On ACK Filtering on a Slow Reverse Channel

Chadi Barakat, Eitan Altman

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On ACK Filtering on a Slow Reverse Channel

Chadi Barakat — Eitan Altman

N° 3908

March 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3908--FR+ENG

On ACK Filtering on a Slow Reverse Channel

Chadi Barakat , Eitan Altman

Thème 1 — Réseaux et systèmes
Projet Mistral

Rapport de recherche n° 3908 — March 2000 — 25 pages

Abstract: ACK filtering has been proposed as a technique to alleviate the congestion at the input of a slow channel located on the reverse path of a TCP connection. Old ACKs waiting at the input of the slow channel are erased when new ACKs are to be queued. In the literature the case of a one-ACK per connection at a time has been studied. In this paper, we show that this is too aggressive for short transfers where ACKs arrive in bursts due to the slow start phase and where the TCP source needs to receive the maximum number of ACKs to increase faster its window. We study first static filtering where a certain ACK queue length is allowed. We show how this length needs to be chosen in order to improve the performance. We present then some algorithms that adapt ACK filtering as a function of the slow channel utilization rather than the ACK queue length. These algorithms provide a good compromise between reducing the queueing delay and passing a large number of ACKs to guarantee a fast window increase.

Key-words: TCP, Slow start, Asymmetric paths, ACK filtering, Buffer management.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Sur le filtrage des ACKs sur un lent chemin de retour

Résumé : Plusieurs problèmes ont été relevés lorsque le chemin de retour d'une connexion TCP ne possède pas assez de bande passante pour faire passer tous les acquittements (ACK) générés par la destination. C'est le cas des utilisateurs téléchargeant des informations de l'Internet à travers un lien haut débit comme un lien satellite ou bien un câble, et envoyant les requêtes et les ACKs sur un lien de bas débit comme une ligne téléphonique. Une congestion apparaît sur le chemin de retour causant une détérioration du débit de la connexion et un problème d'iniquité dans le cas de plusieurs connexions en compétition pour le chemin de retour. Le filtrage des ACKs à l'entrée du goulot d'étranglement a été proposé pour éliminer cette congestion. Un ACK qui arrive à un buffer et qui trouve un certain nombre des ACKs de la même connexion, efface quelques ACKs et prend leur place. Dans la littérature un seul cas est étudié, celui d'un ACK qui efface tous les ACKs de la même connexion en attente dans le buffer. Dans ce papier, on étudie le problème de filtrage des ACKs dans son cas le plus général. On montre qu'effacer tous les ACKs est une opération très agressive qui a de mauvaises conséquences sur la phase slow start de TCP où les ACKs arrivent en rafales et ou le plus grand nombre possible des ACKs doit être passé à la source pour l'aider à augmenter rapidement sa fenêtre. On présente une étude qui montre combien d'ACKs il faut garder dans le buffer pour ne pas affecter la phase slow start de la source. Ceci constitue notre première famille d'algorithmes qui utilisent le nombre d'ACKs dans le buffer comme indication pour commence le filtrage des ACKs. On présente ensuite une autre famille d'algorithmes qui se servent de l'utilisation de la bande passante comme information pour le début du filtrage. Des simulations avec `ns` montrent le gain de performance qu'apporte ces algorithmes.

Mots-clés : TCP, Slow start, Chemins asymétriques, Filtrage des ACKs, Gestion des buffers.

1 Introduction

Accessing the Internet via asymmetric links is becoming common with the introduction of satellite and cable networks. Users download data from the Internet via a high speed link (e.g. a satellite link at 2 Mbps) and send requests and Acknowledgements (ACKs) via a slow reverse channel (e.g. a dial-up modem line at 64 Kbps). This kind of paths is called asymmetric paths and it is often studied in the literature [1, 3, 5, 8, 15] (see Figure 1 for an example of such paths). It has been shown that the slowness of the reverse channel limits the throughput of a TCP transfer running in the forward direction. TCP, the most widely used Internet protocol, is known to generate a large number of ACKs which are necessary for the sender operation. They are used as a clock to trigger the transmission of new data, to slide and to increase the congestion window [11, 17]. From TCP point of view, an asymmetric path has been defined as one where the reverse channel is not fast enough to carry the flow of ACKs resulting from a good utilization of the forward direction bandwidth. A queue of ACKs builds up in the buffer at the input of the reverse link (we call it the *reverse buffer* in the sequel) causing an increase in the Round Trip Time (RTT) and an overflow of the buffer. The result is a performance deterioration for many reasons. First, for a given window, an increase in the RTT reduces the throughput since the throughput of a TCP connection is equal at any moment to the window size divided by the RTT. Second, the increase in the RTT as well as the loss of ACKs reduce the window increase rate which has mainly an impact on the slow start phase. Third, the loss of ACKs results in gaps in the ACK clock which leads to burstiness at the source. Fourth, the loss of ACKs reduces the capacity of TCP (especially Reno [9]) to recover from losses without Timeout and slow start. Another problem has been reported in case of multiple connections contending for the reverse channel. This is the problem of deadlock of a new connection sharing the reverse channel with already running connections [15]. Due to the overflow of the reverse buffer, this connection suffers from the loss of its first ACKs which prohibits it from increasing its window. This deadlock continues until the dominant connections reduce their rates. We can see this last problem as a result of an unfairness in the distribution of the available bandwidth on the slow channel between the different flows. The main reason for such unfairness is that a flow of ACKs is not responsive to drops as a TCP packet flow. The already running connections continue to grab most of the bandwidth on the slow link until they reduce their transmission rate due to the loss of a data packet or the end of data at the source.

Many solutions have been proposed for this problem of bandwidth asymmetry. Except the header compression solution (e.g. the SLIP algorithm in [12]) which

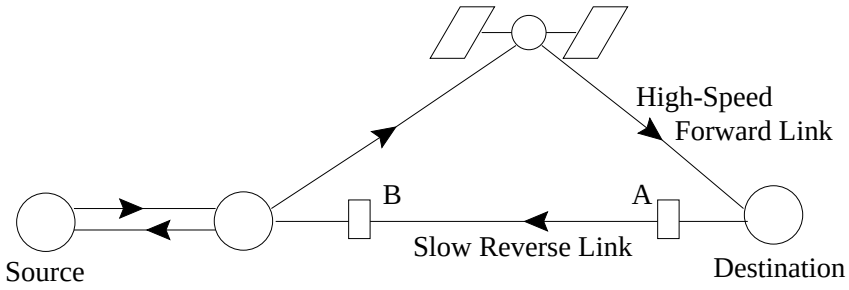

Figure 1: An asymmetric path

proposes to reduce the size of ACKs in order to increase the rate of the reverse channel in terms of ACKs per unit of time, the other solutions try to match the rate of ACKs to the rate of the reverse channel. The matching can be done by either delaying ACKs at the destination [3, 8] or filtering them in the reverse buffer [1, 3]. Adaptive delaying of ACKs at the destination, called also ACK congestion control [3], requires the implementation of new mechanisms at the receiver together with some feedback from the network. The idea is to adapt the generation rate of ACKs as a function of the reverse buffer occupation. ACK filtering however requires only modification at the reverse buffer. It profits from the cumulative nature of the information carried by ACKs. Indeed, an ACK carries the sequence number of the next expected in-order byte and this information can be safely substituted by the one carried by a subsequent ACK with a larger sequence number. From the ACK content point of view, there is no need for queueing ACKs in the reverse buffer. Thus, when an ACK arrives at the reverse channel, the reverse buffer is scanned for the ACKs from the same connection and some (or all) of these ACKs are erased. The buffer occupancy is then maintained at low levels without the loss of information.

In this paper, we ask the question how many ACKs we must authorize a connection to have in the reverse buffer before filtering its ACKs. In the literature, the case of a one ACK per-connection at a time has been studied [1, 3]. When an ACK arrives at the reverse channel, the buffer erases any ACK from the same connection waiting in the buffer and replaces them by the new one. Clearly, this behavior optimizes the end-to-end delay and the queue length, but it ignores the fact that TCP uses the ACKs to increase its window. This may not have an impact on the congestion avoidance phase but it has certainly an impact on slow start where the window is small and needs to be increased as quick as possible to get good performance. The

impact on slow start comes from the fact that TCP is known to be bursty during this phase [2, 4, 14] and that filtering bursts of ACKs will result in a small number of ACKs reaching the source and a slow window increase. The impact will be important on short transfers which dominate most of today's Internet traffic [6] and particularly on those who cross long delay links (e.g. satellite links) where slow start is already slow enough [5]. Authorizing some number of ACKs from a connection to be queued in the buffer before filtering will have the advantage of absorbing these bursts of ACKs required for a faster window increase. However, this threshold must be kept at small values in order to limit the end-to-end delay. A certain tradeoff appears and one must predict an improvement in the performance as the threshold increases followed by a deterioration in the performance when it becomes large (see Figure 6 for an example). We study first the case where the ACK filtering threshold is set to a fixed value and we show how it must be chosen. We present then some algorithms that adapt ACK filtering as a function of the slow link utilization rather than the ACK queue length. This corresponds to a dynamic setting of the ACK filtering threshold. The objective is to pass as many ACKs as possible to the source while maintaining the end-to-end delay at small values. In case of many connections, the filtering must be adapted in a way to share fairly the slow channel bandwidth between the different connections.

In the next section, we present a description of the impact of ACK filtering (or old ACK dropping) on the performance. We show how delaying ACK filtering until a certain queue length accelerates slow start and improves the performance without impacting the end-to-end delay. In section 3, we describe our adaptive algorithm we call *Delayed Filtering* that doesn't relate filtering to the number of queued ACKs but rather to the link utilization or to the ACK presence in the reverse buffer. The propositions are validated via simulations with `ns`, the network simulator developed at LBNL [16]. In section 4, we consider the case of multiple connections and we extend our algorithm to this case. Section 5 concludes the paper.

2 Impact of ACK filtering on TCP performance

In contrast to the other works studying the impact of bandwidth asymmetry on TCP, we focus on short transfers which are considerably affected by the slow start phase. Long transfers are dominated by the congestion avoidance phase where an aggressive ACK filtering has no great impact since the window is normally large and the burstiness of ACKs is less important than in slow start mode. Consider first the case of a single transfer. We assume that buffers on the forward direction are large

enough to absorb the burstiness resulting from the loss of ACK. Our objective is to show that delaying filtering until a certain number of ACKs get queued, shortens the slow start phase and improves the performance if this threshold is correctly set.

2.1 TCP and network model

Let μ_r be the bandwidth available on the reverse path and let T be the constant component of the RTT (equal to the RTT in absence of queueing in the forward and the reverse directions). μ_r is measured in terms of ACKs per unit of time. Assume that packet queueing appears only in the reverse buffer at the entry of the reverse link. Thus, the RTT remains constant whenever the reverse channel is not fully utilized, this means whenever the number of ACKs arriving at the reverse buffer per RTT is less than $\mu_r T$. Note here that due to the burstiness of TCP traffic, data packets belonging to the same window may experience different RTT if these packets or their corresponding ACKs get queued in the buffer behind previous data packets or previous ACKs from the same window. Some data packets will see then a longer RTT than T although the reverse channel is not fully utilized. Later, we explain how the RTT is measured. For the moment, take it as the smallest RTT seen by a window of data packets.

Assume for the moment that the reverse buffer is large and that ACKs are not filtered. This assumption will be eliminated later. The window at the sender grows then exponentially with an increasing rate function of the frequency at which the receiver acknowledges packets. Suppose that the receiver acknowledges every d packets, thus the window increases by a factor $\alpha = 1 + 1/d$ every RTT. Note that most of TCP implementations acknowledge every other data packet ($d = 2$) [17]. Denote by $W(n)$ the congestion window size at the end of the nth RTT. It follows that,

$$W(n+1) = (d+1)W(n)/d = \alpha W(n).$$

For $W(0) = 1$, this gives $W(n) = \alpha^n$ which shows well the exponential increase.

Once the reverse link is fully utilized, ACKs start to arrive at the source at a constant rate of μ_r ACKs per unit of time. Here, the window together with the RTT start to increase linearly with time. Recall that we are working during the slow start mode where the window is increased by one packet upon every ACK arrival. The instantaneous throughput, which is equal to the window size divided by the RTT, stops then increasing and becomes limited by the reverse channel. This continues until ACKs start to be filtered or dropped in the reverse buffer. The RTT stops then increasing and the instantaneous throughput resumes its increase with the window size. Thus, the first remark we can derive here is that the ACK queue length needs to

be maintained at small values to get a small RTT and to improve thus the throughput (saying one ACK per-connection). But, ACKs may arrive at the reverse buffer in separate bursts at a rate higher than μ_r without having an average rate higher than μ_r. This happens frequently during slow start where the source transmits packets in bursts due to the fast window increase [2, 14]. An early filtering will reduce then the number of ACKs reaching the source whereas these bursts can be absorbed without causing any increase in the RTT. Such absorption has an important impact on slow start. It permits to more ACKs to reach the source, which accelerates the window increase. Given that the RTT remains constant whenever the reverse channel is not fully utilized, a faster window increase results in a faster throughput increase and thus in a better performance. Thus, the general guideline for ACK filtering is to accept all ACKs until the slow channel becomes fully utilized and here to filter them in order to limit the RTT. We consider first the case where the length of the ACK queue is used to indicate the start of filtering. An old ACK is filtered once the number of ACKs in the reverse buffer exceeds a certain threshold that we fix along the connection lifetime. Later, we present algorithms that adapt ACK filtering as a function of the slow link utilization. This permits a simpler implementation together with a better performance than fixing a certain number of ACKs. The following study is however necessary to understand how the performance varies as a function of the filtering threshold.

2.2 ACK filtering threshold

First, let us find the maximum queue length the reverse buffer must allow before the start of filtering. The objective is to absorb the bursts of ACKs caused by slow start and to start filtering just after the full utilization of the slow channel. We call this queue length the optimum filtering threshold and we denote it by δ_o. We study then the impact on the window increase of a filtering threshold δ less than δ_o. As we said, we focus on the slow start phase of short TCP transfers.

During slow start, TCP is known to transmit packets in long bursts (see [4] for a description of this bursty behavior). A burst of $W(n)$ packets is transmitted at the beginning of Round Trip n. It causes the generation of $W(n)/d$ ACKs which reach the source at the end of the RTT (Figure 2). Given that the reverse channel is the bottleneck and due to the window increase at the source, bursts of ACKs can be assumed to have a rate μ_r at the output of the reverse channel and a rate $\alpha\mu_r$ at the input of the reverse channel. This can be easily proved by assuming that the time length of the burst of $W(n)/d$ ACKs at the output of the slow channel is equal to the time length of the burst of $W(n+1)/d$ ACKs at its input. Thus, during the

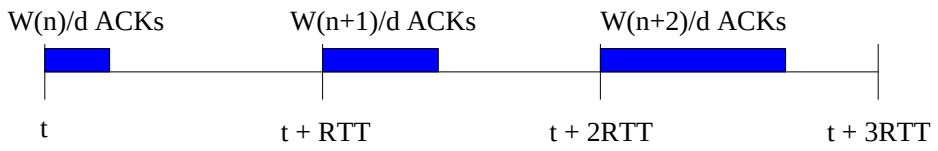

Figure 2: Bursts of ACKs as they cross the reverse channel

receipt of a long burst of ACKs, a queue builds up in the reverse buffer at a rate $(\alpha - 1)\mu_r$. A long burst of X ACKs at a rate $\alpha\mu_r$ causes the building of a queue of length $X/(d+1)$ ACKs. The full utilization of the reverse channel requires the receipt of a long burst of ACKs of length $\mu_r T$ and thus the absorption of a queue of length $\mu_r T/(d+1)$. This is the optimum δ the buffer must use,

$$\delta_o = \mu_r T/(d+1). \tag{1}$$

It guarantees that the reverse channel is fully utilized before the start of filtering which is necessary to get a fast window increase during slow start. This threshold increases with the increase in the RTT and the slow channel rate. Also, it increases with the volume of generated ACKs.

2.3 Early ACK filtering

We consider now the case where the ACK filtering threshold is set to less than δ_o. Let $\delta < \delta_o$ be the maximum allowed length of the ACK queue. When an ACK arrives and finds more than δ ACKs in the buffer, the oldest ACK is erased and the new ACK is queued. We call it a *static* filtering strategy since δ is not changed during the connection lifetime. It can be seen as a drop from front strategy with a buffer of size δ. A $\delta = 1$ corresponds to the ACK filtering policy studied in the litterature [1, 3]. Let us study the impact of δ on the window increase rate.

Starting from $W(0) = 1$, the window continues to increase exponentially until Round Trip n_0 where ACKs start to be filtered. This happens when the ACK queue length reaches δ which in turn happens when the reverse buffer receives a long burst of length $\delta(d+1)$ at a rate $\alpha\mu_r$. Given that the length of the long burst of ACKs received during Round Trip n_0 is equal to $W(n_0)/d$, we write

$$W(n_0) = \alpha^{n_0} = \delta d(d+1).$$

After Round Trip n_0, ACKs start to be filtered and the window increase rate slows. The RTT, which is equal to the time between the arrival of two long bursts at the slow channel, remains equal to T whenever the reverse channel is not fully utilized. Thus, the window variation as a function of time is identical, with a certain scaling factor, to its variation as a function of Round Trip number. Hence, we study the growth of the window as a function of n, the Round Trip number, rather than time. To show the impact of δ on the window increase rate, we define the following variables in addition the those of the previous section. Consider $n > n_0$ and put ourselves in the region where the filtering is active but the slow channel is not fully utilized. We know that the maximum window increase rate is achieved when the reverse channel is fully utilized and the best performance is obtained when we reach the full utilization as soon as possible. Let $N(n)$ represent the number of ACKs that leave the slow channel during Round Trip n. As usual, $W(n)$ represents the window size during this Round Trip. Given that we are in slow start, the window as well as the length of the long burst of packets sent in the next Round Trip will be,

$$W(n+1) = W(n) + N(n).$$

This burst of data packets generates $W(n+1)/d$ ACKs at the destination which reach the slow reverse channel at a rate faster than μ_r (it is even faster than αmu_r due to ACK filtering). The duration of this burst is equal to the duration of the burst $N(n)$ at the output of the slow channel in the previous Round Trip. Recall that we are working in a case where the bandwidth available in the forward direction is very important compared to the rate of the slow reverse channel so that many packets can be transmitted at the source between the receipt of two ACKs. During the receipt of the $W(n+1)/d$ ACKs, the reverse buffer overflows and the slow channel transmits $N(n)$ ACKs. The ACKs stored in the reverse buffer whose number is equal to δ are then sent. This forms a burst of ACKs at the output of the slow channel of length,

$$N(n+1) = N(n) + \delta.$$

These ACKs only reach the source and the rest of the burst of $W(n+1)/d$ ACKs is dropped. Figure 3 shows the content of the reverse buffer as a function of time after the start of ACK filtering and before the full utilization of the slow reverse channel. We can write,

$$\begin{aligned} N(n) &= N(n-1) + \delta = N(n_0) + (n - n_0)\delta \\ &= W(n_0)/d + (n - n_0)\delta = \delta(d + 1 + n - n_0) \end{aligned}$$

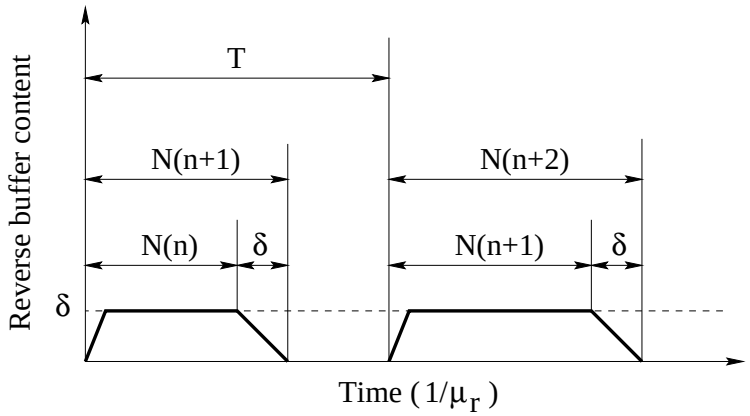

Figure 3: Reverse buffer content as a function of time

Thus, for $n > n_0$,

$$\begin{aligned} W(n) &= W(n-1) + N(n-1) = W(n_0) + \sum_{k=n_0}^{n-1} N(k) \\ &= \delta[d(d+1) + (n-n_0)(d+1) + (n-n_0)(n-n_0-1)/2] \end{aligned}$$

We remark that due to the small δ, the window increase changes from exponential to polynomial and it slows with δ. More time is required to fully utilize the reverse channel and the source wastes a long time during slow start before reaching the maximum window increase rate of μ_r packets per unit of time. We can also see this from the expression of $N(n)$. For $\delta < \delta_o$ and after $n > n_0$, the utilization of the slow channel is proportional to $N(n)$ which increases by δ ACKs every T. However, before n_0, the increase is exponential and the utilization is proportional to $W(n)$ which is multiplied by a factor α every T.

Now the question that one may ask here is whether the δ ACKs we let queue in the buffer to absorb the bursts of ACKs and to accelerate the window increase have an impact on the RTT in the steady state. Certainly this increases the RTT but our simulations show that for short transfers, the gain caused by accelerating the window increase during slow start is more important than the throughput we lost due this increase in the RTT. For long transfers the situation is different since the slow start phase is less important and the RTT has more impact on the performance. There is no need to queue ACKs once the reverse channel is fully utilized. The appropriate

filtering strategy is the one that starts with a large δ than switches to a small one (say $\delta = 1$) once we are sure that there is enough ACKs to fully utilize the slow channel. The implementation of such strategy requires an estimation of the slow link utilization and the adaptation of the filtering as a function of this utilization rather than the queue size. We show the impact of such strategy on the performance when presenting our algorithms.

2.4 Simulations

Consider a simulation scenario where a TCP Reno source transmits packets of size 1000 Bytes over a forward link of 10 Mbps to a destination that acknowledges every data packet ($d = 1$). The forward buffer, the receiver window as well as the slow start threshold at the beginning of the connection are set to high values. ACKs cross a slow link of 100 Kbps back to the destination. The Round Trip Time is set to 200 ms. We use the `ns` simulator [16] and we monitor the packets sent during the slow start phase at the beginning of the connection.

We implement an ACK filtering strategy that limits the number of ACKs in the reverse buffer to δ and we compare the performance for different values of δ. The buffer size itself is set to a large value. We provide three figures where we plot as a function of time, the window size (Figure 4), the instantaneous throughput (Figure 5) and the last acknowledged sequence number (Figure 6). For such a scenario, the calculation gives a δ_o equal to 30 ACKs (equation (1)). Normally, a δ less than δ_o slows the window growth and requires more time to reach the linear window increase phase. We see this for the $\delta = 1$ and $\delta = 3$ in Figure 4. The other δ gives the same window increase given that they start filtering after the full utilization of the reverse channel. This curve however is not sufficient to study the performance since the same window may mean different performance if the RTTs are not the same. To show this, we plot the instantaneous throughput by counting the number of acknowledged bytes every T (Figure 5). For small δ, the RTT can be considered as always constant and the curves for the instantaneous throughput and the window have the same look. The instantaneous throughput saturates when it reaches the available bandwidth in the forward direction (at 1000 Kbps). It also saturates somewhere in the middle (e.g. at 3s for $\delta = 1$ and at 1.5s for $\delta \geq 30$). This last saturation corresponds to the time between the full utilization of the reverse channel and the convergence of the RTT to its limit when δ ACKs are always present in the reverse buffer. We know that the minimum value of the RTT is T, the two-way propagation delay. We know also that in absence of queueing on the forward direction, the maximum value of the RTT is seen when δ ACKs start to be always present in the reverse

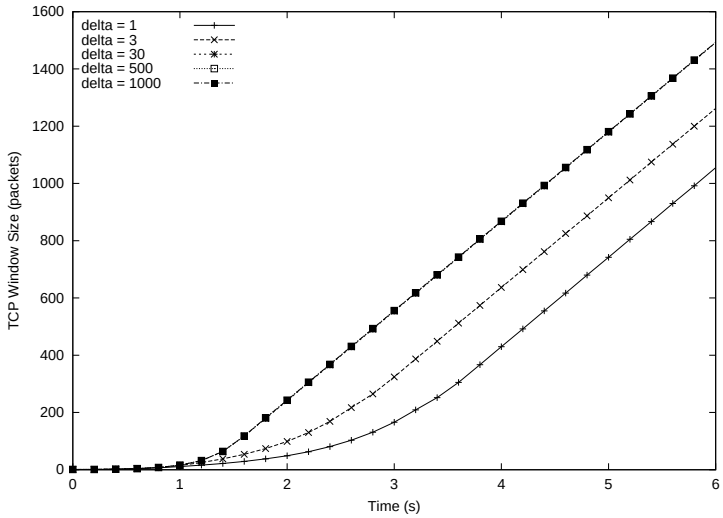

Figure 4: TCP window vs. time

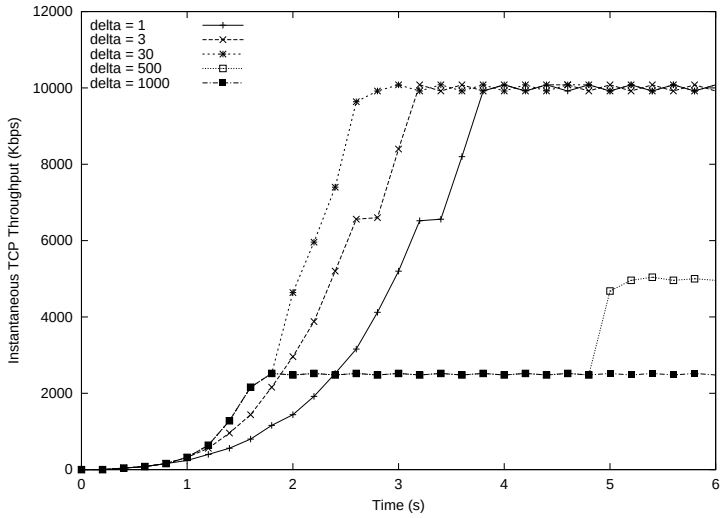

Figure 5: Instantaneous throughput vs. time

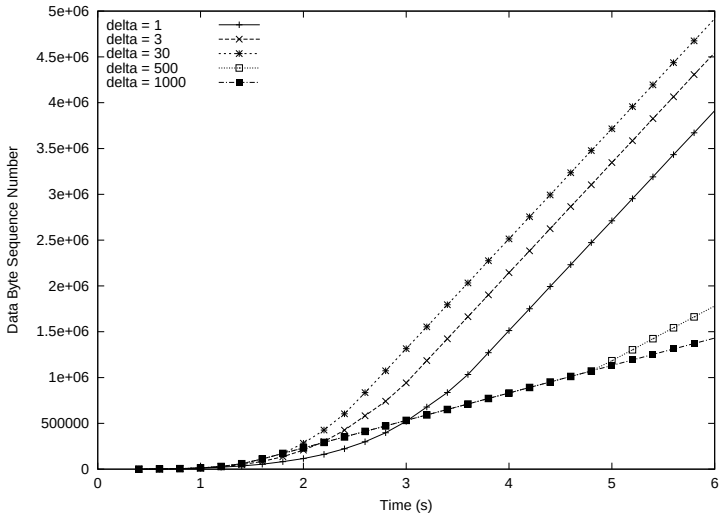

Figure 6: Last acknowledged sequence number vs. time

buffer. The maximum value is equal to T plus the transmission time of δ ACKs on the slow channel ($T + \delta/\mu_r$). Some time is required to move between these two limits and during this time, the instantaneous throughput remains constant due to a linear increase of both the window size and the RTT. Once the RTT stabilizes, the instantaneous throughput resumes its increase with the window. For small δ, the convergence of the RTT is quick and the saturation lasts a short time. However, for very large δ, a long time is needed for the RTT to stop its increase. The increase stops when ACKs start to be filtered. This happens around 5s for the case $\delta = 500$. The third figure (Figure 6) is an indication of the overall performance. We see well how taking $\delta = \delta_o$ gives the best performance since it forms a good compromise between delaying the filtering to improve the window increase and bringing it forward to reduce the RTT. While increasing δ, the overall performance improves until δ_o then worsens. Recall that we are talking about short transfers. This conclusion is not valid in case of very long transfers where slow start has a slight impact on the overall performance.

3 Delayed filtering : Case of a single connection

Tracking the instantaneous queue length for filtering is not a good guarantee for good performance. First, in practice the arrival of ACKs may be completely different than the theoretical arrival we described (due to cross traffic in the forward direction as an example). Second, the calculation of the optimum threshold δ_o is difficult since it requires knowledge of the RTT and the acknowledging strategy. Third, setting the filtering threshold to a fixed value leads to an unnecessary increase in the RTT in the steady of the connection. An aggressive filtering needs to be applied once the slow reverse channel is fully utilized. Some mechanisms must be implemented in the reverse buffer to absorb the bursts of ACKs when the slow channel is not well utilized and to filter ACKs with a small δ otherwise. Using the average queue length as in Random Early Detection (RED) buffers [10] does not work since the objective is not to control the average queue as in case of data packets but rather to find a signal for a good utilization of the slow link.

The simplest solution is to measure the rate of ACKs at the output of the reverse buffer (before being transmitted over the slow link) and to compare it to the link bandwidth. Measuring the rate at the output rather than at the input of the reverse buffer is better since ACKs are spread over time which increases the precision of the measurement tool. In case we don't know the link bandwidth, we can measure how frequent ACKs are present in the buffer. This gives an idea on how well the link is utilized. A full utilization during a certain interval is equivalent to a continuous presence of ACKs in the buffer during this interval. We called this measure *the presence rate* of ACKs. When one of these two measures indicates that the link is fully utilized (e.g. the utilization exceeds a certain threshold that we fix to 90% in simulations), we start to apply the classic filtering described in the literature [3]: erase all old ACKs when a new ACK arrives. Once the utilization drops below a certain threshold, filtering is halted until the utilization increases again. This guarantees a maximum window increase during slow start and a minimum RTT in the steady state. We can see it as a dynamic threshold ACK filtering where δ is set to infinity when the slow link is under-utilized and to 1 when it is well utilized. Also, it can be seen as a transformation of the rate of the input flow of ACKs from λ to $\min(\lambda, \mu_r)$ without the loss of information, of course if the reverse buffer is large enough to absorb bursts of ACKs before the start of filtering. Recall that we are always working with the case of a single connection. The case of multiple concurrent connections is studied later.

3.1 Utilization measurement

We assume that the slow link bandwidth is known and we use the output rate approach. The Time Sliding Window (TSW) algorithm defined in [7] is used for ACK rate measurement. When a new ACK leaves the buffer, the time between this ACK and the last one is measured and the average rate is updated by taking a part of this new measurement and by taking the rest from the past. The difference from classic low pass filters is that the decay time of the rate with the Time Sliding Window algorithm is a function of time not the frequency of measurements. The coefficient of contribution of the new measurement varies with the importance of the measurement. A large time between the current ACK and the last one contributes to the average rate more than a short time. The decay time of the algorithm is controlled via a time window that decides how much the past is important. The algorithm is defined as follows. Let `Rate` be the average rate, `Window` be the time window, `Last` be the time when the last ACK has been seen, `Now` the current time, `Size` be the size of the packet (40 Bytes for ACKs). Thus, upon packet departure,

```
Volume = Rate*Window + Size;
Time = Now - Last + Window;
Rate= Volume /Time;
Last=Now;
```

The same algorithm can be applied for the measurement of the presence rate. `Size` is taken equal to 1 if the queue is found not empty upon update and zero if it is found empty. But, in this case, we need to update the rate also upon packet arrival to account for the switch from empty to non-empty. Thus, upon packet departure,

```
Volume = Rate*Window + 1;
Time = Now - Last + Window;
Rate= Volume/Time;
Last=Now;
```

Upon packet arrival,

```
if (length(queue)==0) Volume = Rate*Window;
else Volume = Rate*Window + 1;
Time = Now - Last + Window;
```

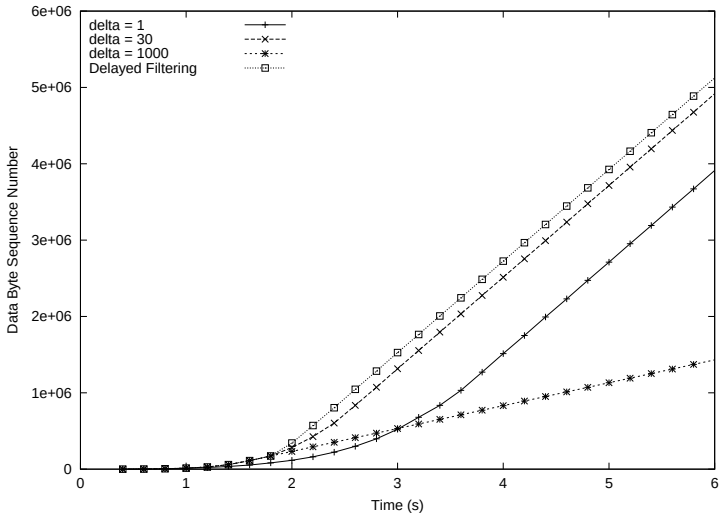

Figure 7: Last acknowledged sequence number vs. time

```
Rate= Volume/Time;
Last=Now;
```

3.2 Simulation

We consider the same simulation scenario as in the previous section. We implement delayed filtering in the reverse buffer and we plot its performance. The time window is taken in the same order of the RTT. Figure 7 shows how it gives as good performance as the case when δ is choosing correctly. Moreover, it outperforms slightly the case of optimum δ due to the erasing of all the ACKs once the slow link is fully utilized. The behavior of delayed filtering can be seen clearly in Figures 8 and 9 where we plot the reverse buffer occupancy as a function of time. These figures correspond to $\delta = 30$ and delayed filtering respectively. Bursts of ACKs are absorbed at the beginning before being filtered some time later. The filtering starts approximately at the same moment in the two cases.

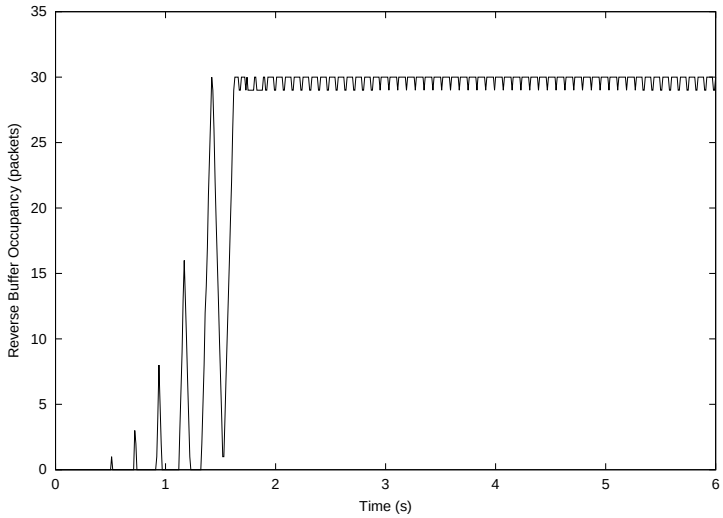

Figure 8: Reverse buffer occupancy for $\delta = 30$

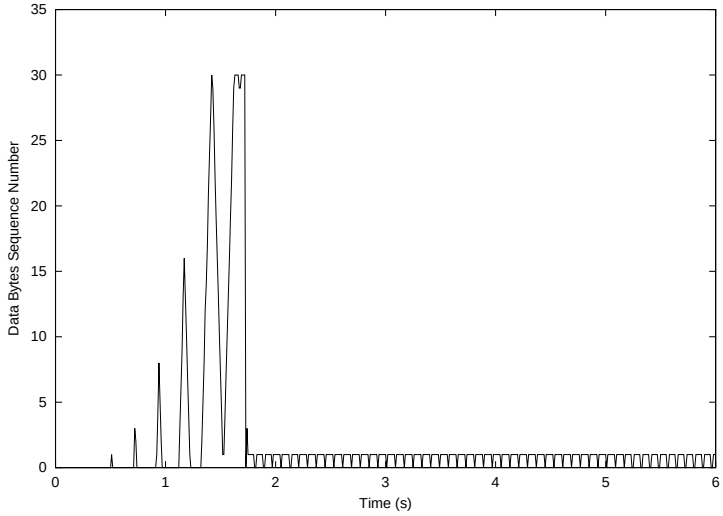

Figure 9: Reverse buffer occupancy for delayed filtering

4 Delayed filtering : Case of multiple connections

Consider now the case of multiple connections running in the same direction and sharing the slow reverse link for their ACKs. Let N be the number of connections. In addition to the problems of end-to-end delay and reverse link utilization, the existence of multiple connections raises the problem of fairness in sharing the reverse link bandwidth. A new connection generating ACKs at less than its fair share from the bandwidth must be protected from other connections exceeding their fair share. Dropping ACKs from a new connection with a small window may be very harmful since the connection may lose its ACK clock and enter in a long Timeout period with a minimum duration of one second in most TCP implementations. We consider in this paper the max-min fairness where the reverse channel bandwidth needs to be equally distributed between the different ACK flows. This is the distribution of the bandwidth we obtained when ACKs are spread over time and when a static ACK filtering with $\delta = 1$ is applied. Our objective is to solve the problem of unfairness caused when applying ACK filtering to a bursty ACK flow rather than finding the appropriate fairness scheme. Other fairness schemes could be studied. As an example, one can imagine to give the ACKs of a new connection more bandwidth than those of already running connections.

We study in this section the performance of different filtering strategies in case of multiple connections. We consider first the case of a large reverse buffer where ACKs are not lost but queued to be served later. There is no requirement for an ACK dropping strategy in this case but rather we need a filtering strategy that limits the queue length, that improves the utilization and that provides a good fairness. Second, we study the case where the reverse buffer is small and where the filtering strategy is not sufficient for good performance since it cannot maintain the queue length to less than the buffer size. ACK filtering needs to be extended by an ACK dropping policy in this case. Consider as an example the case of a static filtering strategy with $\delta = 1$ which is the most aggressive strategy. It results sometimes in a queue of length N ACKs since a one ACK per-connection is authorized in the reverse buffer at a time. This will cause ACK dropping if the reverse buffer size is less than N.

4.1 Case of a large buffer

Applying delayed filtering to the aggregate of ACK flows guarantees the good utilization and the short RTT but it does not guarantee the fairness in reverse channel bandwidth sharing. What we did instead is the application of delayed filtering to

every connection. A list of active connections is maintained in the reverse buffer. For every connection, we store the average rate of its ACKs at the output of the reverse buffer. These rates are updated upon every ACK departure. When an ACK arrives to the reverse buffer, the list of connections is crossed for the corresponding entry. If no entry is found, a new entry for this connection is created with an initial average rate. ACKs belonging to the same connection as that of the arriving ACK are then filtered if the associated average rate exceeds the slow link bandwidth divided by the number of active connections, otherwise the arriving ACK is queued in the buffer without searching the old ACKs for filtering. Now, when an ACK leaves the buffer, the average rates of the different connections are updated. The entry corresponding to one connection is freed when its average rate falls below a certain minimum threshold. A Time Sliding Window algorithm is again used for ACK rate measurement but note here that for connections to which the leaving ACK does not belong we must write,

```
Volume = Rate*Window;
Time = Now - Last + Window;
Rate= Volume /Time;
Last=Now;
```

Keeping an entry per-connection in a buffer seems to be the only problem with these algorithms. We argue that this is now possible with the increase in the processing speed. A per-connection state has been proposed in [13] to calculate the rate of a connection and to control the content of its ACK headers. We argue also that the problem is less important in our case since the number of connections crossing the reverse channel is in general small. In case of a large number of connections, we can group them into flows since as we will see later there is no benefit from continuing with delayed filtering after a certain number of connections. The bandwidth share of a connection becomes small and delayed filtering converges to a static filtering with $\delta = 1$. The problem with flows is that the question of fairness is solved at the flow level not at the connection level. In the sequel, we limited ourselves to a per-connection delayed filtering.

Delaying the filtering of ACKs from a connection separately from the other connections while keeping the classic First-In First-Out service does not result in a complete isolation of connections. The accepted burst of ACKs from an unfiltered connection results in an increase in the RTT of the other connections. A Round Robin scheduling is required for such isolation. The same weight is associated to

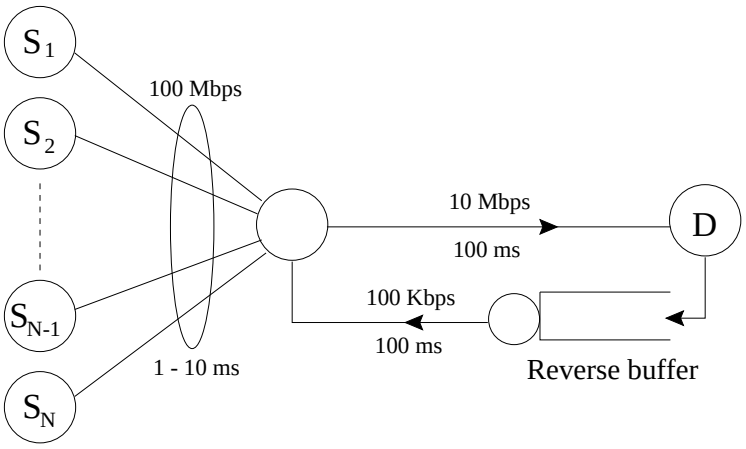

Figure 10: Simulation scenario

the different queues since ACK packets have the same size and since we are looking for a max-min fairness. ACKs from a new connection get queued to be served later during the RTT. This queueing continues until the new connection fills its share of the reverse channel bandwidth.

We implement the different filtering algorithms we cited above into the `ns` simulator [16] and we use the simulation scenario of Figure 10 to compare their performance. N TCP Reno sources transmit short files of size chosen randomly with a uniform distribution between 10 KBytes and 10 Mbytes to the same destination D. The propagation delay of access links is also chosen randomly with a uniform distribution between 1 and 10ms to introduce some phasing in the network. The ACKs of all the transfers return to the sources via a 100 Kbps slow channel. A source S_i establishes a connection to D, transmits a file then releases the connection, waits a small random time and then establishes a new connection to transfer another file to D. We give the reverse buffer a large size and we change the number of sources from 1 to 20. For every N, we run the simulations for 1000s and we calculate the average throughput during a file transfer. We plot the results in Figure 11 where we show the performance as a function of N for five algorithms: the no-filtering algorithm ($\delta = +\infty$), the classic filtering algorithm ($\delta = 1$), the delayed filtering algorithm with all the connections grouped into one flow, the per-connection delayed filtering algorithm with FIFO and with Round Robin scheduling.

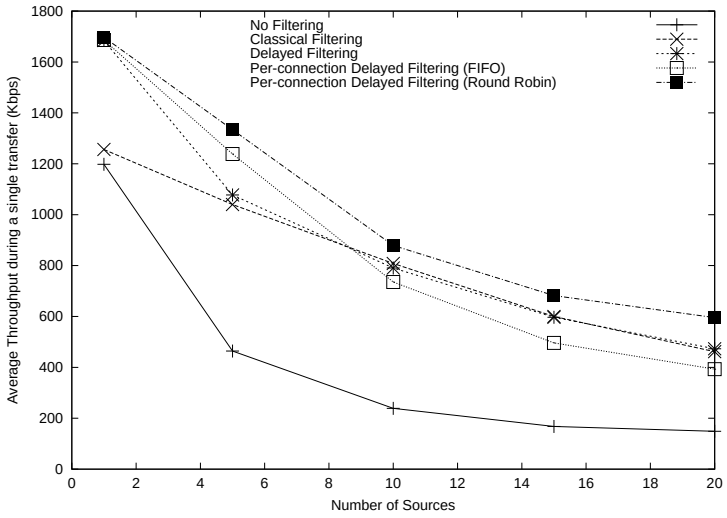

Figure 11: Case of multiple connections and a large buffer

The no-filtering case gives the worst performance due to the long queue of ACKs that builds up in the buffer. Classic filtering solves this problem and improves the performance but it is too aggressive so that it doesn't give the best performance especially for small number of connections. For large number of connections, the fair bandwidth share of a connection becomes small and classic filtering gives performance close to the best policy, the per-connection filtering with Round Robin scheduling. Single-flow delayed filtering is no other than classic filtering beyond a small number of connections and per-connection filtering with FIFO scheduling gives worse performance than the Round Robin scheduling due to the impact of unfiltered connections on the RTT of filtered connections.

4.2 Case of a small buffer

In the previous case, ACK filtering was sufficient to get good performance. An arriving ACK was always able to find a place in the reverse buffer. The question we ask here is, given a certain filtering algorithm, what happens if we decide to queue an ACK and we find that the buffer is full. In fact, this is the open problem of buffer management with a difference here in that the flows we are managing are not responsive as TCP data flows. The other difference is that in our case dropping is

preceded by filtering which reduces the overload of ACKs on the reverse buffer. The buffer management policy is used only in the particular case where filtering is not enough to avoid the reverse buffer overflow. We can see the relation between filtering and dropping as two consecutive boxes. The first box which is the filtering box tries to eliminate the unnecessary information from the flow of ACKs. The filtered flow of ACKs is then sent into the second box which contains the reverse buffer with the appropriate drop policy. ACKs are dropped when the buffer overflows. We associate in this section a drop policy to every algorithm we proposed and we compare the performance under a small buffer assumption.

For classic filtering ($\delta = 1$), we use the normal drop tail policy. The buffer space is fairly shared between the different connections (one ACK per connection) and we don't have enough information to use another more intelligent drop policy. Note that in order to avoid the per-connection state, most of the current drop policies [18] use the content of the buffer to get an idea on the bandwidth sharing. The same drop tail policy is used in case of single-flow delayed filtering when ACKs are filtered. When ACKs are not filtered, we use the Longest Queue Drop policy described in [18]. The oldest ACK of the connection with the longest queue is dropped. Now, for the per-connection delayed filtering case, we profit from the information available for filtering in the dropping procedure. The oldest ACK of the connection having the highest rate is dropped. Recall that the oldest ACK of a connection is the first ACK of this connection encountered when sweeping the buffer from its head. In case of Round Robin scheduling, it is the ACK at the head of the queue associated to the connection.

We repeat the same simulation of the previous section but now with a small reverse buffer of 10 packets. The average throughput is shown as a function of the number of sources in Figure 12. In this case, the RTT is not very important and the difference in performance is mainly due to the difference in fairness. The difference in fairness is in turn due to the difference in the drop policy. Per-connection delayed filtering with Round Robin scheduling gives the best performance since it guarantees the best isolation between flows. Single-flow delayed filtering gives better performance than classic filtering in case of small number of connections but it converges to classic filtering when N increases. The case of the classic filtering algorithm at large number of connections is a little surprising. The major problem of the no-filtering algorithm is that the reverse buffer is almost full which causes many losses from a new connection. This connection is blocked until the dominant connections reduce their rates. One of the objectives of ACK filtering was to free some space in the reverse buffer for ACKs from a new connection. But, as the number of concurrent

connections approaches the buffer size, the empty space in the buffer moves to zero and ACKs from new connections tend again to be lost. The performance of classic filtering moves to that of no-filtering as N increases. Moreover, we see that it drops below the no-filtering case when N is larger than the buffer capacity.

5 Conclusions

We studied in this paper the advantage of delaying the filtering of ACKs until the full utilization of the slow reverse channel. This has an important impact on the slow start phase of the connection which requires the arrival of the maximum amount of ACKs to increase faster the congestion window. We studied first static filtering where the length of the allowed ACK queue is fixed to a certain value. Then we presented some algorithms that use the rate of ACKs or the presence rate of ACKs in the buffer as an indication to start filtering. Per-connection filtering with Round Robin scheduling gives the best performance. We saw also that grouping a small number of connections into a single flow and applying delayed filtering to the aggregate flow results in performance close to applying delayed filtering to each connection separately. This means that grouping connections into flows will be a good solution to extend our algorithms to the case of a large number of connections.

References

[1] M. Allman et al., " Ongoing TCP Research Related to Satellites", *Internet Draft*, work in progress, Sep 1999.

[2] E. Altman et al., "Performance Modeling of TCP/IP in a Wide-Area Network", *IEEE Conference on Decision and Control*, Dec 1995.

[3] H. Balakrishnan, V. Padmanabhan, and R. Katz, "The Effects of Asymmetry on TCP Performance", *ACM Mobicom*, Sep 1997.

[4] C. Barakat and E. Altman, "Performance of Short TCP Transfers", *Networking 2000 (Performance of Communications Networks)*, May 2000.

[5] C. Barakat, E. Altman, and W. Dabbous, "On TCP Performance in a Heterogeneous Network : A Survey", *IEEE Communications Magazine*, Jan 2000.

[6] Neal Cardwell, Stefan Savage, and Tom Anderson, "Modeling TCP Latency", *IEEE Infocom*, Mar 2000.

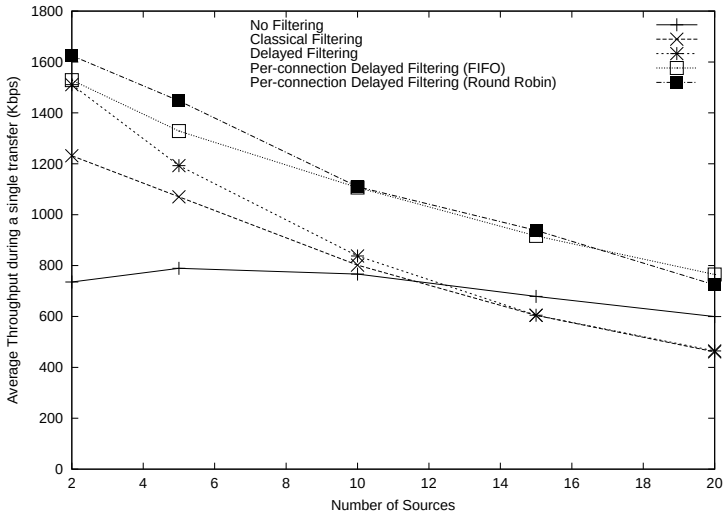

Figure 12: Case of multiple connections and a small buffer

[7] D. Clark and W. Fang, "Explicit Allocation of Best Effort Packet Delivery Service", *IEEE/ACM Transactions on Networking*, Aug. 1998.

[8] R. Durst, G. Miller, and E. Travis, "TCP Extensions for Space Communications", *ACM Mobicom*, Nov 1996.

[9] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", *ACM Computer Communication Review*, Jul 1996.

[10] S. Floyd and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Aug. 1993.

[11] V. Jacobson, "Congestion avoidance and control", *ACM Sigcomm*, Aug 1988.

[12] V. Jacobson, "Compressing TCP/IP Headers for Low-speed Serial Links", *RFC 1144*, Feb 1990.

[13] S. Karandikar, S. Kalyanaraman, P. Bagal, and Bob Packer, "TCP Rate Control", *ACM Computer Communication Review*, Jan 2000.

[14] T.V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss", *IEEE/ACM Transactions on Networking*, Jun 1997.

[15] T. V. Lakshman, U. Madhow, and B. Suter, "Window-based error recovery and flow control with a slow acknowledgment channel: a study of TCP/IP performance", *IEEE Infocom*, 1997.

[16] The LBNL Network Simulator, *ns*, http://www-nrg.ee.lbl.gov/ns.

[17] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", *RFC 2001*, 1997.

[18] B. Suter, T.V. Lakshman, D. Stiliadis, and A.K. Choudhary, "Design Considerations for Supporting TCP with Per-flow Queueing", *IEEE Infocom*, Mar 1998.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399