Experimental Investigation of the Google Congestion Control for Real-Time Flows

Luca De Cicco
Politecnico di Bari, Italy
l.decicco@poliba.it

Gaetano Carlucci
Politecnico di Bari, Italy
g.carlucci@poliba.it

Saverio Mascolo
Politecnico di Bari, Italy
mascolo@poliba.it

ABSTRACT

Enabling real-time communication over the Internet is of ever increasing importance due to the use of Internet for audio/video communication. The RTCWeb IETF working group has been established with the goal of standardizing a set of protocols for inter-operable real-time communication among Web browsers. In this paper we experimentally evaluate the Google Congestion Control (GCC) which has been recently proposed in the RTCWeb IETF WG. By setting up a controlled testbed, we have evaluated to what extent GCC flows are able to track the available bandwidth, while minimizing queuing delays, and fairly share the bottleneck with other GCC or TCP flows. We have found that the algorithm works as expected when a GCC flow accesses the bottleneck in isolation, whereas it is not able to provide a fair bandwidth utilization when a GCC flow shares the bottleneck with either a GCC or a TCP flow.

Categories and Subject Descriptors

H.4.3 [**Information systems applications**]: Communications Applications—*Computer conferencing, teleconferencing, and videoconferencing*

Keywords

Congestion Control, RTCWEB, RMCAT, Real-time flows

1. INTRODUCTION AND RELATED WORK

Enabling real-time communication over the Internet is a hot topic, due to the diffusion of broadband connections and mobile devices with enough processing resources to support high quality audio/video communication.

Despite the fact that video conferencing applications, such as Skype, have been widely used since more than one decade, an inter-operable and efficient set of standard protocols specifically designed for the transport of audio/video flows, is still missing. Recently, IETF and W3C have established two joint working groups: 1) IETF RTCWeb aims at standardizing a set of protocols such as a congestion control algorithm to transport real-time flows; 2) W3C WebRTC aims at standardizing a set of HTML5 APIs to enable real-time communication within Internet browsers.

In this paper we carry out an experimental investigation of the congestion control algorithm proposed by Google within the RTCWeb IETF WG, which has already been implemented in the Google Chrome and Firefox browsers. In particular, by using a controlled testbed which allows bandwidth and propagation times to be set, we investigate to what extent the congestion control algorithm is able to 1) fully utilize the available bandwidth, 2) fairly share the bottleneck bandwidth with concurrent flows, and 3) contain queuing delays. To the best of our knowledge, this is the first experimental investigation of the GCC.

The design of an efficient congestion control algorithm for multimedia traffic is a long standing and open issue. The rate-based approach is the favorite one since it produces a smoother traffic *wrt* window-based algorithms. In this category, several congestion control algorithms have been proposed, among which we cite the TCP Friendly Rate Control (TFRC) [4] and the Rate Adaptive Protocol (RAP) [11].

The use of a delay-based approach, instead of the classic loss-based technique employed by the TCP, to transport delay-sensitive traffic is a long debated issue [1]. In fact, it is well-known that the probing phase employed by loss-based algorithms tends to fill the bottleneck queue and, as a consequence, flows can be affected by delays that are unacceptable for real-time communication [5].

Main issues which have been addressed in delay-based algorithms are measurements [10] and fairness issues when sharing the bottleneck with loss-based flows [1, 6]. For instance, it has been shown that TCP Vegas, the first most known delay-based algorithm, is not able to get the fair share when competing with TCP NewReno or TCP Westwood+ [6]. In [13] Sprout, a stochastic-based algorithm, has been proposed to contain delays while maximizing the throughput; Sprout has been experimentally evaluated in a wireless emulated where exhibits an improvement *wrt* Skype, Facetime, and Hangout in a single flow scenario.

Nowadays, several video conferencing applications are available such as Skype, iChat, Google Hangout, and Cisco Movi. An evaluation of these applications can be found in [14]. In [2, 3] it has been shown that both Skype VoIP and Skype Video employ a congestion control algorithm which adjusts the sending rate to match the time varying network bandwidth.

In 2011 the IETF RTCWeb working group[1] has been established with the aim of standardizing a set of protocols to enable real-time audio and video communication within any browser, without the need of installing any plug-in or additional third-party software. An overview of WebRTC features can be found in [8].

The Network Assisted Dynamic Adaptation (NADA) congestion control algorithm [15] has been proposed within the RMCAT IETF working group[2] by Cisco. The algorithm regulates the sending rate based on both implicit and explicit congestion notifications (ECN). At the date of this writing, no implementation has been provided.

The paper is organized as follows: Section 2 describes the Google congestion control; Section 3 describes the experimental testbed and the metrics employed for the Google congestion control evaluation; Section 4 shows the experimental results and Section 5 concludes the paper.

2. GOOGLE CONGESTION CONTROL

The Google Congestion Control (GCC) algorithm in [9] runs over the UDP and it encapsulates the audio/video frames in RTP packets. It has been implemented in the open-source WebRTC that is available in the latest versions of the web browser Google Chrome. The congestion control is applied only to the video streams since the audio streams bitrate are considered negligible.

Figure 1 shows the main components involved in the congestion control scheme. GCC employs two controllers: a *sender-side controller,* which computes the target sending bitrate A_s, and a *receiver-side controller,* which computes the rate A_r that is sent to the sender under the conditions given in Section 2.2. In the following we describe how the two controllers compute the rates A_s and A_r.

2.1 The sender-side congestion control

The sender-side controller is a *loss-based* congestion control algorithm that acts every time t_k the k-th RTCP report message arrives at the sender or every time t_r the r-th REMB message, which carries A_r, arrives at the sender. The frequency at which RTCP reports are sent is time varying and it also depends on the backward-path available bandwidth; the higher the backward-path available bandwidth, the higher is the RTCP reports frequency. The REMB messages will be discussed in Section 2.2. The RTCP reports include the *fraction of lost packets* $f_l(t_k)$ computed as described in [12]. The sender uses $f_l(t_k)$ to compute the sending rate $A_s(t_k)$, measured in kbps, according to the following equation:

$$A_s(t_k) = \begin{cases} \max\{X(t_k), A_s(t_{k-1})(1-0.5f_l(t_k))\} & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1}) + 1\text{kbps}) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases} \tag{1}$$

where $X(t_k)$ is the TCP throughput equation used by the TFRC [4]. The rationale of (1) is the following: 1) when the fraction lost is considered small ($0.02 \le f_l(t_k) \le 0.1$), A_s is kept constant, 2) if a large fraction lost is estimated ($f_l(t_k) > 0.1$) the rate is multiplicatively decreased, but not below $X(t)$, whereas, 3) when the fraction lost is considered negligible ($f_l(t_k) < 0.02$), the rate is linearly increased.

Finally, when a REMB is received at time t_r, A_s is set as:

$$A_s(t_r) \leftarrow \min(A_s(t_r), A_r(t_r)). \tag{2}$$

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

2.2 The receiver-side controller

The receiver-side controller is a *delay-based* congestion control algorithm which computes A_r according to the following equation:

$$A_r(t_i) = \begin{cases} \eta A_r(t_{i-1}) & \text{Increase} \\ \alpha R(t_i) & \text{Decrease} \\ A(t_{i-1}) & \text{Hold} \end{cases} \tag{3}$$

where, t_i denotes the time the i-th group of RTP packets carrying a video frame is received, $\eta \in [1.005, 1.3]$, $\alpha \in [0.8, 0.95]$, and $R(t_i)$ is the receiving rate measured in the last 500ms. Figure 1 shows the “arrival-time filter”, the “over-use detector” and the “remote rate controller” blocks involved in the receiver-base controller; the “remote rate controller” is a finite state machine (see Figure 2) in which the state of (3) is changed by the signal produced by the “over-use detector” based on the output of the arrival-time filter. Finally, $A_r(t_i)$ cannot exceed $1.5R(t_i)$.

In the following we provide more details on the blocks. The goal of the *arrival-time filter* is to estimate the queuing time variation $m(t_i)$. To the purpose, it measures the one way delay variation $d_m(t_i) = t_i - t_{i-1} - (T_i - T_{i-1})$, where T_i is the timestamp at which the i-th video frame has been sent and t_i is the timestamp the at which it has been received. In [9] the one way delay variation is considered as the sum of three components: 1) the transmission time variation, 2) the queuing time variation $m(t_i)$, and 3) the network jitter $n(t_i)$. In [9] the following mathematical model is proposed:

$$d(t_i) = \frac{L(t_i) - L(t_{i-1})}{C(t_i)} + m(t_i) + n(t_i) \tag{4}$$

where $L(t_i)$ is the i-th video frame length, $C(t_i)$ is an estimation of the path capacity, and $n(t_i)$ is the network jitter modeled as a Gaussian noise. With this model, it is possible to extract from the one way delay variation $d(t_i)$ the queuing time variation $m(t_i)$. In particular, in [9] a Kalman filter computes $[1/C(t_i), m(t_i)]^T$ to steer to zero the residual measurement error $d(t_i) - d_m(t_i)$.

The goal of the *over-use detector* is to produce a signal to drive the state of (3) based on $m(t_i)$. The rationale is the following: if $m(t_i)$ increases above a threshold, and keeps increasing for a certain amount of time, or for a certain amount of consecutive frames, it is assumed that the network is congested and thus the “overuse” signal is triggered. On the other hand, if $m(t_i)$ decreases below a threshold, the network is considered underused and the “underuse” signal is generated. When $m(t_i)$ is close to zero, the network should be considered stable and the “normal” signal is generated.

The goal of the *remote rate controller* is to compute A_r according to (3) by using the signal produced by the over-use detector, which drives the finite state machine shown in Figure 2.

The signaling from the receiver to the sender is done through REMB messages carrying A_r or through RTCP reports [12]. In [9] the frequency at which the REMB messages should be sent is still considered an open issue. However, in the Google Chrome implementation REMB messages are sent either every 1s, if A_r is decreasing, or immediately, if A_r decreases more than 3%.

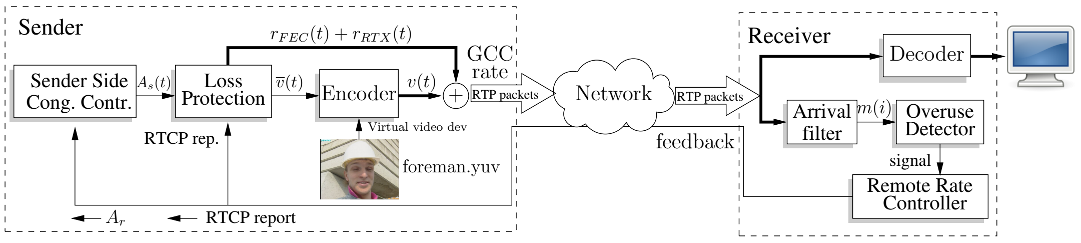


Figure 1: Congestion control architecture

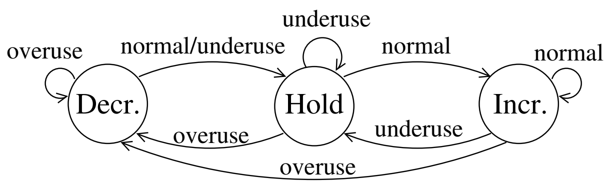


Figure 2: Remote rate controller finite state machine

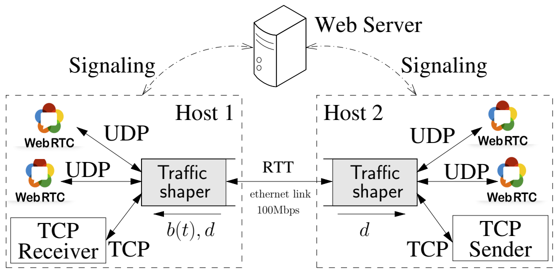


Figure 3: Experimental testbed

2.3 Loss protection

The Google Chrome implementation of the GCC algorithm [9] employs Forward Error Correction (FEC) and retransmissions to counteract packet losses. In particular, the sender side congestion controller produces a target sending bitrate $A_s(t)$ which feeds the *loss protection* block shown in Figure 1. The loss protection block computes $\overline{v}(t)$, the target encoding rate, which feeds the video encoder. If the loss protection is not active, $\overline{v}(t) = A_s(t)$, otherwise $\overline{v}(t) = A_s(t) - r_{RTX}(t) - r_{FEC}(t)$, where $r_{FEC}(t)$ is the sending rate of redundant video frames and $r_{RTX}(t)$ is the retransmission rate. Moreover, the FEC rate $r_{FEC}(t)$ cannot be more than half of the total sending rate $A_s(t)$, i.e. $r_{FEC}(t) \leq 0.5A_s(t)$. Finally, Google Chrome retransmits at most $A_s(t) \cdot RTT$ bytes of video data when it receives NACK messages.

3. EXPERIMENTAL TESTBED

Figure 3 shows the testbed employed to evaluate the performance of the congestion control algorithm. Two hosts, Host 1 and Host 2, are connected through a bottleneck emulating a WAN scenario. The NetEm linux module along with the traffic shaper `tc`[3] to set delays and available bandwidth on the bottleneck. In particular, the traffic shaper on the Host 1 sets a one-way delay d and a limitation on the available bandwidth $b(t)$ for the traffic that is received form the Host 2. The traffic shaper on the Host 2 only sets the one-way delay d for the traffic that comes from the Host 1 and does not set any bandwidth limitation. Thus, the minimum round trip time of the connection is $RTT_m = 2d$. The buffers have been set to 60KB to emulate a typical home gateway [7]. The considered bandwidths are in the range $[500, 3000]$kbps which are the typical of ADSL uplink speeds and cable connections [7]. On Host 1 `tcpdump`[4] has been used to measure the received bitrate of both video and TCP flows.

Both the hosts run a Chromium browser which generates the video flows. The GCC is implemented in the WebRTC sources[5] used by the Chromium browser. We have modified the WebRTC sources to log the key variables involved in the congestion control. Host 1 is equipped with a TCP receiver and Host 2 runs a TCP sender which uses the TCP Cubic congestion control, since it is the default version used by the Linux kernel. The TCP sender logs the congestion window, slow-start threshold, RTT, and sequence number. A web server[6] provides the HTML pages that handle the signaling between the peers using the `PeerConnection` javascript API.

The same video sequence is used as input to the WebRTC video encoders to enforce experiments reproducibility. Towards this end, the linux kernel module `v4l2loopback`[7] is used to create a virtual webcam device which cyclically repeats the *Foreman*[8] YUV test sequence. We have measured that, without bandwidth limitations, the WebRTC encoder limits $A_s(t)$ to the maximum value of 2Mbps.

In the following we describe the metrics employed to assess the performance of GCC in the considered scenarios. For each experiment, we compute the following metrics:

Channel Utilization $U = R/b$: where b is the known available bandwidth and R is the average received rate measured by using `tcpdump`.

Good Utilization $g = v/b$: where v is the average video bitrate without considering the bandwidth used for FEC and retransmissions.

Loss ratio $l =$ (lost bytes)/(received bytes): it is measured by the traffic shaper tool.

4. RESULTS

In this section we present the results of the experimental evaluation we have carried out by employing the testbed and the scenarios described in Section 3. We will show the dynamics of key variables such as the video flow rate, the FEC rate $r_{FEC}(t)$, the retransmission rate $r_{RTX}(t)$, the RTT, and we will show to what extent the GCC satisfies the re-

[3]http://lartc.org/

[4]http://www.tcpdump.org/
[5]http://code.google.com/p/chromium/
[6]http://code.google.com/p/webrtc-samples/
[7]https://github.com/umlaeute/v4l2loopback
[8]http://www.cipr.rpi.edu/resource/sequences/sif.html

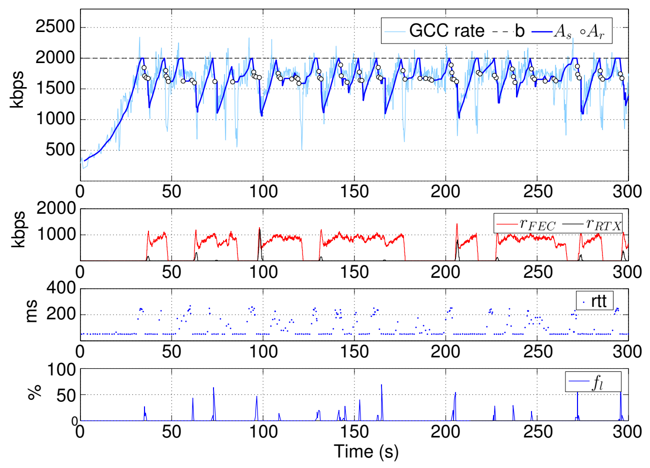


Figure 4: One GCC video flow over a link with 2000kbps capacity and $RTT_m = 50$ms

quirements defined in the IETF RMCAT[9] working group. Among the other features, the WG[10] requires that a congestion control algorithm for multimedia flows should provide low queuing and jitter delays when in the absence of competing heterogeneous traffic and a reasonable share of bandwidth when competing with other homogeneous or heterogeneous flows.

4.1 One GCC flow over a bottleneck with constant available bandwidth

In this Section we investigate the performance of a single video flow over a bottleneck with a constant available bandwidth. The available bandwidth $b(t)$ has been set to $b_i \in \{500, 1000, 1500, 2000\}$kbps and four values of the propagation delay $RTT_m = 2d$ have been considered, i.e. $RTT_{m,j} \in \{30, 50, 80, 120\}$ms. For each of the 16 couples $(b_i, RTT_{m,j})$, we have run 5 tests and we have evaluated the metrics defined in Section 3 by averaging over the 5 experiments.

Figure 4 shows the results of an experiment in which a single GCC video flow is sent over a bottleneck with an available bandwidth $b = 2000$kbps and an $RTT_m = 50$ms. The figure shows that the sending rate is set by using the sender-side congestion control law (1) unless a REMB message is received carrying a new value of A_r which is the output of the receiver-side congestion control law (3). Figure 4 shows the couples $(t_r, A_r(t_r))$, i.e. the event of a REMB reception along with the value of A_r contained in the message. The figure confirms that, when the RTT significantly increases, REMB messages are sent to prevent the sender-side algorithm to further increase the sending rate. Moreover, when the sender detects an increased value of the *fraction loss*[11] f_l, retransmissions are triggered and the FEC action is activated. It can be seen that, when the FEC action is on, $r_{FEC}(t)$ is set to 50% of the sending rate, i.e. the maximum amount of FEC is used.

Figure 5 (a) shows, for each of the considered bottleneck bandwidth b_i and round trip delays $RTT_{m,j}$, the measured channel utilization U which is the sum of 1) the good channel utilization g, 2) the fraction of FEC defined as $r_{FEC}(t)/b$,

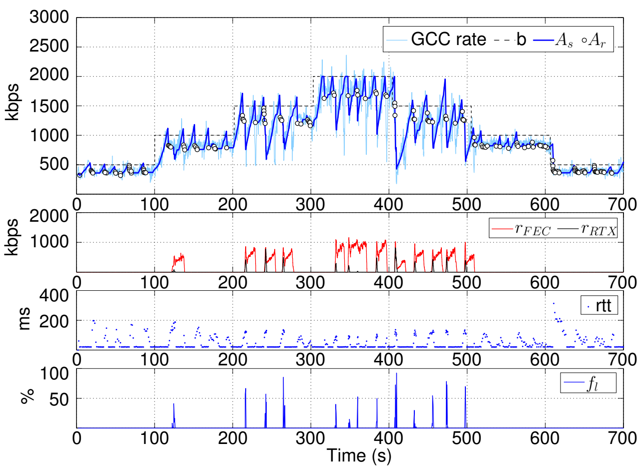


Figure 6: One GCC video flow over a stair-case available bandwidth and $RTT_m = 50$ms

and 3) the fraction of retransmissions defined as $r_{RTX}(t)/b$.

The figure shows that the channel utilization is slightly above 0.8 and is not significantly affected by the RTT_m or the available bandwidth. Figure 5 (b) shows that the loss ratio is not affected by the RTT, but it increases when the bandwidth increases and reaches a maximum value of 0.028. This confirms that the receiver-side delay-based controller works as expected, i.e. it decreases the rate when an increase of the queuing time is sensed, thus limiting the sending rate according to (2). Moreover, since the algorithm only reacts to queuing delay variations on the forward path, the measured metrics do not depend significantly on the RTT_m. Let us now focus on the loss protection that is used when packet losses are present. Figure 5 (a) shows that the fraction of bandwidth taken by the FEC is roughly proportional to the loss ratio l shown in Figure 5 (b), and in the case of $b = 2000$kbps, it is more than 20% of the available bandwidth.

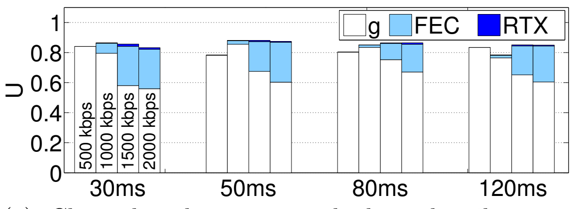
We have collected all the RTT samples reported in the RTCP feedbacks during all the experiments with the same available bandwidth b_i and we have computed the queuing delay, defined as $Q_{b_i}(t) = RTT(t) - RTT_m$. Based on the collected samples $Q_{b_i}(t)$, we have computed the four cumulative distribution function (CDF), one for each considered bottleneck bandwidth b_i, which are shown in Figure 5 (c). The figure shows that the median queuing delay is very close to zero, whereas the 90-th percentile is below 0.25s, confirming that the algorithm is able to contain the queuing delay.

4.2 One GCC flow over a variable available bandwidth bottleneck

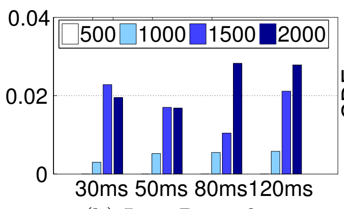
In this Section we investigate how the GCC throttles the sending rate when step-like changes of the available bandwidth occur. First, we have measured the transient time required to match a step-like increase of the available bandwidth. With this purpose, we have performed several experiments in which a GCC flow accesses a bottleneck whose bandwidth $b(t)$ changes from a minimum value of 400kbps to 3000kbps after 60s. We have found that GCC reaches a steady-state value of around 2Mbps after a transient time of roughly 30s.

With the information gathered by this simple experiment,

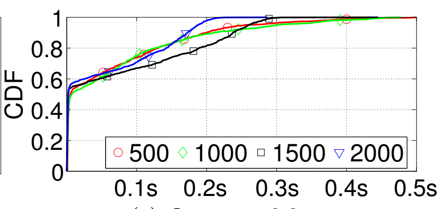
[9]http://tools.ietf.org/wg/rmcat/
[10]http://tools.ietf.org/html/draft-singh-rmcat-cc-eval-02
[11]The *fraction loss* is the percentage of lost RTP packets since the previous RTCP report was received [12].



(a) Channel utilization, good channel utilization, fraction of FEC, and fraction of retransmissions



(b) Loss Ratio l



(c) Queuing delay

Figure 5: One GCC flow over a bottleneck with constant capacity

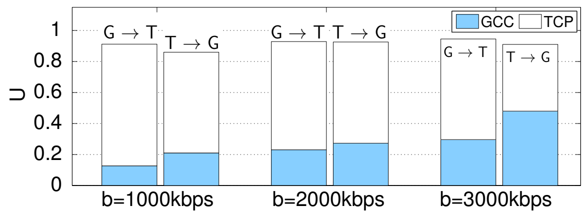


Figure 7: Channel Utilization when one GCC flow (G) shares the bottleneck with a TCP flow (T)

we now let the available bandwidth $b(t)$ to vary as a staircase to check how a GCC flow adapts the sending rate to increases and decreases of $b(t)$. In particular, starting from a bandwidth equal to 500kbps, b is increased every 100s of 500kbps until a bandwidth of 2000kbps is reached. Then, $b(t)$ is decreased using the same pattern. We have repeated this experiment for three different propagation delays, i.e. $RTT_{m,j} = \{30, 50, 80\}$ms. Due to the lack of space, we only show one experimental result obtained for $RTT_m = 50$ms. Figure 6 shows that the sending rate is able to quickly match the variable available bandwidth $b(t)$. Moreover, the figure confirms that the algorithm is able contain queuing delays thanks to the receiver-side congestion controller. Finally, the measured channel utilization is slightly above 80%.

4.3 One GCC with one concurrent TCP flow

This experiment investigates the behaviour of a GCC flow in the presence of a TCP flow. We consider three different available bandwidths, i.e $b_i \in \{1000, 2000, 3000\}$kbps with a propagation delay of $RTT_m = 50$ms. For each value of b_i, we have considered two cases: 1) a GCC flow starts at $t = 0$s and a TCP flow enters the bottleneck at $t = 100$s; 2) a TCP flow starts at $t = 0$s and a GCC flow is started at $t = 100$s.

Figure 7 shows the overall channel utilization and the TCP and GCC bandwidth shares obtained in this scenario. Three groups of bars are shown, each group for a different value of b_i: the first bar in the groups shows the channel utilization when GCC starts before TCP, the second when GCC starts after TCP. The figure clearly shows that at lower bandwidths GCC is not able to get a fair share. In the worst case, obtained when $b = 1000$kbps, the GCC flow utilizes only 13% of the channel capacity. Moreover, it appears that when the TCP flow starts first, the GCC flow is able to grab a slightly higher bandwidth share *wrt* the case when the TCP starts after the GCC flow.

To give a further insight on this issue, Figure 8 (a) shows the dynamics of the GCC flow and a TCP flow when $b = 1000$kbps and a TCP flow is started before the GCC flow. When the GCC flow joins the bottleneck it quickly gets a fair share until when, at around $t = 145$s, many consecutive REMB messages are received by the sender and the sending rate is decreased according to the value of A_r computed by the receiver-side congestion controller. The situation gets worse after $t = 220$s, when a large number of REMBs are received and the GCC flow is almost starved by the TCP.

Let us now look at Figure 8 (b) which shows the results obtained when $b = 3000$kbps: in this case the GCC sending bitrate is driven only by the sender-side controller (no REMB messages are received) and is able to get a larger bandwidth share *wrt* the TCP flow. This is due to the fact that the sender-side controller is loss-based and its probing phase is more aggressive than that of the TCP (see eq. (1)).

4.4 Two concurrent GCC flows

In this scenario we assess the fairness of two GCC flows when sharing a bottleneck with constant capacity. We consider a link with three different constant bandwidths $b_i \in \{1000, 2000, 3000\}$kbps and a propagation delay of 50ms. For each b_i we have run 5 tests.

Figure 9 shows the results of three experiments carried out with $b = 1000$kbps. It shows that the two GCC flows exhibit an unpredictable behavior: in Figure 9 (a) the first flow does not leave the bandwidth to the second flow, that obtains only a 10% channel utilization; Figure 9 (b) shows the opposite situation in which the second flow starves the first one that only gets 21% of the channel bandwidth; finally, in Figure 9 (c) the two flows fairly share the bandwidth.

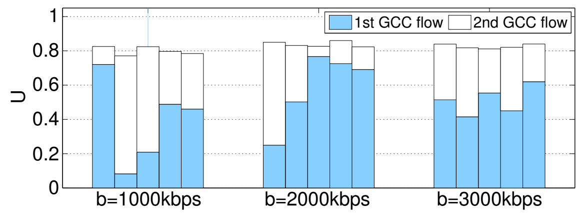
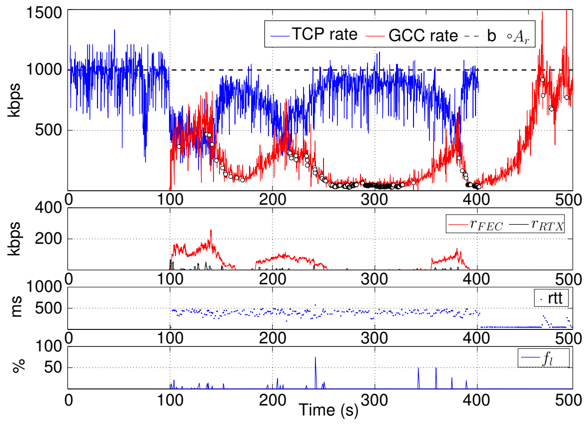


Figure 10: Channel utilization in the case of two GCC flows sharing a bottleneck ($RTT_m = 50$ms)

Figure 10 shows the channel utilization grouped for each of the considered available bandwidths b_i. Each bar in a group represents the channel utilization of the two GCC flows obtained in a single experiment. The figure shows that in the case of $b = 1000$kbps and $b = 2000$kbps a very poor fairness is obtained, whereas a better one is obtained when $b = 3000$kbps, even though the first flow always gets a higher bandwidth share.


(a) $b = 1000$kbps, $RTT_m = 50$ms

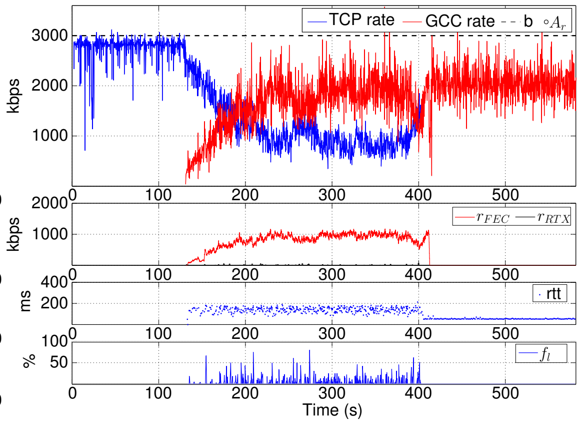
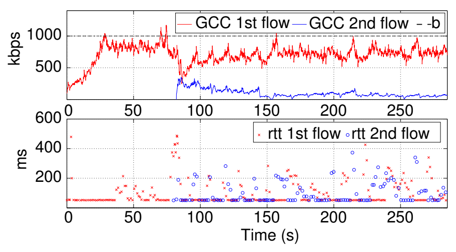
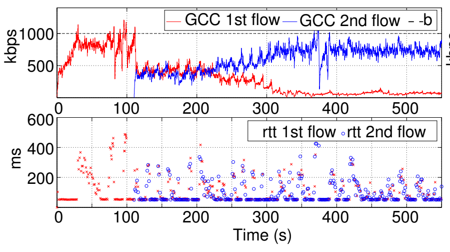

(b) $b = 3000$kbps, $RTT_m = 50$ms

Figure 8: A GCC flow sharing the bottleneck with a TCP flow


(a) The 2nd flow does not get a fair share


(b) The 2nd flow starves the 1st flow

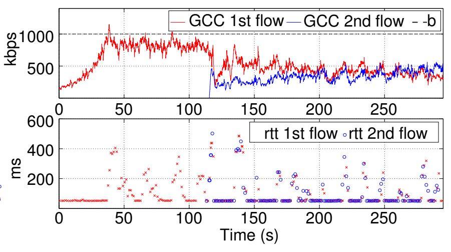

(c) The flows share the bandwidth fairly

Figure 9: Two GCC flows share a bottleneck with $b = 1000$kbps and $RTT_m = 50$ms

5. CONCLUSIONS AND FUTURE WORK

In this paper we have carried out an experimental investigation of the WebRTC congestion control proposed by Google (GCC). The main results we have found are: (1) when a single GCC flow accesses the bottleneck, the channel utilization is above 0.8, even when the available bandwidth changes following a stair-case pattern, and queuing delays are contained; (2) a video flow controlled by the GCC gets starved when sharing the bottleneck with a TCP flow if the bottleneck capacity is less or equal to 1000 kbps; (3) when two GCC video flows share the bottleneck, the algorithm behavior appears unpredictable and exhibit poor fairness. The cause of the unfairness issues exhibited by GCC when sharing the bottleneck with TCP, or with another GCC flow, is currently under investigation.

6. REFERENCES

[1] L. Budzisz et al. On the fair coexistence of loss- and delay-based tcp. *IEEE/ACM Trans. Netw.*, 19(6):1811–1824, Dec. 2011.

[2] L. De Cicco et al. Skype video congestion control: An experimental investigation. *Computer Networks*, 55(3):558–571, 2011.

[3] L. De Cicco and S. Mascolo. A Mathematical Model of the Skype VoIP Congestion Control Algorithm. *IEEE Trans. on Automatic Control*, 55(3):790–795, Mar. 2010.

[4] S. Floyd et al. TCP Friendly Rate Control (TFRC): Protocol Specification. *RFC 5348*, 2008.

[5] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *Comm. of the ACM*, 55(1):57–65, Jan. 2012.

[6] L. A. Grieco and S. Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM SIGCOMM CCR*, 34(2):25–38, 2004.

[7] C. Kreibich et al. Netalyzr: illuminating the edge network. In *Proc. ACM IMC '10*, pages 246–259, 2010.

[8] S. Loreto and S. P. Romano. Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *IEEE Internet Computing*, 16(5):68–73, 2012.

[9] H. Lundin et al. Google congestion control algorithm for real-time communication on the world wide web. *Draft IETF*, 2013.

[10] R. S. Prasad et al. On the effectiveness of delay-based congestion avoidance. In *Proc. PFLDNet '04*, 2004.

[11] R. Rejaie et al. RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. INFOCOM '99*, pages 1337–1345, 1999.

[12] H. Schulzrinne et al. RTP: A Transport Protocol for Real-Time Applications. *RFC 3550, Standard*, 2003.

[13] K. Winstein et al. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proc. USENIX NSDI '13*, April 2013.

[14] Y. Xu et al. Video telephony for end-consumers: measurement study of Google+, iChat, and Skype. In *Proc. ACM IMC '12*, pages 371–384, 2012.

[15] X. Zhu and R. Pan. NADA: A Unified Congestion Control Scheme for Real-Time Media. *Draft IETF*, Mar. 2013.