Performance Evaluation of WebRTC-based Video Conferencing

Bart Jansen
Delft University of Technology
b@rtjansen.nl

Timothy Goodwin
Computer Science
Columbia University
t.goodwin@columbia.edu

Varun Gupta
Electrical Engineering
Columbia University
varun@ee.columbia.edu

Fernando Kuipers
Delft University of Technology
f.a.kuipers@tudelft.nl

Gil Zussman
Electrical Engineering
Columbia University
gil@ee.columbia.edu

ABSTRACT

WebRTC has quickly become popular as a video conferencing platform, partly due to the fact that many browsers support it. WebRTC utilizes the Google Congestion Control (GCC) algorithm to provide congestion control for real-time communications over UDP. The performance during a WebRTC call may be influenced by several factors, including the underlying WebRTC implementation, the device and network characteristics, and the network topology. In this paper, we perform a thorough performance evaluation of WebRTC both in emulated synthetic network conditions as well as in real wired and wireless networks. Our evaluation shows that WebRTC streams have a slightly higher priority than TCP flows when competing with cross traffic. In general, while in several of the considered scenarios WebRTC performed as expected, we observed important cases where there is room for improvement. These include the wireless domain and the newly added support for the video codecs VP9 and H.264 that does not perform as expected.

Keywords

WebRTC, Congestion Control, Performance Evaluation

1. INTRODUCTION

WebRTC provides Real-Time Communication (RTC) capabilities via browser-to-browser communication for audio (voice calling), video chat, and data (file sharing). It allows browsers to communicate directly with each other in a peer-to-peer fashion, which differs from conventional browser to web-server communication. One of the main advantages of WebRTC is that it is integrated in most modern web browsers and runs without the need to install external plugins or applications. The World Wide Web Consortium (W3C) [4] has set up an Application Programming Interface (API), which allows developers to easily implement WebRTC using JavaScript, while the Internet Engineering Task Force (IETF) [14] defines the WebRTC protocols and underlying formats.

To realize the low latency and high throughput necessary for real-time communication, WebRTC prioritizes transmitting data using UDP instead of TCP. WebRTC over TCP is used as a last resort, when all UDP ports are blocked, which can be the case in heavily-protected enterprise networks. Since UDP does not support any form of congestion control, WebRTC uses a custom-designed congestion control algorithm that adapts to changing network conditions.

With the high-level API, WebRTC makes it easy for application developers to develop their own video streaming applications. The disadvantage of this high-level approach is that the performance details, especially the way congestion is handled, are completely hidden from application developers. At the same time, recent research evaluating the performance of WebRTC has only partially addressed this gap (see Section 7 for more details).

In this paper, *we take a closer look at the performance of WebRTC, mainly focusing on the Google Congestion Control (GCC) algorithm*, which is the most widely used congestion control algorithm for WebRTC. We evaluate its performance using the latest web browsers across a wide range of use cases. Our key contributions consist of studying the effects of different synthetic network conditions on the latest implementations of WebRTC, comparing WebRTC's performance on mobile devices, analyzing the performance of the newly added video codecs VP9 and H.264, and evaluating the impact of wired and wireless networks on WebRTC. The source code for reproducing the experimental conditions described in this paper is available at:
`https://github.com/Wimnet/webrtc_performance`

In particular, our experimental study includes the following:

(i) **Baseline Experiments:** We study the effects of varying latency, packet loss, and available bandwidth by emulating different performance environments using Dummynet. We establish benchmarks for the performance of WebRTC in different scenarios.

(ii) **Cross Traffic:** We study the effects of TCP cross traffic and multiple WebRTC streams sharing the same bottleneck. Our evaluations indicate that with the recent enhancements to the congestion control mechanism, WebRTC streams receive slightly higher priority when competing with TCP flows.

IFIP WG 7.3 Performance 2017. Nov. 14-16, 2017, New York, NY USA

(iii) **Multi-Party Topology:** We compare the performance of a mesh and Selective Forwarding Unit (SFU) based topologies for group video calls using WebRTC. Our evaluation highlights inherent trade-offs between performance and deploying additional infrastructure for multi-party video calls.

(iv) **Video Codecs:** We study the performance of three widely used video codecs, VP8, VP9, and H.264, on WebRTC. Our experiments demonstrate that the newly added H.264 and VP9 codecs do not perform as expected in the presence of congestion or packet losses.

(v) **Mobile Performance:** We evaluate the performance of WebRTC on mobile devices and demonstrate the impact of limited computational capacity on call quality.

(vi) **Real Wireless Networks:** We experimentally evaluate video calls on WebRTC in real networks, specifically focusing on wireless networks. Our experiments show that WebRTC can suffer from poor performance over wireless due to bursty losses and packet retransmissions. We identify key areas for improvement and briefly look at cross-layer approaches for improving video quality.

Our performance evaluation focuses on a few key metrics such as data rate, frame rate, Round Trip Time (RTT), and call setup time, which have been shown to be the key factors that affect the user video experience [20, 3]. Overall, this paper presents a thorough performance evaluation of WebRTC and discusses various performance-related trade-offs.

The remainder of this paper is organized as follows. In Section 2, we briefly describe the congestion control algorithm used by WebRTC and in Section 3, we describe the setup used to conduct the experiments. Section 4 presents the performance analysis results in synthetic network conditions. Section 5 focuses on the impact of video codecs and mobile devices on call quality. Section 6 takes a closer look at the performance of WebRTC in the wireless domain over real networks. Section 7 presents related work and we conclude in Section 9, where we also discuss future research directions.

2. CONGESTION CONTROL

Table 1: GCC notation.

Parameter	Description
A_r	Estimated available rate at the receiver
A_s	Sender rate
$R(i)$	Measured receive rate for last 500ms
t_i	Arrival time of i^{th} video frame
d_i	Measured one-way delay gradient
m_i	Filtered one-way delay gradient
γ_i	Dynamic over-use threshold
t_k	Arrival time of k_{th} RTCP report
$f_l(t_k)$	Fraction of lost packets

WebRTC uses the Google Congestion Control (GCC) algorithm [15], which dynamically adjusts the data rate of the video streams when congestion is detected. In this section, we provide a brief overview of GCC. More details can be found in [10]. WebRTC typically uses UDP (unless all UDP ports are blocked), over which it uses the Real-time Transport Protocol (RTP) to send media packets. It receives feedback packets from the receiver in the form of RTP Control Protocol (RTCP) reports. GCC controls congestion in two ways: delay-based control (section 2.1) at the receiving end and loss-based control (section 2.2) at the sender side.

2.1 Receiver-side controller

The receiver-side controller is delay-based and compares the timestamps of the received frames with the time instants of the frames' generation. The receiver-side controller consists of three different subsystems: (i) arrival time filter, (ii) over-use detector, and (iii) rate controller. These different subsystems of the receiver-side controller are shown on the right side of Figure 1. The arrival-time filter (Section 2.1.1) estimates the changes in queuing delay to detect congestion. The over-use detector detects the congestion by comparing the estimated queuing delay changes from the arrival-time filter with an adaptive threshold (Section 2.1.2). The rate controller makes the decisions to increase, decrease, or hold the estimated available rate at the receiver, A_r, based on the congestion estimated derived from the over-use detector (Section 2.1.3). $A_r(i)$ for the i^{th} video frame is given as follows:

$$A_r(i) = \begin{cases} \eta A_r(i-1) & \text{Increase} \\ \alpha R(i) & \text{Decrease} \\ A_r(i-1) & \text{Hold} \end{cases} \tag{1}$$

Where $\eta = 1.05$, $\alpha = 0.85$, and $R(i)$ is the measured received rate for the last 500 ms. The received rate can never exceed $1.5R(i)$:

$$A_r(i) = \min(A_r(i), 1.5R(i)) \tag{2}$$

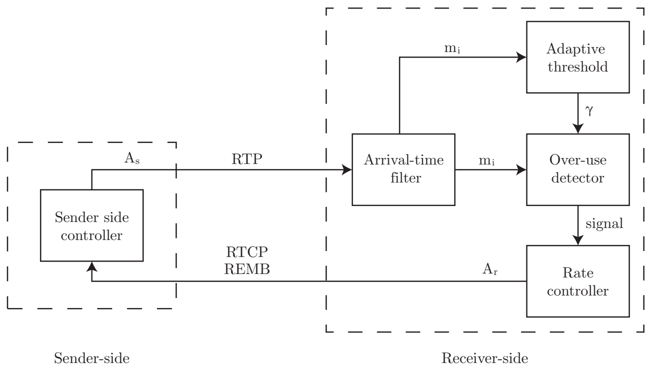

Figure 1: Diagram illustrating how sender and receiver determine and exchange their available rate.

2.1.1 *Arrival-time filter*

The arrival-time filter continuously measures the time instants at which packets are received. It uses the time of arrivals to calculate the inter-arrival time between two consecutive packets: $t_i - t_{i-1}$, and the inter-departure time between the transmission of the same packets: $T_i - T_{i-1}$. It then calculates the one-way delay variation d_i, defined as the difference between inter-arrival time and inter-departure time as follows:

$$d_i = (t_i - t_{i-1}) - (T_i - T_{i-1}) \tag{3}$$

This delay shows the relative increase or decrease with respect to the previous packet. The one-way delay variation is larger than 0 if the inter-arrival time is larger than the inter-departure time. The arrival-time filter estimates the one-way queuing delay variation m_i. The calculation of m_i is based on the measured d_i and previous state estimate m_{i-1}, whose weights are dynamically adjusted by a Kalman filter to reduce noise in estimation. For instance, the weight for the current measurement d_i is weighed more heavily than the previous estimate m_{i-1} when the error variance is low. For more details, see [15].

2.1.2 *Over-use detector*

The estimated one-way queuing delay variation (m_i) is compared to a threshold γ. Over-use is detected, if the estimate is larger than this threshold. The over-use detector does not signal this to the rate controller, unless over-use is detected for a specified period of time. The over-use time is currently set to *100ms* [10]. Under-use is detected when the estimate is smaller than the negative value of this threshold and works in a similar manner. A normal signal is triggered when $-\gamma \leq m_i \leq \gamma$.

The value of the threshold has a large impact on the overall performance of the GCC congestion algorithm. A static threshold γ can easily result in starvation in the presence of concurrent TCP flows, as shown in [11]. Therefore, a dynamic threshold was implemented as follows:

$$\gamma_i = \gamma_{i-1} + (t_i - t_{i-1}) * K_i * (|m_i| - \gamma_{i-1}) \tag{4}$$

The value of the gain, K_i, depends on whether $|m_i|$ is larger or smaller than γ_{i-1}:

$$K_i = \begin{cases} K_d & |m_i| < \gamma_{i-1} \\ K_u & \text{otherwise} \end{cases} \tag{5}$$

where $K_d < K_u$. This causes the threshold γ to increase when the estimated m_i is not in the range of $[-\gamma_{i-1}, \gamma_{i-1}]$ and decrease when it does fall in that range. This helps increasing the threshold when, e.g., a concurrent TCP flow enters the bottleneck and avoids starvation of the WebRTC streams. According to [11], this adaptive threshold results in 33% better data rates and 16% lower RTTs when there is competing traffic sharing the same bottleneck.

2.1.3 *Rate controller*

The rate controller decides whether to increase, decrease, or hold A_r at the receiver depending on the signal received from the over-use detector. Initially, the rate controller keeps increasing A_r until over-use is detected by the over-use detector. Figure 2 further illustrates how the rate controller adjusts based on the signals received by the over-use detector.

A congestion/over-use signal always results in decreasing the rate, while under-use always results in keeping the rate unchanged. The state of the rate controller translates into available rate at the receiver, A_r, as shown in equation (1). A_r is sent back to the sender as an *REMB (Receiver Estimated Maximum Bandwidth)*[1] message in an RTCP report (Figure 1).

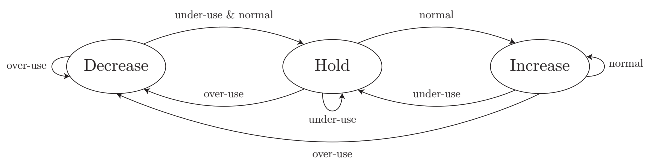

Figure 2: Rate controller state changes based on the signal output of the over-use detector.

2.2 Sender-side controller

The sender-side controller is loss-based and computes the sending rate at the sender, A_s in Kbps and is shown on the left side of Figure 1. A_s is computed every time (t_k) the k^{th} RTCP report or an REMB message is received from the receiver. The estimation of A_s is based on the fraction of lost packets $f_l(t_k)$ as follows:

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0.5 f_l(t_k)) & f_l(i) > 0.1 \\ 1.05 A_s(T_{k-1}) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases} \tag{6}$$

If the packet loss is between 2% and 10%, the sending rate remains unchanged. If more than 10% of the packets are reported lost, the rate is multiplicatively decreased. If the packet loss is smaller than 2%, the sending rate is linearly increased. Furthermore, the sending rate can never exceed the last available rate at the receiver $A_r(t_k)$, which is obtained through REMB messages from the receiver as seen in Figure 1.

3. EXPERIMENTAL SETUP

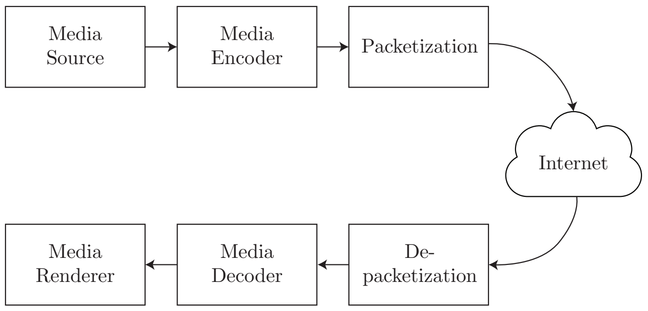

Figure 3: WebRTC's media processing pipeline.

In this section, we describe the setup used for experimental evaluation throughout the paper. WebRTC handles all media processing as illustrated in Figure 3. Raw media from the audio and video source are first preprocessed and then encoded at a given target rate. These frames are then packetized and sent to the receiver over RTP/UDP. These frames are subsequently depacketized and decoded, which provides the raw video input that can be rendered at the receiver.

[1]https://tools.ietf.org/html/draft-alvestrand-rmcat-remb-03

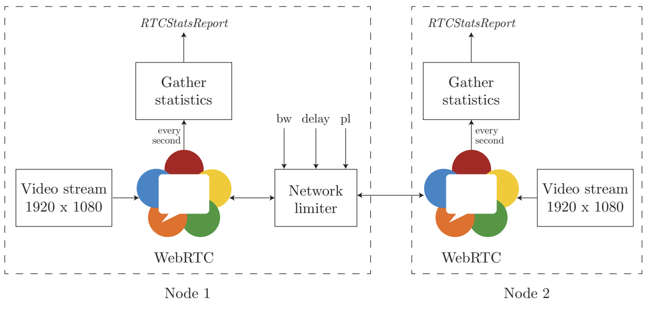

Figure 4: Experimental setup used for performance evaluation where the network limiter is simulated using Dummynet.

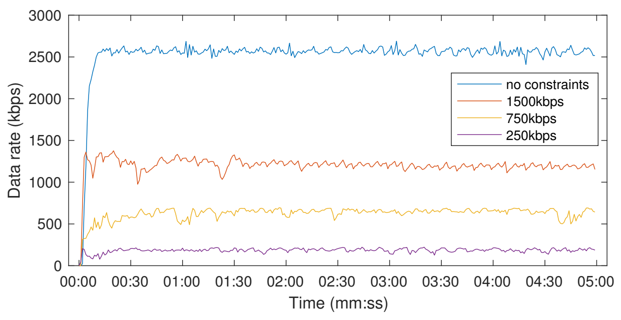

Figure 5: Data rate with limited bandwidth and without any constraints (100Mbps or more available bandwidth).

Our evaluation of WebRTC is divided into two parts. In the first part, we emulate synthetic network conditions to study the performance of WebRTC in controlled settings. In the second part, we focus on experimental evaluation on real networks and particularly focus on wireless networks. The experimental evaluation setup for two users is shown in Figure 4.

For the first part, we emulate different network characteristics using Dummynet [6], which allows us to add latency, packet loss, and limit the bandwidth for both uplink and downlink. To avoid additional latency and network limitations, we connect both WebRTC endpoints to the same local network via a wire.

In all of our experiments, we use devices with sufficient processing and memory capacity to ensure that the encoding and decoding of the video streams are not affected due to the devices themselves. To ensure this, we leverage WebRTC's RTCStatsReport API functionality which indicates if the video quality is limited due to memory or computation power at the devices. Unless mentioned otherwise, we use the most recent version of WebRTC (supported by Google Chrome version 52 and onwards) at all clients, with the default audio and video codecs OPUS and VP8, respectively. Instead of using a webcam feed and microphone audio signal, we exploit Google Chrome's fake-device functionality to feed the browser a looping video and audio track to obtain comparable results. For all our tests (unless mentioned otherwise), we use the following video with a resolution of 1920x1080 at 50 frames per second with constant bitrate: *in_to_tree*[2].

To obtain performance metrics, we use WebRTC's built-in *RTCStatsReport*[3], which contains detailed statistics about data being transferred between the peers.

4. SYNTHETIC NETWORK CONDITIONS

In this section we evaluate the performance of WebRTC's GCC algorithm in synthetic yet typical network scenarios using Dummynet.

4.1 Static network conditions

Figure 5 shows the results for the cases when both the uplink and downlink bandwidth are limited to 1500Kbps, 750Kbps, and 250Kbps. We notice that WebRTC is currently limited to sending at 2500Kbps, as set in the browser[4]. When the bandwidth is limited, it uses 80% of the available bandwidth and is able to maintain a constant data rate. By continuously lowering the available bandwidth in additional experiments, we observed that a minimum of 20Kbps is necessary to establish a video call between two parties. However, at least 250Kbps of available bandwidth is necessary to obtain a somewhat acceptable frame rate (25 Frames Per Second (FPS)) at the lowest possible video resolution (480x270). It does take longer to reach the maximum data rate, especially when we look at the 250Kbps, where it takes approximately 10 seconds for any data to flow between both nodes.

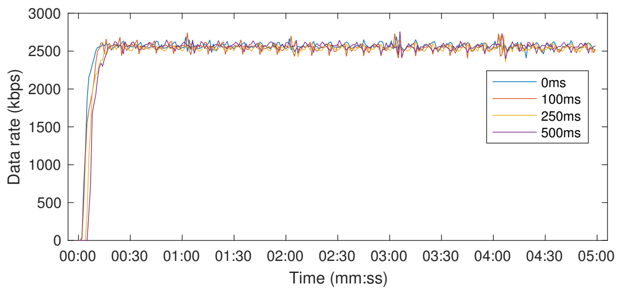

Figure 6: Data rate with additional latency.

Next, we add extra latency to the call, as shown in Figure 6. As expected, this does not affect the data rate, since the GCC algorithm only responds to latency variation. However, it leads to delays in the conversation. ITU-T Recommendation G.114 [1] specifies that one-way transmission delay should preferably be kept below 150ms, and delays above 400ms are considered unacceptable. When adding delay, we also observe that it takes longer to set up the call and for data to flow between both end points, which negatively affects user experience. Once data flows, it takes approximately 10 seconds to reach its maximum data rate, regardless of the added delay. This delay is less than what is expected from the GCC equations where the rate would increase with 5% as shown in equation (6). This is because once a connection is established, WebRTC uses a *ramp-up function*[5] to get to the highest possible data rate as soon as

[2]https://media.xiph.org/video/derf/
[3]https://developer.mozilla.org/en-US/docs/Web/API/RTCStatsReport
[4]https://chromium.googlesource.com/external/webrtc
[5]https://bugs.chromium.org/p/webrtc/issues/detail?id=1327

possible.

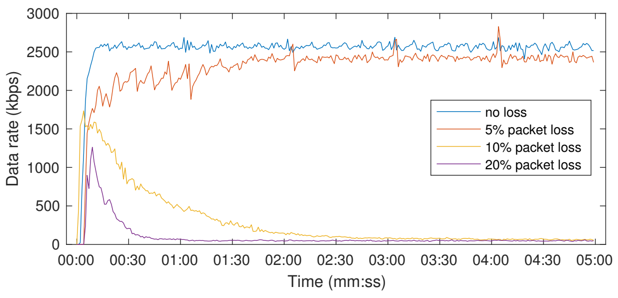

Figure 7: Data rate with packet loss.

For the next experiment we drop a certain percentage of all packets: 5%, 10%, and 20%. The results are shown in Figure 7. The results match our expectations based on Equation (6). GCC only decreases the sending rate when more than 10% packet loss is detected. The sending rate remains unchanged between 2% and 10% and the rate is increased when less than 2% of the packets are lost. Therefore, 5% packet loss slowly converges to the maximum data rate and at 10% packet loss, the data rate converges to a minimum of 50Kbps, which almost completely consists of audio data (the audio stream is not subject to congestion control by GCC due to its low data rate [12]).

4.2 Network adaptability

Besides experiencing a constant delay or being limited by a constant bandwidth, a more common scenario is that these network characteristics change during a call. In this section, we look at how fast WebRTC adapts to new conditions. We simulate this behavior by changing the network constraints every minute according to a predefined schema.

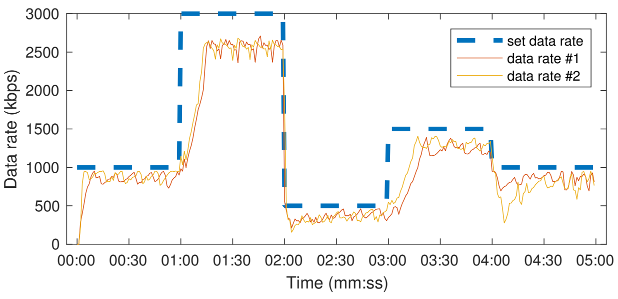

Figure 8: Data rate with changing bandwidth for both nodes.

In Figure 8, we cap the available bandwidth for a minute consecutively at 1000Kbps, 3000Kbps, 500Kbps, 1500Kbps and 1000Kbps. In this scenario, the bandwidth utilization is 77% of the available bandwidth, which is slightly less than the 80% bandwidth utilization when the available bandwidth is not changed. This is mostly due to the delay it takes to reach a steady bandwidth when more bandwidth becomes available at minute 1:00 and 3:00 where, respectively, 16 and 18 seconds are needed. As seen in Equation (1), this delay confirms what we expect from GCC, since the bandwidth increases linearly with a factor 1.05 when underuse is detected. This is because REMB messages are sent every second, which increase the bandwidth with 5% every second. Theoretically, we would expect a rate of 1000Kbps $\times 1.05^{16} \approx 2200$Kbps after the first minute and 500Kbps $\times 1.05^{18} \approx 1200$Kbps after bandwidth is freed at the third minute, both close to the respectively reached 2500Kbps and 1350Kbps.

Table 2: Changing latency sequence.

Minute	Latency change (from - to)	Steepness
0 - 1	0ms - 250ms	exponential
1 - 2	250ms	N.A.
2 - 3	250ms - 0ms	linear
3 - 4	0ms - 500ms	linear
4 - 5	500ms - 0ms	exponential

As shown in Section 4.1, WebRTC's congestion algorithm does not respond directly to different latencies, but changes its data rate based on latency variation. Therefore, we gradually change the latency at 0.5 seconds intervals with both linear and exponential functions as shown in Table 2 and Figure 9 (bottom). The resulting data rate is shown in Figure 9 (top).

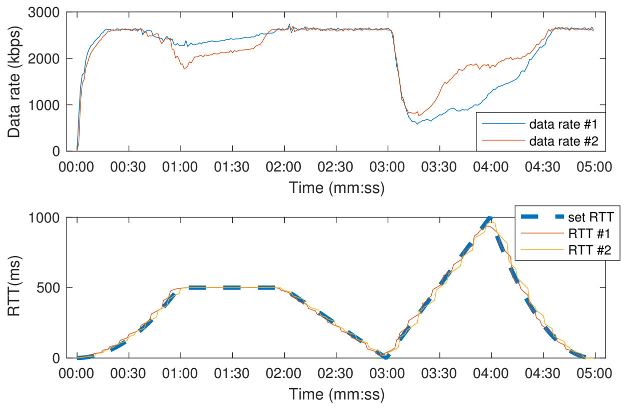

Figure 9: Data rate and resulting RTT when continuously changing the latency for both nodes.

The actual round trip time is close to the set data rate. Unlike other experiments, we notice that the data rate is significantly different for both parties even though additional latency is added in both directions. As expected, the data rate climbs up to the maximum data rate when latency is decreased (at minute 2 and 4) or kept constant (minute 1). More unexpectedly, the GCC does not seem to kick in until after 40 seconds even though the RTT is increasing exponentially. This is presumably due to ramp-up function described earlier, which allows WebRTC to reach the maximum data rate faster. We observe that GCC responds actively to the RTT transition at minute 3, where a decreasing and subsequently increasing RTT results in a large drop in data rate.

In addition to studying the effect of different packet loss values, we also consider how packet loss that changes during the lifespan of a call affects the call characteristics. Here we change the packet loss every minute from 10%, 0%, 5%, 7.5% and 15% as shown in the bottom graph of Figure 10. The resulting data rate is shown in Figure 10. The results are

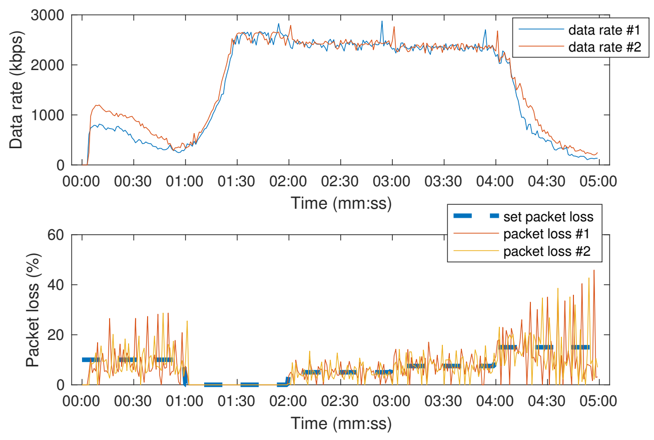

Figure 10: Data rate and packet loss while changing packet loss for both nodes.

comparable to what we observed in Section 4.1. A packet loss of 5% and 7.5% only slightly drop the data rate (minute 2 and 3), whereas a packet loss >= 10% reduces the data rate significantly. It takes approximately 30 seconds to reach the maximum data rate when packet loss is removed after the first minute. This is consistent with our expectations according to the 5% increase in data rate when packet loss is less than 2% (equation (6)), as set by GCC. The data rate increases with 5% every second for 30 seconds, coming down to 550Kbps $\times 1.05^{30} \approx$ 2400Kbps, which is close to the reached 2500 Kbps shown at minute 1.

Lastly, we study the effects of changing both the latency and the available bandwidth. We simulate the effect of hand-off, which for instance occurs when a cellular receiver moves from one Base Station to another. For this experiment, we also limit the available uplink and downlink bandwidth differently, since it is common for the uplink rate to be lower than the downlink counterpart. The experimental procedure is shown in Table 3.

Table 3: Experiment procedure for changing both latency, uplink and downlink bandwidth.

Minute	Round trip time	Downlink	Uplink
0 - 1	60ms	3000Kbps	3000Kbps
1 - 2	200ms	750Kbps	250Kbps
2 - 3	500ms	250Kbps	100Kbps
3 - 4	150ms	1250Kbps	500Kbps
4 - 5	200ms	750Kbps	250Kbps

The resulting data rates and latencies are shown in Figure 11. We notice that the bandwidth utilization is 69% of the available bandwidth, which is significantly lower than the 77% bandwidth utilization (Figure 8) when there is no additional latency. The limited bandwidth also results in additional latency, especially when the bandwidth is extremely limited (250Kbps downlink / 100Kbps uplink) at minute 2 when the RTT increases to more than 2 times the value it was set.

4.3 Cross traffic

WebRTC traffic competes with cross traffic when there

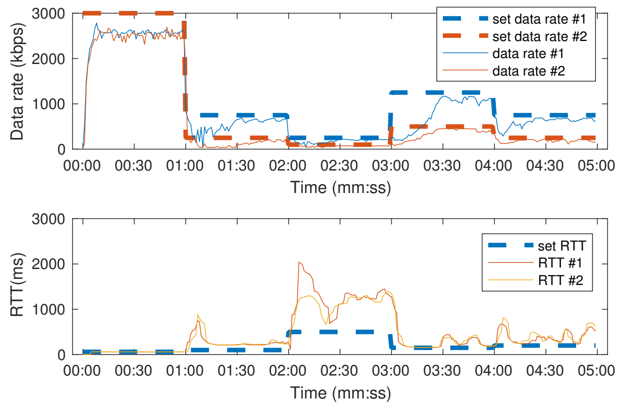

Figure 11: Resulting data rates and Round-Trip-Time for the experiment in Table 3 for both nodes.

are other TCP/UDP flows active that share the same bottleneck. It has been shown in previous measurement studies that in the presence of concurrent TCP flows, WebRTC's UDP streams could starve due to less aggressive congestion control [10].

Recently, Google Congestion Control has been updated to include an adaptive threshold (γ), with the aim of guaranteeing fairness when competing with concurrent flows [10]. In this section, our goal is to evaluate the impact of the adaptive threshold on fairness. We first evaluate the performance of a single WebRTC stream which is competing with other WebRTC streams while sharing the same bottleneck link. Next, we conduct the same test with competing TCP flows.

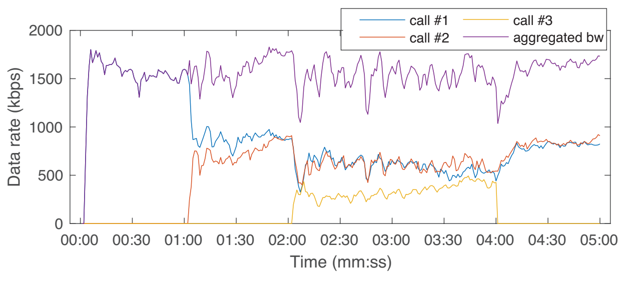

Figure 12: Distribution of bandwidth across three WebRTC calls.

We first limit the available bandwidth to 2000Kbps and test how the available bandwidth is distributed when three WebRTC calls share this bottleneck. To test how fast the congestion control algorithm adapts, we stagger the start time of calls. We start with one call, add a second call after one minute, and add a third call after 2 minutes. To see how fast WebRTC adapts once bandwidth is freed, we drop the third call in minute 4. The results of this experiment are shown in Figure 12. The cumulative data rate is 78%, which is comparable to our earlier measured bandwidth utilizations (Figures 5 and 8). We see that the data rate momentarily drops when a new stream enters or leaves the bottleneck (minute 01:00, 02:00 and 04:00). The data rates converge subsequently to their fair share value, but the time duration for convergence is almost a minute when two streams compete and even longer with 3 streams. The Jain Fairness Index values in the case of two streams and three streams are 0.98 and 0.94, respectively. Since both scores are close to 1, fairness is maintained.

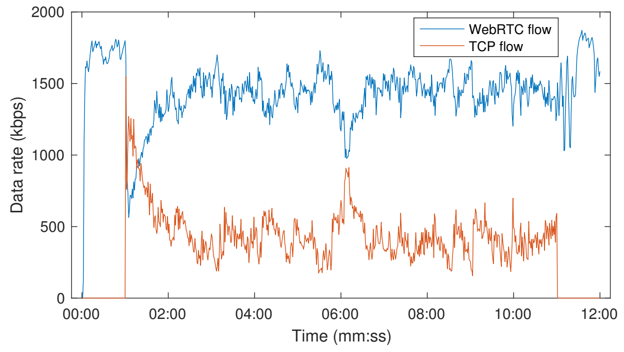

Figure 13: Distribution of bandwidth when a single RTP flow competes with a TCP flow when bandwidth is limited to 2000Kbps.

To study the effects of TCP cross traffic, we generate TCP flows using *iperf*[6]. We limit the bandwidth to 2000Kbps, initiate a 12 minute call between two nodes, and introduce a ten-minute competing TCP flow at minute 01:00. The results are shown in Figure 13. Surprisingly, WebRTC's RTP flow has a significantly higher average data rate from 01:00 - 11:00 compared to the TCP flow (on average 1408Kbps vs. 451Kbps) with a resulting Jain Fairness Index of 0.79. The newly introduced adaptive threshold provides better fairness and WebRTC's RTP flows no longer starve when competing with TCP flows. However, optimal fairness is not achieved and the adaptive threshold prioritizes WebRTC's RTP flows more aggressively than desired.

4.4 Multi-party topology comparison

In this section, we compare the performance of several topologies that can be used for multi-party video conferencing. We evaluate 2-person, 3-person, and 4-person video conferencing for these topologies.

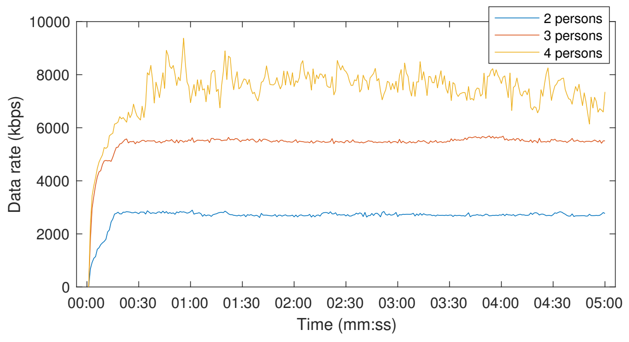

Figure 14: Average data rates for 2-, 3-, and 4-person meshed calls.

In a meshed network, every participant uploads its stream $n-1$ times and downloads the other $n-1$ streams directly from the other peers, where n equals the number of participants. The results for 2-, 3-, and 4-person meshed calls are shown in Figure 14. The data rates in this graph show both the average uplink and download data rates. The rates for 3-person calls are close to two times the rates of 2-person calls (factor 2.03). Surprisingly, 4-person calls have less than 3 times the rate compared to 2-person calls (factor 2.77), mostly due the long startup delay. The rate is also very volatile compared to the other calls, which maintain a constant data rate even though we averaged out several 4-person calls. This volatile behavior is due to CPU limitations, because every person needs to simultaneously both encode its own video stream three times and decode the three incoming streams.

[6]https://github.com/esnet/iperf

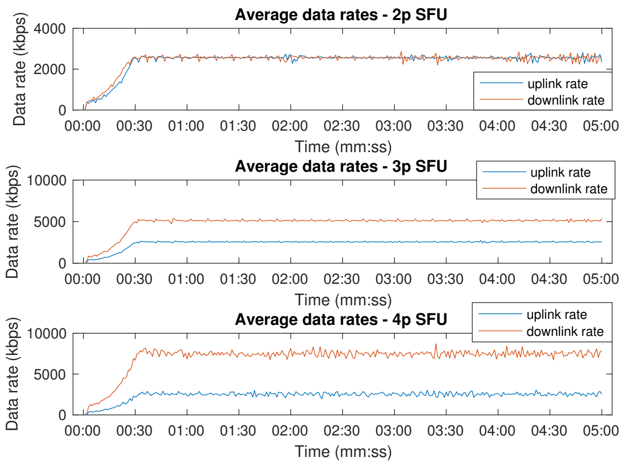

Figure 15: Average data rates for 2-, 3-, and 4-person calls using a SFU.

By introducing an extra server to forward the streams, we can reduce the necessary uplink bandwidth. By utilizing a server as a Selective Forwarding Unit (SFU), all the participants only have to upload their stream once and the SFU forwards these to the other participating clients. This approach introduces extra latency, because streams are relayed, but significantly reduces both CPU usage (for encoding all streams) and necessary uplink bandwidth. The results are shown in Figure 15. Compared to meshed calls, it takes significantly longer to reach a stable data rate (30 seconds vs. 15 seconds). For a 3-person call, the average downlink rate is 2.00 times higher than the uplink rate. For a 4-person call, the downlink rate is 2.95 times higher.

5. IMPACT OF VIDEO CODECS AND MOBILE DEVICES

In this section, we experimentally study the impact of different video codecs and mobile devices on video call quality over WebRTC. We use synthetic network conditions similar to the previous section to generate common network conditions for our evaluations.

5.1 Video codec comparison

By default, Google Chrome utilizes the VP8 video codec for WebRTC video calls. Recent versions of Google Chrome (starting with v48) provide support for the more advanced VP9 codec. VP9 is expected to provide the same objective video quality as VP8, but at a lower bitrate, due to its more efficient compression efficiency [18]. This, however, comes at the expense of extra CPU power. VP9 is therefore useful when bandwidth is limited (e.g., cellular and congested networks). Support for the H.264 video codec has also been added in Chrome v50, which allows hardware-accelerated decoding for many devices.

Even though these newly supported codecs are not used by default, WebRTC can be instructed to use them by altering the generated Session Description Protocol (SDP). In this section, we compare the VP8, VP9, and H.264 codecs. Since these newly added codecs are still under development, we use the 720p variant of the video to prevent CPU limitations while encoding/decoding. We conducted experiments with varying network conditions as described in Table 4. We limit the tests to changing only one parameter each minute. The results are shown in Figure 16 and Table 5.

Table 4: Mobile performance evaluation procedure.

Minute	Round trip time	Data rate	Packet loss
0 - 1	0ms	∞	0%
1 - 2	0ms	1250Kbps	0%
2 - 3	0ms - 500ms - 0ms	1250Kbps	0%
3 - 4	0ms	∞	0%
4 - 5	0ms	∞	15%
5 - 6	0ms	∞	0%
6 - 7	0ms - 500ms	1250Kbps	0%

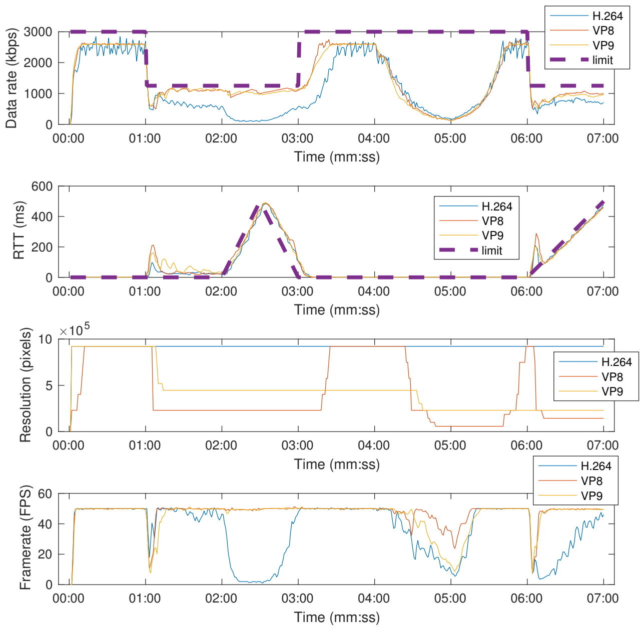

Figure 16: Average data rates, RTT, resolution, and framerate for different video codecs when the network varies according to Table 4.

Table 5: Average call characteristics for different video codecs.

	VP8	VP9	H.264
Data rate (Kbps)	1439.7	1422.5	1154.2
RTT (ms)	83.6	84.2	77.6
Framerate (FPS)	47.4	45.5	36.1
Packet loss (%)	2.51	2.44	2.43
Resolution (pixels)	858x483	892x502	1279x719

As shown in Table 5, the data rate of H.264 is more heavily affected by the limited bandwidth and the added latency when compared to VP8 and VP9. H.264 also differs in the way it uses its data rate. While VP8 and VP9 balance between frame rate and resolution, H.264 only lowers the video frame rate, while maintaining a constant maximum resolution which could be because it is hardware accelerated and thus depends more on its browser implementation. As shown in Figure 16, this causes the frame rate to drop to an unusable value 1FPS around the 2:30 minute mark. As expected, VP9 outperforms VP8 when congestion occurs, due to its more efficient compression capabilities. This can be seen from the higher resolutions from minute 1 to 3. When the congestion is removed at minute 3 or minute 5, VP9, however, does not scale back up to the original resolution (1280x720), while VP8 does which might be because VP9 is fairly new and not yet optimized.

5.2 Mobile devices

For the mobile performance evaluation, we perform seven-minute experiments covering all different network variations. Since we cannot inject a custom video stream for mobile devices, we fall back on using the camera of the mobile devices and force the use of the rear-camera to generate a higher quality stream. Unfortunately, Safari does not support WebRTC on iOS. Therefore, we use the Cordova plugin[7] to handle the WebRTC natively, while using the same performance measurement framework. The experimental parameters change according to Table 4.

The different call characteristics of the test described in Table 4 are shown in Figure 17 and Table 6. The video on iOS and Android mobile platforms is limited to a resolution of 640x480 at 30FPS, even though their cameras are able to handle higher resolutions. Furthermore, iOS and Android behave similarly across all characteristics. Their data rates are both significantly less than Chrome when there is no congestion (1750Kbps vs. 2500Kbps) and their average RTT is much higher. It also takes longer for both mobile platforms to reach the maximum data rate when compared to Chrome (20 seconds vs. 10 seconds).

Table 6: Average call characteristics for different platforms.

	Chrome	iOS	Android
Data rate (Kbps)	1237.8	1022.5	1047.3
RTT (ms)	80.0	95.4	100.0
Framerate (FPS)	42.96	27.9	27.8
Packet loss (%)	2.30	2.18	2.2
Resolution (pixels)	1006x566	602x339	675x380

[7]https://github.com/eface2face/cordova-plugin-iosrtc

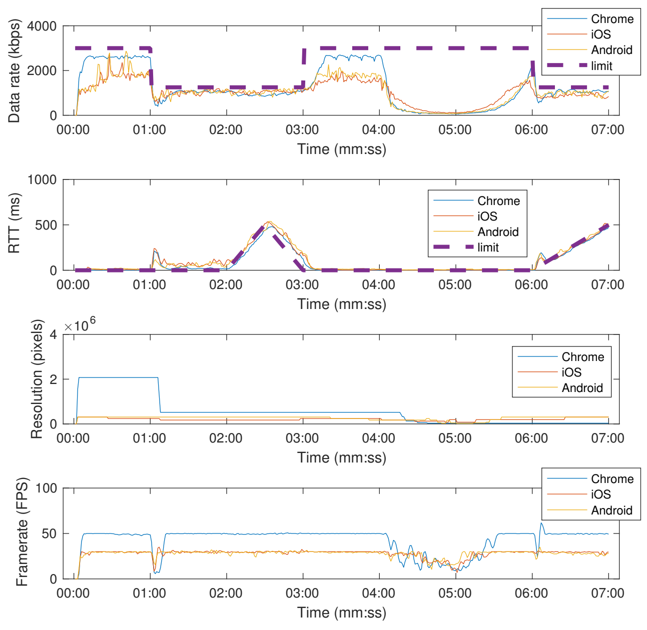

Figure 17: Average data rates, RTT, resolution, and frame rate for iOS, Android and Chrome network when the network varies as shown in Table 4.

6. WIRELESS PERFORMANCE

In this section, we evaluate the performance of WebRTC over real networks. We specifically focus on studying the impact of a WiFi hop on WebRTC.

6.1 Benchmarking

In Section 4, we observed that GCC is sensitive to changes in latency and packet losses. Transmitting over wireless networks may result in bursty packet losses and dynamic latencies due to subsequent retransmissions, especially if the end-to-end Round Trip Time (RTT) of the WebRTC connection is large. In this section, we characterize the effects of wireless links on the performance of WebRTC by comparing against the performance on wired links.

We consider 3 types of WebRTC nodes: (i) a local wireless node, (ii) a local wired node, and (iii) remote wired nodes. We used a 2013 ASUS Nexus 7 tablet as a local wireless node connected to an IEEE 802.11 DD-WRT enabled Access Point (AP). The wired node is either a local machine located in our lab in New York City or a remote server running in Amazon EC2 cloud. We consider two cases for the remote server: one in the AWS Oregon availability zone and one in the AWS Sydney availability zone which provide different magnitudes of RTT. This allows us to study the impact of higher RTT as compared to the local machine.

Both the local and remote machines run Ubuntu 14.04 with Google Chrome 57.0 as the browser. We use the same injected video files for a fair comparison. Moreover, all the machines have sufficient computational power to eliminate the impact of devices on video performance. A virtual display buffer was used on the EC2 servers to run WebRTC on Chrome in headless mode. For the wireless node, we used 5GHz channels to minimize the interference from other IEEE 802.11 networks. To emulate the conditions of high loss environments, the AP transmission power was set to 1mW. We experiment with different channel conditions with the wireless node being in the same room as the AP (approximately 5 feet away), as well as outside of the room (approximately 25 feet away).

Table 7 shows average call statistics for two fully-wired calls with one wired node located in the NYC area in the lab and the other node in Oregon or Sydney. The NYC node was injecting a video encoded at 50FPS, and the remote nodes were using a video encoded at 60FPS. The average RTTs for the Oregon and Sydney calls were 77.74ms and 214.86ms, respectively. Accordingly, we term these scenarios as "medium" and "high" call latencies as compared to "short" latency scenario with both nodes in the NYC area. These results establish a baseline performance of WebRTC in realistic network conditions.

Table 7: Baseline statistics of wired calls with differing RTTs.

Call Path	Data Rate	Frame Rate	Frame Width
NYC to Sydney	2971.11	49.58	1278.39
Sydney to NYC	2352.66	58.51	1280.00
NYC to Oregon	3001.45	49.68	1280.00
Oregon to NYC	2305.43	58.47	1242.83

Next, we perform video calls with one wireless node and the other node either being a local wired node or one of the two remote nodes. A 720p video encoded in 50FPS was used across all 3 cases. On the wireless node, the camera on the Nexus tablet was used as video source, because video could not be injected into the Android distribution of Chrome without rooting the device.

Figure 18 depicts the data rate, frame rate, frame width, and the RTT for a single call with high latency between a server located in Oregon area and a wireless node in the lab. For comparison, we also show the performance of a fully wired call in a similar scenario. Adding a wireless hop in typical indoor conditions creates a significant change in RTT characteristics. We observe that the peaks in RTT at 20, 30, and 50 seconds correspond to drops in frame rates, which lead to poor video quality for the user. Furthermore, we observe these RTT peaks to persist even after frame rates and data rates drop.

A comparison of packet inter-arrival times between a wired and wireless call is shown in Figure 19. Further, Figure 19 effectively illustrates how the wireless hop changes the delay variation d_i (according to (3)) used by GCC's arrival-time filter[8]. In all our experiments, the number of packet losses was relatively low (packet losses are handled by retransmissions). Thus, the large variation in packet inter-arrival times generally results in variations in video quality since GCC relies on packet inter-arrival times for congestion control.

Figure 20 shows performance results of experiments for the near and far scenarios. Although the calls are two-way, the figures depict call performance statistics for the data received at the wireless node. Each result is an average of four identical experiments of 200 seconds each. Increasing

[8]The impact of packet inter-departure time is minimal and we exclude that from our calculations.

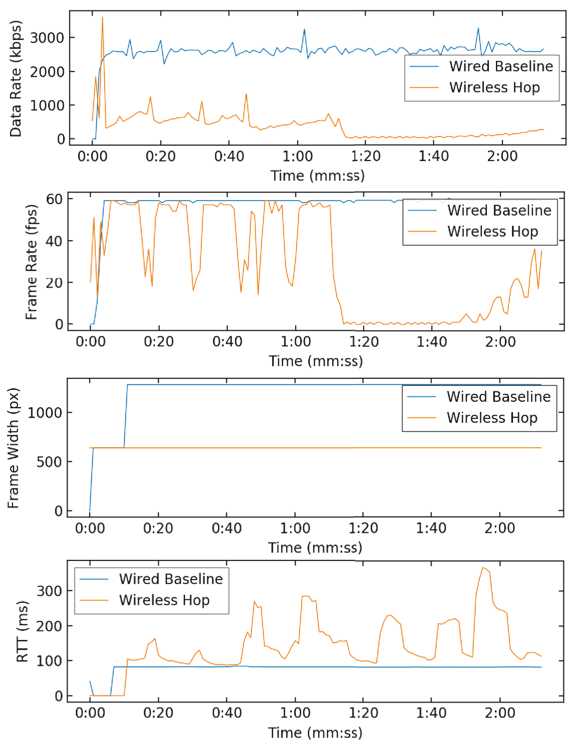

Figure 18: Experimental results for a call between NYC and a remote server located in the AWS Oregon region. A fully wired call is compared with a wireless hop call.

the distance from the access point generally increased RTTs as well as packet loss. The received frame rate and frame resolution reduced as well. With higher RTTs, the impact of wireless link is more apparent. For instance when the remote server is in Oregon, the average frame rate and frame widths for the case of wireless node far from the AP are more than 25% lower than when wireless node is near the AP. This difference is approximately 10% when the wired node is located in the lab and the RTT is small.

In summary, our experiments characterize the performance of WebRTC video calls on both wired and wireless networks. We observed that bursty losses and retransmissions can degrade the call quality, especially when the end-to-end RTT is long. In the next section, we briefly explore cross-layer techniques to enhance the performance of WebRTC over wireless. More specifically, we study the tradeoff between higher packet losses and lower packet inter-arrival times by adjusting the wireless MAC layer retry limit.

6.2 Impact of MAC retry limits

In lossy wireless environments, when the wireless node is far from the AP, we observed that the video stream frequently freezes. The typical duration of such freezes is a few seconds and subsequently, the stream resumed at a much lower frame rate and resolution. In the traces of these calls, as shown in Figure 18, we observed multiple spikes in RTT values, where the RTT would quickly rise to 2x or 3x the previous value before dropping back down again. These spikes occurred throughout the duration of the call despite changes in other call parameters.

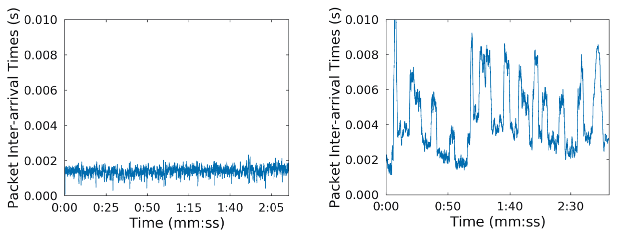

Figure 19: Comparison of time delta characteristics for a wired call (left) and a call with a wireless hop (right).

To better understand this variation in RTT, we inspected the Wireshark traces of the call experiments obtained on a separate device, placed near the AP, and operating in monitor mode. We used Wireshark to decode the traces as RTP streams and observed a high number of packet retransmissions, typically in immediate succession. These packet retransmissions lead to spikes in RTT which subsequently results in poor video quality. Our objective is to identify if reducing the number of retransmissions at the expense of higher packet losses may improve video quality. We note that recent papers have explored cross-layer retry limit adaptation mechanisms for latency-sensitive applications, such as OpenSDWN [21] as well as approaches proposed in [26] and [9].

The DD-WRT enabled AP provides parameters to control the *Long Retry Limit* for data packets between values of 1 and 15. We set the Long Retry Limit to two extreme values of 1 and 15 and compared the impact of the AP automatically configuring the retry limit when the wireless node was located far from the AP. Furthermore, we evaluated retry limit and access point proximity combinations across the local wired node as well as the two remote nodes to achieve low, medium, and high baseline RTT magnitudes. All nodes used the same injected 720p video at 50FPS.

Figure 23 depicts the differences in call performance for the highest baseline latency with the retry limit set to the maximum of 15, as well as the minimum of 1. We observe that disabling MAC layer retries (by setting the limit to 1) reduces the variation in RTTs at the expense of higher packet losses. Figures 23(a) and 23(b) show the average frame rate and average frame width for different retry limits. Disabling retransmissions leads to reduced values for both which leads to very poor video quality. With retries disabled, RTT variations are significantly reduced and closely resemble the RTT characteristics of the fully wired calls as depicted in Figure 18. Packet losses are much higher in the wireless call with retries disabled than they are in the virtually lossless wired baseline in Figure 18. Qualitatively, we observed the video freezes less frequently.

We observed a trade-off between RTT variation and packet losses when controlling MAC retransmissions in lossy wireless environments. This trade-off is visually depicted in Figure 21. On both extremes of this trade-off, however, we found that call quality still suffers, as GCC responds heavily to packet losses.

Since GCC uses packet inter-arrival times as well as packet loss information, there may be room for further modifications that would allow GCC to exploit cross-layer tech-

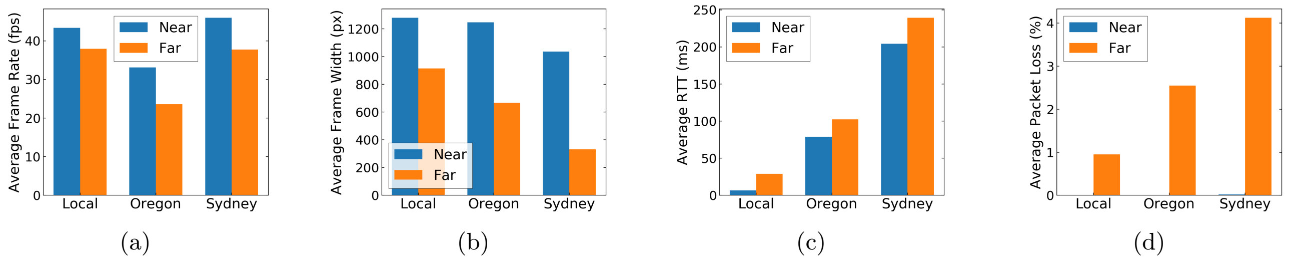

Figure 20: Experimental results for calls with the wireless node at a static position near (same room) and far (outside of room) from the AP: (a) the average frame rate, (b) average frame width, (c) average RTT, and (d) packet loss.

niques such as MAC layer retransmits or PHY layer rate adaptation. In WebRTC's current implementation, however, we observe that GCC is too sensitive to packet loss to benefit from MAC-layer retransmission adaptation. A future direction is to study the impact of lowering PHY layer transmission rates to guarantee successful packet delivery at the expense of reduced bandwidth. Figure 22 shows the PHY transmission rate when the wireless node is near or far from the AP. The PHY transmission rate is usually higher than the minimum transmission rate (6 Mbps). Reducing the PHY transmission rate may reduce the number of packet losses while still ensuring sufficient bandwidth for the WebRTC call.

7. RELATED WORK

Performance evaluation and design of congestion control algorithms for live video streaming have received considerable attention. Below, we highlight the most relevant work.
Congestion control for multimedia: TCP variants such as Tahoe and Reno [16] have shown to lead to poor performance for multimedia applications since they rely only on losses for congestion indication. The approaches to address the shortcomings of these techniques can be divided in two categories.

The first variety of congestion control algorithms use variants of delay to infer congestion. Delay based variants of TCP such as Vegas [5], and FAST [24] rely on measuring round trip delays but they are more reactive than proactive in congestion control. LEDBAT [22] relies on measuring one way packet delays to ensure high throughput while minimizing delays. Sprout [25] utilizes stochastic forecasts of cellular network performance to achieve the same goals. The second category of congestion control relies on Active Queue Management (AQM) techniques. NADA [27] uses Explicit Congestion Notifications (ECN) and loss rate to obtain an accurate estimate of losses for congestion control.
WebRTC congestion control: SCReAM [17] is a hybrid loss and delay based congestion control algorithm for conversational video over LTE. FBRA [19] proposes a FEC-based congestion control algorithm that probes for the available bandwidth through FEC packets. In the case of losses due to congestion, the redundant packets help in recovering the lost packets.
WebRTC performance evaluation: Several papers have studied the performance of WebRTC. Most related work focuses on a single aspect of the protocol or use outdated versions of WebRTC in their performance analyses. [2] analyzes the Janus WebRTC gateway focusing on its performance and scalability only for audio conferencing in multiparty calls. [8] focuses on comparison of end-to-end and AQM-based congestion control algorithms. [7] evaluates the performance of WebRTC over IEEE 802.11 and proposes techniques for grouping packets together to avoid GCC's action on bursty losses.

[10] presents the design of the most recent version of the GCC algorithm used in the WebRTC stack. While [10] provides preliminary analysis of GCC in some synthetic network conditions, it does not focus on WebRTC's performance on mobile devices or real wired and wireless networks. Its main focus is on inter-protocol fairness between different RTP streams and RTP streams competing with TCP flows.

[23] provides an emulation based performance evaluation of WebRTC. However, all flaws identified in [23] have been subsequently addressed in WebRTC. For instance, the data rate no longer drops at high latencies (but instead responds to latency variation), the bandwidth sharing between TCP and RTP is fairer due to the newly introduced dynamic threshold, and the available bandwidth is shared more equally when competing RTP flows are added.

A more realistic performance study using real network effects is done in [13], where the performance of WebRTC is measured with mobile users in different areas. Even though the WebRTC implementation used is outdated, the paper suggests that WebRTC's over-reliance on packet loss signals leads to under-utilization of the channel due to mobility.

8. LESSONS LEARNED

We believe that our evaluation and insights derived from it can serve as a useful guide for developers of applications leveraging WebRTC. While we have done an extensive evaluation of the performance of GCC and WebRTC in a wide variety of environments, there are several open issues and directions for future research.

The new changes in the GCC algorithm include an adaptive threshold for congestion control. Our evaluations show that this ensures better fairness between competing WebRTC's RTP and TCP flows than reported in earlier studies. However, optimal fairness is still not achieved and the adaptive threshold prioritizes WebRTC's RTP flows more aggressively than desired.

We compared the performance of a mesh and Selective Forwarding Unit (SFU) based topologies for group video

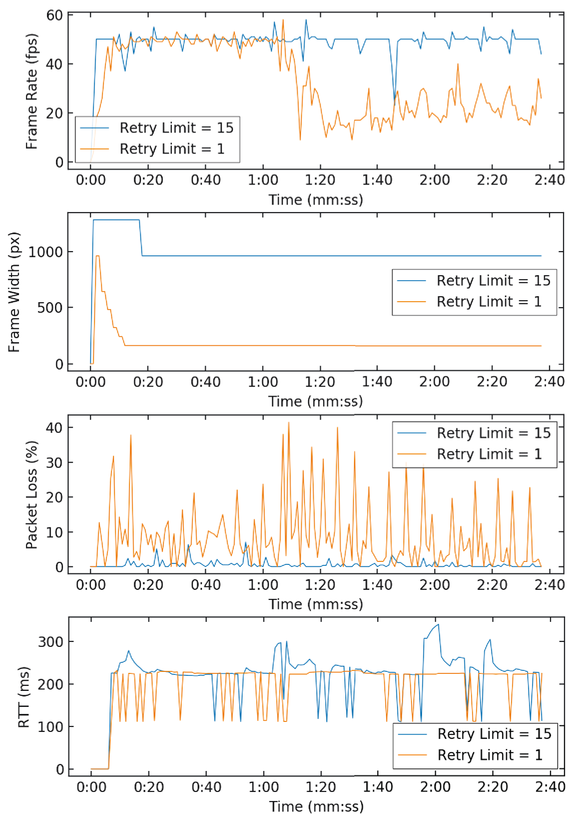

Figure 21: Experimental results for a wireless call between NYC and Sydney with high (above) and low (below) MAC layer retry limits.

calls using WebRTC. Our evaluation shows that adding an SFU can significantly improve the performance of multi-party video call. The positioning and dimensioning of SFU in the network are some interesting future research directions.

Our experiments demonstrated that the newly added H.264 and VP9 codecs do not perform as expected in the presence of congestion or packet losses. It is not immediately clear if this performance issue is due to codec design or an implementation flaw and requires further investigation.

We experimentally evaluated video calls on WebRTC in real networks, specifically focusing on wireless networks. Our experiments show that WebRTC can suffer from poor performance over wireless due to bursty losses and packet retransmissions.

In future work, we will consider modifications to the GCC algorithm to improve its performance with bursty packet losses and large variations in RTT. Further, we will study more complex cross-layer approaches to address the performance issues of WebRTC over wireless, including PHY-layer rate adaptation and dynamic adaptation of retransmission limits along with congestion control.

9. CONCLUSION

In this paper, we evaluated the performance of WebRTC-based video conferencing, with the main focus being on the Google Congestion Control (GCC) algorithm. Our evaluations in synthetic, yet typical, network scenarios show that

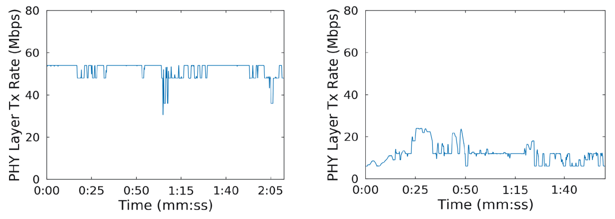

Figure 22: Comparison of PHY data rate for tablet positioned "Near" (left) and "Far" (right) from the AP.

WebRTC is sensitive to variations in RTT and packet losses. We also evaluated the impact of different video codecs, mobile devices, and topologies on WebRTC video calls. Further, our evaluations on real wired and wireless networks show that bursty packet losses and retransmissions over long RTTs can especially lead to poor video performance. The source code for setting up and evaluating the experimental environments described in this paper is available at: `https://github.com/Wimnet/webrtc_performance`.

10. ACKNOWLEDGEMENTS

We would like to thank Rodda John, Columbia University, for his help in implementing scripts to analyze wireless performance data. This work was supported in part by NSF grants CNS-1423105 and CNS-1650685.

11. REFERENCES

[1] One-way transmission time. *ITU-T, G.114* (May 2003).

[2] Amirante, A., Castaldi, T., Miniero, L., and Romano, S. P. Performance analysis of the janus webrtc gateway. In *Proc. ACM AWeS'15* (2015).

[3] Ammar, D., De Moor, K., Xie, M., Fiedler, M., and Heegaard, P. Video QoE killer and performance statistics in WebRTC-based video communication. In *Proc. IEEE ICCE'16* (2016).

[4] Bergkvist, A., Burnett, D. C., Jennings, C., Narayanan, A., and Aboba, B. Webrtc 1.0: Real-time communication between browsers. online, 2016. `http://www.w3.org/TR/webrtc/`.

[5] Brakmo, L. S., and Peterson, L. L. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE J. Sel. Areas Commun. 13*, 8 (1995), 1465–1480.

[6] Carbone, M., and Rizzo, L. Dummynet revisited. *SIGCOMM Comput. Commun. Rev. 40*, 2 (2010), 12–20.

[7] Carlucci, G., De Cicco, L., Holmer, S., and Mascolo, S. Making Google congestion control robust over Wi-Fi networks using packet grouping. In *Proc. ACM ANRW'16* (2016).

[8] Carlucci, G., De Cicco, L., and Mascolo, S. Controlling queuing delays for real-time communication: the interplay of E2E and AQM algorithms. *ACM SIGCOMM Computer Commun. Rev. 46*, 3 (2016).

[9] Chen, W., Ma, L., and Shen, C.-C. Congestion-aware MAC layer adaptation to improve

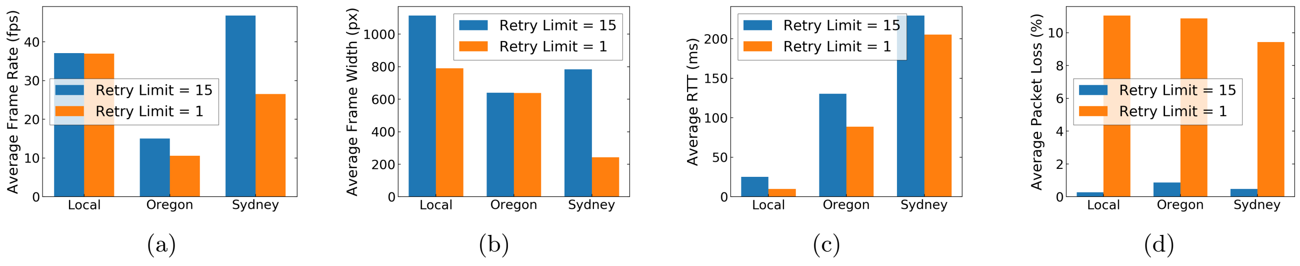

Figure 23: Experimental results for calls with MAC layer retry limits varied between the maximum and minimum values on the AP: the average (a) frame rate, (b) frame width, (c) RTT, and (d) packet loss.

video telephony over Wi-Fi. *ACM Trans. Multimedia Comput. Commun. Appl. 12*, 5s (2016), 83:1–83:24.

[10] Cicco, L. D., Carlucci, G., Holmer, S., and Mascolo, S. Analysis and design of the google congestion control for web real-time communication (WebRTC). In *Proc. ACM MMsys'16* (2016).

[11] Cicco, L. D., Carlucci, G., and Mascolo, S. Understanding the dynamic behaviour of the google congestion control for RTCWeb. In *Proc. IEEE PV'13* (2013).

[12] De Cicco, L., Carlucci, G., and Mascolo, S. Experimental investigation of the google congestion control for real-time flows. In *Proc. ACM SIGCOMM FhMN'13* (2013).

[13] 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 *Proc. PV'13* (2013).

[14] Hardie, T., Jennings, C., and Turner, S. Real-time communication in web-browsers. online, 2012. `https://tools.ietf.org/wg/rtcweb/`.

[15] Homer, S., Lundin, H., Carlucci, G., Cicco, L. D., and Mascolo, S. A Google congestion control algorithm for real-time communication. IETF draft, 2015. `https://tools.ietf.org/html/draft-ietf-rmcat-gcc-01`.

[16] Jacobson, V. Congestion avoidance and control. In *Proc. ACM SIGCOMM'88* (1988).

[17] Johansson, I. Self-clocked rate adaptation for conversational video in LTE. In *Proc. ACM SIGCOMM CSWS'14* (2014).

[18] Mukherjee, D., Bankoski, J., Grange, A., Han, J., Koleszar, J., Wilkins, P., Xu, Y., and Bultje, R. The latest open-source video codec VP9 - an overview and preliminary results. In *IEEE PCS'13* (2013).

[19] Nagy, M., Singh, V., Ott, J., and Eggert, L. Congestion control using FEC for conversational multimedia communication. In *Proc. ACM MMSys'14* (2014).

[20] Nam, H., Kim, K.-H., and Schulzrinne, H. QoE matters more than QoS: Why people stop watching cat videos. In *Proc. IEEE INFOCOM'16* (2016).

[21] Schulz-Zander, J., Mayer, C., Ciobotaru, B., Schmid, S., Feldmann, A., and Riggio, R. Programming the home and enterprise WiFi with OpenSDWN. In *Proc. ACM SIGCOMM'15* (2015).

[22] Shalunov, S., Hazel, G., Iyengar, J., and Kuehlewind, M. Low extra delay background transport (LEDBAT). IETF RFC 6817, 2012.

[23] Singh, V., Lozano, A. A., and Ott, J. Performance analysis of receive-side real-time congestion control for WebRTC. In *Proc. IEEE PV'13* (2013).

[24] Wei, D. X., Jin, C., Low, S. H., and Hegde, S. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw. 14*, 6 (2006), 1246–1259.

[25] Winstein, K., Sivaraman, A., Balakrishnan, H., et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. USENIX NSDI'13* (2013).

[26] Yiakoumis, Y., Katti, S., Huang, T.-Y., McKeown, N., Yap, K.-K., and Johari, R. Putting home users in charge of their network. In *Proc. ACM UbiComp'12* (2012).

[27] Zhu, X., and Pan, R. NADA: A unified congestion control scheme for low-latency interactive video. In *Proc. IEEE PV'13* (2013).