A Dynamic Approach to Estimate Receiving Bandwidth for WebRTC

Razib Iqbal[1], Shervin Shirmohammadi[2], Rasha Atwah[3]
[1]*Missouri State University, Springfield, MO, USA*
[2]*University of Ottawa, Ottawa, ON, Canada*
[3]*King Abdulaziz University, Jeddah, Saudi Arabia*

ABSTRACT
Web Real-Time Communication (WebRTC), drafted by the World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF), enables direct browser-to-browser real-time communication. As its congestion control mechanism, WebRTC uses the Google Congestion Control (GCC) algorithm. But using GCC will limit WebRTC's performance in cases of overusing due to using a fixed decreasing factor, known as alpha (α). In this paper, we propose a dynamic alpha model to reduce the receiving bandwidth estimate during overuse as indicated by the overuse detector. Using our proposed model, the receiver can more efficiently estimate its receiving rate in case of overuse. We implemented our model over both unconstrained and constrained networks. Experimental results show noticeable improvements in terms of higher incoming rate, lower Round-Trip Time, and lower packet loss compared to the fixed alpha model.

Keywords: WebRTC, Bandwidth Estimation, Congestion Control, Video Conferencing, Dynamic alpha.

INTRODUCTION
Popularity of mobile technologies such as smartphones and tablets has encouraged competition amongst the video conferencing system researchers and developers to enhance the Real-Time Communication (RTC) technology as well as to implement various applications based on RTC. Accessibility and mobility of multimedia and video conferencing applications has also motivated the creation of browser-to-browser video conferencing technologies that can compete with the current standalone video conferencing applications [1],[2]. In 2012, Web Real-Time Communication (WebRTC) was proposed with the aim to implement RTC into web browsers, enabling audiovisual calls between two or more end users with no need to install third party plug-ins or specific applications. Real-Time Communication on the Web (RTCWeb) is an open source project of the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) team that aims to standardize protocols and infrastructures for WebRTC [3]. Web developers can add WebRTC features to their browsers using standard Hyper Text Markup Language (HTML5) with simple Javascript APIs [4]. Therefore, it allows any web browser running on any operating system and any device to connect seamlessly with any other web browser running on any operating system and any device via the internet and communicate in real-time, as long as both web browsers support WebRTC. WebRTC is open source, and it is currently implemented in three browsers: Google Chrome, Firefox, and Opera [3].
Like any other RTC application, WebRTC has to deal with the quality from two different perspectives – user and network. From the user's perspective, services such audio-video streams have to be provided with less packet loss and delays, along with a high incoming rate and quality. From the network perspective, WebRTC has to have a mechanism that can deal with bottleneck problems and congestion to produce acceptable performance during a video session. Therefore,

the IETF and W3C working groups proposed a congestion control mechanism, called Google Congestion Control (GCC) algorithm, for WebRTC to satisfy the quality requirements for real-time applications [5]. GCC has two main controllers: Receiver-side controller and Sender-side controller, as shown in Figure 1 [9]-[11]. The Receiver-side controller computes the next expected receiving rate A_r. It is composed of five main components: Arrival-time filter (estimates the queuing time variation T), Remote rate region (computes the threshold γ), Overuse detector (produces appropriate signal, overuse/underuse/normal, based on T and γ), Remote rate controller (estimates the next expected receiving rate A_r), and Receiver Estimated Max Bitrate (REMB) message processing unit (notifies sender with the next expected receiving rate). The Sender-side controller consists of two main components: TCP-Friendly Rate Control (TFRC) bandwidth estimation, and Sending Rate Controller. According to [11], sender-side controller is a loss-based congestion control algorithm which computes the target sending bitrate A_s.

When congestion occurs, users experience packet losses or delays in picture or sound. In such a scenario, to decrease bandwidth usage and counter congestion, GCC's receiver-side controller uses an overuse detector (which takes into account the queuing delay), and when the detector indicates overuse, GCC reduces the current incoming rate by a fixed factor denoted as *alpha* (α). As per [11], alpha is a decreasing factor that is typically chosen to be within the interval of [0.80, 0.95]. The receiver-side controller multiplies α by the current incoming rate to estimate the new receiving rate. But, this alpha is a fixed amount and does not take into account the amount of queuing delay. Therefore, regardless of how much overuse has occurred, the receiver-side has to decrease its rate by the fixed alpha. It is intuitive that this behavior is not efficient. One would expect a faster decrease of the incoming rate for more severe congestion, and a slower decrease for less severe congestion. In other words, the amount of decrease in incoming rate should be proportional to how bad the congestion is. But GCC currently does not take this proportion into account. This behavior eventually reflects negatively on WebRTC's performance, resulting in a video session with a less than optimal video quality.

In order to address this issue, we propose a dynamic rate adaption model, called dynamic alpha model, which takes the amount of network delay into consideration, and calculates the incoming rate reduction amount proportionally. We implemented this model in the WebRTC reference code [3] and evaluated it in both constrained (in terms of bandwidth capacity, packet loss rate, and delay) and unconstrained networks. Analysis of our experimental results shows that our dynamic alpha model will improve WebRTC's performance when congestion occurs.

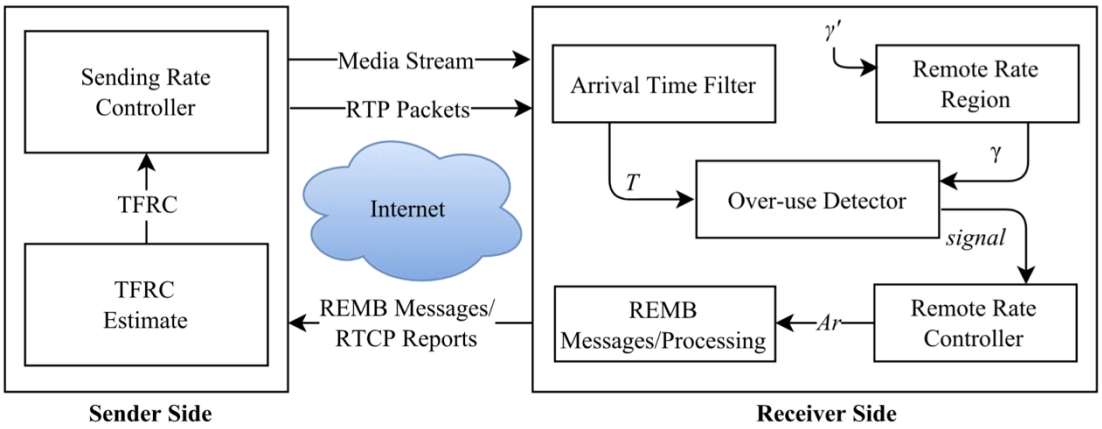

Figure 1. GCC Receiver-side and Sender-side components

The rest of this paper is organized as follows: we continue by describing in more details the inner workings of GCC, and then present our proposed model for dynamic alpha. We then describe our experimental setup, performance results, and their analysis, before concluding the paper.

RELATED WORK

In this section we explain the inner workings of GCC. But before that, and since the sender side in the GCC algorithm uses TCP-Friendly Rate Control (TFRC) to estimate the sending rate, we start with a description of the TFRC algorithm.

TCP-Friendly Rate Control

According to [21], RFC 5348 defines the TCP-friendly rate control as "TFRC is a congestion control mechanism designed for unicast flows operating in the Internet environment and competing with TCP traffic." In other words, TFRC is a congestion control algorithm that provides a smooth transmission rate for real-time applications [8]. TFRC is designed to give the best performance for applications that use a fixed segment size and vary their sending rates in response to congestion [21]. However, for applications that do not use a fixed segment size, such as video applications, the TFRC perhaps gives less performance because these applications vary their sending rates according to the needs of the application. The Google Congestion Control sender-side implements TFRC to estimate the sending rate based on Eq. 2. Later, we will cover how the sender side adjusts its sending rate based on the TFRC rate, and more details are presented in subsequent sections. TFRC calculates its transmission rate, according to the TCP throughput equation shown below [20]:

$$X = \frac{S}{R\sqrt{\frac{2bp}{3}+(t\_RTO(3\sqrt{\frac{3bp}{8}})p(1+32p^2))}} \qquad (1)$$

In the above equation, X is the sending rate in byte/second, S is packet size in bytes, R is Round trip time in second, p is loss event rate between 0 and 1, t_RTO is the TCP retransmission timeout values in a second, and b is the number of packets acknowledged by a single TCP.
The receiver uses timestamps and Round-Trip Time (RTT) to determine if losses belong to the same loss event or not. The TCP throughput equation takes the RTT to calculate the TFRC rate. The sender can adjust its sending rate, according to the TFRC rate [8]. According to [19], RFC 3448 is suited for many multimedia streaming applications. However, it is also used for the continuous flow of data [8]. The development of TFRC is still ongoing in the IETF group.

Google Congestion Control algorithm

In [12], authors presented a performance analysis of receiver-side real-time congestion control for WebRTC. According to [6], congestion occurs when resource demands exceed the capacity. From the network perspective, if we have two or more sources and each source increases its sending rate beyond the link capacity, then after a while the network will get congested. From the user perspective, congestion contributes to decreased quality, in the form of delay and/or packet loss. In order to allow the user to use the Internet as efficiently as possible, many congestion control mechanisms have been proposed [6]. As mentioned earlier, WebRTC uses the Google Congestion Control (GCC) Algorithm [5] for congestion control. GCC runs over the User Datagram Protocol (UDP) where the audio/video frames are encapsulated in Real Time Protocol (RTP) packets. The

congestion control is applied only to the video streams since the audio streams bitrate are considered negligible. The GCC algorithm follows a rate-based congestion avoidance approach, where the sender is monitoring the packet loss and calculates the sending rate based on the feedback received from the receiver-side. The receiver-side informs the sender of the next expected receiving rate to prevent the sender from exceeding this rate. Sender-side then estimates the sending rate based on the feedback from the receiver side and applying the TFRC. As per [7], GCC sender-side implements TFRC to estimate the sending rate based on Eq. 1.

The feedback from the receiver arrives in the form of Real-time Transport Control Protocol (RTCP) reports, which carry the packet loss rate (p) and Round-Trip Time (RTT). Also, the Receiver Estimated Maximum Bit (REMB) messages contain the maximum rate (Ar) that is expected to be handled by the receiver side. Therefore, the new sending rate must be less than the receiver's rate estimation and larger than the TFRC. As in [9], the sender side uses the packet loss rate to estimate the sending rate (As), which is given by Eq. 2:

$$As(i) = \begin{cases} max\{\mathrm{S}(i), As\ (i-1)(1-0.5p)\} & p > 0.10 \\ As(i-1) & 0.02 < p < 0.10 \\ 1.05(As(i-1) + 1\mathrm{kbps}) & p < 0.02 \end{cases} \quad (2)$$

In Eq. 2, $S(i)$ is the TCP throughput at time i that is used by TFRC, and p is the packet loss rate. The relationship between the packet loss rate and the estimation of the sending rate in Eq. 2 can be summarized as follows:

1) If p is larger than 10%, then the sending rate is decreased.
2) If p is less than 2%, then the sending rate is increased.
3) If p is between 2% and 10%, then the sender maintains the previous sending rate.

In WebRTC, the receiver-side monitors both the video stream and changes in frame delay. According to [9], the receiver side utilizes an overuse detector and estimates the receiving rate based on Eq. 3. In this equation, $Ar(i-1)$ is the previous receiving rate estimate, η is the receiving rate increase factor, $R(i)$ is the current incoming rate, and α is the incoming rate decrease factor. α is a fixed value (normally chosen between 0.80 and 0.95) while the receiving rate estimate is constrained by the following condition [11]: $Ar(i) < 1.5 \times R(i)$

$$Ar(i) = \begin{cases} \eta Ar(i-1) & \text{Increase} \\ \mathrm{Ar}(i-1) & \text{Hold} \\ \alpha\ R(i) & \text{Decrease} \end{cases} \quad (3)$$

The overuse detector monitors the changes in the frame delay and the state of the bottleneck to produce one of the following three signals: Overuse, Underuse, or Normal. Algorithm 1 is derived from the reference codes available at [3], which is used to detect overuse.

We can interpret the stream flow between receiver and sender from Figure 1. The media stream flow is sent by the sender to the receiver in RTP packets while the arrival-time filter computes the queuing delay variation, T. The remote rate region sets the threshold based on Algorithm 2. There are three region statuses that are set, based on whether we are far from congestion (MaxUnknown), close to congestion (NearMax), or congested (AboveMax). According to [11], the default threshold value used by the Chrome browser is 25/60 ms, but when we are close to congestion, the threshold is halved. Then, the over-use detector receives the queuing delay variation T from the arrival-time

filter and the threshold from the remote control region to produce the appropriate signal. Based on the over-use detector signal, the remote rate controller estimates the receiving rate based on Eq. 3, and sends it to the sender via REMB messages with the packet loss rate value. On the sender-side, the sender receives the packet loss rate and the receiving rate estimate. Sender-side calculates the sending rate according to Eq. 1, and compares this rate with the TFRC throughput before setting the sending rate for the next stream segment.

Algorithm-1: Overuse detector pseudo code

```
1   //Overuse Detector
2   IF number of deltas < 2 THEN
3     Overuse detector state = Normal
4     /*compute T, which is the offset estimate multiply of number of deltas*/
5     T = min (number of deltas, 60) * offset
6   //compare T with threshold (γ)
7   IF T > γ
8     // Initialize the timer of time of overusing (the time spent in overusing period)
9     IF time of overusing = -1 THEN
10      time of overusing = ts _deltas/2
11    ELSE
12      time of overusing = time of overusing + ts _deltas
13      // increase Overusing counter by 1
14      Overuse counter = +1
15    END IF
16    /* check if the time of overusing counter reaches Overusing time threshold
17    and check Overuse counter too */
18    IF time of overusing > Overusing time threshold AND Overuse counter >1
19      /* check whether if the current offset estimate larger than the previous one
20      or not */
21      IF current offset > previous offset THEN
22        time of overusing = 0 //reset the counter
23        //reset the counter
24        Overuse counter = 0
25        Overuse detector state = Overuse
26      END IF
27    END IF
28  ELSE IF T < - γ
29    //reset the counter
30    time of overusing = 0
31    Overuse detector state = Underuse
32  ELSE
33    time of overusing = 0
34    //reset the counter
35    Overuse counter = 0
36    Overuse detector state = Normal
37  END IF
```

Algorithm-2: Rate control region algorithm

```
IF Control-Region = MaxUnKnown THEN
   /*γ is threshold, γ' is default threshold; i.e. chrome threshold 25/60 ms */
   γ = γ'
END IF
//close to congestion
IF Control-Region = AboveMax //congestion occurs OR Control-Region = NearMax THEN
   γ = γ/2   //halved the threshold
END IF
```

The dynamic behavior of the WebRTC congestion control has a significant impact on WebRTC's performance as perceived by the user. Authors in [9] showed that a TCP flow starves a WebRTC flow when they share the same link due to the threshold mechanism employed by GCC's receiver side. The researchers found that the threshold, γ, has a significant impact on the dynamics of the receiving rate *Ar*, channel utilization, queuing delay, loss ratio, and the fairness of the WebRTC flow when coexisting with a TCP flow. Therefore, authors in [9] have suggested not to use the default value of the threshold, which is γ = 25/60 ms. As a remedy, authors in [10] proposed a mathematical model for adapting γ dynamically to provide fair coexistence of WebRTC flows with TCP flows.

PROPOSED DYNAMIC ALPHA MODEL

As we discussed in the previous section, receiver-side of the WebRTC attempts to reduce the receiving rate estimate by multiplying the current incoming rate with a fixed value, called alpha (α). Value of fixed alpha is within the interval [0.80, 0.95]. In the existing reference implementation [3], the receiver-side decreases its rate without taking into account the difference between the queuing delay variation (*T*) and the threshold (γ) that the overuse detector has provided (see Algorithm 1). In other words, the receiver has to decrease its rate by the same fixed α regardless of whether the queuing delay is slightly smaller or significantly greater than the threshold. As we explained before, intuitively, this is not optimal; the rate decrease should not be fixed, but should be dynamic depending on whether the queuing delay is slightly smaller or significantly greater than the threshold. Dynamically setting the rate decrease should positively affect the overall performance in terms of incoming bit rate. Therefore, we propose to adjust the value of α, termed *dynamic alpha*, according to the actual difference between the queuing delay variation (*T*) and the threshold (γ) set by the remote rate region, as presented in Eq. 4.

$$Dynamic\_Alpha = \sqrt{e^{-\left(\frac{(T-\gamma)}{T}\right)}} + a \qquad (4)$$

In the above equation, *a* is a scalar parameter calculated by Eq. 5, where $b = \sqrt{e^{-\left(\frac{(T-\gamma)}{T}\right)}}$.

$$a = \begin{cases} \frac{0.99-b}{1.3} \; ; \; if \; Remote \; Region = AboveMax \; or \; MaxUnknown \\ \frac{1-b}{1.14} \; ; \; if \; Remote \; Region = NearMax \end{cases} \qquad (5)$$

Our proposed dynamic alpha allows the receiver-side to decrease its receiving rate based on the magnitude of difference between the queuing delay variation (T) and the threshold (γ). Therefore, a small difference between T and γ results in a dynamic alpha near unity; i.e., the dynamic alpha can take the value of 0.99 or 0.98 rather than using 0.85 or 0.95 in the fixed alpha case. In other words, the dynamic alpha allows a small decrease of the receiving rate estimate for a small value of T, which leads to a higher incoming rate for the bitstream. In contrast, when T significantly exceeds γ, the dynamic alpha assumes a smaller value; i.e., the dynamic alpha can be reduced to 0.90 or even 0.85 for the large differences.

It should be noted that in Eq. 4 we are using an exponential term $e^{-\left(\frac{(T-\gamma)}{T}\right)}$ as opposed to directly using the linear term $T-\gamma$. The reason is to avoid constantly changing the incoming rate due to frequent and fast fluctuations that happen in the linear term, because of temporary and short network condition changes. The exponential term will smoothen these fast fluctuations and will ensure fewer changes in the incoming rate.

In Eq. 5, we use two equations to calculate a by using two different values (1.30 and 1.14) in order to keep the dynamic alpha within [0.09, 0.99]. Based on the network status, remote rate region sets two values for threshold - the default threshold (γ), and the default threshold divided by two ($\gamma/2$). Therefore, our scalar parameter a changes to adapt these two threshold values. To verify our model practically, we applied the following steps:

1. We observed the T value (the queuing delay) for 20 sessions with the fixed alpha
2. We chose different values for T related to its observed max and min values
3. We used the different T value with the two values of the threshold.

Using the above steps, the expected dynamic alpha values are shown in Table 1.

Table 1. Different assumed T values and the expected dynamic alpha

threshold (γ)=25	
T	*dynamic alpha*
26	0.99
45	0.95
100	0.92
1000	0.91
5000	0.90
threshold (γ)=25/2	
T	*dynamic alpha*
13	0.99
20	0.96
45	0.93
100	0.91
5000	0.90

To implement our algorithm, we modified some parts of the WebRTC reference code [3]. We defined a new object called dynamic alpha, and we set the initial value to 0.90 in the *../src/webrtc/modules/remote_bitrate_estimator/aimd_rate_control.cc* file. To make the object visible, we also added it to the header file *aimd_rate_control.h*. Since our algorithm uses some values from the overuse detector, such as threshold and delay, we made those objects global in order to allow for using them by the *aimd_rate_control.cc* file in our bitrate calculations. We also created a new header file called *global.h* to define the global objects, which are the saved values of γ and T, and used in *overuse_detector.cc*. Finally, we implemented Eq. 4 in the *aimd_rate_control.cc* file.

EXPERIMENT SETUP AND RESULTS

The experimental test-bed we used consists of two laptops, each side acting as an independent client with our modified WebRTC code. A Dell laptop (Intel Core i7-000 M, 8 GB RAM, Windows 7 64-bit) was designated as the sender, and we tagged it as Client-1; A Lenovo ThinkPad W540 laptop (Intel Core i7-7400MQ - 8 GB RAM, Windows 7 64-bit) was designated as the receiver, and we tagged it as Client-2. We used NEWT [16], a software-based network emulator, to emulate the real-world connection behavior. We also utilized a virtual video camera tool [17] to inject the video sequence and to ensure that the video stream is consistent. In order to support the experiments' reproducibility, we used a 30seconds long publicly available Wave Hand video clip [18].

We used both constrained and unconstrained network scenarios to obtain the results for comparison with the existing GCC model. The purpose of the unconstrained network test is to analyze the WebRTC behavior in an uncontrolled environment with no constraints. Based on this test case, we chose which network attributes would work in the next phase, and we considered this as the reference case. For a connection speed at approximately 75 Mbps, we observed the WebRTC behavior for GCC and our proposed model by repeating the scenario for both models and collecting the performance metrics. We repeated each scenario twice - first time with the original GCC and second time with our proposed dynamic alpha model. We observed the overuse detector states to make sure that congestion occurred during the session(s). We eliminated the sessions that did not contain an overuse signal because our proposed method is applicable for scenarios involving congestion. To emulate the constrained network, we set various network conditions to investigate how WebRTC with GCC and with the proposed model adapts to varying available bandwidth, packet loss, and propagation delay. Table 2 shows the test cases and respective network attributes for the constrained scenarios.

Table 2. Selected attributes for constrained network scenarios

Test case Number	Bandwidth (kbps)	Packet Loss Rate	Propagation Delay (msec)
1	50, 200, 500, 1000	none	none
2	unconstrained	2%, 5%, 10%, 12%	none
3	unconstrained	none	50, 200, 500, 1000
4	500, 1000	5%, 10%	50, 250

In Figure 2-4, we show the performance comparison for the unconstrained scenario. Figure 2 shows the incoming rate experienced by the receiver side for the unconstrained network scenario. We ran 12 video sessions where duration of each video session was two minutes. For each session, we computed the average for each of the evaluation criteria. From Figure 2, it is evident that the incoming rate is increased by 33% on average. This higher incoming rate is due to the different values that are assumed by the dynamic alpha.

The performance of RTT and packet loss fraction are shown in Figure 3 and Figure 4, respectively. According to [14] and [15], packet loss fraction is computed based on the expected number of packets since the last time frame and the total number of packets received since the last time frame. In Figure 4, we can see that the number of packet loss fraction is about the same, occasionally a few more in dynamic alpha method, compared to the fixed alpha method. However, our method achieves a lower RTT, 16% on average, as shown in Figure 3. Since the results reported in Figure 2-4 are based on the unconstrained network scenario, in Figure 3 and Figure 4, we see a slight increase in RTT and packet loss for specific sessions in the dynamic alpha compared to the fixed alpha due to the time of day when we run our experiments. Variation in reported results for both models increases during the peak hours. For example, experiments on a weekend at 11am show more consistency and expected behavior in incoming rate, RTT, and packet loss, compared to test runs on a weekday 11am. This behavior is consistent with the observations reported in [13].

In Figures 5-16, we show the performance comparison for the constrained scenarios based on Table 2. For each of those scenarios, we collected data for three 30 seconds long WebRTC video sessions. We set the same video sequence over the virtual cam. Here we present the average of the performance metrics from all three sessions for each scenario, and compare them with the original WebRTC receiver's rate model.

WebRTC with varying bandwidth constraints – In this test scenario, we varied bandwidth capacity (50 kbps to 1000 kbps) with no packet loss rate and delay constraints in order to investigate the impact of network bandwidth availability in the GCC with the fixed alpha and the dynamic alpha models. Figure 5 shows the incoming rate that is experienced by the receiver side. We observed that the dynamic alpha model gives a higher incoming rate constantly compared to what we experienced with the fixed alpha. Our proposed dynamic alpha also decreased the round trip time (see Figure 6). Moreover, in Figure 7, we illustrate that the dynamic alpha decreased the packet loss fraction by 29.4% when the bandwidth capacity was at 50kbps. However, with the bandwidth capacity at 200/500/1000kbps, the packet loss fraction tends to increase slightly.

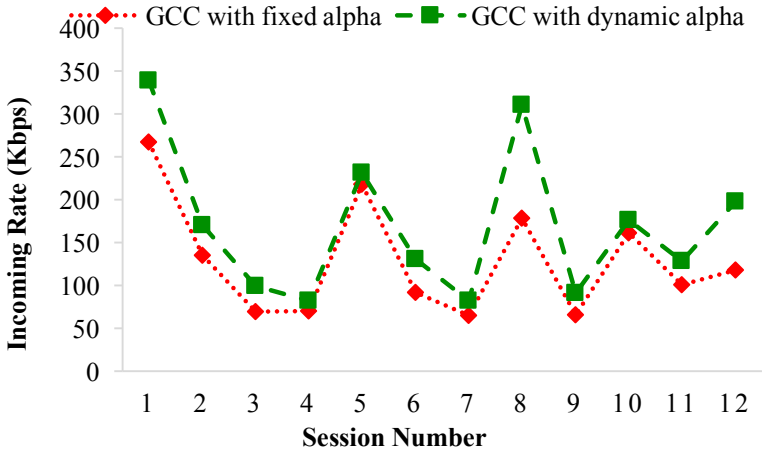

Figure 2. Average incoming rate - Unconstrained scenario

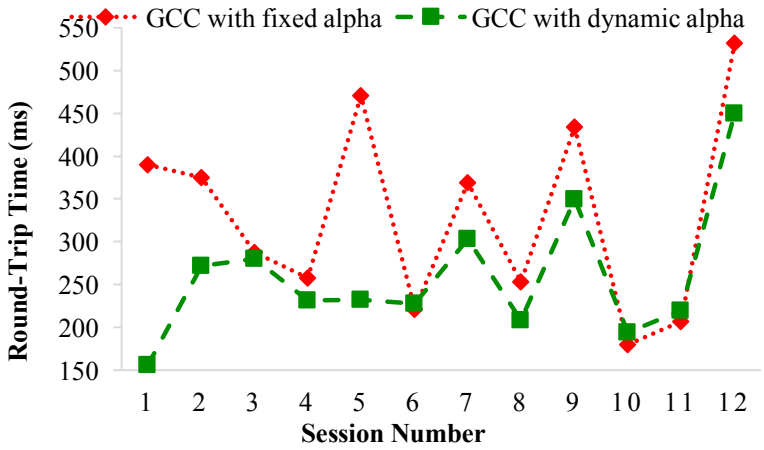

Figure 3. Average round trip time - Unconstrained scenario

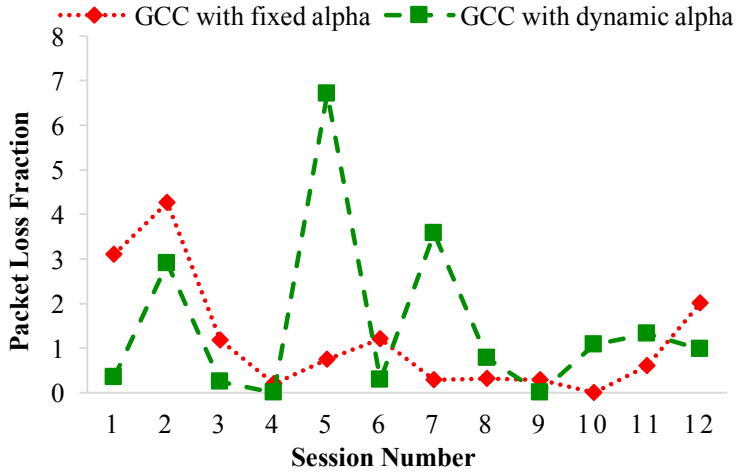

Figure 4. Average packet loss fraction - Unconstrained scenario

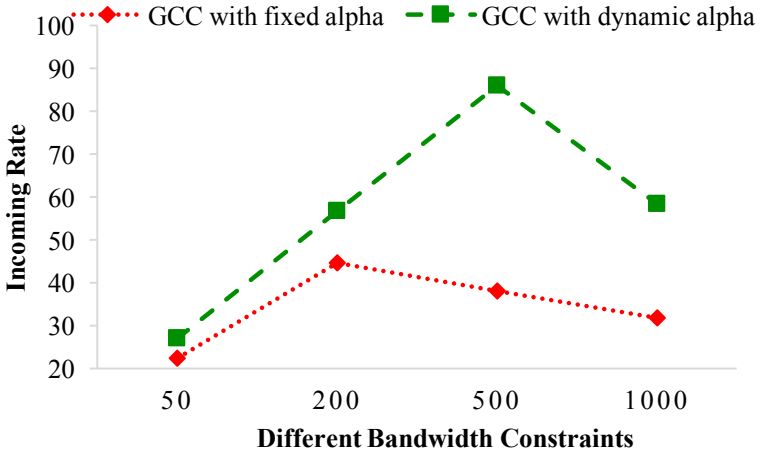

Figure 5. Incoming rate under different bandwidth constraints

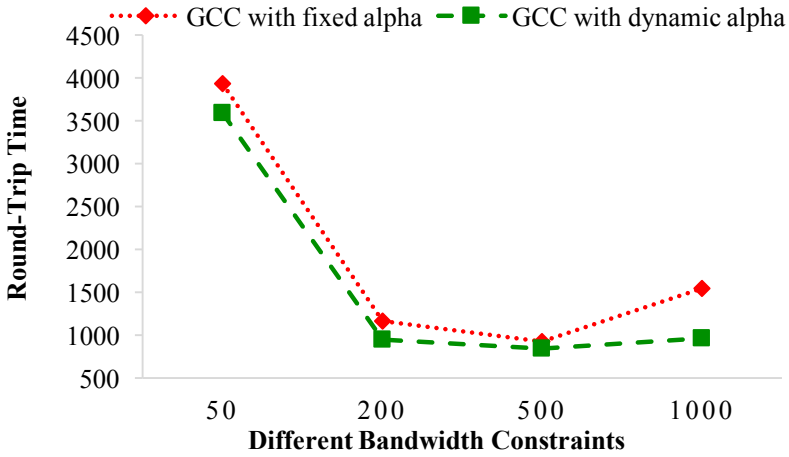

Figure 6. RTT under different bandwidth constraints

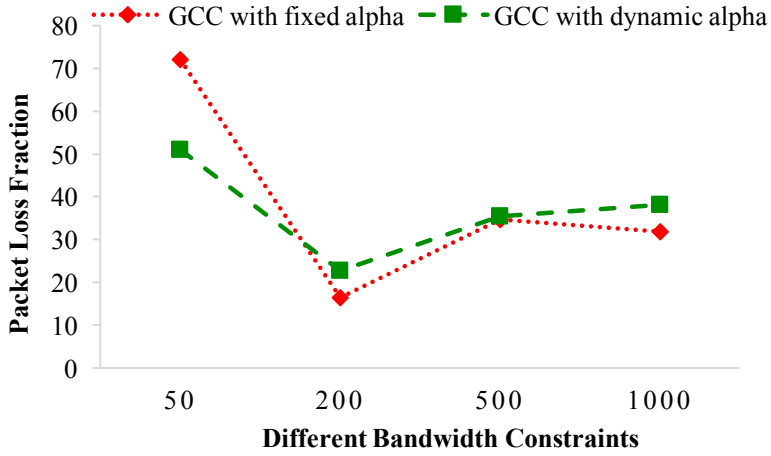

Figure 7. Packet loss fraction under different bandwidth constraints

WebRTC with varying packet loss rate constraints – In this test case scenario, we introduced random packet losses in our test-bed. The packet loss rate varied from 2% to 12%. Based on the experimental setup, dynamic alpha gave a much higher incoming rate compared to the fixed alpha (see Figure 8). In Figure 9 and Figure 10, we show that the proposed dynamic alpha method decreases the packet loss fraction and RTT for different packet loss rates, respectively.

WebRTC with two-way propagation delay constraints – This scenario was aimed at investigating how the WebRTC's receiving rate reacts to different latencies. Figure 11 illustrates the average incoming rates for the three sessions under different delay constraints. As shown in Figure 11, the proposed dynamic alpha increases the incoming rate in general. The increased amount is approximately 4% to 23%. Packet loss fraction is reduced (approximately 13% to 29%) in the case of dynamic alpha compared to the fixed alpha (see Figure 12). As shown in Figure 13, our proposed dynamic alpha decreases the round trip time in all the cases. It reduced round trip time by 75% over the network with a 50ms delay and approximately 26% with a 250ms and 500ms delay. The dynamic alpha cannot give the same amount of reduction when the delay is 1000ms; however, it is still able to reduce RTT by 1%.

WebRTC with varying Bandwidth, Packet Loss Rate, and Delay constraints – In this final test scenario, we present more extensive test results. This scenario is aimed at investigating how the WebRTC's receiving rate reacts to various network conditions. We merged different bandwidth constraints with different packet loss rate and two-way delay constraints. This test case consists of two sub-test cases: a) we set available bandwidth to 500 kbps, packet loss rate to 5%, and propagation delay to 50ms, and b) we increase available bandwidth to 1000 kbps, packet loss rate to 10%, and propagation delay to 250ms. We present these results in Figure 14 to Figure 16. Figure 14 shows that the proposed dynamic alpha improved the incoming rate by around 19% in both sub-test cases. In Figure 15, we can see that when the bandwidth capacity is 500kbps with 5% packet loss rate and two-way delay is 50ms, the packet loss fraction is increased in our model. However, when we increase the bandwidth capacity to 1000 kbps, the packet loss fraction is decreased compared to the dynamic alpha model. Finally in Figure 16, test case (a) shows slight increase in the RTT by 2%, whereas, test case (b) illustrates significant decrease in RTT by 30%.

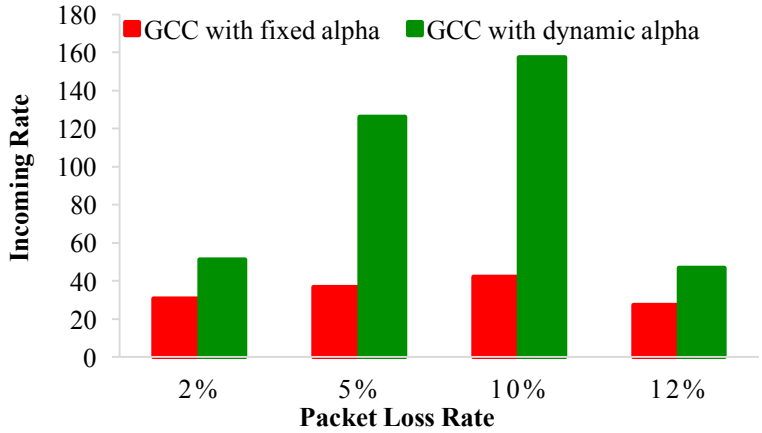

Figure 8. Incoming rate under different packet loss rate

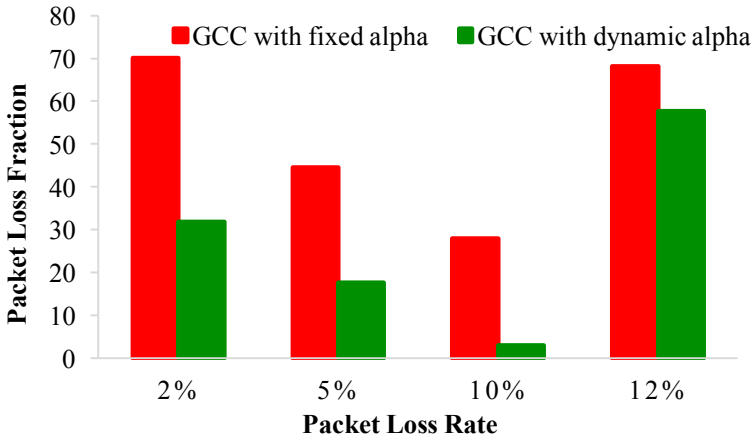

Figure 9. Packet loss fraction (number of packets) under different packet loss rate

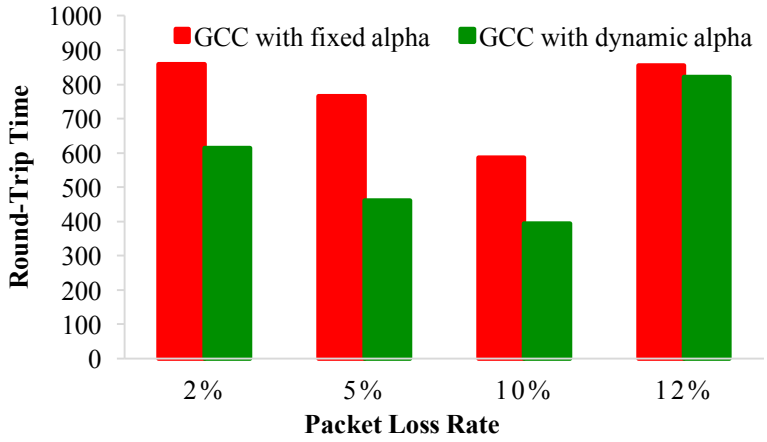

Figure 10. Round trip time under different packet loss rate

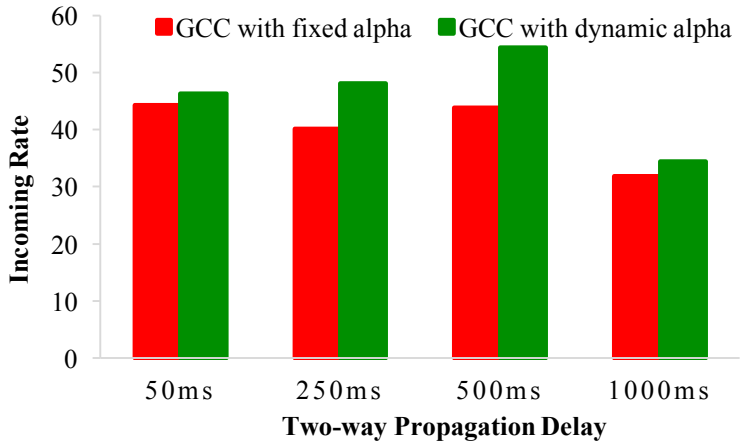

Figure 11. Incoming rate under different two-way delays

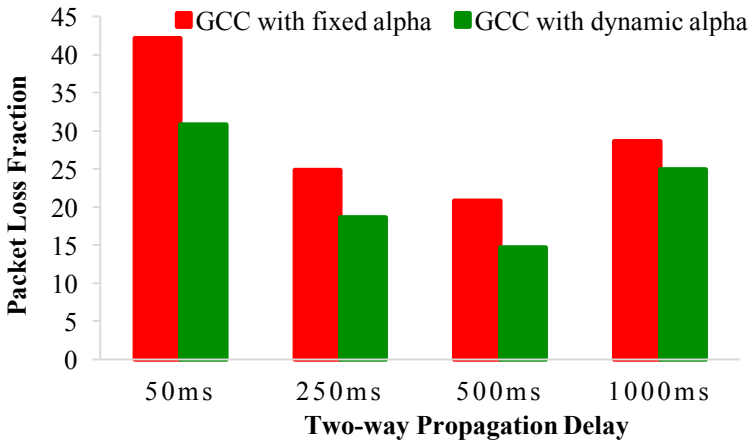

Figure 12. Packet loss fraction under different two-way delays

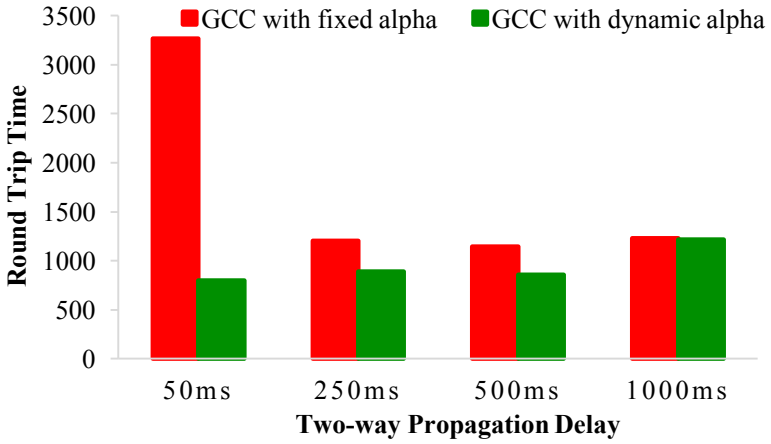

Figure 13. Round trip time under different two-way delays

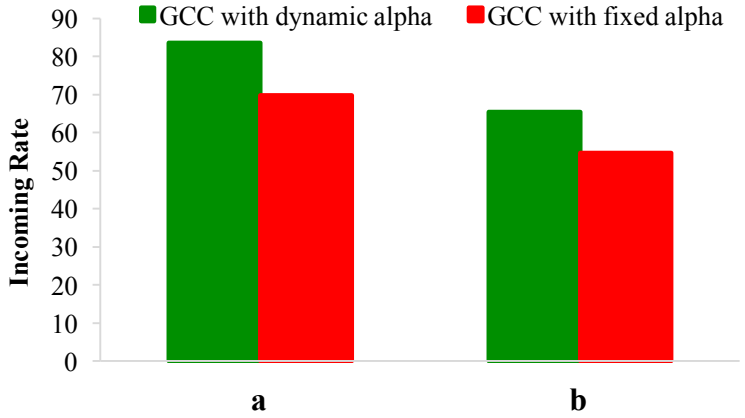

Figure 14. Incoming rate under different network constraint

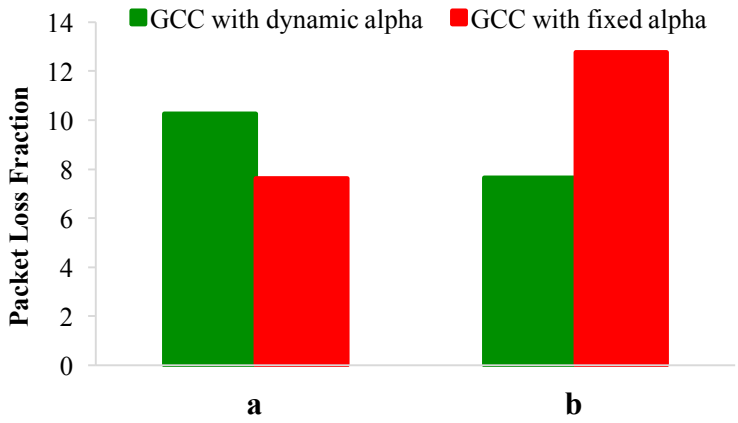

Figure 15. Packet loss fraction under different network constraints

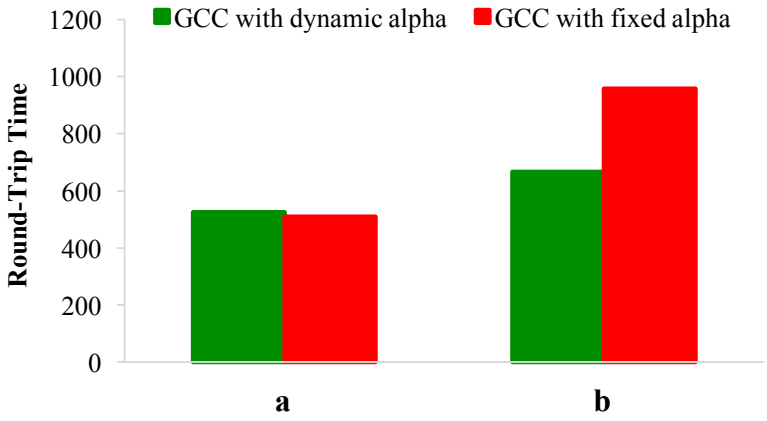

Figure 16. Round trip time under different network constraints

CONCLUSION AND FUTURE WORK

In this paper, we proposed a rate adaptation model for WebRTC receiver-side controller. In order to observe the performance of the proposed rate adaption model, we carried out experiments using a real network and obtained extensive performance measurements. We performed our experiments under both unconstrained and constrained networks. We also compared our model with the Google rate adaption model. Existing Google congestion receiver controller model allows the receiver-side controller to decrease its receiving rate by a fixed amount when congestion occurs. Whereas, our proposed model is able to determine accurate estimates of the receiving rate for the WebRTC receiver-side controller. Based on the experimental results, we found that WebRTC with a dynamic alpha improves overall performance regardless of the network bandwidth availability and network delays. For future work we note that our model was designed based on the Chrome threshold, which is 25/60ms. Therefore, our model may require modifications for compatibility with other browser thresholds. Comparison and/or combination with other suggested algorithms, such as Low Extra Delay Background Transport (LEDBT) and Datagram Congestion Control Protocol (DCCP), and also performing the experiments in a real Internet environment for prolonged amount of time will further highlight the benefits and limitations of our proposed approach. Finally, further evaluation of our model in terms of the subjective video quality can help determine how much impact our proposed approach has on the user's quality of experience.

REFERENCES

[1] Alonso, M., Geiger, G., & Jorda, S. (2004). An Internet Browser Plug-in for Real-time Audio Synthesis. *International Conference on Web Delivering of Music*, 23-26.

[2] Y. Xu et al. (2012). Video telephony for end-consumers: measurement study of Google+, iChat, and Skype. *ACM Internet Measurement Conference*, 371–384.

[3] Webrtc.org, (2016). WebRTC. Retrieved 10 March 2016, from http://www.webrtc.org.

[4] Johnston, A., & Burnett, D. (2014). *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. St. Louis, MO: Digital Codex LLC.

[5] Lundin, H., Holmer, S., & Carlucci, G., et al. (2015). A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. Retrieved 10 March 2016, from https://tools.ietf.org/html/draft-alvestrand-rmcat-congestion-03.

[6] Welzl, M. (2005). *Network congestion control: managing internet traffic*. John Wiley & Sons.

[7] Handley, M., Floyd, S., Padhye, J., and Widmer, J. (2008). RFC 5348 - TCP Friendly Rate Control (TFRC): Protocol Specification. Retrieved 10 March 2016, from https://tools.ietf.org/html/rfc5348.
[8] VishnuVardhan, S., & Chenna Reddy, P. (2012). Congestion Control in Real-Time Applications. *International Journal of Computer Applications*, 51(3), 33-37.
[9] De Cicco, L., Carlucci, G., & Mascolo, S. (2013). Understanding the Dynamic Behavior of the Google Congestion Control for RTCWeb. *International Workshop in Packet Video (PV)*, San Jose, CA, USA.
[10] Carlucci, G., De Cicco, L., & Mascolo, S. (2014). Modeling and control for web real-time communication. *IEEE Annual Conference in Decision and Control (CDC)*, Los Angeles, CA, USA.
[11] De Cicco, L., Carlucci, G., & Mascolo, S. (2013). Experimental investigation of Google congestion control for real-time flows. *ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, USA.
[12] Singh, V., Lozano, A. A., & Ott, J. (2013). Performance analysis of receive-side real-time congestion control for WebRTC. *International Workshop in Packet Video (PV),* San Jose, CA, USA.
[13] Sessini, P., & Mahanti, A. (2006). Observations on round-trip times of TCP connections. *Simulation Series*, 38(3), 347.
[14] Holmer, S., Shemer, M., & Paniconi, M. (2013). Handling packet loss in WebRTC. *ICIP*, 1860-1864.
[15] H. Schulzrinne et al. (2003). RTP: A Transport Protocol for Real-Time Applications. *RFC 3550,* Standard.
[16] Software Informer, Network Emulator for Windows Toolkit. Get the software safe and easy. Retrieved 10 March 2016, from http://network-emulator-for-windows-toolkit.software.informer.com/.
[17] Web Solution Mart (2016). Virtual Webcam. Retrieved 10 March 2016, from www.download3k.com/Internet/Instant-Messengers-Chat/Download-Virtual-Webcam.html.
[18] Fake Webcam - Play video as webcam. Wave Hand Clip. Retrieved 10 March 2016 from http://www.fakewebcam.com/avatar.asp.
[19] Groups.google.com. (2015). Google Groups. Retrieved 2015, from https://groups.google.com/forum/#!forum/discuss-webrtc.
[20] Tools.ietf.org. (2003). RFC 3448: TCP- Friendly Rate Control (TFRC): Protocol Specification. Retrieved from https://www.ietf.org/rfc/rfc3448.txt
[21] Tools.ietf.org. (2008). RFC 5348 - TCP Friendly Rate Control (TFRC): Protocol Specification. Retrieved from https://tools.ietf.org/html/rfc5348.