Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol

Sadjad Fouladi, John Emmons, and Emre Orbay, *Stanford University;*
Catherine Wu, *Saratoga High School;*
Riad S. Wahby and Keith Winstein, *Stanford University*

https://www.usenix.org/conference/nsdi18/presentation/fouladi

Salsify: Low-Latency Network Video Through Tighter Integration Between a Video Codec and a Transport Protocol

Sadjad Fouladi°   John Emmons°   Emre Orbay°
Catherine Wu+   Riad S. Wahby°   Keith Winstein°

°*Stanford University,* +*Saratoga High School*

Abstract

Salsify is a new architecture for real-time Internet video that tightly integrates a video codec and a network transport protocol, allowing it to respond quickly to changing network conditions and avoid provoking packet drops and queueing delays. To do this, Salsify optimizes the compressed length and transmission time of each frame, based on a current estimate of the network's capacity; in contrast, existing systems generally control longer-term metrics like frame rate or bit rate. Salsify's per-frame optimization strategy relies on a purely functional video codec, which Salsify uses to explore alternative encodings of each frame at different quality levels.

We developed a testbed for evaluating real-time video systems end-to-end with reproducible video content and network conditions. Salsify achieves lower video delay and, over variable network paths, higher visual quality than five existing systems: FaceTime, Hangouts, Skype, and WebRTC's reference implementation with and without scalable video coding.

1 Introduction

Real-time video has long been a popular Internet application—from the seminal schemes of the 1990s [26, 10] to today's widely used videoconferencing systems, such as FaceTime, Hangouts, Skype, and WebRTC. These applications are used for person-to-person videoconferencing, cloud video-gaming, teleoperation of robots and vehicles, and any setting where video must be encoded and sent with low latency over the network.

Today's systems generally combine two components: a transport protocol and a video codec. The transport sends compressed video to the receiver, processes acknowledgments and congestion signals, and estimates the average data rate of the network path. It supplies this estimate to the codec, a distinct module with its own internal control loop. The codec selects encoding parameters (a frame rate and quality setting) and generates a compressed video stream with an average bit rate that approximates the estimated network capacity.

In this paper, we explore and evaluate a different design for real-time Internet video, based on a video codec that

System	Video delay 95th %ile vs. Salsify-1c (lower is better)	Video quality SSIM vs. Salsify-1c (higher is better)
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 1: Performance of Salsify (single-core version) and other real-time Internet video systems over an emulated AT&T LTE network path. Full results are in Section 5.

is integrated tightly into the rest of the application. This system, known as Salsify, combines the transport protocol's packet-by-packet *congestion control* with the video codec's frame-by-frame *rate control* into one algorithm. This allows Salsify to avoid provoking in-network buffer overflows or queueing delays, by matching its video transmissions to the network's varying capacity.

Salsify's video codec is implemented in a purely functional style, which lets the application explore alternative encodings of each video frame, at different quality levels, to find one whose compressed length fits within the network's instantaneous capacity. Salsify eschews an explicit bit rate or frame rate; it sends video frames when it thinks the network can accommodate them.

No individual component of Salsify is exactly new or was co-designed expressly to be part of the larger system. The compressed video format (VP8 [36]) was finalized in 2008 and has been superseded by more efficient formats in commercial videoconferencing programs (e.g., VP9 [14] and H.265 [32]). Salsify's purely functional implementation of a VP8 codec has been previously described [9], its loss-recovery strategy is related to Mosh [38, 37], and its rate-control scheme is based on Sprout [39].

Nonetheless, as a concrete system that integrates these components in a way that responds quickly to network variation without provoking packet loss or queueing delay, Salsify outperforms the commercial state of the art—Skype, FaceTime, Hangouts, and the WebRTC implementation in Google Chrome, with or without scalable video coding—in terms of end-to-end video quality and delay (Figure 1 gives a preview of results).

These results suggest that improvements to video *codecs* may have reached the point of diminishing returns in this setting, but changes to the architecture of video *systems* can still yield significant benefit. In separating the codec and transport protocol at arm's length—each with its own rate-control algorithm working separately—today's applications reflect current engineering practice in which codecs are treated largely as black boxes, at the cost of a significant performance penalty. Salsify demonstrates a way to integrate these components more tightly, while preserving an abstraction boundary between them.

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 measurement testbed for black-box video systems in Section 4. Section 5 presents the results of the evaluation. We discuss the limitations of the system and its evaluation in Section 6.

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. These results are described more fully in Appendix A.

Salsify is open-source software, and the experiments reported in this paper are intended to be reproducible. The source code and raw data from the evaluation are available at https://snr.stanford.edu/salsify.

2 Related work

Adaptive videoconferencing. Skype and similar programs perform adaptive real-time videoconferencing over an Internet path, by sending user datagrams (UDP) that contain compressed video. In addition to Skype, such systems include FaceTime, Hangouts, and the WebRTC system, currently in development to become an Internet standard [1]. WebRTC's reference implementation [35] has been incorporated into major Web browsers.

These systems generally include a video codec and transport protocol as independent subsystems, each with its own rate-control logic and control loop. The transport provides the codec with estimates of the network's data rate, and the video encoder selects parameters (including a frame rate and bit rate) to match its average bit rate to the network's data rate. Salsify, by contrast, merges the rate-control algorithm of each component into one, leveraging the functional nature of the video codec to keep the length of each compressed frame within the transport's instantaneous estimate of the network capacity.

Joint source-channel video coding. The IEEE multimedia communities have extensively studied low-latency real-time video transmission over digital packet networks (a survey is available in Zhai & Katsaggelos [45]). The bulk of this work targets heavily-multiplexed networks, where data rates are treated as fixed or slowly varying,

and packet loss and queueing delay can be modeled as random processes independent of the application's own behavior (e.g., [5]). In this context, prior work has focused on combining *source coding* (video compression) with *channel coding* (forward error correction) in order for the application to gracefully survive packet drops and delays caused by independent random processes [45].

Salsify is aimed at a different regime, more typical of today's Internet access networks, where packet drops and queueing delays are influenced by how much data the application chooses to send [39, 13], the bottleneck data rate can decay quickly, and forward-error-correction schemes are less effective in the face of bursty packet losses [33]. Salsify's main contribution is not in combining video coding and error-correction coding to weather packet drops that occur independently; it is in merging the rate-control algorithms in the video codec and transport protocol to avoid provoking packet drops (e.g., by overflowing router buffers) and queueing delay with its own traffic.

Cross-layer schemes. Schemes like SoftCast [19] and Apex [30] reach into the physical layer, by sending analog wireless signals structured so that video quality degrades gracefully when there is more noise or interference on the wireless link. Much of the IEEE literature [45] also concerns regimes where modulation modes and power levels are under the application's control. Salsify is also designed to degrade gracefully when the network deteriorates, but Salsify isn't a cross-layer scheme in the same way—it does not reach into the physical layer. Like Skype, FaceTime, etc., Salsify sends conventional UDP datagrams over the Internet.

Low-latency transport protocols. Prior work has designed several transport protocols and capacity-estimation schemes for real-time applications [21, 18, 39, 17, 6]. These schemes are often evaluated with the assumption that the application always has data available to the transport, allowing it to run "full throttle"; e.g., Sprout's evaluation made this assumption [39]. In the case of video encoders that produce frames intermittently at a particular frame rate and bit rate, this assumption has been criticized as unrealistic [16]. Salsify's transport protocol is based on Sprout-EWMA [39], but enhanced to be video-aware: the capacity-estimation scheme accounts for the intermittent (frame-by-frame) data generated by the video codec.

Scalable or layered 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 dis-

card 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.) Scalability is particularly useful in multiparty videoconferences, because it allows a relay node to adapt a sender's video stream to different receiver network capacities by discarding enhancement layers, without re-encoding. Salsify is aimed at unicast situations; in this setting, we evaluated a contemporary SVC system, VP9-SVC as part of WebRTC in Google Chrome, and found that it did not improve markedly over conventional WebRTC.

Measurement of real-time video systems. Prior work has evaluated the performance of integrated videoconferencing applications. Zhang and colleagues [46] varied the characteristics of an emulated network path and measured how Skype varied its network throughput and video frame rate. Xu and colleagues [41] used Skype and Hangouts to film a stopwatch application on the receiver computer's display, producing two clocks side-by-side on the receiver's screen to measure the one-way video delay.

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. The system measures the end-to-end video quality and delay of every frame.

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 [44] and inferential approaches to selecting relays and routings [20, 12]. 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 systems use several techniques to recover from packet loss. RTP and WebRTC applications sometimes retransmit the lost packet, and sometimes re-encode missing slices of a video frame *de novo* [28, 25]. By contrast, Salsify's functional video decoder retains old states in memory until the sender gives permission to evict them. If a network has exhibited recent packet loss, the encoder can start encoding new frames in a way that depends only on an older state that the receiver has acknowledged, allowing the frame to be decoded even if intervening packets turn out to have been lost. This approach has been described as "prophylactic retransmission" [37].

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's 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 bit rate. The transport protocol [17] updates the encoder's frame rate and target bit rate 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 bit rate, 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 Google's VP8 format [36]. 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 (We previously described an earlier version of this codec in ExCamera [9].)

The state includes copies of previous decoded frames, known as reference images, and probability tables used for entropy coding. At a resolution of 1280×720, the internal state of a VP8 decoder is about 4 MiB. To compress a new image, the video encoder takes advantage of its similarities with the reference images in the state. The video decoder can be modeled as an automaton, with coded frames as the inputs that cause state transitions between a source and target state. The automaton starts in the source state, consumes the compressed frame, outputs an image for display, and transitions to the target state.

In typical implementations, whether hardware or software, this state is maintained internally by the encoder/decoder and is inaccessible to the application. The encoder ingests a stream of raw images as the input, and produces a compressed bitstream. When a frame is encoded, the internal state of the encoder changes and there is no way to undo the operation and return to the previous state. Salsify's VP8 encoder and decoder, by contrast, are pure functions with no side effects and all state maintained externally. The interface is:

$$\textbf{decode}(\texttt{state}, \texttt{frame}) \rightarrow (\texttt{state}', \texttt{image})$$
$$\textbf{encode}(\texttt{state}, \texttt{image}, \texttt{quality}) \rightarrow \texttt{frame}$$

Using this interface, the application can explore different quality options for encoding each frame and start decoding from a desired state. This allows Salsify to (1) encode frames at a size and quality that matches the network capacity and (2) efficiently recover from packet loss.

Encoding to match network capacity. Current video encoders, including Salsify's, are unable to compress a single frame to accurately match a specified coded length. Only after compressing a frame does the encoder discover the resulting length with any precision. As a result, current videoconferencing systems generally track the network's capacity in an average sense, by matching the encoder's average bit rate over time to the network's data rate.

Salsify, by contrast, exploits the functional nature of its video encoder to optimize the compressed length of each individual frame, based on a current estimate of the network's 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, after knowing the exact size of the compressed frames. 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 higher, one lower) whichever settings were successful in the previous frame.

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. Salsify's loss recovery strategy condenses into picking the *right* source state for encoding frames. In the absence of loss, the encoder produces a sequence of compressed frames, each one depending on the target state resulting from the previous frame, even if that frame has not yet been acknowledged as received—the sender assumes that all the packets in flight will be delivered. Packet loss, however, causes incomplete or missing frames at the receiver, putting its decoder in a different state than the one assumed by the sender and corrupting the decoded video stream. To remedy this, when the sender detects packet loss (via ACKs from the receiver, § 3.2), it resynchronizes the receiver's state by creating frames that depend on a state that the receiver has explicitly acknowledged (Algorithm 1.3 and Algorithm 2.3); these frames will be usable by the receiver, even if intermediate packets are lost. This approach requires the sender and receiver to save the sequence of target states in memory, only deleting them when safe. Specifically, upon receiving a frame based on some state, the receiver discards all older ones; and when the sender receives an ACK for some state, it discards all older ones.

In case of packet reordering, if a fragment for a new frame is received before the current frame is complete, the receiver still decodes the incomplete frame—which puts its decoder in an invalid state—and moves on the next frames. The sender recognizes this situation as packet loss and handles it the same way. Packet reordering within the fragments of a frame is not disruptive, as the receiver first waits for all the fragments before reassembling and decoding a frame.

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.

Other than the frame serial number and fragment index, each frame's header contains the hash of its source state, and the hash of its target state. With these headers, a compressed video frame becomes an idempotent operator that

Algorithm 1 Salsify transport protocol

```
 1: procedure ON-RECEIVING-ACK(ack)
 2:    set to values indicated by ack:
          mean_interarrival_time,
          known_receiver_codec_state,
          num_packets_outstanding
 3:    if ack indicates loss then
 4:       /* enter loss recovery mode for next 5 seconds */
 5:    end if
 6:    max_frame_size ← MTU × (100 ms /
          mean_interarrival_time −
          num_packets_outstanding)
 7: end procedure
```

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. The receiver stores the target state in memory, in the case that the sender wants to use that state for loss recovery. In reply to each fragment, the receiver sends an acknowledgment message that contains the frame number and the fragment index, along with its current decoder hash.

The receiver treats the incoming packets as a packet train [21, 18] to probe the network and maintains a moving average of packet inter-arrival times, similar to WebRTC [17, 6] and Sprout-EWMA [39]. This estimate is communicated to the sender in the acknowledgment packets. However, 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 2). 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. As fragment i is received, the receiver calculates the smoothed inter-arrival time, τ_i, as

$$\tau_i \leftarrow \alpha(T_i - T_{i-1} - \text{grace-period}_i) + (1 - \alpha)\tau_{i-1},$$

where T_i is the time fragment i is received. The value of α is 0.1 in our implementation, approximating a moving average over the last ten arrivals.

At the sender side, Salsify's transport protocol estimates the desired size of a frame based on the latest average inter-arrival time reported by the receiver. To calculate the target size at time step i, the sender first estimates an upper bound for the number of packets already in-flight, N_i, by subtracting the indices of the last-sent packet and the last-acknowledged packet. Let τ_i be the latest average inter-arrival time reported by the receiver at i. If the sender aims to keep the end-to-end delay less than d (set to 100 ms in our implementation) to preserve

interactivity, there can be no more than d/τ_i packets in flight. Therefore, the target size is $(d/\tau_i - N_i)$ MTU-size fragments (Algorithm 1.6). At the time of sending, the sender will pick the largest frame that doesn't exceed this length. If both encoded versions are over this size, the sender discards the frame and moves on to sending the next frame. To be able to receive new feedback from the receiver, if more than four frames are skipped in a row, the sender sends the low quality version (Algorithm 2).

Algorithm 2 Salsify sender program

```
 1: procedure SEND-NEXT-FRAME
 2:    image ← NEXT-IMAGE-FROM-CAMERA
 3:    source_state ← loss_recovery_mode
          ? known_receiver_codec_state
          : last_sent_frame.target_codec_state

 4:    frame_lower_quality ← ENCODE(
          source_state, image,
          last_sent_frame.quality − DECR)
 5:    frame_higher_quality ← ENCODE(
          source_state, image,
          last_sent_frame.quality + INCR)

 6:    frame_to_send ← SELECT-FRAME(
          frame_lower_quality,
          frame_higher_quality)
 7:    if frame_to_send ≠ null then
 8:       SEND(frame_to_send)
 9:       consecutive_skip_count ← 0
10:       last_sent_frame ← frame_to_send
11:    end if
12: end procedure
13:
14: function SELECT-FRAME(lower, higher)
15:    if higher.length < max_frame_size then
16:       return higher
17:    else if lower.length < max_frame_size or
          consecutive_skip_count ≥ 4 then
18:       return lower
19:    else
20:       consecutive_skip_count++
21:       return null
22:    end if
23: end function
```

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 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).
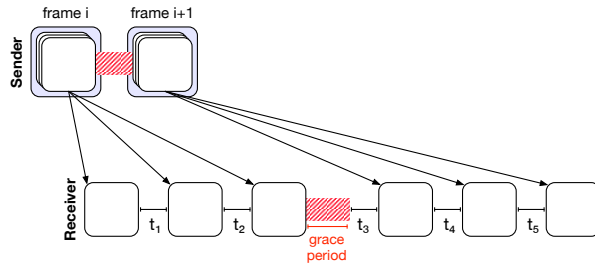
Figure 2: The receiver maintains a moving average of packet inter-arrival times, t_is. The sender includes the delay between sent packets as a "grace period," so the receiver can account for the sender's pauses between frames.

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) [34], 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 §5.1).

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.

4.2 Design

Figure 3 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.[1] 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 3 and 4. The quality and delay metrics are computed in postprocessing by matching the barcodes on sent and received frames, then comparing corresponding frames.

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.

The measurement testbed computes SSIM using the Xiph Daala tools package. For network emulation, we use Cellsim [39], 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 applications were not sensitive to this parameter and conducted

---

[1]The two barcodes were designed to detect tearing within a frame, when the receiver displays pieces of two different source frames at the same time. In our evaluations, we did not see this occur.
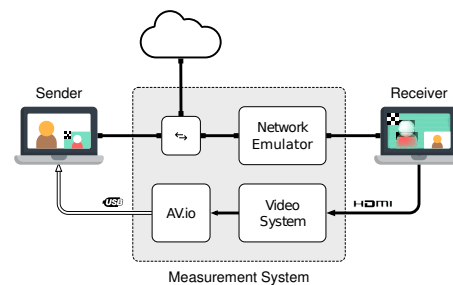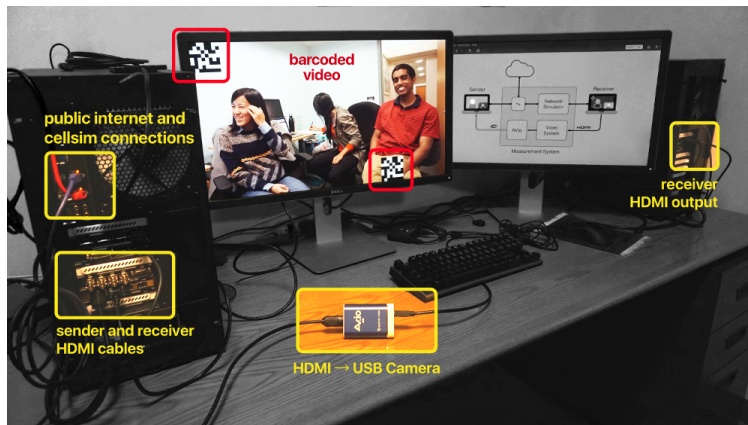
Figure 3: Testbed architecture. The system measures the performance of an unmodified real-time video system. It emulates a webcam to supply a barcoded video clip to the sender. The sender transmits frames to the receiver via an Ethernet connection. The measurement testbed interposes on the receiver's 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.
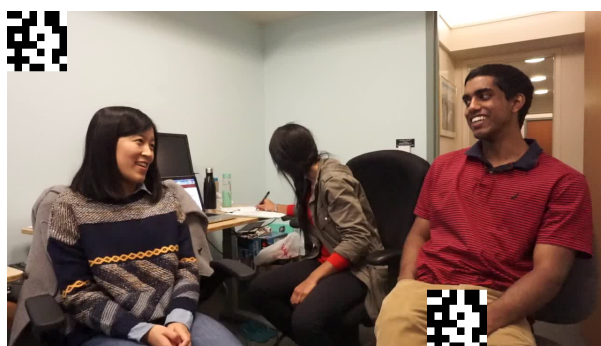
Figure 4: An example barcoded video frame sent by the measurement testbed (§4.2). The barcodes each represent a 64-bit random number that is unique over the course of the video.

remaining tests with a 256-packet buffer. The round-trip delay was set to 40 ms for cellular traces. We developed new software for barcoding, playing, capturing, and analyzing video. It comprises about 2,500 lines of C++.

5 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 intermittent link and an emulated Wi-Fi link.

5.1 Setup, calibration, and method

Setup. We ran all experiments using the measurement testbed described in Section 4. Figure 5 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.[2] 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

---

[2]The arguments were: out/Release/chrome `--enable-webrtc-vp9-svc-2sl-3tl` `--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)	Chrome 55.0 Chrome 62.0 (Figure 6e)	Edited CSS to hide self view.
WebRTC	Chrome (Linux)	Chrome 62.0, https://appr.tc Chrome 55.0 (Figure 7) Chrome 65.0 (Figure 8)	Edited CSS to hide self view.
WebRTC (VP9-SVC)	Chrome (Linux)	Chrome 62.0, https://appr.tc	Edited CSS to hide self view.

Figure 5: Application 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.

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. The network traces are long enough to cover the whole length of the video.

5.2 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 [27]. 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 6a, 6b and 6c 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). 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 on the PC workstation tested.

Salsify's loss-recovery strategy requires the sender and receiver keep a set of decoder states in memory, in order to recover from a known state if loss occurs. After the receiver acknowledges a state, or the sender sends a frame based on a state, the other program discards the older

states. In our AT&T LTE trace experiment, Salsify-2c sender kept 6 states on average, each 4 MiB in size, during the course of its execution, while the receiver kept 3 states at a time on average. Additionally, in the same experiment, Salsify-2c picked the better-quality option in 50%, the lower-quality option in 32%, and not sending the frame at all in 18% of the opportunities to send a frame.
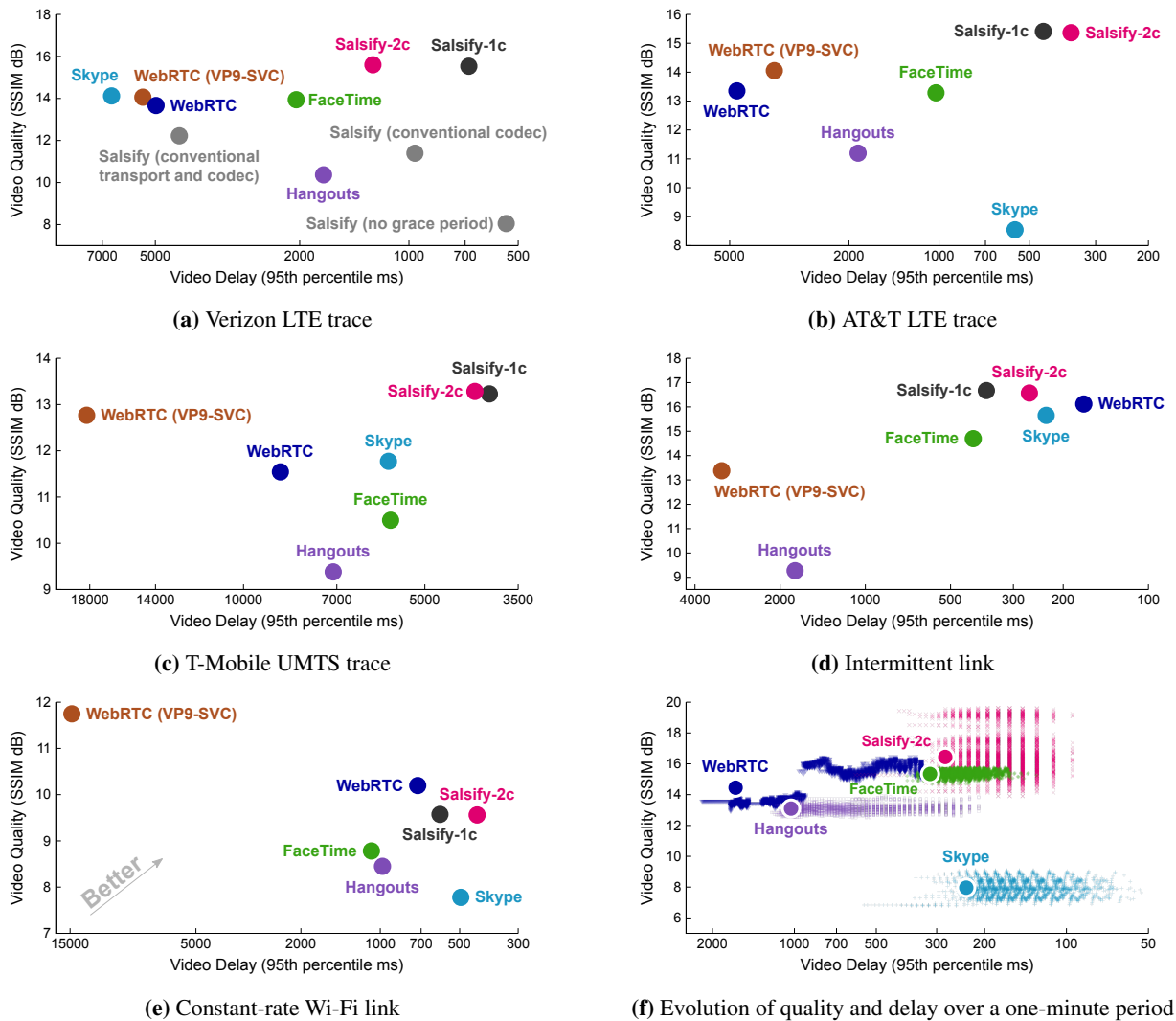
Figure 6f shows how the quality and delay of different schemes vary during a one-minute period of AT&T LTE trace where the network path was mostly steady in capacity.

Experiment 2: intermittent link. In this experiment, we evaluated Salsify's method of loss resilience. We ran each system on a two-state intermittent link. The link's capacity is 12 Mbps with no drops until a "failure" arrives, which happens on average after five seconds (exponentially distributed). During failure, all packets are dropped until a "restore" event arrives, on average after 0.2 seconds of failure. The experiment's duration is 10 minutes.

Figure 6d 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: emulated Wi-Fi. In this experiment, we evaluated Salsify and the other systems on a challenged network path that, unlike the cellular traces, does not vary its capacity with time. This emulated Wi-Fi network path matches the behavior of a long-distance free-space Wi-Fi hop, with the emulation parameters taken from [42], including an average data rate of about 570 kbps and Poisson packet arrivals. Figure 6e shows the results. Salsify is on the Pareto frontier, and WebRTC also performs well when the network data rate does not vary with time.

Experiment 4: component analysis study. In this experiment, we removed the new components implemented

(a) Verizon LTE trace

(b) AT&T LTE trace

(c) T-Mobile UMTS trace

(d) Intermittent link

(e) Constant-rate Wi-Fi link

**(f)** Evolution of quality and delay over a one-minute period

Figure 6: Figures (a)–(e) show the end-to-end video quality and video delay over four emulated networks. Salsify-1c and Salsify-2c 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 "intermittent link" network, which occasionally drops packets but is otherwise a constant 12Mbps (§5), and an emulated constant-rate Wi-Fi link. The AT&T, T-Mobile, Wi-Fi, and and intermittent-link scenarios were held out during development. Figure (f) shows the evolution of the quality and delay during a one-minute period of the AT&T LTE trace when network capacity remained roughly constant.

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 6a as the "Salsify (no grace period) dot"; without this component, Salsify underestimates the network capacity and sends low-quality, low-bitrate video.

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 frame size and extrapolating the resulting size to the full frame. The target size was selected using the same transport protocol in normal Salsify. The result is also in Figure 6a as "Salsify (conventional codec)."

As shown in the plot, Salsify's performance is again substantially reduced. This is a result of two factors: (1) The transport no longer has access to a choice of frames at

(a) Throughput

(b) Frame delay
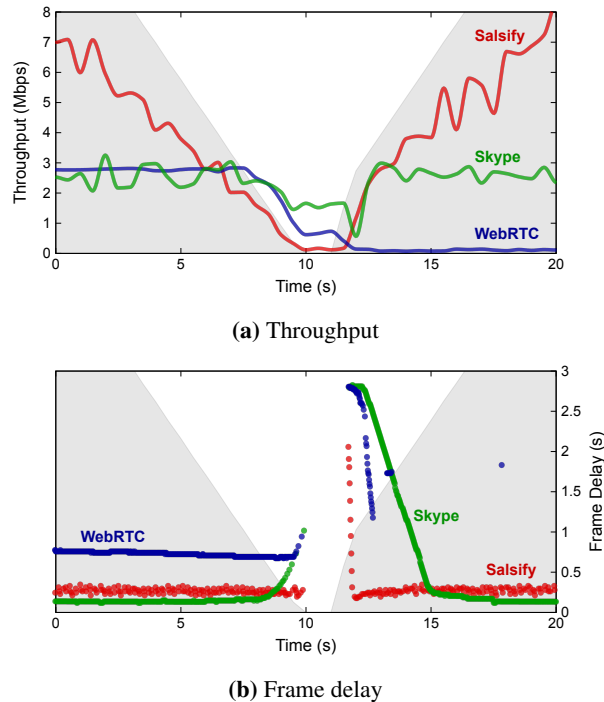
Figure 7: Salsify's reacts more quickly to changes in network conditions than other video systems. This is illustrated by comparing the performance of Skype, WebRTC, and Salsify over a network path whose capacity decreases gradually to zero, then back up again (instantaneous network capacity shown in gray).

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, and (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.

Finally, we created an end-to-end system that emulates the behavior of the conventional videoconferencing systems, by removing the distinctive features of both Salsify's video codec and transport protocol. Rather than operating frame by frame, the transport protocol in this implementation estimates the network's average data rate and updates the quality settings of the video codec, once every second. As the "Salsify (conventional transport and codec) dot" in Figure 6a shows, this implementation has a similar performance to Skype and WebRTC.

We conclude that each of Salsify's distinctive features— the video-aware transport protocol, purely functional codec, and the frame-by-frame coupling between them that merges the rate-control algorithms of each module— contributes positively to the system's overall performance.

Experiment 5: capacity ramp. In this experiment, we evaluated how Salsify, Skype, and WebRTC handle a network with a gradual decrease in data rate (to zero), then

a gradual resumption of network capacity. We created the synthetic network trace depicted in light gray in Figures 7a and 7b. The experiment's duration is 20 seconds.

Figure 7a shows the data transmission rate each scheme tries to send through the link, versus time. Salsify's throughput smoothly decreases alongside link capacity, then gracefully recovers. The result is that Salsify's video display recovers quickly after link capacity is restored, as shown in Figure 7b.

In contrast, Skype reacts slowly to degraded network capacity, and as a result induces loss on the link and a standing queue, both of which delay the resulting video for several seconds. 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 a handful of frames (marked with blue dots) in the eight seconds after the link begins to recover.

Experiment 6: one-second dropout experiment. In this experiment, we compared the effect of packet loss on Salsify-2c and WebRTC by introducing a single dropout for 1 s while running each application on an emulated link with a constant data rate of 500 kbps. All of the packets that were scheduled to be delivered during the outage were dropped. Figure 8 shows the results. After the network is restored, WebRTC's transport protocol retransmits the *packets* that were lost during the outage, causing a spike in delay before its video codec starts encoding new frames of video (WebRTC's baseline delay is also considerably larger). Salsify does not have the same independence between its video codec and transport protocol; upon recovery of the network, Salsify's functional video codec can immediately encode new frames in terms of reference images that are already present at the receiver. This results in faster recovery from the dropout.
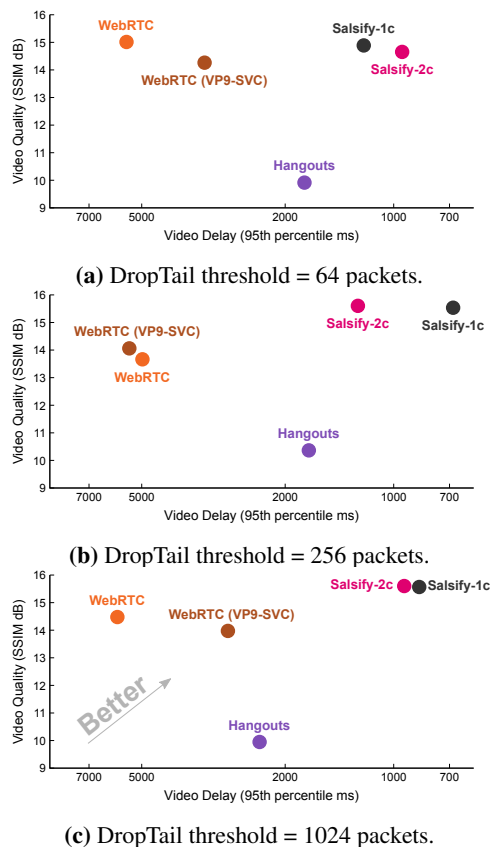
Experiment 7: sensitivity to queueing policy. In this experiment, we quantified the performance impact of network buffer size on Salsify, Hangouts, WebRTC, and WebRTC (VP9-SVC) for the Verizon-LTE network trace. The plots in Figure 9 show the performance of each system on the trace across various DropTail 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 because all schemes are striving for low delay and therefore are unlikely to build up a large-enough standing queue to see DropTail-induced packet drops. We ran the remaining tests using the middle setting (256 packets).

5.3 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

(a) Frame delay

(b) Frame quality
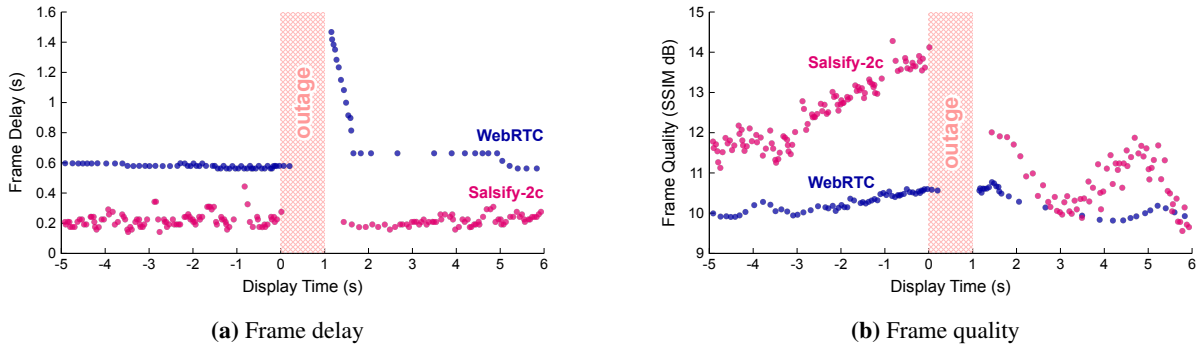
Figure 8: Comparison of the response of Salsify and WebRTC (implementation in Chrome 65) to a single loss event for one second, while communicating over a network path with a constant data rate of 500 kbps. During the loss episode, all packets were dropped. WebRTC displays frames out of a receiver-side buffer during the outage. In contrast to Salsify's strategy of always encoding the most recent video in terms of references available at the receiver, WebRTC's transport protocol retransmits packets lost during the outage before its video encoder starts encoding new frames. This causes a spike in the video delay and wide variations in the frame rate.

(a) DropTail threshold = 64 packets.

(b) DropTail threshold = 256 packets.

(c) DropTail threshold = 1024 packets.

Figure 9: Sensitivity to queueing policy. We measured performance over the Verizon LTE network trace with different in-network buffer sizes. The tested systems were not particular sensitive to this parameter. We ran the other experiments with a buffer size of 256 packets.

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 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 UDP connection between the two client machines. Rather, the clients communicated through a Google relay server, still via UDP. 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 ∼5 ms on average.

6 Limitations and Future Work

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

6.1 Limitations of Salsify

No audio. Salsify does not encode or transmit audio. When testing other applications, we disabled audio to

avoid giving Salsify an unfair advantage. Adding audio to a videoconferencing system creates a number of opportunities for future work in QoE optimization—e.g., it is generally beneficial for the receiver to delay audio in a buffer to allow uninterrupted playback in the face of network jitter, even at the expense of some added delay. To what extent should video be similarly delayed to keep it in sync with the audio, and what are the right metrics to evaluate any compromise along these axes?

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 a 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. On power-constrained devices, only hardware codecs are sufficiently power-efficient, so we are hopeful that Salsify's results will motivate hardware codec implementers to expose state as Salsify does.

Improved conventional codecs could render Salsify's functional codec unnecessary. The benefit of Salsify's purely functional codec is principally in its ability to produce (by trial and error) compressed frames whose length matches the transport protocol's estimate of the network's instantaneous capacity. To the extent that conventional codecs also grow this capability, the benefits of a functional codec in this setting will shrink.

Benefits are strongest when the network path is most variable. Salsify's main contribution is in combining the rate-control algorithms in the transport protocol and video codec, and exploiting the functional codec to coax individual compressed frames that match the network's instantaneous capacity, even when it is highly variable. On network paths that exhibit such variability (e.g. cellular networks while moving), Salsify demonstrated a significant performance advantage over current applications. On less-variable networks, Salsify's performance was closer to existing applications.

6.2 Limitations of the evaluation

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 video streams simultaneously.

The traces do not reflect multiple flows sharing the same queue. To achieve a fair evaluation of each application, we used the same test video and ran over a series of reproducible network emulators. We did not evaluate the schemes over "wild" real-world paths. The trace-based network emulation replays the actual packet timings (in both the forward and reverse direction) captured from cellular networks. These traces capture several phenomena, including the effect of multiple hops, ACK compression in the reverse path, and cross traffic from other flows and users sharing the network while the traces were recorded, reducing the available data rate of the network path. However, the emulation does not capture cross traffic that shares the same bottleneck queue as the application under test. Generally speaking, no end-to-end application can achieve low-latency video when the bottleneck queue is shared with "bufferbloating" cross traffic [13].

7 Conclusion

In this paper, we presented Salsify, a new architecture for real-time Internet video that tightly integrates a video codec and a network transport protocol. 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, (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 capacity, and (3) Salsify merges the rate-control algorithms in the video codec and transport protocol to avoid provoking packet drops and queueing delay with its own traffic.

In an end-to-end evaluation, Salsify achieved lower end-to-end video delay and higher quality when compared with five existing systems: Skype, FaceTime, Hangouts, and WebRTC's reference implementation with and without scalable video coding (VP9-SVC).

It is notable that Salsify achieves superior visual quality than other systems, as Salsify uses our own implementation of a VP8 codec—a largely superseded compression scheme, and an unsophisticated encoder for that scheme. The results suggest that further improvements to video *codecs* may have reached the point of diminishing returns in this setting, but changes to the architecture of video *systems* can still yield significant benefit.

Acknowledgments

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] CHOU, P. A., AND MIAO, Z. Rate-distortion optimized streaming of packetized media. *IEEE Transactions on Multimedia 8*, 2 (April 2006), 390–404.

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

[7] 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).

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

[9] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)* (2017), USENIX Association, pp. 363–376.

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

[11] 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).

[12] 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).

[13] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the Internet. *Queue 9*, 11 (Nov. 2011), 40:40–40:54.

[14] GRANGE, A., DE RIVAZ, P., AND HUNT, J. VP9 Bitstream & Decoding Process Specification version 0.6, March 2016. http://www.webmproject.org/vp9/.

[15] 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).

[16] HERMANNS, N., AND SARKER, Z. Congestion control issues in real-time communication—"Sprout" an example. Internet Congestion Control Research Group. https://datatracker.ietf.org/meeting/88/materials/slides-88-iccrg-3.

[17] 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.

[18] 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* (2002), SIGCOMM '02, ACM, pp. 295–308.

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

[20] 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).

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

[22] 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.

[23] 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.

[24] 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).

[25] 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/.

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

[27] 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.

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

[29] 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).

[30] 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* (2010), SIGCOMM '10, ACM, pp. 15–26.

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

[32] 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.

[33] SWETT, I. QUIC FEC v1. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk.

[34] 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.

[35] WEBRTC.ORG. WebRTC Native Code. https://webrtc.org/native-code.

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

[37] WINSTEIN, K., AND BALAKRISHNAN, H. Mosh: A State-of-the-Art Good Old-Fashioned Mobile Shell. In *;login:* (37, 4, August 2012).

[38] WINSTEIN, K., AND BALAKRISHNAN, H. Mosh: An interactive remote shell for mobile clients. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), USENIX. Available at https://mosh.org., pp. 177–182.

[39] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* (2013), USENIX, pp. 459–471.

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

[41] 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).

[42] YAN, F. Y., MA, J., HILL, G., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for Internet congestion-control research. Measurement at http://pantheon.stanford.edu/result/1622/.

[43] YAP, K.-K., HUANG, T.-Y., YIAKOUMIS, Y., MCKEOWN, N., AND KATTI, S. Late-binding: how to lose fewer packets during handoff. In *Proceedings of the 2013 Workshop on Cellular Networks: Operations, Challenges, and Future Design* (2013), ACM, pp. 1–6.

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

[45] ZHAI, F., AND KATSAGGELOS, A. *Joint Source-Channel Video Transmission*. Morgan & Claypool, 2007. https://doi.org/10.2200/S00061ED1V01Y200707IVM010.

[46] 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.

A   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 Stanford University. The participants were all Stanford graduate students and were unpaid. In both studies, we varied video delay and quality over the ranges observed in the comparative evaluation (Section 5). Our results show that small variations in video delay greatly affect mean opinion score; video quality also affects mean opinion score but less so.

In the first 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 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 × delay: {10 dB, 14 dB, 18 dB} × {300ms, 1200ms, 2200ms, 4200ms}).

The second user study put participants behind the wheel of a race car in a simulated environment: a PlayStation 4 playing the "Driveclub" videogame. Us-

ing a second test jig, the visual quality and the delay between the PlayStation's HDMI output and the participant's display were 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 × delay: {8 dB, 11 dB, 14 dB} × {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 10. 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 teleoperation of vehicles, achieving low video delay is more critical than increasing video quality, even more than in person-to-person videoconferencing.

The equations for the best fit lines are given below.

$$\text{QoE}_{\text{video call}} = -6.39 \cdot 10^{-6} \times \text{DELAY}_{\text{ms}} + 6.22 \cdot 10^{-2} \times \text{SSIM}_{\text{dB}} + 3.30$$

$$\text{QoE}_{\text{driving}} = -1.92 \cdot 10^{-3} \times \text{DELAY}_{\text{ms}} + 1.01 \cdot 10^{-1} \times \text{SSIM}_{\text{dB}} + 2.67$$

(a) Video call   **(b)** Driving simulation
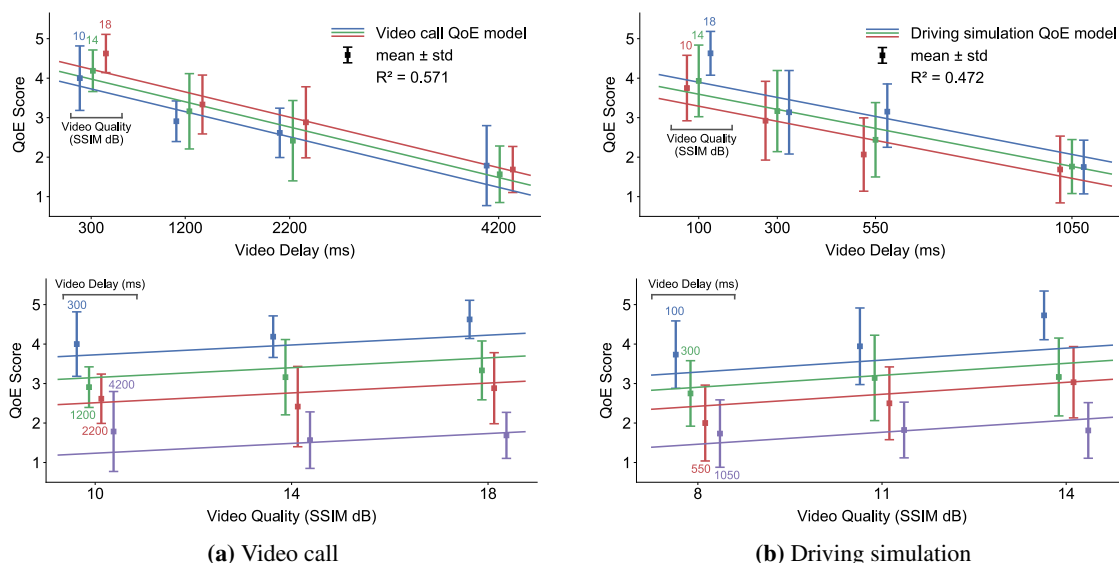
Figure 10: The results of the two user studies. The data from each study were fit to a two-dimensional linear model—one for videoconferencing, one for driving—using ordinary least squares. The upper plots project the learned bilinear models onto the delay-QoE axes; similarly, the lower plots show the 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).

B Numerical evaluation results

System	Trace	Video Quality (SSIM dB)		Video Delay (ms)		Data
		p25	mean	mean	p95	
Salsify-1c		15.1	15.5	517.4	684.0	↗
Salsify-2c		15.1	15.6	496.7	1256.8	↗
FaceTime		15.0*	13.9	658.6	2044.2	↗
Hangouts	Verizon LTE	9.8	10.4	560.9	1719.0	↗
Skype		15.1*	14.1	1182.6	6600.2	↗
WebRTC		13.2	13.7	973.0	4977.4	↗
WebRTC (VP9-SVC)		13.6	14.1	1196.1	5411.9	↗
Salsify-1c		15.0	15.4	349.1	448.5	↗
Salsify-2c		15.0	15.4	282.1	362.4	↗
FaceTime		12.6	13.3	469.4	1023.6	↗
Hangouts	AT&T LTE	10.7	11.2	846.4	1862.4	↗
Skype		8.2	8.5	322.1	557.4	↗
WebRTC		12.4	13.4	934.7	4729.9	↗
WebRTC (VP9-SVC)		13.5	14.1	775.2	3547.2	↗
Salsify-1c		13.0	13.2	840.1	3906.8	↗
Salsify-2c		12.9	13.3	803.3	4129.4	↗
FaceTime		8.8	10.5	1206.8	5699.6	↗
Hangouts	T-Mobile UMTS	8.5	9.4	1012.0	7096.9	↗
Skype		11.1	11.8	1451.8	5745.9	↗
WebRTC		10.4	11.5	1795.7	8685.5	↗
WebRTC (VP9-SVC)		12.1	12.8	2585.2	18215.3	↗
Salsify-1c		15.9	16.7	265.2	373.6	↗
Salsify-2c		15.8	16.6	181.9	263.3	↗
FaceTime		14.6	14.7	280.2	415.8	↗
Hangouts	Intermittent Link	9.1	9.3	437.0	1771.4	↗
Skype		15.5	15.7	128.4	229.7	↗
WebRTC		16.0	16.1	155.8	169.1	↗
WebRTC (VP9-SVC)		12.3	13.4	1735.2	3216.9	↗
Salsify-1c		9.0	9.6	317.7	593.9	↗
Salsify-2c		9.0	9.6	234.8	429.2	↗
FaceTime		8.5	8.8	609.5	1080.2	↗
Hangouts	Emulated Wi-Fi Link	8.2	8.4	514.4	980.5	↗
Skype		7.5	7.8	250.9	495.1	↗
WebRTC		10.0	10.2	315.2	721.0	↗
WebRTC (VP9-SVC)		11.4	11.7	2512.4	14767.3	↗

Figure 11: Summary of results of the evaluation (Section 5). 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. In the PDF version of this paper, the icons in the data column link to the raw data for each item, within the repository at https://github.com/excamera/salsify-results.