uOttawa

L'Université canadienne
Canada's university

School of Electrical Engineering and Computer Science

Dynamic Alpha Congestion Controller for WebRTC

By

Rasha Jamal M. Atwah

Submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of

Master of Science in Systems Science

Supervisor: Dr. Shervin Shirmohammadi

University of Ottawa
Ottawa, Ontario
18^{th}December 2015

Abstract

Video conferencing applications have significantly changed the way in which people communicate over the Internet. Web real-time communication (WebRTC), drafted by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF), has added new functionality to web browsers, allowing audio/video calls between browsers without the need to install any video telephony applications.

The Google Congestion Control (GCC) algorithm has been proposed as WebRTC's receiver congestion control mechanism, but its performance is limited due to using a fixed incoming rate decrease factor, known as an *alpha* (α). In this thesis, we have proposed a dynamic alpha model to reduce the receiving bandwidth estimate during overuse, as indicated by the overuse detector.

Experiments using our specific testbed show that our proposed model achieves a higher incoming rate and a lower Round-Trip Time while slightly increasing the packet loss rate in some cases compared to fixed alpha model.

Our mathematical model proves that it is necessary to use an adaptive alpha α as the receiver-side controller**.** The experimental results show improvement in the term of incoming rate, Round-Trip Time, and packet fraction loss rate in some cases. Our model increases the amount of incoming rate and decreases Round-Trip Time and fraction loss.

Acknowledgments

On the eve of this succession, I express my gratitude to the Almighty Allah, who has bestowed upon me the strength and patience to complete my research successfully.

I am highly indebted to my supervisor, Dr. Shervin Shirmohammadi, for allowing me to pursue my graduate research with his team at the DISCOVER lab. I sincerely appreciate his support and guidance throughout this undertaking. I will always remain grateful to him for his patience, kindness, and encouragement.

Appreciation also goes to Dr. Razib Iqbal, now an assistant professor at Missouri State University, for taking the time to clarify my understanding of the concepts and for providing me relevant suggestions during the research and development phase of this thesis. Without his support, guidance, and motivation, this project would not have come to fruition at any point throughout my degree.

Last but not least, I must thank my parents, my kids Ajwan and Amjad, my brothers, and my sisters. Especial thanks to my brother, Abdulrahman, he left his life and stayed with me to help and support my study. These are the people who have poured me a bounty of affection, and I have no words to express my gratitude towards them.

Thank you,

Rasha Jamal M. Atwah

Ottawa, Canada

Table of Contents

List of Figures

List of Algorithms

List of Tables

Abbreviations

DCCP	Datagram Congestion Control Protocol
DSACK	Duplicate Selective Acknowledge
ECN	Explicit Congestion Notification
ERD	Early Random Drop
FTP	File Transfer Protocol
HTML5	Hyper Text Markup Language
IETF	Internet Engineering Task Force
LEDBT	Low Extra Delay Background Transport
NAT	Network Address Translation
PSNR	Peak Signal-to-Noise Ratio
RED	Random Early Detection
RTC	Real-Time Commination
RTCP	Real-Time Control Protocol
RTCWeb	Real-Time Communication for Web
RTO	Retransmission Timeout
RTT	Round-Trip Time
SACK	Selective Acknowledgment
SSIM	Structure Similarity
SWS	Silly Window Syndrome
TCP	Transmission Control Protocol
W3C	World Wide Web Consortium
WebRTC	Web Real-Time Communication

Chapter 1

Introduction

In this chapter, we define the problem and demonstrate our goal. We started with a brief introduction to the research area; we explain the motivation behind our work; we present the used methodology, and the challenges we face while working on our thesis, and we outline the general structure of the thesis. Finally, we present our work's contributions.

The enormous popularity of mobile technologies such as smartphones and tablets has encouraged competition amongst video conferencing system developers to implement Real-Time Communication (RTC) applications and to enhance further the RTC technology. Video conferencing systems have had a paradigm shift in our lives due to how much cheaper, faster, and more efficient. Skype and iChat are two of the most popular examples of video conferencing systems that allow RTC.

The Internet and RTC applications have made our world smaller. We have nearly instant communication with anybody in the world by using any real-time application software. People in any location can communicate via the Internet in various ways, including through social networks (like Facebook, Twitter, etc.), video/audio applications, and gaming. The business industry has also benefitted from real-time communication technology. Real-time communication applications have increased the efficiency of business communication, regardless of the company's size, the number of employees, or even where the employees are located.

Video conferencing technology became readily available to the public with affordable costs and acceptable video compression technology during the 1990s [31]. In 1992, CU-SeeMe was introduced. Figure 1 shows Global Schoolhouse students communicating via CU-SeeMe with a video frame rate of 3-9 frames per second [31]. In the 2000s, Skype was getting popularity as a free video conferencing application that worked via the Internet with low cost and low video quality. However, the current version of the Skype application is more efficient and much higher quality compared to the original version. In 2010, video conferencing developers extended the capabilities of video conferencing applications for use on mobile systems.

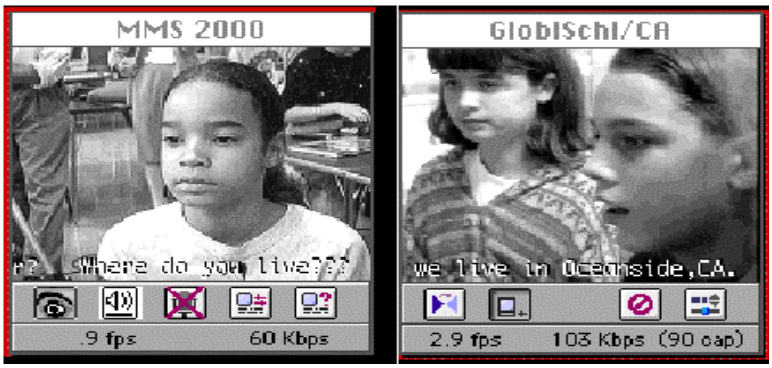

Figure 1: CU-SeeMe video call (1993).

In 2012, Web Real-Time Communication (WebRTC) was proposed as a new function that aimed to implement RTC into web browsers, enabling video/audio calls between two or more end users with no need to install third party plug-ins or specific applications. Some video/audio service providers require their users to install specific applications, such as Google Hangouts [12]. Also, some social networks require installing a specific application, such as Facebook, which provides video/audio chat services for their subscribers, but the users have to install third party plug-ins to improve the features of the existing browser [3]. Integrated WebRTC enhances web browser features without the risk of session set-up failures due to conflicting plug-ins.

Real-Time Communication on the Web (RTCWeb) is an open source project of the Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) team that aims to standardize protocols and infrastructures for WebRTC [28]. Web developers can add WebRTC features to their browsers via utilizing standard Hyper Text Markup Language (HTML5) with simple Javascript APIs [19]. As with other Internet services that are provided to the public, WebRTC must provide a certain level of quality to compete with other real-time interactive applications, particularly with the growth of mobile users that has rapidly increased the network load. WebRTC has to deal with quality from two different perspectives: the user's and the network. From the user's perspective, services such as WebRTC and real-time technology have to be provided with less packet loss and delays, along with a high incoming rate and video quality. From the network perspective, the Internet infrastructure relies on the shared link, and when the link reaches its capacity, WebRTC must have a mechanism that can deal with bottleneck problems and congestion to produce acceptable performance during a WebRTC video session. Therefore, IETF and W3C working groups have proposed a congestion control mechanism for WebRTC to satisfy the quality requirements for real-time applications [25]. Their solution was the Google

Congestion Control Algorithm, which has two main controllers: Receiver-side controller and Sender-side controller.

This thesis proposes a rate adaption model for the receiver-side controller under congestion that provides a high incoming rate with less delay. We implemented our model over unconstrained and constrained networks. We compared our model performance with Google's model regarding incoming rate, Round-Trip Time (RTT), and packet loss fraction, in most cases.

1.1 Motivation and Research Problem

WebRTC API contains a congestion control mechanism that is proposed as the Google Congestion Control Algorithm [25]. This master thesis focuses on the Google Congestion Control's receiver-side controller.

When congestion occurs, users experience packet losses or delays in picture or sound. The Google congestion receiver-side controller reduces the receiving rate to stabilize the algorithm. The receiver reduces the current incoming rate by a fixed amount. This reducing amount is called an alpha (α). Alpha is a decreasing factor that is typically chosen to be in the interval [0.80, 0.95]. The receiver controller multiplies the decreasing factor (alpha) by the current incoming rate to estimate the new receiving rate.

As we explained previously, alpha is fixed amount which means it does not take into account any parameters such as bottleneck network status or queuing delays. When congestion occurs, the queuing delay becomes greater than the threshold and regardless of how much this difference is, the receiver has to decrease its rate by fixed alpha. Therefore, the big difference between queuing delay and threshold signs the same decreasing amount (fixed alpha) same as the small difference. Thus, this reflects negatively on WebRTC performance as a general, and it presents in the form of a low-quality video session.

The overall aim of this thesis is to propose a dynamic rate adaption model that reacts with the network status and queuing delays rather than simply reducing the incoming rate by fixed alpha. The way to achieve this goal is proposing a dynamic alpha model can able to adapt to the network changes. Therefore, suggesting a dynamic alpha mathematical model will enhance WebRTC performance when the congestion occurs.

We implemented our model based on WebRTC reference code [29], and for evaluating the model, we assumed some scenarios to investigate how the receiver side will behave. The scenarios include realistic environment with constrained (some network constraints are set such as bandwidth capacity, packet loss rate, and delay), and unconstrained networks (no constraints

are set). Our experiments contain five test cases that are grouped into two based on whether the network is constrained or unconstrained. Over an unconstrained network, we only have one test case, and the rest test cases are over constrained network. The network parameters that are changed from test case to another are the bandwidth capacity, packet loss rate, and two-way delay. We evaluated our model according to different performance metrics, such as packet loss, RTT, and incoming rate. Matlab has been used to plot and analysis the experiment results. The analysis of the experimental results provided an overall evaluation of the WebRTC video performance.

During this research, the following research question was identified: What is the general impact of using a dynamic reduction model to estimate the receiving rate?.

To answer this question, the following things are needed:

- Observation of WebRTC behavior over a real environment with no constraints.
- Observation of WebRTC over a constrained network.
- Implementation of the rate adapting model in WebRTC reference code.
- Evaluation of the proposed model's impact on interactive, real-time WebRTC sessions.
- Comparing the performance metrics to the Google Congestion Control Algorithm's performance metrics.

1.2 Thesis Methodology

The main phases of our research methodology are shown in Figure 2, and we outline the main details for each phase.

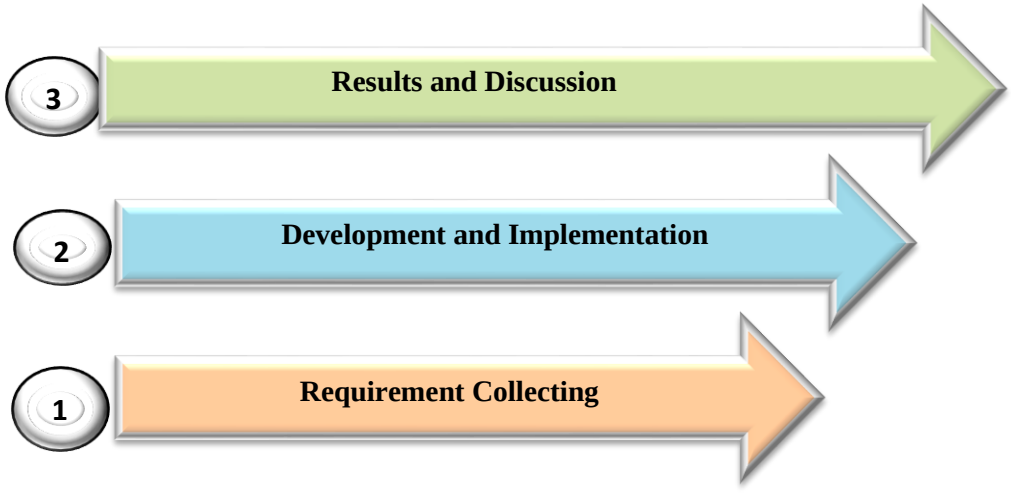

Figure 2: Thesis Methodology.

Phase 1: Requirement Collecting

In this initial step, we collected published literature related to WebRTC and the congestion control topic. We also defined our research area and proposed research questions to establish our main objectives. The research questions are:

- What are the problems highlighted by the previous researchers?
- What is the problem that has to be our main goal?
- What is the possible reason behind this problem?
- What are the parameters that have direct and indirect relations with the problem?
- How to combine this parameter in the form of a mathematical equation?

At the end of this phase, we had the main research goal and an initial image of what our system would be.

Phase 2: Development and Implementation

In this phase, we worked on a number of activities. We proposed a rate adaption model for the receiver-side controller in case congestion occurs. Furthermore, we investigate the efficiency of the proposed model over an unconstrained network, without any network conditions, and over a constrained network, with specific network conditions. We performed some activities in parallel, such as building the WebRTC environment and working on the mathematical equation. However, some of the activities relied on each other, and we worked on them task-by-task sequence, such as building the WebRTC and then implementing our model. The implementation phase took longer than the other phases for many reasons:

- We had to suggest different mathematical models and discuss our model in weekly meetings to explore various options as well as to improve the design.
- We spent significant time building the WebRTC and fixing the set-up errors.
- We had to rely on the IETF RTCWeb group members in the WebRTC Google discussion group [13] for clarifications for some of our building errors. We also had to design the experimental scenarios and test our model.

Phase 3: Results and Discussion

We used analytical methods to evaluate the proposed model and our statistical results. We compared our results with the Google's model. To share our findings with the research community, we published our initial results at the IEEE International Symposium on Multimedia (ISM) 2015 conference.

1.3 Challenges

Working with WebRTC as a new technology presents a number of challenges:

- Since WebRTC was first proposed in 2012, we did not find many documentations or previous literature that investigated and analyzed the Google Congestion Control. There is even a lack of analytical experiments regarding the general congestion control mechanisms and behaviors.
- Currently, WebRTC is still under development, and there have been many changes added to the WebRTC code that are not yet updated in the internet draft published in this document [25].
- WebRTC in this thesis was built according to the instructions guide that is available on the WebRTC official site [28]. With little experience with the WebRTC environment, we faced some technical problems related to the installation, which delayed setting up our testbed environment.
- Almost all documents regarding WebRTC dive into the JavaScript code and HTML5 without focusing on the other aspects of WebRTC, such as security issues and suggestions for improving the congestion control mechanism.

1.4 Thesis Structure

The Rest of the thesis is structured as follows:

- Chapter 1 is an introduction chapter that explains real-time communication and video conferencing, clarifies the challenges in this work, and summarizes the thesis methodology.
- Chapter 2 discusses the concept of congestion and presents the background for the work.
- Chapter 3 illustrates the Google Congestion Control and the latest research related to the topic.

• Chapter 4 describes the experimental environment setup and presents the implementations of the proposed rate adaption model.
• Chapter 5 explains the scenarios and presents their results.
• Chapter 6 concludes the thesis and illustrates the future work.

1.5 Contributions in this Thesis

The Google Congestion Control Algorithm for the receiver-side uses fixed alpha to calculate the receiving rate. In this thesis, we make the following contributions:

• We model a dynamic Alpha to compute the receiving rate. Since the receiver controller contains overuse detector which is responsible for determining whether the network overuse, underuse, or normal, the receiver can then take advantage of the overuse detector information, such as the difference between the threshold and the delay variation. This information helps to estimate its receiving rate in case of overuse. To the best of our knowledge, we are the first researchers who work on modeling alpha.

• We propose our mathematical model to estimate the receiving rate in case of overusing. Implementing this mathematical model increased the incoming rate, decreased RTT, and improved packet loss in most cases.

1.6 Research Publication

[1] Rasha Atwah, Razib Iqbal, Shervin Shirmohammadi, and Abbas Javadtalab, “Dynamic Alpha Congestion Controller for WebRTC”, IEEE International Symposium on Multimedia, Miami, USA, Dec14 – 16, 2015.

Chapter 2

Background

This chapter provides a brief background of congestion control concepts, rate adaption algorithms and earlier research findings. We explain TCP congestion control that is used to transmit reliable date and how it works. We illustrate some congestion control algorithms that are currently used in the Internet applications. Also, since the sender side in the GCC algorithm uses TCP-Friendly Rate Control (TFRC) to estimate the sending rate, we provided more details about the TFRC algorithm.

2.1 Congestion Control and Flow Control

There is no standard definition of congestion. According to [33], "Congestion occurs when resource demands exceed the capacity." Therefore, we can set our own definition from two perspectives: the network perspective, and the user perspective. From the network perspective, if we have two or more sources, and each source increases its sending rate beyond the link capacity, after a while the network will get congested. From the user perspective, congestion is a form of decreased quality, which appears in the form of delays or packet loss. To allow the user to use the network as efficiently as possible, and since the Internet provides the best effort service to its users, there have been many proposed congestion control technologies and mechanisms [30].

Congestion control or congestion avoidance is the process of controlling or avoiding the congestion to attain a higher throughput, lower loss ratio, and smaller delays with a lower average queue size. First, we must point out the difference between two terms: congestion control and flow control. Congestion control is for protecting the network from being congested; however, flow control is for protecting the receiver from overload by informing the sender to slow its sending rate [30].

There are few commonly used congestion avoidance mechanisms, such as Drop Tail, Random Drop, Early Random Drop (ERD), and Random Early Detection (RED), etc. Drop Tail is one of the simplest congestion avoidance mechanisms; however, it gives lower throughput. It is a first _in_first out (FIFO) queuing mechanism [9]. When the queue becomes full, the last arrived packets are dropped. In the ERD mechanism, when the queue size exceeds a certain threshold, the packets are dropped with a fixed drop probability [14]. RED is a congestion control mechanism

proposed in the early 1990s [9]; however, it is still being used today due to its efficiency and considerable throughput. [10]. In RED, after each packet arrives, the average queue size is computed, and the congestion is detected. There are two threshold levels on the average queue size: the Max threshold and the Min threshold. When the average queue size reaches the Min threshold, the arriving packets are dropped based on a creation probability. If the queue size is reached or begins to be greater than the Max threshold, the arriving packets are dropped [9].

2.2 Transmission Control Protocol (TCP) Congestion Control

Transmission Control Protocol (TCP) is a protocol for providing reliable data transmission across an unreliable IP communication channel [10]. TCP is the most widely used protocol on the Internet, and most internet applications, such as email, web browsers, and File Transfer Protocol (FTP) download, use TCP. Figure 3 illustrates the Internet architecture and the position of TCP.

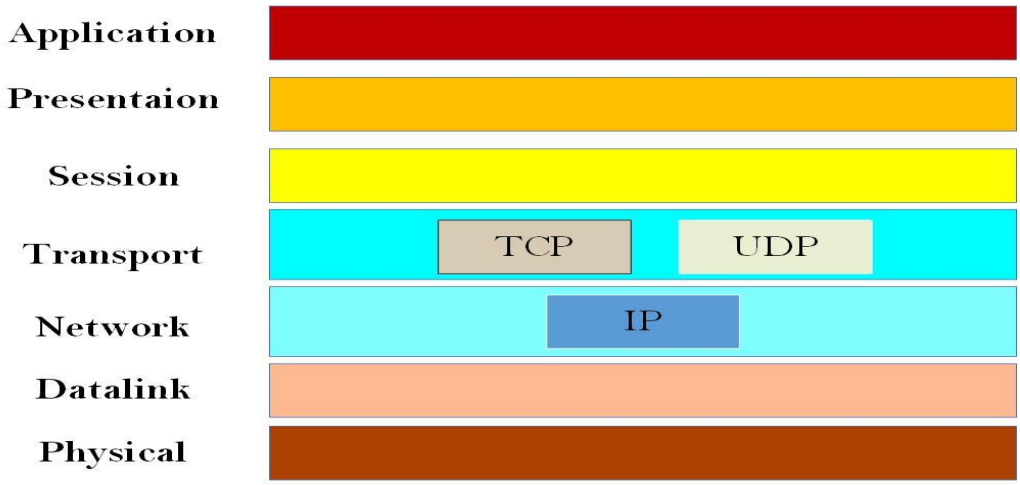

Figure 3: Internet layers with TCP/UDP protocol architecture.

The importance of TCP appears in the congested network when the packet loss increases; TCP recovers the lost packets by retransmitting them. The Internet had experienced a problem called *congestion collapse* for the first time in the 1980s, and RFC 896 [1]is one of the earliest documents that mentioned the *congestion collapse* term [10]. According to [10], congestion collapse occurs when the sender's rates increase, which leads to reducing throughput. When the congestion collapse occurs, the packets are lost, which means most packets in the network are retransmitted,

[1] https://tools.ietf.org/html/rfc896.

causing congestion. The solution to such a problem was described in RFC 5681[1], which proposed a slow-start algorithm and a sender rate adaption in case packet loss is detected and a retransmission at lower rates [21]. This solution was implemented in TCP itself and in any application that uses TCP. Approximately 127 documents have been published related to the TCP topic [10].

One of the earliest specifications of TCP was described in RFC 675[2] in 1974 by the name "Transmission Control Program." However, the most important TCP document was described in RFC 793[3] in 1981, and it is considered the foundation of TCP [10]. RFC 793 replaced the earlier document, RFC 675, due to its robustness. RFC 793 acknowledged the reliable full-duplex transmission and pointed out that retransmission timeout should be dynamically determined [10]. According to [10], all TCP details are described in the 'standard track' RFCs that are shown in Figure 4. We point out the main philosophy of the design of TCP in subsequent sections.

[1] https://tools.ietf.org/html/rfc5681

[2] https://tools.ietf.org/html/rfc675

[3] https://www.rfc-editor.org/rfc/rfc793.txt.

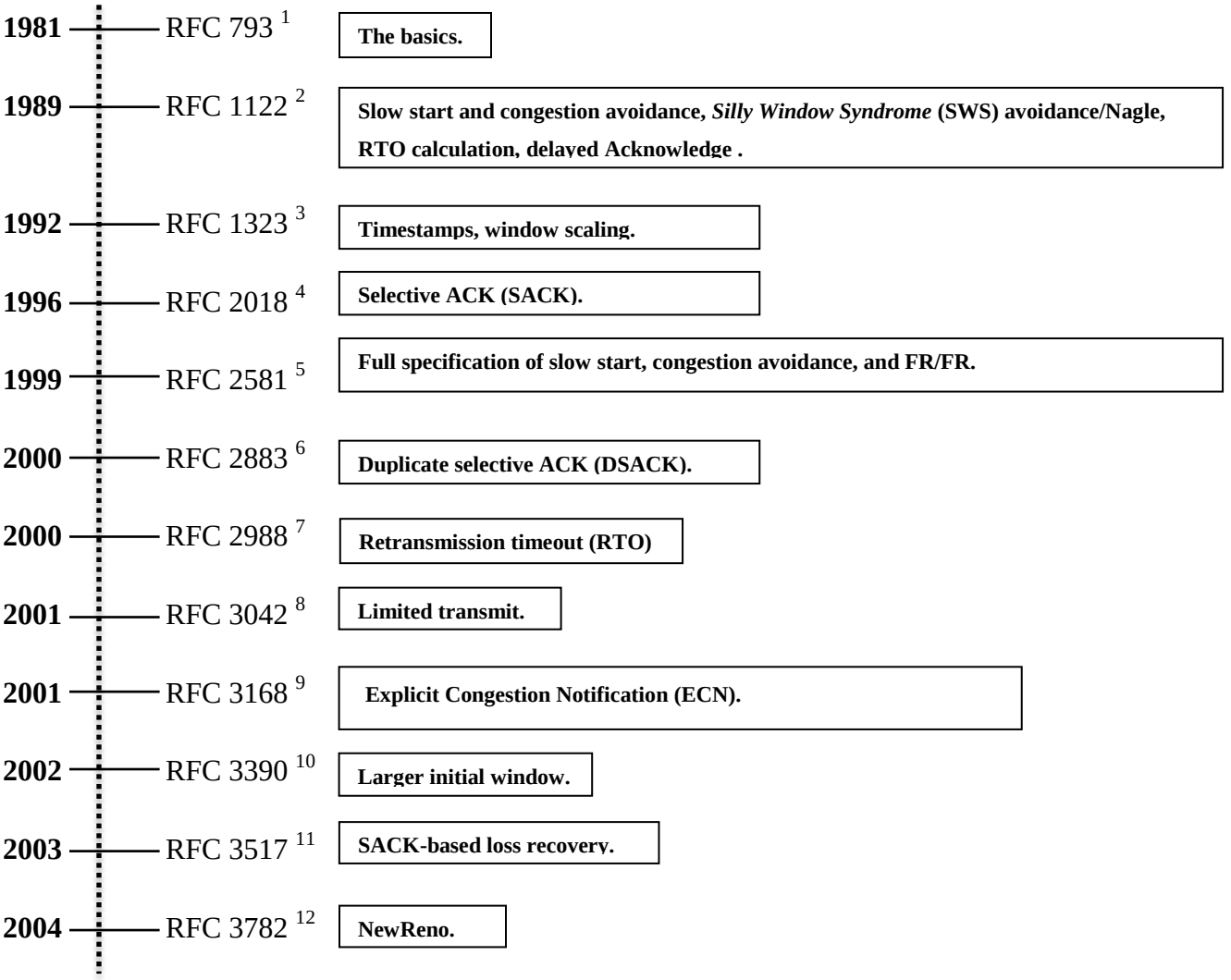

Figure 4: TCP specification (standard track).

[1] https://www.rfc-editor.org/rfc/rfc793.txt.

[2] https://tools.ietf.org/html/rfc1122.

[3] https://www.ietf.org/rfc/rfc1323.txt.

[4] https://tools.ietf.org/html/rfc2018.

[5] https://tools.ietf.org/html/rfc2581.

[6] https://tools.ietf.org/html/rfc2883.

[7] https://tools.ietf.org/html/rfc2988.

[8] https://tools.ietf.org/html/rfc3042.

[9] https://tools.ietf.org/html/rfc3168.

[10] https://tools.ietf.org/html/rfc3390.

[11] https://tools.ietf.org/html/rfc3517.

[12] https://tools.ietf.org/html/rfc3782.

This paragraph and the next one are based on [20]. TCP implements windows-based flow control on its transmission rate. Using windows-based or a sliding window means that the receiver allows the sender a certain amount of data ('window'), and the sender cannot send more than the full window before receiving the *acknowledgments* (ACK). TCP gives a sequence number to each *segment* and notifies the sender with the next expected sequence number that the receiver is willing to receive, as illustrated in Figure 5, which shows the TCP header. When the ACK is received, the received segment is removed from the transmission buffer. However, if the *retransmission timeout* (RTO) timer expires before the corresponding ACK is received, the TCP retransmits the lost segment [20]. The receiver can also notify the sender that a certain segment has not been received by sending a *negative acknowledgment* (NACK).

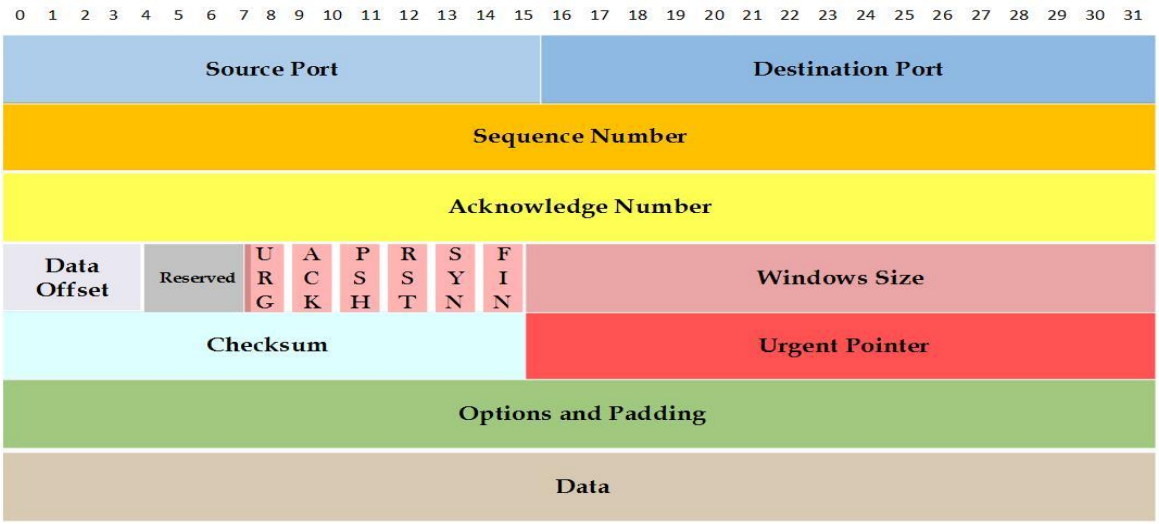

Figure 5: The TCP header.

To understand TCP congestion control mechanisms, we have to define the following terms to illustrate the four control algorithms that are implemented in TCP:

- The *congestion window* (cwnd) is a TCP state variable that limits the amount of data that can be sent by the TCP sender.
- The *receiver window* (rwnd) is the advertised receiver window.
- The *maximum segment size* (MSS) is the largest segment size that can be transmitted.
- The *slow-start threshold* (ssthresh) is another TCP state variable that determines whether TCP should do a slow-start or a congestion avoidance algorithm to control the data transmission. This variable was proposed in RFC 2581 in 1999. The initial ssthresh value may be arbitrarily high, but it can be reduced in response to congestion and is often set to

64 kb [10-20]. When cwnd < ssthresh, the slow-start algorithm is used. On the contrary, the congestion avoidance algorithm is used when cwnd > ssthresh. In the case of cwnd = ssthresh, the TCP sender may have the chance to use a slow-start or a congestion avoidance algorithm.

2.2.1 Slow Start and Congestion Avoidance

Slow-start and congestion avoidance algorithms are used to control data transmission between the TCP sender and receiver. The TCP sender implements the congestion window state variable (cwnd) to limit the amount of transmitting data; however, the TCP receiver implements the advertised window (rwnd) to limit the outstanding data [20]. The slow-start threshold (ssthresh) determines whether a slow-start or a congestion avoidance algorithm is used.

The slow-start algorithm is used when data begins transferring between two peers; if there is not enough information about the network conditions, such as available bandwidth, the TCP sender starts transmitting the data in a slower mode to avoid congestion [20]. The process starts with only one segment sent, and then one ACK is received. During the slow start, TCP increases cwnd by the MSS bytes for each ACK received, at most. When the slow-start threshold (ssthresh) is reached, or congestion is detected, the slow start ends. .

By combining the slow start and congestion avoidance algorithms, we get a single congestion control algorithm that works as follows:

1. The TCP sender works under the slow-start mode at the beginning of the connection, and it remains like that until ssthresh is exceeded or congestion is observed.
2. The TCP sender switches to the congestion avoidance mode until packet loss is detected.
3. The TCP sender goes back to the slow-start mode [10].

2.2.2 Fast Retransmit and Fast Recovery

When an out-of-order segment arrives, the TCP receiver sends a *duplicate* ACK to the sender to inform the sender that it received an out-of-order segment. According to [20], there are three reasons that can cause the duplicate ACKs: a dropped segment, re-ordering segments, or a replicated ACK or data segment. Receiving three duplicate ACKs is an indicator of packet loss, [10]. Therefore, when the sender receives the duplicate ACKs, the sender uses the fast retransmit algorithm to resend the lost segment and repair the loss. After the sender receives the duplicate ACK, it retransmits the lost segment even without waiting for the retransmission timer to expire. Until a non-duplicated ACK arrives, the fast-recovery algorithm is used to transmit the new data.

2.2.3 Various TCP Congestion Control methods

We give more details about some congestion control mechanisms such as slow start, congestion avoidance, and re-transmission. In this section, we set the main points of some of commonly used congestion control and avoidance mechanisms that have proposed for TCP/IP protocols, namely: Tahoe, Reno, New-Reno, SACK, and Vegas. According to [1], the paper starts with giving a brief look at each of the congestion control algorithms and noting some differences between each others. Here, we outline the main parts of reference [1].

TCP Tahoe is TCP congestion control algorithm that was suggested by Jacobson in [16]. TCP implements the principle of 'conservation of packets' by using the acknowledgment to clock outgoing packets and maintains a congestion window cwnd to reflect the network capacity. TCP Tahoe uses slow-start and congestion avoidance algorithms that we explain in details in the previous section. The main thing in Tahoe is that it detects packet losses by timeout. However, one of TCP Tahoe problems is that it takes a complete timeout interval to detect packet loss and sometimes, it takes even longer. Also, whenever a packet is lost, TCP Tahoe has to wait for the timeout and the pipeline is emptied which offers the main cost in high bandwidth delay links.

TCP Reno uses the basic principle of TCP Tahoe, but it detects the packet loss earlier, [17]. TCP Reno suggested an algorithm called, "Fast Retransmit" that is mentioned in the previous section. Another modification that is suggested by TCP Reno is avoiding reduce the congestion windows to 1 after a packet loss. When the packet losses are small, Reno works very well over TCP. However, with multiple packet losses in one window, TCP Reno is same as Tahoe in performance. Both of them cannot perform too well under high packet loss. Moreover, no received DACK if the windows are very small when the loss occurs. Thus, TCP Reno cannot effectively detect multiple packet losses.

TCP New Reno is a slight modification over TCP Reno that can detect multiple packet losses [23]. This modification is on the fast-recovery phase that allows multiple re-transmission. Furthermore, TCP New-Reno overcomes the problem faced by Reno of reducing the congestion window many times by avoiding exit fast-recovery phase until all outstanding data is acknowledged. The drawback of New-Reno is that it takes one RTT to detect each packet loss.

Selective Acknowledgment (SACK) is an extension of TCP Reno which the segment acknowledged selectively [2]. SACK does not send a new packet before checking which segment

unreceived and resend it [8]. SACK overcomes the problems faced by TCP Reno and TCP New-Reno which are a detection of multiple lost packet and re-transmission of more than one lost packet in one RTT. Since each ACK contains a block that describes the selective segment, the sender has the idea of which segments have been acknowledged. The major problem in SACK that is the selective acknowledgments are not provided by the receiver. Also, SACK is not very easy to incorporate in TCP.

TCP Vegas has modified extension of Reno. TCP Vegas still uses the other mechanisms of Reno and Tahoe. However, to prevent congestion in the network, TCP Vegas suggests three major changes:1) a modified slow start algorithm.2) a modified congestion avoidance, and 3) new re-transmission mechanism. Vegas modified slow start and congestion avoidance algorithms to be more accurate in measuring the available bandwidth more than Tahoe. Thus, it uses network resources efficiently. Also, the new re-transmission mechanism overcomes Reno's problem of not being able to detect packet loss when the window is small by preventing many of the coarse-grained timeouts. It checks if the current time segment transmission is greater than RTT or not. If the current time segment transmission > RTT, the Vegas immediately retransmits the segment with no need for waiting for three DACK or coarse timeout, [4].

2.3 TCP-Friendly Rate Control (TFRC)

According to [24], 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 [26]. 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 [24]. 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 Equation 2.1. 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 [15]:

$$X = \frac{S}{R\sqrt{\frac{2bp}{3}} + \{t\_RTO\ (3\sqrt{\frac{2bp}{8}})p\ (1+32\ p^2)} \qquad \textbf{(Equation 1.1)}$$

Where 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 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 [26]. According to [13], RFC 3448 is suited for many multimedia streaming applications. However, it is also used for the continuous flow of data [26]. The development of TFRC is still ongoing in the IETF group.

Chapter 3

The Google Congestion Control (GCC)

In this chapter, we explain the main components in the GCC algorithm. We show the rate adaption model for both the receiver-side and the sender-side. Furthermore, we present the algorithms for the overuse detector and the rate region to provide a deeper understanding for the overuse detector and rate region mechanisms

3.1 Introduction to Google Congestion Control for Real-Time Commination (WebRTC)

The increased popularity of the accessibility and mobility of multimedia and video conferencing applications has motivated the creation of browser-to-browser video conferencing technology that can compete with current standalone video conferencing applications. WebRTC is an effort to embed real-time communication capabilities into browsers and mobile applications via HTML5 tags and simple JavaScript APIs [19].

The long-term goal of WebRTC is to allow 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 to communicate in real-time, as long as both web browsers support WebRTC. WebRTC is open source and is currently implemented on three browsers: Google Chrome, Firefox, and Opera [28]. Without the need for any native applications, WebRTC offers many rich features, such as running across different operating systems and browsers, desktop and mobile compatibility, and support for multiple media sources.

Like any other real-time multimedia system operating over a best-effort network such as the Internet, WebRTC requires a congestion control mechanism that satisfies the congestion control requirements [18]. Requiring a congestion control mechanism is to ensure the smooth flow of media packets (audio, video, etc.) in the face of dynamically changing network parameters (packet loss rate, bandwidth, delay, etc.) so as to improve the user's experience. The IETF and W3C working groups have proposed the Google Congestion Control Algorithm (GCC) [25] as the current WebRTC congestion control mechanism. According to [11], GCC employs two main controller models: Receiver-side and Sender-side controllers. Figure 6 illustrates the main components of each controller, where the receiver-side controller itself is composed of the five

main components shown in Table 1: arrival-time filter, remote rate region, overuse detector, remote rate controller, and receiver estimated max bitrate (REMB) message processing.

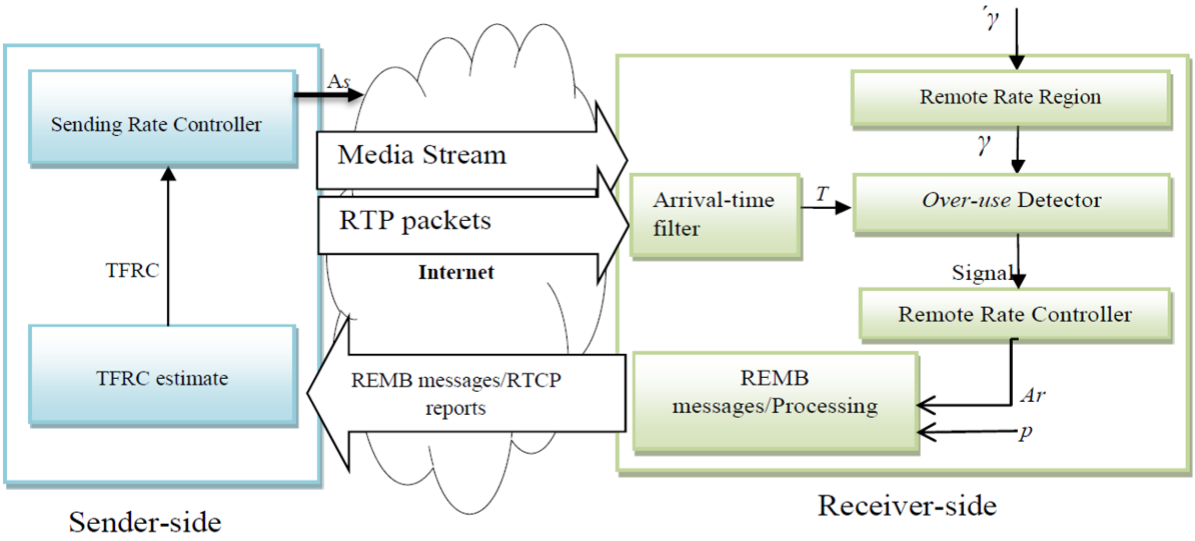

Figure 6: GCC Receiver-side and Sender-side components.

The sender-side consists of two main components: TCP-friendly rate control (TFRC) bandwidth estimation and the sending rate controller. The dynamic behavior of the WebRTC congestion control has a significant impact on WebRTC's performance as perceived by the user. The work in [11] is one of the earliest documents that focuses on understanding the dynamic behavior of GCC. It showed that TCP flow starves a WebRTC flow when they share the same link. The reason behind such issue is the threshold mechanism that is employed by the GCC's receiver side. The researchers found that a threshold $\bar{\gamma}$ has a significant impact on the dynamic of the receiving rate Ar(i). They showed the impact on some metrics such as the channel utilization, the queuing delay, the loss ratio, and the fairness of the WbRTC flow when coexisting with a TCP flow [11]. They suggested not to use the default value of the threshold $\bar{\gamma} = 25/60$ ms.

As a remedy, authors in [5] proposed a mathematical model for adapting the threshold, $\bar{\gamma}$, dynamically to provide fair coexistence of WebRTC flows with TCP flows. Their results showed that the proposed model can fairly share the bottleneck between WebRTC flows and TCP flows.

Component Name	Component Functions
Arrival-Time Filter	Estimating the queuing time variation T
Remote Control Region	Computing the threshold γ
Overuse Detector	Producing a signal (i.e. Overuse, Underuse, or Normal) based on the queuing delay variation T and the threshold γ
Remote Rate Controller	Estimating the next expected receiving rate (Ar)
REMB Messages Processing	Notifying the sender with the next expected receiving rate (Ar)

Table 1: Receiver-side controller's main components.

3.2 Google Congestion Control Algorithm

Congestion occurs when a data network's resources reach near their maximum capacity. According to [30], we can define congestion, from the end-user perspective, as a decrease in the service quality due to a high network load. Subsequently, we can use congestion control or congestion avoidance techniques to use the network as efficiently as possible. Therefore, congestion control is an important requirement for all applications that share the Internet [26]. The GCC is the proposed congestion avoidance algorithm for WebRTC, and it is a rate-based control algorithm; the sender is aware of its sending rate, and the receiver informs the sender of the next expected receiving rate to prevent the sender from exceeding this rate. GCC is an internet draft [25] that explains the congestion control implemented in WebRTC; however, we must note that there are many changes being made to WebRTC's current implementation that have not yet been added to this internet draft. The internet draft highlights two congestion control algorithms that are implemented on the receiver side and the sender side. The sender side is a loss-based module, while the receiver side is a delay-based module, as described next.

3.2.1 Sender-side controller

The sender is responsible for monitoring packet loss and adjusting the sending rate. The sender-side estimates the sending rate by using the TCP-friendly rate controller (TFRC) equation [6] based on the feedback from the receiver side. 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. As in [11], the sender side uses the packet loss rate to estimate the sending rate (As), which is given by Equation 3.1:

$$\boldsymbol{As(i)} = \begin{cases} \boldsymbol{max[S(i), As\ (i-1)(1-0.5p)]} & \boldsymbol{if\ \ p > 0.10} \\ \boldsymbol{As(i-1)} & \boldsymbol{if\ \ 0.02 < p < 0.10} \\ \boldsymbol{1.05[As(t-1) + 1kbps]} & \boldsymbol{if\ \ p < 0.02} \end{cases} \qquad \textbf{(Equation 2.1)}$$

Where $S(i)$ is TCP throughput at time i that is used by TFRC [24], and p is the packet loss rate. In Equation 3.1, the relationship between the packet loss rate and the estimation of the sending rate 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.

Also, as per [6], the new sending rate must be less than the receiver's rate estimation and larger than the TFRC, as shown in the following:

$$TFRC < As(i) < Ar(i)$$

3.2.2 Receiver-side controller

While the sender's main purpose is monitoring packet loss and computing the sending rate, the receiver's purpose is monitoring the video stream and the changes in frame delay. According to [11], the receiver side estimates the receiving rate given in Equation 3.2 based on the state of the overuse detector. The overuse detector monitors the changes in the frame delay and the state of the bottleneck to produce one of three signals: Overuse, Underuse, or Normal, according to Algorithm 1 which is derived from the codes in [29] and shown on the next page. In addition, the receiver side computes the packet loss rate (p) and sends it with the receiving estimate (Ar) and RTT to the sender side via REMB messages and RTCP reports.

$$Ar(i) = \begin{cases} \eta Ar(i-1) & \text{Increase} \\ Ar(i-1) & \text{Hold} \\ \alpha R(i) & \text{Decrease} \end{cases} \qquad \textbf{(Equation 3.2)}$$

Here, 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. The α is a fixed value (normally chosen between 0.80 and 0.95) while the receiving rate estimate is constrained by the following condition [6]:

$$Ar(i) < 1.5 \times R(i)$$

Where R(i) is the incoming rate. We can interpret the stream flow between the receiver and the sender from Figure 6. The media stream flow is sent by the sender to the receiver with the RTP packets while the arrival-time filter computes the queuing delay variation, T. The remote control region sets the threshold according to Algorithm 2, also shown below. 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 [9], 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 overuse detector receives the queuing delay variation T from the arrival-time filter and the threshold from the remote control region to produce a signal according to Algorithm1. Based on the overuse detector signal, the remote rate controller estimates the receiving rate based on Equation 3.3 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. The sender side calculates the TFRC and the sending rate, according to Equation 3.1. The sender compares the sending rate with the TFRC throughput to ensure compliance with the constraint in Section 3.2.2. Finally, the sender sets its sending rate and sends the next stream.

```
1   //Overuse Detector
2   IF number of deltas < 2
3   Then 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
10  Then time of overusing = ts _deltas/2
11  Else time of overusing = time of overusing + ts _deltas
12  // increase Overusing counter by 1
13  Overuse counter = +1
14  /* check if the time of overusing counter reaches Overusing time threshold
15  and check      Overuse counter too */
16  IF time of overusing > Overusing time threshold AND Overuse counter >1
17  /* check whether if the current offset estimate larger than the previous one
18  or not */
19  IF current offset > previous offset
20  Then time of overusing = 0 //reset the counter
21  //reset the counter
22  Overuse counter = 0
23  Overuse detector state = Overuse
24  Else T < - γ
25  //reset the counter
26   time of overusing = 0
27   Overuse detector state = Underuse
28   //reset the counter
29  Else time of overusing = 0
30  //reset the counter
31  Overuse counter = 0
32  Overuse detector state = Normal
```

Algorithm 1: Overuse detector pseudo-code**.**

1 //Rate control Region

2 **IF** Control-Region = *MaxUnknown*

3 **Then** $\boldsymbol{\gamma = \gamma'}$

4 $/*$ γ is $threshold$, γ' is the default *threshold*s

5 i. e. chromium $threshold$ is $\frac{25}{60}$ ms $*/$

6 **IF** Control-Region = *AboveMax* //congestion occurs

7 **OR** Control-Region = *NearMax* //close to congestion

8 **Then** $\boldsymbol{\gamma = \frac{\gamma}{2}}$ //Halved the *threshold*

Algorithm 2: Rate control region algorithm.

3.3 Summery of WebRTC Google Congestion Control Algorithm

In this section, we presented the current rate adaption that is used in the WebRTC implementation. The algorithm consists of two main modules: the receiver side, which is delay-based, and the sender side, which is packet loss-based. The receiver side has an arrival-time filter that is used for considering the inter-arrival time of video frames. The receiver sends (REMB) messages to the sender to let the sender know the next maximum expected rate that can receive. The RTCP reports are also sent by the receiver with REMB messages. The RTCP reports have the packet loss and the RTT. Based on the feedback from the receiver, the sender calculates its sending rate. The sending rate is increased if the packet loss is less than 2%; however, the sending rate is decreased if the packet loss is larger than 10%. The new sending rate is limited by an upper bound (the receiving rate) and a lower bound (TFRC throughput) [21]. Also, we have to mention that there are many changes in the current implementation they do not add yet to the Internet draft.

Chapter 4

Proposed Design and Implementation

In this chapter, we present our proposed model with some details, and we present a proof-of-concept for it.Then, we explain our testbed environment and the test cases that we implemented. We present the network conditions and experiment with the GCC original model and our model on constrained and unconstrained networks. We chose the performance parameters that we used to evaluate our performance. The evaluation parameters are related to the network overview, such as the incoming rate, packet loss, and Round-Trip Time.

4.1 The Proposed Design

When the overuse detector generates an overuse signal, the receiver attempts to reduce the receiving rate estimate by multiplying the current incoming rate by a fixed value, α, that is within the interval [0.80, 0.95]. It is interesting to note that in the WebRTC's recent implementation, α is set to be 0.90 or 0.95.

In this scenario, the receiver has to decrease its rate by this fixed α factor, 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, (Chapter 3, Section 3.2.2). 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. Intuitively, this is not optimal; the rate decrease should not be fixed, but it should be dynamic and should depend 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 and packet loss rate. To do so, in this thesis, 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). To calculate the dynamic alpha, we propose Equation 4.1:

$$\mathbf{Dynamic_{Alpha}} = \sqrt{\mathbf{e}^{-\left(\frac{(\mathbf{T}-\boldsymbol{\gamma})}{\mathbf{T}}\right)}} + \mathbf{a} \qquad \textbf{(Equation 4.1)}$$

Where T is the queuing delay variation, the threshold is the threshold that is set by the remote rate region as shown by Algorithm 2 (Chapter 3, Section 3.2.2), and $\boldsymbol{a}$ is a scalar parameter calculated in Equation 4.2:

$$\boldsymbol{a} = \begin{cases} (\mathbf{0.99} - \sqrt{\mathbf{e}^{-\left(\frac{(\mathbf{T}-\boldsymbol{\gamma})}{\mathbf{T}}\right)}})/\mathbf{1.3} \\ \textbf{if Remote Region} = \textbf{AboveMax or MaxUnknown} \\ (\mathbf{1} - \sqrt{\mathbf{e}^{-\left(\frac{(\mathbf{T}-\boldsymbol{\gamma})}{\mathbf{T}}\right)}})/\mathbf{1.14} \\ \textbf{if Remote Region} = \textbf{NearMax} \end{cases} \quad \textbf{(Equation 4.2)}$$

To simplify the formula in Equation 4.2, we define b as given in Equation 4.3:

$$\boldsymbol{b} = \sqrt{\boldsymbol{e}^{-\left(\frac{(\boldsymbol{T}-\boldsymbol{\gamma})}{\boldsymbol{T}}\right)}} \quad \textbf{(Equation 4.3)}$$

The final formula for calculating the scalar parameter *a* is given by Equation 4.4:

$$\boldsymbol{a} = \begin{cases} (\mathbf{0.99} - \boldsymbol{b})/\mathbf{1.3} \\ \boldsymbol{if\ Remote\ Region} = \boldsymbol{AboveMax\ \ or\ MaxUnknown} \\ (\mathbf{1} - \boldsymbol{b})/\mathbf{1.14} \\ \boldsymbol{if\ Remote\ Region} = \boldsymbol{NearMax} \end{cases} \quad \textbf{(Equation 4.4)}$$

Let us explain the justification for our design. The dynamic alpha allows the receiver to decrease its receiving rate based on the magnitude of the 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 0.90 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 media (video, audio, etc.). However, 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.

We also designed two different values of *a* as shown in Equation 4.2 to be more adaptable to the two default values of the threshold: 25/60 and (25/60) /2. Designing *a* is based on the value of the threshold, which is determined by the remote rate region in Equation 4.2, and the final formula for *a* is shown in Equation 4.4 based on the value of the threshold. Also, in Equation 4.2, we use two equations to calculate *a* by using two different values (1.30 and 1.14) to adapt to a threshold value, which is determined by remote rate region. Based on network status, remote rate region sets two

values for threshold:1) the default threshold (γ), and 2) the default threshold by two ($\gamma/2$). Therefore, our scaler parameter *a* changes to adapt these two threshold values. To verify our model theoretically, 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. The expected dynamic alpha values are shown in Table 2.

Default 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

Table 2: Different assumed T values and the expected dynamic alpha.

We describe the details for the testbed that we used for running the WebRTC test cases. We implemented our proposed rate adaption model in WebRTC reference code and building WebRTC in two hosts. Then, we observed various network parameters to analyze the performance of our model. Measurements plotting were carried out by using Matlab.

4.2 Building WebRTC for Windows

To build the WebRTC, we followed the building instructions for Windows x64 that are provided in [29]. First, we installed the prerequisite software, such as Depot Tools, Visual Studio 2013 Community, and Windows 10 SDK. Then, we compiled the code as our baseline. We avoided updating the code continuously due to the bugs that appeared in each version since we successfully built our testbed.

4.3 Implementation of our proposed changes

To implement our algorithm, we modified some parts of the WebRTC reference code. Our changes and modifications were made in different files, which are described in the next lines.

We defined a new object called *dynamic alpha*, and we gave it the initial value = 0.90 in the ../src/webrtc/modules/remote_bitrate_estimator/*aimd_rate_control.cc* file. To make the object visible, we also added it in the header file *aimd_rate_control.h.*

```
dynamic_alpha(0.9f)
```

Since our algorithm uses some values from the overuse detector, such as threshold and delay, we made those objects global to allow for using them by the *aimd_rate_control.cc* file in our algorithm's calculation. We created a new header file called *global.h* to define the global objects, which are the saved values of γ and T.

```
extern double global_threshold;

extern double global_T;
```

We added the following lines in *overuse_detector.cc*:

```
global_T = T;

global_threshold_ = threshold;
```

Going back to the *aimd_rate_control.cc* file, we replaced some code lines to compute the receiving rate. The code lines that were replaced are as follows:

```
current_bitrate_bps = static_cast<uint32_t>(beta_  incoming_bitrate_bps + 0.5);
```

is replaced by

current_bitrate_bps = static_cast<uint32_t>(dynamic_alpha incoming_bitrate_bps + 0.5);

Also, we replaced

*current_bitrate_bps=static_cast<uint32_t>(beta_ avg_max_bitrate_kbps_ * 1000 + 0.5f);*

with

*current_bitrate_bps = static_cast<uint32_t>(dynamic_alpha avg_max_bitrate_kbps_ * 1000 + 0.5f);*

In the same file, we wrote the code lines according to Equation 4.1 and replaced them with the original code as follows:

beta_ = 0.9f;

is replaced by

dynamic_alpha=sqrt(exp(-(global_T_global_threshold_)/global_T_)) + (0.99 - (sqrt(exp(-(global_T_ - global_threshold_)/global_T_))))/1.3;

Also, we replaced

beta_ = 0.95f;

with

dynamic_alpha =sqrt(exp(-(global_T_ - global_threshold_)/global_T_)) + (1 - (sqrt(exp(-(global_T_ - global_threshold_)/global_T_))))/1.14;

4.4 Network Setup

The experimental testbed consists of two laptops having similar specifications. Each side acts as an independent client. The DELL laptop (Intel Core i7-000 M, 8 GB RAM, Windows 7 64-bit) was designated as the sender, and we called it Client-1; the Lenovo ThinkPad W540 laptop (Intel Core i7-7400MQ - 8 GB RAM, Windows 7 64-bit) was designated as the receiver, and we called it Client-2. We set up an experimental testbed over a real network. The WebRTC native code is implemented on both sides [29]. Two clients ran the WebRTC Version March 2015, and an Ethernet cable connected them. To emulate a wired network, we used a software-based network

emulator, NEWT [22]. NEWT is used to emulate a variety of network attributes, such as available bandwidth, packet loss, and propagation delay.

4.5 Video Call Setup

Multiple video streaming sessions were completed between both sides, the sender and the receiver, and vice versa. We used a virtual video camera tool [27] to inject the video sequence and to ensure that the video stream is consistent. Virtual camera tool emulates a webcam in our system and works like a real camera. By using virtual camera tools, we can set our video clip sequence. Also, to support the experiments' reproducibility, we used the publicly available Wave Hand Video Clip [7]. To generate the video flows, Host A ran the peer-connection client JavaScript API as a client and peer-connection server JavaScript API as a server, while Host B ran the peer-connection client JavaScript API as a client.

4.6 Test Cases

The NEWT [22] tool that was used in this study can simulate different scenarios over wired access networks. We designed the test cases to be as clear and as simple as possible. We have four scenarios are under constrained network in addition to one scenario under unconstrained network, which was without any constraints. We repeated each scenario twice - the first time with the original GCC and second time with our proposed model. We observed the overuse detector states to make sure that congestion occurred during the session. Since our proposed method works when congestion appears, we eliminated each session that did not contain an overuse signal (see Figure 7). We took the same action for the session with only Normal signal. However, we accepted the green session, which showed congestion.

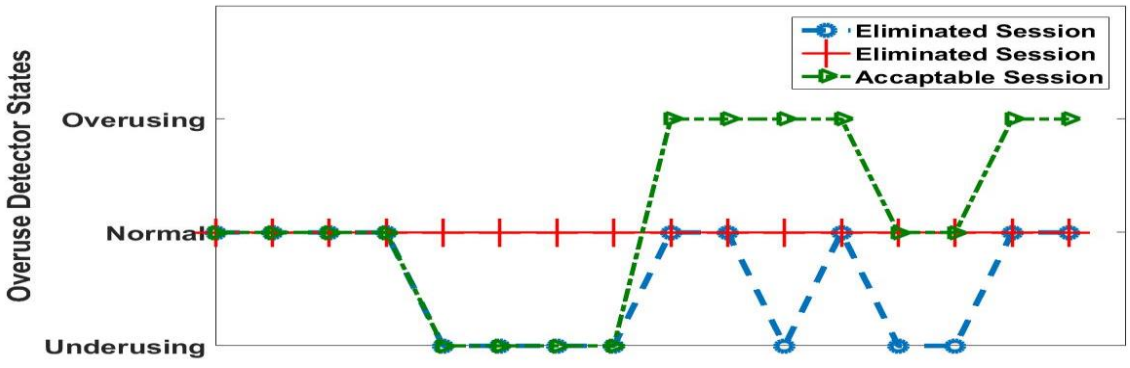

Figure 7: Overuse status for three different sessions.

For each scenario, we collected the performance metrics that we will describe in the next section. In each scenario, we compared the WebRTC performance metrics for the proposed model with the performance metrics for the existing GCC model. The following sub-sections are the test cases that were considered and implemented in this thesis.

4.6.1 Unconstrained Network

In Test case 1, we do not set any network attributes to the network emulator. The purpose of this test is to analyze the WebRTC behavior in an uncontrolled environment with no constraints, such as limited bandwidth. Based on this test case, we chose which network attributes would work in the next phase. We called this case the reference case. The Ethernet cable speed is approximately 75 Mbps. We observed the WebRTC behavior for GCC and the proposed model by repeating the scenario for each model and collecting the performance metrics. Figure 8 shows the testbed for this case.

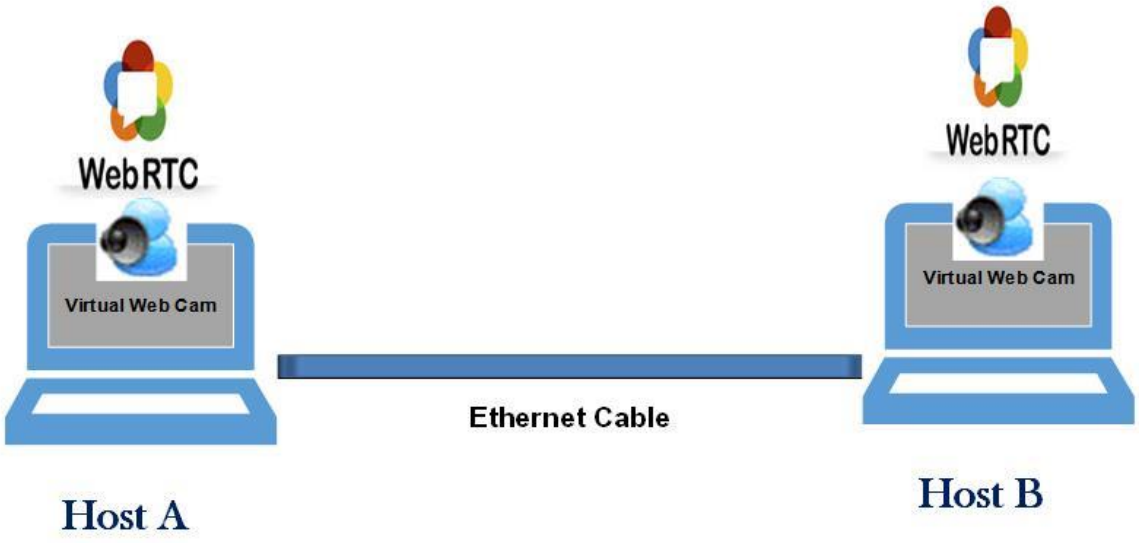

Figure 8: Testbed Architecture (unconstrained network).

4.6.2 Constrained Network

In the next test cases, we set various network conditions to investigate how WebRTC with GCC and with the proposed model adapts to varying the available bandwidth, packet loss, and propagation delay. Figure 9 illustrates the testbed for these test cases. Table 3 shows the test cases and the network attributes.

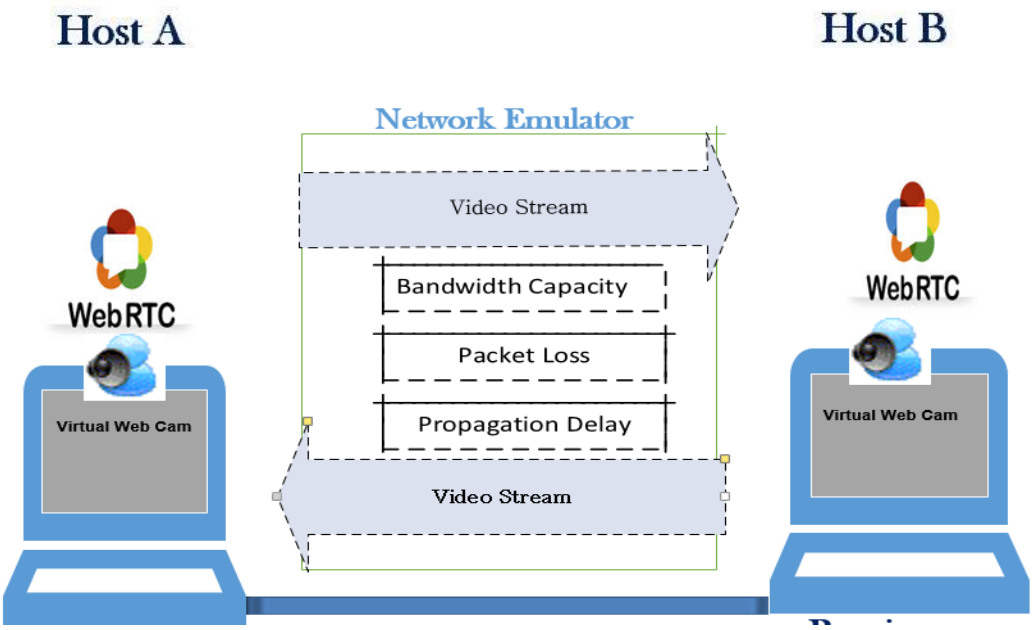

Figure 9: Testbed Architecture (constrained network).

	Test Cases Number	The Available Bandwidth	The Packet Loss Rate	The Prorogation Delay
Unconstrained Network	Test Case 1	No Constraints	No Constraints	No Constraints
Constrained Network	Test Case 2	50 kbps 200 kbps 500 kbps 1000 kbps	Null	Null
	Test Case 3	Null	2% 5% 10% 12%	Null
	Test Case 4	Null	Null	50 ms 250 ms 500 ms 1000 ms
	Test Case 5	500 kbps 1000 kbps	5% 10%	50 ms 250 ms

Table 3: Test cases with network attributes.

Varying available bandwidth. First, we investigated how the WebRTC responded to varying available bandwidth in the network. Since NEWT was used in this study to emulate different network attributes, we used it to vary the bandwidth capacity from 50 kbps to 1000 kbps with no random packet loss rate and no propagation delay.

Varying packet loss. In this test case, we varied the packet loss rate from 2% to 12% with no two-way propagation delay. The purpose of this test was investigating the WebRTC response to the packet loss.

Varying propagation delay. In this test, we measured the WebRTC's performance metrics by varying the two-way delay propagation delay from 50 ms to 1000 ms in the NEWT emulator.

Varying available bandwidth – packet loss -propagation delay. In this test, we merged different network conditions and measured the WebRTC's performance metrics by varying available bandwidth, packet loss rate, and two-way delay; we set 500 kbps with 5% packet loss rate and 50 ms propagation delay. Also, we set 1000 kbps with 10% packet loss rate and 250 ms two-way delay. As previous test cases, we used the NEWT emulator to set these conditions.

4.7 WebRTC Performance Metrics

In this section, we chose the evaluation metrics that usually trigger the congestion control mechanisms in WebRTC. Therefore, to evaluate our system, we chose three performance metrics:

1. Incoming Rate: the amount of data (kbps) that the receiver accepts during the video session.

2. Round-Trip Time (RTT): the time it takes to send the signal and the time it takes to receive the acknowledgment in ms.

3. Packet Loss Fraction: the number of lost packets during the video session.

Chapter 5

Results and Analysis

This chapter presents the experimental results of the rate adaptation model of the WebRTC and our proposed model. Our experiments were implemented over a constrained and an unconstrained network. We show the performance enhancement regarding the incoming rate, and RTT. We present and discuss the results of the proposed model in our thesis. First, we implemented our model over an unconstrained network. Then, we used a network emulator [22], which is capable of performing test cases over various wired access networks. The network emulator allows us to implement our model over constrained network.

5.1 Case 1: Unconstrained Network

In this test case, we did not add any network conditions to our experiment to observe WebRTC behavior under a real network. In this test case, we ran 12 video sessions with no constraints. Duration of each video session was two minutes. For each session, we computed the average of the evaluation metrics, which were directly gotten from the WebRTC source code (see Section 4.4).

Figure 10 shows the incoming rate that is experienced by the receiver side. The green rectangles illustrate the average incoming rate for all sessions using our proposed dynamic alpha while the red triangles illustrate the rate for sessions using the fixed alpha. The superiority of our proposed approach is clear; we can see 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.

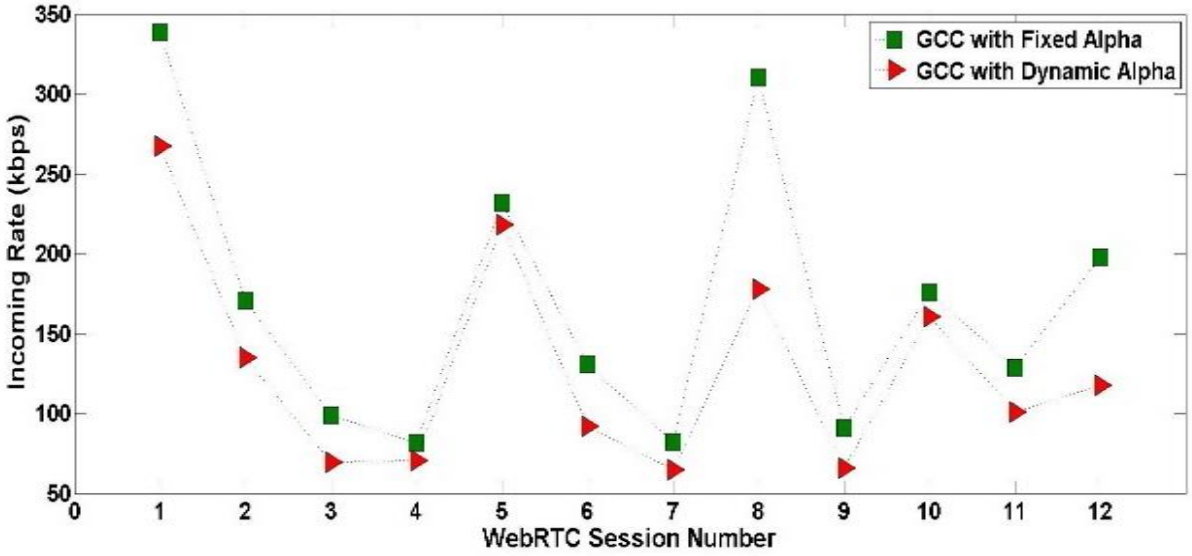

Figure 10: Average incoming rate (kbps) for 12 sessions.

The performance of RTT, and packet loss fraction are shown in Figure 11, and Figure 12, respectively. We can see that number of the packet loss fraction is about the same, occasionally a few more, compared to the fixed alpha method. The numbers do not show a significant difference in terms of the number of packets lost, as can be seen in Figure 12. However, our method achieves a lower RTT, by 16% on average, as shown in Figure 11.

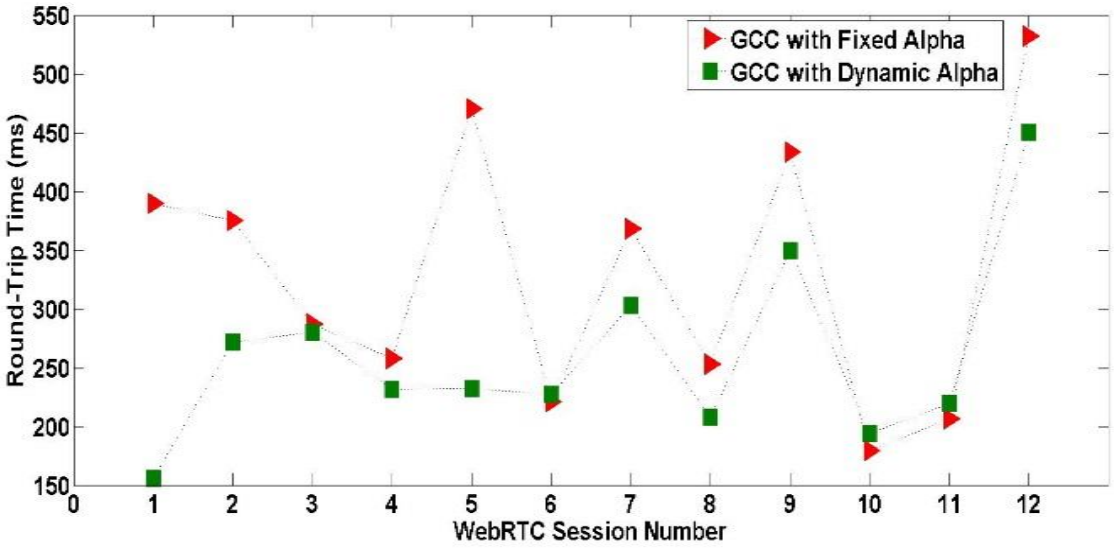

Figure 11: Average of Round-Trip Time (ms) for 12 sessions.

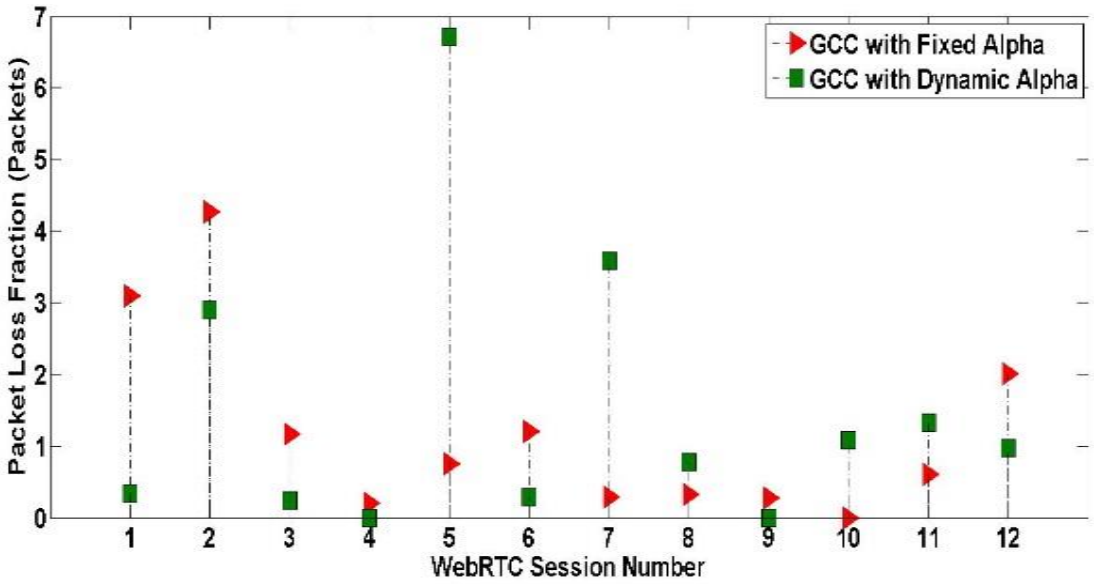

Figure 12: Average of Packet Loss Fraction (packets) for 12 sessions.

The WebRTC's original model results and the results of the proposed model for each session are shown in Table 4.

Session Number	Incoming Rate (Kbps)		Packet Loss Fraction (Packets)		Round-Trip Time (ms)	
	WebRTC Original	**Proposed Model**	**WebRTC Original**	**Proposed Model**	**WebRTC Original**	**Proposed Model**
1	267.43	338.99	0.34	3.1	389.92	156.047
2	135.08	170.53	2.9	4.27	375.28	271.64
3	69.67	99.15	0.24	1.17	287.47	280.5
4	70.45	81.72	0	0.2	257.67	231.4
5	218.2	231.75	6.71	0.75	470.67	232.22
6	92.09	130.92	0.29	1.2	221.4	227.76
7	65.02	82.11	3.58	0.29	368.7	303.18
8	178.22	310.64	0.78	0.32	252.93	208.33
9	65.78	91.09	0	0.28	433.99	349.5
10	160.99	176.17	1.08	0	179.72	194.35
11	100.96	128.72	1.32	0.6	206.65	219.91
12	117.572	197.84	0.97	2.01	532.199	450.01

Table 4: WebRTC results for the original model and the proposed model.

5.2 Constrained Network

The scenarios that are shown in Table 3 aim at investigating how WebRTC congestion control algorithm with fixed and dynamic alphas react to sudden changes of available bandwidth, packet loss rate, and network delays. For each of those scenarios, we have considered three WebRTC video sessions. The time duration of each video call was considered to be 30 seconds. The video session is established when both users (Host A and Host B) were connected together. After 30 seconds, the video session was terminated by the users. We set the same video sequence over the virtual came. We computed the average of the performance metrics for the three sessions. We observed the results of how the original WebRTC receiver's rate model and our proposed model performed based on both sides: Receiver and Sender.

To investigate the impact of the bandwidth capacity, the packet loss rate, and the delay on the GCC with fixed and dynamic alphas, we set different network constraints.

5.2.1 Case 2: WebRTC with varying bandwidth constraints

In this test case, we varied bandwidth capacity to investigate the impact of network bandwidth availability in the GCC with the fixed alpha and the dynamic alpha. We varied the bandwidth capacity on the network emulator from 50 kbps to 1000 kbps, with no packet loss rate and delay constraints. Figure 13 shows the incoming rate that is experienced by the receiver side. As with the previous graphs, the green rectangles illustrate the average incoming rate for the three video sessions that used our proposed dynamic alpha, while the red triangles illustrate the rate for the three sessions that used the GCC with the fixed alpha. We observed that the dynamic alpha gives a higher incoming rate compared to what we experienced with the fixed alpha.

Table 5 shows each bandwidth constraint and the percentage of increase in the incoming rate for the GCC with fixed and dynamic alphas.

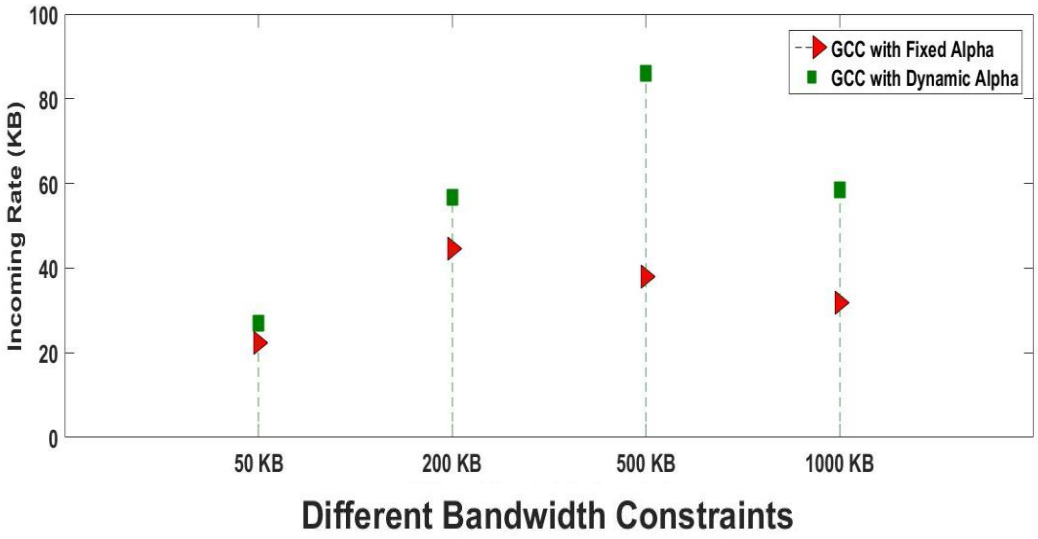

Figure 13: Incoming rate increase amount (kbps) under different bandwidth constraints.

Bandwidth Constraints	Incoming Rate Increasing Amount (%)
50 KB	+20.60 %
200 KB	+26.92 %
500 KB	+125.91 %
1000 KB	+83.39 %

Table 5: Incoming rate increase amount by percentage.

The incoming rate increase is between 20% to 125%, which is considered a good improvement. Furthermore, our proposed dynamic alpha decreased RTT (see Table 6, and Figure 14).

Moreover, the dynamic alpha slightly decreases the packet loss fraction by 29.4% when the bandwidth capacity was 50 KB. However, with 200, 500, and 1000 KB, the packet loss fraction tended to increase. Figure 15 illustrates acceptable decrease in packet loss fraction under 50 KB bandwidth capacity, yet slight increases under other bandwidth constraints.

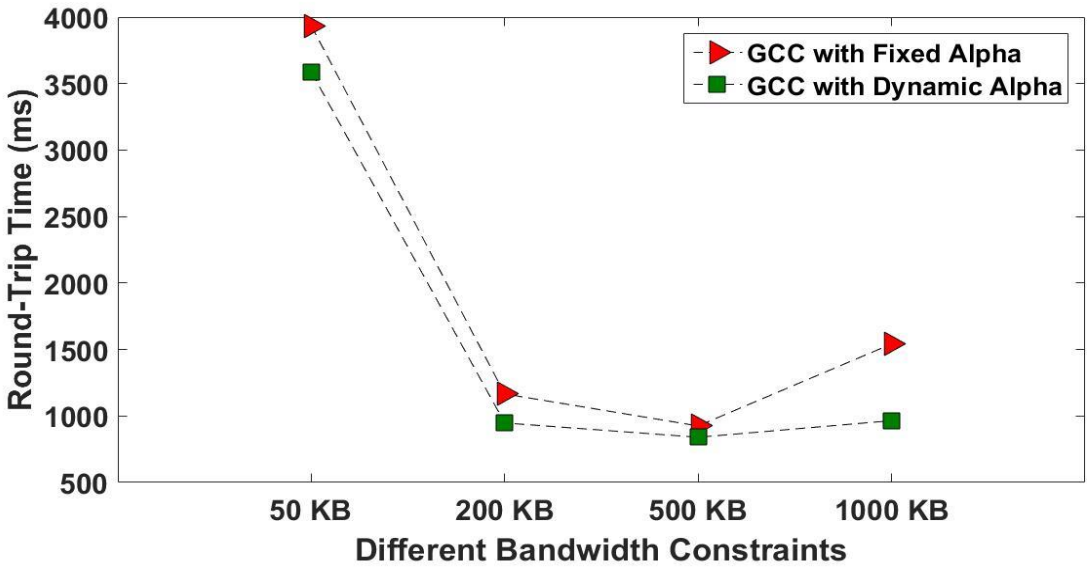

Figure 14: RTT decrease amount by percentage under different bandwidth constraints.

Bandwidth Constraints	RTT Decreasing Amount (%)
50 KB	-8.82%
200 KB	-18.72%
500 KB	-9.22%
1000 KB	-37.43%

Table 6: RTT decrease by percentage.

We can see that our proposed method loses about the same number of packets, occasionally a few more, compared to fixed alpha method. The numbers do not show a significant difference in terms of the number of packets lost, as can be seen in Figure 15. However, our method achieves a higher incoming rate, by 125%, and a lower RTT, by 37%, on average, as previously shown in Figure 13: Incoming rate increase amount (kbps) under different bandwidth constraints.

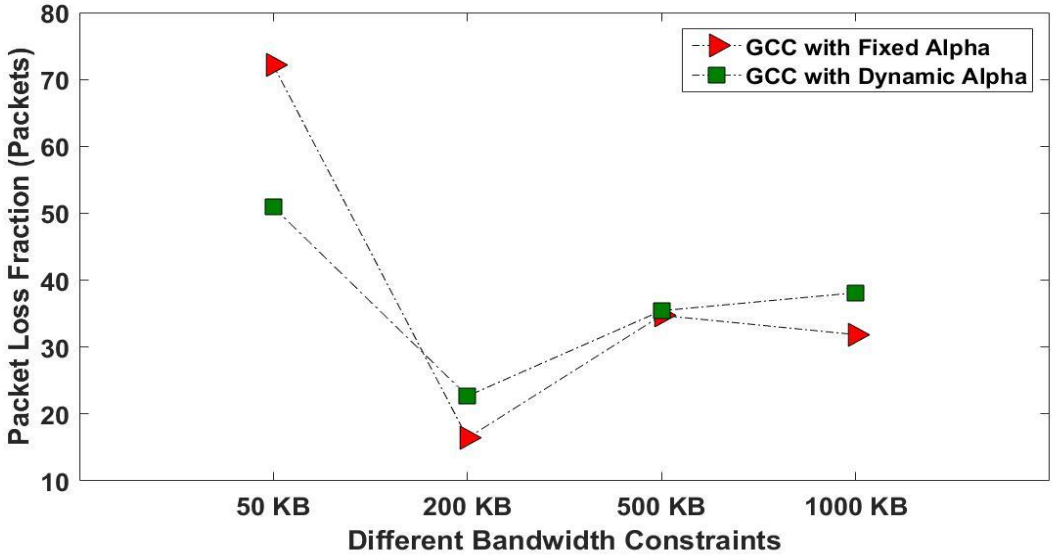

Figure 15: Packet Loss Fraction under different bandwidth constraints.

Bandwidth Constraints	Packet Loss Fraction Amount (%)
50 KB	-29.4%
200 KB	+38.29%
500 KB	+2.01%
1000 KB	+19.61%

Table 7: Packet Loss Fraction under different Bandwidth Capacity.

From the results of this experiment, WebRTC with a dynamic alpha is able to improve performance regardless of the network bandwidth capacity. Also, it is worth noticing that WebRTC shows low performance when the bandwidth capacity is equal to or under 200 KB.

5.2.2 Case 3: WebRTC with varying packet loss rate constraints

In this test case, we introduced random packet losses in the network. The packet loss rate varied from 2% to 12%. The measurement results are illustrated in Figure 16 to Figure 18. Figure 16 shows the impact of packet losses in the network on the GCC with fixed and dynamic alphas. The red and green bars present the average incoming rates for the GCC with a fixed and a dynamic

alpha, respectively. Based on the experimental testbed, Figure 16 shows the huge difference between the incoming rates with fixed and dynamic alphas. The dynamic alpha is able to give a much higher incoming rate compared to the fixed alpha.

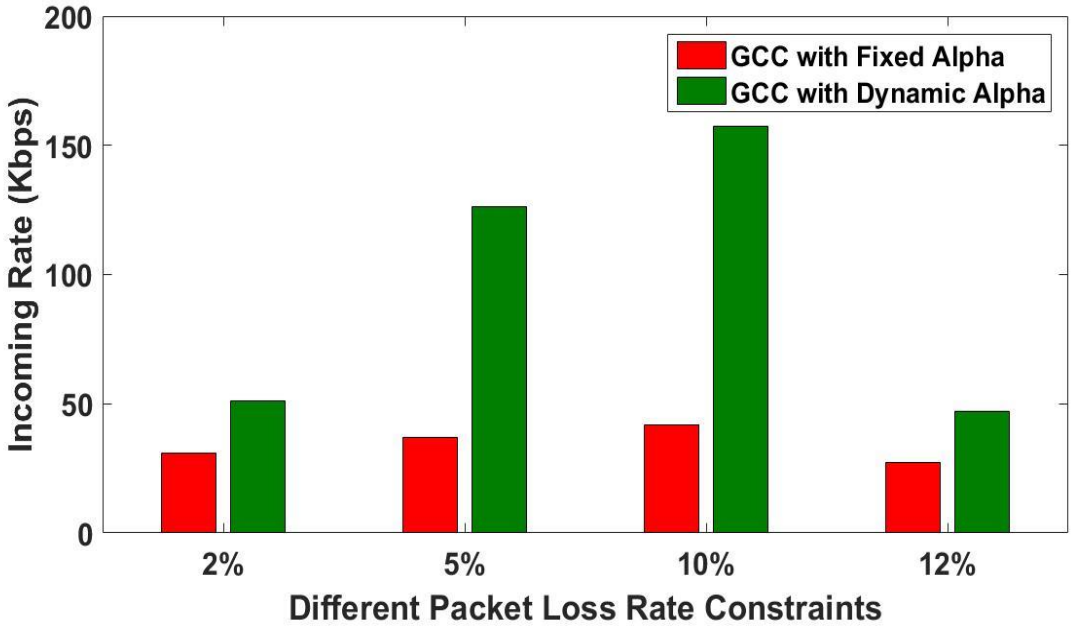

Figure 16: Incoming rate (kbps) under different packet loss rate constraints.

Table 8 shows each bandwidth constraint and the amount of increase in the incoming rate. It can be viewed that the proposed dynamic alpha increases the incoming rate up to 274%.

Packet Loss Rate Constraints	**The Incoming Rate Increasing Amount (%)**
2%	+ 66.21%
5%	+ 242.26%
10%	+ 274.94%
12%	+ 72.86%

Table 8: The incoming rate increase percentage.

Another enhancement that can be provided by the proposed dynamic alpha is the decrease in the packet loss fraction when the network faces different packet loss rates. The proposed dynamic alpha decreases the packet loss fraction under 2%, 5%, 10%, and 12% packet loss rates (see Figure 17). The packet loss fraction decrease is around 60% (see Table 9).

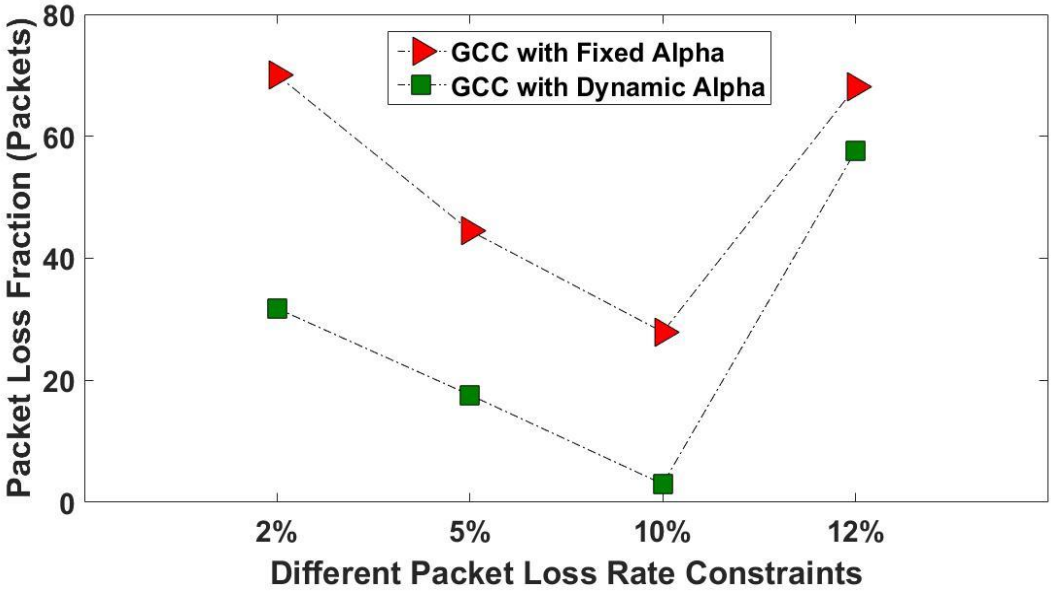

Figure 17: Packet Loss Fraction (number of packets) under different packet loss rate constraints

Packet Loss Rate Constraints	The Losses Fraction Decreasing Amount (%)
2%	-54.63%
5%	-60.48%
10%	-89.28%
12%	-15.37%

Table 9: The percentage of Packet Loss Fraction decrease.

Also, the GCC with the dynamic alpha is still able to reduce RTT, in general. Figure 18 shows the decrease in RTT under various packet loss rates: 2%, 5%, 10%, and 12%.

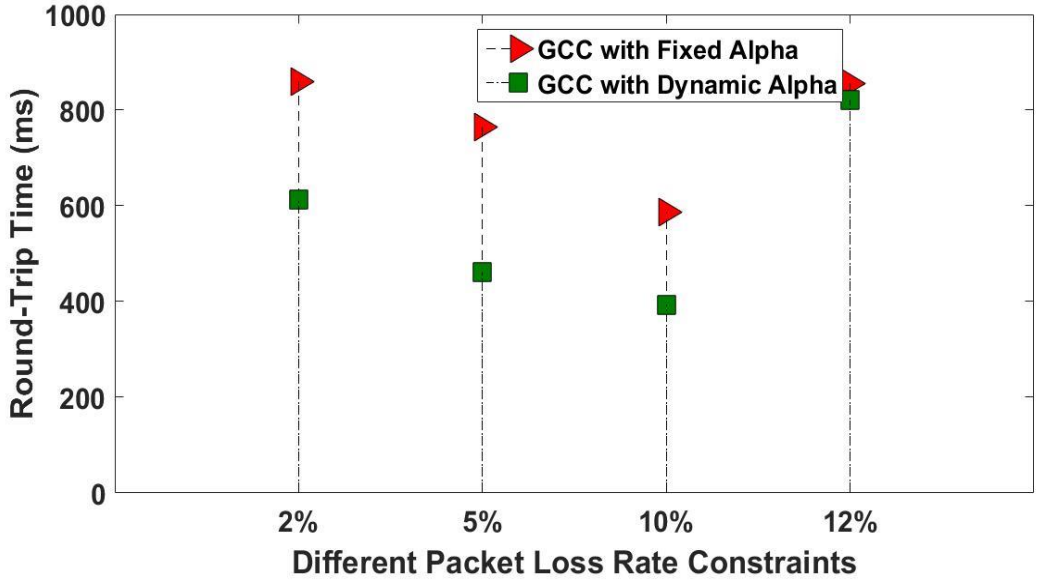

Figure 18: Round-Trip Time (ms) under different packet loss rate constraints.

Table 10 clearly presents the percentage of decrease in RTT. Generally, RTT decreased between 3% and 39%.

Packet Loss Rate Constraints	The RTT Decreasing Amount (%)
2%	-28.53%
5%	-39.80%
10%	-32.78%
12%	-3.86%

Table 10: The percentage of RTT decrease.

Our results indicate that a dynamic alpha can give high performance compared to fixed alpha under various packet losses. The proposed dynamic alpha increases the incoming rate while decreasing the packet loss fraction and Round-Trip Time.

5.2.3 Case 4: WebRTC with two-way propagation delay constraints

This scenario is aimed at investigating how the WebRTC's receiving rate reacts to different latencies. We varied the propagation delays and observed the impact on the performance of

WebRTC. We introduced two-way delays of 50, 250, 500, and 1000 ms before starting to report our results. It is worth noting that WebRTC under 500 and 1000 ms delays cannot be run for more than 23 seconds. Therefore, the results that we plotted here for delays of 500 ms and 1000 ms are for sessions 23 seconds long. Figure 19 illustrates the average incoming rates for the three sessions under different delay constraints. As shown in Figure 19, the proposed dynamic alpha increases the incoming rate in general. The increased amount is approximately 4% to 23% (see Table 11).

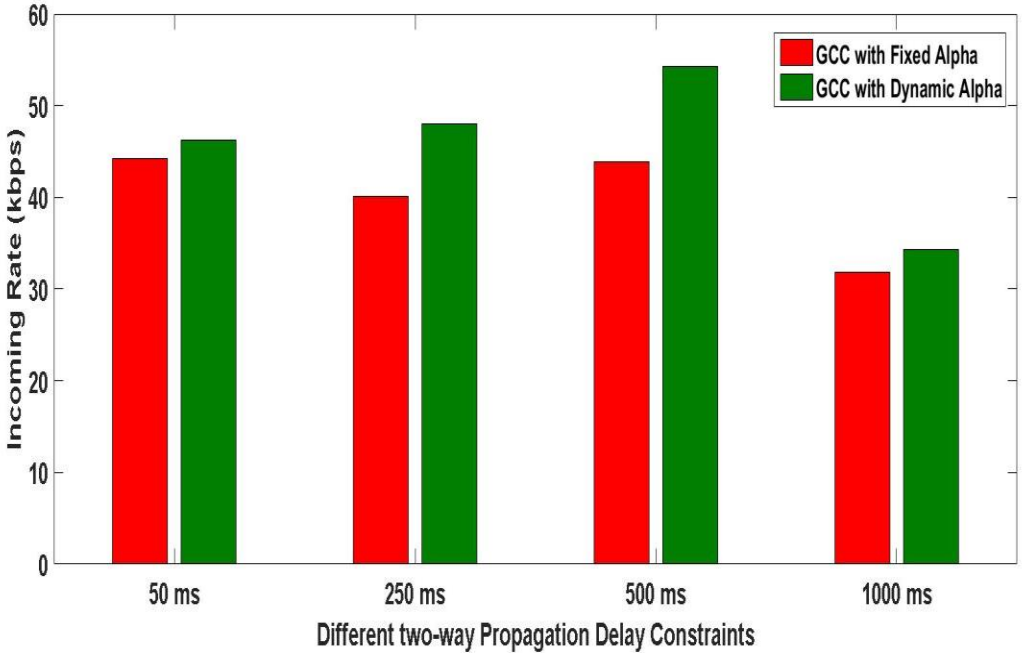

Figure 19: Incoming rate (kbps) under different two-way delay constraints.

Delay Constraints (ms)	The Incoming Rate Increasing Amount (%)
50	+4.42%
250	+19.68%
500	+ 23.93%
1000	+ 8.09%

Table 11: The percentage of the incoming rate increase**.**

The packet loss fraction clearly reduces with the proposed dynamic alpha compared to the fixed alpha (see Figure 20). The decrease is approximately 13% to 29% (see Table 12). Table 11 summarizes the packet loss fraction decrease under the current scenario.

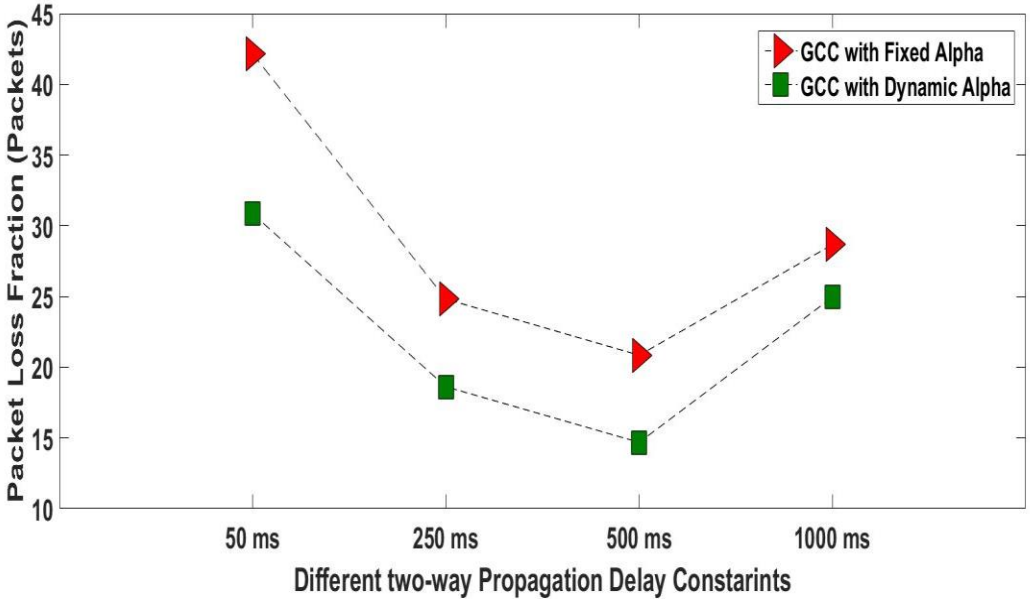

Figure 20: Packet loss fraction (packets) under different two-way delay constraints

Delay Constraints (ms)	Packet Loss Fraction Decreasing Amount (%)
50	-26.89%
250	-24.96%
500	-29.52%
1000	-12.95%

Table 12: The percentage of Packet Loss Fraction decrease.

As shown in Figure 21, our proposed dynamic alpha can decrease RTT in all cases. It reduces RTT by 75% over the network with a 50 ms delay and approximately 26% with a 250 ms delay and a 500 ms delay. The dynamic alpha cannot give the same amount of reduction when the delay is 1000 ms; however, it is still able to reduce RTT by 1% (see Table 13).

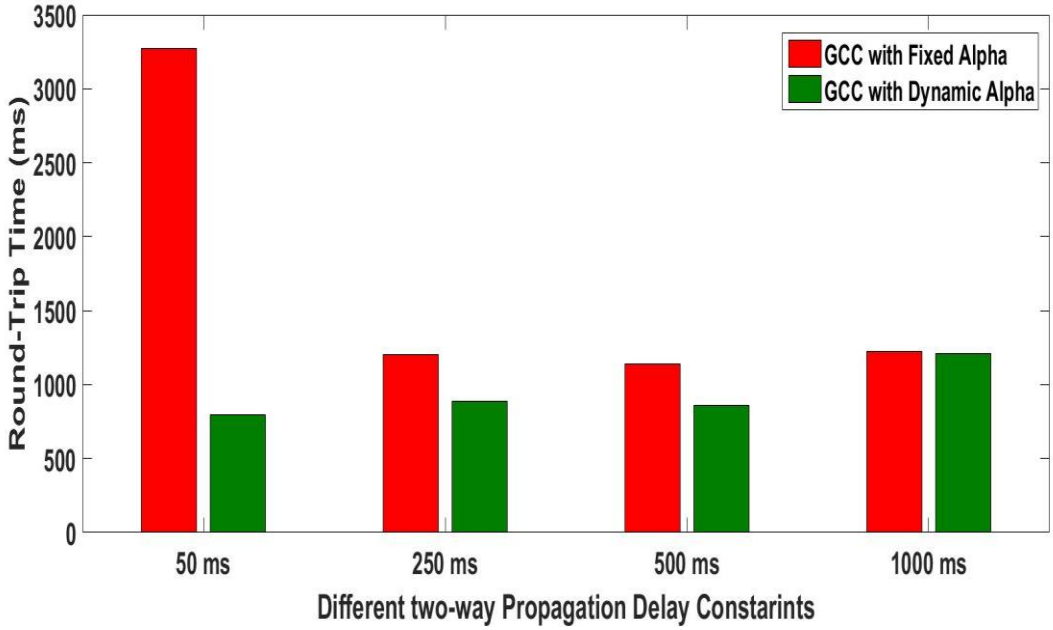

Figure 21: Round-trip time decrease amount (ms) under different two-way delay constraints.

Delay Constraints (ms)	RTT Decreasing Amount (%)
50	-75.68%
250	-26.21%
500	-25.00%
1000	-1.05%

Table 13: The percentage of RTT decrease.

From the current scenario, we found that the proposed dynamic alpha improves WebRTC performance under various delays. However, WebRTC shows low performance and cannot exceed 30 seconds long when the network delay is more than 250 ms.

5.2.4 Case 5: WebRTC with varying Bandwidth, Packet Loss Rate, and Delay constraints

In this test case, we present more extensive measurement 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: Test Case 5 (a) and Test Case 5 (b). In Test Case 5 (a), the bandwidth capacity is set to 500 kbps, the packet loss rate is 5%, and the propagation delay is 50 ms. In Test Case 5(b), we increase the bandwidth capacity to be 1000 kbps, the packet loss rate is 10%, and the propagation delay is 250 ms.

Figure 22 to Figure 24 show the measurement results of Test Case 5. The proposed dynamic alpha is keeping improve the incoming rate. The improvement of the incoming rate by percentage is shown in Table 14. The incoming rate is increased by 19% in both sub-test cases 5 (a) and 5 (b).

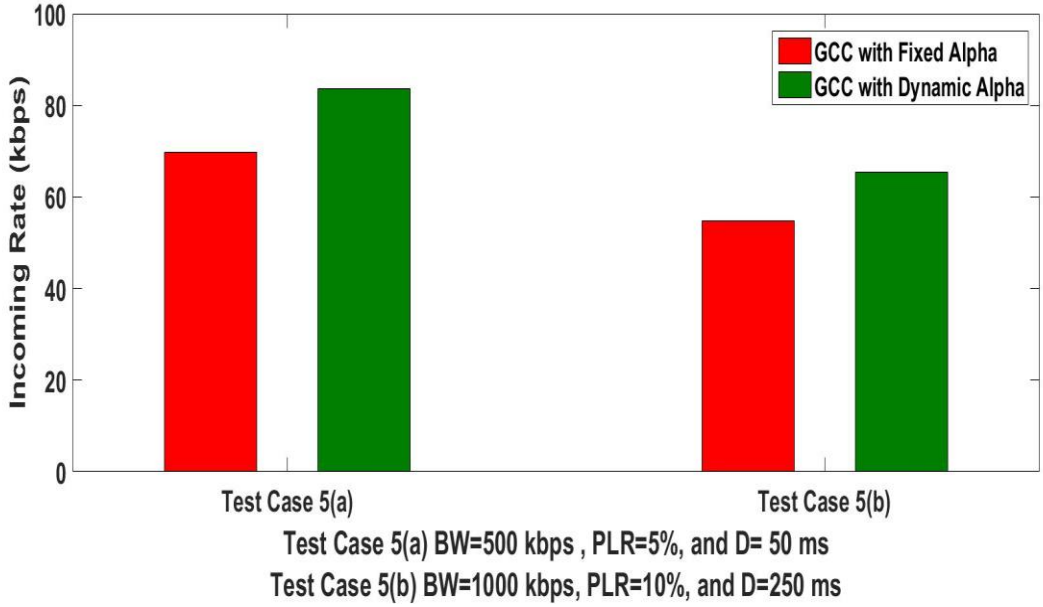

Figure 22: The incoming rate under different Network constraints.

Test Case	The Incoming Rate Increasing Amount (%)
Test Case 5 (a)	+ 19.71%
Test Case 5 (b)	+ 19.42%

Table 14: The percentage of incoming rate increase for Case 5.

Moving to packet loss fraction, when the bandwidth capacity is 500 kbps with 5% packet loss rate and two-way delay is 50 ms, the packet loss fraction is increased by implementing our model

(Test Case 5 (a)). However, when we increase the bandwidth capacity to 1000 kbps, the packet loss fraction is decreased. The packet loss fraction in Test Case 5 (b) can decrease even with increasing the packet loss rate to 10% and the delay to 250 ms. So, the decreasing of packet loss fraction in Test Case 5 (b) is due to increasing the bandwidth capacity.

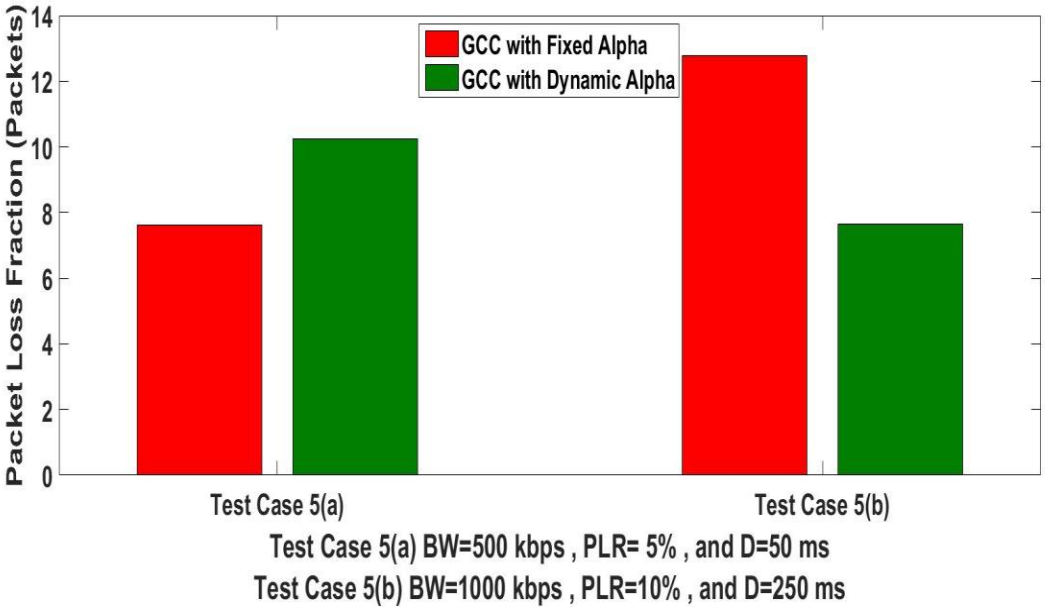

Figure 23: Packet loss fraction under different Network constraints.

Test Case	**Packet Loss Fraction Amount (%)**
Test Case 5 (a)	+ 34.69%
Test Case 5 (b)	-40.17%

Table 15: The percentage of packet loss fraction difference for Test Case 5.

Figure 23 presents the increasing and the decreasing in packet loss fraction in both Test Cases 5 (a) and 5 (b), respectively, and Table 15 shows the percentage of the increase and the decreased percent of packet loss fraction.

Test case 5 (a) also shows slightly increase in RTT by 2%. However, Test Case 5 (b) illustrates significant decreasing in RTT by 30% (see Figure 24, and Table 16). We assume the same reason behind decreasing RTT in Test Case 5 (b) which is the increasing on the bandwidth capacity.

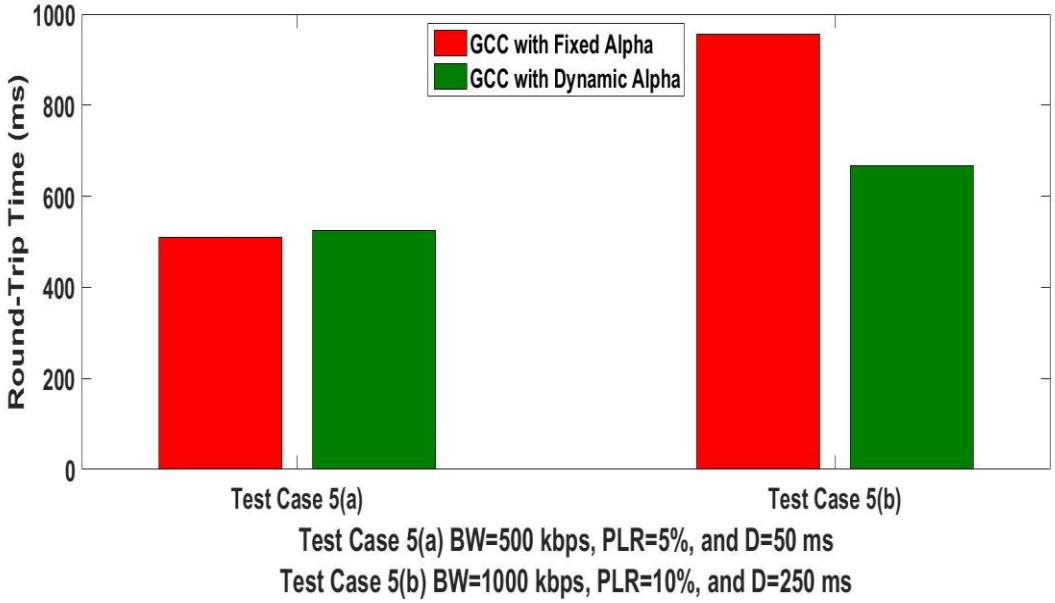

Figure 24: RTT over different Network constraints.

Test Case	RTT Amount (%)
Test Case 5 (a)	+ 2.93%
Test Case 5 (b)	-30.26%

Table 16: The percentage of RTT difference for Test Case 5.

From the current scenario, we found that the proposed dynamic alpha improves WebRTC performance under various Network constraints in terms of incoming rate, packet loss fraction, and RTT in Test Case 5 (b). However, WebRTC only shows improvement in the incoming rate in Test Case 5 (a). We see a slight increase in RTT around 2%, this is due to the time that we run our experiment. We run this experiment on Saturday afternoon that is a weekend day, and as we know this is the time that the network usually becomes busy.

According to all experimental results, we notice that incoming rate, RTT, and packet loss fraction are showing higher or lower based on the experiment time. For example, if our experiment runs in rush hours such as Saturday afternoon, the incoming rate shows little increase with a little RTT decrease. However, on Monday at five a.m., our experiments show high incoming rate reaches to 200% increase and higher RTT decrease reaches to 75%.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize what we covered in the previous chapters. We conclude our results and contributions. Furthermore, we highlighted some discussions and suggested some future work related to our studies.

6.1 Conclusion

In this thesis, we proposed a rate adaptation model for WebRTC receiver controller to improve real-time interactive video during overusing. Our model is built based on the Google Congestion Control algorithm current implementation

The baseline simulation results for the WebRTC show that a fixed factor (fixed alpha) is not the best way to calculate the receiving rate decrease. Therefore, we have to take advantage of the *overuse detector* information, including the difference between the delay variation and the threshold. The reason for proposing such a dynamic alpha model is to determine accurate estimates of the receiving rate for the WebRTC receiver side during overuse, which reflects positively on the WebRTC performance.

In order to observe the performance of the rate adaption model, we carried out an experiment using a real network and obtained extensive performance measurements. We implemented our model under an unconstrained and a constrained network. Our studies compare two major rate adaption models: the Google rate adaption model and our proposed model. The first adaption model was the Google Congestion Receiver Controller model (only under Overusing), which let the receiver controller estimate its receiving rate by decreasing it by a fixed amount when congestion occurs. The second adaption model was our proposed model that we showed in Chapter 3, which used the difference between the threshold and the delay variation to decrease the receiving rate by a dynamic factor (dynamic alpha).

Based on the experimental results, we showed that our model improves the performance regarding the incoming rate and RTT. The dynamic alpha can increase the incoming rate and decrease RTT. In most cases, our model was able to decrease the number of packet loss fraction as well. However, in some cases, our model displayed a slight increase in the packet loss fraction. Mostly, the proposed model shows good performance when compared to the Google model. Based on our

observation the increasing and the decreasing in the packet loss fraction due to the network status whether the network is busy or not.

6.2 Discussions and Future Work

First of all, we have to note that our model was designed based on the Chrome threshold, which is 25/60 ms. Therefore, our model may require modifications for compatibility with different browser thresholds. Also, we carried out our experiments in a simulated environment. If we could apply our work on the Internet, it would be more realistic and stronger.

There are also numerous ongoing discussions in the IETF groups in order to find out how to improve the performance of WebRTC sessions. Therefore, there are several suggested alternative rate adaption models and protocols. There are other congestion control algorithms might enhance WebRTC performance, such as Low Extra Delay Background Transport (LEDBT) and Datagram Congestion Control Protocol (DCCP).

Another discussion is whether we can add a different rate adaption model based on the network conditions. Also, another discussion is about what can be used as a better alternative of RTCP as feedback mechanisms.

Further researches could be done in order to determine the usability limits of WebRTC and how to reduce theses limits. Also, future work can include more experiments on what we can add to enhance WebRTC performance on the mobile environment.

Some studies discuss WebRTC framework and what we can do to improve WebRTC framework. Some suggested parameters to add them to WebRTC framework such as adding more overuse detector states.

In future studies, this work may be extended to undertake a further evaluation of our model in terms of the subjective video quality, by taking into account the viewer's subjective rating. In addition, the future evaluation has to cover the objective video quality. Also, we may further extend the work by proposing and comparing more rate adaption models. Since our model shows improvement in packet loss fraction in some cases, adding packet loss fraction to our model may control packet loss fraction decreasing and increasing.

References

[1] *A* Comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas (1st ed.). Retrieved from http://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf

[2] Ait-Hellal, O., & Altman, E. (2000). Analysis of TCP Vegas and TCP Reno. Telecommunication Systems, *15* (3-4), 381-404. http://dx.doi.org/10.1023/A:1019159332202.

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

[4] Brakmo, L., & Peterson, L. (1995). TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE J. Select. Areas Commun., 13* (8), 1465-1480.http://dx.doi.org/10.1109/49.464716

[5] Carlucci, G., De Cicco, L., & Mascolo, S. (2014). Modeling and control for web real-time communication. *In Decision and Control (CDC), 2014 IEEE 53rd Annual Conference, Los Angeles*, pp. 6824 – 6829.

[6] Cicco, L., Carlucci, G., & Mascolo, S. (2013). Experimental investigation of Google congestion control for real-time flows. *In FhMN '13 Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking*, USA.

[7] *Fake Webcam - Play video as webcam, apply effects on webcam*. Retrieved from http://www.fakewebcam.com/avatar.asp.

[8] Fall, K., & Floyd, S. (1996). Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, *26* (3), 5-21. http://dx.doi.org/10.1145/235160.235162.

[9] Floyd, S., & Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions On Networking*, *1* (4), 397-413. http://dx.doi.org/10.1109/90.251892.

[10] Forouzandeh, F. (2005). FPGA Implementation of Congestion Control Routers in High Speed Networks(Computer Science Master thesis). Concordia University.

[11] G., L., Cicco, D., & Mascolo, S. (2013). Understanding the Dynamic Behavior of the Google Congestion Control for RTCWeb. *In Packet Video Workshop (PV), 2013 20th International*. San Jose, CA.

[12] Google.com,. (2015). Google Hangouts. Retrieved 12 August 2015, from http://www.google.com/+/learnmore/hangouts/.

[13] Groups.google.com,. (2015). Google Groups. Retrieved 2015, from https://groups.google.com/forum/#!forum/discuss-webrtc.

[14] Hashem, E. (1989). Analysis of random drop for gateway congestion control, Cambridge.

[15] Tools.Ietf.org,. (2003). *RFC 3448:* TCP- Friendly Rate Control (TFRC): Protocol Specification. Retrieved from https://www.ietf.org/rfc/rfc3448.txt

[16] Jacobson, V. (1988). Congestion avoidance and control. *In SIGCOMM '88 Symposium proceedings on Communications architectures and protocols*, USA, pp. 314-329.

[17] Jacobson, V. (1990). Modified TCP Congestion Avoidance Algorithm. Retrieved from http://ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail.

[18] Jesup, H. (2012). *draft-jesup-rtp-congestion-reqs-00* - Congestion Control Requirements For Real Time Media. Tools.ietf.org. Retrieved from https://tools.ietf.org/html/draft-jesup-rtp-congestion-reqs-00.

[19] Johnston, A., & Burnett, D. (2013). *WebRTC*. St. Louis, MO: Digital Codex LLC.

[20] MARTIGNON, F. (2002). Enhanced Bandwidth Estimation and Loss Differentiation in the TCP Congestion Control Scheme, (Ph.D). Politecnico di Milano.

[21] Öhman, K. (2014). A Flexible Adaptation Framework for Real-Time Communication, (Master thesis). Luleå University of Technology.

[22] Software Informer, Network Emulator for Windows Toolkit. *Get the software safe and easy.*. Retrieved 2015, from http://network-emulator-for-windows-toolkit.software.informer.com/.

[23] Tools.ietf.org,. (1999). *RFC 2582* - The NewReno Modification to TCP's Fast Recovery Algorithm. Retrieved November 2015, from https://tools.ietf.org/html/rfc2582.

[24] Tools.ietf.org,. (2008). *RFC 5348* - TCP Friendly Rate Control (TFRC): Protocol Specification. Retrieved from https://tools.ietf.org/html/rfc5348.

[25] Tools.ietf.org, (2015). *draft-alvestrand-rtcweb-congestion-03* - A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. Retrieved July 2015, from https://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion-03.

[26] VishnuVardhan, S., & Chenna Reddy, P. (2012). Congestion Control in Real-Time Applications. *International Journal Of Computer Applications*, *51*(3), 33-37. http://dx.doi.org/10.5120/8025-1176.

[27] Webcam, V., & Mart, W. Virtual Webcam 8.0.2.454 Quick review - Free download - Free Virtual Webcam adds as a real camera. Download3k.com. Retrieved 2015, from http://www.download3k.com/Internet/Instant-Messengers-Chat/Download-Virtual-Webcam.html.

[28] Webrtc.org, (2015). WebRTC. Retrieved June 2015, from http://www.webrtc.org

[29] Webrtc.org, Development - *WebRTC*. Retrieved 2015, from http://www.webrtc.org/native-code/development.

[30] Welzl, M. (2005). Network congestion control, England.

[31] Wikipedia, (2015). Videoconferencing. Retrieved 1 August 2015, from https://en.wikipedia.org/wiki/Videoconferencing#Impact_on_the_general_public.