Understanding the Dynamic Behaviour of the Google Congestion Control for RTCWeb

Luca De Cicco, *Member, IEEE,* Gaetano Carlucci, and Saverio Mascolo, *Senior Member, IEEE*

***Abstract*—Real-time communication over the Internet is of ever increasing importance due the diffusion of portable devices, such as smart phones or tablets, with enough processing capacity to support video conferencing applications. The RTCWeb 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 focus on the Google Congestion Control (GCC), recently proposed in such WG, which is based on a loss-based algorithm run at the sender and a delay-based algorithm executed at the receiver. In a recent work we have shown that a TCP flow can starve a GCC flow. In this work we show that this issue is due to a threshold mechanism employed by the delay-based controller. By carrying out an extensive experimental evaluation in a controlled testbed, we have found that, when the threshold is small, the delay-based algorithm prevails over the loss-based algorithm, which contains queuing delays and losses. However, a small threshold may lead to starvation of the GCC flow when sharing the bottleneck with a loss-based TCP flow.**

***Keywords*—Real-time communication, RTCWeb, RMCAT, congestion control**

I. Introduction

According to a recent report by Cisco, today video traffic accounts for the most part of the global Internet traffic [4]. Despite the fact that video conference applications have been widely used over the Internet since more than one decade, an inter-operable and efficient set of standard protocols, specifically designed for audio/video flows, is still not available. Such applications generate *delay-sensitive* flows, i.e. the user perceived quality is affected, among other factors, by the latency of the connection [13]. Consequently such flows typically do not employ the TCP, which implements reliability through retransmissions at the cost of delayed delivery of packets, but favor the UDP that does not retransmits lost packets. Unfortunately, UDP does not implement congestion control.

Implementing an effective congestion control algorithm for video conferencing applications is crucial, to both avoid network congestion collapse, and meet real-time constraints, while mitigating packet losses due to congestion. Thus, well-designed delay-sensitive applications must adapt to network available bandwidth at least to some extent, such as in the case of Skype [6], [7] and other applications [22].

L. De Cicco, G. Carlucci, and S. Mascolo are with the Dipartimento di Ingegneria Elettrica e dell'Informazione at Politecnico di Bari, Via Orabona 4, 70125, Bari, Italy Emails: l.decicco@poliba.it, g.carlucci@poliba.it, mascolo@poliba.it

This work has been partially supported by the Italian Ministry of Education, Universities and Research (MIUR) through the PLATINO project (PON01 01007).

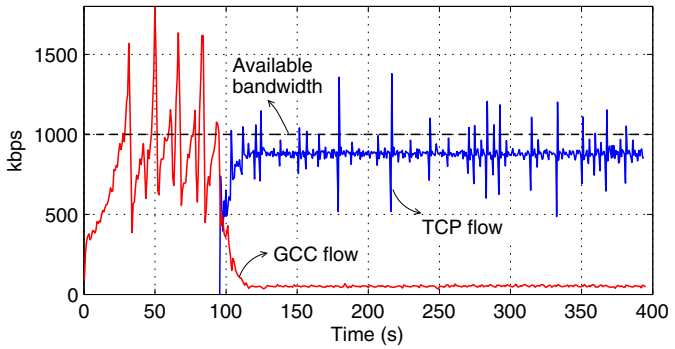


Figure 1. One GCC flow sharing a 1Mbps bottleneck with one TCP flow

The IETF working group (WG) *RTP Media Congestion Avoidance Techniques* (RMCAT) has been established in 2012 to propose the standardization of congestion control algorithms using the RTP [20]. Another working group, the IETF RTCWeb, aims at standardizing a set of protocols to enable inter-operable real-time communication among Web browsers. Among all the proposals submitted to the mentioned working groups, the one made by Google [16] has been already implemented in Chrome and Firefox browsers.

A preliminary experimental investigation of the Google Congestion Control (GCC) can be found in [5]. The algorithm is based on two controllers: a *loss-based* controller, executed at the sender, that probes the available bandwidth, and a *delay-based* one, running at the receiver, that aims at keeping the delay low. In [5] it has been shown that GCC provides low delay and a high channel utilization in the case there is no competing traffic, but it does not show a friendly behavior when it shares the bottleneck with a TCP flow.

Figure 1 shows the sending rate of one GCC flow and one TCP flow sharing a 1Mbps bottleneck with a propagation round trip time of 50ms. The figure clearly shows that the GCC flow gets starved by the TCP flow starting at $t = 95$s.

The aim of this paper is to further investigate the reasons of such unfriendly behavior when GCC flows coexist with concurrent TCP traffic. We have found that the cause of the unfriendly behavior is in the delay-based controller that employs a static threshold mechanism to trigger a sending rate reduction when an inflated one-way-delay is measured at the receiver.

In particular, through a customized and fully automated testbed allowing bandwidth, propagation times, and threshold to be set, we investigate to what extent the value of the threshold affects the performance of the congestion control algorithm in terms of 1) channel utilization, 2) queuing delays,

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

3) losses, and 4) friendliness when sharing the bandwidth with a concurrent TCP flow.

The paper is organized as follows: Section II reviews the literature related to congestion control for delay-sensitive flows and active queue management for containing queuing delays; Section III describes the GCC algorithm and provides details on the role of the threshold in the receiver side controller; Section IV describes the experimental automated testbed; Section V presents the experimental results and Section VI concludes the paper.

II. Related work

Delay-sensitive traffic, such as the one generated by video-conferencing applications, requires low queuing delays and packet losses, while ensuring a good channel utilization and fairness when competing with homogeneous and heterogeneous traffic. It is well-known that TCP, the most used transport protocol, is not suitable for real-time applications since it favors reliability through retransmissions over packet delivery timeliness; furthermore, since the standard TCP infers congestion when losses are detected, its probing phase tends to fill the bottleneck queue and, as a consequence, flows can be affected by delays that are unacceptable for real-time communication [9], especially in cellular networks that employ significantly large buffers possible yielding to several seconds of queuing delay [14], [1].

Two complementary approaches can be employed to tackle this issue: *end-to-end*, placing the control in the end-points, and *active queue management* (AQM), addressing the problem in the routers.

The first research efforts on the transport of delay-sensitive traffic were focused on the design of delay-based TCP congestion control variants to minimize end-to-end latency (see [2] and more recent proposals [3], [11]). The main issues which have been addressed in delay-based algorithms are related to measurements [19] and fairness when sharing the bottleneck with loss-based flows [3], [10], [11]. In [10] it has been shown that TCP Vegas [2], a well-known delay-based algorithm, is not able to get the fair share when competing with TCP NewReno or TCP Westwood+. To tackle such fairness issues, in [3], [11] the proposed algorithms are able to infer the presence of concurrent loss-based flows and switch to a loss-based mode, returning to delay-based operations when loss-based flows are no longer present. Recently, a stochastic-based approach has been proposed to contain delays while maximizing the throughput [21]; it has been shown that the algorithm outperforms applications such as *Skype*, *Facetime* and *Hangout* in a single flow scenario, however, the performance in the case of multiple flows sharing a bottleneck has not been evaluated yet.

The Network Assisted Dynamic Adaptation (NADA) congestion control algorithm [23] has been recently proposed within the IETF RMCAT working group. The simulation results have shown that the algorithm is able to contain queuing delays and to provide a good fairness when several NADA flows share the bottleneck.

The Google Congestion Control (GCC) has been proposed within RMCAT and RTCWeb[1] working groups [16]. GCC has been implemented in Google Chrome and Firefox. A preliminary investigation on GCC has been conducted in [5].

AQM algorithms were originally proposed to tackle the "Lock-Out" phenomenon and absorb packet bursts while maintaining the steady-state queue size low. Despite the fact that many algorithms have been proposed over the past two decades, the wide adoption of those algorithms has been slowed down mainly due to difficulty in tuning the algorithms parameters. Recently, several AQM algorithms [17], [18] have been proposed to tackle the *bufferbloat* phenomenon [9]. Codel [17] (*Controlled Delay Active Queue Management*) aims at containing the queuing latency, instead of the queue length, while maximizing the throughput. Codel does not require any parameter tuning so it works across a wide range of conditions, with different links and round trip times. Codel rationale is the following: using per-packet timestamp, the algorithm computes the packet sojourn time in the queue, and based on this value, it evaluates whether the packet should be dropped or not. Another recent proposal is PIE (*Proportional Integral controller Enhanced*) [18] that, similarly to Codel, is self-tuning and aims at controlling the queuing latency. PIE employs a PI controller which computes a random probability to mark or drop packets, based on a target queuing latency. Authors argue that PIE require less computational efforts *wrt* Codel since PIE does not need to track the per-packet sojourn time.

III. Google Congestion Control

Figure 2 shows the architecture of the Google Congestion Control (GCC) that is composed of two algorithms: the *receiver-side controller* computes the rate A_r and sends it to the sender; the *sender-side controller* computes the target sending bitrate A_s that cannot exceed A_r. In this paper we mainly focus on the receiver-side congestion controller that aims at keeping the queuing delay small. The description reported in the following is based on both the draft [16] and an analysis of the Chromium code base.

A. 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 variable and it depends on the backward-path available bandwidth; the higher the backward-path available bandwidth, the higher is the RTCP reports frequency. The REMB format is an extension of the RTCP protocol [20] that is being discussed within the RMCAT WG[2] (see also Section III-B). The RTCP reports include the fraction of lost packets $f_l(t_k)$ computed as described in [20]. The sender uses $f_l(t_k)$ to compute the sending rate $A_s(t_k)$,

[1] http://datatracker.ietf.org/wg/rtcweb/

[2] http://tools.ietf.org/html/draft-alvestrand-rmcat-remb-02

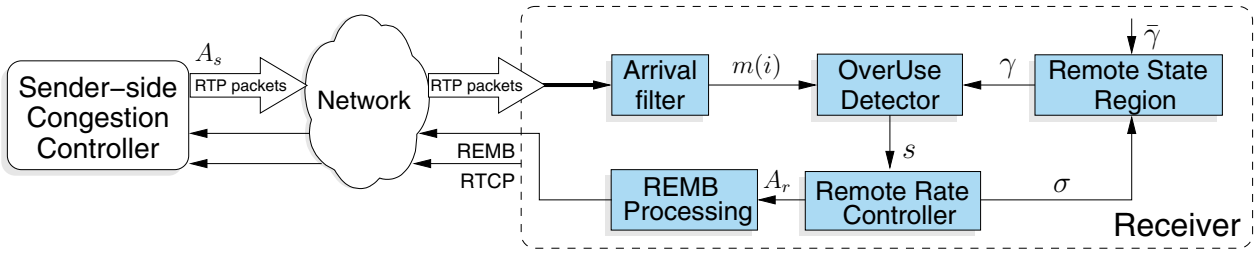


Figure 2. Google congestion control architecture showing the receiver internal structure

measured in kbps, according to the following equation:

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

where $X(t_k)$ is the TCP throughput equation used by the TFRC [8]. The rationale of (1) is the following: 1) when the fraction lost is considered small ($0.02 \leq f_l(t_k) \leq 0.1$), A_s is kept constant, 2) if a high 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 multiplicatively increased.

After A_s is computed through (1), the following assignment is performed to ensure that A_s never exceeds the last received value of A_r:

$$A_s \leftarrow \min(A_s, A_r). \quad (2)$$

B. The receiver-side controller

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

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

where, t_i denotes the time the i-th group of RTP packets carrying one 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 2 shows a detailed block diagram of the receiver-side controller that is made of several components described in the following.

The *remote rate controller* is a finite state machine (see Figure 3) in which the state σ of (3) is changed by the signal s produced by the *over-use detector* based on the output of the *arrival-time filter*. The *remote state region* block sets the threshold γ employed by the *over-use detector* according to the operating region of the receiver. The *REMB Processing* decides when to send A_r to the sender through a REMB message based on the value of the rate A_r. Finally, it is important to notice that $A_r(t_i)$ cannot exceed $1.5R(t_i)$.

In the following we give more details on each block.

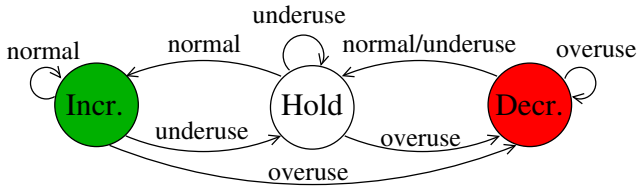


Figure 3. Remote rate controller finite state machine [16]

1) The arrival-time filter: 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 at which it has been received. The one way delay variation is considered as the sum of three components [16]: 1) the transmission time variation, 2) the queuing time variation $m(t_i)$, and 3) the network jitter $n(t_i)$. In [16] the following mathematical model is proposed:

$$d(t_i) = \frac{\Delta L(t_i)}{C(t_i)} + m(t_i) + n(t_i) \quad (4)$$

where $\Delta L(t_i) = L(t_i) - L(t_{i-1})$, $L(t_i)$ 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 [16] a Kalman filter computes $\theta(t_i) = [1/C(t_i), m(t_i)]^\intercal$ to steer to zero the residual $d(t_i) - d_m(t_i)$ and filter out the measurement noise.

2) The over-use detector: Every time t_i a video frame is received, the *over-use detector* produces a signal s that drives the state σ of the FSM (3) based on $m(t_i)$ and a threshold γ. The Algorithm 1 shows in details how s is generated: when $m(t_i) > \gamma$, the algorithm tracks the time spent in this condition by increasing the variable t_{OU} of the frame inter-departure time ΔT. When t_{OU} reaches $\bar{t}_{OU} = 100$ms and $m(t_i) > m(t_{i-1})$, the *overuse* signal is generated. On the other hand, if $m(t_i)$ decreases below γ, the *underuse* signal is generated, whereas the *normal* signal is triggered when $-\gamma \leq m(t_i) \leq \gamma$.

3) The remote state region: This block computes the threshold γ as follows: by default $\gamma = \overline{\gamma}$ with $\overline{\gamma} = 25/60$ms, however, when the system is considered to be close to the congestion, the threshold is halved, i.e. $\gamma = \overline{\gamma}/2$. In particular, γ is halved when $\sigma =$ decrease or when A_r is considerably lower that the incoming bitrate $R(t)$.

```
if |m_i| > γ then
    if m_i > 0 then
        t_OU ← t_OU + ΔT;
        if t_OU > T̄_OU then
            if m_i ≥ m_{i−1} then
                t_OU ← 0;
                s ← Overuse;
    else
        t_OU ← 0;
        s ← Underuse;
else
    t_OU ← 0;
    s ← Normal;
```

Algorithm 1: Over-use Detector pseudo-code

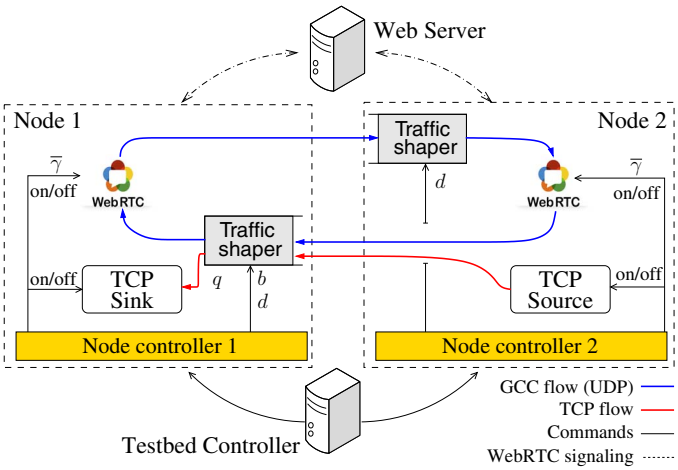


Figure 4. Experimental testbed

4) Remote rate controller: This block computes A_r according to (3) by using the signal s produced by the over-use detector, which drives the finite state machine shown in Figure 3.

5) REMB Processing: This block notifies the sender with the computed rate A_r through REMB messages. The REMB messages are sent either every 1s, or immediately, if $A_r(t_i) < 0.97A_r(t_{i-1})$, i.e. when A_r has decreased more than 3%.

IV. Experimental testbed

A. Testbed

Figure 4 shows our experimental testbed consisting of four Linux machines: two nodes running one Chromium browser[3] and IPerf to generate or receive long-lived TCP flows; another node runs a web server which handles the signalling required to establish the video calls between the browsers; the last node is the testbed *controller* which orchestrates the experiments. The testbed controller undertakes the following tasks: 1) it places the WebRTC calls in an automated way; 2) it sets the threshold $\overline{\gamma}$ employed by the receiver-side controller on both Node 1 and 2; 3) it sets the one-way delay d on both Node 1 and 2 resulting in a round-trip propagation delay $RTT_m = 2d$; 4) it sets the available bandwidth b and the buffer size q on Node 1, that represents the bottleneck.

The bottleneck, that emulates the WAN scenario, has been created through the NetEm linux module that imposes a one-way delay d in both the directions and with the token bucket filter (tbf) queuing discipline[4] that creates a bandwidth constraint b for the traffic received by Node 1. The buffers size on Node 1 have been set in the range $[15, 75]$kB according to the measurements reported in [15]. The considered bandwidths constraints are in the range $[500, 3000]$kbps which are the typical of ADSL uplink speeds and cable connections [15].

We have modified the WebRTC sources to log the key congestion control algorithm variables and allow the threshold $\overline{\gamma}$ to be set through a configuration file.

The TCP Source uses the TCP Cubic congestion control, the default version used by the Linux kernel, and logs the congestion window, the slow-start threshold, the RTT, and the sequence number.

The web server[5] provides the HTML pages that handle the signaling between the peers using the `PeerConnection` javascript API.

The node controller employs several bash scripts to send via `ssh` the commands to set up the testbed and manage the experiments.

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`[6] is used to create a virtual webcam device which cyclically repeats the *Foreman*[7] YUV test sequence. We have measured that, without bandwidth limitations, the WebRTC encoder limits $A_s(t)$ to the maximum value of 2Mbps while the minimum value for $A_s(t)$ is 50kbps. Finally, the audio stream has been turned off since its bit rate can be considered negligible.

B. Metrics

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;`
- **Loss ratio** $l =$ (packets lost)/(packets received), measured by the traffic shaper tool;
- **number of delay-based decrease events** n_{dd}, i.e. the number of times that a received REMB message reduces the sending rate A_s to the rate A_r computed by the delay-based controller according to (2);
- **Queuing delay** T_q, measured averaging the value $RTT(t) - RTT_m$, over all the RTT samples reported in the RTCP feedbacks during an experiment.

[3] http://code.google.com/p/chromium/

[4] http://lartc.org/

[5] http://code.google.com/p/webrtc-samples/

[6] https://github.com/umlaeute/v4l2loopback

[7] http://www.cipr.rpi.edu/resource/sequences/sif.html

Table I. PARAMETERS EMPLOYED IN THE EXPERIMENTAL EVALUATION

	Parameter	Values	
Chromium	$\overline{\gamma}$ (ms)	10/60, 15/60,..., 60/60, 65/60	
	Call duration (s)	1 GCC	300
		1 GCC vs 1 TCP	400
Bottleneck	q - Queue length (kB)	15, 30, 45, 60, 75	
	RTT_m(ms)	50	
	b - Bandwidth (Mbps)	1 GCC	0.5, 1.0, 1.5, 2.0
		1 GCC vs 1 TCP	1.0, 2.0, 3.0

V. RESULTS

In this section we present the experimental results obtained by employing the testbed described in Section IV. In total, we have measured more than 1200 video calls established between two Chromium browsers, corresponding to roughly 120 hours of calls.

Two scenarios have been considered: (S1) a single GCC flow over a bottleneck; (S2) a GCC flow sharing the bottleneck with a TCP flow. Table I summarizes the bottleneck (queue length, RTT, bandwidth) and GCC flow parameters (the threshold $\overline{\gamma}$) employed in the experimental evaluation.

For both the considered scenarios, and for each of the combinations of the parameters shown in Table I, we have run three experiments and evaluated the metrics defined in Section IV-B by averaging over the three experiments. For the single GCC flow scenario $b \in \{0.5, 1.0, 1.5, 2.0\}$Mbps, whereas in the "one GCC vs one TCP" scenario $b \in \{1.0, 2.0, 3.0\}$Mbps.

In particular, the effect of the threshold $\overline{\gamma}$ on the metrics defined in Section IV-B will be investigated; moreover we will assess to what extent GCC satisfies the requirements defined by the IETF RMCAT[8] working group[9]. Among the requirements, the algorithm 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. In particular, a video flow should not get starved when sharing the bottleneck with other flows, and it should not starve the other flows.

A. Investigating the influence of γ on the performance of a single GCC flow accessing a bottleneck

In this Section we investigate the influence of the threshold $\overline{\gamma}$ on the performance of a single GCC flow over a bottleneck link using the parameters shown in Table I. In this scenario 720 experiments have been carried out resulting in 60 hours of video calls.

Figure 5 shows the results obtained with a buffer size equal to 30kB. Let us focus on the number of delay-based decrease events n_{dd} as function of the threshold $\overline{\gamma}$: for each of the considered bottleneck bandwidths, n_{dd} decreases when $\overline{\gamma}$ increases. The decrease of n_{dd} is due to the fact that, with a larger $\overline{\gamma}$, a larger variation of $m(t_i)$ is allowed without triggering the *overuse* signal according to Algorithm 1. Thus, with a large $\overline{\gamma}$, the algorithm favors the use of the loss-based sender-side controller over the delay-based one. As a consequence, a better channel utilization is achieved at the expense of a larger queueing delay and a higher packet loss ratio.

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

[9]http://tools.ietf.org/html/draft-singh-rmcat-cc-eval-02

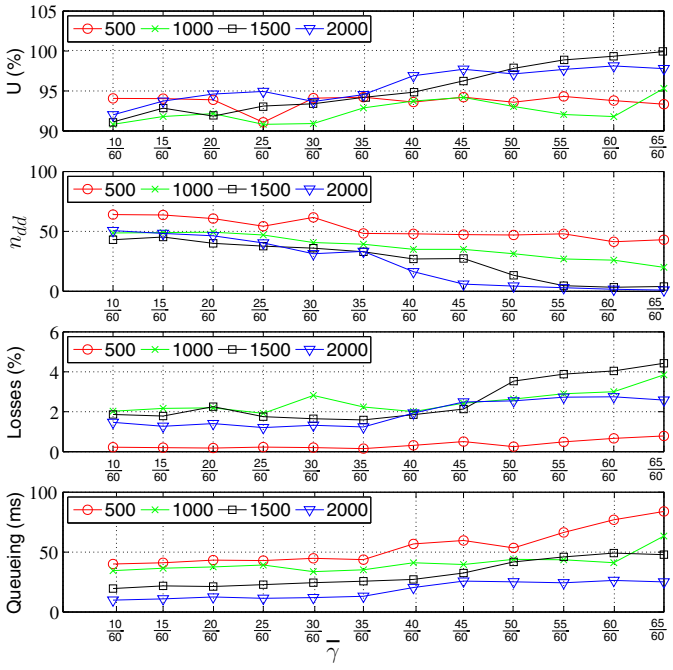
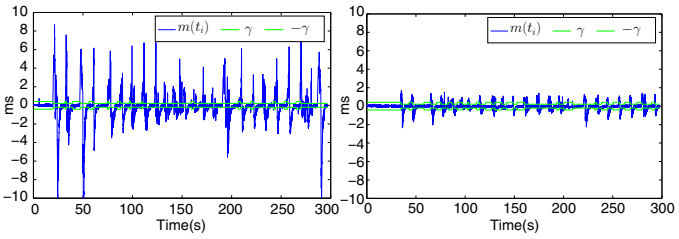


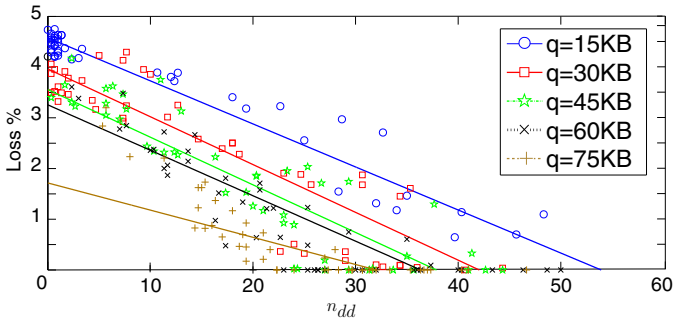
Figure 5. One GCC flow: Channel Utilization U, number of delay-based decrease events n_{dd}, packet loss ratio l, queuing delay T_q, in the case of $q =$ 30 kB and a constant available bandwidth $b \in \{500, 1000, 1500, 2000\}$



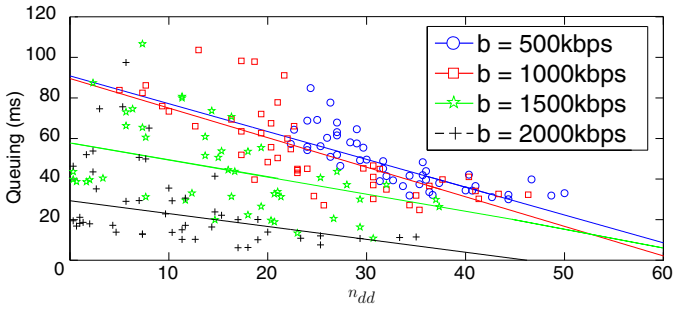
(a) $b_i = 500$kbps (b) $b_i = 2000$kbps

Figure 6. Queuing delay variation $m(t_i)$ measured with $\bar{\gamma}$ =25/60ms, $q_i =$ 30kB at $b_i = 500$kbps and $b_i = 2000$kbps

Figure 6 compares the measured queuing delay variation $m(t_i)$ when b is either 500kbps or 2000kbps: in the case $b = 500$kbps the *arrival filter* measures a larger queuing delay variation and, as a consequence, a larger number delay-based decrease events is triggered; on the other hand, when $b = 2000$kbps, a small queuing delay variation is measured and less delay-



(a) Loss ratio function of n_{dd}. Data grouped by buffer size q_i.



(b) Queuing delay function of n_{dd}. Data grouped by available bandwidth b_i.

Figure 7. Queuing delay and loss ratio function of n_{dd}

based decrease events occur. This explains the larger packet losses exhibited by the GCC flow with higher values of b.

To get a further insight of the influence of n_{dd} on the performance of the algorithm, Figure 7 shows the packet loss percentage and the queuing time as a function of n_{dd}.

In particular, Figure 7(a) shows the pairs (n_{dd}, l) grouped by the value of the buffer size q; for each group we also show the linear regression of the data. The negative slope of the linear regression highlights the fact that the losses decrease when the number of delay-based events increases. Furthermore, the figure shows that the losses decrease when the buffer size increases. Interestingly, when the buffer size is large enough, the delay-based algorithm is able to completely avoid losses as it can be inferred by looking at the samples on the x-axis.

Figure 7(b) shows the data grouped by the available bandwidth b; for each group it is also shown the linear regression of the data. As expected, the queuing delay is affected by the available bandwidth b, exhibiting larger values of T_q for smaller bandwidths according to (5). Moreover, for a fixed value of b, the queuing time decreases when n_{dd} increases, confirming that the delay-based component of GCC is able to contain the queuing delays.

B. The influence of the threshold γ on the friendliness with a concurrent TCP flow

In this scenario we investigate the influence of γ on the behavior of a GCC flow in the presence of a concurrent TCP flow. The parameters employed in this scenario are summarized in Table I. In total, in this scenario 540 experiments have been carried out resulting in 60 hours of video calls.

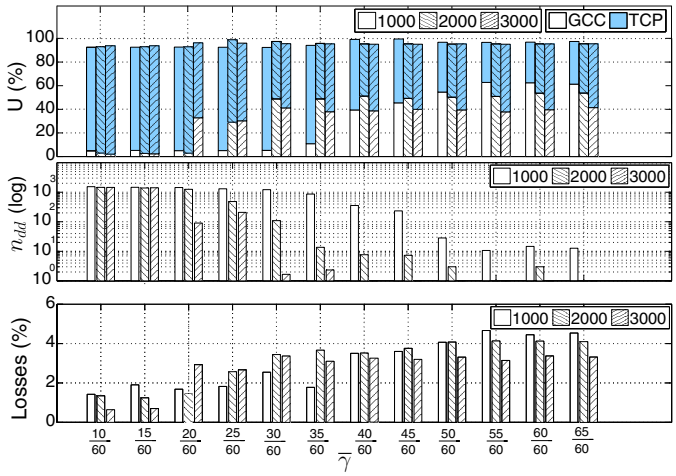


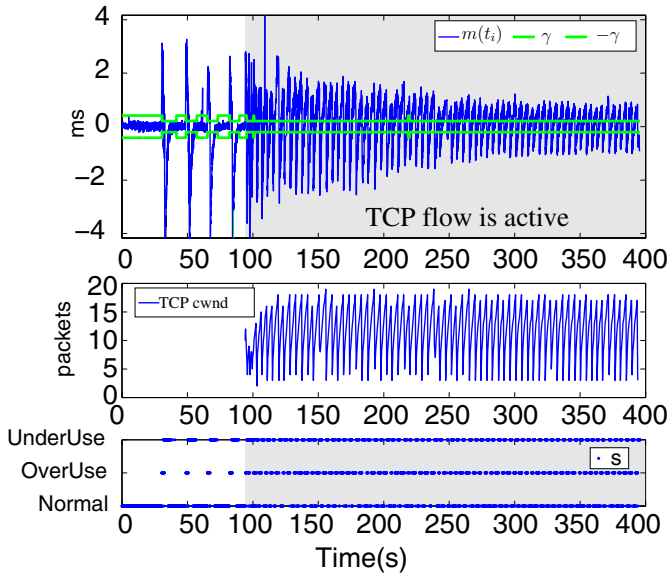
Figure 8. A GCC flow with a concurrent TCP flow: channel utilization U, number of delay-based decrease events n_{dd}, packet loss ratio l, in the case of $q = 30$ kB and a constant available bandwidth $b \in \{1000, 2000, 3000\}$kbps

The GCC flow starts at $t = 0$s and a TCP flow is started at $t = 100$s; every video call lasts 400s so that the GCC flow and the TCP flow share the bottleneck for 300s. The metrics defined in Section IV-B have been evaluated only in the time interval $t \in [100, 400]$ where both the GCC flow and the TCP flow were sharing the bottleneck.

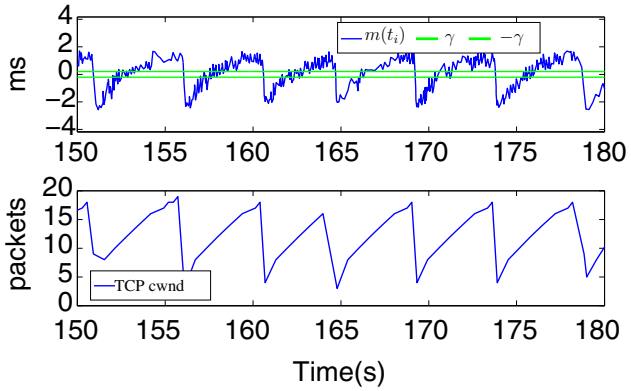
Figure 8 shows the measured metrics obtained with a buffer size $q = 30$kB. Figure 8 clearly shows that the threshold $\overline{\gamma}$ has a remarkable impact on the friendliness between the two flows sharing the bottleneck: when $\overline{\gamma}$ is small, starvation of the GCC flow occurs; in particular, when $\gamma < 40/60$ms we observed the starvation of the GCC flow when $b = 1000$kbps; on the other hand, with an increased value of $\overline{\gamma}$, the GCC flow was able to get the fair share. The number of delay-based decrease events n_{dd} is remarkably higher *wrt* the single flow scenario: in particular, for low values of $\overline{\gamma}$ we have measured $n_{dd} > 1000$ whereas in the single flow scenario the maximum value we have measured was less than 100 (see Figure 5).

To get a further insight, Figure 9 (a) shows the dynamics of the queuing time variation $m(t_i)$, the signal s generated by the *over-use detector,* and the congestion window (cwnd) of the TCP flow in the case of $\overline{\gamma} = 25/60$ms, $b = 1000$kbps, $q = 30$kB. It is clear that, when the TCP flow starts at $t = 100$s, $m(t_i)$ oscillates over the threshold $+\gamma$ having the effect of generating a large number of *overuse* signals. Consequently, the remote rate controller FSM (Figure 3) is driven to the *decrease* mode reducing the value of A_r according to (3).

This can be observed in Figure 10 (a), which shows the state σ of the remote rate controller FSM, and in Figure 10 (b), which shows the effects of σ on the rate A_r. In particular, when the TCP flow joins the bottleneck at $t = 100$s, A_r gets exponentially decreased from a value greater than the available bandwidth b to roughly 50kbps in about 20s. Thus, the REMB messages carrying A_r, that are periodically sent by the receiver-based controller to the sender, make the delay-based congestion controller to prevail over the loss-based

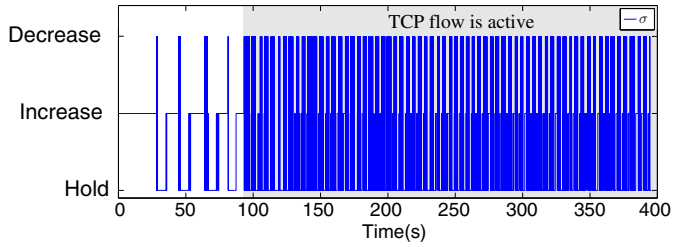


(a) Complete dynamics

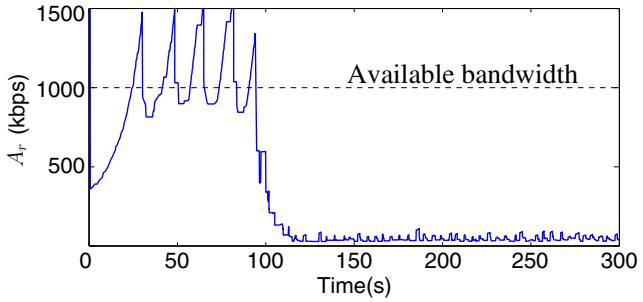


(b) Zoom of the time interval $t \in [150, 180]$

Figure 9. Queuing delay variation $m(t_i)$, TCP congestion window (cwnd), and over-use detector signal s in the case of one GCC coexisting with a TCP flow over a 1Mbps bottleneck with $q = 30$kB



(a) Remote rate controller state σ



(b) Remote rate A_r

Figure 10. Remote rate controller behavior in the case of one GCC coexisting with a TCP flow over a 1Mbps bottleneck with $q = 30$kB

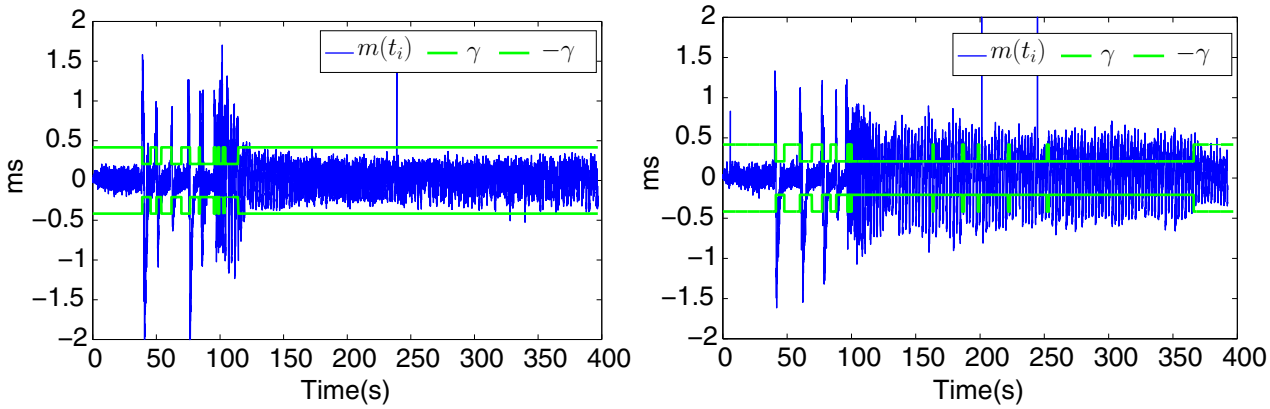
controller. This undesired behavior is due to the fact that the TCP congestion control loss-based algorithm tends to fill the bottleneck queue causing the queuing delay to oscillate. The *arrival filter* measures the queuing delay variation $m(t_i)$ due to both the GCC flow and the TCP congestion control. We argue that, in this experiment, the dynamics of $m(t_i)$ is mainly driven by the dynamics of the TCP flow. To confirm this, Figure 9 (b) shows the congestion window of the TCP flow and $m(t_i)$ in the time interval $t \in [150, 180]$s. The figure clearly shows that $m(t_i)$ follows the dynamics of *cwnd* and not that of the self-induced delay.

For a given value of $\overline{\gamma}$, the issue described above is less remarkable in the presence of higher available bandwidths: in fact, the measured queuing variation $m(t_i)$ will be lower (see Figure 6) and it is more likely that it will oscillate within the range $[-\overline{\gamma}, \overline{\gamma}]$; thus, *overuse* signals will not be generated and, as a consequence, the loss-based algorithm will prevail over the delay-based one; this is confirmed by the number of delay-based decreases events n_{dd} that, for a given $\overline{\gamma}$, is inversely proportional to b.

In particular, Figure 11 shows the dynamics of $m(t_i)$ of two experiment runs with $b = 2000$kbps, $\gamma = 25/60$ms, $q = 30kB$. In the first run, shown in Figure 11(a), the queuing delay variation induced by the TCP flow is well contained in the range $[-\overline{\gamma}, \overline{\gamma}]$ and the algorithm does not generate overuse signal, thus behaving as a loss-based flow and achieving a reasonable friendliness. Figure 11(b) shows that in the second run, the value of $\overline{\gamma}$ is halved by the "*remote state region*" (see Figure 2) thus generating a large number of *over-use* signals causing the GCC flow starvation. At this point it is important to notice that the situation gets worse with larger buffer sizes. In fact, with a larger buffer, the queuing delay variation induced by the TCP will be larger, leading the GCC to starvation even for higher values of the available bandwidth. Due to space constraints we cannot show the results for larger buffers. However, we have found that the threshold γ that is required to get a fair share increases with the buffer size.

VI. Conclusions

In this paper we have considered the Google Congestion Control (GCC) that has been recently proposed within the RTCWeb and RMCAT IETF working groups and implemented in Google Chrome and Firefox web browsers. In our previous work we have shown that GCC is not able to get the fair share when coexisting with a TCP flow [5]. In this paper, we have found in the *remote rate controller* the cause of such issue. In particular, it turns out that a threshold $\overline{\gamma}$ has a significant impact on the dynamics of the rate A_r computed by the remote rate controller. An extensive experimental evaluation has been



(a) $\overline{\gamma}$ is not halved: friendliness with the TCP flow is achieved

(b) $\overline{\gamma}$ is halved: the GCC flow gets starved

Figure 11. Measured queuing delay variation $m(t_i)$ in the case of concurrent TCP flow ($b = 2000$kbps, $\gamma = 25/60$ms, $q = 30$kB)

carried out to quantify the effect of $\overline{\gamma}$ on the channel utilization, the queuing delay, the loss ratio, and the fairness achieved by a GCC flow when coexisting with a TCP flow.

In particular, the results suggest that a static value of $\overline{\gamma}$ should not be used. In fact, with the default value used by Chromium browser, i.e. $\overline{\gamma} = 25/60$ms, the GCC algorithm obtained a satisfactory performance in the single flow scenario, but failed to prevent starvation when coexisting with TCP flows.

Regarding the single flow scenario, we have found that, if $\overline{\gamma}$ is not too large, GCC provides a good channel utilization while containing the queuing delays and packet losses. However, when $\overline{\gamma}$ increases the loss-based algorithm prevails over the delay-based one and larger queuing delays and packet losses are measured.

On the other hand, in the case a of GCC flow sharing the bottleneck with a TCP flow, a small value of $\overline{\gamma}$ makes the delay-based algorithm to prevail over the loss-based one, leading to the starvation of the GCC flow.

REFERENCES

[1] S. Alfredsson, G. Del Giudice, J. Garcia, A. Brunstrom, L. De Cicco, and S. Mascolo. Impact of tcp congestion control on bufferbloat in cellular networks. In *Proc. of IEEE WoWMoM '13*, 2013.

[2] L. S. Brakmo and L. L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995.

[3] L. Budzisz, R. Stanojević, A. Schlote, F. Baker, and R. Shorten. On the fair coexistence of loss- and delay-based tcp. *IEEE/ACM Trans. Netw.*, 19(6):1811–1824, Dec. 2011.

[4] Cisco. Cisco Visual Networking Index:Forecast and Methodology 2009-2014. *White Paper*, June 2010.

[5] L. De Cicco, G. Carlucci, and S. Mascolo. Experimental investigation of the google congestion control for Real-Time flows. In *Proc of ACM SIGCOMM 2013 Workshop on Future Human-Centric Multimedia Networking (FhMN 2013)*, Hong Kong, P.R. China, Aug. 2013.

[6] 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.

[7] L. De Cicco, S. Mascolo, and V. Palmisano. Skype video congestion control: An experimental investigation. *Computer Networks*, 55(3):558–571, 2011.

[8] S. Floyd, M. Handley, J. Pahdye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. *RFC 5348*, 2008.

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

[10] 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.

[11] D. A. Hayes and G. Armitage. Revisiting tcp congestion control using delay gradients. In *Proc. of NETWORKING '11*, pages 328–341. 2011.

[12] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. A control theoretic analysis of red. In *Proc. of IEEE INFOCOM '01*, volume 3, pages 1510–1519, 2001.

[13] ITU-T. One-way transmission time. *Recommendation G.114*, May 2003.

[14] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks. In *Proc. of ACM IMC '12*, pages 329–342, 2012.

[15] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Proc. of ACM IMC '10*, pages 246–259, 2010.

[16] H. Lundin, S. Holmer, and H. Alvestrand. Google congestion control algorithm for real-time communication on the world wide web. *Draft IETF*, 2013.

[17] K. Nichols and V. Jacobson. Controlling queue delay. *Comm. of the ACM*, 55(7):42–50, July 2012.

[18] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *Proc. of IEEE HPSR '13*, July 2013.

[19] R. S. Prasad, M. Jain, and C. Dovrolis. On the effectiveness of delay-based congestion avoidance. In *Proc. of PFLDNet '04*, volume 4, 2004.

[20] H. Schulzrinne, S. Casner, S. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *RFC 3550, Standard*, 2003.

[21] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proc. of USENIX NSDI '13*, April 2013.

[22] Y. Xu, C. Yu, J. Li, and Y. Liu. Video telephony for end-consumers: measurement study of google+, ichat, and skype. In *Proc. of ACM IMC '12*, pages 371–384, 2012.

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