Towards a sender-based TCP friendly rate control (TFRC) protocol

Guillaume Jourjon, Emmanuel Lochin, Patrick Sénac

Towards a Sender-Based TCP Friendly Rate Control (TFRC) Protocol

Guillaume Jourjon, Emmanuel Lochin and Patrick Sénac.

***Abstract*—Pervasive communications are increasingly sent over mobile devices and personal digital assistants. This trend is currently observed by mobile phone service providers which have measured a significant increase in multimedia traffic. To better carry multimedia traffic, the IETF standardized a new TCP Friendly Rate Control (TFRC) protocol. However, the current receiver-based TFRC design is not well suited to resource limited end systems. In this paper, we propose a scheme to shift resource allocation and computation to the sender. This sender-based approach led us to develop a new algorithm for loss notification and loss-rate computation. We detail the complete implementation of a user-level prototype and demonstrate the gain obtained in terms of memory requirements and CPU processing compared to the current design. We also evaluate the performance obtained in terms of throughtput smoothness and fairness with TCP and we note this shifting solves security issues raised by classical TFRC implementations.**

***Index Terms*—Transport, TFRC, Protocol Implementation.**

I. Introduction

THE recently standardized DCCP protocol [1] is perceived as the most efficient mechanism to carry multimedia traffic. DCCP can apply multiple congestion control mechanisms, and identifies TCP-Friendly Rate Control (TFRC) as congestion control ID #3 (DCCP/CCID3) [2]. TFRC is a congestion control mechanism for unicast flows operating in a best-effort Internet environment [3]. TFRC reproduces the TCP window-based congestion control mechanism through an equation model of the TCP equivalent throughput. The smooth rate variation, induced by this congestion control mechanism, makes it a good candidate for the delivery of an efficient transport service to client-server multimedia and continuous stream applications. However, in such a media streaming scenario, if multimedia servers are powerful processing and communication engines, this is not usually the case of mobile clients. Indeed, these clients are resource-limited end-systems and are much more sensitive to communication and system processing while focusing on application layer.

Therefore the lightening of recurrent communication processing is a critical issue for increasing the performance and autonomy of mobile end systems. One of the main costs of the TFRC mechanism comes from the periodic computation of both the RTT and the loss rate of data carried by a connection. In particular, RFC 3448 [3] proposes the loss rate estimation to be done on the receiver side. A classical receiver-based solution achieves a periodic estimation of the loss event rate before sending it to the sender. This computation requires maintenance of a loss event history data structure. Such a receiver-based solution does not comply with the capacities and resource constraints (i.e. in terms of energy consumption and overall computational performance) of light mobile receivers (e.g. PDAs, mobile phones) which are increasingly populating the Internet.

RFC 3448 suggests that this computation could also be done on the sender-side: *"It would be possible to implement a sender-based variant of TFRC where the receiver uses reliable delivery to send information about packet losses and the sender computes the packet-loss rate and the acceptable transmit rate"*. Indeed, a sender-based architecture would allow to ease future enhancements of the TFRC protocol. For instance, TCP, which is sender-based, allows to enable the deployment of novel protocol variant (such as CUBIC inside GNU/Linux kernel) as it concerns only a sender modification. Thus, in a future deployment of TFRC, a sender-based variant will ease the deployment of novel proposals.

In this paper, we develop this idea by specifying and evaluating the design of a sender-based implementation of the TFRC congestion control mechanism. In our proposal, TFRC flow's receiver just returns simple and light feedback packets to the sender by using a SACK-oriented mechanism [4] that insures the reliable transfer of theses feedback packets. This scheme is known to be robust to lossy channels while not entailing heavy and complex error control mechanisms [4]. Moreover, the proposed sender-based approach is more robust to selfish receivers as all the key operations are located in the sender side. Some solutions to secure TFRC from selfish receivers have been proposed in [5] using RTSP [6]. Another sender-based solution has been proposed in [2] where the receiver sends back loss event intervals to the sender. These two solutions, as it will be explained later, remain more complex than our new proposal.

This paper is structured as follows: section II introduces the context of this study and provides some background information. Section III gives insights into the design of the new congestion control protocol architecture. Section IV compares the performance of the proposed congestion control protocol with respect to the standard TFRC implementation. We quantify the benefits of our proposal in terms of algorithmic processing and communication load in section V. Finally, section VI provides some conclusions and future directions.

Part of the results was presented at IEEE ICC 2007, Glasgow, UK, 2007. This paper extends this previous version and presents a significant contribution in terms of performance evaluation of the algorithm proposed and in particular concerning the efficiency, throughtput smoothness and intra-protocol fairness of this proposal.

Author affiliations: Guillaume Jourjon is with the NICTA, Emmanuel Lochin and Patrick Sénac are with the Université de Toulouse, DMIA, ISAE.

Contacts: guillaume.jourjon@nicta.com.au, emmanuel.lochin@isae.fr, patrick.senac@isae.fr

II. Context and Related work

TFRC estimates the equivalent TCP sending rate X from equation (1). This equation depends on the mean packet size s and two periodically processed parameters: the packet loss-event rate p and the round trip time RTT. RTO refers to the TCP retransmission time-out value which is usually a linear function of the RTT.

$$X = \frac{s}{(RTT \cdot \sqrt{\frac{p \cdot 2}{3}} + RTO \cdot \sqrt{\frac{p \cdot 27}{8}} \cdot p \cdot (1 + 32 \cdot p^2))} \quad (1)$$

During the initialization phase, TFRC acts as TCP does during the slow-start algorithm. This slow-start phase also occurs during the transfer after the RTO time-out expires. This phase is followed by a congestion avoidance phase as soon as the receiver detects a loss. At this step, TFRC needs to periodically estimate the loss event rate, p, in order to compute the sending rate X. The receiver evaluates the packet loss rate by a sliding window-based loss-history structure. This structure stores the eight most recent loss-event intervals and process the loss-event rate with a low path filter that smoothes the loss event variation. A loss event and its related interval of packets is defined as one or more lost packets during a duration of at least one RTT [3]. In other words, several packets lost during an RTT define a single loss event and the duration of a loss interval is greater than or equal to the RTT. The algorithm used at the receiver side is given in Fig. 1.

```
ReceivePacket() {
  Add packet to packet history;
  p_new = new value of packet loss rate;
  if (p_new > p_old){
    feedback timer expiration;
    do CreateFeedback();
  }
}
CreateFeedback() {
  compute average packet loss rate;
  calculate measured receive rate;
  prepare and send feedback packet;
  restart feedback timer;
}
```

Fig. 1. Original algorithm of the receiver

Two main issues can be identified in the receiver-based algorithm. Firstly, the receiver must continuously maintain and update the loss event history data structure. The management of this data structure is an undesirable processing and memory overhead for resource limited mobile receivers. Secondly, the receiver has to continuously process the loss-event rate and send it to the sender, at least once per RTT, and as soon as it observes a loss-event rate increase. Once again, this processing load squeezes the remaining processing capacity. Moreover, such a receiver-based implementation cannot guarantee that selfish receivers do not try to trick the sender by deliberately sending a smaller loss event rate in an attempt to get higher bandwidth [5]. Our solution requires fewer and simpler modifications to the TFRC header and algorithm than the proposal in [5]. Another sender-based solution has been proposed in [2] where the receiver sends back loss event intervals to the sender. To the best of our knowledge, this solution has never been either tested or implemented. In comparison to our proposal, this solution is supposed to be closer to the original algorithm but the receiver remains more complex since it has to maintain a structure able to distinguish a loss from a loss event.

III. Design

This section presents the design of our sender-based TFRC protocol named TFRC$_{light}$. The design of this protocol is based on the shifting of the loss-rate estimation to the sender side. We identify and propose several changes entailed by this shifting, mainly in the feedback packet structure and in the data structures managed by the receiver. The aim of our new TFRC protocol architecture and design is to reduce the receiver load. We discuss in this section the design of TFRC$_{light}$ by first presenting the problems that resulted from shifting packet-loss-rate estimation. Then, we define and experimentally validate efficient solutions to these problems.

A. Notification of packet loss

In the traditional receiver-based TFRC, the receiver has to periodically send feedback information to the sender. These feedback messages contain two parameters that allow the sender to estimate the current RTT value. These parameters are respectively (1) the timestamp of the last packet received (`Last Timestamp`) and (2) the amount of time elapsed between the receipt of the last packet and the generation of the feedback (`Processing Time`). We present these fields of the TFRC header in Fig. 2.

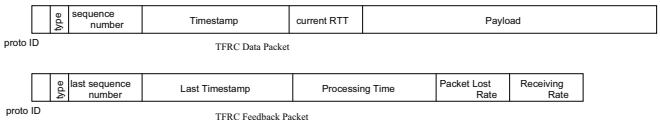


Fig. 2. TFRC headers for data and feedback packets

Moreover, feedback packets also contain information about the packet loss rate (`Packet Loss Rate`) and the received throughput (`Receiving Rate`) as processed by the receiver. In TFRC$_{light}$, the packet loss rate is no longer processed and returned by the receiver. Nevertheless, the receiver remains the only entity able to detect the loss of a packet and to notify the sender about this loss.

In order to perform this notification, we propose the maintenance of a compact and light data structure at the receiver. This data structure is a simple bits vector (i.e. a SACK vector) that describes, from a given packet number, the distribution of packets received and lost. In other words, if a given packet is received, the bit is set to 0 otherwise 1. This vector is periodically sent to flow source. Such a data structure leverage on the SACK mechanism used when some degree of reliability is needed, and gets the benefit of the redundancy offered by successive SACK vectors. Therefore and in this case, our approach does not entail any additional data structure at the receiver. Thus, our approach delivers two services for the price of one that are: congestion and error control.

When the SACK vector sending period is lower than the duration covered by the SACK vector, this vector offers redundancy that contributes to the reliable delivery of loss information. The value of the feedback packet sending period will be discussed in the next section. The right vector length can be chosen by considering that the sender-based and receiver-based implementation should react similarly to packet losses. Indeed, as defined in [3], the sender no-feedback timer expires after $4 * RTT$. Where RTT, is the exponentially weighted moving average of the round trip time processed by the sender and sent in each packet. A SACK-based mechanism is intrinsically robust to a maximum period of data losses equivalent to the vector range. Then, the loss vector length should cover at least:

$$4 * RTT * PacketSendingRate \; packets$$

Where $PacketSendingRate$, is the sending rate computed by the receiver as the received packet rate. In order to reproduce the no-feedback timer behaviour of the standard receiver based version of TFRC, the loss information vector length must be dynamically recomputed with a period of $4 * RTT$.

The data structure used to managed the SACK vector is a circular buffer, with a pointer keeping track of the most recently received packet. In the next section, we first consider a simple initial scheme for managing this structure. Then, from the issues raised by this scheme, we will propose a solution that conforms to the standard TFRC behaviour.

The message headers for the simple initial scheme are given in Fig. 3. Where in the feedback header, the SACK structure replaces the packet loss rate and in the the data header, we added the `nbSeq sync` in order to provide reliability in a future work.

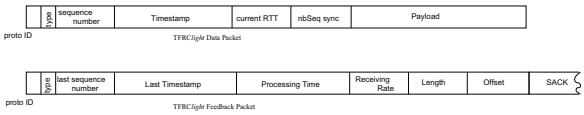


Fig. 3. Data and Feedback packet headers for a first version of TFRC$_{light}$

B. Loss event definition in TFRC$_{light}$

Although the previously introduced data and protocol data unit structures are necessary for implementing an efficient sender-based TFRC protocol, they are not sufficient. Indeed, the loss history structure is based on the loss event definition given in section II. Let's remind that a loss event is defined as the detection of one or more lost packets during one RTT. For keeping track of loss events, the receiver needs the receiving time of each packet to detect if lost packets belong to the same loss event interval.

Since the sender and the receiver cannot maintain a synchronous behaviour, the simple SACK structure previously introduced does not allow the sender to construct an accurate loss event history structure even if feedback packets are sent every RTT. Indeed, without a careful design, in certain cases, a loss event may be falsely detected. In Fig. 4, we give an illustration of such false detection. The time axis represents the data-packet arrival time. We also show on this axis the times, t_n, when the receiver sends feedback. As an example, below this axe, we show the SACK vectors associated to three successive feedback messages. At t_1, the feedback message reports two losses represented by the two bits set in the `SACK` field. The `Offset` is equal to 100.

In the original TFRC, a timer of RTT time units should have been triggered at the estimated receiving time of the lost packet with the sequence number of 106. This timer range is represented in Fig. 4 by the two-way arrows. At t_2, when the receiver sends its second feedback packet, the SACK vector `Offset` is now equal to 112 and as the RTT period is expired, a loss event should have been detected. At this time, the traditional TFRC algorithm closes the previous loss interval and restarts a new one from packet number 119 (i.e. the first lost packet following the RTT duration). Finally at t_3, the losses reported for packets 125 and 127 belong to the previous loss event as the RTT timer expired at packet number 130. Since no other packet is lost after this expiration, there is no new loss event. The problem of false detection can potentially result from an interpretation as a loss event of this third feedback with `Offset` field which is equal to 124 and its two marked bits in the vector.

As shown in Fig. 4, the TFRC mechanism is supposed to see two loss events associated with two successive loss event intervals of at least one RTT duration. In TFRC$_{light}$, if we merely shift the packet loss rate estimation, and rely on a simple and direct interpretation of SACK vector information, since there is no information about the estimated time of the packet loss, and the sender and receiver are not synchronous, the TFRC mechanism will see three loss events. Indeed, it will receive three disjoint feedback messages (one per RTT) with a non-null `SACK` field. Therefore, a simple logical interpretation of these feedbacks leads to the identification of three loss event instead of only two.

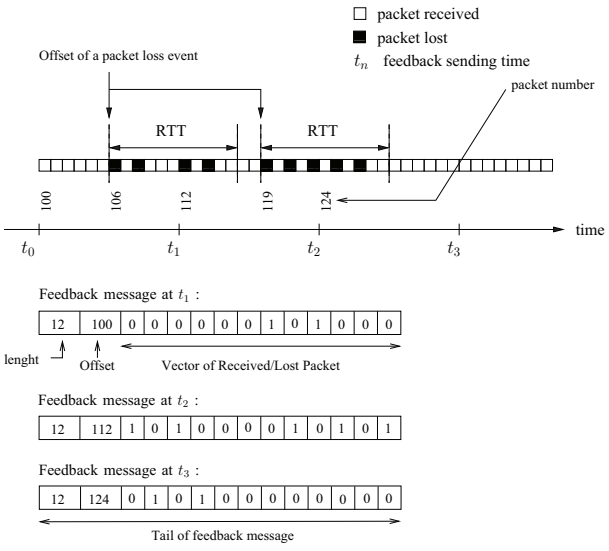
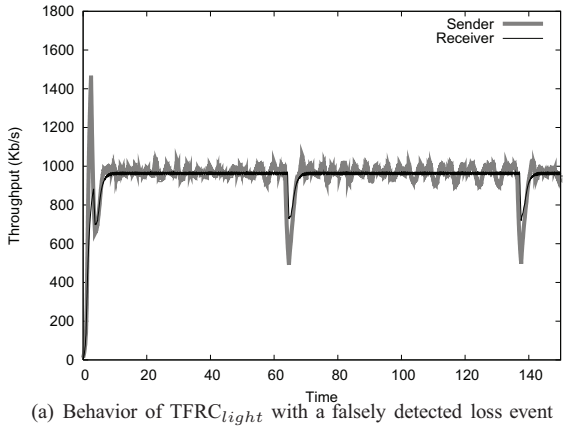


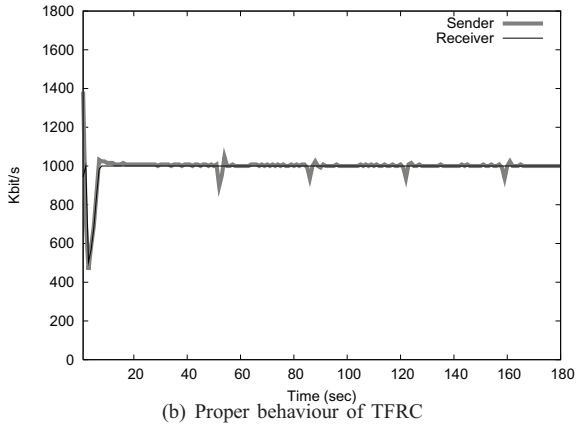
Fig. 4. Illustration of loss event misinterpretation issues

Fig. 5 presents the impact on the resulting sender behaviour of such at false detection issue. We give in this figure

the instantaneous throughput measured at the sender and at the receiver. Fig. 5(a) shows the resulting throughputs of a TFRC$_{light}$ with a bad interpretation of loss events. This real experiments involve an architecture with two nodes that generate traffic and are connected with wired link of $1Mbit/s$ and $RTT = 100ms$. In Fig. 5(a), TFRC$_{light}$ detects five loss events just after the slow-start phase (between $t = [0, 10]$)[1]. However a correct implementation of TFRC would have detected four loss events only as illustrated in 5(b).



(a) Behavior of TFRC$_{light}$ with a falsely detected loss event



(b) Proper behaviour of TFRC

Fig. 5. Comparison of TFRC$_{light}$ with a false detection and a traditional TFRC implementation in a network with a bandwidth of $1Mbit/s$, and an RTT=$100ms$

As a result, when a new loss event occurs (i.e. $t = 63s$ and $t = 137s$), the sender will decrease its emission rate more than needed. In Fig. 5(a), this behaviour induces the two large rate dips. Such an important throughput decrease is explained by the way the loss history structure is built. Indeed, as the mechanism gets successive loss events, the corresponding entries in the loss history structure will be filled with loss intervals shorter than they should be. Resulting in a reduction of the processed weighted average loss interval length and increase the loss event rate. Then, this loss-event rate increase causes an excessive reduction of the sending rate as given by equation (1).

In order to solve this issue we propose the following modifications.

[1]Observed by the addition of a memory variable inside the core protocol

1) New receiver algorithm: At the receiver side the structure remains similar to the one presented in the previous section. The algorithm used by the receiver side is shown in Fig. 6.

```
ReceivePacket(){
  Add packet to received packet;
}
CreateFeedback(){
  calculate measured receive rate;
  prepare and send feedback packet;
  restart feedback timer;
}
```

Fig. 6. Receiver algorithm

In this proposal, the receiver is no longer responsible for computing the packet loss rate. This algorithm supposes the existence of a new structure that records the arrival or loss of packets in order to allow SACK vectors to be built.

2) Modification at the sender side: In order to detect a loss event at the sender side, the server has to set up a structure that stores a timestamp of the packet sent. This structure is identical to the one that traditional receiver-based TFRC receivers use to compute the packet-loss rate, except that instead of keeping trace of the packet-arrival time, this new structure stores the packet-sending time.

Based on this new structure the sender is now able to detect loss events from a sender perspective by considering the sending time of the packets reported as lost in the received SACK vectors. Furthermore because the sender keeps track of packets sending time, the `TimeStamp` field in both data and feedback headers is no longer needed. Fig. 7 gives the resulting new structure of the TFRC$_{light}$ headers associated with the data and feedback packets.

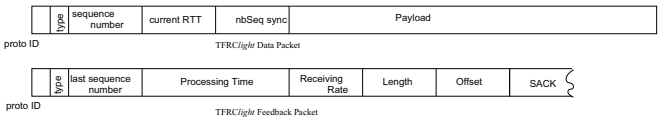


Fig. 7. Reconsidering TFRC$_{light}$ headers once the false detection of loss events problem is solved

3) Translation from Loss History to Loss Events: With the new data structure managed by the sender, the sender is now aware of the sending time of each packet. This information, combined with the received SACK vectors, allows the sender to process the packet loss rate as detailed in Fig. 8.

```
for(int i=0; i<lenghtACK; i++)
{
  if(vector[i]==0)
    add Packet(offset+i) loss History;
    p_new=new value of packet loss rate;
  else
    process loss history to loss event;
}
compute average packet loss rate;
```

Fig. 8. Algorithm for the SACK vector analysis at the sender

In section 5.2 of RFC 3448, the authors explain how to build loss events from the loss history. This operation needs:

- S_{loss}: the sequence number of the lost packet;
- S_{before}: the sequence number of the last packet to arrive such that $S_{before} < S_{loss}$;
- S_{after}: the sequence number of the first packet to arrive such that $S_{loss} < S_{after}$;
- T_{before}: the reception time of S_{before};
- T_{after}: the reception time of S_{after}.

In the presented solution, the sender is not aware of T_{before} and T_{after}. Nevertheless, the sender must estimate the arrival time of S_{loss}. In our proposal, we use sending times, instead of arrival times, to build loss events. These sending times are corrected by the following factor, which the sender evaluates whenever it receives feedback packets from the flow's receiver (where X_{sent} and X_{recv} are respectively the sending and receiving rates):

$$\alpha = \frac{X_{sent}}{X_{recv}}$$

The determination of the new event is accomplished in the same way as in the original TFRC except that the time reference is no longer the arrival time but is now the sending corrected by the factor α. Based on this new loss event, a loss interval is built and stored in the loss history structure managed by the sender. Then, the sender uses the same weighted sliding window algorithm as described in [3] in order to compute the packet loss rate from a weighted moving average of the loss event interval.

4) Discussion: As feedback messages are not systematically sent when a loss is detected, we recommend that feedback messages be sent at least one per RTT.

IV. Validation of TFRC$_{light}$

In this section, we present an evaluation of our proposal from several experiments over an emulated network. We have implemented a user level prototype of TFRC$_{light}$ in Java and evaluated the TFRC$_{light}$ prototype over a simple testbed composed of two end-systems and a network emulated by a FreeBSD/Dummynet pipe [7]. The main interest to develop such protocol in Java language is to easily port our implementation over Java compliant mobile devices and plug on top of this framework a streaming application[2].

We compare TFRC$_{light}$with TCP and TFRC. In the rest of this section, we use metrics as proposed in [8], [9], [10] to study through representative examples, the behaviour of TFRC$_{light}$.

A. Evaluation Strategy

The performance evaluation of TFRC$_{light}$ has been achieved regarding four criteria:

- Efficiency (in terms of throughput obtained);
- Intra-protocol fairness;
- TCP-friendliness;
- Stability (in terms of instantaneous throughput oscillations).

First, we provide the definitions of these metrics, then in the next section, we evaluate TFRC$_{light}$ according to these metrics. Finally we quantify our contribution in terms of CPU and memory usage.

1) Efficiency (Throughput): In [8] the authors define the efficiency of their protocol as the aggregate throughput of all the concurrent flows. Here, we use a normalised definition to our study.

$$E = \frac{\sum_{i=1}^{n} \overline{x_i}}{C} \tag{2}$$

Suppose there are n TFRC$_{light}$ flows in the network crossing a bottleneck of $C\ Mbit/s$. We denote $\overline{x_i}$, the throughput of the i^{th} flow. Then, the equation (2) represents the percentage of used bandwidth.

2) Intra-protocol fairness: The fairness metric represents how flows share fairly the bandwidth. In order to quantify this, the commonly used method is the $max - min$ fairness [9]. In this method the lowest throughput is maximised. In the following part of this section, since there is only one bottleneck in all experiments, we will use the Jain's fairness criteria [10] in order to measure this characteristic of TFRC$_{light}$. Therefore, this fairness is given by the equation (3).

$$F = \frac{(\sum_{i=1}^{n} \overline{x_i})^2}{n \sum_{i=1}^{n} \overline{x_i}^2} \tag{3}$$

where in this case $\overline{x_i}$ is the average throughput of the i^{th} TFRC$_{light}$ flow and n is the number of flows competing for the bandwidth. F is always less than or equal to 1. If $F = 1$, then all flows have the same throughput.

3) TCP friendliness: TCP-friendliness is nowadays subject to discussion among the networking community. In particular, some researchers claim that, from different point of views, this qualification for a flow is not a meaningful criterion for the service providers [11]. In this study, we used a metric following the axiom that a flow is TCP-friendly if *the non-TCP source obtains a long-run term average sending rate not larger that the one TCP would have obtained under the same circumstances*. This results in evaluating the TCP-friendliness with the equation (4):

$$T_1(X) = \frac{\frac{1}{n}\sum_{i=1}^{n} \overline{x_i}}{\frac{1}{m}\sum_{i=1}^{m} \overline{y_i}} \tag{4}$$

where X is the protocol being studied, $\overline{x_i}$ the average throughput of the i^{th} X flow, n the number of X flows, $\overline{y_i}$ the average throughput of the i^{th} TCP flow and m the number of TCP flows. In this formula if T_1 is less than 1 then the non-TCP flow is TCP-friendly, if T_1 is equal to 1 then we have an ideal friendliness and finally if T_1 if higher than 1 then the non-TCP flow overruns TCP.

4) Stability (oscillations): The last metric in use in this section is the throughtput smoothness criteria. TFRC is renowned for being a good candidate for multimedia traffic due to the smoothness of its delivered throughput.

In order to quantify this throughtput smoothness, we con-

[2]These algorithms have been implemented inside the Chameleon protocol available here http://nicta.com.au/people/jourjong/chameleon_protocol/

sider the average throughput for each time unit interval. For each of these intervals, we compute the standard deviation of the throughput for each flow [12] and obtain the following metric equation (5). This index corresponds to the sample standard deviation normalised by the average throughput.

$$S = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{\overline{x_i}}\sqrt{\frac{1}{m-1}\sum_{j=1}^{m}(x_i(k)-\overline{x_i})^2}\right) \quad (5)$$

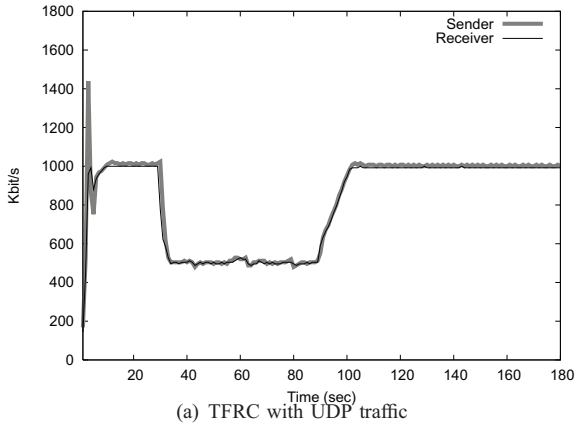
where $\overline{x_i}$ is the average throughput of the i^{th} TFRC$_{light}$ flow, n is the number of flows, $x_i(k)$ is the throughput of the i^{th} TFRC$_{light}$ flow for the k^{th} time interval and m is the number of time intervals.

B. General Behaviour of the proposal

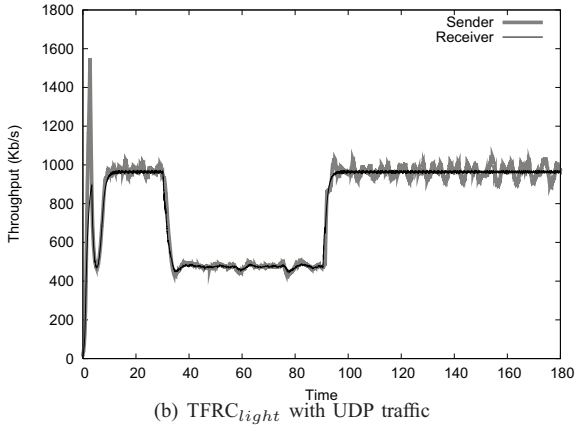
For all experiments, the bandwidth and the RTT were respectively set to $1Mbit/s$ and $100ms$. In both Figs. 9, we report the sending/receiving instantaneous throughputs measured respectively at the sender/receiver sides. The results of our experiments show that our sender-based protocol has the same behaviour as traditional receiver based TFRC implementations.

We made several measurements to validate this new architectural design and report in this section only a representative sample. It is always difficult to compare the performance of a real implementation and a simulated one since the simulation reproduces an ideal case without the overhead introduced by real measurements. Nevertheless, we show that the TFRC$_{light}$ receiver throughput is as stable as the ns-2 receiver throughput[3]. Concerning the sender throughput, TFRC$_{light}$oscillates more than ns-2 TFRC implementation. This can be explained by the overhead introduced by our user level TFRC$_{light}$ implementation.

In the experiment illustrated in Fig. 9, we introduced a concurrent UDP flow with a rate of $500Kbits/s$ between $t = [30sec, 90sec]$. This test aims to verify the responsiveness of TFRC$_{light}$ compared to ns-2 TFRC. In Fig. 9, due to their packets being multiplexed with a non-responsive UDP flow, both TFRC and TFRC$_{light}$ brutally decrease their rate during the UDP flood. Furthermore, both implementations react the same way to the losses induced by the UDP flow. When the UDP flow stops, both implementations respond similarly. Eventually, we can conclude from this scenario that the modifications proposed and implemented in TFRC$_{light}$ result in a behaviour similar to ns-2 TFRC.



(a) TFRC with UDP traffic



(b) TFRC$_{light}$ with UDP traffic

Fig. 9. TFRC and TFRC$_{light}$ with a network bandwidth of $1Mbit/s$, an RTT=$100ms$ and introduction of an UDP flow at $t = [30s, 90s]$

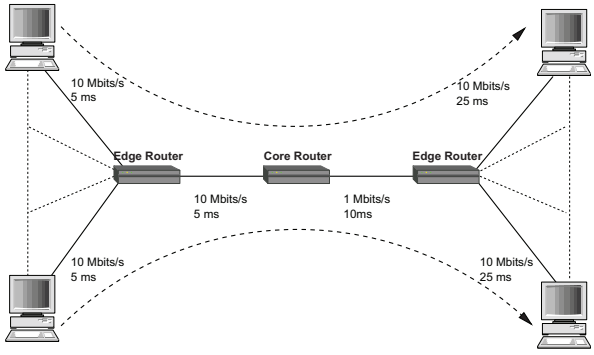


Fig. 10. Topology

C. Efficiency, Fairness and Stability of the proposal

In the set of experiments discussed in this section, we consider only TFRC$_{light}$ flows and measure different criteria. The topology of the network is displayed on Fig. 10.

In this topology, we made the number of TFRC$_{light}$ flows varying from 1 to 4 following two patterns. These two patterns differ in terms of flows' duration. Indeed, in the first pattern, every flow starts together but does not have the same duration as depicted in Fig. 11. In the second pattern, the starting and stopping time of each flow is the same. Thanks to these two different patterns, we aim to study the long run behaviour of our proposal and its reactiveness when flows leave the network.

1) Different Stopping Times: Fig. 11 represents the perceived throughput at the receiver side. This throughput is computed using a time sliding window of one second as explained in [13].

In Fig. 12, we show that our proposal equally shared the bandwidth between flows. The difference observed during the first period of the experiment can be explained by two main

[3]The ns-2 TFRC implementation is used as reference as *BSD and GNU/Linux implementations are still experimentals at the time of this study. These measurements are only use to verify that our prototype behaves as the standard TFRC reference chosen.

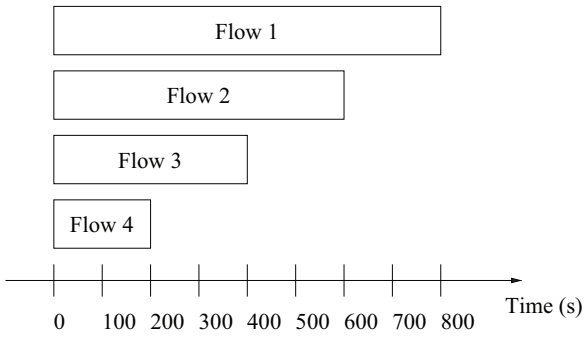


Fig. 11. Stopping times of the different flows

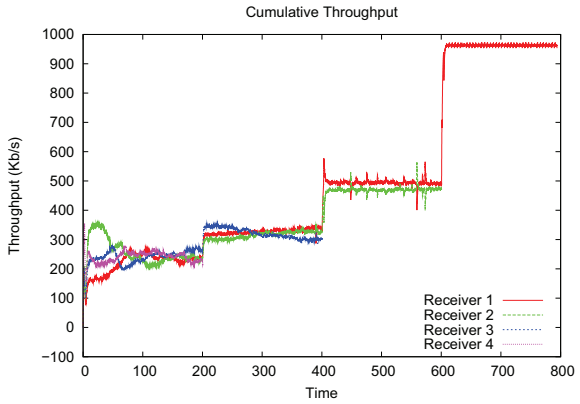


Fig. 12. Receiving throughput of the different flows

reasons. First, our implementation was in Java language, therefore the start of each flow is conditioned by the Java virtual machine performance. As a result a slight shifting between the start might occur. Second, all the flows experience their first loss event at different times. Following the increase and variance in buffering delay some flows have more difficulty in reaching the throughput equilibrium.

The behaviours represented in Fig. 12 are confirmed by the values of the previously introduced metrics displayed in Table I. These results have to be compared to what TCP would experience in the same condition as given in Table II.

TABLE I
PERFORMANCE METRICS OF TFRC$_{light}$ FLOW IN THE SCENARIO WITH DIFFERENT STREAMING DURATION

	0-200s	200-400s	400-600s	600-800s
Efficiency	0.949	0.974	0.961	0.958
Stability (oscillations)	0.137	0.043	0.031	0.035
Intra-protocol fairness	0.996	0.999	0.999	1

2) Long-term Behaviour: We present in this section the characteristics of our proposal in the case of a long-run communication. Indeed, as underlined in [14], studying TFRC as to be accomplished in the case of long term behaviour since TFRC models TCP once the connection is well established. In order to process this analysis, we perform the same experiment as the previous one but without different stopping times.

TABLE II
PERFORMANCE METRICS OF TCP FLOW IN THE SCENARIO WITH DIFFERENT STREAMING DURATION

	0-200s	200-400s	400-600s	600-800s
Efficiency	0.964	0.965	0.965	0.975
Stability (oscillations)	0.023	0.079	0.167	0.218
Intra-protocol fairness	0.993	0.999	0.999	1

TABLE III
RESULT OF THE LONG-RUN EXPERIMENT

	TFRC$_{light}$
Efficiency	0.965
Stability (oscillations)	0.058
Intra-protocol fairness	0.999

As expected, the results for the long run behaviour of TFRC$_{light}$ were a stabilised adjustment of the first test-period of the previous set of experiments. As a result, TFRC$_{light}$ is more stable in the long-run experiments than in short-run experiment. In the same way, TFRC$_{light}$ reached the equilibrium and therefore the intra-fairness property was enforced. Concerning the efficiency metric, TFRC$_{light}$ is more efficient in the long term behaviour study than in the previous one due to the fact that the equilibrium is reached compared to the time period $0 - 200s$ in the previous experiment. This is explained by the nearly equal to one intra-fairness metric.

D. TCP-Friendliness

In the following experiments, we show that the proposed sender-based TFRC remains TCP friendly. The results of the TFRC friendliness property are given in Table IV. These measurements give the average throughput observed at the receiver after 200 seconds of transfer. We have driven the first experiment with 5 TFRC$_{light}$ flows only. We also studied the multiplexing behaviour of TFRC$_{light}$ flows with TCP and TFRC flows. The results sum-up in Table IV show that TFRC$_{light}$ flows occupied a fair share of the bandwidth when multiplexed with TCP and TFRC flows. These results show that our proposal is friendly with TCP as it did not obtain a bandwidth higher than TCP. On the contrary, TFRC$_{light}$is less friendly with our TFRC user space implementation. This can be explained by the fact that TFRC$_{light}$, as explained in the section III, does not react as quickly as the original TFRC algorithm when losses occur in the network.

V. QUANTIFICATION OF THE SHIFTING SCHEME

In Table V, we summarize the benefits and drawbacks of the proposed design compared to the original algorithm.

The main advantages of our solution are the removal of the packet-history structure and the removal of the packet-loss rate computation at the receiver. Conversely, we have introduced a

TABLE IV
FAIRNESS INDEX FOR DIFFERENT FLOW MELTING

	T(TFRC$_{light}$)	T(TCP)	T(TFRC)
5TFRC$_{light}$ and 5TFRC	1.05	N/A	0.95
5 TFRC$_{light}$ and 10TCP	0.92	1.08	N/A

TABLE V
SUMMARY OF THE BENEFITS AND DRAWBACKS OF TFRC$_{light}$

benefits	• suppression of the loss history structure • no processing of the packet-loss rate • protection from misbehaving receivers • simpler timer management • simpler sender's algorithm
drawbacks	• new structure for Sack vectors management • loss events built from sender point of view • feedback sent periodically only

new light structure that allows the receiver to build the Sack vector sent to the sender in feedback messages. This structure has a size of the order of $4RTT * Bandwidth/(packetsize)$. For instance, in the case of a transmission with a bandwidth of $1Mbit/s$, an RTT of $100ms$ and a packet size of $1000Bytes$, the structure has a maximum size of $50bits$. This structure is actualized for each data packet received. In the original receiver-based design of TFRC, the receiver had to manage a complex structure that stores information concerning the arrived or lost packets. The stored information includes:

- the packet timestamp (16bits);
- the packet size (8bits);
- the arrival time (16bits).

Therefore, the elementary size of an entry is $40bits$. Furthermore, this structure potentially entails an unbounded size. Indeed, this structure is emptied after detecting a loss event only. As an example in Fig. 5, there are no losses between $t = 63$ and $t = 137$. During this entire period, the structure has to be updated at a rate of $1Mbit/s$ which corresponds to $125packet/s$. This structure for the given example would contain:

$$40 * 125 * (137 - 63) = 370Kbits$$

when it can be released. In this particular case, with TFRC$_{light}$, the memory use would decreases from $370Kbits$ to $50bits$. This comparison remains true in another sender-based approach such as proposed in [2]. Indeed, in this proposal, the receiver is still responsible for the differentiation between a loss event and a packet lost. Therefore, it still needs to maintain a structure storing information of the arrival time of the packet as described above.

Nevertheless, the following CPU's cycle comparison can only be applied to the original TFRC if the sender-based option is configured to compute the packet-loss rate at the sender side. Indeed, this option can also be activated only to double check the packet-loss rate field in the feedback header. Therefore, the receiver still computes this estimation.

To estimate the computation benefit of our proposal, let us consider how in normal TFRC [3] the loss rate estimate is processed for every received packet as shown in Fig. 1. The basic algorithmic sequence for computing the loss rate estimate entails the following set of elementary arithmetic operations: eight additions, eight multiplications, one division and one maximum operation. For instance, at rate of $1Mbit/s$ with a packet size of $1Kbyte$, this estimation should be computed 125 times per second. These elementary operations can be translated into CPU cycles as follows[4]:

- division = 70 cycles
- multiplication = 15 cycles
- addition, maximum = 0.5 cycles

As a result, for the given example, in the original TFRC, the receiver has to use $24312.5\ cycles/s$.

Furthermore, after a slow-start phase, the receiver has to initiate its loss history. This initialization is done from the inversion of equation (1) in order to find the packet loss rate corresponding to the measured received rate. This initialization is usually done with a binary search and uses the list of elementary operations sum up in Table VI.

TABLE VI
LIST OF THE NUMBER OF ELEMENTARY OPERATIONS
($n = number\ of\ iterations$)

	$+$	$*$	$/$	$sqrt$
binary search	$4n + 4$	$8n + 8$	$2n + 2$	n
CPU cycles	0.5	15	70	70

The worst case of this binary search can be observed when this algorithm diverges, which can occur when the solution of the inversion of (1) is outside the $[0, 1]$ range. This potential of divergence leads to an upper bound on the number of iterations done during the binary search. Therefore, in order to compute the inversion of (1) for most cases, the maximum number of iterations is usually set to 50. Indeed, we implemented the binary search of the inversion and found out that the algorithm converges in 15 iterations for $RTT = 400ms$ and $bandwidth = 1Mbit/s$.

In conclusion, for the worst case it takes 16862 CPU cycles for the initialization process. In our proposal, all of this computational process is achieved at the sender side. Moreover, we have shown in section IV that this simplification entails a congestion-control behaviour that strictly conforms to receiver-based TFRC implementations.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented and evaluated the design of a sender-based TFRC congestion control mechanism This design is driven by the aim of shifting the computation of the loss rate estimation from the receiver to the sender, in order to alleviate the processing and memory needs of "light" receivers. This shifting requires the sending of loss-resilient feedbacks, and is accomplished through the use of a SACK-like mechanism. This results in a significantly lightened

[4]According to Intel PIV documentation

computational load on the receiver which is particularly useful for mobile clients with computation and energy constraints.

We have shown that the proposed sender-based TFRC architecture behaves identically to the official ns-2 implementation and remains friendly to TCP streams. This validation has been accomplished through well accepted metrics which confirmed that our architecture remains as efficient as the original TFRC. We have also quantified the benefits of this shift from the perspective of computations and memory.

Furthermore, the proposed solution allows the security issues raised in [3] to be resolved. These security issues are related to the forwarding of false loss event rates by the receiver. Such misbehaviour is no longer possible with our solution when associated with nonce mechanisms. We plan to further validate our proposal by performing a large range of experimental measurements on a multi-hop testbed.

VII. Acknowledgments

The authors would like to thank Sebastien Ardon and Max Ott for continuous help and support. This work has been supported by National ICT Australia funding.

References

[1] E. Kohler and M. Handley and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," IETF, Request For Comments 4340, Mar. 2006.

[2] S. Floyd, E. Kohler, and J. Pahdye, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)," IETF, Request For Comments 4342, Mar. 2006.

[3] M. Handley, S. Floyd, J. Pahdye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," IETF, Request For Comments 3448, Jan. 2003.

[4] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," IETF, Request For Comments 2883, July 2000.

[5] M. Georg and S. Gorinsky, "Protecting tfrc from a selfish receiver," in *Proc. of Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS/ICNS 2005)*, Oct. 2005.

[6] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," IETF, Request For Comments 2326, Apr. 1998.

[7] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communications Review*, vol. 27, no. 1, Jan. 1997.

[8] Yunhong Gu et Robert Grossman. "UDT: Udp-based data transfer for high speed wide area networks.", *Elsevier Computer Network*, 51(7), August 2007.

[9] D. Bertsekas and R. Gallager. "Data Networks". *Prentice-Hall*, Englewood Cliffs, New Jersey, 1992.

[10] R. Jain. "The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling." *Wiley-Interscience*, New York, NY, April 1991.

[11] B. Briscoe, "Flow Rate Fairness: dismantling a religion". *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, Apr. 2007

[12] C. Jin, D. X. Wei, and S. H. Low. "FAST TCP: motivation, architecture, algorithms, performance." *IEEE Infocom '04*, Hongkong, China, Mar. 2004.

[13] W. Fang, N. Seddigh, AL., "A Time Sliding Window Three Colour Marker", Request For Comments 2859, IETF (Jun. 2000).

[14] A. Gurtov and S. Floyd, "Modeling wireless links for transport protocols", *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, Apr. 2004

Guillaume Jourjon is researcher at NICTA. He received his PhD from the University of New South Wales and the Toulouse University of Science in 2008. Prior to his PhD, he received a Engineer Degree from the ENSICA, a French aeronautical engineering school in Toulouse and a Master of Research in telecommunications and networking. He also received a DEUG in Physics and Chemistry (Major) and Mathematic (Minor) from the University of Toulouse III.

Emmanuel Lochin received his Ph.D from the LIP6 laboratory of Pierre and Marie Curie University - Paris VI in December 2004. From July 2005 to August 2007, he held a researcher position in the Networks and Pervasive Computing research program at National ICT Australia, Sydney. He joined ISAE in September 2007 as researcher and network security officer and is also member of the research group OLC (Tools and Software for Communication) of the LAAS-CNRS. His research interests include DTN, new transport protocols and congestion control.

Patrick Sénac graduated from the Ecole Nationale Superieure d'Ingenieurs d'Hydraulique d'Electrotechnique, d'Electronique et d'Informatique de Toulouse (ENSEEIHT) in 1983 and received the Ph.D. degree in computer science in 1996 from Toulouse University, France. Patrick Sénac is professor of computer science and head of the Mathematics, Computer Science and Control Department at the Institut Supérieur de l'Aéronautique et de l'Espace (ISAE) in Toulouse, France, and his also member of the research group OLC (Tools and Software for Communication) of the LAAS-CNRS. He has been working during the last decade on the modeling and design of synchronization constraints in distributed multimedia/hypermedia systems. Patrick Sénac is currently leading the French national research group on Communication Networks of the CNRS and is member of the committee of experts in Communication networks of the CNRS. His current research interests focus on the modeling and design of advanced transport protocols and end to end communication mechanisms over the Internet.