from the seminal schemes of the 1990s [22, 8] to today's widely used videoconferencing systems, such as FaceTime, Hangouts, Skype, and the browser-based WebRTC system [1]. Real-time video applications are used for person-to-person videoconferencing, cloud videogaming, tele-operation of robots and vehicles, and any setting where video must be encoded and sent with low latency over the network.

These systems combine two components, a transport protocol and a video codec, to transmit live video over unpredictable networks. The transport responds to congestion feedback and estimates the capacity, or bitrate, of the network path. It supplies these estimates to the codec, which compresses the input video at a quality level that tries to approximate the requested bitrate on average. The transport then sends the compressed video stream.

The architecture of these applications has been influenced by a desire to allow codecs and transports implemented by different parties to be composed together, for hardware codec designs to be standardized, and for applications to support pluggable codecs that can be upgraded over time. As a result, video applications typically execute the codec and transport in separate control loops, interfacing at arm's length: the codec runs at a steady frame rate and quality level, with these parameters adjusted occasionally in response to transport-layer congestion feedback. While compression methods have changed and improved markedly in recent years, the overall architecture of these programs has remained largely the same.

In this paper, we describe a system and a series of experiments that indicate that more radical changes to the architecture of Internet video systems can markedly improve performance. We present Salsify, a real-time video system with a redesigned transport protocol, video codec, and system architecture. These diverge from the conventional design:

1. Salsify's transport protocol is video-aware; its bandwidth-estimation scheme accounts for the fact that video encoders send data in bursts rather than

System	Video delay 95th %ile vs. Salsify-1c	Video quality SSIM vs. Salsify-1c
Salsify-1c	[449 ms]	[15.4 dB]
FaceTime	2.3×	−2.1 dB
Hangouts	4.2×	−4.2 dB
Skype	1.2×	−6.9 dB
WebRTC	10.5×	−2.0 dB
WebRTC (VP9-SVC)	7.9×	−1.3 dB

Figure 2: Performance of Salsify (single-core version) and other real-time Internet video systems over an emulated AT&T LTE network path. The testbed (§4) replayed the same network conditions and video input to each system, then measured the end-to-end delay and quality at the receiver. Full results are in Section 6.

"full throttle." This improves its link utilization and awareness of congestion.

2. Salsify's video codec is implemented in a functional-programming style, allowing its internal state to be saved and restored. Salsify uses this to match the codec's output to the network's capacity: instead of trying to predict the most appropriate coding parameters in advance, Salsify explores two compression levels for every frame (either serially or in parallel), transmits the compressed frame that best matches the network conditions, then resynchronizes its encoders with the state induced by the transmitted frame.
3. Salsify combines the codec's and transport protocol's control loops so that both components run in lockstep, and frames are encoded and sent when the network can accommodate them (Figure 1). This improves responsiveness on variable networks.

We built an end-to-end measurement testbed to compare Salsify's performance with other systems. The testbed plays a pre-recorded videoconference call through a USB device that simulates a UVC webcam into a sender computer that runs an unmodified version of the system-under-test (e.g. Skype, Facetime, etc.). The sending computer transmits to a separate receiving computer, over a network emulator whose trace of varying capacity is synchronized to the video. The testbed captures the receiving computer's display output, calculating the end-to-end video delay and quality relative to what was sent.

Salsify consistently outperformed conventional real-time video systems, in both quality and delay, by substantial margins. Figure 2 shows a preview of the results.

We also performed two user studies to estimate the relative importance of quality and delay on the subjective quality of experience (QoE) of real-time video systems. In the first study, participants engaged in a videoconference call with a partner; in the second study, participants drove a vehicle in a simulated environment by playing the "Driveclub" videogame on a PlayStation 4. We manipulated the quality and delay of the subject's video in both studies, and asked participants to rate their quality of experience on a 1-to-5 scale.

We found that users will accept large decreases in video quality in exchange for small improvements in delay. The trade-off between delay and quality depends on the application; videogame players were more sensitive to delay than participants in the videoconference. The results suggest that the benefits of Salsify's tighter integration between the codec and transport—considerably reduced end-to-end delay, with the same or better visual quality—can yield significant payoffs to QoE.

This paper proceeds as follows. Section 2 discusses background information on real-time video systems and related work. We describe the design and implementation of Salsify in Section 3 and of our end-to-end measurement testbed for black-box real-time video systems in Section 4. Section 5 describes the user studies, and Section 6 presents the results of the evaluation. We discuss the limitations of the system and analysis in Section 7.

Salsify is open-source software; we have published an anonymized version of the code and raw data from the evaluation at https://github.com/salsify-anon.

2 Related work

Adaptive videoconferencing. The earliest packet-video systems operated over cell-switched networks with reserved capacity at a particular bitrate. In that setting, the transport protocol and video codec could operate completely independently, because the bitrate did not change.

By contrast, Skype, FaceTime, and similar programs are intended for *adaptive* real-time videoconferencing over a best-effort network path with contending cross traffic. In the general Internet, network capacity is uncertain and variable. The transport needs to provide feedback to the video encoder so it can match its transmissions to the network's evolving capacity. Systems in this area include Skype, FaceTime, Hangouts, and the WebRTC system [1], whose open-source reference implementation has been incorporated into several Web browsers.

Metrics for codecs. Much effort has gone into the development of new video codecs, culminating in standards like H.265 [28] and VP9 [29]. These systems are typically evaluated on quality metrics as a function of average bitrate, metrics such as PSNR (the ratio of the peak value, e.g. 255, to the error introduced by lossy compression) or SSIM [30] ("structural similarity").

Metrics for transport protocols. Meanwhile, a parallel literature has studied transport protocols intended for real-time applications [5, 32, 13, 5, 14]. These systems are often evaluated on network-centric measurements (throughput, queueing delay of packets), sometimes with the assumption that a hypothetical application can exactly fill the congestion window to allow the transport protocol

to run "full throttle" [32]. In the case of video encoders that produce frames intermittently at a particular frame rate, with bitrate varying according to the scene complexity, this assumption has been criticized as unrealistic [12]. Salsify complements this work with a video-aware transport protocol that accounts for the intermittent nature of data generated by the video codec.

Scalable video coding. Several video formats support scalable encoding, where the encoder produces multiple streams of compressed video: a base layer, followed by one or more enhancement layers that improve the quality of the lower layers in terms of frame rate, resolution, or visual quality. Scalable coding is part of the H.262 (MPEG-2), MPEG-4 part 2, H.264 (MPEG-4 part 10 AVC) and VP9 systems. A real-time video application may use scalable video coding to improve performance over a variable network, because the application can discard enhancement layers immediately in the event of congestion, without waiting for the video codec to adapt. (Improvements in quality, however, must wait for a coded enhancement opportunity.) We benchmarked a contemporary SVC system, VP9-SVC as part of WebRTC in Google Chrome, in our measurements and found that it performed similarly to conventional WebRTC. Scalability is also useful in multiparty videoconferences (by allowing a relay node to accommodate different network capacities to each receiver, without transcoding the video), an application Salsify does not target.

Cross-layer schemes. Outside the context of digital packet video, SoftCast [15] and Apex [26] combine adaptive video coders and physical-layer protocols, sending wireless signals structured so that video quality degrades gracefully when there is more noise or interference on the wireless link. Salsify is also designed to degrade gracefully when the network deteriorates, but Salsify is not a cross-layer scheme in the same way and does not reach into the physical layer. Like Skype, FaceTime, and WebRTC, Salsify is an application that sends conventional UDP datagrams over the Internet.

Measurement of real-time video systems. Prior work has studied the performance of integrated videoconferencing applications. Zhang and colleagues [37] varied the characteristics of an emulated network path, and measured how Skype varied its video frame rate (as well as network-centric measurements such as throughput). Xu and colleagues [34] used Skype to film a stopwatch application on the receiver computer's display, producing two clocks side-by-side on the receiver's screen. By taking screenshots, the authors were able to measure Skype's video delay under different network conditions.

Salsify complements this literature with an end-to-end measurement of videoconferencing systems' video quality as well as delay. From the perspective of the sending computer, the testbed appears to be a USB webcam that captures a repeatable video clip. On the receiving computer, the HDMI display output is routed back to the testbed, so that the system can measure end-to-end video quality and delay between source and destination.

QoE-driven video transport. Recent work has focused on optimization approaches to delivery of adaptive video. Techniques include control-theoretic selection of pre-encoded video chunks for a Netflix-like application [36] and inferential approaches to selecting relays and routings [16, 10]. Generally speaking, these systems attempt to evaluate or maximize performance according to a function that maps various metrics into a single quality of experience (QoE) figure. Our evaluation includes two user studies to calibrate a QoE metric and find the relative impact of video delay and visual quality on quality of experience in real-time video applications (a videochat and a driving-simulation videogame).

Loss recovery. Existing video systems use a variety of techniques for recovering from packet loss. RTP's audio-visual profile [24] includes "slice loss indication" that a video coder can use to resend a missing segment of the image. WebRTC uses this and other methods [21]. By contrast, Salsify's functional video decoder retains old states in memory until the sender gives permission to evict them, so that if a frame is lost in whole or in part, the encoder can send new frames that depend only on older frames that the receiver has acknowledged, without having to re-encode missing slices *de novo*.

3 Design and Implementation

Real-time Internet video systems are built by combining two components: a transport protocol and a video codec. In existing systems, these components operate independently, occasionally communicating through a standardized interface. For example, in WebRTC, which has an open-source reference implementation, the video encoder reads frames off the camera at a particular frame rate and compresses them, aiming for a particular average bitrate. The transport protocol [13] updates the encoder's frame rate and target bitrate on a roughly one-second timescale. WebRTC's congestion response is generally reactive: if the video codec produces a compressed frame that overshoots the network's capacity, the transport will send it (even though it will cause packet loss or bloated buffers), but the WebRTC transport subsequently tells the codec to pause encoding new frames until congestion clears. Skype, FaceTime, and Hangouts work similarly.

Salsify's architecture is more closely coupled. Instead of allowing the video codec to free-run at a particular frame rate and target bitrate, Salsify fuses the video codec's and transport protocol's control loops into one. This architecture allows the transport protocol to communicate network conditions to the video codec before

each frame is compressed, so that Salsify's transmissions match the network's evolving capacity, and frames are encoded when the network can accommodate them.

Salsify achieves this by exploiting its codec's ability to save and restore its internal state. Salsify's transport estimates the number of bytes that the network can safely accept without dropping or excessively queueing frames. Even if this number is known before encoding begins for each frame, it is challenging to predict the encoder parameters (quality settings) that cause a video encoder to match a pre-specified frame length.

Instead, each time a frame is needed, Salsify tries encoding with two different sets of encoder parameters in order to bracket the available capacity. The system examines the encoded sizes of the resulting compressed frames and selects one to send, based on which more closely matches the network capacity estimate. The state induced by this frame is then restored and used as the basis for both versions of the next coded frame. We implemented two versions of Salsify: one that does the two encodings serially on one core (Salsify-1c), and one in parallel on two cores (Salsify-2c).

3.1 Salsify's functional video codec

Salsify's video codec is written in about 11,000 lines of C++ and encodes/decodes video in real-time according to Google's VP8 format [31]. It differs from previous implementations of VP8 and other codecs in one key aspect: it exposes the internal "state" of its encoder/decoder to the application in explicit state-passing style.[1] This state includes copies of previous decoded frames (known as references) and other information helpful to compression (e.g. probability tables). At a resolution of 1280×720, the internal state of a VP8 codec is about 4 MiB.

In typical implementations, whether hardware or software, this state is maintained internally by the encoder/decoder and is inaccessible to the application. Salsify's VP8 encoder and decoder, on the other hand, are pure functions in the functional-programming sense, and the internal state of an encoder or decoder can be saved and restored by the application.

This allows Salsify to (1) encode frames at a rate and quality that matches the network capacity and (2) efficiently recover from packet loss.

Encoding to match network capacity. Two compression levels are explored for each frame by running two encoders (serially or in parallel), initialized with the same internal state but with different quality settings. Salsify selects the resulting frame that best matches the network conditions, delaying the decision of which version to send until as late as possible. Since the encoder is implemented in explicit state-passing style, it can be resynchronized to the state induced by whichever version of the frame is chosen to be transmitted. Salsify chooses the two quality settings for the next frame based on surrounding (one slightly higher, one slightly lower) whichever settings were successful in the previous frame.[2]

There is also a third choice: not to send *either* version, if both exceed the estimated network capacity. In this case, the next frame will be encoded based on the same internal state. Salsify is therefore able to vary the frame cadence to accommodate the network, by skipping frames in a way that other video applications cannot (conventional applications can only pause frames on *input* to the encoder—they cannot skip a frame after it has been encoded without causing corruption).

Loss recovery. In the absence of loss, the encoder produces a sequence of compressed frames, each one depending on the state left behind by the previous frame (even if that frame has not yet been acknowledged as received). If a loss has recently occurred, the encoder may instead create a compressed frame that depends on an earlier state that the receiver has explicitly acknowledged. In doing so, the encoder knows that the new frame will be usable by the receiver if received, even if intermediate packets are lost. The receiver saves multiple states in memory, so the sender can make frames that depend on them until it receives an acknowledgment from the receiver that it has received a later frame. At this point, the sender instructs the receiver to discard the earlier states.

3.2 Salsify's transport protocol

We implemented Salsify's transport protocol in about 2,000 lines of C++. The sender transmits compressed video frames over UDP to the receiver, which replies with acknowledgments. Each video frame is divided into one or more MTU-sized fragments.

With the headers shown in Figure 3, a compressed video frame becomes an idempotent operator that acts on the identified source state at the receiver, transforming it into the identified target state, and producing a picture for display in the process. In reply to each fragment, the receiver sends an acknowledgment message that contains the frame and fragment numbers and a "catalog" of hashes of available decoder states. The sender is allowed to base a future frame on any of these states.

Salsify's transport protocol estimates the desired size of a frame—the congestion window—as follows. The receiver timestamps incoming UDP datagrams upon receipt, and estimates the current rate of the link with an exponentially weighted moving average of the packet inter-arrival

[1] We previously described a different application built on an earlier version of this encoder; citation withheld for anonymous submission.

[2] We arrived at the current 1-core and 2-core versions of Salsify after paring down an earlier 8-core version; after testing, we found that just two options (plus the option to not send a frame at all) were sufficient.

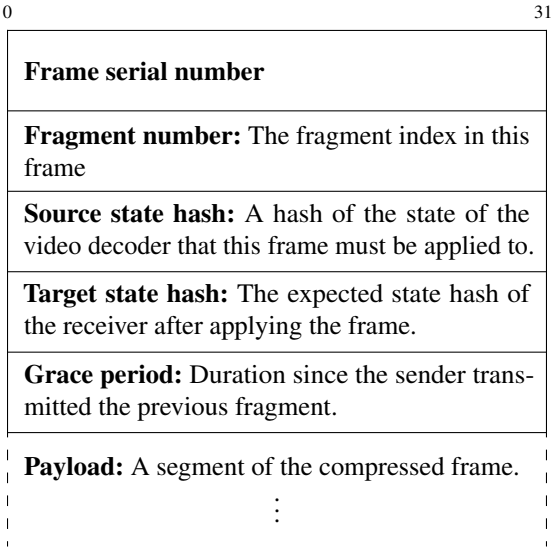


Figure 3: Contents of frame fragments sent by Salsify's transport protocol. The "grace period" field improves the network bandwidth estimate by ensuring that the gaps between frames are not mistaken for congestion on the network.

time, similar to WebRTC [13, 5]. This estimate is communicated to the sender in the ack packets.

To calculate the congestion window at time t_i, the sender first estimates an upper bound for the number of packets already in-flight, $N(t_i)$, by subtracting the indices of the last-sent packet and the last acknowledged packet. Let $\tau(t_i)$ be the average inter-arrival time reported by the receiver at t_i. If the sender aims to keep the end-to-end delay less than d in order to preserve interactivity, there can be no more than $\frac{d}{\tau(t_i)}$ packets in-flight. Therefore, the congestion window is $\frac{d}{\tau(t_i)} - N(t_i)$ MTU-size fragments.

The grace period. The sender does not transmit continuously—it pauses between frames. As a result, the inter-arrival time between the last fragment of one frame and the first fragment of the next frame is not as helpful an indicator of the network capacity (Figure 4). This pause could give the receiver an artificially pessimistic estimate of the network because the application is not transmitting "full throttle." To account for this, the sender includes a *grace period* in each fragment, which tells the receiver about the duration between when the current and previous fragments were sent. The receiver subtracts this value from the packet inter-arrival time before updating the moving average.

4 Measurement testbed

To evaluate Salsify, we built an end-to-end measurement testbed for real-time video systems that treats the sender and receiver as black boxes, emulating a time-varying network while measuring application-level video metrics

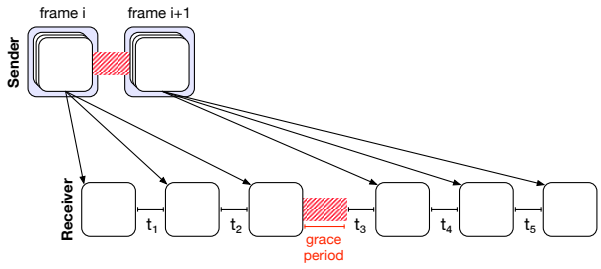


Figure 4: The receiver maintains a moving average of packet inter-arrival times, t_is. The sender includes the delay between sent packets as a "grace period," so the receiver can account for the sender's pauses between frames.

that affect quality of experience. This section describes the testbed's metrics and requirements (§4.1), its design (§4.2), and its implementation (§4.3).

4.1 Requirements and metrics

Requirements. The testbed needs to present itself as a webcam and supply a high-definition, 60 fps video clip in a repeatable fashion to unmodified real-time video systems. At the same time, the testbed needs to emulate a varying network link between the sender and receiver in the system, with the time-varying behavior of the emulated network synchronized to the test video. Finally, the testbed needs to capture frames coming out of the display of an unmodified receiver, and quantify their quality (relative to the source video) and delay.

Metrics. The measurement testbed uses two principal metrics for evaluating the video quality and video delay of a real-time video system. For quality, we use mean *structural similarity* (SSIM) [30], a standard measure that compares the received frame to the source video.

To measure interactive video delay, the testbed calculates the difference between the time that it supplies a frame (acting as a webcam) and when the receiver displays the same frame (less the testbed's inherent delay, which we measure in §6.1).

For frames on the 60 fps webcam input that weren't sent or weren't displayed by the receiver, we assign an arrival time equal to the *next* frame shown. As a result, the delay metric rewards systems that transmit with a higher frame rate. The goal of this metric is to account for both the frame rate chosen by a system, and the delay of the frames it chooses to transmit. A system that transmits one frame per hour, but those frames always arrive immediately, will still be measured as having delay of up to an hour, even though the rare frame that *does* get transmitted arrives quickly. A system that transmits at 60 frames per second, but on a one-hour tape delay, will also be represented as having a large delay.

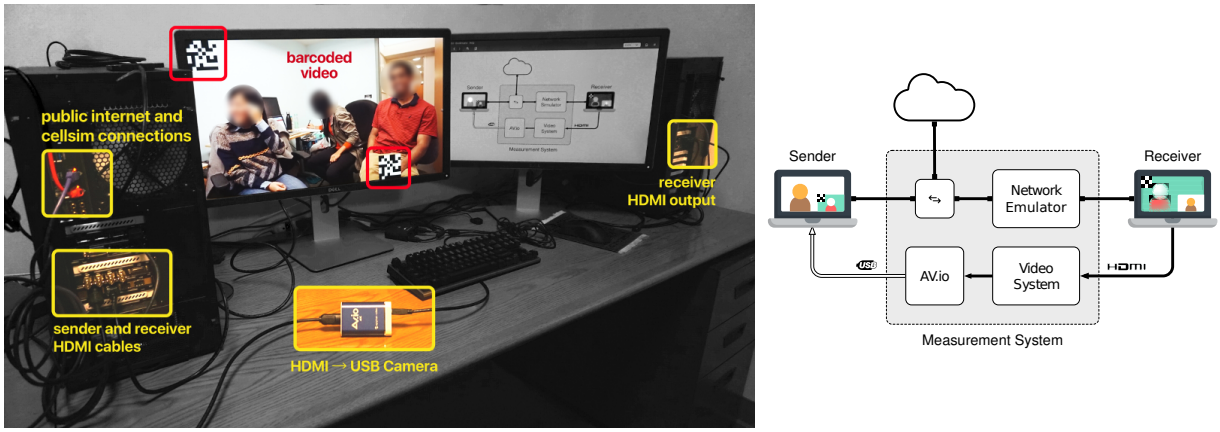


Figure 5: Testbed architecture. The system measures the performance of an unmodified real-time video system. It emulates a webcam to supply a barcoded video clip to the sender. The sender transmits frames to the receiver via an Ethernet connection. The measurement testbed interposes on the receiver's network connection and controls network conditions using a network emulator synchronized to the video. The receiver displays its output in a fullscreen window via HDMI, which the testbed captures. By matching barcodes on sent and received frames, the testbed measures the video's delay and quality, relative to the source. The measurement testbed timestamps outgoing and incoming frames with a dedicated hardware clock, eliminating the effect of scheduling jitter in measuring the timing of 60 fps video frames.

4.2 Design

Figure 5 outlines the testbed's hardware arrangement. At a high level, the testbed works by injecting video into a sending client, simulating network conditions between sender and receiver, and capturing the displayed video at the receiving client. It then matches up frames injected into the sender with frames captured from the receiver, and computes the delay and quality.

Hardware. The sender and receiver are two computers running an unaltered version of the real-time video application under test. Each endpoint's video interface to the testbed is a standard interface: For the sender, the testbed emulates a UVC webcam device. For the receiver, the testbed captures HDMI video output.

The measurement testbed also controls the network connection between the sender and receiver. Each endpoint has an Ethernet connection to the testbed, which bridges the endpoints to each other and to the Internet.

Video analysis. To compute video-related metrics, the testbed logs the times when the sending machine is presented with each frame, captures the display output from the receiver, and timestamps each arriving frame in hardware to the same clock.

The testbed matches each frame captured from the receiver to a frame injected at the sender. To do so, the testbed preprocesses the video to add two small barcodes, in the upper-left and lower-right of each frame.[3] Together, the barcodes consume 3.6% of the frame area. Each barcode encodes a 64-bit random number that is unique over the course of the video. An example frame is shown in figures 5 and 6. The quality and delay metrics are computed in postprocessing by matching the barcodes on sent and received frames, then comparing corresponding frames.

[3]The two barcodes were designed to detect "tearing" within a frame, when the receiver presents pieces of two different source frames at the same time. In our evaluations, we did not see this occur.

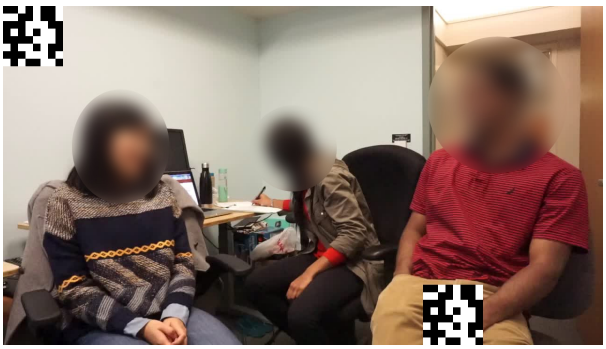
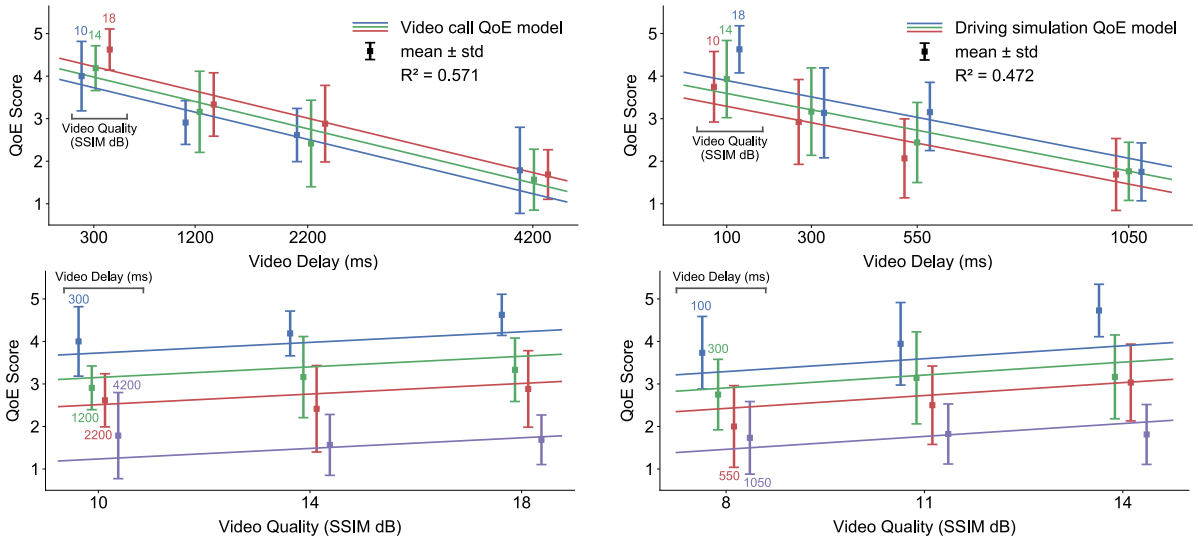


Figure 6: An example barcoded video frame sent by the measurement testbed (§4.2). The barcodes each represent a 64-bit random number that is unique over the course of the video.

4.3 Implementation

The measurement testbed is a PC workstation with specialized hardware. To capture and play raw video, the system uses a Blackmagic Design DeckLink 4K Extreme 12G card, which emits and captures HDMI video. The DeckLink timestamps incoming and outgoing frames with its own hardware clock. To convert outgoing video to the UVC webcam interface, the testbed uses an Epiphan AV.io HDMI-to-UVC converter. At a resolution of 1280×720 and 60 frames per second, raw video consumes 1.8 gigabits per second. The testbed uses two SSDs to simultaneously play back and capture raw video.



(a) Video Call **(b)** Driving Simulation

Figure 7: The results of the user studies (videoconferencing in the left column; driving simulator in the right column). The data from each study were fit to a two-dimensional linear model—one for videoconferencing, one for driving—using ordinary least squares. The upper plots project the learned bilinear models onto the delay-QoE axes; similarly, the lower plots show the quality-QoE projection. We found that for a given delay, the quality has only a small impact on the QoE (upper plots); conversely, for a given quality, the delay has a large impact on the QoE (lower plots). Furthermore, in the driving study, users rated delay as almost twice as important (relative to visual quality) as they did in the videochat study.

The measurement testbed computes SSIM using the Xiph Daala tools package. For network emulation, we use Cellsim [32], always starting the program synchronized to the beginning of an experiment. To explore the sensitivity to queueing behavior in the network, we configured Cellsim with a DropTail queue with a dropping threshold of 64, 256, or 1024 packets; ultimately we found most applications were not sensitive to this parameter and conducted remaining tests with a 256-packet buffer. The round-trip delay was set to 40 ms. We developed new software for barcoding, playing, capturing, and analyzing video. It comprises about 2,500 lines of C++.

Figure 8: Structural similarity (SSIM) is a measure of perceptual image quality. From left to right: original image, 15.6 dB, 9.1 dB.

5 User studies to calibrate QoE metrics

As part of the development of Salsify, we conducted two user studies to quantify the relative impact of video delay and video quality on quality of experience (QoE) in real-time video applications. These studies were approved by the Institutional Review Board at our institution. In both studies, we varied the delay and quality over ranges typically seen in real-world applications. Our results show that small variations in video delay greatly affect mean opinion score; video quality also affects mean opinion score but less dramatically. We also observed that the relative importance of these parameters depends on the application (i.e., some applications are more sensitive to video delay than others).

In our first user study, participants engaged in a simulated long-distance video conference call with a partner. As part of this study, we built a test jig that captured the audio and video of both participants and played it to their partner with a precisely controlled amount of added delay and visual quality degradation, achieved by encoding and then decoding the video with the x264 H.264 encoder at various quality settings. Participants conversed for one minute on each setting of delay and quality; after each one-minute interval, participants scored their subjective quality of experience on a scale from 1 (worst) to 5 (best). Twenty participants performed this user study and every participant experienced the same 12 video delay and video quality settings (SSIM dB $\times$ delay: $\{10\,\text{dB}, 14\,\text{dB}, 18\,\text{dB}\} \times \{300\text{ms}, 1200\text{ms}, 2200\text{ms}, 4200\text{ms}\}$).

The second user study put participants behind the wheel of a race car in a simulated environment: a PlaySta-

tion 4 playing the "Driveclub" videogame. Using a second test jig, the visual quality and the delay between the PlayStation's HDMI output and the participant's display were precisely controlled. Participants drove their simulated vehicle for 45 seconds on each quality and delay setting, then rated their quality of experience from 1 (worst) to 5 (best). Seventeen participants performed this user study and all participants experienced the same 12 video delay and video quality settings (SSIM dB $\times$ delay: $\{8\,\text{dB}, 11\,\text{dB}, 14\,\text{dB}\} \times$ {100ms, 300ms, 550ms, 1050ms}).

Results and interpretation. We used a two-dimensional linear equation as our QoE model; the model for each user study was fit using ordinary least squares. The resultant best-fit lines (one for the videochat, and one for the driving simulation) are shown in Figure 7. Using the learned coefficients from the videoconferencing study, we predict that a 100 ms decrease in video delay produces the same quality of experience improvement as a 1.0 dB increase in visual quality (SSIM dB). Likewise, in the driving simulation we predict that a 100 ms decrease in video delay is equivalent to a 1.9 dB increase in visual quality. This suggests that in settings such as tele-operation of vehicles, achieving low video delay is more critical than increasing video quality, even moreso than in person-to-person videoconferencing.

6 Evaluation of Salsify

This evaluation answers the question: how does Salsify compare with five popular real-time video systems in terms of video delay and video quality when running over a variety of real-world and synthetic network traces? In sum, we find that, among the systems tested, Salsify gave the best delay and quality by substantial margins over a range of cellular traces; Salsify also performed competitively on a synthetic "lossy link" trace that we generated.

6.1 Setup, calibration, and method

Setup. We ran all experiments using the measurement testbed described in Section 4. Figure 9 lists applications and versions. Tests on macOS used late-model MacBook Pro laptops running macOS Sierra. WebRTC (VP9-SVC) was run on Chrome with command line arguments to enable VP9 scalable video coding; the arguments were suggested by video-compression engineers on the Chrome team at Google.[4] Tests on Linux used Ubuntu 16.10 on desktop computers with recent Intel Xeon E3-1240v5 processors and 32 GiB of RAM. We tested Salsify using the same Linux machine.

All machines were physically located in the same room during experiments and were connected to each other and the public Internet through gigabit Ethernet connections. Care was taken to ensure that no other compute- or network-intensive processes were running on any of the client machines while experiments were being performed.

Calibration. To calibrate the measurement testbed, we ran a loopback experiment with no network: we connected the testbed's UVC output to the desktop computer described above, configured that computer to display incoming frames fullscreen on its own HDMI output using `ffplay`, and connected that output back to the testbed.

We found that the delay through the loopback connection was 4 frames, or about 67 ms; in all further experiments we subtracted this intrinsic delay from the raw results. The difference between the output and input images was negligible, with SSIM in excess of 25 dB, which corresponds to 99.7% absolute similarity.

Method. For each experiment below, we evaluate each system on the testbed using a specified network trace, computing metrics as described in Section 4.1. The stimulus is a ten minute, 60 fps, 1280×720 video of three people having a typical videoconference call. We preprocessed this video as described in Section 4.2, labeling each frame with a barcode.

6.2 Results

Experiment 1: variable cellular paths. In this experiment, we measured Salsify and the other systems using the AT&T LTE, T-Mobile UMTS ("3G"), and Verizon LTE cellular network traces distributed with the Mahimahi network-emulation tool [23]. The experiment's duration is 10 minutes. The cellular traces vary through a large range of network capacities over time: from more than 20 Mbps to less than 100 kbps. The AT&T LTE and T-Mobile traces were held out and not used in Salsify's development, although an earlier (8-core) version of Salsify was previously evaluated on these traces before we developed the current 1-core and 2-core versions.

Figures 10a, 10b and 10c show the results for each scheme on each trace. Both the single-core (Salsify-1c) and dual-core (Salsify-2c) versions of Salsify outperform all of the competing schemes on both quality and delay (and therefore, on either QoE model from the user study).

Somewhat contrary to our expectations, we saw little difference in performance between the serial and parallel versions of Salsify; this suggests that having the two video encoders run *in parallel* is not very important, at least if the codec is sufficiently fast to keep up with the application's idea of when frames ought to be sent. To the extent there is a difference, the single-core version sometimes

[4]The arguments were: out/Release/chrome --enable-webrtc-vp9-svc-2sl-2tl --fake-variations-channel=canary --variations-server-url=https://clients4.google.com/chrome-variations/seed .

Application	Platform	Version	Configuration change
Skype	macOS	7.42	Turned off Skype logo on the receiver.
FaceTime	macOS	3.0	Blacked out self view in post-processing.
Hangouts	Chrome (Linux)	55.0 (Chrome)	Edited CSS to hide self view.
WebRTC	Chrome (Linux)	62.0 (Chrome), https://appr.tc	Edited CSS to hide self view.
WebRTC (VP9-SVC)	Chrome (Linux)	62.0 (Chrome), https://appr.tc	Edited CSS to hide self view.

Figure 9: Applications versions tested. For each application, we slightly modified the receiver to eliminate extraneous display elements that would have interfered with SSIM calculations. For WebRTC (VP9-SVC), we passed command line arguments to Chrome to enable the scalable video codec.

performs better than the dual-core version. This is not as much of a paradox as it may seem: the delay metric cares only about the total time between when an event is captured by the webcam and when it is first displayed by the receiver. A scheme with frames arriving slightly less often can score as well as a scheme with a higher frame rate, if the second scheme also has higher end-to-end delay.

Experiment 2: intermittent link. In this experiment, we evaluated Salsify's method of loss resilience. We evaluated each system on an intermittently lossy 12 Mbps link. The link's capacity is 12 Mbps until a "failure" arrives, which happens on average every five seconds; after failure, a "restore" event arrives on average 0.2 seconds later. The experiment's duration is 10 minutes.

Figure 10d shows the results for each scheme. The Salsify schemes had the best quality, and their delay was better than all schemes except Skype and WebRTC. Salsify and WebRTC are both on the Pareto frontier of this scenario; further tuning will be required to see if Salsify can improve its delay without compromising quality.

Experiment 3: component analysis study. In this experiment, we removed the new components implemented in Salsify one-by-one to better understand their contribution to the total performance of the system. First, we removed the feature of Salsify's transport protocol that makes it video-aware: the "grace period" to account for intermittent transmissions from the video codec. The performance degradation of this configuration is shown in Figure 10a as the "Salsify (no grace period) dot"; without this component, Salsify underestimates the network capacity and sends low quality, low-bitrate video.

Second, we then removed Salsify's explicit state-passing style video codec, replacing it with a conventional codec where the state is opaque to the application, and the appropriate encoding parameters must be predicted upfront (instead of choosing the best compressed version of each frame after-the-fact). The codec predicted these parameters by performing a binary search for the quality setting on a decimated version of the original frame, attempting to hit a target bitrate and extrapolating the resulting size to the full frame. The result of this experiment is also in Figure 10a, as "Salsify (conventional codec)."

As shown in the plot, Salsify's performance is again substantially reduced. This is result of two factors: (1) The transport no longer has access to a menu of frames at transmission time; if the capacity estimate changed during the time it took to compress the video frame, Salsify will either incur delay or have missed an opportunity to improve the quality of the video. (2) It is challenging for any codec to choose the appropriate quality settings upfront to meet a target size; the encoder will be liable to under- or overshoot its target.

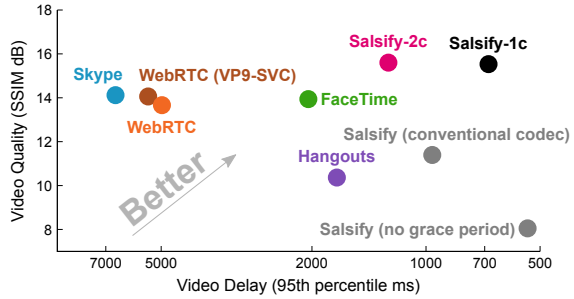
Experiment 4: capacity ramp and brief dropout. In this experiment, we evaluated how Salsify, Skype, and WebRTC handle a network with a gradual decrease in bandwidth (to zero), a brief dropout, and then a gradual resumption of network capacity. We created the synthetic network trace depicted in light gray in Figures 11a and 11b. The experiment's duration is 20 seconds.

Figure 11a shows the amount of throughput each scheme tries to send through the link, versus time. Salsify's throughput smoothly decreases alongside link capacity, then gracefully recovers after the dropout. The result is that Salsify does not build up significant queueing delay when link capacity is degraded, as shown in Figure 11b.

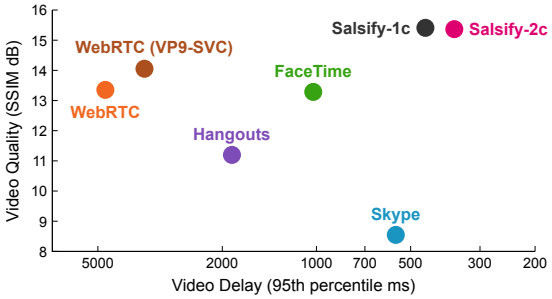
In contrast, Skype reacts slowly to degraded network capacity, and as a result induces loss on the link and builds up a standing queue that takes several seconds to dissipate. WebRTC reacts to the loss of link capacity, but ends up stuck in a bad mode after the network is restored; the receiver displays only three frames (marked with blue dots) in the eight seconds after the link begins to recover.

Experiment 5: sensitivity to queueing parameters. In this experiment, we quantified the performance impact of network buffer size on Salsify, Hangouts, WebRTC, and WebRTC (VP9-SVC) for a Verizon-LTE network trace. The plots in Figure 12 show the performance of each system on the trace across various DropTail thresholds (at 64, 256, and 1024 MTU-sized packets).

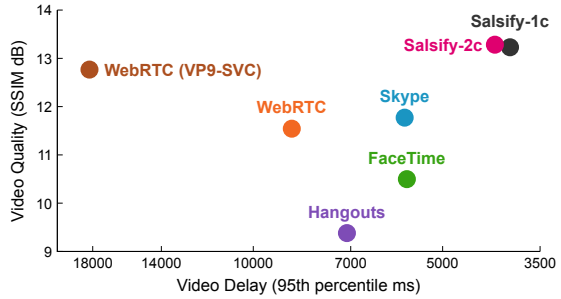
The performance of the tested systems was not significantly influenced by the choice of buffer size, perhaps be-



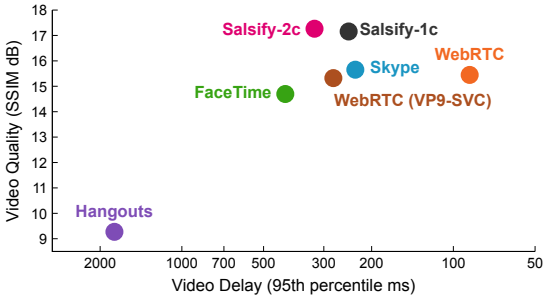
(a) Verizon LTE Trace



(b) AT&T LTE Trace



(c) T-Mobile UMTS Trace

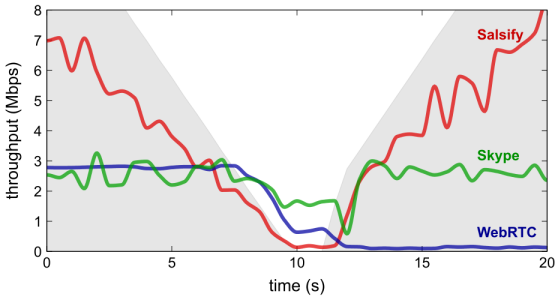


(d) Lossy link

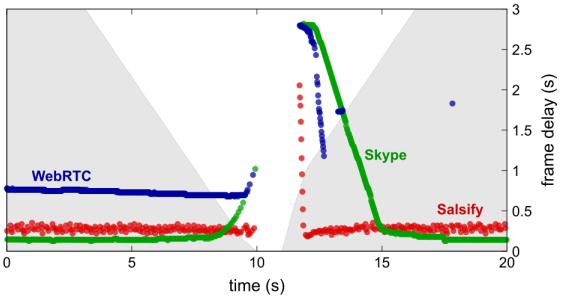
		Video Quality (SSIM dB)		Video Delay (ms)	
System	**Trace**	p25	mean	mean	p95
Salsify-1c	Verizon LTE	15.1	15.5	517.4	684.0
Salsify-2c	Verizon LTE	15.1	15.6	496.7	1257.0
FaceTime	Verizon LTE	15.0*	13.9	658.6	2044.0
Hangouts	Verizon LTE	10.0	10.7	598.1	2254.0
WebRTC	Verizon LTE	13.2	13.7	973.0	4977.0
WebRTC (VP9-SVC)	Verizon LTE	13.6	14.1	1196.1	5412.0
Skype	Verizon LTE	15.1*	14.1	1182.6	6600.0
Salsify-1c	AT&T LTE	15.0	15.4	349.1	449.0
Salsify-2c	AT&T LTE	15.0	15.4	282.1	362.0
FaceTime	AT&T LTE	12.6	13.3	469.4	1024.0
Hangouts	AT&T LTE	10.7	11.2	846.4	1862.0
WebRTC	AT&T LTE	12.4	13.4	934.7	4730.0
WebRTC (VP9-SVC)	AT&T LTE	13.5	14.1	775.3	3547.0
Skype	AT&T LTE	8.2	8.5	322.1	557.0
Salsify-1c	T-Mobile UMTS	13.0	13.2	840.1	3907.0
Salsify-2c	T-Mobile UMTS	12.9	13.3	803.3	4129.0
FaceTime	T-Mobile UMTS	8.8	10.5	1206.8	5700.0
Hangouts	T-Mobile UMTS	8.5	9.4	1012.0	7097.0
WebRTC	T-Mobile UMTS	10.5	11.3	1643.6	9863.0
WebRTC (VP9-SVC)	T-Mobile UMTS	12.1	12.8	2585.2	18215.0
Skype	T-Mobile UMTS	11.1	11.8	1451.8	5746.0
Salsify-1c	Lossy Link	16.2	17.3	258.3	325.0
Salsify-2c	Lossy Link	16.3	17.2	180.3	243.0
FaceTime	Lossy Link	14.6	14.7	280.2	416.0
Hangouts	Lossy Link	9.1	9.3	437.0	1771.0
WebRTC	Lossy Link	15.3	15.4	70.0	87.0
WebRTC (VP9-SVC)	Lossy Link	15.1	15.3	152.0	277.0
Skype	Lossy Link	15.5	15.7	128.4	230.0

(e) Summary table

Figure 10: The above figures show the end-to-end video quality and video delay over four emulated time-varying networks. Salsify-1c and Salsify-2c simultaneously achieve better video quality (both on average and the "worse" tail at 25th percentile) and better video delay (both on average and the worse tail at 95th percentile) than other systems for the three real-world network traces (AT&T, T-Mobile, Verizon). Salsify-1c and Salsify-2c perform competitively on the artificially generated "lossy link" network, which occasionally drops packets but is otherwise a constant 12Mbps (§6). The AT&T, T-Mobile, and lossy link traces were held-out during development. The best results on each metric are highlighted. Two entries marked with a * have a 25th-percentile SSIM that is higher than their mean SSIM; this indicates a skewed distribution of video quality.

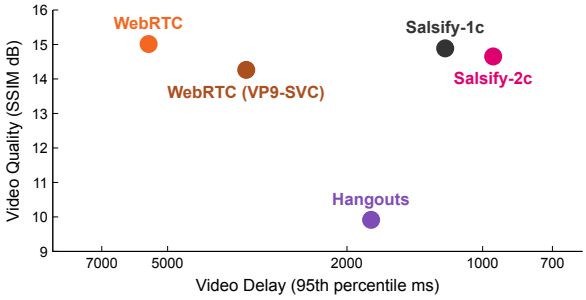


(a) Throughput

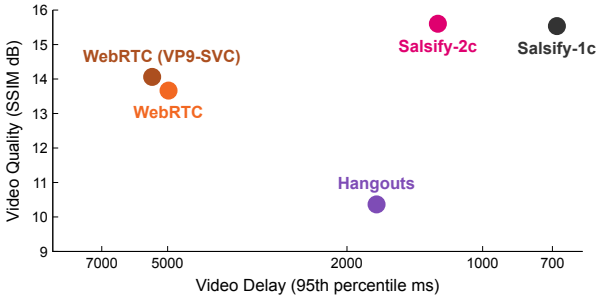


(b) Frame delay

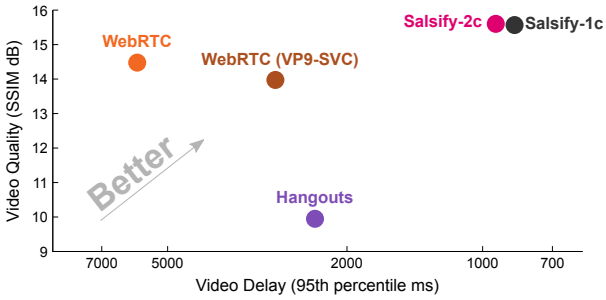
Figure 11: Salsify's fusion of the video codec's and transport protocol's control loops allows it to react more quickly to changes in network conditions than other video systems. This is illustrated by comparing the performance of Skype, WebRTC, and Salsify while communicating over a network path whose capacity decreases gradually to zero, then back up again (instanteous network capacity shown in gray).



(a) DropTail threshold = 64 packets.



(b) DropTail threshold = 256 packets.



(c) DropTail threshold = 1024 packets.

Figure 12: Sensitivity to queueing behavior. We measured performance over the Verizon LTE network trace (the same one used in Figure 10a) with different in-network buffer sizes. The tested systems were not found to be sensitive to this parameter. For all other experiments, we used a buffer size of 256 packets.

cause all schemes are striving for low delay and therefore are unlikely to build up a large-enough standing queue to see DropTail-induced packet losses. We ran the remaining tests using the middle setting (256 packets).

6.3 Modifications to systems under test

Although the testbed was designed to work with unmodified real-time video systems as long as the sender accepts input on a USB webcam and the receiver displays output fullscreen via HDMI, in practice we found that to evaluate the commercial systems fairly, small modifications were needed. We describe these here.

FaceTime two-way video and self view. Unlike the other video conferencing programs we tested, FaceTime could not be configured to disable bidirectional video transmission. We physically covered the webcam of the receiver when evaluating FaceTime in order to minimize the amount of data the receiver needed to send.

Also, like most video conferencing programs, FaceTime provides a self view window so users can see the video they are sending. This self view window cannot be disabled in FaceTime and is present in the frames captured by our measurement testbed. To prevent this from unfairly lowering FaceTime's SSIM score, we blacked out the self view window in post-processing (in both the sent and received raw videos) before computing SSIM. These regions accounted for approximately 4% of the total frame area.

Hangouts & WebRTC watermarks and self view. By default, Google Hangouts and our reference WebRTC client (appr.tc) had several watermarks and self view windows. Since these program run in the browser, we modified their CSS files to remove these artifacts so we could accurately measure SSIM.

Hangouts did not make P2P connections. Unlike all the other systems we evaluated, Google Hangouts did not make a direct connection between the two client machines. Rather, the clients communicated through a relay server.

We measured this delay by pinging the Google server used by the client machines. The round trip delay was < 20 ms in all cases and $\sim$5 ms on average. Given the delays we measured for Google Hangouts, this additional source of delay accounts for less than 1% of the total delays.

7 Limitations and Future Work

Salsify and its analysis feature a number of important limitations and opportunities for future work.

No audio. Salsify does not encode or transmit audio. When testing other video conferencing applications, we disabled the microphone on both the sender and receiver clients to avoid giving Salsify an unfair advantage. However, it is likely that other systems were designed with receiver-side buffering to smooth out jitter and avoid underflows in the audio playback. This could cause extra delay that Salsify would also incur if it wanted to provide smooth audio playback.

Unidirectional video. Our experiments used a dedicated sender and receiver, whereas a typical video call has bidirectional video. This is because the testbed only has one Blackmagic card (and pair of high-speed SSDs) and cannot send and capture two independent raw-video streams simultaneously.

Bidirectional video, like audio, requires the sender to reserve network capacity for a control channel. We turned off receiver side video when testing each application in order to avoid disadvantaging other video conferencing systems in our benchmarks.

No end-to-end QoE analysis of actual schemes. Because of a limited availability of research subjects, we performed our QoE studies on synthetically created delays and quality degradations, added to a videochat and driving-simulator videogame. We have not conducted an actual user study of Salsify itself. In informal testing, our subjective evaluation was consistent with the objective measurements reported.

Most codecs do not support save/restore of state. Salsify includes a VP8 codec—an existing format that we did not modify—with the addition that the codec is in functional style and allows the application to save and restore its internal state. Conventional codecs, whether hardware or software, do not support this interface, although we are hopeful these results will encourage implementers to expose such an interface. We discuss the consequences of this limitation below, in the conclusion.

8 Conclusion

In this paper, we presented Salsify, a new design for the architecture of real-time Internet video systems. Salsify improves upon existing systems in three principal ways: (1) a video-aware transport protocol achieves accurate estimates of network capacity without a "full throttle" source or sender-side buffering, (2) a functional video codec allows the application to experiment with multiple settings for each frame to find the best match to the network's evolving capacity, and (3) Salsify fuses the video codec's and transport protocol's control loops so that both components run in lockstep, and frames are encoded and sent when the network can accommodate them, rather than at a fixed rate.

In an end-to-end evaluation, Salsify (both single-core and dual-core versions) achieved lower end-to-end video delay and higher video quality, by substantial margins, when compared with five existing systems: Skype, FaceTime, Hangouts, and WebRTC with and without scalable video coding (VP9-SVC).

It is somewhat notable that Salsify achieves superior visual quality than other systems, as Salsify uses our own implementation of a VP8 codec, which is less sophisticated than schemes used by other systems. By contrast, Chrome's WebRTC uses VP9 implemented by the format's creators, and we believe Skype and FaceTime use commercial implementations of H.264; each of these codecs has undergone hundreds of thousands or millions of person-hours of development and optimization. The results suggest that further improvements to video *codecs* may have reached the point of diminishing returns in this setting, but radical changes to the architecture of real-time video *systems* can still yield significant improvements in performance.

Salsify's video codec was implemented in software, which makes it ill-suited for power-constrained environments such as mobile phones or battery-powered sensors, which typically rely on ASICs for video compression and decompression. For a practical implementation of Salsify on such devices, a hardware codec is probably necessary. We are not aware of any fundamental reason that a hardware codec could not also expose its internal state to be saved and later restored—the only requirement that Salsify's architecture imposes on the codec. We hope that our measurements motivate codec implementers, both hardware and software, to expose state in this fashion, so that future real-time video systems may benefit.

References

[1] ALVESTRAND, H. T. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft draft-ietf-rtcweb-overview-16, Internet Engineering Task Force, Nov. 2016. Work in Progress.

[2] CHEN, M., PONEC, M., SENGUPTA, S., LI, J., AND CHOU, P. A. Utility maximization in peer-to-peer systems. In *ACM SIGMETRICS* (June 2008).

[3] CHEN, X., CHEN, M., LI, B., ZHAO, Y., WU, Y., AND LI, J. Celerity: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications 31*, 9 (Sept. 2013), 155–164.

[4] CHENG, R., WU, W., CHEN, Y., AND LOU, Y. A cloud-based transcoding framework for real-time mobile video conferencing system. In *IEEE MobileCloud* (Apr. 2014).

[5] CICCO, L. D., CARLUCCI, G., AND MASCOLO, S. Experimental investigation of the Google congestion control for real-time flows. In *ACM FhMN* (Aug. 2013).

[6] ELMOKASHFI, A., MYAKOTNYKH, E., EVANG, J. M., KVALBEIN, A., AND CICIC, T. Geography matters: Building an efficient transport network for a better video conferencing experience. In *CoNEXT* (Dec. 2013).

[7] FENG, Y., LI, B., AND LI, B. Airlift: Video conferencing as a cloud service. In *IEEE ICNP* (Feb. 2012).

[8] FREDERICK, R. Experiences with real-time software video compression. In *Proceedings of the Sixth International Workshop on Packet Video* (1994).

[9] FUND, F., WANG, C., LIU, Y., KORAKIS, T., ZINK, M., AND PANWAR, S. S. Performance of DASH and WebRTC video services for mobile users. In *IEEE PV* (Dec. 2013).

[10] GANJAM, A., JIANG, J., LIU, X., SEKAR, V., SIDDIQUI, F., STOICA, I., ZHAN, J., AND ZHANG, H. C3: Internet-scale control plane for video quality optimization. In *NSDI* (May 2015).

[11] HAJIESMAILI, M. H., MAK, L., WANG, Z., WU, C., CHEN, M., AND KHONSARI, A. Cost-effective low-delay cloud video conferencing. In *IEEE ICDCS* (June 2015).

[12] HERMANNS, N., HAMM, L., AND SARKER, Z. A framework and evaluation of rate adaptive video telephony in 4g lte. In *WTC 2014; World Telecommunications Congress 2014; Proceedings of* (2014), VDE, pp. 1–6.

[13] HOLMER, S., LUNDIN, H., CARLUCCI, G., CICCO, L. D., AND MASCOLO, S. A Google congestion control algorithm for real-time communication, 2015. draft-alvestrand-rmcat-congestion-03.

[14] JAIN, M., AND DOVROLIS, C. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2002), SIGCOMM '02, ACM, pp. 295–308.

[15] JAKUBCZAK, S., AND KATABI, D. A cross-layer design for scalable mobile video. In *MobiComm* (Sept. 2011).

[16] JIANG, J., DAS, R., ANANTHANARAYANAN, G., CHOU, P. A., PADMANABHAN, V. N., SEKAR, V., DOMINIQUE, E., GOLISZEWSKI, M., KUKOLECA, D., VAFIN, R., AND ZHANG, H. VIA: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM* (Aug. 2016).

[17] KESHAV, S. A control-theoretic approach to flow control. In *Proceedings of the Conference on Communications Architecture &Amp; Protocols* (New York, NY, USA, 1991), SIGCOMM '91, ACM, pp. 3–15.

[18] LI, J., CHOU, P. A., AND ZHANG, C. Mutualcast: An efficient mechanism for content distribution in a peer-to-peer (P2P) network. Tech. Rep. MSR-TR-2004-98, Microsoft Research, 2004.

[19] LIANG, C., ZHAO, M., AND LIU, Y. Optimal bandwidth sharing in multiswarm multiparty P2P video-conferencing systems. *IEEE/ACM Trans. Networking 19*, 6 (Dec. 2011), 1704–1716.

[20] LIU, X., DOBRIAN, F., MILNER, H., JIANG, J., SEKAR, V., STOICA, I., AND ZHANG, H. A case for a coordinated internet video control plane. In *SIGCOMM* (Aug. 2012).

[21] LUMIAHO, L., AND NAGY, M., Oct. 2015. Error Resilience Mechanisms for WebRTC Video Communications http://www.callstats.io/2015/10/30/error-resilience-mechanisms-webrtc-video/.

[22] MCCANNE, S., AND JACOBSON, V. Vic: A flexible framework for packet video. In *Proceedings of the Third ACM International Conference on Multimedia* (New York, NY, USA, 1995), MULTIMEDIA '95, ACM, pp. 511–522.

[23] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for http. In *USENIX Annual Technical Conference* (2015), pp. 417–429.

[24] OTT, J., AND WENGER, D. S. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). RFC 4585, July 2006.

[25] PONEC, M., SENGUPTA, S., CHIN, M., LI, J., AND CHOU, P. A. Multi-rate peer-to-peer video conferencing: A distributed approach using scalable coding. In *IEEE ICME* (June 2009).

[26] SEN, S., GILANI, S., SRINATH, S., SCHMITT, S., AND BANERJEE, S. Design and implementation of an "approximate" communication system for wireless media applications. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 15–26.

[27] SEUNG, Y., LENG, Q., DONG, W., QIU, L., AND ZHANG, Y. Randomized routing in multi-party internet video conferencing. In *IEEE IPCCC* (Dec. 2014).

[28] SULLIVAN, G. J., OHM, J.-R., HAN, W.-J., AND WIEGAND, T. Overview of the high efficiency video coding (hevc) standard. *IEEE Trans. Cir. and Sys. for Video Technol. 22*, 12 (Dec. 2012), 1649–1668.

[29] *VP9 Bitstream & Decoding Process Specification Version 0.6*, March 2016. http://www.webmproject.org/vp9/.

[30] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing 13*, 4 (2004), 600–612.

[31] WILKINS, P., XU, Y., QUILLIO, L., BANKOSKI, J., SALONEN, J., AND KOLESZAR, J. VP8 Data Format and Decoding Guide. RFC 6386, Oct. 2015.

[32] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI* (Apr. 2013).

[33] WU, Y., WU, C., LI, B., AND LAU, F. C. M. vSkyConf: Cloud-assisted multi-party mobile video conferencing. In *ACM MCC* (Aug. 2013).

[34] XU, Y., YU, C., LI, J., AND LIU, Y. Video telephony for end-consumers: Measurement study of Google+, iChat, and Skype. In *IMC* (Nov. 2012).

[35] YAP, K.-K., HUANG, T.-Y., YIAKOUMIS, Y., MCKEOWN, N., AND KATTI, S. Late-binding: how to lose fewer packets during handoff. In *Proceeding of the 2013 workshop on Cellular networks: operations, challenges, and future design* (2013), ACM, pp. 1–6.

[36] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *SIGCOMM* (Aug. 2015).

[37] ZHANG, X., XU, Y., HU, H., LIU, Y., GUO, Z., AND WANG, Y. Modeling and analysis of Skype video calls: Rate control and video quality. *IEEE Trans. Multimedia 15*, 6 (Oct. 2013), 1446–1457.