53rd IEEE Conference on Decision and Control
December 15-17, 2014. Los Angeles, California, USA

Modelling and Control for Web Real-Time Communication

Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo

Abstract— **Congestion control for real-time media communication over the Internet is currently being addressed in IETF and W3C bodies aiming at standardizing a set of inter-operable protocols and APIs to enable real-time communication between Web browsers. In this paper we propose a mathematical model of the congestion control algorithm for real-time flows proposed by Google. Based on this model, we design a control algorithm that provides fair coexistence of real-time flows with TCP flows.**

Index Terms— **Real-time communication, WebRTC, congestion control**

I. Introduction and Related Work

The design of an efficient congestion control is crucial to both avoid network congestion collapse and maximize the user perceived quality. The requirements of real-time multimedia traffic [7] differ significantly with respect to those of bulk data which essentially require the minimization of flow completion time [4]. In particular, such applications generate *delay-sensitive* flows, meaning that the user perceived quality is affected not only by the goodput, which is generally related to the video image quality, but also by the connection latency that should be kept low to allow a seamless and interactive communication between peers.

For this reason such flows do not employ the TCP, which implements reliability through retransmissions at the cost of delayed delivery of packets, but favor the UDP, which does not implement retransmissions. Since the UDP does not provide a congestion control algorithm, video conferencing applications have to implement this feature at the application layer. As a matter of fact, well-designed delay-sensitive applications adapt to network available bandwidth at least to some extent, such as in the case of Skype [3] and other applications [11].

The IETF working group (WG) *RTP Media Congestion Avoidance Techniques*[1] (RMCAT) has been established in 2012 with the purpose of standardizing congestion control algorithms over the RTP. The IETF RTCWeb[2] and the W3C WebRTC working groups focus respectively on the standardization of a set of protocols and set of HTML5 APIs to enable inter-operable real-time communication among Web browsers. Two end-to-end congestion control algorithms have been proposed so far within the IETF RMCAT working group: 1) the Network Assisted Dynamic Adaptation (NADA) congestion control algorithm [10] whose simulation have shown that the algorithm is able to contain queuing delays and to provide a reasonable fairness when several NADA flows share the bottleneck; 2) the Google Congestion Control (GCC) [8], a *hybrid loss-based/delay-based.* The Google's proposal is particularly interesting since it has already been implemented in Google Chrome, Firefox and Opera browsers thus having a potential user base of more than 1 billion users.

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: `g.carlucci@poliba.it`, `luca.decicco@poliba.it`, `mascolo@poliba.it`

This work has been partially supported by the project "Platform for Innovative services in the Future Internet" (PLATINO - PON01 01007) funded by Italian Ministry of Education, Universities and Research (MIUR).

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

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

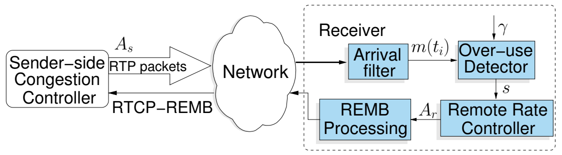


Fig. 1: Google congestion control architecture showing the receiver internal structure

In a preliminary work we have found that GCC is able to keep the queuing delay low while providing a high channel utilization in the case there is no competing traffic [1]. However, in [2] we have shown that a GCC flow can be starved by a concurrent TCP flow.

In this paper we propose a mathematical model of GCC to show that a static threshold mechanism employed by the receiver side controller is the cause of performance issues. The theoretical findings are validated through an experimental evaluation on the real system. Then, we propose an adaptive threshold mechanism at the receiver-side controller to overcome the issue of fair coexistence with TCP best effort flows.

II. The Google Congestion Control

Fig. 1 shows an essential architecture of the Google Congestion Control (GCC) algorithm. The sender employs a UDP socket to send RTP packets and receive RTCP feedback reports from the receiver. The overall congestion control algorithm is distributed at the client and at the sender. The *receiver-side controller* is a delay-based algorithm that strives to keep the queuing delay small. Towards this end, the algorithm computes the rate A_r and feeds it back to the sender to limit the sending rate computed by a *loss-based* algorithm running at the sender.

The following description is based on both the draft [8] 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[3] message, which carries A_r, arrives at the sender. The RTCP reports include the fraction of lost packets $f_l(t_k)$ computed as described in the RTP RFC. The sender uses $f_l(t_k)$ to compute the sending rate $A_s(t_k)$, measured in kbps, according to the following equation:

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

The rationale of (1) is simple: 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, 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 equation shown in the states of Fig. 3, where, t_i denotes the time the i-th group of RTP packets are depacketized to form a video frame at the receiver, $\eta \in [1.005, 1.3]$, $\alpha \in [0.8, 0.95]$, and $R(t_i)$ is the receiving rate measured in the last 500ms. Fig. 1 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 Fig. 3) in which the state σ is changed by the signal s produced by the *over-use detector* based on the output of the *arrival-time filter*. 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.

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 [8]: 1) the transmission time variation, 2) the *queuing time variation* $m(t_i)$, and 3) the network jitter $n(t_i)$. The following mathematical model is proposed in [8] :

$$d(t_i) = \frac{\Delta L(t_i)}{C(t_i)} + m(t_i) + n(t_i) \quad (3)$$

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 bottleneck link capacity, and $n(t_i)$ is the network jitter modeled as a Gaussian noise. A Kalman filter is employed to extract the state vector $\theta(t_i) = [1/C(t_i), m(t_i)]^\intercal$ from (3) that aims to steer to zero the difference $d_m(t_i) - d(t_i)$. Fig. 2 shows how a new state estimate is obtained: at each step the *innovation* or *residual* $d_m(t_i) - d(t_i)$ is multiplied with the Kalman gain $K(t_i)$ which provides the correction to the state prediction. The Kalman gain employed in the algorithm is an array of two components $k_m(t_i)$ and $k_c(t_i)$. We are interested in $k_m(t_i)$ that provides the correction to the queuing time variation $m(t_i)$ at each step according to:

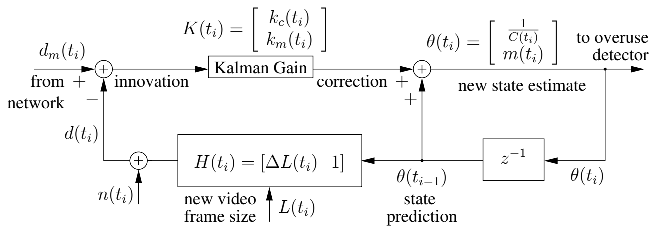
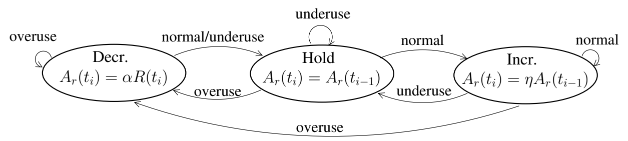


Fig. 2: State space estimation with Kalman filter



underuse
overuse
normal/underuse
normal
normal
Decr.
$A_r(t_i) = \alpha R(t_i)$
Hold
$A_r(t_i) = A_r(t_{i-1})$
Incr.
$A_r(t_i) = \eta A_r(t_{i-1})$
overuse
underuse
overuse

Fig. 3: Remote rate controller finite state machine and over-use detector signaling

$$m(t_i) = (1 - k_m(t_i)) \cdot m(t_{i-1}) + k_m(t_i) \cdot (d_m - \frac{\Delta L(t_i)}{C(t_i)}) \quad (4)$$

Eq. (4) shows that the Kalman filter in Fig. 2 in the case of the random walk model (identity state matrix) is equivalent to an EWMA filter [5] that is made of two additive terms: the first one takes into account the contribution of the previous state estimate $m(t_{i-1})$ and the second term accounts for the contribution of the measurement $d_m - \Delta L(t_i)/C_i$. The gain $k_m(t_i) \in [0, 1]$ balances this contribution: if $k_m(t_i)$ is large more weight goes to the measurement, otherwise more weight goes to the previous state estimate. The Kalman gain is updated according to the process and the measurement noise estimated [8].

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 (Fig. 3) based on $m(t_i)$ and a threshold γ. When $m(t_i) > \gamma$, 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) Remote rate controller: This block computes A_r according to the equations shown in the finite state machine of Fig. 3 driven by the signal s produced by the over-use detector. A_r is increased (*Increase* state), decreased (*Decrease* state) or kept constant (*Hold* state) depending on its state.

4) 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 $A_r(t_i) < 0.97 A_r(t_{i-1})$, i.e. when A_r has decreased more than 3%.

[3]http://tools.ietf.org/html/draft-alvestrand-rmcat-remb-03

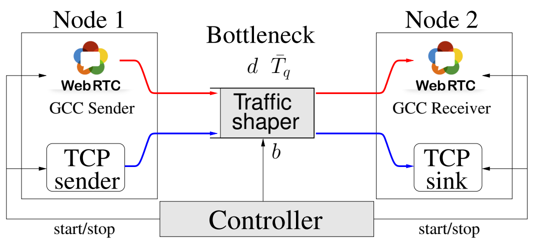


Fig. 4: Experimental testbed

III. Experimental testbed

Fig. 4 shows the testbed that we have used to evaluate the algorithm in an emulated WAN scenario. It consists of: two machines are connected through an Ethernet cable running a Chromium browser each[4] and an application to generate or receive TCP long-lived flows; another machine is the testbed *controller* orchestrating the experiments. The testbed controller undertakes the following tasks: 1) it places the WebRTC calls in an automated way; 2) it sets the one-way delay d in both directions resulting in a round-trip propagation delay $RTT_m = 2d$; 3) it sets the link capacity b and the bottleneck buffer size $\bar{T}_q$; 4) it starts the TCP flows.

The bottleneck, that emulates an Internet 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) policy[5] that creates a link capacity constraint b for the traffic received by Node 1.

We have modified the WebRTC sources to log the key variables of the congestion control algorithm. The TCP Source uses the TCP Cubic congestion control algorithm, the default version used in the Linux kernel.

For each experiment, we compute the following metrics to evaluate the performance of the algorithm: 1) *Channel Utilization:* $U = r_r/b$, where b is the known link capacity and r_r is the average received rate; 2) *Loss ratio*: $l = (\text{packets lost})/(\text{packets received})$; 3) *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); this metric is directly related to the weight of the delay-based component on the computation of the sending rate: a very small n_{dd} means that the overall behaviour of the algorithm is loss-based; 4) *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.

IV. Why an Adaptive threshold γ is necessary

In this section we show that the threshold γ, used in the *over-use detector,* must be made adaptive otherwise two issues can occur: 1) the delay-based controller could be disabled (Section IV-A) and 2) the GCC flow may be starved by a concurrent TCP flow (Section IV-B).

[4]http://code.google.com/p/chromium/
[5]http://lartc.org/

A. The single flow case

We start by considering a GCC flow accessing a bottleneck of constant bandwidth capacity b with a drop tail queue whose maximum size is equal to q_M. In the following we prove that with a static setting of γ the receiver-side controller may be inhibited, with the consequence of making the overall congestion control algorithm loss-based.

Proposition 1: Let us consider one GCC flow accessing a bottleneck with a maximum queuing time $\overline{T}_q$. The receiver-side component cannot generate over-use signals, and thus the receiver-side controller is inactive, if the following condition holds:

$$\gamma > \sqrt{2\frac{\overline{T}_q}{\tau}} + \frac{\overline{T}_q}{\tau}, \tag{5}$$

where τ is the time constant of the exponential increase phase of the sending rate computed using (1) by the loss-based algorithm.

Proof: We start by recalling that the receiver-side controller has the goal of stopping the exponential increase of the sending rate (1) before the queuing delay gets too large. Towards this end, the overuse detector of the receiver-side controller compares $m(t)$ with a static threshold γ and it triggers a decrease of the rate if $m(t) > \gamma$. Thus, if the maximum value m_M of the one way queuing delay $m(t)$ is less than γ the generation of over-use signal is inhibited. Therefore, to prove this proposition we need to show that if (5) holds, it turns out that $m_M < \gamma$.

We start by computing $m_M = \max(m(t))$. Since $m(t)$ is defined as the queuing delay variation, we can write:

$$m(t) = \dot{T}_q(t) = \frac{\dot{q}(t)}{b} \tag{6}$$

where we have used the model $T_q(t) = q(t)/b$ [6]. The queue length, measured in bytes, can be modelled as follows [9]:

$$\dot{q}(t) = \begin{cases} 0 & q(t) = 0, r(t) < b \text{ or} \\ & q(t) = q_M, r(t) > b \\ r(t) - b & \text{otherwise} \end{cases} \tag{7}$$

The fraction of lost bytes $f_l(t)$ can be computed as the ratio between the queue overflow rate $o(t)$ and the sending rate $r(t)$, i.e. $f_l(t) = o(t)/r(t)$. Since $o(t) = r(t) - b$ when $q(t) = q_M$ and $r(t) > b$, we can write:

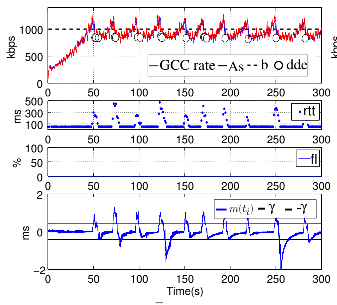
$$f_l(t) = \begin{cases} 1 - \frac{b}{r(t)} & q(t) = q_M, r(t) > b \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

The following fluid flow model for $r(t)$ can be derived from (1) and (8) after some computations:

$$\dot{r}(t) = \begin{cases} -\frac{1}{2}(r(t) - b) & f_l(t) > 0.1 \\ \frac{1}{\tau} r(t) & f_l(t) < 0.02 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

By combining (7) and (6) we obtain:

$$m(t) = \frac{\dot{q}(t)}{b} = \begin{cases} 0 & q(t) = 0, r(t) < b \text{ or} \\ & q(t) = q_M, r(t) > b \\ \frac{r(t)}{b} - 1 & \text{otherwise} \end{cases} \tag{10}$$


(a) $\overline{T}q$ = 700ms

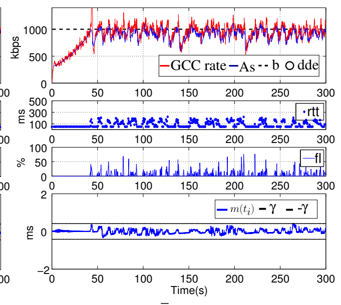

(b) $\overline{T}q$ = 150ms

Fig. 5: GCC dynamics in isolation in the case of a bottleneck bandwidth $b = 1000$kbps

Since we are interested in finding the maximum of $m(t)$, we consider the exponential increase phase of $r(t)$ that is valid when $f_l(t) < 0.02$ according to (9). Let $t_0 = 0$ denote the time instant for which $r(t_0) = b$, i.e. the instant after which the queue starts to build up, and let us analyze the dynamics of $m(t)$ for $t > t_0$. Under these assumptions it turns out that $r(t) = r(t_0)\exp(t/\tau) = b \cdot \exp(t/\tau)$ and $m(t) = r(t)/b - 1$ giving:

$$m(t) = \exp(t/\tau) - 1. \tag{11}$$

Eq. (11) is a monotonically increasing function until the point $q(t) = q_M$, i.e. when the queue is full and packets start to get dropped. After that point the rate starts to decrease according to the exponential decrease phase described in (9). Thus, the maximum value of $m(t)$ is equal to $m(t_M)$ where t_M is the time instant at which the queue becomes full, i.e. $q(t_M) = q_M$. By considering (7) and integrating between t_0 and t_M we obtain:

$$q(t_M) = q_M = \int_{t_0}^{t_M} (r(\xi) - b) d\xi = b\tau(\exp(t_M/\tau) - 1) - bt_M. \tag{12}$$

By computing the second order McLaurin expansion for the exponential and substituting it in (12) we obtain the following approximation:

$$t_M \simeq \sqrt{\frac{2q_M}{\alpha b}}.$$

Thus, the maximum queuing delay is:

$$m_M = m(t_M) \simeq \frac{1}{\tau}\left(\sqrt{\frac{2\tau q_M}{b}} + \frac{q_M}{b}\right) = \sqrt{2\frac{\overline{T}_q}{\tau}} + \frac{\overline{T}_q}{\tau}. \tag{13}$$

The proposition is proved by observing that if the condition (5) holds, it turns out that $m_M < \gamma$. ■

Fig. 5 compares the GCC rate, the RTT, and the losses in the case of a single GCC flow accessing a 1Mbps bottleneck with the default static value of γ used in the Google Chromium browser. Fig. 5(a) shows the case of a drop-tail queue with a maximum queuing time $\overline{T}_q = 700$ms. Every time the sending rate reaches the link capacity b, the one way delay variation $m(t)$ gets above the threshold γ triggering the finite state machine (Fig. 3) in decrease state. In Fig. 5(a) the white dots mark the time instants where the rate A_r computed by the delay-based controller is less than the rate A_s computed by the loss-based controller which we define as "delay-based decrease events" according to (2). It is clear that, following a delay-based decrease event, both the queuing delay and $m(t)$ quickly decrease to zero. Fig. 5(a) also shows that the controller is able to avoid packet losses, i.e. $f_l(t) = 0$ throughout the whole duration of the experiment.

On the other hand, Fig. 5(b) shows that when $\overline{T}_q = 150$ms the delay-based controller run at the receiver is disabled and it is not able to avoid losses and control the queuing delay. This is due to the fact that $|m(t)| < \gamma$ for the whole duration of the experiment, i.e. the value of γ is too large for $\overline{T}_q = 150$ms.

B. The case of a concurrent TCP flow

In this section we consider the case of a GCC flow with a concurrent TCP flow and we show that a static setting of the threshold γ may lead to the starvation of the GCC flow.

In this scenario, the queuing delay variation can be expressed as the sum of two components:

$$m(t) = m_{GCC}(t) + m_{TCP}(t) = \frac{r_{GCC}(t) + r_{TCP}(t)}{b} - 1 \tag{14}$$

where $m_{GCC}(t)$ and $m_{TCP}(t)$ are the queuing delay variations of the GCC and the TCP flow respectively and $r_{GCC}(t)$ and $r_{TCP}(t)$ are the sending rates of the GCC and TCP flow respectively.

In the following we show that the maximum queuing delay variation $m_{TCP,M}$ due to a TCP flow can be much larger than that of a GCC flow. In particular, by using similar arguments employed to derive (13) we obtain:

$$m_{TCP,M} = \frac{\max(\dot{q}(t))}{b} = \frac{\max(r_{TCP}(t)) - b}{b}. \tag{15}$$

A well-known approximation of the TCP throughput is $r(t) = w(t)/RTT(t)$ [6], where $w(t)$ is the congestion window of the TCP flow and $RTT(t)$ is the round trip time. The maximum value that the congestion window can assume is the queue size plus the inflight bytes $b \cdot RTT_{min}$ [9], i.e. $\max(w(t)) = q_M + b \cdot RTT_{min}$, whereas the minimum value of $RTT(t)$ is the round trip propagation RTT_{min}. It turns out that $\max(r_{TCP}(t)) = q_M/RTT_{min} + b$ which yields to:

$$m_{TCP,M} = \frac{q_M}{RTT_{min} \cdot b} = \frac{\overline{T}_q}{RTT_{min}}, \tag{16}$$

The ratio between $m_{TCP,M}$ and $m_{GCC,M}$ is given by:

$$\frac{m_{TCP,M}}{m_{GCC,M}} = \frac{\frac{\overline{T}_q}{RTT_{min}}}{\sqrt{2\frac{\overline{T}_q}{\tau}} + \frac{\overline{T}_q}{\tau}} \tag{17}$$

as a function of the queuing time $\overline{T}_q$ for RTT_{min} and τ, i.e. the time constant of the GCC flow exponential increase phase. It is clear that when $\overline{T}_q$ increases the ratio monotonically increases. Since in this case $m(t)$ contains the component $m_{TCP}(t)$ due to the TCP flow, if $m_{TCP}(t) \gg$

TABLE I: Experimental testbed parameters

Parameter	Values	
T_q - Max queuing time (ms)	150, 350, 700	
RTT_m (ms)	50	
b - Capacity (Mbps)	Single flow	0.5, 1.0, 1.5, 2.0
	Concurrent TCP	1.0, 2.0, 3.0

$m_{GCC}(t)$ the GCC flow will decrease A_r not due to the self-inflicted delay, but due to the TCP flow. This explains why, with a static threshold γ, with larger queue sizes the TCP flow starves the GCC flow.

Fig. 7(b-left) shows the dynamics of the GCC flow when a TCP flow joins into the bottleneck. When the TCP flow starts at $t = 80$s, $m(t)$ starts to oscillate over the threshold γ having the effect of generating a large number of *overuse* signals. Consequently, the remote rate controller FSM is driven to the *decrease* mode reducing the value of A_r according to the finite state machine in Fig. 3. 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 controller.

V. The Design of the Adaptive Threshold

In this Section we propose a control law to dynamically set the threshold γ used by the *overuse detector* (see Fig. 1).

Eq. (14) show that the dynamics of $m(t_i)$ is due to both the GCC flow and the TCP flow. Thus, the basic idea is to have a dynamic threshold $\gamma(t_i)$ that tracks the queuing delay variation that is caused by both the GCC flow and the TCP flow.

We propose the following adaptive threshold:

$$\gamma(t_i) = \gamma(t_{i-1}) + \Delta T \cdot K(t_i)(|m(t_i)| - \gamma(t_{i-1})) \quad (18)$$

where $\Delta T = t_i - t_{i-1}$, i.e. the time elapsed between the reception of two consecutive frames, and the gain $K(t)$ is defined as follows:

$$K(t_i) = \begin{cases} K_d & |m(t_i)| < \gamma(t_{i-1}) \\ K_u & \text{otherwise} \end{cases}$$

with $K_d < K_u$. By using (18), the delay-based controller at the receiver compares the one way delay variation $m(t_i)$ with a threshold $\gamma(t_i)$ that is a low-pass filtered version of $|m(t_i)|$. In particular, when $m(t_i)$ overshoots $\gamma(t_i)$ the delay-based controller reduces m and the threshold γ follows m with a slower time constant so that m keeps below γ and does not trigger several delay-based decrease events. This keeps until m again overshoots γ and the delay-based algorithm again reduces m. It is worth noting that when $\gamma > m$, γ follows m with a shorter time constant so that fewer decrease events are required to make $m < \gamma$.

VI. Experimental Results

Table I summarizes the bottleneck parameters employed in the experimental evaluation. For each combination of the parameter in Table I, we have run three experiments and compared the metrics defined in Section III between GCC and AT-GCC by averaging over the three experimental results. The proposed controller (18) has been tuned by using $K_u = 0.021$ and $K_d = 0.0006$.

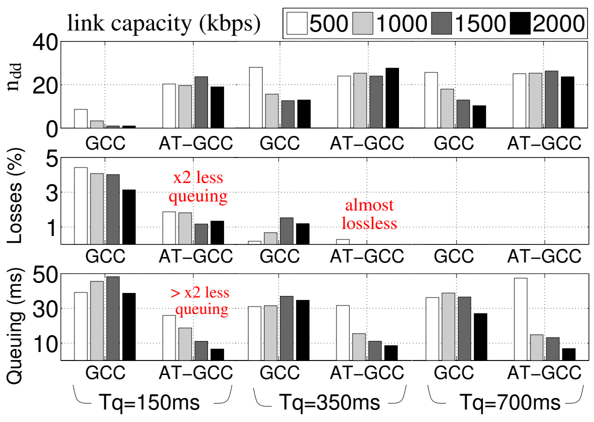


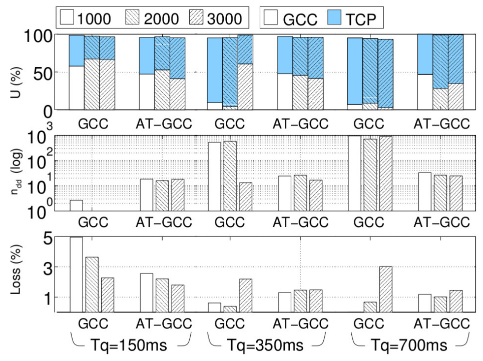
Fig. 6: A single GCC or AT-GCC flow over a bottleneck of constant capacity $b \in \{500, 1000, 1500, 2000\}$kbps

A. A single GCC or AT-GCC flow

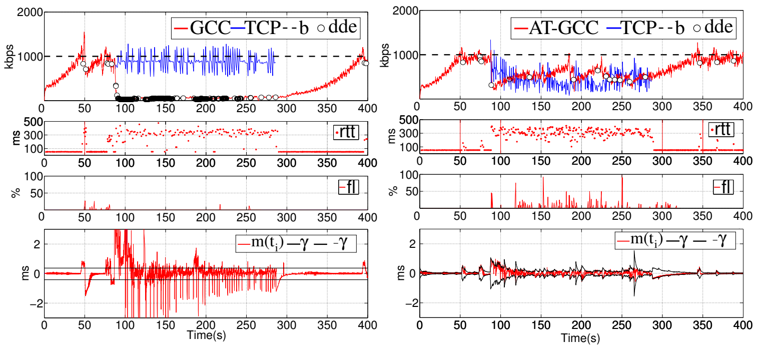
In the following we compare the proposed AT-GCC algorithm with the GCC algorithm when they access a bottleneck in isolation. Fig. 6 shows the results: each bar represents the metrics obtained for a certain link capacity and the bars are grouped for the maximum queuing time. Let us analyze the performance of the GCC algorithm in the case of $\bar{T}_q = 150$ms: since the number of delay-based decrease events n_{dd} is almost zero for $b > 0.5$Mbps, it results that the delay-based receiver controller is inhibited if $b \in \{1, 1.5, 2\}$Mbps. The reason is that the static value of γ turns out to be too large in the case of a small queue (see Section IV-A). On the other hand, when the proposed controller is used (18) the delay-based controller is always kept active and n_{dd} is roughly equal to 20 for each considered value of b. As a consequence, when AT-GCC is used the loss percentage is halved and the queuing time is reduced by a factor greater than 2. In the case of $\bar{T}_q = 350$ms AT-GCC still outperforms GCC in terms of queuing time and loss percentage for $b > 500$kbps even though GCC exhibits better results *wrt* the case of $\bar{T}_q = 150$ms. This is due to the fact that, when $\bar{T}_q$ is larger, the receiver controller can detect variations of $m(t_i)$ (see (13)). Finally, in the case of $\bar{T}_q = 700$ms, both the algorithms produce no losses even though AT-GCC is able to reduce the queuing time for $b > 500$kbps due to its faster reaction at the onset of an increase of $m(t_i)$. The average channel utilization is higher than 90% and it is not shown due to space limitation. Finally, it is interesting to notice that the proposed adaptive threshold mechanism (18) stabilizes n_{dd} to roughly 20 in every considered case.

B. One AT-GCC or one GCC flow in the presence of a concurrent TCP flow

We now consider the case of one AT-GCC or one GCC flow when sharing the bottleneck link with one TCP flow. Fig. 7(a) shows the results grouped using the same criterion of Fig. 6. Let us consider a buffer corresponding to a



(a) Channel utilization, n_{dd}, and loss percentage in the case of $b \in \{1, 2, 3\}$Mbps and $\overline{T}_q \in \{150, 350, 700\}$ms



(b) Dynamics of GCC (left) and AT-GCC (right) when coexisting with a TCP flow in the case of $\overline{T}_q = 350$ms and $b = 1000$kbps

Fig. 7: One AT-GCC or one GCC flow with a concurrent TCP flow over a bottleneck of constant capacity

maximum queuing delay of $\bar{T}_q = 150$ms in the case of GCC: in this situation the bandwidth share obtained by GCC is higher than that of the TCP flow. This is due the fact that, with such a small queue, it results that $|m(t_i)| = |m_{GCC}(t_i) + m_{TCP}(t_i)| < \gamma$ even in the presence of TCP. This is confirmed by the fact that the measured number of delay-based decrease events n_{dd} is zero. Since the receiver-side controller is inhibited, the GCC flow is driven only by the loss-based controller (1) and it results to be more aggressive than the TCP. On the other hand, when the proposed adaptive threshold (18) is used, the delay-based receiver controller is kept active and we have measured $n_{dd} \simeq 25$ regardless of the value of b. Moreover, the proposed controller makes the AT-GCC sending rate less aggressive so that the bottleneck is fairly shared with TCP.

At $\overline{T}_q = 350$ms the AT-GCC continues to fairly share the bandwidth with the TCP flow. However, the GCC flow is starved by the concurrent TCP flow when the bottleneck capacity b is less than 3000kbps, confirming the analysis given in Section VI-B; when $b = 3000$kbps, $|m(t_i)| < \gamma$ and as a result no overuse signals will be generating having the effect of disabling the receiver-side controller.

Finally, at $\bar{T}_q = 700$ms the AT-GCC flow continues to reasonably share the bandwidth with the TCP flow while the GCC flows is starved for each value of the link capacity considered, as a consequence of (17). Similarly to the previous case, we notice that the proposed controller (18) stabilizes n_{dd} to roughly 25 in every considered case.

Fig. 7(b) shows the dynamics of one GCC or one AT-GCC flow under the same the bottleneck conditions in the presence of a concurrent TCP flow. The behavior of GCC has been already explained in Section VI-B. Let us consider the case of AT-GCC: as soon as the TCP flow joins the bottleneck, an increase of $m(t_i)$ is detected. This causes the generation of an overuse signal and, as a consequence, a delay-decrease event which reduces the GCC sending rate. Then, due to the control law (18) the threshold $\gamma(t_i)$ is increased, in this case proportionally to $m_{TCP}(t_i)$, avoiding the generation of many consecutive overuse signals which in the case of GCC cause the starvation of the GCC flow. This makes AT-GCC to operate more often in loss based mode as it can be inferred by looking at the fraction loss $f_l(t_k)$ shown in Fig. 7(b-right).

VII. Conclusions

In this paper we have proposed a mathematical analysis of the Google Congestion Control (GCC) algorithm for real time-flows which proves that it is necessary to use an adaptive threshold γ at the delay-based controller running at the receiver. Starting from this analysis we have proposed a control algorithm to dynamically set the threshold γ. The proposed controller, named AT-GCC, has been implemented in the Chromium browser and experimentally compared with GCC. The results show that AT-GCC is able to fairly share the bottleneck with a TCP concurrent flow regardless of the size of the bottleneck queue. Moreover, in the case of a single flow accessing a bottleneck, AT-GCC is able reduce the queuing delay up to a factor of $1/2$ *wrt* the queuing obtained with GCC. In future works we will simplify the overall congestion control algorithm to make it completely mathematically tractable.

References

[1] 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.

[2] L. De Cicco, G. Carlucci, and S. Mascolo. Understanding the Dynamic Behaviour of the WebRTC Google Congestion Control. In *Proc. of Packet Video Workshop 2013*, San Jose, CA, USA, Dec. 2013.

[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] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[5] J. Durbin and S. J. Koopman. *Time series analysis by state space methods*. Oxford University Press, 2012.

[6] 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.

[7] R. Jesup. Congestion Control Requirements for RMCAT. *IETF Draft*, 2013.

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

[9] S. Mascolo. Modeling the Internet congestion control using a Smith controller with input shaping. *Control engineering practice*, 14(4):425–435, 2006.

[10] Z. Xiaoqing and R. Pan. NADA: A Unified Congestion Control Scheme for Low-Latency Interactive Video. In *In Proc. of Packet Video Workshop 2013*, San Jose, CA, USA, Dec. 2013.

[11] Y. Xu, C. Yu, J. Li, and Y. Liu. Video Telephony for End-Consumers: Measurement Study of Google+, iChat, and Skype. *IEEE/ACM Transactions on Networking*, 22(3):826–839, June 2014.