A Mismatch Controller for Implementing Rate-based Transport Protocols

Luca De Cicco
Politecnico di Bari
Dipartimento di Elettrotecnica ed Elettronica
Via Re David, 200
Bari, Italy
Email: ldecicco@poliba.it

Saverio Mascolo
Politecnico di Bari
Dipartimento di Elettrotecnica ed Elettronica
Via Re David, 200
Bari, Italy
Email: mascolo@poliba.it

***Abstract*—End-to-end rate-based congestion control algorithms, such as TFRC, RAP or ARC, are advocated for audio/video transport over the Internet instead of window-based protocols. The reason is that a smoother dynamics of packet sending is advantageous when avoiding abrupt rate changes is of importance. Once the sending rate has been computed by a generic rate-based congestion controller, all algorithms proposed in the literature schedule packets to be sent spaced at intervals that are equal to the inverse of the sending rate. In this paper we show that such an implementation omits to consider a key feature. In fact, the scheduled sending time of a packet is affected by significant uncertainty due to the fact that it is handled by the Operating System, which manages a CPU shared by other processes. A significant experiment reported in the paper shows that a required constant sending rate can in practice turn into an effective sending rate that is as low as one half of the desired one. To overcome this problem, a Rate Mismatch Controller (RMC) is designed aiming at counteracting the disturbance on the effective sending time due to the CPU time-varying load. Experimental results using Linux OS highlight the effectiveness of the proposed controller.**

I. Introduction

Today, the most part of the Internet traffic is handled by the TCP [1], which implements a congestion control protocol [15] that has been extremely successful to guarantee network stability without admission control. The TCP congestion control is a window-based protocol that sends a window $W(t_k)$ of packets every time t_k an acknowledgment packet is received. This behaviour originates the *bursty* nature of the TCP, i.e. the fact that packet are sent in burst of length $W(t_k)$. From the point of view of the network, the burstiness of the TCP increases network buffer requirements since queue sizes at least equal to the order of $W(t_k)$ must be provided for efficient link utilization. From the point of view of the user application, sending a burst of $W(t_k)$ packets is simple to be implemented but is not appropriate when the content to be transported requires a smoother sending dynamics such as in the case of a real-time video call. Therefore, today there are two active forces that are pushing towards the reduction of TCP burstiness: one is the spreading of gigabit networks for which burstiness mitigation means an important reduction of network buffer sizes; the other is the evolution of Internet from being an efficient platform for best-effort data delivery also to one for multimedia time-sensitive content [2], [3], [4].

The basic idea for reducing traffic burstiness induced by window-based congestion control is to design rate-based congestion control where packets are sent equally spaced in time at interval proportional to the inverse of the sending rate *r*. The sending rate *r* is computed every sampling time, f.i. every RTT, or every time a feedback report (or ACK) packet is received from the network or from the receiver. Feedbacks can be implicit, such as timeouts or DUPACKs, or explicit such as *Explicit Congestion Notification* (ECN) [22]. Once the sending rate r is computed, it is passed to a sending engine, or *send loop*, which is in charge of scheduling packets queued in the transmission buffer at the specified rate r.

Several rate-based schemes have been proposed in literature for the transport of multimedia streams [11], [12], [17], [24] but much less attention has been devoted to the implementation at user space of a rate-based congestion control algorithm. This may be at the root of the fact that, up to now, there is no evidence that a rate-based protocol is emerging as a widespread adopted solution. In fact, it is worth noting that YouTube employs standard TCP for delivering videos and implementations of peer-to-peer video distribution systems, which accounts for the 65% of the peer-to-peer traffic that in turn accounts for 60% of the Internet traffic [19], also employs TCP even though the use of TFRC had been long debated [8], [29]. Finally, Skype audio/video implements proprietary congestion control mechanisms at application layer over the UDP protocol [9], [10].

The analysis or design of rate-based control algorithms is out of the scope of this paper which indeed focuses on implementing a rate-based congestion control algorithm at application level over a general purpose Operating System. The problem here is that the unpredictable load of a general purpose Operating System (OS), which is due to other processes that share the CPU, forbids exact timing in packet sending. In fact, the scheduled sending time of a packet is affected by significant uncertainty due to the fact that it is handled by the Operating System, which is shared by other processes. A significative experiment reported in the paper shows that a required constant sending rate can in practice turn into an effective sending rate

that is as low as one half of the desired one. To overcome this problem, we propose a Rate Mismatch Controller (RMC) aiming at counteracting the disturbance on the effective sending time due to the CPU time-varying load.

By building upon the consolidated models of network congestion control proposed in [20], [21], the Rate Mismatch Controller is designed to form the inner loop of a cascade control configuration where the rate-based control is the outer loop. In the proposed model, the unpredictable and time-varying CPU load can be modeled as a disturbance acting on the send loop. It is worth noting that we are focusing on application layer protocols that are usually implemented in the *user space* in order to be portable on different platforms. The fact that the protocol runs in user space makes the interaction between the operating system and the application an even more critical factor.

The rest of this paper is organized as follows: in Section II the state of the art of the proposed solutions in the literature is presented; in Section III we focus on defining implementation issues of send loops; in Section IV we extend the model introduced in [20] by considering the effects of operating system interaction with the send loop, we propose a cascade control configuration to overcome the issues dealt in III and we perform a mathematical analysis in order to prove the effectiveness of the proposed controller; Section V provides a performance evaluation of the proposed controller carried out by using the Linux OS; finally, Section VI summarizes the main findings presented in this work.

II. Related Work

Research on rate-based congestion control algorithms has been active since a decade and has produced a significant amount of literature. In comparison, issues raised by the implementation of a rate-based sending protocol have received less attention. The first simple solution to the issue of implementing a rate-based congestion control can be found in [24], where a rate-based congestion control protocol named *Rate Adaptation Protocol* (RAP) is proposed. In that paper, authors propose to evenly space packets at intervals equal to the *inter-packet gap* (IPG) t_{ipg}, which is computed as follows:

$$t_{ipg} = \frac{s}{r(t)} \tag{1}$$

where s is the packet size and $r(t)$ is the computed rate. Equation (1) implies that the highest the rate the closest the packets should be sent in order to reflect the instantaneous sending rate $r(t)$. At first glance, this simple algorithm seems to be able to provide a sending rate that matches $r(t)$ and mitigates burstiness. However, as it will become more clear shortly, the algorithm neglects the important feature that a general purpose OS cannot guarantee perfect timing in packet sending due to other processes and timer granularity.

A more involved approach addressing the issue of implementing a rate-based congestion control is presented in [12] where a send loop is proposed to implement the rate-based TCP Friendly Rate Control (TFRC) algorithm. The proposed solution is shown in Algorithm 1.

Algorithm 1 Send loop proposed in RFC 3448

Let us define:

$$\Delta = \min\left(\frac{t_{ipi}}{2}, \frac{t_g}{2}\right) \tag{2}$$

and t_g as the o.s. timer granularity. The algorithm follows:

1) Send k-th packet at time t_k
2) Evaluate $t_{ipi,k} \leftarrow \frac{s}{r(t_k)}$ so that the k+1-th packet should be sent at time $t_{k+1} = t_k + t_{ipi,k}$
3) Check the system time t_{now}, evaluates Δ by using (3) and if $t_{now} > t_{k+1} - \Delta$:
   a) send the packet immediately
   b) otherwise, schedule a timer whose length is $t_{k+1} - t_{now}$
4) When the timer expires the algorithm restarts in 1.

The algorithm is based on the one proposed in [24] but it considers for the first time the uncertainty caused by the operating system on the duration of the inter-packet gaps, which are named *inter packet interval* t_{ipi} in [12]. In particular, the third step of the Algorithm 1 tries to counteract the effect of imprecise timer duration by sending a packet (step 3.a) without waiting for all the inter packet interval in the case this interval has elapsed except that for an amount equal to:

$$\Delta = \min\left(\frac{t_{ipi}}{2}, \frac{t_g}{2}\right) \tag{3}$$

We interpret the step 3.a as a heuristic aiming at anticipating the packet sending time to compensate when the sending times are delayed due to imprecise timers.

Moreover, an additional note in [12] considers the case when t_{ipi} is too small because the rate is high. In such cases authors recommend to send short bursts of several packets separated by intervals of the OS timer granularity.

TCP pacing is another technique aiming at spacing packets sending in order to mitigate burstiness when window-based congestion control protocols are used [5], [14], [18]. In fact, TCP produces a very bursty traffic when accessing high-speed networks that can lead to link underutilization and high packet losses in case router buffers are not large enough. Recently, it has been shown that sub-RTT time scale burstiness that are due to the nature of the TCP packet sending mechanism can lead to macroscopic effects on steady state bandwidth sharing [27]. TCP pacing evenly spaces a congestion window worth of packets in a RTT by scheduling timers whose length is equal to $RTT/cwnd$. The implementation issues affecting this technique have been studied in [16], [26]. In particular, [26] points out that software timer based approaches are not accurate enough when high rates need to be produced. The solution proposed in the paper is a module executed in kernel space, which inserts *dummy packets* between real packets in order to implement packet pacing. Dummy packets have to be later discarded by the switch where the network interface card (NIC)

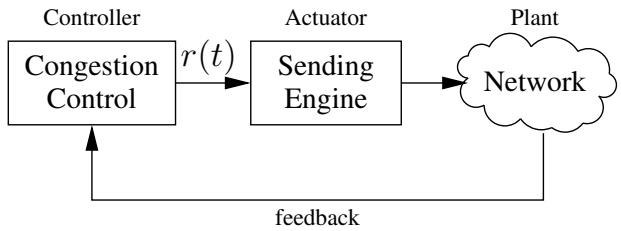

Figure 1. Sending Engine (send loop) which actuates the congestion control algorithm

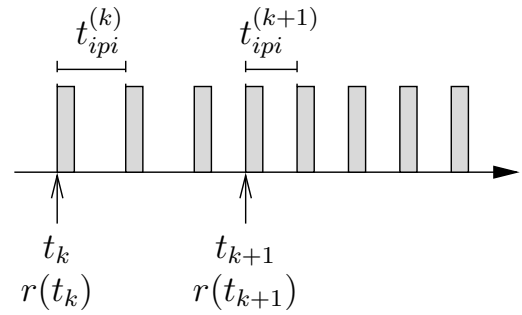

Figure 2. Rate-based packet sending: timers are scheduled to send packets at the specified rate

is connected. The proposed solution becomes very involved when multiple flows access the same link because in this case packet gaps length have to be recalculated accordingly [26]. In [16] authors propose a solution that needs an ad-hoc designed NIC and modifications to the operating system and to packet headers in order to be implemented.

III. Send Loop issues for rate-based applications

The TCP window-based congestion control evaluates and immediately sends the amount of data W when an ACK is received. In this case the sending of packets is ACK-clocked and there are no open implementations issues. On the other hand, in the case of rate-based congestion control algorithms a stream of packets has to be sent at the rate computed by the controller by scheduling packets to be sent at precise instants using timers. For this reason, in the case of rate-based congestion control, packets are sent using a send loop that is asynchronous *wrt* to the reception of ACK packets. Moreover, packet scheduling is affected by significant uncertainty due to the fact that the OS has to manage timers along with other processes, which introduces unpredictable delays in timer expiration instant. Figure 1 shows the block named *Sending Engine* that represents the send-loop machinery required to implement packet sending at the rate $r(t)$ computed by the controller. In the case of a window-based congestion control, the *Sending Engine* block implements the trivial task of sending the whole amount of data W immediately on ACK reception. On the other hand, in the case of rate-based control, the *Sending Engine* has the difficult task of providing a packet sending rate close to the one computed by the controller in the presence of timers affected by uncertain duration.

It is worth to notice that if window-based algorithms could be made not ACK-clocked such as in the case of TCP pacing, rate-based congestion control algorithms cannot be made ACK-clocked since packets must be sent at precise instants that are asynchronous wrt ACK reception.

We can assume without loss of generality that rate-based congestion control schemes evaluate the input rate every time a new feedback report (or ACK) is received by the sender. Assuming that the k-th feedback report is received at time t_k, the *send loop* has to schedule packets to be sent so that the resulting rate matches $r(t_k)$ in the time interval $[t_k, t_{k+1}]$. Let us define the set $P_k = \{(p_i, t_{k,i}) | 1 \leq i \leq n\}$ where $t_k = t_{k,0} < t_{k,1} < \ldots < t_{k,n} = t_{k+1}$ indicates that at time $t_{k,i}$ a packet of size p_i has to be sent. We define a packet scheduling policy to be *zero-bursty*, if it is allowed to schedule just one packet at once, that is, $\forall i \in \{1, \ldots n\}$, $\forall k : t_{k,i} \neq t_{k,i+1}$ and such that the packets in each interval are evenly spaced. It is important to notice that in order to schedule the packet p_i to be sent at time $t_{k,i}$ a timer will be set at time $t_{k,i-1}$ whose length is $t_{k,i} - t_{k,i-1}$.

Thus, we can say that in order to have a zero-bursty scheduling policy, given p_i and $r(t_k)$ we have to schedule n timers of equal lengths. It will soon become clear that the zero-bursty scheduling cannot be enforced for any given $r(t_k)$.

In first instance, the sender has to schedule a timer whose duration becomes smaller and smaller when the rate increases. However, timer durations are lower bounded by the operating system *timer granularity* t_g that depends on the frequency the CPU scheduler is invoked. By noting that typical values for t_g are in the order of $1-10ms$, (1) gives the maximum achievable rate:

$$r_{max} = \frac{s}{t_g} \tag{4}$$

Equation (4) implies that even with a timer granularity as low as 1 ms and a packet size of $1500\,B$ a maximum rate of $12\,Mbps$ is obtainable.

In second instance, the sending rate produced by packet gap-based algorithms is not accurate due to the fact that timers are not precise in a general purpose operating system [6].

Let us take a closer look at the way operating systems assign processes to the CPU. For sake of simplicity and without loss of generality, let us focus on round-robin FIFO schedulers which allocate the CPU to processes for a short amount of time, called time-slice or quantum Q, and then schedules the next process in the *wait queue*. Since not all processes make use of the whole quantum Q, we define $T \leq Q$ as the actual length of a quantum period. Moreover, let us assume that ρ is the offered load, and $1/\mu$ is the average program length in seconds. In such a scenario it has been found that the expected value of the waiting time of a process in the wait queue is given by [23]:

$$E[t_w] = \frac{\rho}{\mu}\left(\frac{1}{1-\rho} - e^{-\mu Q}\left(\frac{Q}{E[T]} - 1\right)\right)$$

For this reason, if a process schedules a sleep timer whose nominal duration is $\bar{t}$ seconds, the process will be actually assigned to the CPU again after $\bar{t} + t_w$ seconds. Therefore, the

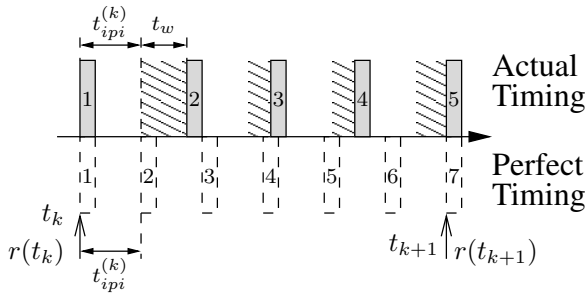

Figure 3. Perfect packet scheduling provides the desired rate, whereas the timers error t_w due to the operating system interaction induce performance degradation

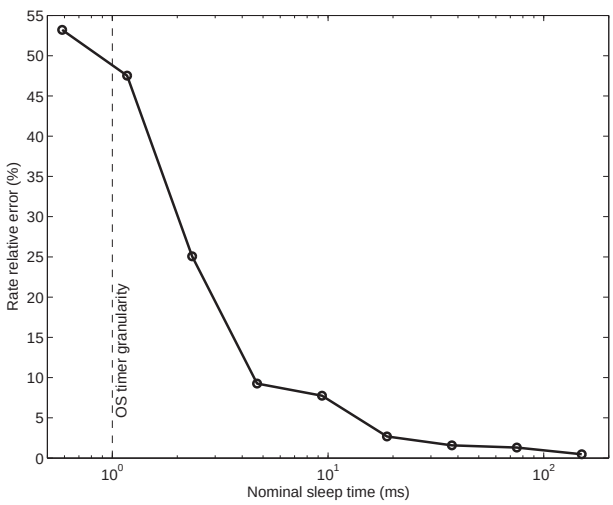

Figure 4. Input rate relative error as function of the nominal sleep time $\bar{t}$

actual packet sending rate produced by the send loop is affected by the Operating System load, which acts as a *disturbance* on the *send loop*. In particular, the effective rate r_e is given by:

$$r_e(t_k) = \frac{p}{\bar{t} + t_w} < r(t_k)$$

Figure 3 shows how the scheduled sleep timer of length $t_{ipi}^{(k)}$ is affected by the delay t_w, which significantly degrades the performance of a rate-based control.

In order to further illustrate these concerns, we have implemented a simple send loop under the Linux 2.6.19 operating system[1], which schedules a packet to be sent every $\bar{t}$ seconds, and we have logged the actual rate achieved for different nominal rates. Let us denote with $\overline{r}$ the nominal rate so that the sleep time is calculated as $p/\overline{r}$, whereas the relative error is evaluated as $100 \cdot (\overline{r} - r_e)/\overline{r}$. Figure 4 shows the effect of the sleep time $\bar{t}$ on the relative rate error. It clearly shows that when the nominal timer length approaches the operating system timer granularity the relative rate error increases up to 53%. It is worth noticing that when the tests were run, the system CPU was idle, so that the disturbance was due only to the operating system scheduler. Moreover, it is worth noting that the Linux scheduler offers advanced features designed to implement a low latency operating system such as kernel preemption, $o(1)$ complexity and dynamic task prioritization [7]. This is not the case with other Operating Systems, such as Symbian to name one, which is characterized by a timer granularity of $\sim 15\,ms$ [25].

For these considerations, it is necessary to design a mechanism able to counteract the uncertainty in timer expirations especially when timer durations are close to the OS timer granularity, as it happens in the case of high rates. In the next Section we design a feedback controller able to compensate the disturbance acting on the exact timer expiration due to OS timer granularity and load.

IV. The Rate Mismatch Controller

In this Section we propose a controller having the goal of producing an effective sending rate $r_e(t)$ that efficiently tracks the rate $r(t)$ computed by a rate-based congestion control algorithm. As we have already discussed, this goal is not trivial due to the fact that the code in charge of sending packets is executed by a CPU that is shared by other concurrent processes.

To this purpose, we start by the general model of a network congestion algorithm introduced in [20] and extended in [21]. The literature on modeling network congestion control and TCP congestion control is rich and a survey of it is beyond the scope of this work. We only say that a significant part of it is unduly complex and nonlinear, in contrast with the thoughts of Van Jacobson that in his cornerstone paper [15] states: "*A network is, to a very good approximation, a linear system. That is, it is composed of elements that behave like linear operators, integrators, delays, gain stages, etc*".

In [20] and [21], the fundamental dynamic elements to be considered when modeling network congestion control are controllers, integrators that model router buffers, and delays that model propagation and queueing delays. In particular, these elements are connected as in the block diagram shown in Figure 5 where the linear controller is modelled by the transfer function $G_c(s)$, the delay T_{fw} from the sender to the bottleneck queue is modelled by the transfer function $e^{-sT_{fw}}$, the delay T_{fb} from the bottleneck queue to the receiver and then back to the sender by the transfer function $e^{-sT_{fb}}$, and finally the bottleneck queue length $q(t)$ is modelled by the transfer function of an integrator, which is $1/s$. Figure 5 also shows the *Rate Mismatch Controller* (RMC) encircled by dashed lines which is here introduced in order to reject the disturbance $d_{cpu}(t)$, which models the effect on exact timing of packet sending. It is worth noting that the control scheme shown in 5 is made of two control loops: (1) the outer loop models the end-to-end rate-based congestion control executed between the sender and the receiver; (2) the inner loop models the feedback control that is designed for rejecting the disturbance on packet sending times. The rationale of using the inner control loop is to counteract the disturbance faster than the outer loop can do due to the fact that the outer loop is affected by network propagation delays.

[1]We have used a Linux Kernel compiled with a timer frequency of $1000\,Hz$, so that the operating system timer granularity is $1\,ms$.

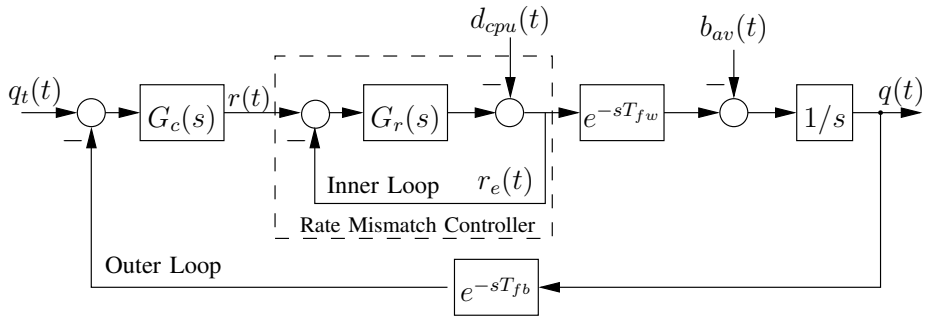

Figure 5. Block diagram of the overall control system

For the sake of simplicity, and in view of the fact that the controller has to be discretized and implemented in a code, we propose the simplest controller that is able to reject a step disturbance:

$$G_r(s) = \frac{k_r}{s} \tag{5}$$

It is important to remark that the dynamics of the inner loop can be made much faster than that of the outer loop. This is important in order to have an efficient rejection of the disturbance but also not to affect in a significant way the dynamics of the outer end-to-end control loop that, in this way, preserves its stability.

A. Discretization of the controller

The controller 5 must be discretized in order to be implemented. To discretize the controller, let us consider its output:

$$Y_r(s) = \frac{k_r}{s}(R_c(s) - R_e(s))$$

Thus by taking the inverse Laplace transform we obtain:

$$y_r(t) = k_r(\int_0^t r_c(\tau)d\tau - \int_0^t r_e(\tau)d\tau) \tag{6}$$

When we have described the model in the continuous time domain (Figure 5) we have assumed that the actual rate $r_e(t)$ was available as a feedback signal. However, in practice the send loop sends packets so that the integral of the actual rate $r_e(t)$:

$$d_e(t) = \int_0^t r_e(\tau)d\tau \tag{7}$$

which is the amount of data $d_e(t)$ that has been injected in the network until the time t , is already known.

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

$$y_R(t) = k_r(\int_0^t r_c(\tau)d\tau - d_e(t)) \tag{8}$$

One motivation of choosing a simple integrator controller is now clear: the variable `bytes_sent` $d_e(t)$ is available and updated every time a packet is sent and it is not affected by any measurement error.

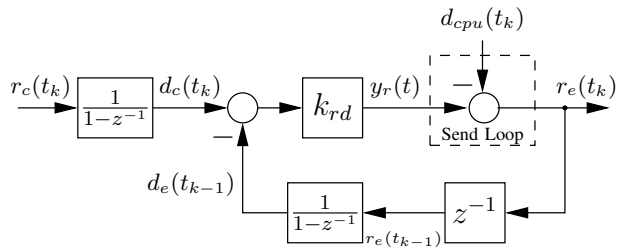

Figure 6. Rate Mismatch Controller

To complete the discretization of (8) we need to discretize the integral $\int_0^t r_c(\tau)d\tau$ that can be done by using for instance the Simpson's Rule:

$$d_c(t) = \int_0^t r_c(\tau)d\tau \rightarrow$$

$$d_c(t_k) = d_c(t_{k-1}) + (t_k - t_{k-1})\frac{r_c(t_k) + r_c(t_{k-1})}{2} \tag{9}$$

where t_k indicates the k-th sampling time. The discretization of $d_e(t)$ is straightforward:

$$d_e(t) = \int_0^t r_e(\tau)d\tau \rightarrow d_e(t_k) = d_e(t_{k-1}) + b_s(t_k) \tag{10}$$

where $b_s(t_k)$ is the amount of data sent in the k-th time interval.

Finally, it should be noted that the feedback variable $d_e(t_k)$ is delayed by one sample interval. In fact, when we are to send data at time t_k we know the amount of data sent until the previous sampling time t_{k-1}, i.e. $d_e(t_{k-1})$. The error is then evaluated as:

$$e(t_k) = d_c(t_k) - d_e(t_{k-1})$$

Thus, the discretized control is as simple as:

$$y_r(t_k) = k_r(d_c(t_k) - d_e(t_{k-1})) \tag{11}$$

The control action expressed by (11) can be intuitively interpreted as follows: a fraction of the amount of data that have not been sent at time t_k because of the disturbance will be sent at the time t_{k+1} thus being able to control the error. By considering equations (11), (9) and (10) the block diagram of the RMC represented in Figure 6 can be easily derived.

Proposition 1: A necessary and sufficient condition for the stability of the proposed controller is $0 < k_{rd} < 2$.

Proof: By computing the transfer function between $R_c(z)$ and $R_e(z)$ it results:

$$G_r(z) = \frac{R_e(z)}{R_c(z)} = \frac{k_{rd} z}{z - 1 + k_{rd}}$$

It is well-known that a linear discrete-time system is asymptotically stable if and only if all its poles lie in the unity circle of the complex plane. This turns out the condition that the only pole of the controller $z = 1 - k_{rd}$ must lie in the unity circle, i.e. $0 < k_{rd} < 2$. ■

Proposition 2: The proposed controller rejects the step disturbances $d_{cpu}(t) = 1(t)$.

Proof: By computing the transfer function between $D_{cpu}(z)$ and $R_e(z)$:

$$\frac{R_e(z)}{D_{cpu}(z)} = \frac{z - 1}{z - 1 + k_{rd}}$$

and considering that D_{cpu} is a step disturbance we can write:

$$R_e(z) = D_{cpu}(z) \frac{z - 1}{z - 1 + k_{rd}} = \frac{z}{z - 1 + k_{rd}}$$

Finally, it is sufficient to use the final value theorem to obtain the steady state value of the output due to the disturbance d_{cpu}:

$$r_e(\infty) = \lim_{k \to \infty} r_e(t_k) = \lim_{z \to 1} \frac{z - 1}{z} R_e(z) = 0$$

■

B. Send loop implementation

In the Section III we have described the heuristic employed by the send loop of the TFRC protocol to compensate disturbances on exact packet sending times due to CPU load and OS scheduler.

As opposed to Algorithm 1 that spaces packets in order to implement a rate-based control, we have proposed a control algorithm that steers to zero the error between the effective rate $r_e(t)$ produced by the send loop and the input rate $r_c(t)$ computed by the end-to-end rate-based control. To the purpose of implementing this control, we need to execute the send loop in an asynchronous thread every sampling time T_s. The sampling time is chosen as a fraction of the minimum round trip time RTT_m as follows:

$$T_s = \max(\frac{RTT_{min}}{N}, T_{s,min})$$

where $T_{s,min}$ is lower bounded by t_g.

The pseudo-code of the proposed send loop is reported in Algorithm 2. At each iteration of the infinite outer loop (line 1), the rate mismatch controller evaluates the data to be sent `(data_to_send)` by using the function `rmc(sending_rate)` and the inner loop (lines 7 to 16) sends a number of packets without exceeding the amount of `data_to_send` bytes (line 10). At this point the thread sleeps for T_s seconds and then the algorithm continues.

Algorithm 2 Pseudo-code of the proposed Send loop

```
1  while (running)
2  {
3    sending_rate = get_congestion_control_rate();
4    data_to_send = rmc(sending_rate);
5    bytes_sent = 0;
6
7    while(bytes_sent<=data_to_send)
8    {
9      packet = get_packet_from_tx_queue();
10     if(data_sent + size (packet) < data_to_send)
11       send(packet);
12     else
13       break;
14
15     bytes_sent = bytes_sent + size(packet);
16   }
17   rmc_update_data_sent(bytes_sent);
18   sleep(T_s);
19 }
```

As in the case of Algorithm 1, the timer duration T_s is affected by error due to OS timer granularity or, worse, it can happen that a context switch allocate the CPU to another process. The rate mismatch controller is able to compensate the effects of this disturbance as it will be shown in the next Section.

V. Experimental results

In this Section we present an experimental evaluation of the proposed controller. The experimental testbed has been set up using the `netem` kernel module [13], which allows WAN scenarios with given bottleneck capacities and delays be emulated. A dumb-bell topology has been considered in the following two cases: 1) a basic scenario where the rate computed by the end-to-end rate-based congestion control algorithm is held constant throughout all the experiment duration; 2) a scenario where the considered end-to-end rate-based congestion control is the well-known TFRC; in this case, the RMC controller has been implemented in the TFRC experimental code provided in [28].

The bottleneck queue length has been set equal to the bandwidth-delay product in all experiments. The round trip propagation time has been set equal to 50ms. The RMC gain has been set equal to 0.7 which provides a good trade-off between burstiness and reaction speed of the controller.

A. The case of constant sending rate $r_c(t)$

In order to evaluate the performances of the RMC we have implemented the send loop described in Section IV in a C user space application. We have considered constant sending rates $r_c(t)$ equal to $1\,Mbps$ or $10\,Mbps$ in two different scenarios:

- *Scenario 1*: the CPU load is close to zero;
- *Scenario 2*: the CPU load is increased by starting a busy-wait cycle in the time interval $[10, 30]$ s.

Figure 7 (a) and (b) (Figure 8 (a) and (b)) show the effective sending rate $r_e(t)$ in the case of a required constant rate

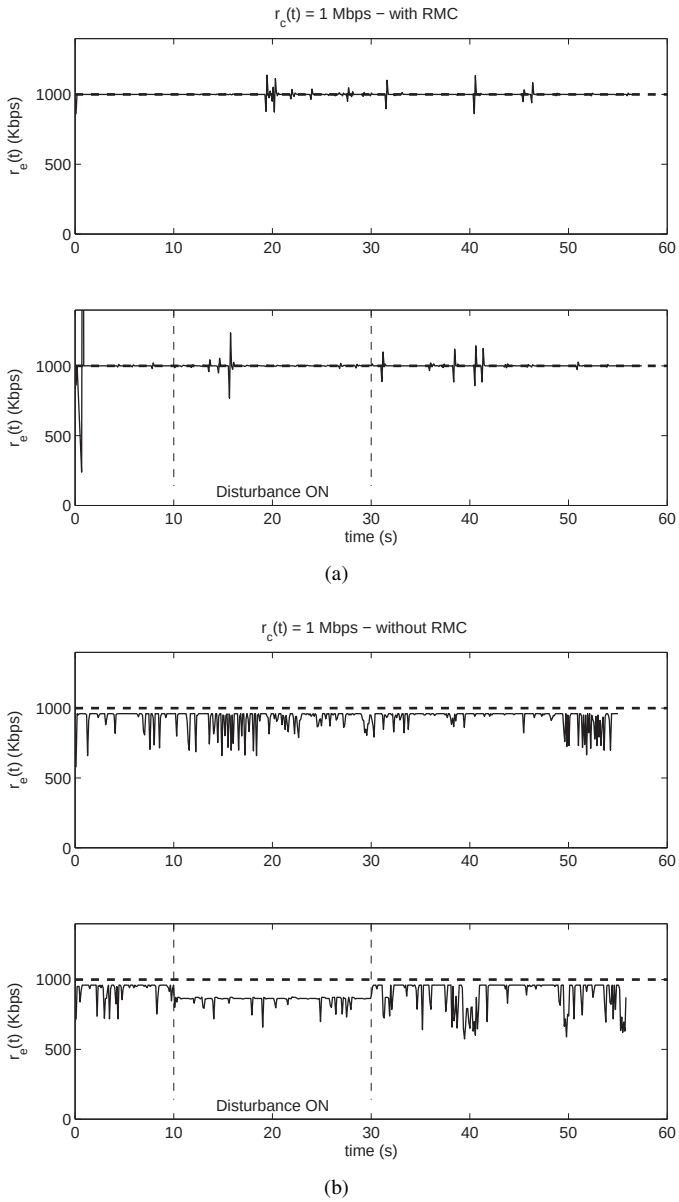

Figure 7. Effective rate $r_e(t)$ generated by send loops in the case of $r_c = 1\,Mbps$: (a) using the RMC and (b) not using the RMC

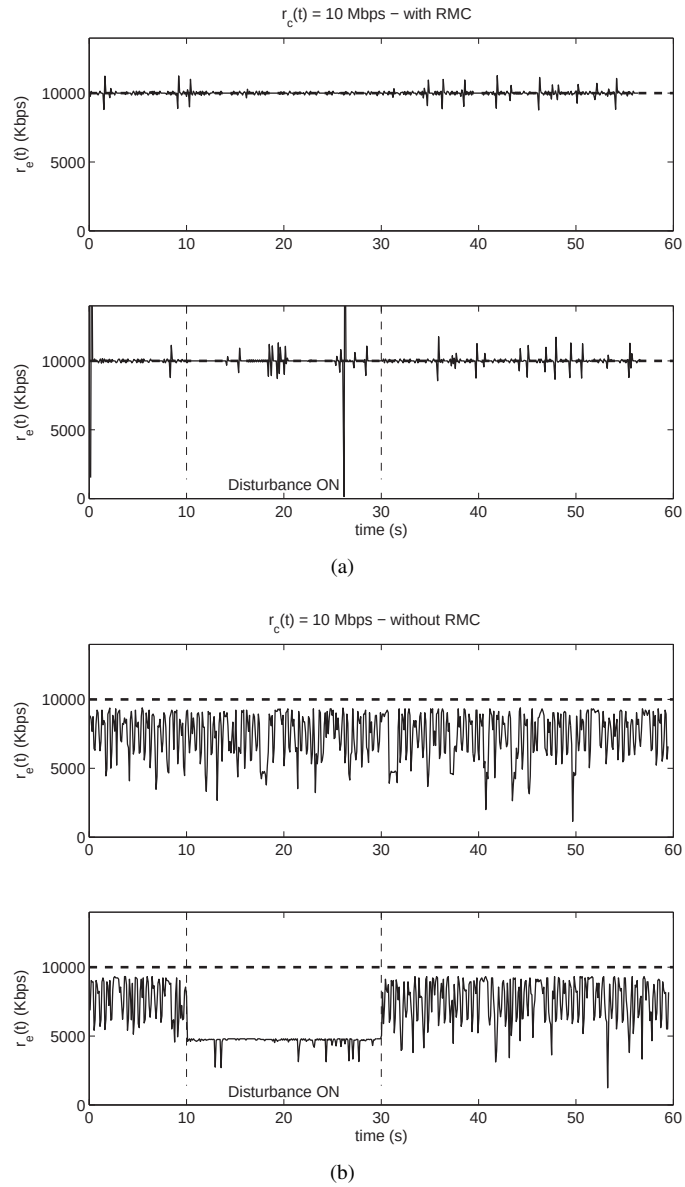

Figure 8. Effective rate $r_e(t)$ generated by send loops in the case of $r_c = 10\,Mbps$: (a) using the RMC and (b) not using the RMC

$r_c(t) = 1\,Mbps$ ($r_c(t) = 10\,Mbps$) when the RMC is/is not used, respectively. In particular, Figures 7 (a) and 8 (a) show that the mismatch between $r_e(t)$ and $r_c(t)$ is close to zero even when the "while 1" process generate a disturbance in the time interval $[10, 30]$s, thus proving the effectiveness of RMC in rejecting timer uncertainties. On the other hand, Figures 7 (b) and 8 (b) show that, in the absence of RMC and when the "while 1" process generate a disturbance in the time interval $[10, 30]$s, the effective sending rate exhibits a step-like decrease that, in the case of Figure 8 (b), is up to half of the desired sending rate. The absence of RMC also provokes a bursty sending rate.

We have also implemented the compensation heuristic used by TFRC and described in Algorithm 1 step 3.a in order to evaluate its effectiveness to reject disturbances on the send loop. We have found that the heuristic seems effective up to moderate bitrates; when the desired sending rate achieves the order of $100Mbps$ some issues starts to appear. Figure 9 compares the effective rate $r_e(t)$ obtained using the proposed RMC with the one obtained using the heuristic described in Algorithm 1 step 3.a when the desired sending rate is $r_c(t) = 100\,Mbps$. When the RMC is used, the effective rate produced by the send loop is very close to desired sending rate and the channel utilization is 100%. On the other hand, the effective sending rate does not match the desired sending rate when the heuristic is used and the channel utilization is 98%, i.e. there is a finite constant error between the desired and effective sending rate.

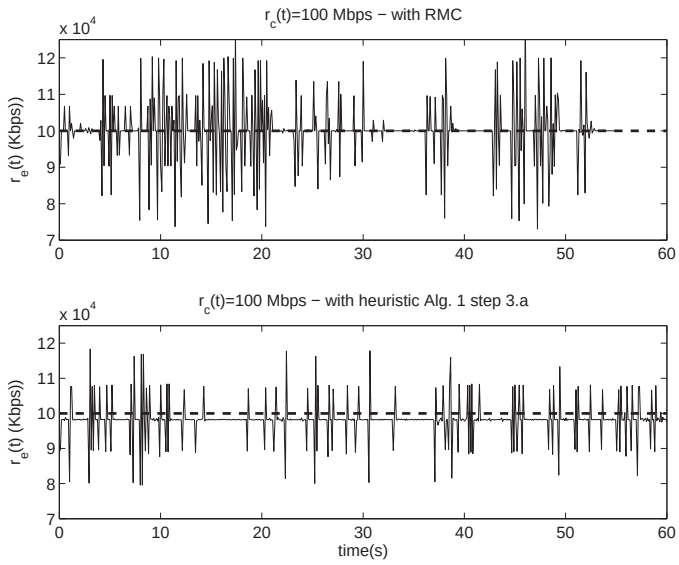

Figure 9. Effective rate produced in the case (a) RMC compensation is used, (b) compensation heuristic in Algorithm 1 step 3.a is used

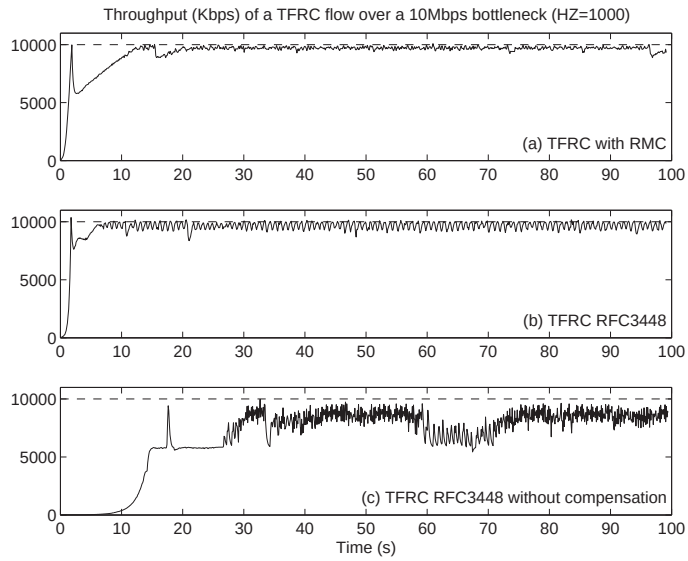

Figure 10. One TFRC flow in the cases: a) with the RMC; b) with the compensation specified in RFC 3448; c) without the compensation specified in RFC 3448 (Algorithm 1 step 3.a).

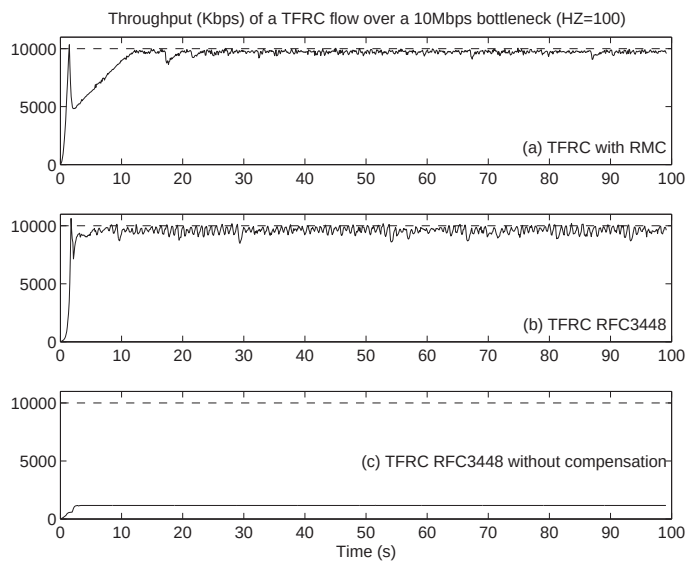

Figure 11. One TFRC flow in the cases: a) with the RMC; b) with the compensation specified in RFC 3448; c) without the compensation specified in RFC 3448 (Algorithm 1 step 3.a).

B. The impact of RMC on the TFRC protocol

In order to investigate the behaviour of the RMC controller in a scenario where the computed sending rate is not constant but is the outcome of an end-to-end rate-based congestion control such as the one modeled by the outer control loop shown in 6, we consider the TCP Friendly Rate Control (TFRC) protocol, which is the most known example of rate-based congestion control algorithms [12].

The Rate Mismatch Controller (RMC) has been implemented in the TFRC experimental code [28]. In the experiment, a TFRC flow has been injected through a dumb-bell topology with a bottleneck of $10\,Mbps$ and an RTT of $50\,ms$.

As we have mentioned in Section II, the TFRC includes a heuristic aiming at compensating errors on the scheduled packet sending times (Algorithm 1 step 3.a). Therefore, we evaluate the sending rate of a TFRC flow in the following cases: (a) TFRC with RMC; (b) TFRC with compensation described in Algorithm 1 step 3.a [12]; (c) TFRC without any compensation. Moreover, we consider two different values for the timer granularity, namely $1ms$ ($HZ = 1000$) and $10ms$ ($HZ = 100$). Achieved throughputs are shown in Figure 10 and 11, respectively.

Figure 10 shows results obtained when the OS has been compiled with a timer granularity equal to $1ms$. In particular, the TFRC with the RMC compensation provides a smooth sending rate that matches the available bandwidth. The TFRC with the RFC 3448 compensation provides a sending rate with a persistent ripple which is due to the heuristic of the mechanism. Interestingly, when no compensation is used, the TFRC exhibits high burstiness and achieves poor link utilization.

By considering the coefficient of variation $CoV = \sigma/\mu$ as an index to measure the burstiness of the sending rate, the TFRC with RMC provides a CoV=1.4%, the TFRC with RFC 3448 compensation provides a CoV=3.1%.

Figure 11, which refers to the case of an OS timer granularity equal to $10\,ms$ (coarse granularity), shows that without compensation of the send loop, the flow achieves a channel utilization as low as 10%.

VI. Conclusions

In this paper we have shown that the *send loop* required for implementing end-to-end rate-based congestion controls is affected by disturbances that have to be rejected. We have designed, implemented and tested a Rate Mismatch Controller

(RMC) that is able to produce an effective sending rate that matches the sending rate computed by a rate-based congestion control. We have shown that, when there is no disturbance rejection, even a constant desired sending rate turns into a bursty sending rate that is even unable to get a satisfactory channel utilization. We have also compared through experiments the heuristic proposed in [12] to reject disturbance, with the RMC. Results show that RMC works well in all considered scenarios, whereas TFRC seems not scale to high speed rates.

ACKNOWLEDGMENTS

This work has been partially supported by Financial Tradeware P.l.c.

REFERENCES

[1] Cooperative Association for Internet Data Analysis. http://www.caida.org/.
[2] Joost - Free online TV. http://www.joost.com/.
[3] Skype. http://www.skype.com/.
[4] YouTube. http://www.youtube.com/.
[5] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *Proc. IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 26–30, 2000.
[6] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.
[7] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005.
[8] Y.H. Chu, A. Ganjam, TS Eugene, S. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early deployment experience with an overlay based internet broadcasting system. In *Proc. USENIX Annual Technical Conference 2004*, Boston, MA, USA, June 2004.
[9] L. De Cicco, S. Mascolo, and V. Palmisano. A Mathematical Model of the Skype VoIP Congestion Control Algorithm. In *Proc. IEEE Conference on Decision and Control '08, to appear*, Cancun, Mexico, December 9–11, 2008.
[10] L. De Cicco, S. Mascolo, and V. Palmisano. Skype Video Responsiveness to Bandwidth Variations. In *Proc. ACM NOSSDAV 2008*, Braunschweig, Germany, May 28–30, 2008.
[11] Luigi Alfredo Grieco and Saverio Mascolo. Adaptive rate control for streaming flows over the internet. *ACM Multimedia Systems Journal*, 9(6):517–532, June 2004.
[12] M. Handley, S. Floyd, and J. Pahdye. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, Proposed Standard, January 2003.
[13] S. Hemminger. Network Emulation with NetEm. In *Proc. Linux Conf Au 2005*, Otago, New Zeland, January 23–28, 2005.
[14] J.C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proc. ACM SIGCOMM '96*, Stanford University, CA, USA, August 28–30, 1996.
[15] V. Jacobson. Congestion avoidance and control. *ACM Comput. Commun. Rev.*, 18(4):314–329, 1988.
[16] K. Kobayashi. Transmission timer approach for rate based pacing TCP with hardware support. In *Proc. International workshop on Protocols for Long Distance Networks (PFLDnet '06)*, Nara, Japan, February 2–3, 2006.
[17] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. RFC 4340, Proposed standard, March 2006.
[18] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge. Paced TCP for High Delay-Bandwidth Networks. In *Proc. IEEE Globecom '99*, Rio de Janeiro, Brazil, December 5–9, 1999.
[19] Jin Li. Peer-Assisted Delivery: the Way to Scale IPTV to the World. ACM NOSSDAV 2007, panel session, June 4–5, 2007.
[20] S. Mascolo. Congestion control in high-speed communication networks using the Smith principle. *Automatica*, 35(12):1921–1935, 1999.
[21] S. Mascolo. Modeling the Internet congestion control using a Smith controller with input shaping. *Control engineering practice*, 14(4):425–435, 2006.
[22] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168, Proposed standard*, September 2001.
[23] Philip J. Rasch. A queueing theory study of round-robin scheduling of time-shared computer systems. *J. ACM*, 17(1):131–145, 1970.
[24] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism forrealtime streams in the Internet. In *Proc. IEEE INFOCOM '99*, New York, NY, USA, March 21–25, 1999.
[25] Jo Stichbury. *Games on Symbian OS: A Handbook for Mobile Development*. John Wiley & Sons, Inc., 2008.
[26] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and Evaluation of Precise Software Pacing Mechanisms for Fast Long-Distance Networks. In *Proc. International workshop on Protocols for Long Distance Networks (PFLDnet '05)*, Lyon, France, February 3–4, 2005.
[27] Ao Tang, Lachlan L. H. Andrew, Krister Jacobsson, Karl H. Johansson, Steven H. Low, and Håkan Hjalmarsson. Window flow control: Macroscopic properties from microscopic factors. In *Proc. of IEEE INFOCOM*, Phoenix, AZ, 15-17 Apr 2008.
[28] J. Widmer. Implementation of the TCP-Friendly Congestion Control Protocol (TFRC). http://www.icir.org/tfrc/code/.
[29] X. Zhang, J. Liu, B. Li, and T.S.P. Yum. CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming. In *Proc. IEEE INFOCOM 2005*, Miami, FL, USA, March 13–17, 2005.