Performance Analysis of Receive-Side Real-Time Congestion Control for WebRTC

Varun Singh
Aalto University, Finland
varun.singh@aalto.fi

Albert Abello Lozano
Aalto University, Finland
albert.abello.lozano@aalto.fi

Jörg Ott
Aalto University, Finland
jorg.ott@aalto.fi

***Abstract*—In the forthcoming deployments of WebRTC systems, we speculate that high quality video conferencing will see wide adoption. It is currently being deployed on Google Chrome and Firefox web-browsers, meanwhile desktop and mobile clients are under development. Without a standardized signaling mechanism, service providers can enable various types of topologies; ranging from full-mesh to centralized video conferencing and everything in between. In this paper, we evaluate the performance of various topologies using endpoints implementing WebRTC. We specifically evaluate the performance of the congestion control currently implemented and deployed in these web-browser, Receive-side Real-Time Congestion Control (RRTCC). We use transport impairments like varying throughput, loss and delay, and varying amounts of cross-traffic to measure the performance. Our results show that RRTCC is performant when by itself, but starves when competing with TCP. When competing with self-similar media streams, the late-arriving flow temporarily starves the existing media flows.**

I. Introduction

The development of WebRTC systems is going to encourage wide adoption of video conferencing on the Internet. The main reason is the shift from a desktop or stand-alone real-time communication (RTC) application to RTC-enabled web-browser. The standardization activity for WebRTC is split between W3C[1] and the IETF[2]. W3C is defining the Javascript APIs and the IETF is profiling the existing media components (SRTP, SDP, ICE, etc.) for use with WebRTC. Until now, to enable voice and video calls from within the browser required each service to implement their real-time communication stack as a plugin, which the user downloads. Consequently, each voice and video service needed to build a plugin containing a complete multimedia stack. With WebRTC the multimedia stack is built into the web-browser internals and the developers need to just use the appropriate HTML5 API. However, WebRTC does not mandate a specific signaling protocol, and the implementers are free to choose from pre-existing protocols (SIP, Jingle, etc.) or write their own. In so doing, WebRTC provides the flexibility to deploy video calls in any application-specific topology (point-to-point, mesh, centralized mixer, overlays, hub-and-spoke).

While multimedia communication has existed for over a decade (via Skype, etc.), there is no standardized congestion-control, but many have been proposed in the past. To tackle congestion control, the IETF has chartered a new working group, RMCAT[3], to standardize congestion-control for real-time communication, which is expected to be a multi-year process [1]; but early implementations are already available.

In this paper, we evaluate the performance of WebRTC video calls over different topologies and with varying amounts of cross-traffic. All experiments are conducted using the Chrome browser and our testbed. With the testbed we are able to control the bottleneck link capacity, the end-to-end latency, link loss rate and the queue size of intermediate routers. Consequently, in this testbed we can investigate the following: 1) utilization of the bottleneck link capacity, 2) (un)friendliness to other media flows or TCP cross-traffic, and 3) performance in multiparty calls. The results in the paper complement the results in [2].

We structure the remainder of the paper as follows. Section II discusses the congestion control currently implemented by the browsers and section III describes the architecture for monitoring the video calls. Section IV describes the testbed and the performance of the congestion control in various scenarios. Section V puts the results in perspective and compares it to existing work. Section VI concludes the paper.

II. Receive-side Real-Time Congestion Control

Real-time Transport Protocol (RTP) [3] carries media data over the network, typically using UDP. In WebRTC, the media packets (in RTP) and the feedback packets (in RTCP) are multiplexed on the same port [4] to reduce the number of NAT bindings and Interactive Connectivity Establishment (ICE) overhead. Furthermore, multiple media sources (e.g, video and audio) are also multiplexed together and sent/received on the same port [5]. In topologies with multiple participants [6], media streams from all participants are also multiplexed together and received on the same port [7].

Currently, WebRTC-capable browsers implement a congestion control algorithm proposed by Google called *Receiver-side Real-time Congestion Control (RRTCC)* [8]. In the following sub-sections, we briefly discuss the important features and assumptions of RRTCC, which are essential for the evaluation. RRTCC has two components, a receiver-side and a sender-side, to function properly both need to be implemented.

[1] http://www.w3.org/2011/04/webrtc/

[2] http://tools.ietf.org/wg/rtcweb/

[3] http://tools.ietf.org/wg/rmcat/

978-1-4799-2172-0/13/$31.00 ©2013 IEEE

A. Receiver Side Control

The receiver estimates the overuse or underuse of the bottleneck link based on the timestamps of incoming frames relative to the generation timestamps. At high bit rates, the video frames exceed the MTU size and are fragmented over multiple RTP packets, in which case the received timestamp of the last packet is used. When a video frame is fragmented all the RTP packets have the same generation timestamp [3]. Formally, the jitter is calculated as follows: $J_i = (t_i - t_{i-1}) - (T_i - T_{i-1})$, where *t* is receive timestamp, *T* is the RTP timestamp, and the *i* and *i-1* are successive frames. Typically, if J_i is positive, it corresponds to congestion. Further, [8] proposes inter-arrival jitter as a function serialization delay, queuing delay and network jitter:

$$J_i = \frac{Size(i) - Size(i-1)}{Capacity)} + m(i) + v(i)$$

Here, $m(i)$ and $v(i)$ are a sample of a stochastic process, which is modeled as white Gaussian process. When the mean of Gaussian process, $m(i)$ is zero, then there is no ongoing congestion. When the bottleneck is overused, the stochastic process $m(i)$ is expected to increase, and when the congestion is abating it is expected to decrease. However, $v(i)$ is expected to be zero for frames that are roughly the same size, but sometimes the encoder produces key-frames (also called I-frames) which are an order of magnitude larger and are expected to take more time to percolate through the network. Consequently, these packets are going to have an increasing inter-arrival jitter just due to their sheer size.

The receiver tracks the J_i and frame size, while Capacity ($C(i)$) and $m(i)$ are estimated. [8] uses a Kalman filter to compute $[\frac{1}{C(i)} \quad m(i)]^T$ and recursively updates the matrix. Overuse is triggered only when $m(i)$ exceeds a threshold value for at least a certain duration and a set number of frames are received. Underuse is signaled when $m(i)$ falls below a certain threshold value. When $m(i)$ is zero, it is considered a stable situation and the old rate is kept. The current receiver estimate, $A_r(i)$ is calculated as follows:

$$A_r(i) = \begin{cases} 0.85 \times RR & \text{overuse}, m(i) > 0 \\ A_r(i-1) & \text{stable}, m(i) = 0 \\ 1.5 \times RR & \text{under-use}, m(i) < 0 \end{cases}$$

The receiving endpoint sends an RTCP feedback message containing the Receiver Estimated Media Bitrate (REMB) [9], A_r at 1s intervals.

B. Sender Side Control

The sender side uses the TFRC equation [10] to estimate the sending rate based on the observed loss rate (p), measured RTT and the bytes sent. If no feedback is received for two feedback intervals, the media rate is halved. In case a RTCP receiver report is received, the sender estimate is calculated based on the number of observed losses (p).

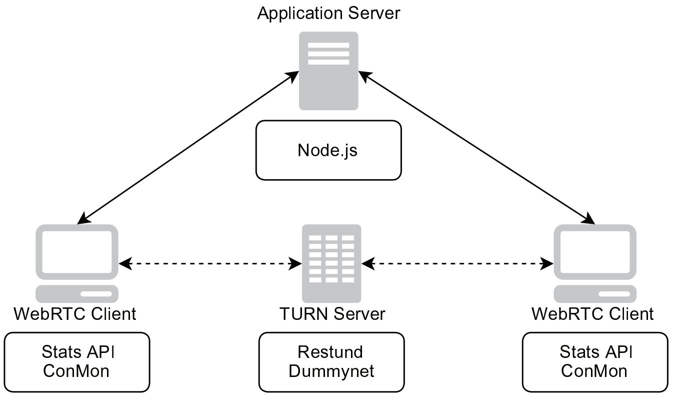


Figure 1. Description of simple testing environment topology for WebRTC.

$$A_s(i) = \begin{cases} A_s(i-1) \times (1 - 0.5p) & p > 0.10 \\ A_s(i-1) & 0.02 \leq p \leq 0.1 \\ 1.05 \times (A_s(i-1) + 1000) & p < 0.02 \end{cases}$$

Lastly, the $A_s(i)$ should be between the rate estimated by the TFRC equation and REMB value signaled by the receiver. The interesting part of the sender side algorithm is that it does not react to losses under 10% and in the case of losses less than 2%, it even increases the rate by a bit over 5%. In these cases, in addition to the media stream, the endpoint generates FEC packets to protect the media stream, the generated FEC packets are in addition to the media sending rate. This is noticeable in the Chrome browser when the rate-controller observes packet loss between 2-10%. In low-delay networks, we also observe retransmissions requests sent by the receiver in the form of Negative Acknowledgements (NACKs) and Picture Loss Indication (PLI). Full details can be found in the Internet-Draft [8].

III. Monitoring Architecture

We run a basic web application to establish WebRTC calls. It runs on a dedicated server and uses *nodejs*[4]. The signaling between browser and the server is over *websockets*[5]. Figure 1 describes an overview of the above architecture. We run multiple TURN servers (e.g., `restund`[6]) on dedicated machines to traverse NATs and firewalls. The WebRTC-capable browsers (Chrome[7] and Firefox[8]) establish a connection over HTTPS with our web server and download the WebRTC javascript (*call.js*) which configures the browser internals during call establishment. The media flows either directly between the two participants or via the TURN servers if the endpoints are behind NATs or firewalls.

[4] http://www.nodejs.org/ v0.10.6
[5] http://socket.io/
[6] http://www.creytiv.com/restund.html
[7] https://www.google.com/intl/en/chrome/browser/, v27.01453.12
[8] Only nightly builds support WebRTC, http://nightly.mozilla.org/

There are two ways to monitor a call in WebRTC: 1) in the web application running at the endpoint, or 2) analyzing packets on the wire at any middle-box. WebRTC provides a Statistics API that provides access to local and remote RTCP stats [11]. The web application javascript queries the browser internals (`RTCStatsCallback`), which returns a dictionary of local stats and remote stats. The local stats are returned immediately while the remote stats are dependent on the RTCP interval. Chrome sends RTCP feedback at 1s intervals, hence, the remote stats are at most ≈1s old. Currently, the stats API returns just the `bytes sent` and `bytes received` per media source, but more network statistics are expected in the future [12], [13]. To capture and analyze packets on the wire, middle-boxes typically use `tcpdump`[9]. However in WebRTC, all multimedia flows are multiplexed on a single port, i.e., media streams from each user, the corresponding media and feedback packets are all sent and received on the same port, mainly to reduce NAT and ICE overhead. The basic `tcpdump` utility is incapable of demultiplexing the media streams, consequently, we wrote our own utility, *Connection Monitor (ConMon)* [14] that detects STUN, RTP, RTCP, TURN-relayed packets and associates each incoming and outgoing packet to a media stream using `SSRC` and `PT`. ConMon logs these packets into local storage, and if logs are compared across measurement points (between endpoints or middle-boxes), we can calculate the latency, inbound & outbound packet and bit rate, and packet loss rate.

IV. Performance Evaluation

In this section, we discuss the performance of WebRTC sessions in various scenarios. We create an experimental testbed where each component in Figure 1 runs on independent machines. Specifically, one for each client, the web server that establishes the video calls and multiple TURN servers. To introduce impairments in the bottleneck, we force all the calls to go through the TURN servers. We use dummynet [15][10] to emulate the variation in link capacity, latency, intermediate router queue length, and use the Gilbert-Elliott Model [17], [18] to model packet loss.

In each WebRTC client (the browser), to have comparative results, instead of using a live camera feed, we setup a fake device[11] and use the *"Foreman"* video sequence[12] in VGA frame size, *30 frames per second*. The maximum target media bit rate is set as 2Mbps by the browser internals, in all the experiments the media rate does not exceed this value.

A. Experimental Methodology

We evaluate the performance of RRTCC in the following scenarios [19]:

- Single RTP flow on a bottleneck link
- Single RTP flow competing with TCP cross traffic on a bottleneck link
- Multiple RTP flows sharing a bottleneck link
- RTP group calls using a full mesh and an Multipoint Control Unit (MCU)

We run each call for 300 seconds, and run each scenario 15 times to derive statistical significance. The performance is tested by introducing the impairments or cross-traffic on the bottleneck. We measure the following metrics for each RTP flow.

- **Throughput**: the sender or receiver bit rate, measured in kbps.
- **Delay**: the end-to-end delay, combining the queuing, serialization, propagation delay; typically measured as a round trip time (RTT) in ms.
- **Loss**: the number of packets that did not arrive at the receiver; packets may be lost due to congestion on the Internet or due to bit corruption in the wireless environment.
- **Residual Loss**: the number of packets lost after retransmissions (retx) or applying Forward Error Correction (FEC).
- **Average Bandwidth Utilization (ABU)**: the ratio of the sending or receiver rate to the (available) end-to-end path capacity.

Using *ConMon*, we capture and log every transmitted and received packet, hence, it is straightforward to plot the instantaneous throughput, delay[13] and loss events. Furthermore, we are also able to plot the delay distribution to measure the self-inflicted delay. Based on these raw statistics, for each scenario we can calculate the mean and the 95% confidence interval for all tests belonging to a scenario.

B. Effects of Varying Bottleneck Link Characteristics

In this scenario, a single bi-directional RTP stream flows through the bottleneck links. In each test run, we vary only one network characteristic and observe the performance of RRTCC. First, we add one-way latency to the bottleneck links, we pick one of the following: 0ms, 50ms, 100ms, 200ms, 500ms and observe that the increase in latency reduces the average media rate. Figure 2 shows the instantaneous receiver rate for each latency. It is surprising because the receiving endpoint does not observe any loss, nor does the RTT measure (2×latency) change. This is shown in Table I.

Next, we introduce loss (0%, 1%, 5%, 10% and 20%) at the bottleneck and observe that in general, an increase in loss rate decreases the media rate. However, RRTCC does not react to losses below 10% can be inferred from Table II and Figure 3. Additionally, we observe that the packet loss is high but the residual losses in all the cases is very low, because the sender uses retransmissions and FEC to protect the media stream. The error resilience strategy works because this scenario only introduces packet losses but no latency and

[9] http://www.tcpdump.org/

[10] Dummynet runs at 1000Hz clock rate and is modified to accept queues of up to 1000 packets for emulating buffer-bloat [16].

[11] Follow the instructions at https://code.google.com/p/webrtc/source/browse/trunk/src/test/linux/v4l2_file_player/?r=2446

[12] http://xiph.org

[13] This requires time synchronization between the machines, we run an NTP server in our testbed and the machines regularly synchronize to it.

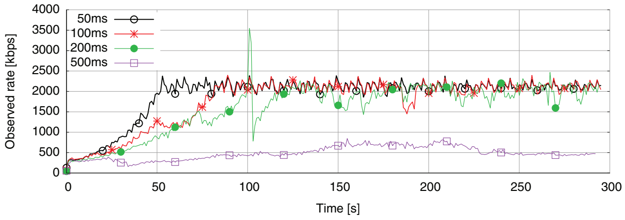


Figure 2. Variation in receiver rate due to different bottleneck latency

	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
0 ms	1949.7±233.62	9.57±2.41	0.011	0.011
50 ms	1913.56±254.86	102.51±1.44	0.05	0.05
100 ms	1485±268.11	202.57±3	0.06	0.06
200 ms	560.82±129.57	401.91±3.33	0.33	0.4
500 ms	255.67±45.85	1001.36±3.99	0.35	0.37

Table I
METRICS FOR A BOTTLENECK WITH VARYING LATENCIES.

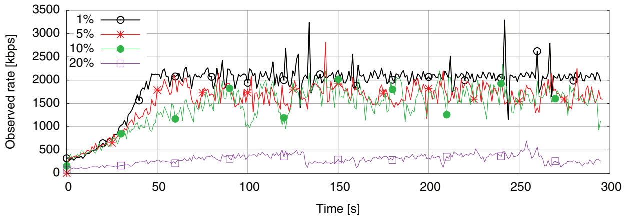


Figure 3. Variation in receiver rate due to different packet loss

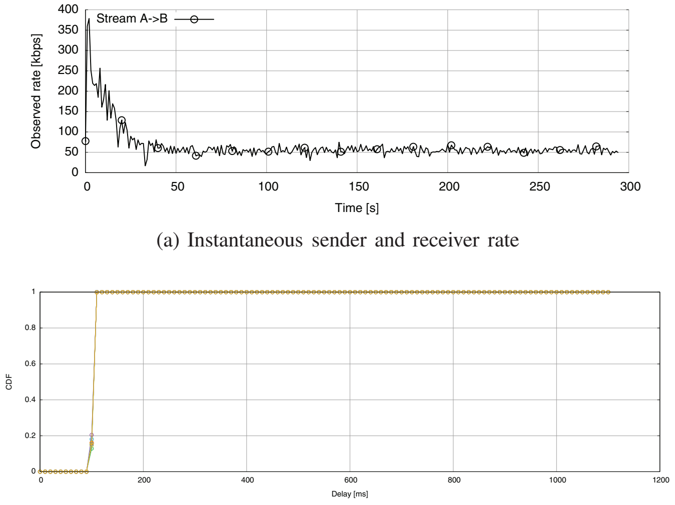
	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
0%	1949.7±233.62	9.57±2.41	0.011	0.011
1%	1986.91±256.78	8.12±1.86	0.09	2
5%	1568.74±178.52	6.98±1.79	0.23	9.77
10%	1140.82±161.92	6.28±3.24	0.49	19.02
20%	314.4±61.98	5.42±4.03	2.43	36.01

Table II
METRICS FOR A BOTTLENECK WITH DIFFERENT PACKET LOSS RATES.

ergo the receiver is able to request retransmission which arrive in time for decoding (RTT<10ms).

Based on the previous two scenarios, we introduce both loss (5%) and latency (100ms) in the bottleneck link and observe that RRTCC can no longer rely on retransmissions. The residual loss increases from 0.23% to 7%, while the overall packet loss (≈9%) remains the same. Figure 4 shows the instantaneous sending rate and the distribution of one-way delay. The sending rate reduces when it encounters loss, but the packet delay remains constant at 100ms.

We vary the queue size at an intermediate router and observe the impact on the performance of RRTCC. We describe the queue sizes as a function of time, i.e., it is the depth of the queue or the amount of time the packet will remain in the queue before it is discarded. However, in practice the queue size is measured in number of packets. We convert the queue depth (measured in time) to queue length (number of packets) using:



(a) Instantaneous sender and receiver rate

(b) CDF of delay distribution

Figure 4. Impact of link loss and latency on throughput and one-way delay.

	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
	1 Mbps			
100 ms	362.87±68.18	31.42±11.9	0.79	1.42
1 s	374.64±58.64	27.48±8.81	0.05	0.05
10 s	438.32±31.58	27.16±7.32	0.04	0.04
	5 Mbps			
	Rate (Kbps)	**RTT (ms)**	**Residual Loss (%)**	**Packet Loss (%)**
100 ms	1965.95±224.02	20.13±4.16	0.02	0.02
1 s	1920.67±234.38	18.45±5.57	0.02	0.02
10 s	1722.08±261.42	17.05±4.52	0.03	0.03

Table III
METRICS FOR A BOTTLENECK WITH DIFFERENT QUEUE LENGTHS AND CAPACITY.

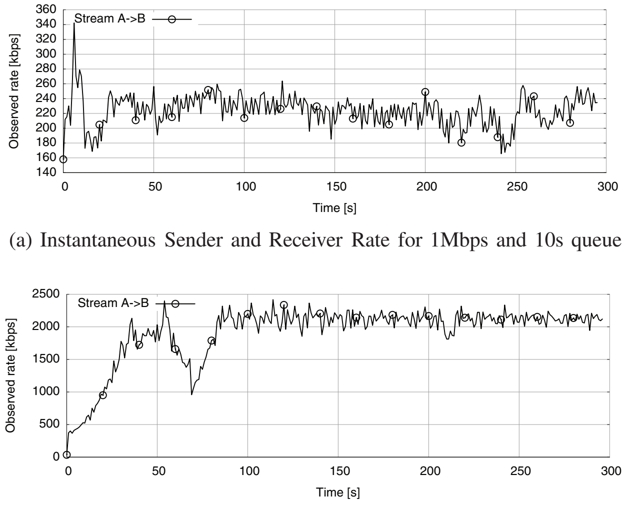
$$\text{QueueSize}_{\text{packets}} = \frac{\text{QueueSize}_{\text{sec}} \times \text{Throughput}_{\text{bps}}}{\text{MTU} \times 8}$$

In our experiments the MTU is 1500 bytes. For example, a router with a throughput of 1Mbps and a 1s queue depth would be capable of handling 83 packets (queue length). We experiment with queue depths of 100ms, 1s and 10s. The 100ms queue depth represents a short queue, while the 10s queue depth represents a buffer-bloated queue. Additionally, we measure RRTCC's performance with two different bottleneck link rates, 1Mbps and 5Mbps.

Our results show that the Average Bandwidth Utilization (ABU) in both scenarios, the 1Mbps and 5Mbps is ≈0.4, therefore varying the router's queue-lengths does not have any impact on the performance of RRTCC. Table III compares the performance of the queue-length in different bottleneck conditions. Figure 5 shows the instantaneous receiver rate for the two scenarios using buffer-bloated queues.

C. Effects of TCP Cross Traffic

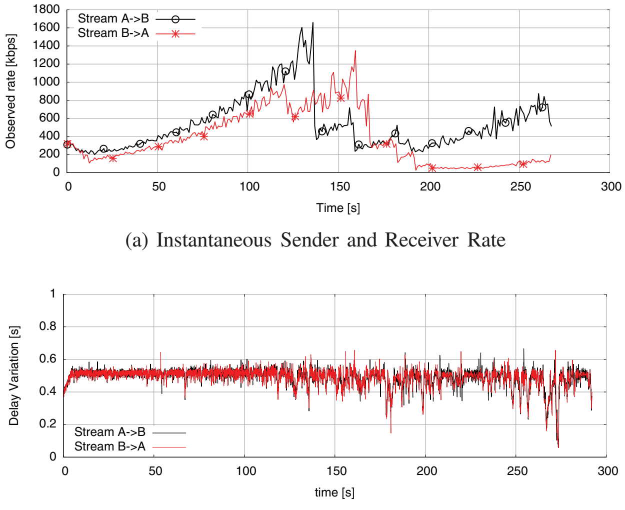
In this scenario, a single RTP flow competes with TCP cross-traffic on the bottleneck with different link capacities. We use long TCP flows that represent large file downloads and



(a) Instantaneous Sender and Receiver Rate for 1Mbps and 10s queue

(b) Instantaneous Sender and Receiver Rate for 5Mbps and 10s queue

Figure 5. Impact of buffer-bloated queues on the performance of RRTCC.



(a) Instantaneous Sender and Receiver Rate

(b) Instantaneous Delay

Figure 6. Impact of TCP cross-traffic on the performance of RRTCC, The bottleneck link capacity is 10/10Mbps.

10/1 Mbps				
	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
0 ms	75.49±25.02	9.84±13.81	0.26	0.22
25 ms	84.02±29.17	57.08±2.97	0.24	0.19
100 ms	80.2±30.6	216.38±4.11	0.19	0.19
10/10 Mbps				
	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
50 ms	481.97±37.6	256.9±85.81	0.06	0.09

Table IV
METRICS FOR A BOTTLENECK WITH VARYING AMOUNT OF TCP CROSS-TRAFFIC.

are emulated by an iperf server running at the corresponding TURN server. Identical iperf clients run at the endpoints. The long TCP flow downloads an unbounded amount of data (representing file downloads) and runs in parallel to the media flows between the web browser and the TURN server, i.e., at any endpoint the TCP data stream and the incoming RTP flow share a common bottleneck, while the outbound RTP flow shares the bottleneck with TCP control packets.

In the first scenario, the RRTCC media flow at each endpoint competes with 2 TCP flows at the bottleneck. The bottleneck capacity is set to 1Mbps upstream and 10Mbps downstream. Effectively, the 10Mbps should be shared between the TCP and remote video and majority of the 1Mbps can be consumed by the local video. Table IV shows that the RRTCC flow starves and does not compete with the TCP, it is only able to manage a fraction of the available bandwidth.

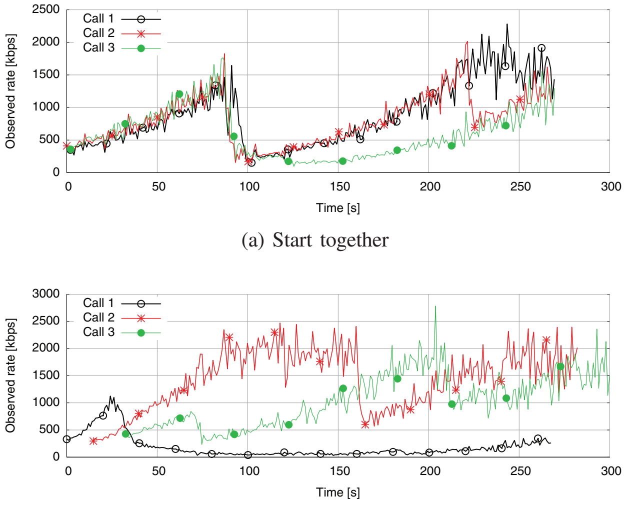
In the second scenario, the upstream and downstream capacity of the bottleneck link is set at 10Mbps, the result is still similar to the previous scenario, the RRTCC flow starves. Figure 6 shows the instantaneous receiver rate and the instantaneous delay. In this scenario, it appears that the TCP keeps on increasing its congestion window, hence, filling up the routers queues. When the RTP flow observes the loaded queues, it tries to avoid it by reducing the rate, therefore ramps up very slowly and on encountering timeouts, reduces its rate by half. We also observe that the rate increases when the delay decreases, this is observed by comparing Figure 6(a) and (b). One possible reason for the decrease in delay is that the routers queue exceed the limit and drops incoming packets, which causes the TCP to go into slow start, allowing the RRTCC flow to ramp-up.

D. Effects of RTP Cross Traffic

In this scenario, multiple RTP flows traverse the bottleneck link, i.e., each flow corresponds to a distinct call. We perform three experiments in this scenario. Firstly, two calls share a common bottleneck, i.e., 2 RTP flows in each direction share the same TURN server. For these tests, we remove dummynet and let the 2 calls compete with each-other without any constraints. Here, too the calls are unable to occupy any substantial share of the bottleneck. This is shown in Table V.

Next, three calls share the bottleneck without any dummynet constraints, in this case the individual media rates are higher but are not able to reach their individual maximum/target rate of 2Mbps. Figure 7(a) shows three calls ramp-up at about the same rate, reach a peak and drop their rate together. Even though the flows start from different endpoints using independent WebRTC stacks, the rates synchronize.

Lastly, three calls share the bottleneck link, but each call starts at 15s intervals. We observe that while the media rate per call on average is higher, almost 3 times higher compared to the first scenario, the first call has a disadvantage and in all the cases, temporarily starves when the new flows appear and after a few minutes starts to ramp-up. Figure 7(b) shows the instantaneous rates of each of the calls. The first call temporarily starves because when it starts it is the only flow on the bottleneck and does not encounter any queues. When the second flow appears, it already observes queues from the existing stream and competes with it, while the initial flow observes an increase in queues and reduces the sending rate



(a) Start together

(b) Start 15s apart

Figure 7. Variation in sending rate for 3 parallel calls a) starting together, b) starting 15s apart. The total duration of the call is 5 mins (300s).

	Rate (Kbps)	RTT (ms)	Residual Loss (%)	Packet Loss (%)
2 calls	469.01±221.09	5.5±5.26	0.04	0.03
3 calls	809.07±202.38	31.48±24.93	0.21	0.23
3 calls (time shifted)	1154.32±250.54	35.15±27.88	0.08	0.91

Table V
RRTCC COMPETING WITH SIMILAR CROSS-TRAFFIC ON THE BOTTLENECK LINK.

to avoid congestion.

E. Multiparty Calls: Full-Mesh

In this scenario, we setup group calls between three participants in a full-mesh topology, i.e., each participant sends its media to the other two participants and receives individual media streams from each participants. Figure 8 shows a sample mesh topology. In this scenario, we do not use any TURN servers and do not apply any constraints on the links. The endpoints A and B are attached to separate switches in the network, while endpoint C is connected via WLAN. The common bottleneck in the mesh is typically the first and last hop before the endpoint, because the incoming media streams share the access link. Similarly, even though the outgoing media stream are destined to different endpoints, they too share the common access link.

Figure 9 shows the sending rate of the media streams at each endpoint. Comparing across the plots, we notice that each endpoint runs an independent congestion control for each flow even though it is encoding the same media stream. It is also observed in the high standard deviation in the sending rate of each participant (See Table VI). Similar to the previous scenario (competing with similar cross-traffic), here too the media rate averages around 400 Kbps.

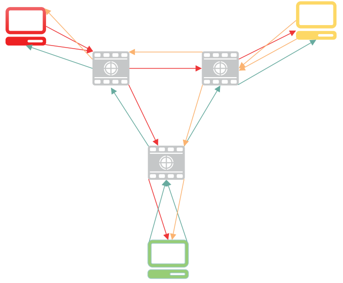
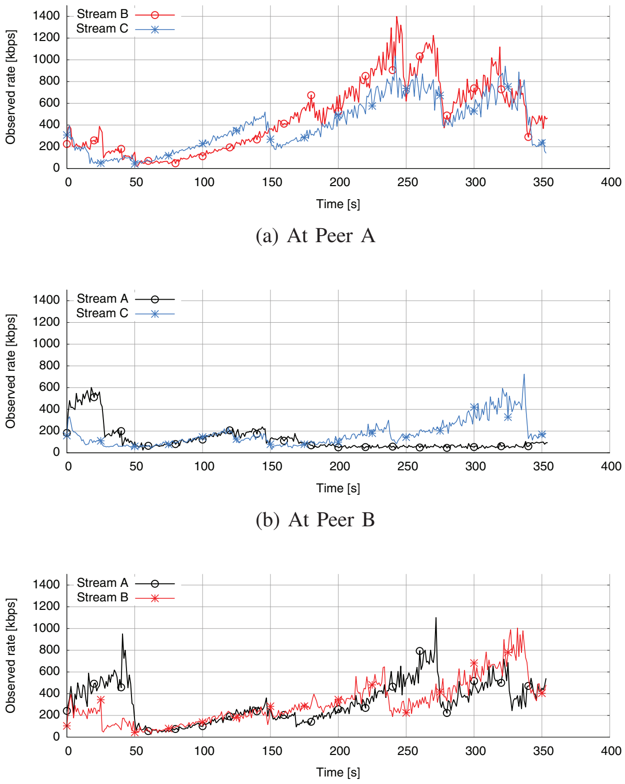


Figure 8. Sample Mesh Topology.



(a) At Peer A

(b) At Peer B

(c) At Peer C

Figure 9. Variation in sending rate for every outgoing stream at each endpoint in a mesh call.

	Peer A	Peer B	Peer C
Rate (Kbps)	333.38±115.13	344.48±95.43	410.77±115.97
RTT (ms)	12.2±5.09	11.58±5.09	5.82±5.15
Residual Loss (%)	0.01	0.01	0.00
Packet Loss (%)	0.01	0.01	0.00

Table VI
THREE-PEER MESH CALL.

F. Multiparty Calls: Mixer

In this scenario, we setup a group call between three participants using a centralized conferencing server operating as a mixer. Figure 10 shows an example topology containing multiple participants and an MCU. The conferencing server is a simple address translator or packet forwarder that receives the media stream and sends the incoming media stream to

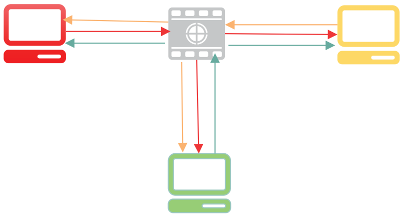


Figure 10. Sample Star Topology.

	Peer A	Peer B	Peer C
Rate (Kbps)	604.31±149.38	403.74±93.99	882.94±228.45
RTT (ms)	13.76±7.88	12.62±7.6	12.8±7.28
Residual Loss (%)	0.05	0.06	0.07
Packet Loss (%)	0.05	0.06	0.07

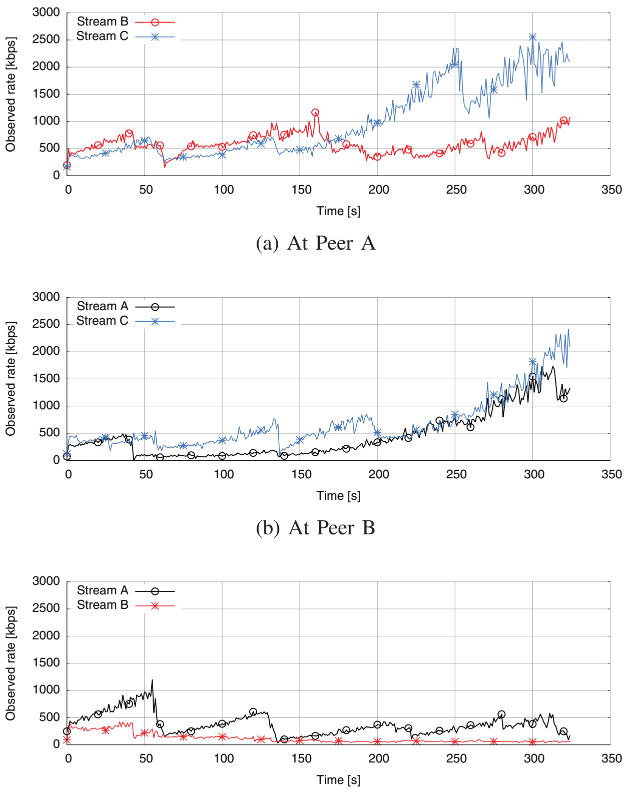
Table VII
MULTIPARTY CALL USING A CENTRALIZED MCU.

the individual remote participants. However, instead of generating the feedback packets for each established flow (RTCP-Terminating MCU [6]), it gathers the feedback from the remote endpoints and sends it to the sender[14]. Hence, even though an endpoint sends one media stream, it receives a feedback report for each participant (in this case, two). For example, a sender may get conflicting reports from two senders, one asking to increase the rate, while the other requesting to lower it. The sender based on the two feedback reports is expected to make a congestion control decision. There is no specified behavior for RRTCC and the sending endpoint tries to make the best congestion control decision.

Figure 11 shows the instantaneous sending bit rate at each endpoint. We observe that in some cases the link is unable to carry the aggregate media stream and throughput suffers. For example, Stream C from both endpoints increases over time, but outgoing streams from Endpoint C suffer. Whereas, the streams from Endpoint A and Endpoint B do not suffer as much. Table VII describes the averaged metrics across 10 test runs.

V. RELATED WORK

The Real-time Transport Protocol (RTP) [3] is designed to carry multimedia data and is favored over TCP for media delivery due to the low-latency requirements. Real-time video is tolerant to some amount of packet loss, instead focuses on reducing the end-to-end latency. If single packet loss occurs, the application is capable of concealing it or uses an error-resilience mechanism (retransmissions, forward error correction) to recover the lost packet. However, error-resilience can increase the playout delay causing the media to pause or skip media frames leading to a poor user experience [20]. Therefore, real-time communication applications need to either implement congestion control or use a transport that



(a) At Peer A

(b) At Peer B

(c) At Peer C

Figure 11. Variation in sending rate for each outgoing stream at every endpoint in a multiparty call using a mixing MCU, the RTCP goes end-to-end.

implements congestion control. Unfortunately, TCP is only suitable for interactive multimedia for paths with very low RTT ($<$ 100ms) [21].

Many congestion control algorithms have been proposed, they chiefly attempt to match the media rate to the available capacity while maintaining a good user-experience. Most prominent is the TCP Friendly Rate Control (TFRC) [10], which is implemented using the information contained in standard RTCP reports, but requires per-packet feedback [22]. One disadvantage of TFRC is that it produces a sawtooth sending rate [23], [24]. RAP [25] uses a windowed-approach and this too exhibits a sawtooth-type of behavior. Any algorithm that consistently produces a sawtooth media rate is not well suited for real-time communication because it generates a poor user-experience [20], [26], [27].

Instead of just relying on RTT and loss for congestion control, Garudadri *et al.* [28] also use the receiver playout buffer to detect underutilization and overuse. Singh *et al.* [29], [30] use a combination of congestion indicators: frame inter-arrival time, playout buffer size for congestion control. Zhu *et al.* [31] use ECN and loss rate to get an accurate estimate of losses for congestion control. O'Hanlon *et al.* [32] propose using a delay-based estimate when competing with similar traffic and using a windowed-approach when competing with TCP-type cross traffic, they switch modes by using a threshold on the observed end-to-end delay, the idea is similar to the

[14] While the MCU described above is not an ideal conferencing server, it is an approximation of a simple conferencing server. Additionally, this type of topology can also occur using RTP overlays or asymmetric conferencing trees. A proper solution to use a MCU would either require scalable video codecs or simulcast, neither is currently supported by WebRTC.

one discussed in [33]. Nagy *et al.* [30] use FEC to probe for available bandwidth, the basic idea is to use more redundancy when aggressively probing for available capacity and use less redundancy when conservatively probing for available capacity. While several of these congestion control algorithms have been proposed in RMCAT, they are yet to be implemented in any WebRTC browser.

VI. Conclusion

In this paper, we carried out a series of experiments on a test-bed emulating real-world conditions. We show that the RRTCC algorithm (currently implemented in Google Chrome): 1) is performant in networks with latencies up to 200ms, but its bandwidth utilization collapses when latencies exceed 200ms; 2) the sending data rate under-utilizes when competing with TCP cross traffic to avoid increase in latencies; 3) shares the bottleneck adequately when competing with similar RTP flows, however, with late arriving RTP flows, the existing RRTCC flows may temporarily starve (for 10s of seconds); 4) path under-utilization is observed when a single flow is sent to two other participant in a mesh call, this is expected because there are two independent congestion controllers working alongside each other on the same media stream. Based on these observations, we summarize that RRTCC works well in low delay networks, can tolerate transient changes (losses or delays or queuing) and can compete within limits against varying amount of cross-traffic.

This is an initial analysis of RRTCC, for future work, we intend to use different types of loss regimes (bursty) based on real-world traces [34]. Furthermore, analyze the impact of short TCP flows instead of the long TCP flows that are used in the current paper. When WebRTC is available on the mobile phone, the analysis needs to be extended to test across wireless and 3G networks.

Acknowledgement

Varun Singh is partially supported by Future Internet Graduate School and the EIT ICT Labs activity RCLD 11882.

References

[1] C. Jennings, T. Hardie, and M. Westerlund, “Real-time communications for the web,” *IEEE Communications Magazine*, vol. 51, no. 4, pp. 20–26, April 2013.

[2] L. De Cicco, G. Carlucci, and S. Mascolo, “Experimental investigation of the google congestion control for real-time flows,” in *Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking*, ser. FhMN ’13. New York, NY, USA: ACM, 2013, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/2491172.2491182

[3] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A transport protocol for real-time applications,” Internet Engineering Task Force, July 2003, RFC 3550.

[4] C. Perkins and M. Westerlund, “Multiplexing RTP Data and Control Packets on a Single Port,” RFC 5761 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5761.txt

[5] C. Perkins, M. Westerlund, and J. Ott, “Web Real-Time Communication (WebRTC): Media Transport and Use of RTP,” http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage, IETF Internet Draft.

[6] M. Westerlund and S. Wenger, “Rtp topologies,” http://tools.ietf.org/html/draft-westerlund-avtcore-rtp-topologies-update, April 2013, IETF Internet Draft.

[7] C. Holmberg, S. Hakansson, and G. Eriksson, “RTP Topologies,” http://tools.ietf.org/html/draft-ietf-rtcweb-use-cases-and-requirements, IETF Internet Draft.

[8] H. Alvestrand, S. Holmer, and H. Lundin, “A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web,” 2013, IETF Internet Draft.

[9] H. Alvestrand, “RTCP message for Receiver Estimated Maximum Bitrate,” http://tools.ietf.org/html/draft-alvestrand-rmcat-remb, 2013, IETF Internet Draft.

[10] S. Floyd *et al.*, “Equation-based congestion control for unicast applications,” in *Proc. ACM SIGCOMM*, August 2000.

[11] A. Bergkvist, D. Burnett, and C. Jennings, “WebRTC 1.0: Real-time Communication Between Browsers,” http://dev.w3.org/2011/webrtc/editor/webrtc.html, 2013.

[12] H. Alvestrand, “A Registry for WebRTC statistics identifiers,” http://tools.ietf.org/html/draft-alvestrand-rtcweb-stats-registry, IETF Internet Draft.

[13] V. Singh, R. Huang, and R. Even, “Additional RTP Control Protocol (RTCP) Extended Report (XR) Metrics for WebRTC Statistics API,” http://tools.ietf.org/html/draft-singh-xrblock-webrtc-additional-stats, IETF Internet Draft.

[14] V. Singh, “Conmon: An application for monitoring connections,” http://vr000m.github.com/ConMon/, 2013.

[15] M. Carbone and L. Rizzo, “Dummynet revisited,” *in Proc. of ACM SIGCOMM CCR*, Jan 2010.

[16] J. Gettys and K. Nichols, “Bufferbloat: dark buffers in the Internet,” *in Proc. of Communications of the ACM*, vol. 55, pp. 57–65, Jan 2012.

[17] E. N. Gilbert *et al.*, “Capacity of a burst-noise channel,” *Bell Syst. Tech. J*, vol. 39, no. 9, pp. 1253–1265, 1960.

[18] E. Elliott, *Estimates of error rates for codes on burst-noise channels*. Bell Telephone Laboratories, 1963.

[19] V. Singh and J. Ott, “Evaluating Congestion Control for Interactive Real-time Media,” http://tools.ietf.org/html/draft-singh-rmcat-cc-eval, July 2013, IETF Internet Draft.

[20] M. Zink, O. Künzel, J. Schmitt, and R. Steinmetz, “Subjective Impression of Variations in Layer Encoded Videos,” in *Proc. of IWQoS*, 2003.

[21] E. Brosh, S. A. Baset, D. Rubenstein, and H. Schulzrinne, “The Delay-Friendliness of TCP,” in *Proc. of ACM SIGMETRICS*, 2008.

[22] L. Gharai and C. Perkins, “RTP with TCP Friendly Rate Control,” March 2011, (Work in progress).

[23] A. Saurin, “Congestion Control for Video-conferencing Applications,” Master’s Thesis, University of Glasgow, December 2006.

[24] V. Singh, J. Ott, and I. Curcio, “Rate adaptation for conversational 3G video,” in *Proc. of INFOCOM Workshop*, Rio de Janeiro, BR, 2009.

[25] R. Rejaie, M. Handley, and D. Estrin, “RAP: An End-To-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet,” in *Proc. of INFOCOM*, Mar 1999.

[26] L. Gharai and C. Perkins, “Implementing Congestion Control in the Real World,” in *Proc. of ICME ’02*, vol. 1, 2002, pp. 397 – 400 vol.1.

[27] H. Vlad Balan, L. Eggert, S. Niccolini, and M. Brunner, “An Experimental Evaluation of Voice Quality Over the Datagram Congestion Control Protocol,” in *Proc. of IEEE INFOCOM*, 2007.

[28] H. Garudadri, H. Chung, N. Srinivasamurthy, and P. Sagetong, “Rate Adaptation for Video Telephony in 3G Networks,” in *Proc. of PV*, 2007.

[29] V. Singh, J. Ott, and I. Curcio, “Rate-control for Conversational Video Communication in Heterogeneous Networks,” in *in Proc. of IEEE WoWMoM Workshop*, SFO, CA, USA, 2012.

[30] M. Nagy, V. Singh, J. Ott, and L. Eggert, “Congestion control using fec for conversational multimedia communication,” *arXiv preprint arXiv:1310.1582*, 2013.

[31] X. Zhu and R. Pan, “ NADA: A Unified Congestion Control Scheme for Real-Time Media,” http://tools.ietf.org/html/draft-zhu-rmcat-nada-01, 2013, IETF Internet Draft.

[32] P. O’Hanlon and K. Carlberg, “Congestion control algorithm for lower latency and lower loss media transport,” http://tools.ietf.org/html/draft-ohanlon-rmcat-dflow, 2013, IETF Internet Draft.

[33] Ł. Budzisz, R. Stanojević, A. Schlote, F. Baker, and R. Shorten, “On the fair coexistence of loss-and delay-based tcp,” *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 6, pp. 1811–1824, 2011.

[34] M. Ellis, C. S. Perkins, and D. P. Pezaros, “End-to-end and network-internal measurements of real-time traffic to residential users,” in *Proc. ACM MMSys*, San Jose, CA, USA, Feb. 2011.