On ACK Filtering on a Slow Reverse Channel *

Chadi Barakat and Eitan Altman
INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis, France
{cbarakat,altman}@sophia.inria.fr

Abstract. ACK filtering has been proposed as a technique to alleviate the congestion on the reverse path of a TCP connection. In the literature the case of a one-ACK per connection at a time in the buffer at the input of a slow channel has been studied. In this paper we show that this is too aggressive for short transfers. We study first static filtering where a certain ACK queue length is allowed. We show analytically how this length needs to be chosen. We present then some algorithms that adapt the filtering of ACKs as a function of the slow channel utilization rather than the ACK queue length.

1 Introduction

Accessing the Internet via asymmetric paths 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 (ACK) via a slow reverse channel (e.g., a dial-up modem line at 64 kbps). Figure 1 shows an example of such asymmetric path. It has been shown [1, 3, 5, 8, 13] that the slowness of the reverse channel limits the throughput of TCP transfers [10, 15] running in the forward direction. A queue of ACKs builds up in the buffer at the input of the reverse channel (we call it the *reverse buffer*) causing an increase in the round trip time (RTT) and an overflow of the buffer (i.e., loss of ACKs). This results in a performance deterioration for many reasons. First, an increase in RTT reduces the throughput since the transmission rate of a TCP connection is equal at any moment to the window size divided by RTT. Second, the increase in RTT as well as the loss of ACKs slows the window increase. 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) to recover from losses without timeout [9]. Another problem has been also 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 [13]. Due to the overflow of the reverse buffer, the new 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 latter problem as a result of an unfairness in the distribution of the slow channel bandwidth between the different flows. The main reason for such unfairness is that a flow of ACKs is not responsive to drops as a TCP data flow.

* A detailed version of this paper is available as an INRIA Research Report at http://www.inria.fr/RRRT/RR-3908.html

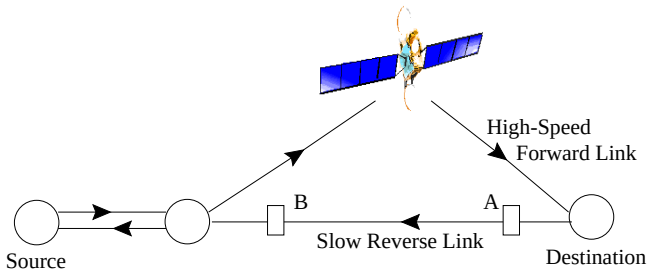

Fig. 1. An asymmetric path

Many solutions have been proposed for this problem of bandwidth asymmetry. Except the header compression solution (e.g., the SLIP algorithm in [11]) which 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 match 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 occupancy. ACK filtering however requires only modification at the reverse buffer. It profits from the cumulative nature of ACKs. An ACK can be safely substituted by a subsequent ACK carrying a larger sequence number. From 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.

In this paper we ask the question of how many ACKs from a connection we must queue in the reverse buffer before we start filtering. In the literature the case of a one ACK per-connection at a time has been studied [1, 3]. When an ACK arrives and before being queued, the reverse buffer erases any ACK from the same connection. Clearly, this behavior optimizes the end-to-end delay and the queue length, but it ignores the fact that TCP uses ACKs to increase its window. This may not have an impact on the congestion avoidance phase. However, it has certainly an impact on slow start where the window is small and needs to be increased as quick as possible to achieve good performance. The impact on slow start comes from the fact that TCP is known to be bursty during this phase [2, 4, 12], and thus ACKs arrive in bursts at the reverse buffer. ACKs may also arrive in bursts due to some compression of data packets or ACKs in the network. Filtering bursts of ACKs will result in few ACKs reaching the source and then in a slow window increase. This negative impact of ACK filtering will be important on short transfers which dominate most of today's Internet traffic [6] and where most of the transfer is done during slow start. In particular, it will be pronounced over 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 the start of filtering will have the advantage of absorbing these bursts of ACKs which will result in faster window increase. However, this threshold must be kept at a small value in order to limit the end-to-end delay. A certain tradeoff then appears; 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 when the ACK filtering threshold (the number of ACKs that a connection can queue) is set to a fixed value. We show analytically how this threshold must be chosen. We present then our algorithm, we call *Delayed ACK Filtering*, that adapts the filtering of ACKs as a function of the slow channel utilization rather than the ACK queue length. This is equivalent 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, our algorithm adapts the filtering in a way to share fairly the slow channel bandwidth between the different connections.

2 Impact of ACK Filtering Threshold

We focus on short transfers which are considerably affected by the slow start phase. We consider first the case of a single transfer. The burstiness of ACKs caused by slow start itself is considered. Our objective is to show that delaying the filtering until a certain number of ACKs get queued, shortens the slow start phase and improves the performance if this threshold is correctly set. We assume in the sequel that router buffers in the forward direction are large enough so that they absorb the burstiness of traffic resulting from the filtering of ACKs. We don't address later the problem of burstiness of traffic in the forward direction since our algorithms reduce this burstiness compared to the classical one-ACK at a time filtering strategy.

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 RTT (in absence of queueing delay). μ_r is measured in terms of ACKs per unit of time. Assume that RTT increases only when ACKs are queued in the reverse buffer. This happens when the reverse channel is fully utilized, which in turn happens when the number of ACKs arriving at the reverse buffer per T is more than $\mu_r T$. In the case of no queueing in the reverse buffer, RTT is taken equal to T. This assumption holds given the considerable slowness of the reverse channel with respect to the other links on the path.

Assume for the moment that the reverse buffer is large and that ACKs are not filtered. The window at the sender grows then exponentially with a rate function of the frequency at which the receiver acknowledges packets. Recall that we are working in the slow start mode where the window is increased by one packet upon every ACK arrival [10]. 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

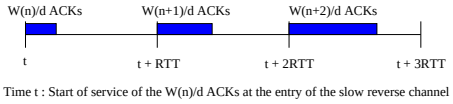

Fig. 2. Bursts of ACKs as they cross the reverse channel

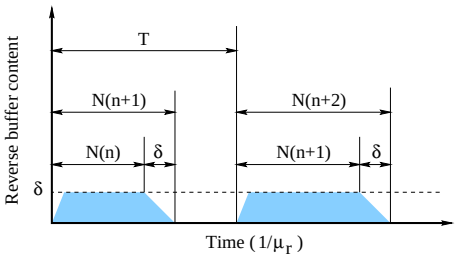

Fig. 3. Reverse buffer occupancy as a function of time

$(d = 2)$ [15]. 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 channel is fully utilized, ACKs start to arrive at the source at a constant rate μ_r. Here, the window together with RTT start to increase linearly with time. The transmission rate, which is equal to the window size divided by RTT, stops increasing and becomes limited by the reverse channel. This continues until ACKs start to be filtered or dropped. RTT stops then increasing and the transmission rate resumes its increase with the window size (see Figure 5). The first remark we can make here is that the ACK queue length needs to be maintained at small values in order to get a small RTT and a better performance. An aggressive filtering (say one ACK per-connection) is then needed. But, due to the fast window increase during slow start, ACKs may arrive at the reverse buffer in separate bursts during which the rate is higher than μ_r, without having an average rate higher than μ_r (see [4] for a description of slow start burstiness). An aggressive filtering will reduce the number of ACKs reaching the source whereas these bursts can be absorbed without causing any increase in RTT. Such absorption will result in a faster window increase. Given that RTT remains constant whenever the reverse channel is not fully utilized, a faster window increase results in a faster transmission rate increase and thus in a better performance. The general guideline for ACK filtering is to accept all ACKs until the slow channel becomes fully utilized, and then to filter them in order to limit the RTT. We consider first the case when a connection is allowed to queue a certain number of ACKs in the reverse buffer. This number, which we denote by δ and which we call the ACK filtering threshold, is maintained constant during the connection lifetime. We study the impact of δ on the performance and we show how it must be chosen. Later, we present algorithms that adapt ACK filtering as a function of the slow channel utilization. This permits a simpler implementation together with a better performance than fixing a certain number of ACKs.

2.2 ACK Filtering Threshold

During slow start, TCP is known to transmit packets in long bursts [4]. 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 its input. During the 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 then the absorption of a queue of length $\mu_r T/(d+1)$. This is the value of δ_o, the optimum filtering threshold, the buffer must use:

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

2.3 Early ACK Filtering

We consider now the case when $\delta < \delta_o$. When an ACK finds δ ACKs in the buffer, the most recently received ACK is erased and the new ACK is queued at its place. We call this a *static* filtering strategy since δ is not changed during the connection lifetime. Let us study the impact of δ on the window increase rate.

Starting from $W(0) = 1$, the window increases 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 $\delta(d+1)$ ACKs 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 slows. To show the impact of δ on the window increase, we define the following variables: Consider $n > n_0$ and put ourselves in the region where the slow channel is not fully utilized. We know that the maximum window increase rate (μ_r packets per unit of time) 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 $N(n)$ denote the number of ACKs that leave the slow channel during round trip n. Given that we are in slow start, we have $W(n+1) = W(n) + N(n)$.

The burst of data packets of length $W(n+1)$ generates $W(n+1)/d$ ACKs at the destination which reach the reverse buffer at a rate faster than μ_r. The duration of this burst of ACKs 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 large 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, a queue of δ ACKs is formed in the reverse buffer and the slow channel transmits $N(n)$ ACKs. The ACKs stored in the reverse buffer whose number is equal to δ are then sent. Thus, $N(n+1) = N(n) + \delta$. Figure 3 shows the occupancy 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 for $n > n_0$,

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

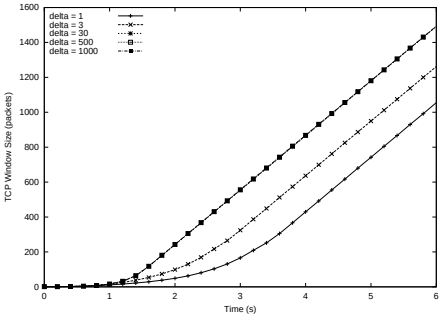

Fig. 4. TCP window vs. time

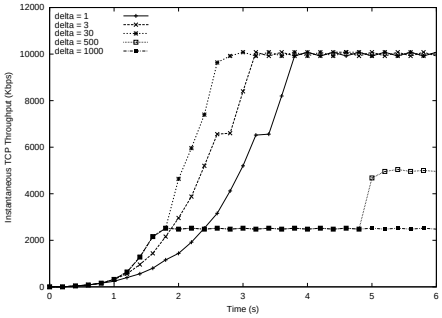

Fig. 5. Transmission rate vs. time

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

We remark that due to the small value of δ, the window increase changes from exponential to polynomial and it slows with δ. The source spends more time before reaching the maximum window increase rate of μ_r packets per unit of time.

2.4 Simulation

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 channel of 100 kbps back to the destination. T is set to 200 ms. We use the `ns` simulator [14] 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 reverse buffer 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 transmission rate (Figure 5), and the last acknowledged sequence number (Figure 6). The transmission rate is averaged over intervals of 200 ms.

For such scenario the calculation gives a δ_o equal to 30 ACKs (Equation (1)). A δ less than δ_o slows the window growth. We see this for $\delta = 1$ and $\delta = 3$ in Figure 4. The other values of δ give the same window increase given that the filtering starts after the full utilization of the reverse channel, and thus the maximum window increase rate is reached at the same time. But, the window curve is not sufficient to study the performance since the same window may mean different performance if RTT is not the same. We must also look at the plot of the transmission rate (Figure 5). For small δ, RTT can be considered as always constant and the curves for the transmission rate and for the window have the

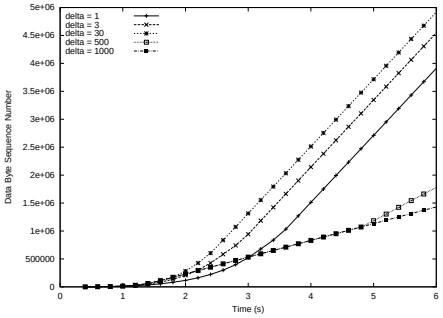

Fig. 6. Last acknowledged sequence number vs. time

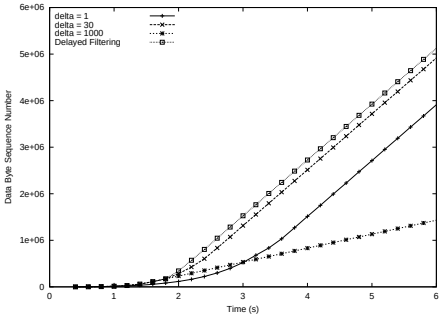

Fig. 7. Last acknowledged sequence number vs. time

same form. However, the transmission rate 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 $\delta = 1$). This latter saturation corresponds to the time between the full utilization of the reverse channel and the convergence of RTT to its limit ($T+\delta/\mu_r$) when δ ACKs start to be always present in the reverse buffer. During the convergence, the transmission rate remains constant due to a linear increase of both the window and RTT. Once the RTT stabilizes, the transmission rate resumes its increase with the window. Note that the stabilization of RTT takes a long time for large δ (around 5s for $\delta = 500$). Now, Figure 6 is an indication of the overall performance. We see well how taking $\delta = \delta_o$ leads to the best performance since it gives 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 $\delta = \delta_o$ then worsens. This conclusion may not be valid for long transfers where slow start has a slight impact on the overall performance.

3 Delayed Filtering : Case of a Single Connection

Tracking the queue length for filtering is not a guarantee for good performance. First, in reality and due to the fluctuation of the exogenous traffic, the arrival of ACKs may be completely different than the theoretical arrival we described. Second, the calculation of the optimum threshold (Equation (1)) is difficult since it requires knowledge of RTT and the acknowledgement strategy (d). Third, setting the filtering threshold to a fixed value leads to an unnecessary increase in RTT in the steady state. 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. For the first and second problems, one can imagine to set the filtering threshold to the bandwidth-delay product of the return path ($\mu_r T$) in order to account for the most bursty case. The cost to pay here is a further increase in RTT in the steady state.

The simplest solution is to measure the rate of ACKs at the output of the reverse buffer and to compare it to the channel bandwidth. Measuring the rate at the output rather than at the input is better since ACKs are spread over

time which increases the precision of the measurement tool. In case we don't know the channel bandwidth (e.g., case of a shared medium), one can measure how frequently ACKs are present in the reverse buffer. When the measurement indicates that the channel is fully utilized (e.g., the utilization exceeds a certain threshold that we fix to 90% in our simulations), we start to apply the classical filtering studied 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 filtering where δ is set to infinity when the slow channel 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 in the case of a single connection. The case of multiple concurrent connections is studied later.

3.1 Utilization Measurement

We assume that the slow channel bandwidth is known. 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 the rest from the past. The difference from classical low pass filters is that the decay time of the rate with the TSW algorithm is a function of the current average rate not only the frequency of measurements. The coefficient of contribution of the new measurement is more important at low rates than at high rates. This guarantees a fast convergence in case of low rates. 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` be the current time, `Size` be the size of the packet (40 bytes for ACKs). Then, upon ACK departure,

```
1) Volume = Rate*Window + Size;     2) Time = Now - Last + Window;
3) Rate= Volume /Time;              4) Last=Now;
```

The same algorithm with a slight change can be applied to measure how frequently ACKs are present in the reverse buffer.

3.2 Simulation

We consider the same simulation scenario. We implement delayed filtering in the reverse buffer and we plot its performance. The time window is taken in the same order as RTT. Figure 7 shows how our algorithm gives as good performance as the case where δ is correctly chosen. Moreover, it outperforms slightly the case of static filtering with $\delta = \delta_o$ due to the erasing of all ACKs once the slow channel is fully utilized. The behavior of delayed filtering can be clearly seen in Figures 8 and 9 where we plot the reverse buffer occupancy. These figures correspond to

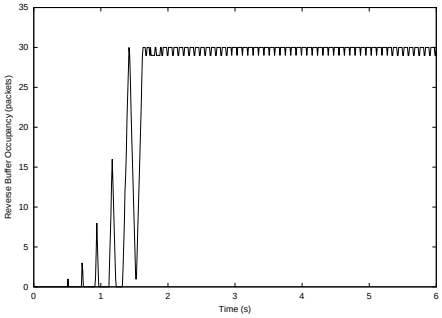

Fig. 8. Reverse buffer occupancy for static filtering

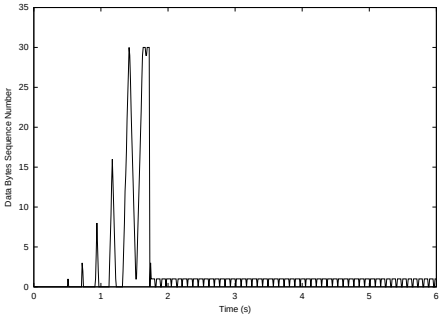

Fig. 9. Reverse buffer occupancy for delayed filtering

static filtering with $\delta = \delta_o = 30$ 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 time in the two cases.

4 Delayed Filtering : Case of Multiple Connections

Consider now the case of multiple connections running in the same direction and sharing the slow reverse channel for their ACKs. Let N be the number of connections. In addition to the problems of end-to-end delay and reverse channel utilization, the existence of multiple connections raises the problem of fairness in sharing the reverse channel 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. We consider in this paper a max-min fairness where the slow channel bandwidth needs to be equally distributed between the different ACK flows. However, other fairness schemes could be studied. As an example, one can imagine to give ACKs from a new connection more bandwidth than ACKs from already running connections. We study the performance of different filtering strategies. We consider first the case of a large reverse buffer where ACKs are not lost but queued to be served later. There is no need here for an ACK drop strategy but rather for 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 ACK filtering is not enough to maintain the queue length at less than the buffer size. ACK filtering needs to be extended in this second case by an ACK drop policy.

4.1 Case of a Large Buffer

To guarantee a certain fairness, we apply 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. When an ACK arrives, the list of connections is checked. If no entry is found, a new entry is created for this connection. Now, if the average rate associated to the connection of the new ACK exceeds the slow channel bandwidth divided by the number of active connections, all ACKs belonging to the same connection

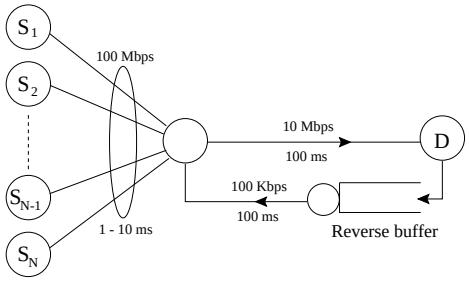

Fig. 10. Simulation scenario

are filtered and the new ACK is queued at the place of the oldest ACK. When an ACK leaves the buffer, the average rates of the different connections are updated. A TSW algorithm is again used for ACK rate measurement. The entry corresponding to a connection is freed once its average rate falls below a certain threshold.

Keeping an entry per-connection seems to be the only problem with our algorithm. We believe that with the increase in processing speed, this problem does not exist. Also and as we will see later, we can stop our algorithm beyond a certain number of connections since it converges to static filtering with $\delta = 1$. This happens when the fair bandwidth share of a connection becomes very small. Classical filtering can be applied in this case without the need to account for full utilization.

Now, delaying the filtering of ACKs from a connection separately from the other connections while keeping the classic FIFO service does not result in a complete isolation. The accepted burst of ACKs from an unfiltered connection increases the RTT seen by other connections. A round robin (RR) scheduling is required for such isolation. ACKs from filtered connections will no longer need to wait after ACKs from an unfiltered one.

We implement the different filtering algorithms we cited above into `ns` and we use the simulation scenario of Figure 10. N TCP Reno sources transmit short files of sizes chosen randomly with a uniform distribution between 10 kbytes and 10 Mbytes to the same destination D. The propagation delays of access links are chosen randomly with a uniform distribution between 1 and 10 ms. ACKs from all the transfers return to the sources via a 100 kbps slow channel. A source S_i transmits a file to D, waits for a small random time, and then transmits another file. We take a large reverse buffer 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 in Figure 11 the performance as a function of N for five algorithms: no-filtering ($\delta = +\infty$), classical filtering ($\delta = 1$), delayed filtering with all the connections grouped into one flow, per-connection delayed filtering with FIFO and with RR scheduling.

No-filtering gives the worst performance due to the long queue of ACKs in the reverse buffer. Classical filtering solves this problem but it is too aggressive so it does not give the best performance especially for small number of connections.

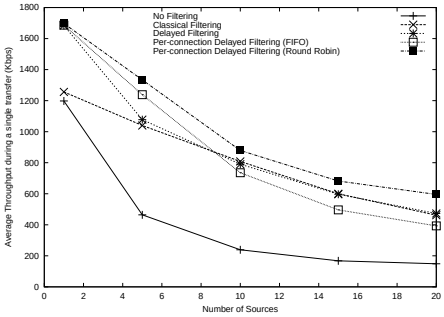

Fig. 11. Case of multiple connections and a large buffer

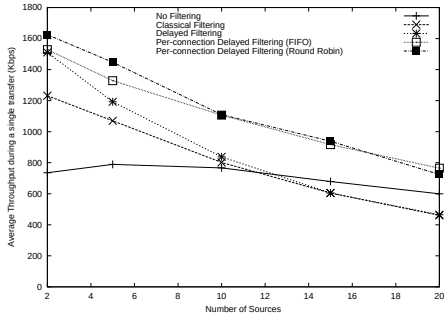

Fig. 12. Case of multiple connections and a small buffer

For large number of connections, the bandwidth share of a connection becomes small and classical filtering gives close performance to that of the best policy, per-connection filtering with RR scheduling. Single-flow delayed filtering is no other than static filtering beyond a small number of connections, and per-connection filtering with FIFO scheduling gives worse performance than RR scheduling due to the impact of unfiltered connections on the RTT of filtered ones.

4.2 Case of a Small Buffer

The question we ask here is 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 to drops as TCP data flows. The other difference is that in our case, ACK dropping is preceded by ACK 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.

For classical filtering 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. 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 [16]. The oldest ACK of the connection with the longest queue is dropped and the new ACK is queued at the end. Now, for per-connection delayed filtering, we profit in the drop procedure of the information available for filtering. The oldest ACK of the connection with the highest rate is dropped.

We repeat the same simulation of the previous section but now with a small reverse buffer of 10 packets. The average throughput is shown in Figure 12 as a function of the number of sources. In this case and especially at large N, the difference in performance is mainly due to the difference in the drop policy. This can

be seen from the difference in performance at large N between per-connection delayed filtering and classical filtering. If the drop policy is not important, these two filtering strategies should give similar performance. Per-connection delayed filtering with RR scheduling gives again the best performance. The relative position of classical filtering to no-filtering is a little surprising. When the number of connections is smaller than the buffer size, classical filtering is able to keep an empty place for a new connection and then to protect it from already running connections. This leads to better performance than in case of no-filtering. However, as the number of connections exceeds the buffer size, the reverse buffer starts to overflow and new connections will no longer be protected. Thus, the performance of classical filtering deteriorates when N increases, and it drops below that of no-filtering for large N. We conclude that when the number of connections is larger than the buffer size, a simple drop tail policy is enough for good performance. This again limits the number of connections that our delayed filtering algorithm need to track.

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.
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. V. Jacobson, "Congestion avoidance and control", *ACM SIGCOMM*, Aug 1988.
11. V. Jacobson, "Compressing TCP/IP Headers for Low-speed Serial Links", *RFC 1144*, Feb 1990.
12. 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.
13. 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.
14. The LBNL Network Simulator, *ns*, http://www-nrg.ee.lbl.gov/ns.
15. W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", *RFC 2001*, 1997.
16. B. Suter, T.V. Lakshman, D. Stiliadis, and A.K. Choudhary, "Design Considerations for Supporting TCP with Per-flow Queueing", *IEEE INFOCOM*, Mar 1998.