Implementing Rate-based Protocols

Luca De Cicco and Saverio Mascolo

***Abstract*—End-to-end rate-based congestion control algorithms are advocated for audio/video transport over the Internet instead of window-based protocols. Once the congestion controller computes the sending rate, all rate-based algorithms proposed in the literature schedule packets to be sent spaced at intervals that are equal to the inverse of the desired sending rate. In this paper we show that such an implementation exhibits a fundamental flaw. In fact, scheduling the 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. To overcome this problem, the 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 [11] 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 packets are sent in bursts 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. Sending a burst of $W(T_k)$ packets is simple to be implemented but it is not appropriate both from the point of view of router buffers and from the point of view of users of real-time applications.

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. The sending rate r_c 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) [16]. Once the sending rate r_c 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_c.

Several rate-based schemes have been proposed in literature for the transport of multimedia streams [8], [9], [13], [18] 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 [15], also employs TCP even though the use of TFRC had been long debated [5], [23]. Finally, Skype audio/video implements proprietary congestion control mechanisms at application layer over the UDP protocol [6], [7].

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 (OS). The problem here is that the unpredictable CPU load, which is due to other processes that share the CPU, prevents exact timing in packet sending. 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. In order to overcome this issue, the *Rate Mismatch Controller* (RMC) is proposed aiming at producing an effective sending rate $r_e(t)$ that efficiently tracks the rate $r_c(t)$ computed by a rate-based congestion control algorithm. 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 emphasizes the effects of the interaction between the OS and the application.

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 propose the RMC 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 [8], [9], [13], [18]. 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 [18], where a rate-based congestion control protocol named *Rate Adaptation*

Luca De Cicco is post-doc at Dipartimento di Elettrotecnica ed Elettronica, Politecnico di Bari, Via Orabona 4, Italy `ldecicco@poliba.it`

Saverio Mascolo is a Faculty member of Dipartimento di Elettrotecnica ed Elettronica, Politecnico di Bari, Via Orabona 4, Italy `mascolo@poliba.it`

Algorithm 1 Send loop proposed in RFC 3448

Let us define:

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

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

1) Send k-th packet at time t_k
2) Evaluate $t_{ipi,k} \leftarrow \frac{p}{r_c(T_k)}$ 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 (2) 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.

Protocol (RAP) is proposed. In that paper, authors suggest to evenly space packets at intervals equal to the *inter-packet interval* (IPI) t_{ipi}, which is computed as follows:

$$t_{ipi} = \frac{p}{r_c(t)} \quad (1)$$

where p is the packet size and $r_c(t)$ is the rate determined by the congestion controller. Equation (1) implies that the highest the rate the closest the packets should be sent in order to reflect the instantaneous sending rate $r_c(t)$. At first glance, this simple algorithm seems to be able to provide a sending rate that matches $r_c(t)$ 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 [9] 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.

The algorithm is based on the one proposed in [18] but it considers for the first time the uncertainty of the inter-packet intervals due to the fact that the send loop process shares the CPU with other processes. 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 Δ given by (2). 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 [9] 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 [2], [10], [14]. 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 [22]. 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 [12], [21]. In particular, [21] 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) 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 [21]. In [12] 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 $W(T_k)$ 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 rate $r_c(t)$ 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.

Figure 1 shows a high-level model of the network congestion control machinery that is made of the following components: 1) a generic congestion control algorithm, acting as the controller, that decides the appropriate sending rate $r_c(t)$ based on the feedback, being it implicit or explicit, provided by the network; 2) a block named *Sending Engine* (or send-loop) representing the actuator of the control system, required to implement packet sending at rate $r_c(t)$; 3) the network that represents the considered plant.

In the case of a window-based congestion control, the *Sending Engine* block implements the 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.

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 $r_c(T_k)$ during the time interval $[T_k, T_{k+1}]$. Let us define the *packet sending policy* as the set

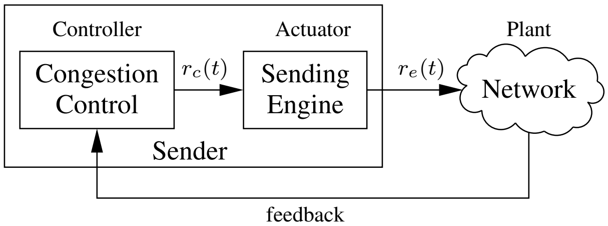


Figure 1: Sending Engine (send loop) which actuates the congestion control algorithm

$P_k = \{(p_i^{(k)}, t_i^{(k)}) | 0 \leq i \leq n_k\}$ with $T_k = t_0^{(k)} < t_1^{(k)} < \ldots < t_n^{(k)} = T_{k+1}$, indicating that at time $t_i^{(k)}$ a packet of size $p_i^{(k)}$ 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, $\forall i \in \{1, \ldots n_k\}$, $\forall k : t_i^{(k)} \neq t_{i+1}^{(k)}$ and such that the packets in each interval are evenly spaced. It is important to notice that in order to schedule the packet $p_i^{(k)}$ to be sent at time $t_i^{(k)}$ a timer will be set at time $t_{i-1}^{(k)}$ whose length is $t_i^{(k)} - t_{i-1}^{(k)}$.

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

The simple send loop employed by [18], [9] does not take into account some key implementation issues.

In first instance, the send-loop has to schedule a timer whose duration becomes smaller and smaller when the rate increases. However, timer durations are lower bounded by the OS *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:

$$r_{max} = \frac{p}{t_g} \tag{3}$$

Equation (3) implies that even with a timer granularity as low as 1 ms and a packet size $p = 1500\,B$ a maximum rate r_{max} 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 OS [3].

Let us take a closer look at the way the CPU scheduler assign processes to the CPU. When a process is running but the CPU is not assigned to it, the process is said to be in the wait queue. The amount of time a process spends in the wait queue before obtaining again the CPU is defined as waiting time t_w and it depends on the CPU load [17]. For this reason, if a process schedules a sleep timer whose nominal duration is $\bar{t}$ seconds, the process will be actually assigned again to the CPU after $\bar{t} + t_w$ seconds.

Therefore, the actual packet sending rate produced by the send loop is affected by the OS load, which acts as a *disturbance* on the *send loop*. In particular, the effective rate r_e is determined as $r_e = s/(\bar{t} + t_w)$, which is less than r_c.

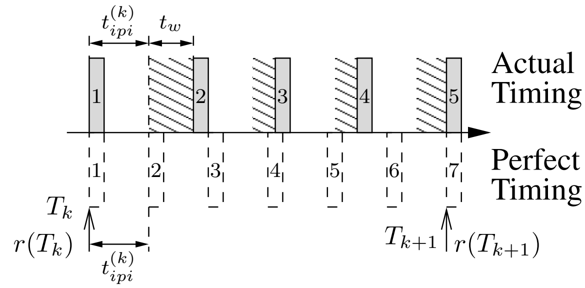


Figure 2: Perfect packet scheduling provides the desired rate, whereas the timers error t_w due to the operating system interaction induce performance degradation

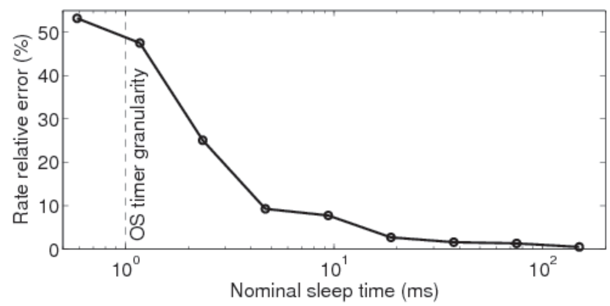


Figure 3: Input rate relative error as function of the nominal sleep time $\bar{t}$

Figure 2 shows how the scheduled sleep timer of length $t_{ipi}^{(k)}$ is affected by the waiting time t_w, which significantly degrades the performance of the rate-based control.

It is important to notice that the entity of the disturbance that is due to the interactions with the operating system depends on the particular implementation of the OS process scheduler and on the way timers are handled.

In order to further illustrate these concerns, we have implemented a simple send loop under the Linux 2.6.19 OS[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 r_c denote the nominal rate so that the sleep time is calculated as s/r_c, whereas the relative error is evaluated as $100 \cdot (r_c - r_e)/r_c$. Figure 3 shows the effect of the sleep time $\bar{t}$ on the relative rate error. In particular, when the nominal timer length $\bar{t}$ approaches the OS timer granularity t_g 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, the Linux scheduler offers advanced features designed to implement a low latency OS such as kernel preemption, o(1) complexity and dynamic task prioritization [4]. This is not the case with other operating systems, such as Symbian to name one, which is characterized by a timer granularity of $\sim 15\,ms$ [20].

It is worth to notice that the concerns we have discussed

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

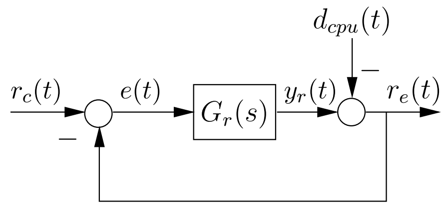


Figure 4: Block diagram of the system

here, have been also recently addressed in a thread on the DCCP IEFT working group mailing list [19] and no solution has been found yet.

For these considerations, in order to actuate the sending rate evaluated by the congestion controller, it is necessary to design a mechanism that is 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 $r_c(t)$ 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 and managed by the OS.

Figure 4 shows the block diagram of the feedback loop in which the *Rate Mismatch Controller* (RMC) is introduced in order to reject the disturbance $d_{cpu}(t)$, which models the effect on exact timing of packet sending. The effective rate $r_e(t)$ is measured and compared to the desired rate $r_c(t)$ to give the error $e(t) = r_c(t) - r_e(t)$ that acts as the input of the RMC. The Rate Mismatch Controller G_r evaluates the rate $y_r(t)$ the send loop should provide in order to counteract the disturbance $d_{cpu}(t)$.

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 control law that is able to reject a step disturbance, that is an integral control law with gain k_r. Therefore, the transfer function of the controller is:

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

so that the output of the controller is:

$$y_r(t) = k_r \int_0^t e(\tau)d\tau \tag{4}$$

In the following the control law (4) is first discretized and then implemented in the proposed send loop.

A. Discretization of the controller

The control law (4) must be discretized in order to be implemented in the send loop. By substituting $e(t) = r_c(t) - r_e(t)$ in (4) it turns out:

$$y_r(t) = k_r(\int_0^t r_c(\tau)d\tau - \int_0^t r_e(\tau)d\tau) \tag{5}$$

When we have described the model in the continuous time domain (Figure 4) we 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 $r_e(t)$:

$$d_e(t) = \int_0^t r_e(\tau)d\tau \tag{6}$$

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 (5) and (6), we obtain:

$$y_r(t) = k_r(\int_0^t r_c(\tau)d\tau - d_e(t)) \tag{7}$$

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. To complete the discretization of (7) we need to discretize the integral $\int_0^t r_c(\tau)d\tau$ that can be done by using the backward Euler method:

$$d_c(t_k) = d_c(t_{k-1}) + (t_k - t_{k-1})r_c(t_k) \tag{8}$$

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

$$d_e(t_k) = d_e(t_{k-1}) + b_s(t_k) \tag{9}$$

where $b_s(t_k)$ is the amount of data effectively 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:

$$y_r(t_k) = k_r(d_c(t_k) - d_e(t_{k-1})) \tag{10}$$

The control action expressed by (10) 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 d_{cpu} will be sent at the time t_{k+1} thus being able to control the error. By considering equations (10), (8) and (9) the block diagram of the RMC represented in Figure 5 can be easily derived.

Let us consider the closed loop dynamical system depicted in Figure 5: it is easy to show that the closed loop response $r_e(t)$ can be made faster by increasing k_r . However, the

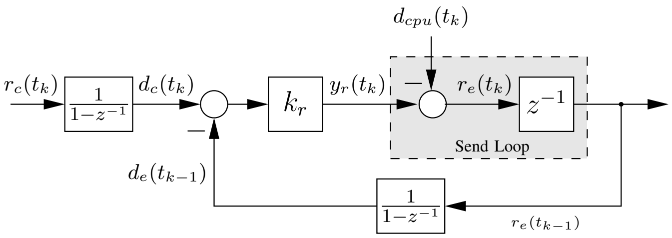


Figure 5: Digital Rate Mismatch Controller

value of k_r is in practice upper bounded because of stability constrains as shown in the following Proposition.

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

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

$$\frac{R_e(z)}{R_c(z)} = \frac{k_r z}{z - 1 + k_r}$$

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 $z = 1 - k_r$ must lie in the unity circle, i.e. $0 < k_r < 2$. ■

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

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

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

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

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

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 this Section 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:

$$T_s = \max\left(RTT_{min}/N, T_{s,min}\right)$$

Algorithm 2 Pseudo-code of the proposed Send loop

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

where $T_{s,min}$ is lower bounded by t_g. It is worth to notice that the lower T_s, the lower is the burstiness generated by the send loop.

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 $y_r(t_k)$ (`data_to_send`) by using the function `rmc(r_c)` 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.

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. However, 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 running on two different versions of the Linux Kernel, namely 2.6.19 and 2.6.28 implementing the last two Linux schedulers. 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. The send loop proposed in RFC 3448 [9] has been carefully implemented by also taking into account the heuristic in step 3.a of Algorithm 1. In particular we have implemented two versions of the send loop described in Algorithm 1: the first one, named *RFC 3448 - No bursts,* does not take into account the issue due to the granularity t_g and it keeps sending one packet per iteration spaced by $t_{ipi} = s/r$ even when t_{ipi} is less than the timer granularity t_g; in the second implementation, named *RFC 3448 - Bursts*, when $t_{ipi} < t_g$, t_{ipi} is set to $\min(s/r, t_g)$ and a burst whose length is $b = r \cdot t_g$ is sent as proposed[2] in [9].

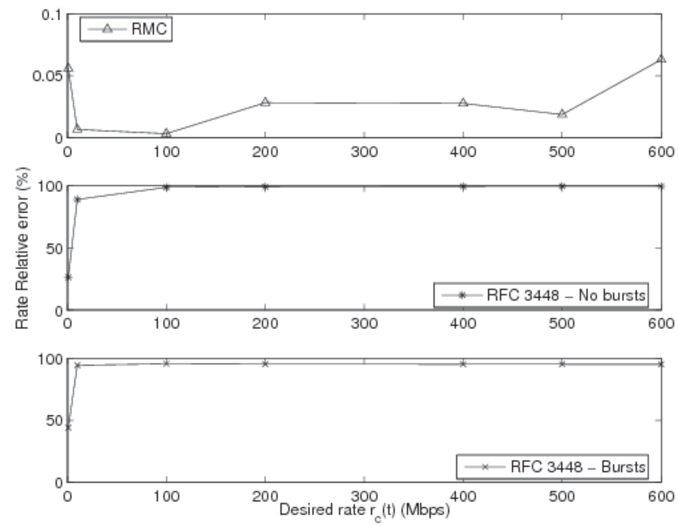
[2]When $t_{ipi} < t_p$, [9] suggests: "TFRC may send short bursts of several packets separated by intervals of the OS timer granularity". The size of the burst to be sent is not specified. However sending a burst of $b = r \cdot t_g$ bytes in t_g seconds should produce the rate r.

The three send loops have been tested considering constant sending rates r_c in the set $R = \{1, 10, 100, 200, 400, 600\} Mbps$. For what concerns the only tunable parameter k_r of the proposed controller we have run experiments by letting $k_r \in [0.1, 1.9]$. Although results are not reported in this paper due to space limitation, experiments confirmed that the disturbance is rejected regardless the value of $k_r \in]0, 2[$, as we found in Proposition 2.

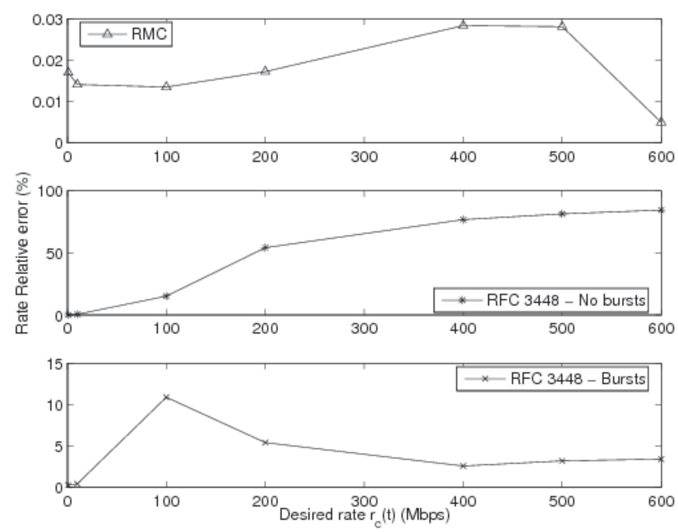
We have compared the proposed controller with the two send loops *RFC 3448 - No Bursts* and *RFC 3448 - Bursts* both in the case a Linux Kernel 2.6.19 or a Linux Kernel 2.6.28.1 is employed. The duration of the experiments is 60 s. In the time interval $[20, 40]$ s the CPU load is increased to $\sim 100\%$ by starting five CPU bound (busy-wait) processes in parallel with the send loop.

Figures 6 (a) and (b) show the average relative error computed as $(r_c - E[r_e(t)])/r_c \cdot 100$ when $r_c \in R$: the figures show that RMC is able to provide a relative error less than 0.06% , that is a channel utilization of 99.94%, regardless the value of r_c or kernel version employed. On the contrary, *RFC 3448 - No Bursts* is not able to contain the disturbance and the relative error is found to be very high: when kernel 2.6.19 is employed it can be as high as $\sim 99\%$, whereas when 2.6.28.1 is used it goes up to 90%. Let us now consider *RFC 3448 - Burst*: Figure 6 shows that even with sending bursts as large as $r_e \cdot t_g$ the relative error can be as high as 95% in the case the Kernel 2.6.19 is used, or up to 4% when Kernel 2.6.28.1 is employed.

Finally, Figures 7 (a) and (b) report the effective rate evolution when $r_c = 500\, Mbps$. When the RMC is used, the effective rate produced by the send loop matches the desired sending rate and a smooth input rate is produced.



(a) Linux Kernel 2.6.19 with HZ=100



(b) Linux Kernel 2.6.28 with HZ=100

Figure 6: Rate relative error comparison

VI. CONCLUSIONS

In this paper we have shown that the *send loop* required for implementing end-to-end rate-based congestion control in high-speed networks is affected by disturbances that have to be rejected. To this purpose, we have designed, implemented and tested a Rate Mismatch Controller (RMC) that is able to produce an effective sending rate that matches the computed sending rate. The experimental results have shown that, when no measures are taken to reject the disturbance, the rate relative error can be as high as 99% in the case of high-speed rates, whereas RMC is always able to provide rate relative errors that are less than 0.06%.

REFERENCES

[1] Cooperative Association for Internet Data Analysis. http://www.caida.org/.

[2] 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.

[3] 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.

[4] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005.

[5] 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.

[6] 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.

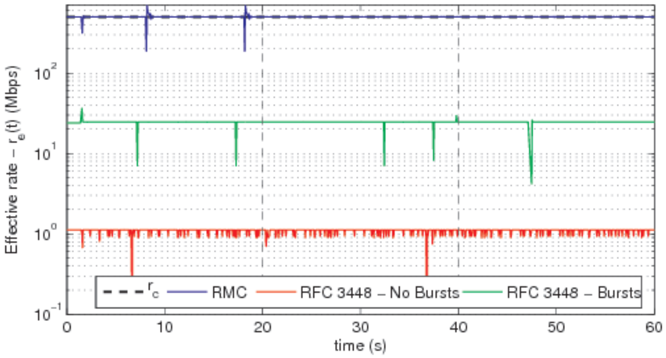
[7] L. De Cicco, S. Mascolo, and V. Palmisano. Skype Video Responsiveness to Bandwidth Variations. In *Proc. ACM NOSSDAV 2008*, Braunschweig, Germany, May 28–30, 2008.

[8] 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.

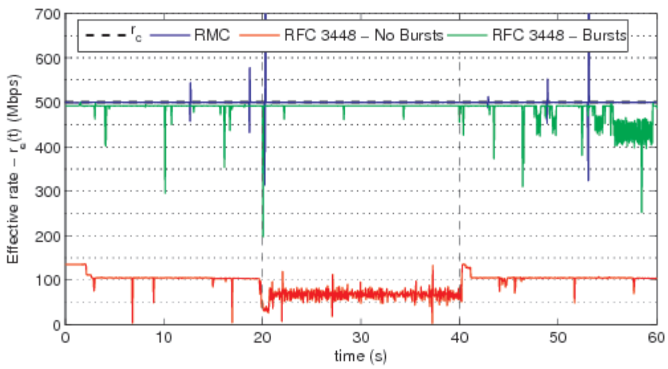
[9] M. Handley, S. Floyd, and J. Pahdye. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, Proposed Standard, January 2003.

[10] 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.

[11] V. Jacobson. Congestion avoidance and control. *ACM Comput. Commun.*



(a) Linux Kernel 2.6.19 with HZ=100



(b) Linux Kernel 2.6.28 with HZ=100

Figure 7: Effective rate $r_e(t)$ when required rate is $500\,\mathrm{Mbps}$

Rev., 18(4):314–329, 1988.

[12] 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.

[13] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. RFC 4340, Proposed standard, March 2006.

[14] 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.

[15] Jin Li. Peer-Assisted Delivery: the Way to Scale IPTV to the World. ACM NOSSDAV 2007, panel session, June 4–5, 2007.

[16] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168, Proposed standard*, September 2001.

[17] Philip J. Rasch. A queueing theory study of round-robin scheduling of time-shared computer systems. *J. ACM*, 17(1):131–145, 1970.

[18] 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.

[19] G. Renker. [dccp] [rfc] dccp ccid-3: High-res or low-res timers? in DCCP IETF WG [electronic discussion list]. Archived at: http://kerneltrap.org/mailarchive/linux-netdev/2008/11/15/4105494, November 2008.

[20] Jo Stichbury. *Games on Symbian OS: A Handbook for Mobile Development*. John Wiley & Sons, Inc., 2008.

[21] 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.

[22] 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.

[23] 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.