Analysis and Design of the Google Congestion Control for Web Real-time Communication (WebRTC)

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

Luca De Cicco
Télécom SudParis, France
luca.de_cicco@telecom-sudparis.eu

Stefan Holmer
Google, Sweden
holmer@google.com

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

ABSTRACT

Video conferencing applications require low latency and high bandwidth. Standard TCP is not suitable for video conferencing since its reliability and in order delivery mechanisms induce large latency. Recently the idea of using the delay gradient to infer congestion is appearing again and is gaining momentum. In this paper we present an algorithm that is based on estimating through a Kalman filter the end-to-end one way delay variation which is experienced by packets traveling from a sender to a destination. This estimate is compared to an adaptive threshold to dynamically throttle the sending rate. The control algorithm has been implemented over the RTP/RTCP protocol and is currently used in Google Hangouts and in the Chrome WebRTC stack. Experiments have been carried out to evaluate the algorithm performance in the case of variable link capacity, presence of heterogeneous or homogeneous concurrent traffic, and backward path traffic.

CCS Concepts

•Networks → Network protocol design; •Information systems → *Web conferencing;*

Keywords

Real-time communication, congestion control, WebRTC

1. INTRODUCTION

Video conferencing QoE is not only affected by the goodput, which is generally related to the video image quality, but also by the connection latency that should be kept as low as possible to allow a seamless communication [4]. This is the reason why flows generated by such applications are considered *delay sensitive*. It is well-known that the TCP, which is still the most used transport protocol over the Internet, is not suitable to deliver traffic generated by real-time applications. The main reason is that the TCP favors reliability and in order delivery through retransmissions over packet delivery timeliness. Although some works have investigated the implications of employing TCP for VoIP traffic [5, 29], the case of video conferencing, – which indeed entails much higher data rates – has never been studied. Thus, video conferencing applications have traditionally used UDP, leaving congestion control to the application layer. It has been shown that several well-known video conferencing applications adapt to network available bandwidth at least to some extent, such as in the case of Skype [8] and other applications [33].

This research area is attracting a renewed attention due to the WebRTC initiative that aims at standardizing an inter-operable and efficient framework for real-time communication in Web browsers using RTP [1]. In this paper we present the *Google Congestion Control* (GCC) [15], an algorithm compliant with the WebRTC framework, that it is implemented in the Chrome Browsers and used in Google Hangouts. The algorithm has been designed to work with RTP/RTCP protocols and is based on the idea of using the delay gradient to infer congestion. For this purpose, a Kalman filter is designed to produce an estimate of the queuing delay gradient which is compared to an adaptive threshold in order to detect congestion.

The remainder of the paper is organized as follows: Section 2 provides a review of the relevant literature on congestion control for delay-sensitive flows; Section 3 describes the proposed algorithm; Section 4 motivates the algorithm design choices employed to detect network congestion; Section 5 presents the experimental testbed and the employed metrics; Section 6 shows the experimental results and finally Section 7 concludes the paper.

2. RELATED WORK

Traditional loss-based TCP is not suitable for video conferencing traffic since its congestion control continuously probes for network available bandwidth introducing periodic cycles during which network queues are first filled and then drained. These queue oscillations induce a time-varying stochastic delay component that adds to the propagation time and make delay-sensitive communications problematic. The idea that network delay was correlated to network congestion has been proposed since 1989 [16]. However, several issues related to delay measurements in delay-based algorithms have been exposed [23], especially in the case of wireless environments [13] and when the bottleneck is shared with loss-based flows [12]. This section provides a review of the relevant literature on congestion control for delay-sensitive flows in wide area networks.

MMSys'16, May 10-13, 2016, Klagenfurt, Austria
© 2016 ACM. ISBN 978-1-4503-4297-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2910017.2910605

The use of RTT to infer congestion. The first efforts focusing on the reduction of the queuing delay were set in the TCP congestion control research domain and, consequently, many algorithms for real-time traffic are rooted on this literature. The first congestion control algorithm specifically designed to contain the end-to-end latency by employing delay measurements is the seminal work by Jain which dates back to 1989 [16]. Since then, several delay-based TCP congestion control variants have been proposed, such as TCP Vegas [3]. Typically, delay-based algorithms require the knowledge of the end-to-end RTT statistics to establish appropriate delay thresholds used to infer congestion, such as in the case of TCP Vegas [3] and TCP FAST [30]. It has been shown that when the RTT is used as a congestion metric a low channel utilization may be obtained in the presence of reverse traffic or when competing with loss-based flows [12]. It is worth mentioning that the problem of reverse traffic is crucial in the context of video conferencing, since video flows are sent in both directions.

The use of one way delay to infer congestion. Another class of algorithms advocates the use of one way delay measurements to rule out the sensitivity to the reverse path congestion. Examples are LEDBAT (over UDP) [26] and TCP-LP [18]. In particular, LEDBAT [26] increases its congestion window at a rate that is proportional to the distance between the measured one way delay and a fixed delay target τ. It has been shown that LEDBAT is affected by the so called *"latecomer effect"*: when two flows share the same bottleneck the second flow typically starves the first one [7].

The use of delay-gradient to infer congestion. The idea of employing RTT gradient to infer congestion has been recently employed to overcome the aforementioned *"latecomer effect"*. Some examples are CDG [14] and Verus [34]. CDG [14] has been designed with the aim of coexisting with loss-based flows while keeping end-to-end delay low. Verus [34] has been specifically designed for cellular networks where sudden link capacity variations make the congestion control design challenging. Recently, it has been shown that accurate delay gradient measurement are achievable in data center networks by employing NIC hardware timestamps [19].

Other approaches. Among recently proposed congestion control algorithms, which do not explicitly infer congestion by employing network delay measurement, we cite Sprout [32], Remy [31], and FBRA [20]. Sprout [32] takes a stochastic approach to contain delays while maximizing the throughput; the paper shows that the algorithm outperforms applications such as Skype, Facetime, and Google Hangouts in a single flow scenario, however, the performance in the case of multiple flows sharing a bottleneck has not been evaluated yet. Remy [31] is a framework to generate congestion control algorithms. By defining a utilization function based on the users requirements Remy employs the a-priori knowledge of the network to train a machine to learn congestion control schemes. FBRA [20] proposes a FEC-based congestion control algorithm: the rationale is to probe the available bandwidth through FEC packets and in the case of losses due to congestion the redundant packets help in recovering the lost packets.

Design for RTP/RTCP. The focus of this paper is the design of a congestion control algorithm for RTP/RTCP over UDP. Even though this research area has been active in the past [27], today it is attracting a renewed attention due to the WebRTC W3C and IETF joint initiative which aims at enabling inter-operable real-time communication among Web browsers [1]. Three end-to-end algorithms have been proposed within the IETF RMCAT working group: 1) Network Assisted Dynamic Adaptation (NADA) [35], is a loss/delay-based algorithm that relies on one way delay measurements; 2) Self-Clocked Rate Adaptation for Multimedia (SCREAM) [17] which inherits some ideas from LEDBAT; 3) Google Congestion Control (GCC) [15] that will be described in the following.

3. BACKGROUND

This section provides background information on the algorithm presented in the paper. We start by defining the queuing delay gradient and then we show how it is employed in the proposed algorithm to detect congestion.

3.1 Queuing delay gradient

The (one way) queuing delay gradient is defined as the derivative of the queuing delay $T_q(t)$ which can be modeled as $T_q(t) = q(t)/C$, where C is the bottleneck link capacity and $q(t)$ is the queue length measured in bits. Thus, the queuing delay gradient $\dot{T}_q(t)$ is equal to:

$$\dot{T}_q(t) = \frac{\dot{q}(t)}{C} \tag{1}$$

The derivative of the queue length $\dot{q}(t)$, can be modeled as follows:

$$\dot{q}(t) = \begin{cases} r(t) - C & 0 \leq q(t) \leq q_M \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $r(t)$ is the queue filling rate measured in bit per second and q_M is the maximum queue length. By definition, $\dot{q}(t)$ is equal to zero, and thus $\dot{T}_q(t) = 0$, when the queue length $q(t)$ stays constant. This can happen in three different conditions: 1) when the queue is zero due to *channel underutilization,* i.e., when the filling rate $r(t)$ is below the link capacity C and the queue eventually gets empty; 2) when the queue is full due to *persistent congestion,* i.e. when the filling rate $r(t)$ exceeds the link capacity C; 3) when the input rate $r(t)$ exactly matches C. In the third case, i.e. $r(t) = C$, the queue stays constant to a value $\overline{q} \in [0, q_M]$. Such a situation, which in [22] is defined as *standing queue,* is regarded as undesirable since it steadily delays the incoming traffic. The proposed algorithm aims at keeping the queue as small as possible without underutilizing the link. To achieve this goal the algorithm has to probe for the available bandwidth increasing its sending rate until a positive queuing delay variation is detected. At this point the sending rate is reduced. For this reason we argue that some queuing delay has to be induced in order to run the delay-based congestion control.

3.2 Algorithm architecture

Figure 1 shows the overall control architecture employed by 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 algorithm has two components: 1) a *delay-based* controller, placed at the receiver, which computes a rate A_r that is fed back to the sender with the aim of containing the delay; 2) a *loss-based* controller, placed at the sender, which computes the target sending bitrate A_s that cannot exceed A_r.

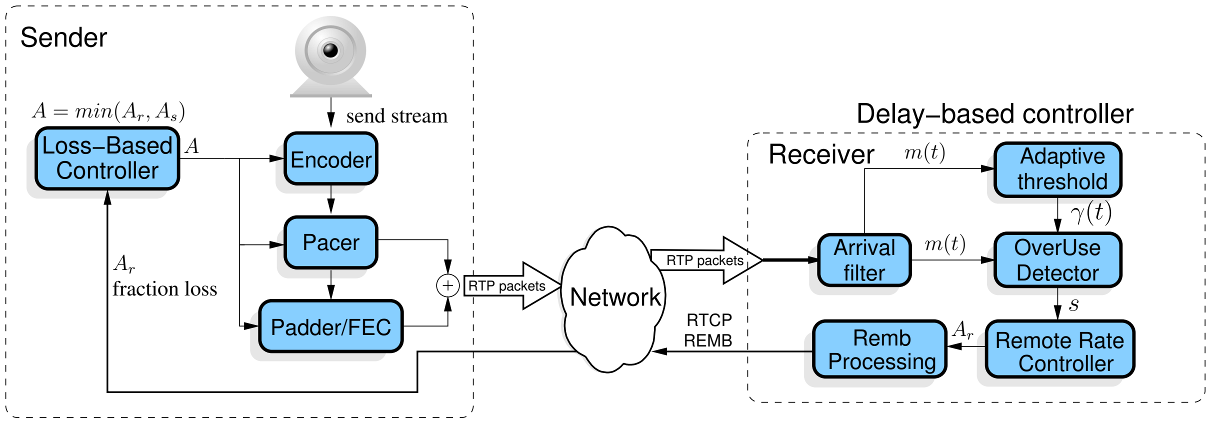


Figure 1: Google Congestion Control architecture

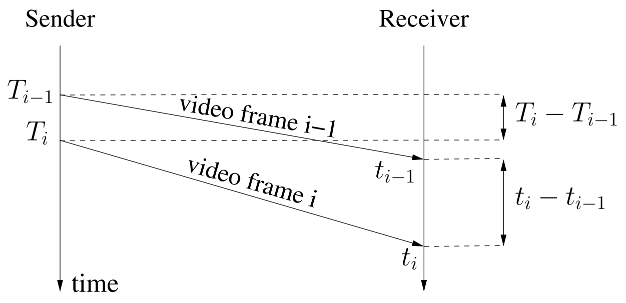


Figure 2: One way delay gradient measurement

In the following we provide a description of the algorithm. Technical details can also be found in the IETF draft [15] and in the WebRTC code repository[1] of the Chromium browser.

3.3 The delay-based controller

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

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

where t_i denotes the time the i-th video frame is received, $\eta = 1.05$, $\alpha = 0.85$, and $R_r(t_i)$ is the receiving rate measured in the last 500ms. Figure 1 shows a block diagram of the delay-based controller that is made of several components described in the following.

The *remote rate controller* is a finite state machine (see Figure 4) in which the state σ of (3) is changed by the signal s produced by the *over-use detector* based on the output $m(t_i)$ of the *arrival-time filter*. The *adaptive threshold* block dynamically sets the threshold $\gamma(t_i)$ used by the *over-use detector*. The *REMB Processing* decides when to send a REMB[2] message based on the value of the rate A_r. Finally, it is important to notice that $A_r(t_i)$ is upper bounded by $1.5R_r(t_i)$.

In the following we provide a detailed description of each block.

[1]https://chromium.googlesource.com/external/webrtc/+/master/webrtc/

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

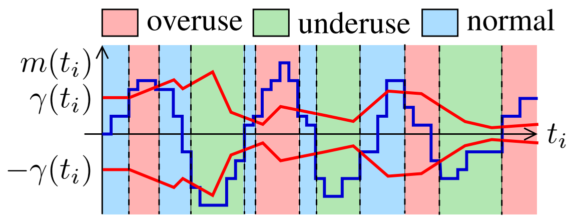


Figure 3: Over-use detector signaling

The arrival-time filter. The goal of this block is to produce an estimate $m(t_i)$ of the one way delay gradient. For this purpose, we employ a Kalman filter that estimates $m(t_i)$ based on the *measured one way delay gradient* $d_m(t_i)$ which is computed as follows (see Figure 2):

$$d_m(t_i) = (t_i - t_{i-1}) - (T_i - T_{i-1}) \tag{4}$$

where T_i is the time at which the first packet of the i-th video frame has been sent and t_i is the time at which the last packet that forms the video frame has been received.

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 $\gamma(t_i)$. Figure 3 shows how s is generated: when $m(t_i) > \gamma(t_i)$, the algorithm starts to track the time spent in this condition and if it is greater than 100ms the *overuse* signal is generated. On the other hand, if $m(t_i)$ decreases below $-\gamma(t_i)$, the *underuse* signal is generated, whereas the *normal* signal is triggered when $-\gamma(t_i) \leq m(t_i) \leq \gamma(t_i)$.

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 4. The aim of the finite state machine is to minimize the queuing delays in the buffers along the end-to-end path. The rationale is the following: when the bottleneck buffers start to build-up, the estimated one way delay gradient $m(t_i)$ becomes positive. The overuse detector detects this variation and triggers an *overuse* signal, which drives the machine into the “decrease” state. As a result, the sending rate is reduced and the bottleneck buffer starts to be drained, up to the point that the estimated one way delay gradient $m(t_i)$ becomes negative. An

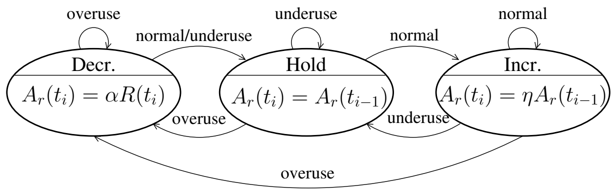

overuse
normal/underuse
underuse
normal
normal
Decr.
$A_r(t_i) = \alpha R(t_i)$
Hold
$A_r(t_i) = A_r(t_{i-1})$
Incr.
$A_r(t_i) = \eta A_r(t_{i-1})$
overuse
underuse
overuse

Figure 4: Remote rate controller finite state machine

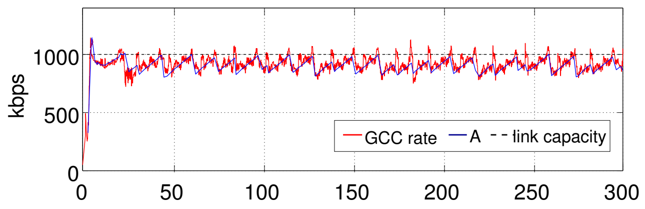


Figure 5: Sending Rate Actuation

underuse signal is then triggered, which drives the machine into the "hold" state. The machine remains in the "hold" state until the bottleneck buffer is emptied. When this occurs, $m(t_i)$ approaches 0 and the overuse detector generates a *normal* signal, which drives the machine into the "increase" state.

Adaptive threshold. The aim of this block, shown in Figure 1, is to adapt the algorithm sensitivity to delay variations based on the network conditions. In particular, this block avoids two issues: 1) the delay-based controller inhibition when the size of bottleneck queue along the path is not sufficiently large and 2) the starvation of GCC flows in the presence of concurrent loss-based TCP traffic. The input of this block is the estimated delay gradient $m(t_i)$; the output is the dynamic value of the threshold $\gamma(t_i)$ which is fed to the *over-use detector*. More details on this block are provided in Section 4.2.

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%.

3.4 The loss-based controller

The loss-based controller complements the delay-based controller in the case losses are measured. The algorithm acts every time t_k the k-th RTCP report message or a REMB message carrying A_r is received by the sender. The REMB format is an extension of the RTCP protocol that is being discussed at the IETF. 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 $A_s(t_k)$ according to the following equation:

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0.5f_l(t_k)) & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1})) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases} \quad (5)$$

The rationale of (5) 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.

3.5 Sending rate actuation

In this Section we describe the mechanism employed to actuate the sending rate computed by the two controllers. Figure 1 shows that the sender sets the target sending bitrate A equal to the minimum between A_r and A_s. The target bitrate A is fed both to the Pacer and to the Encoder. The Encoder strives to produce a bitrate as close to this target as possible. However, the encoder cannot change the rate as frequently as the Pacer's rate to avoid video quality flickering which is known to have an adverse impact on the QoE [21]. For this reason the encoder may not be able to produce a rate precisely matching the target A. In the case the encoder produces a rate higher than the target A, the Pacer is allowed to drain its queue at a rate $f \cdot A$, where f is the Pacing factor equal to 1.5. In this way the pacer can quickly empty its queue avoiding queuing delays at the sender. On the other hand, whenever the Pacer produces a rate lower than the target A, padding or FEC might be added. As a result of this mechanism, an average sending bitrate equal to A is produced. Figure 5 shows the case of one GCC flow over a 1Mbps link. The figure shows that the actual rate generate by the GCC sending engine nicely tracks the target bitrate A set by the congestion control algorithm.

4. CONGESTION DETECTION

In this Section, we motivate the design choices adopted to infer network congestion. In particular, we cover two crucial aspects: 1) the way the estimate $m(t_i)$ is produced by means of a Kalman filter and 2) the design of the adaptive threshold used to detect congestion.

4.1 Kalman filter

4.1.1 Design

We assume that the delay gradient is affected by a jitter noise. To filter out this noise and produce the estimate $m(t)$, a Kalman filter is employed. The design of the Kalman filter is based on a system model which consists of: 1) a dynamical state equation which describes the dynamics of the state vector, and 2) an a static output equation describing the relationship between the state vector and the measurement [11].

Let us consider the following model of the state vector[3] which is composed by two components:

$$\boldsymbol{\theta}(t_i) = \begin{bmatrix} \frac{1}{C(t_i)} \\ \mu(t_i) \end{bmatrix} \quad (6)$$

where $1/C(t_i)$ is the inverse of the bottleneck link capacity and $\mu(t)$ is the one way queuing delay gradient; t_i denotes the time the i-th video frame is received. The system model is given by:

$$\begin{cases} \boldsymbol{\theta}(t_{i+1}) = \boldsymbol{\theta}(t_i) + \boldsymbol{w}(t_i) & \text{(state equation)} \\ d_m(t_i) = \frac{\Delta L(t_i)}{C(t_i)} + \mu(t_i) + n(t_i) & \text{(output equation)} \end{cases} \quad (7)$$

[3]Throughout the paper vectors are represented with boldface fonts to differentiate them from scalar variables.

The state evolution at time t_{i+1} is given by its state at time t_i plus a Gaussian state noise $\boldsymbol{w}(t_i)$. The model of the measured one way delay variation $d_m(t_i)$ is given by the sum of three components: 1) the transmission time variation, given by the ratio between the size variation of two consecutive video frames $\Delta L(t_i)$ and the bottleneck link capacity $C(t_i)$, 2) the *queuing delay gradient* $\mu(t_i)$, and 3) $n(t_i)$ modeled as a Gaussian noise. However, in the following we show that the system described by (7) may lack the observability property of dynamical systems [11]. A dynamical system is observable if there exists a finite N, such that for every initial state $\boldsymbol{\theta}(t_0)$, N measurements $\{d(t_0), d(t_1), ..., d(t_{N-1})\}$ uniquely distinguish the initial state $\boldsymbol{\theta}(t_0)$ [11]. We can prove the following proposition:

PROPOSITION 1. *The system described by eq. (7), assumed as noise-free, is not observable if the size variation $\Delta L(t_i)$ of two consecutive samples is constant.*

PROOF. We consider two consecutive measurements $N = 2$, since we have a two-dimensional system:

$$\begin{cases} d(t_0) = \frac{\Delta L(t_0)}{C(t_0)} + \mu(t_0) \\ d(t_1) = \frac{\Delta L(t_1)}{C(t_1)} + \mu(t_1) = \frac{\Delta L(t_1)}{C(t_0)} + \mu(t_0) \end{cases} \quad (8)$$

From (7) we have that $\boldsymbol{\theta}(t_1) = \boldsymbol{\theta}(t_0)$ in a noise-free system. The solution $\boldsymbol{\theta}(t_0)$ to (8) is unique if and only if the two equations are linearly independent or, in other terms, if the observability matrix:

$$\mathcal{O} = \begin{bmatrix} \Delta L(t_0) & 1 \\ \Delta L(t_1) & 1 \end{bmatrix} \quad (9)$$

has rank($\mathcal{O}$) equal to 2 [11]. This condition does not hold if $\Delta L(t_0) = \Delta L(t_1)$, which concludes the proof. □

Thus, according to Proposition 1, the system is observable if and only if $\Delta L(t_i)$ changes at each step. Unfortunately, this condition does not hold in general and, in such cases, the state estimation $\boldsymbol{\theta}(t_i)$ is indeterminate.

This is the reason why we propose to simplify the system as follows:

$$\begin{cases} \overline{m}(t_{i+1}) = \overline{m}(t_i) + w(t_i) & \text{(state equation)} \\ d_m(t_i) = \overline{m}(t_i) + n(t_i) & \text{(output equation)} \end{cases} \quad (10)$$

where $\overline{m}(t)$ is now the only state variable, which is the model of the one way delay gradient. This choice can also be related to the fact that we can measure the one way delay gradient as a whole, but we cannot distinguish the part that is due to the queuing (i.e. $\mu(t_i)$) from the part that is due to transmission and processing time. Moreover, we argue that the contribution given by transmission time variation $\Delta L(t_i)/C(t_i)$ can be considered negligible with respect to the queuing delay gradient $\mu(t_i)$. Now, since system (10) is scalar, observability is always guaranteed.

The state noise $w(t_i)$ is modeled as a stationary Gaussian process with zero mean and variance $Q(t_i) = E[w(t_i)^2]$. Similarly to $w(t_i)$, the measurement noise $n(t_i)$ – which takes into account the network jitter – is considered as a stationary Gaussian process with zero mean and variance $\sigma_n^2(t_i) = E[n(t_i)^2]$.

Both the state noise variance and measurement noise variance are important parameters which are required to be tuned appropriately. This issue will be addressed in Section 4.1.2.

Figure 6 shows how a new state estimate $m(t_i)$ is obtained: at each step the *innovation* or *residual* $z(t_i)$ is multiplied by the

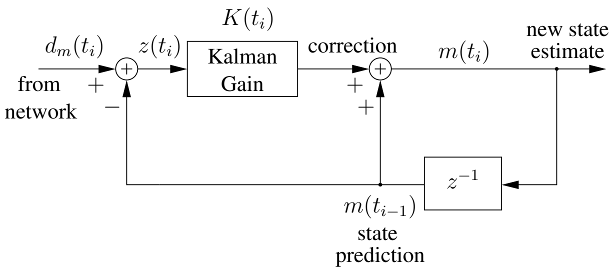


Figure 6: State estimation with a Kalman filter.

Kalman gain $K(t_i)$ which provides the correction to the estimation $m(t_i)$ according to:

$$m(t_i) = (1 - K(t_i)) \cdot m(t_{i-1}) + K(t_i) \cdot (d_m(t_i)) \quad (11)$$

Eq. (11) shows that the Kalman filter, in this specific case, is equivalent to an EWMA filter made of two additive terms [9]: the first taking into account the contribution of the previous state estimate $m(t_{i-1})$ and the second accounting for the contribution of the measurement $d_m(t_i)$. The gain $K(t_i)$ balances these two contributions: if $K(t_i)$ is large, more weight is given to the measurement, conversely when $K(t_i)$ is small more weight is given to the state estimate. The Kalman gain is updated according to the process and the measurement noise variance:

$$K(t_i) = \frac{P(t_{i-1}) + Q(t_i)}{(P(t_{i-1}) + Q(t_i)) + \sigma_n^2(t_i)} \quad (12)$$

where $P(t_k)$ is the system error variance defined as:

$$P(t_i) = E[(\overline{m}(t_i) - m(t_i))^2] \quad (13)$$

which we want to minimize at steady state. This value can be recursively computed at each step as follows:

$$P(t_i) = (1 - K(t_i)) \cdot (P(t_{i-1}) + Q(t_i)) \quad (14)$$

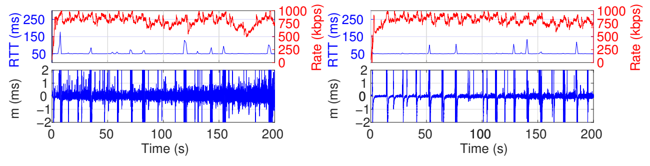
An interpretation of (12) is the following: as the measurement error variance $\sigma_n^2(t_i)$ approaches zero, the measurement is trusted more and thus the Kalman gain increases. Vice versa, as the error covariance P approaches zero, the measurement is trusted less and the Kalman gain decreases.

4.1.2 Implementation

Several parameters have to be tuned to implement the Kalman filter: 1) the statistics of the state and measurement noise processes, 2) the initial conditions of P.

We start by considering the state noise variance Q. First, we assume Q to be constant and we carry out a set of experiments to appropriately tune it.

Figure 7 shows the effect of Q on both the estimate of the one delay gradient $m(t)$ and the sending rate. Four different values of Q have been tested in the case of a single GCC flow over a 1Mbps link. Figure 7 also reports the RTT dynamics which is correlated to the one way delay gradient. We notice that, when $Q = 10^{-4}$, the reaction to RTT variations is slow, thus provoking large queuing delays which are clearly visible in the corresponding RTT dynamics (Figure 7(d)). Figure 7(c) shows that, when $Q = 10^{-3}$, the estimated delay gradient is well correlated


(a) $Q = 10^{-1}$ (b) $Q = 10^{-2}$

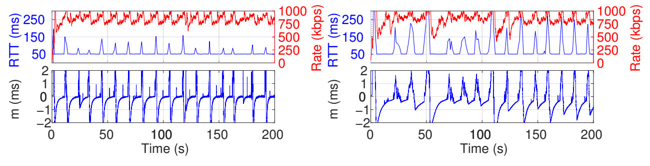

(c) $Q = 10^{-3}$ (d) $Q = 10^{-4}$

Figure 7: Effect of the state noise variance Q on the measured gradient, RTT and Sending Rate

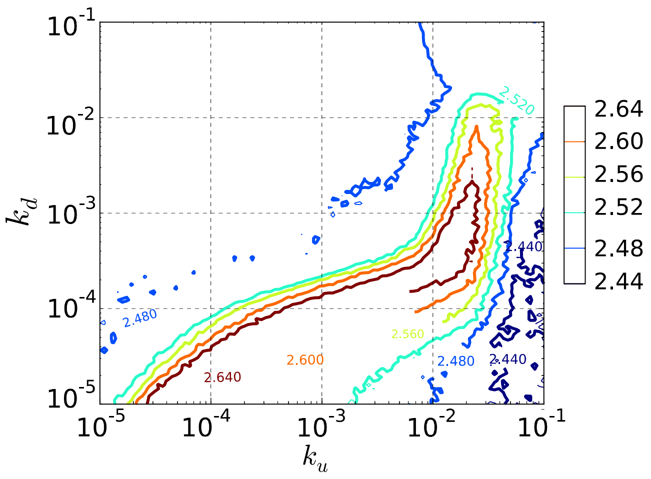


Figure 8: Contour plot of the objective function U as a function of k_u and k_d

with the RTT dynamics: in correspondence to RTT increases, $m(t)$ exhibits positive spikes and in correspondence to RTT decreases $m(t)$ shows negative spikes. Finally, $m(t)$ is steered to 0 when the RTT is kept close to the propagation delay $RTT_{min} = 50$ms since in this case the delay gradient is zero. When $Q = 10^{-1}$ and $Q = 10^{-2}$ are used, $m(t)$ becomes very sensitive to the jitter noise causing link underutilization due to the fact that the reaction to congestion might be driven by the jitter noise. Based on this analysis, we have set Q equal to:

$$Q = 10^{-3} \tag{15}$$

We now focus on the issue of estimating the measurement noise variance $\hat{\sigma}_n^2(t_i)$. We estimate $\hat{\sigma}_n^2(t_i)$ by using an exponential moving average filter of the *residual* $z(t_i) = d_m(t_i) - m(t_{i-1})$:

$$\hat{\sigma}_n^2(t_i) = \beta \cdot \hat{\sigma}_v^2(t_{i-1}) + (1-\beta) \cdot z^2(t_i) \tag{16}$$

where $\beta = 0.95$ and $d_m(t_i)$ is measured according to (4). This is a typical methodology employed when information about the measurement noise is not available [2].

Finally, regarding the system initial conditions, quick convergence is guaranteed when the initial system error variance $P(0)$ is larger than the state noise variance Q. We have set $P(0) = 10^{-1}$. In these conditions, the initial estimate of the state can be freely set to any value (we have chosen $m(0) = 0$).

4.2 Adaptive threshold design

The goal of the adaptive threshold block of Figure 1 is to adapt the sensitivity of the algorithm to the delay gradient based on network conditions. We show that the threshold $\gamma(t_i)$, used by the *over-use detector,* must be made adaptive otherwise two issues can occur: 1) the delay-based control action may have no effect when the size of the bottleneck queue along the path is not sufficiently large and 2) the GCC flow may be starved by a concurrent loss-based TCP flow. Toward this end, we propose the following adaptive threshold:

$$\gamma(t_i) = \gamma(t_{i-1}) + \Delta T \cdot k_\gamma(t_i)(|m(t_i)| - \gamma(t_{i-1})) \tag{17}$$

where $\Delta T = t_i - t_{i-1}$, and t_i is the time instant the i-th video frame is received. The gain $k_\gamma(t_i)$ is defined as follows:

$$k_\gamma(t_i) = \begin{cases} k_d & |m(t_i)| < \gamma(t_{i-1}) \\ k_u & \text{otherwise} \end{cases} \tag{18}$$

where k_u and k_d determine, respectively, the speed at which the threshold is increased or decreased. By using (17), the delay-based controller at the receiver compares the one way delay gradient $m(t_i)$ with a threshold $\gamma(t_i)$ which is a low-pass filtered version of $|m(t_i)|$. The rationale is the following: when $m(t_i)$ overshoots $\gamma(t_i)$, the delay-based controller reduces $m(t_i)$ and the threshold $\gamma(t_i)$ follows $m(t_i)$ with a slower time constant so that $m(t_i)$ stays below $\gamma(t_i)$ and leads the algorithm into the decrease state. This continues until $m(t_i)$ again overshoots $\gamma(t_i)$ and the delay-based algorithm again reduces $m(t_i)$. This adaptation avoids the GCC flow starvation when a TCP flow enters the bottleneck. It is worth noting that when $\gamma(t_i) > m(t_i)$, $\gamma(t_i)$ follows $m(t_i)$ with a shorter time constant so that fewer decrease events are required to make $m(t_i) < \gamma(t_i)$.

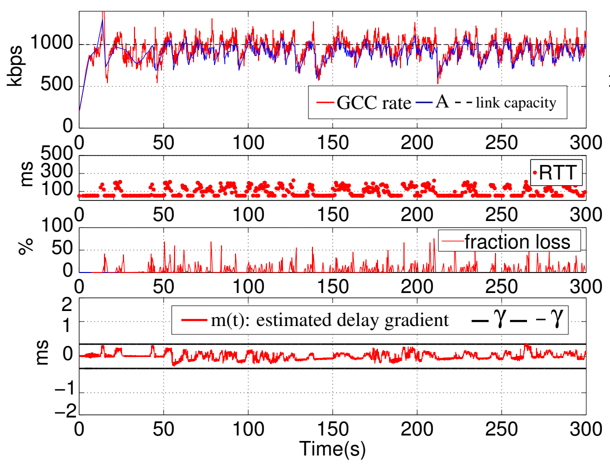
Choice of the threshold parameters. A key feature in the design of the adaptive threshold resides in the tuning of the parameters k_u and k_d which define the time constant at which the threshold $\gamma(t_i)$ adapts to the delay gradient. In other words, the parameters k_u and k_d determine the algorithm sensitivity to the estimated one way delay $m(t)$. In order to tune these parameters we have defined an optimization problem by employing the objective function proposed in [28]:

$$U(\boldsymbol{x}) = \sum_{i=1}^{N} U^\alpha(x_i) \tag{19}$$

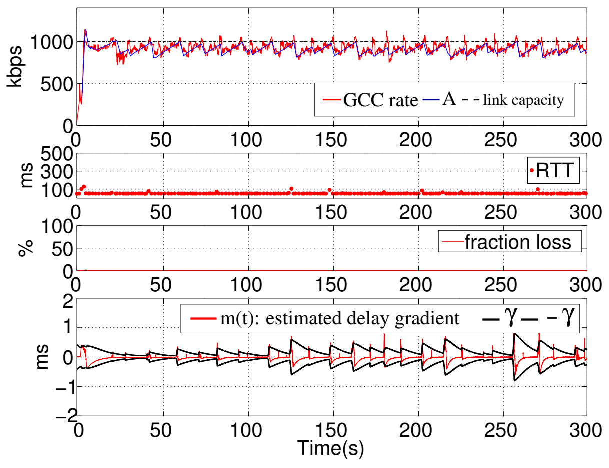
where $U^\alpha(x_i)$ is the objective function measured for the i-th flow. The overall utilization is obtained as the sum of $U^\alpha(x_i)$ for each of the N concurrent flows. x_i is the average throughput of the i-th flow and $U^\alpha(x)$ is a concave function given by:

$$U^\alpha(x_i) = \frac{x_i^{1-\alpha}}{1-\alpha} \tag{20}$$

For any value of $\alpha > 0$ this optimization problem is Pareto-efficient [28]. The maximization of U implies a fair



(a) Static threshold



(b) Adaptive threshold

Figure 9: Effect of the adaptive threshold in the case of a single GCC flow over a 1Mpbs link with queue size $\overline{T}_q = 150$ms

allocation of the throughput among concurrent flows. In the case of video conferencing we need to extend (19) to consider the impact of the queuing delay on system performance [31]:

$$U(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{N} (U^{\alpha}(x_i) - \delta \cdot U^{\beta}(y_i)) \tag{21}$$

The same rationale used for $U^{\alpha}(x_i)$ applies to $U^{\beta}(y_i)$ where y_i refers to the queuing delay measured for the i-th flow. The parameters α and β express the trade-off between fairness and efficiency for throughput and delay whereas δ expresses the relative importance of delay with respect to throughput. Following the rationale of [31] we have used $\alpha = 2$, $\beta = 1$, i.e. we put more emphasis on the throughput fairness, and $\delta = 0.15$, which guarantees a good balance between $U^{\alpha}(x_i)$ and $U^{\beta}(y_i)$.

We have defined three different scenarios in which we measure the objective function (21). These are three evaluation scenarios described in Section 6: 1) a single GCC flow over a bottleneck with variable link capacity (Section 6.1), 2) multiple concurrent GCC flows over a bottleneck with constant link capacity (Section 6.2), 3) a GCC flow against a TCP flow (similarly to the case of Section 6.3). The objective function has been computed for every value of k_u and k_d in the range $[10^{-5}, 0.1] \times [10^{-5}, 0.1]$ divided in 200×200 intervals in logarithmic scale. The contour map of the sum of the normalized objective functions U obtained in each scenario for every couple (k_u,k_d) is shown in Figure 8. U increases when $k_u > k_d$ i.e. when the time constant at which $\gamma(t_i)$ is increased is smaller than the time constant at which it is decreased. However when $k_u \gg k_d$ (in the bottom-right corner of Figure 8) the threshold decreases too slowly reducing the algorithm sensitivity to the delay inflation; this induces high queuing delays and, as a consequence, U decreases. On the other hand, when $k_u < k_d$ the algorithm becomes very sensitive to the delay gredient which leads to throughput degradation in the presence of concurrent TCP flows. The optimum value is obtained for (k_u,k_d) $= (0.01, 0.00018)$. This guarantees a good balance between high throughput, delay reduction, intra and inter-protocol fairness.

Influence of the bottleneck queue size on the delay gradient. In this paragraph we show why the threshold must be adaptive in order to avoid the inhibition of the delay-based algorithm when

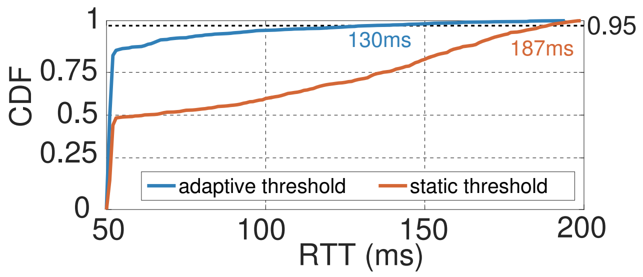
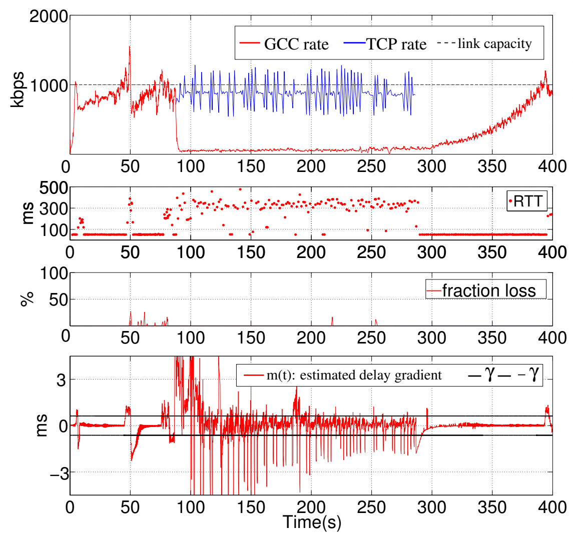
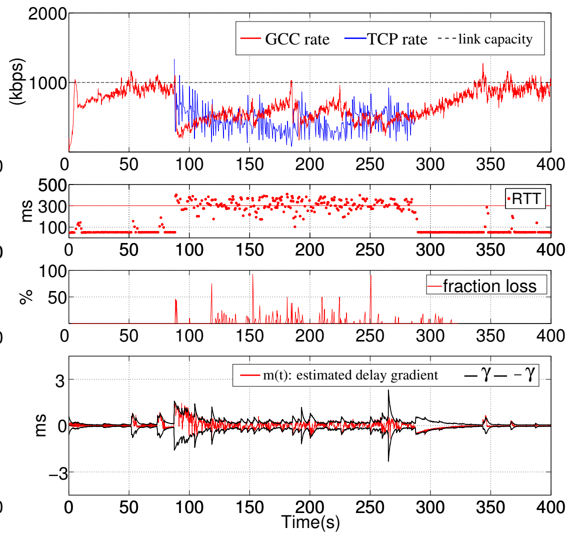


Figure 10: RTT comparison in the case of a single GCC flow

the size of the bottleneck queue is not sufficiently large. For this purpose we consider the case of a single flow accessing a bottleneck. We show that an undesirable phenomenon can arise if the threshold is statically set to a value $\bar{\gamma}$: in particular, if $\bar{\gamma}$ is larger than the maximum measurable delay gradient m_M the delay-based algorithm has no effect. Figure 9 (a) shows a real experiment (see Section 5 for details on the testbed) where a single GCC flow enters a 1Mbps bottleneck link with a drop-tail queue whose maximum queuing time is $\overline{T}_q = 150$ms. Figure 9 compares the GCC rate, the RTT, and the fraction loss dynamics in the case of a static setting of the threshold (Figure 9 (a)) and in the case of the adaptive threshold (Figure 9 (b)). With a static threshold $\bar{\gamma}$, the delay-based controller becomes ineffective and is not able to react to delay inflation. This is due to the fact that the maximum value of the delay gradient m_M, which depends on $\overline{T}_q$, is smaller than $\bar{\gamma}$ i.e. the value of $\bar{\gamma}$ is too large for $\overline{T}_q = 150$ms. Figure 9 (b) shows that, with the adaptive threshold, $\gamma(t)$ follows $m(t)$ with a slower time constant and, when $m(t)$ overshoots $\gamma(t)$, the delay-based algorithm can decrease the sending rate. Moreover, since $k_u > k_d$ the threshold is increased quicker than how it is decreased. Figure 9 (b) also shows that the controller is able to avoid packet losses, i.e. $f_l(t) = 0$ throughout the whole duration of the experiment. To further show the benefits of the adaptive threshold, Figure 10 compares the cumulative distribution function of the measured RTT in the two experiments reported in Figure 9. The median value of the RTT is very close to the propagation delay $RTT_{min} = 50$ms set by the testbed in both of the cases, whereas thanks to the adaptive threshold the 95th percentile of the RTT is reduced from 187ms to 130ms.



(a) Static threshold



(b) Adaptive threshold

Figure 11: One GCC flow vs one TCP flow. Bottleneck parameters: queue size $\overline{T}_q = 350$ms, link capacity $C = 1000$kbps

Effect of a concurrent TCP flow on the delay gradient. In this paragraph we show that the threshold must be adaptive in order to avoid the starvation of a GCC flow in the presence of a concurrent loss-based flow. Towards this end, we consider a single GCC flow with a concurrent long-lived TCP flow. We show that a static setting of the threshold might lead to the starvation of the GCC flow. In this scenario, the one way delay gradient can be expressed as the sum of two components:

$$m(t) = m_{\mathrm{GCC}}(t) + m_{\mathrm{TCP}}(t) \tag{22}$$

where $m_{\mathrm{GCC}}(t)$ and $m_{\mathrm{TCP}}(t)$ are the delay gradients of the GCC and the TCP flow respectively. It can be shown that the maximum delay gradient $m_{\mathrm{TCP,M}}$ due to a TCP flow can be much larger than that of a GCC flow [6]. Since in this case $m(t)$ contains the component $m_{\mathrm{TCP}}(t)$, if $m_{\mathrm{TCP}}(t) \gg m_{\mathrm{GCC}}(t)$ the GCC flow will decrease the sending rate not due to the self-inflicted delay, but due to the TCP flow.

Figure 11 shows a real experiment, conducted by employing the testbed described in Section 5, in which a GCC flow competes with a TCP Cubic flow over 1Mbps bottleneck link with a drop-tail queue whose maximum queuing time is $\overline{T}_q = 350$ms. Figure 11 (a) shows that, when a static threshold is used, the GCC flow gets starved. In particular, when the TCP flow is started, $m(t)$ begins to oscillate above the threshold $\overline{\gamma}$ mainly due to the delay gradient of the TCP flow $m_{\mathrm{TCP}}(t)$, which triggers a large number of overuse signals. Consequently, the remote rate controller FSM enters the decrease mode which reduces the value of A_r according to (3). On the other hand, Figure 11 (b) shows that the adaptive threshold is able to solve the starvation issue and allows the GCC flow to share the bottleneck fairly with the concurrent TCP flow. In particular, after TCP is started, $\gamma(t)$ follows $m(t)$ with a smaller time constant which avoids the generation of a large number of consecutive overuse signals and prevents the starvation of the GCC flow.

5. TESTBED

Figure 12 shows the experimental testbed employed to emulate a WAN scenario. It consists of four Linux machines equipped with a Linux kernel 3.16.0. Two nodes, each one running several sessions of Chromium browsers[4] and an application to generate or receive TCP long-lived flows, are connected through an Ethernet cable. With this configuration the behavior of GCC flows when competing against TCP flows can be analyzed. Another node runs a web server which handles the signaling required to establish the video calls between the browsers. The last node is the testbed *controller* that orchestrates the experiments via `ssh` commands. The testbed controller undertakes the following tasks: 1) it places the WebRTC calls starting the GCC flows; 2) it sets the link capacity C and the bottleneck queue size $\overline{T}_q$ on Node 1; 3) it sets the propagation delay RTT_{min} on Node 2 [5]; 4) it starts the TCP flows when required.

The queue size q_M on Node 1 has been set by considering the maximum time required to drain the buffer $\overline{T}_q$, i.e. $q_M = \overline{T}_q \cdot C$. The round trip propagation delay $RTT_{min} = 50$ms considered as the sum of the propagation delays on the direct and reverse path (25ms each) has been set on Node 2 through the NetEm Linux module, whereas the bottleneck buffer size has been set to $\overline{T}_q = 300$ms by following the evaluation criteria provided in [25]. We have used a Token Bucket Filter (TBF) to set the ingress link capacity C of Node 1. We have turned off the NIC optimization parameters, i.e. TCP segmentation offload, jumbo frames, generic segmentation offload, since they could interfere with the experiments.

Video and TCP settings. The TCP sources employ the CUBIC congestion control which is the default in Linux kernel. The congestion window, the slow-start threshold, the RTT, and the sequence number are logged. A Web server[6] provides the HTML page that handles the signaling between the peers using

[4]https://chromium.googlesource.com/chromium/src.git

[5]It is recommended to split the rate control from the delay emulation

[6]https://apprtc.appspot.com/

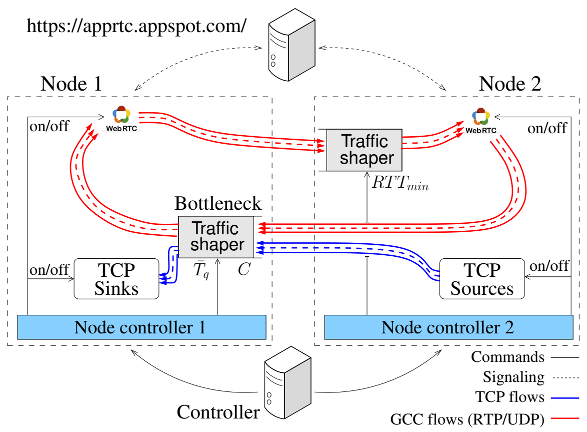


Figure 12: Experimental testbed

the WebRTC JavaScript API. The same video sequence is used to enforce experiments reproducibility. To the purpose, we have used the Linux kernel module `v4l2loopback`[7] to create a virtual webcam device which cyclically repeats the *"Four People"*[8] YUV test sequence. Chromium encodes the raw video source with the VP8 video encoder[9]. We have measured that, without any bandwidth limitation, VP8 limits the sending bitrate $A_s(t)$ to a maximum value of 2Mbps.

Metrics. In order to quantitatively assess the performance of GCC we consider QoS metrics such as packet loss ratio, average bitrate, and delay, which are known to be well correlated with QoE metrics through, for instance, the IQX hypothesis [10]. Following this approach has the merit of focusing the discussion on metrics that are not sensitive to application specific aspects, such as the employed video encoder. Moreover, splitting the evaluation of QoE metrics from QoS metrics also follows the guidelines recently defined within the IETF RTP Media Congestion Avoidance Techniques (RMCAT) working group. In particular, we consider:

- **Channel Utilization** $U = R_r/C$, where C is the known link capacity and R_r is the average received rate;
- **Loss ratio** $l = (\text{bytes lost})/(\text{bytes sent})$;
- **50th and 95th percentile of queuing delay**, measured as $RTT(t) - RTT_{min}$ over all the RTT samples reported in the RTCP feedbacks during the experiments;
- **Jain's Fairness Index**: $J_{FI}(t) = \frac{(\sum_{i=1}^{N} x_i(t))^2}{N\sum_{i=1}^{N} x_i(t)^2}$, where $x_i(t)$ is the measured instantaneous throughput of the i-th flow and N is the total number of competing flows.

6. EXPERIMENTAL EVALUATION

This section presents the experimental results obtained employing the testbed described in Section 5. The analysis is based on the evaluation criteria under discussion in the IETF WG RMCAT [25]. It is worth to mention that a much broader experimental evaluation has been performed, but due to space constraints we have decided to include only experimental results of the scenarios defined in [25]. We argue that this choice facilitates experimental comparisons with other algorithms proposed in RMCAT. In particular, the goal is to check if GCC satisfies the requirements defined in [24] i.e. low queuing in the absence of competing heterogeneous traffic and a reasonable share of bandwidth when competing with other homogeneous or heterogeneous flows. For every scenario ten experiments have been run. For each of them a Table is provided which contains the average value of the metrics measured in the ten runs.

[7]https://github.com/umlaeute/v4l2loopback
[8]https://people.xiph.org/~thdavies/x264_streams/FourPeople_1280x720_30/
[9]http://www.webmproject.org/

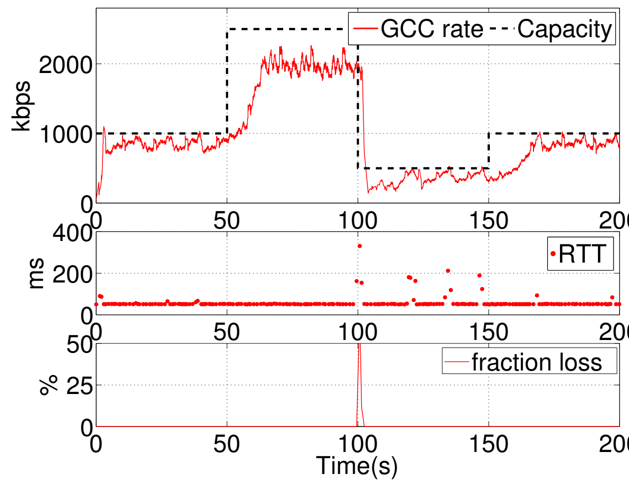


Figure 13: Rate, RTT and fraction loss dynamics in the case of a single GCC with variable link capacity

Metric	Value
Channel Utilization	84%
Queuing Percentile (50th - 95th)	20ms - 195ms
Loss Ratio	0.02%

Table 1: Average value of the metrics measured in the case of a single GCC with variable link capacity

6.1 Single GCC with variable link capacity

This scenario investigates how quickly the video bitrate controlled by GCC reacts to sudden changes of the link capacity. For this purpose, we vary the link capacity every interval of 50 seconds. We consider four time intervals in which the capacity is set respectively to 1Mbps, 2.5Mbps, 0.5Mbps, and 1Mbps, such that the video call lasts 200s. Table 1 shows the average value of the metrics measured throughout the ten experiments. Figure 13 shows the dynamics of the GCC rate, RTT, and fraction loss. This experiment shows that GCC matches the time-varying link capacity. It is important to observe that in the time interval in which the capacity is set to 2.5Mbps, the GCC flow cannot match the link capacity since the video encoder does not produce more than 2Mbps. Furthermore, GCC is able to contain the queuing delay since the RTT is kept close to the round trip propagation delay $RTT_{min} = 50$ms during the entire video call. We have measured an average value of the 50th percentile of the queuing delay equal to 20ms, whereas the 95th percentile is 195ms. We notice that, in order to react to the link capacity drop, the algorithm has to reduce the sending rate to keep a low queuing delay and loss ratio. When the link capacity increases, GCC takes about 25 seconds to reach

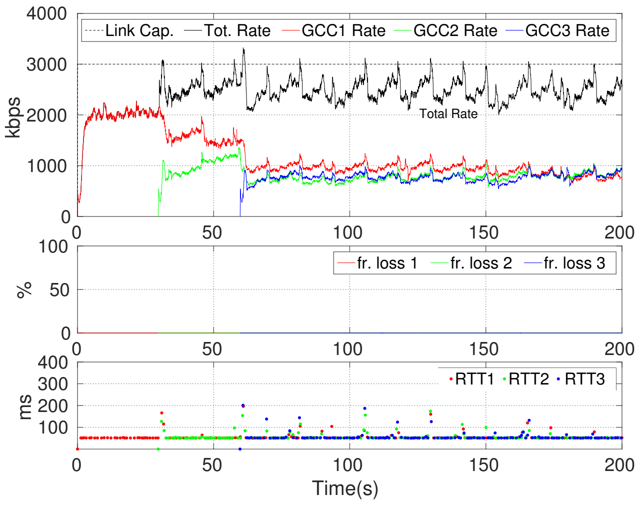


Figure 14: Rate, fraction loss, and RTT dynamics in the case of three concurrent GCC flows over a 3Mbps link

Metric	Value
Cumulative Utilization	86%
Queuing Percentile (50th - 95th)	10ms - 61ms
Cumulative Loss Ratio	0%
Jain Fairness Index	0.93

Table 2: Average value of the metrics measured in the case of three concurrent GCC flows over a 3Mbps link

the new link capacity value. This results in an average channel utilization equal to 84%.

6.2 Intra-protocol fairness

The aim of this scenario is to investigate the GCC intra-protocol fairness. Toward this end, we have considered three concurrent GCC flows over a 3Mbps link. Each flow is started 30 seconds after the other. The experiment lasts 200 seconds. Table 2 shows the average value of the metrics measured over ten experiments when all the three flows are active. Figure 14 shows the dynamics for one experiment. This experiment shows that GCC is not affected by the *"late-comer effect"* [7]; the three GCC flows fairly share the link and the measured Jain Fairness Index approaches 0.93. Convergence can be intuitively shown by considering that the equivalent estimated delay gradient measured by each flow is given by:

$$m(t) = m_{\mathrm{GCC}_1}(t) + m_{\mathrm{GCC}_2}(t) + m_{\mathrm{GCC}_3}(t) \qquad (23)$$

so that an increase of the delay gradient induced by one of the flow triggers an "*overuse signal*" for any of the concurrent flows. Since the bitrate is decreased according to (3), a flow with higher bitrate will experience a higher rate reduction, thus eventually leading to convergence. No packet is lost during the whole duration of the experiment. Concerning the queuing delay, in this experiment we have measured an average value of the 50th percentile equal to 10ms ($RTT_{min} = 50$ms), whereas the measured 95th percentile is equal to 61ms.

To complete the investigation on the intra-protocol fairness we consider a scenario with two flows in the case of a link with variable capacity. The link capacity is varied every interval of 25 seconds. We consider 5 intervals in which the capacity is set respectively to 4Mbps, 2Mbps, 4Mbps, 1Mbps and 2Mbps, such that the video call lasts 125 seconds. Table 3 shows the average value of the metrics measured over the ten experiments. Figure 15 shows the dynamics of the variables in one experiment. The Jain Fairness index is equal to 0.87, which confirms that GCC provides intra-protocol fairness.

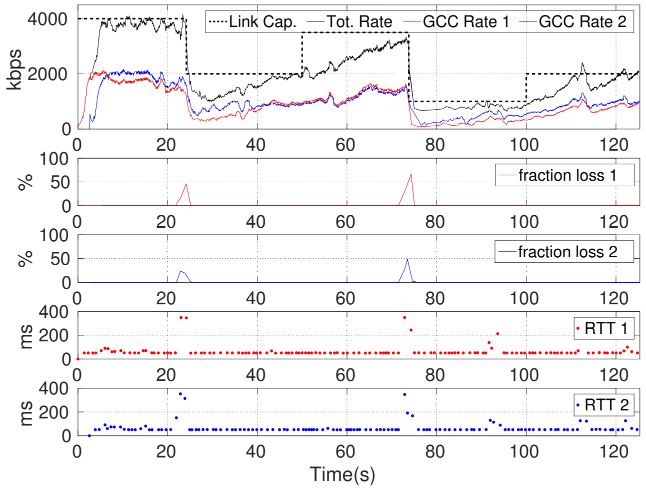


Figure 15: Rate, fraction loss and RTT dynamics in the case of two GCC flows over a link with variable capacity

Metric	Value
Cumulative Utilization	85%
Queuing Percentile (50th - 95th)	15ms - 103ms
Cumulative Loss Ratio	1%
Jain Fairness Index	0.87

Table 3: Average value of the metrics measured in the case of two GCC flows over a link with variable capacity

6.3 One GCC flow with concurrent long-lived TCP Flows

This scenario aims at testing the inter-protocol fairness. We have considered one GCC flow against 99 long-lived TCP flows over a bottleneck link with a constant capacity set equal to 100Mbps. The video call lasts 400 seconds and the TCP flows are active for 200 seconds in the time interval [100, 300] seconds. Table 2 summarizes the metrics obtained in the ten experiments measured when the TCP flows are active. Figure 16 shows the dynamics obtained by one of the experiment carried out. The figure shows that, as soon as the TCP flows enter the bottleneck, the GCC sending rate is reduced close to the fair share at 1Mbps and oscillates around this value when the TCP flows are active. This is made possible by the adaptive threshold (see Section 4.2) that, by reducing the sensitivity of the delay-based controller, leads the GCC flow to be controlled by both the loss-based and delay-based algorithms in the presence of concurrent loss-based traffic. In

Metric	Value
GCC Throughput	1.3Mbps
Queuing Percentile (50th - 95th)	250ms - 295ms
Loss Ratio	1.2%
Jain Fairness Index	0.81

Table 4: Average value of the metrics measured in the case of one GCC flow sharing a 100Mbps link with 99 TCP flows

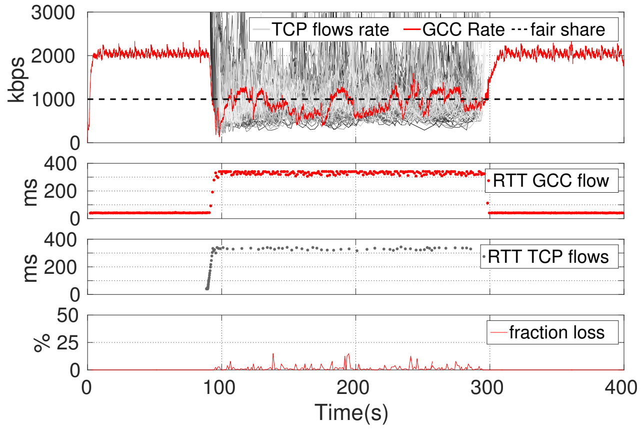


Figure 16: The case of one GCC flow sharing a 100Mbps link with 99 TCP flows

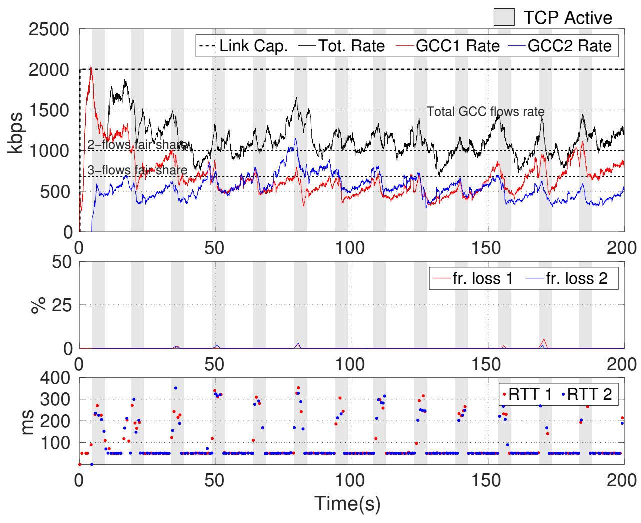


Figure 17: Rate, fraction loss, and RTT dynamics of two GCC flows in the presence of short-lived TCP flows

fact, the value of the fraction loss measured for the GCC flow is greater than zero when the TCP flows are active confirming that the algorithm is also operating in loss-based mode. As expected, the queuing delay cannot be contained in the presence of TCP traffic which tends to fills the bottleneck buffer hindering the real-time interaction.

6.4 GCC flows in the presence of short-lived TCP flows

The aim of this scenario is to investigate how the GCC flows behave in the presence of short-lived TCP traffic. To the purpose, we have considered two concurrent GCC flows over a 2Mbps link; 5 seconds after the beginning of the video calls we start one TCP flow for 3 seconds every 12 seconds. This scenario is interesting since the interaction among the flows differs from the case of a long-lived TCP flow since the short-lived TCP flows provoke several traffic bursts that are typical in the case of small file transfer (i.e. web page download). Table 5 summarizes the metrics obtained for the two GCC flows, whereas Figure 17 shows the dynamics of one experiment: during the time interval in which the TCP flow is active (depicted in gray) the RTT increases due to the TCP flows

Metric	Value
GCC flows Utilization	72%
Queuing Percentile (50th - 95th)	109ms - 285ms
Loss Ratio	0.7%
Jain Fairness Index	0.78

Table 5: Average value of the metrics measured in the case of two GCC flows in the presence of short-lived TCP flows

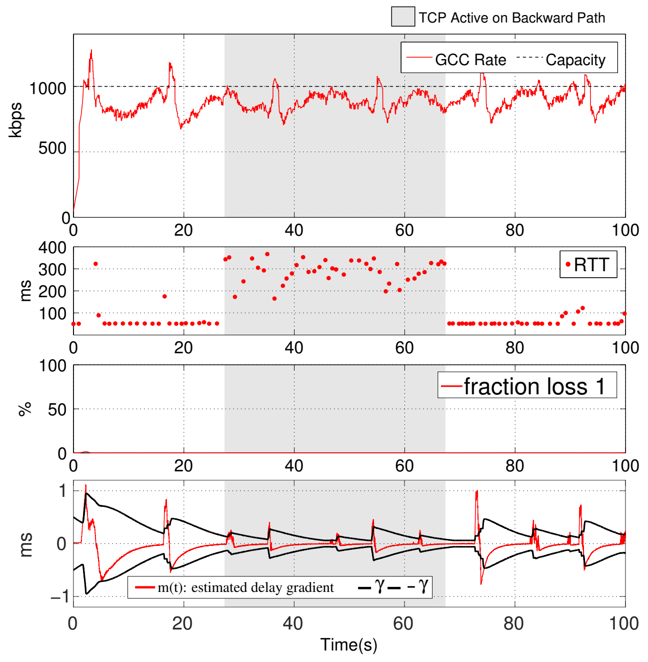


Figure 18: Rate, fraction loss, RTT and delay gradient dynamics for the GCC flow on the direct path

that inflate the bottleneck buffer. When the TCP traffic is not present the GCC flows sending rates increase up to 1Mbps which is the ideal bandwidth fair-share for two concurrent flows. When TCP is started the sending rate of the flows is reduced due to the induced delay inflation. Therefore, the overall channel utilization measured for the two GCC flows is equal to 72%.

6.5 The Effect of Reverse Traffic

This scenario assesses the effect of reverse traffic on the metrics. For this purpose, a video call is established on the direct path, whose capacity is 1Mbps, for 100 seconds. During the call, on a 1Mbps reverse path link, a TCP flow is started for 40 seconds. Table 6 summarizes the average value of the metrics obtained for the GCC flows considering all the runs. Figure 18 shows one experimental result dynamics: we depict in gray the time interval during which the TCP flow is active on the reverse path. This test clearly shows that the GCC rate and the fraction loss are not

Metric	Value
Channel Utilization	90%
Queuing Percentile (50th - 95th)	10ms - 159ms
Loss Ratio	0%

Table 6: Average value of the metrics measured for the GCC flow on the direct path

affected by the reverse traffic. This is due to the fact that the estimated delay gradient on the direct path is not affected by the presence of reverse traffic, even though the RTT inflation confirms the TCP flow presence. The average channel utilization measured on the direct path is 90%. The median queuing delay measured on the direct path is 10ms and the 95th percentile is 159ms.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a congestion control algorithm for real-time flows that is currently used in Google Hangouts. We have carried out an experimental evaluation based on the guidelines under definition in the IETF RMCAT working group. Results obtained using Google Chromium browser show that: 1) it adapts the sending rate to track the link capacity, 2) it provides intra-protocol and inter-protocol fairness with long and short-lived TCP flows and 3) it is not affected by reverse traffic. As future work we plan a massive measurement campaign involving a large set of Chrome users.

8. ACKNOWLEDGMENTS

This work has been partially supported by the Italian Ministry of Education, Universities and Research (MIUR) through the MAIVISTO project (PAC02L1 00061). Any opinions, findings, conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. Webrtc 1.0: Real-time communication between browsers. *W3C Working Draft*, Feb. 2015.

[2] R. Bos, X. Bombois, and P. M. Van den Hof. Designing a Kalman filter when no noise covariance information is available. In *Proc. of the 16th IFAC World Congress*, volume 16, pages 212–212, Jul. 2005.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communicationss*, 13(8):1465–1480, Oct. 1995.

[4] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl. Reducing Internet Latency: A Survey of Techniques and their Merits. *IEEE Comm. Surveys & Tutorials*, in press.

[5] E. Brosh, S. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The Delay-Friendliness of TCP for Real-Time Traffic. *IEEE/ACM Transactions on Networking*, 18(5):1478–1491, Oct. 2010.

[6] G. Carlucci, L. De Cicco, and S. Mascolo. Modelling and Control for Web Real-Time Communication. In *53rd IEEE Conference on Decision and Control*, Los Angeles, CA, USA, Dec. 2014.

[7] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti. The Quest for LEDBAT Fairness. In *IEEE Global Telecommunications Conference*, pages 1–6, Dec. 2010.

[8] L. De Cicco and S. Mascolo. A mathematical model of the Skype VoIP congestion control algorithm. *IEEE Transactions on Automatic Control*, 55(3):790–795, Mar. 2010.

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

[10] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *IEEE Network*, 24(2):36–41, 2010.

[11] G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital control of dynamic systems*, volume 3. Addison-wesley Menlo Park, 1998.

[12] 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, Apr. 2004.

[13] A. Gurtov and S. Floyd. Modeling wireless links for transport protocols. *ACM SIGCOMM CCR*, 34(2):85–96, Apr. 2004.

[14] D. A. Hayes and G. Armitage. Revisiting TCP Congestion Control Using Delay Gradients. In *Proceedings of the 10th International IFIP TC 6 Conference on Networking - Volume Part II*, pages 328–341, Jul. 2011.

[15] S. Holmer, H. Lundin, G. Carlucci, L. De Cicco, and S. Mascolo. Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. *IETF Draft*, June 2015.

[16] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM CCR*, 19(5):56–71, Oct. 1989.

[17] I. Johansson. Self-clocked Rate Adaptation for Conversational Video in LTE. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, pages 51–56, Chicago, Illinois, USA, Aug. 2014.

[18] A. Kuzmanovic and E. Knightly. TCP-LP: low-priority service via end-point congestion control. *Networking, IEEE/ACM Transactions on*, 14(4):739–752, Aug 2006.

[19] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM CCR*, 45(5):537–550, Oct. 2015.

[20] M. Nagy, V. Singh, J. Ott, and L. Eggert. Congestion Control Using FEC for Conversational Multimedia Communication. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 191–202, Singapore, Mar. 2014.

[21] P. Ni, R. Eg, A. Eichhorn, C. Griwodz, and P. Halvorsen. Flicker effects in adaptive video streaming to handheld devices. In *Proc. of ACM International Conference on Multimedia*, pages 463–472, Nov. 2011.

[22] K. Nichols and V. Jacobson. Controlling queue delay. *Queue, ACM*, 10(5):20:20–20:34, May 2012.

[23] R. S. Prasad, M. Jain, and C. Dovrolis. On the effectiveness of delay-based congestion avoidance. In *Proc. of Workshop on Protocols for Fast Long-Distance Networks*, volume 4, Feb. 2004.

[24] J. Randell and Z. Sarker. Congestion control requirements for RMCAT. *Draft IETF*, Dec. 2014.

[25] Z. Sarker, V. Singh, X. Zhu, and R. M. Test Cases for Evaluating RMCAT Proposals. *IETF Draft*, Aug. 2015.

[26] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low extra delay background transport (LEDBAT). *RFC 6817*, Dec. 2012.

[27] D. Sisalem and A. Wolisz. Lda+: a tcp-friendly adaptation scheme for multimedia communication. In *IEEE International Conference on Multimedia and Expo*, volume 3, pages 1619–1622 vol.3, Aug. 2000.

[28] R. Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[29] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via tcp: An analytic performance study. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(2):16:1–16:22, May 2008.

[30] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

[31] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. *ACM SIGCOMM CCR*, 43(4):123–134, Oct. 2013.

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

[33] 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, Jun. 2014.

[34] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. *ACM SIGCOMM CCR*, 45(5):509–522, Oct. 2015.

[35] X. Zhu, R. Pan, S. Mena, P. Jones, J. Fu, S. D'Aronco, and C. Ganzhorn. Nada: A unified congestion control scheme for real-time media. *IETF Draft*, Mar. 2015.