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 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 one-ACK per connection at a time has been studied. In this paper we show that this latter scheme 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 possible number of ACKs in order to quickly increase its congestion window. We study first static filtering where a fixed ACK queue length is allowed. We show analytically how this length needs to be chosen. We present then some mechanisms that adapt the filtering of ACKs as a function of the slow channel utilization rather than the ACK queue length. These mechanisms provide a good compromise between reducing the ACK queueing delay and passing to TCP source a large number of ACKs that guarantee a fast window increase.

Keywords

TCP, Slow start, Asymmetric paths, ACK filtering, Buffer management.

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 (ACK) via a slow reverse channel (e.g., a dial-up modem line at 64 kbps). This kind of paths is called asymmetric path and is often studied in the literature [1, 3, 6, 9, 16]. Figure 1 shows an example of such asymmetric path. It has been shown that the slowness of the reverse channel limits the throughput of TCP transfers running in the forward direction. TCP [12, 18], the most widely used Internet protocol, is known to generate a large number of ACKs which are necessary for the sender operation. ACKs are used as a clock that triggers the transmission of new data, slides and increases the congestion window. From TCP point of view, an asymmetric path has been defined as a 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

[*]For more description of the problem studied in this paper and of the proposed mechanisms, we refer to [4].

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 throughput 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 slow 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 and slow start [10]. 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 [16]. 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. One can see this latter 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 channel until they reduce their transmission rates due to the loss of a data packet in the forward direction, 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 [13]) 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, 9], 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 the information carried by ACKs. Indeed, a TCP 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 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 ACKs from the same TCP connection (by comparing IP addresses and port number), 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 of how many ACKs we must authorize a connection to queue in the reverse buffer before ACK filtering starts. In the literature, the case of one ACK per-connection at a time has been studied [1, 3]. When an ACK arrives at the reverse channel 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 congestion 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, 5, 15], 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 a

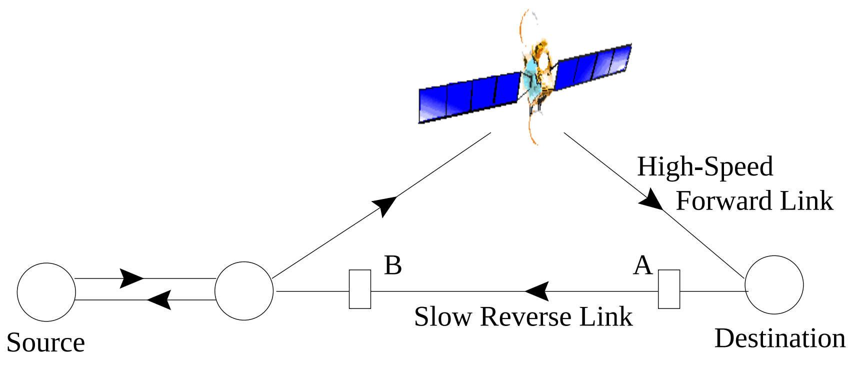

Figure 1: An asymmetric path

small number of ACKs reaching the source and then in a slow window increase. This negative impact of ACK filtering will be important on short transfers which are the majority in today's Internet traffic [7], and where most of, if not all, the bytes of the transfer are transmitted during slow start. The impact will be more pronounced over long delay links (e.g., satellite links) where slow start is already slow enough [6]. 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 a faster window increase. However, this threshold must be kept at a small value in order to limit the end-to-end delay. A clear tradeoff appears then; one must predict an improvement of the performance as the threshold increases, followed by a deterioration of the performance when the threshold 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 fixed. We show analytically how this threshold must be chosen. We present then our mechanism called *Delayed ACK Filtering*, that adapts the filtering of ACKs as a function of the slow channel utilization rather than the ACK queue length. Our mechanism is equivalent to a dynamic setting of the ACK filtering threshold. The objective is to pass as many ACKs as possible to the TCP source while maintaining the end-to-end delay at small values. In case of many connections, our mechanism adapts the filtering in a way to distribute fairly the slow channel bandwidth among the different connections.

In the next section, we describe the impact of ACK filtering on the performance of TCP. We show how delaying ACK filtering until a certain queue length is built, accelerates slow start and improves the performance without impacting the end-to-end delay. In Section 3, we present our adaptive mechanism that does not relate filtering to the number of queued ACKs, but rather to the channel utilization or to the ACK presence in the reverse buffer. Our propositions are validated by simulations on `ns-2`, the network simulator developed at LBNL [17]. In Section 4, we study the case of multiple connections and we extend our mechanism to this case. Section 5 concludes the paper.

2 Impact of ACK Filtering Threshold

In contrast to the other works in the literature studying the impact of bandwidth asymmetry on long-lived TCP transfers, we focus on short transfers which are considerably affected by the slow start phase. Long-lived transfers are dominated by the congestion avoidance phase where an aggressive ACK filtering has no great impact since the window is usually large and the burstiness of ACKs is less important than in slow start. We study 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 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 the buffers in routers located in the forward direction are large enough so that they absorb the burstiness of traffic resulting from the filtering of ACKs. We will not 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 the RTT (in the 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. Note here that due to the burstiness of TCP traffic, data packets belonging to the same window may experience different RTTs if their corresponding ACKs get queued in the reverse buffer behind previous ACKs belonging to the same window. Some data packets may then see a longer RTT than T, even if the reverse channel is not fully utilized. Later, we explain how we measure the RTT. For the moment, we take RTT as the smallest round-trip time seen by a window of data packets.

For instance, assume that the reverse buffer is large and that ACKs are not filtered. This assumption will be removed later. The congestion window at the TCP sender grows then exponentially at a rate that is function of the frequency at which the receiver acknowledges packets. Recall that we are working in the slow start mode where TCP congestion window is increased by one packet upon every ACK arrival [12]. 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 ($d = 2$) [18]. 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 illustrates the exponential increase of the window during slow start.

Once the reverse channel is fully utilized, ACKs start to arrive at the source at a constant rate equal to 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 in the reverse buffer. The RTT stops then increasing and the transmission rate resumes its increase with the window size (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

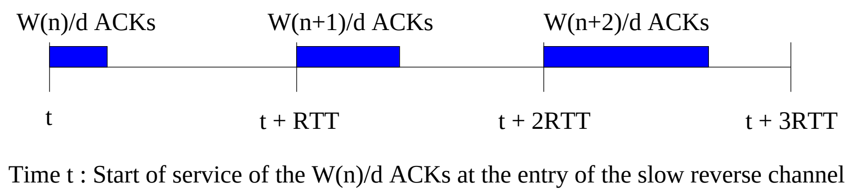
W(n)/d ACKs
W(n+1)/d ACKs
W(n+2)/d ACKs
t
t + RTT
t + 2RTT
t + 3RTT
Time t : Start of service of the W(n)/d ACKs at the entry of the slow reverse channel

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

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 [5] for a description of slow start burstiness). An aggressive filtering reduces then the number of ACKs reaching the source, while 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 will result in a faster transmission rate increase and thus in a better performance. The general guideline for ACK filtering is therefore 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 mechanisms that adapt ACK filtering as a function of the slow channel utilization. This allows a simpler implementation together with a better performance than fixing a certain number of ACKs to be queued for the start of filtering. The study in this section is however necessary to understand how the performance of TCP varies as a function of the filtering threshold.

2.2 ACK Filtering Threshold

We compute the maximum queue length the reverse buffer must allow before the start of filtering. The objective must be to absorb the bursts of ACKs caused by slow start and to start filtering ACKs just after the full utilization of the slow channel. We call this queue length the *optimal filtering threshold*, and we denote it by δ_o. We study then in the next section the impact on the window increase of a filtering threshold $\delta < \delta_o$. We recall that our study focuses on the slow start phase of short TCP transfers.

During slow start, TCP is known to transmit data packets in long bursts [5]. 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 proven by noticing 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. 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 in the reverse buffer. 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 $\mu_r T/(d+1)$ by the reverse buffer. This is the optimal δ the reverse buffer must use,

$$\delta_o = \mu_r T/(d+1). \qquad (1)$$

This threshold guarantees that the reverse channel is fully utilized before the start of ACK filtering, which is necessary to obtain a fast window increase during slow start. Note that the optimal threshold δ_o increases with RTT and the slow channel rate. It also increases with the volume of generated ACKs.

2.3 Early ACK Filtering

We consider now the case where the ACK filtering threshold δ is set to less than δ_o. When an ACK arrives and finds δ ACKs in the reverse 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. A $\delta = 1$ corresponds to the ACK filtering strategy studied in the literature [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 ACKs 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 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 of time by a multiplicative 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 evaluate the impact of δ on the window increase rate, we define some new variables, in addition to the those of the previous section. We look at round-trip n, and we put ourselves in the region where the filtering is active but the slow channel is not fully utilized, i.e., $n > n_0$. 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 regime 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, 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 slow reverse channel at a rate faster than μ_r (it is even faster than $\alpha\mu_r$ 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 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

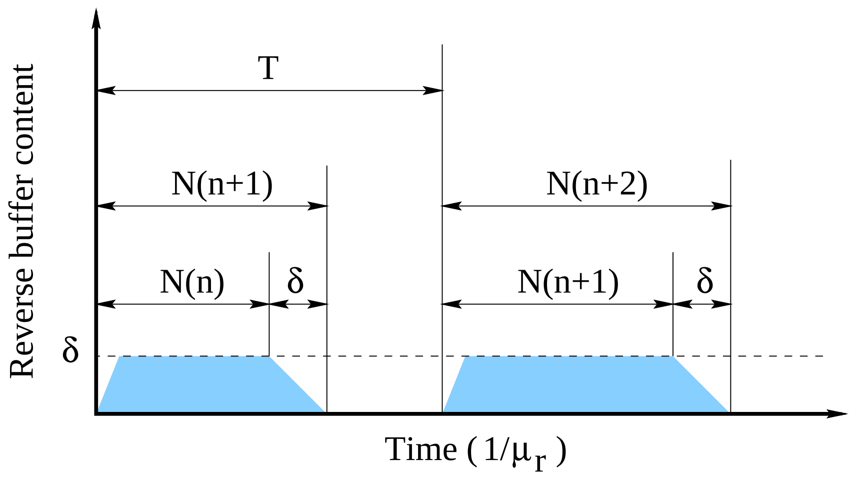

Figure 3: Reverse buffer content as a function of time

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 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 $n > n_0$,

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

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

We remark that due to the small value δ (less than δ_o), 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. We can also conclude this behavior 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 multiplied by a factor α every T.

The question that one may ask here is whether the δ ACKs we let queue in the reverse buffer to absorb the bursts of ACKs and to accelerate the window increase have an impact on the round-trip time 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 lose due to this increase in 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.

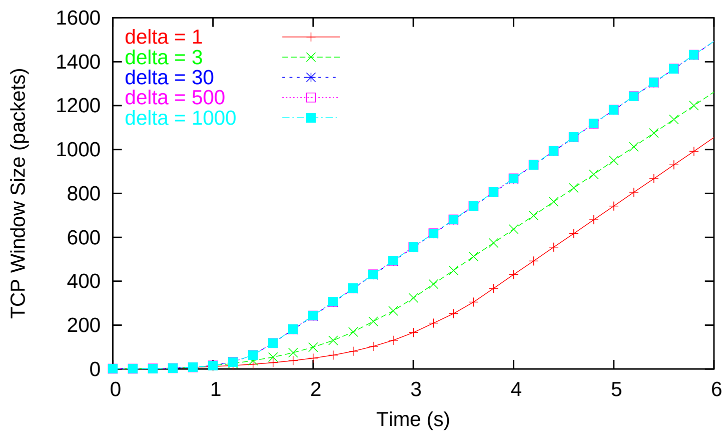

Figure 4: TCP window vs. time

The appropriate filtering strategy is the one that starts with a large δ, then 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 channel utilization and the adaptation of the filtering of ACKs as a function of this utilization rather than of the queue size. The impact of this optimal strategy on the performance of TCP will be illustrated when we present and validate our adaptive mechanism for ACK filtering.

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 channel of 100 kbps back to the destination. The Round-Trip Time T is set to 200 ms. We use the `ns` simulator [17] and we monitor the packets sent during the slow start phase at the beginning of the connection. We implement in `ns` an ACK filtering strategy that limits the number of ACKs in the reverse buffer to δ, and we compare the performance of TCP for different values of δ. The reverse 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 transmission rate (Figure 5), and the lastly acknowledged byte sequence number (Figure 6). The transmission rate is average over intervals of 200 ms.
For such a scenario, the analysis gives δ_o equal to 30 ACKs (Equation (1)). A δ less than δ_o slows the window growth and requires more time to reach the linear window increase phase (once the slow channel becomes fully utilized). We see this with the lines $\delta = 1$ and $\delta = 3$ in Figure 4. The other values of δ give the same window increase since the filtering of ACKs 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 the RTT is not the same. We must 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 size have the same form. However, the transmission rate saturates when it reaches the available bandwidth in the forward direction

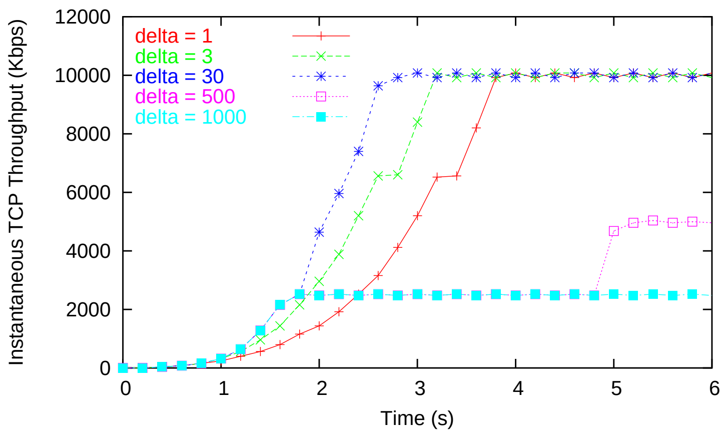

Figure 5: Transmission rate vs. time

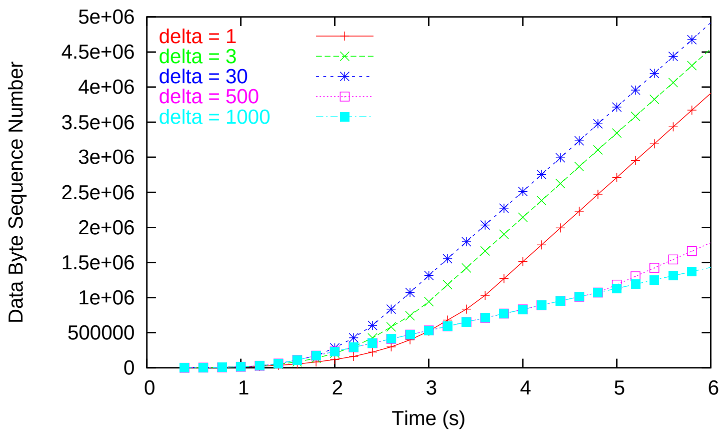

Figure 6: Last acknowledged sequence number vs. time

(at 10,000 Kbps). It also saturates somewhere in the middle (e.g., at 3s for $\delta = 1$ and at 1.5s for $\delta \geq 30$). This latter 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 RTT is T. We also know that in the absence of queueing in the forward direction, the maximum value of the RTT is seen when δ ACKs start to be always present in the reverse buffer. The maximum value of RTT is then equal to T plus the transmission time of δ ACKs on the slow channel, i.e., $T + \delta/\mu_r$. During the convergence of RTT, the transmission rate of TCP remains constant due to a linear increase of both the window size and the RTT. Once the RTT stabilizes, the transmission rate resumes its increase with the window. For small δ, the convergence of RTT is quick and the saturation of the transmission rate lasts for a short time. However, for large δ, a long time is needed for the RTT to stop its increase. This happens around 5s for the case $\delta = 500$.
Figure 6 is an indication of the overall performance. This figure shows the evolution of the acknowledged byte sequence number during the transfer. We see well how taking $\delta = \delta_o$ leads to the best performance since it gives a good compromise between delaying the filtering of ACKs 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 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 instantaneous queue length for filtering ACKs is not a good guarantee for good performance. First, the arrivals of ACKs in reality at the slow channel may be different than the theoretical arrival process we described. This is due to many factors as the fluctuation introduced by the exogenous traffic in both directions. Second, the computation of the optimal threshold δ_o in (1) is difficult since it requires the knowledge of RTT and the acknowledging strategy at the receiver (d). Third, setting the filtering threshold to a fixed value leads to an unnecessary increase in the RTT in the steady state of the connection. An aggressive filtering (say $\delta = 1$) needs to be applied once the slow reverse channel is fully utilized. Some mechanism must be implemented in the reverse buffer to absorb the bursts of ACKs when the slow channel is not fully utilized, and to filter ACKs with a small δ otherwise. Using the average queue length as in Random Early Detection (RED) buffers [11] does not work here since the objective is not to control the average queue length as in the case of data packets, but rather to find a signal for a good utilization of the slow channel. 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 will be 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 (before being transmitted over the slow channel) and to compare this rate to the channel 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 measurements. In case we do not know the channel bandwidth (e.g., case of a shared medium), one can measure how frequent ACKs are present in the reverse buffer. This gives an idea on how well the channel is utilized. A full utilization during a certain interval is equivalent to a continuous presence of ACKs in the

buffer. We call this measure *the presence rate* of ACKs. When one of these two measures (rate of ACKs, presence rate of ACKs) indicates that the channel is fully utilized (e.g., the utilization exceeds a certain threshold that we fix to 90% in simulations), we start to apply the classical 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 strategy guarantees a maximum window increase rate during slow start and a minimum RTT in the steady state. One can see it as a dynamic filtering strategy 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. Note that we are always working with the case of a single connection. The case of multiple concurrent connections will be studied later.

3.1 Utilization measurement

We assume that the slow channel bandwidth is known and we use the output rate approach. The time sliding window (TSW) algorithm defined in [8] 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 classical low pass filters is that the decay time of the rate with the TSW algorithm is constant and not a function of 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 of ACKs, `Window` be the time window, `Last` be the time when the last ACK has been seen, `Now` be the current time, and `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,

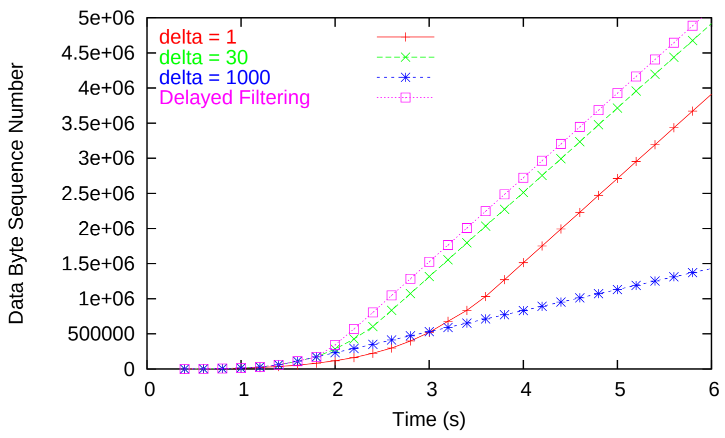

Figure 7: Last acknowledged sequence number vs. time

```
if (length(queue)==0) Volume = Rate*Window;
else Volume = Rate*Window + 1;
Time = Now - Last + Window;
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 as RTT. Figure 7 shows how our algorithm gives as good performance as the case when δ is correctly chosen, i.e., $\delta = \delta_o$. Moreover, it outperforms slightly the case of optimal δ due to the erasure of all the 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 as a function of time. These figures correspond to static filtering with $\delta = \delta_o = 30$ and delayed filtering respectively. Bursts of ACKs are absorbed at the beginning of the transfer before being filtered some time later. The filtering starts approximately at the same moment in both 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. Dropping ACKs from a new connection with a small window can be very harmful since the connection can lose its ACK clock and enter in a long timeout period with a minimum duration of one second in most TCP implementations. We consider a max-min fairness where the slow channel bandwidth needs to be equally distributed among the different

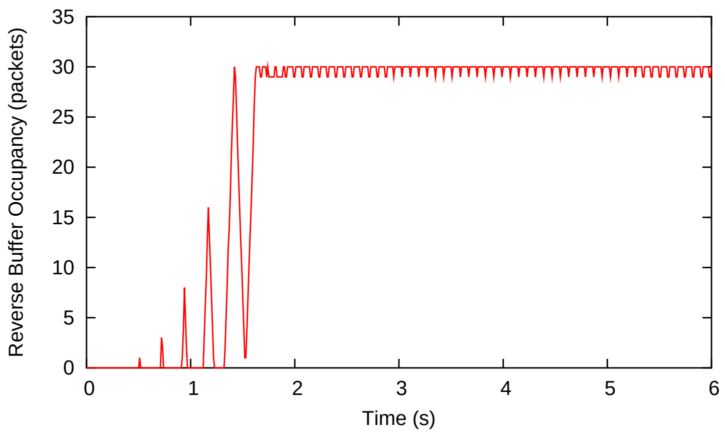

Figure 8: Reverse buffer occupancy for $\delta = 30$

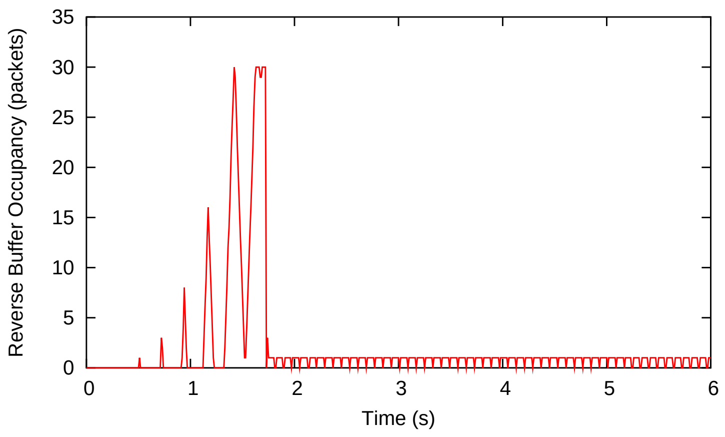

Figure 9: Reverse buffer occupancy for delayed filtering

ACK flows. We should obtain this fairness 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 by applying ACK filtering to a bursty ACK flow rather than finding the appropriate fairness scheme. Other fairness schemes can 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 need here for an ACK dropping policy 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 when the reverse buffer is small and when 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 dropping policy. Consider as example the case of a static filtering strategy with $\delta = 1$, which is the most aggressive strategy. It may result in N ACKs queued in the reverse buffer, since one ACK per-connection at a time is authorized. Even in this most aggressive case, some ACKs will be dropped 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 propose 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 at the reverse buffer, the list of connections is checked for the corresponding entry. If no entry is found, the ACK is considered to belong to a new connection, and a new entry is created with an initial average rate. 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 as that of the arriving ACK 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. The entry corresponding to one connection is freed when its average rate falls below a certain minimum threshold. A TSW algorithm is again used for ACK rate measurement, but note here that for connections to which the departing ACK does not belong we must write,

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

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 really exist. A per-connection state has been proposed in [14] for Internet routers to calculate the rate of a connection and to control the content of its ACK headers. The problem is less important in our case since the number of connections crossing the reverse channel is in general small. As we will see later, we can stop our algorithm beyond a certain number of connections since it converges to static

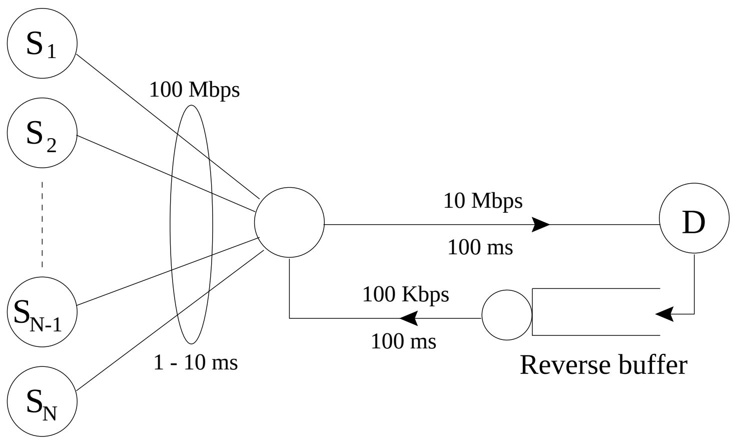

Figure 10: Simulation scenario

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.

Delaying the filtering of ACKs from a connection separately from the other connections while keeping the classical First-In First-Out (FIFO) service does not result in a complete isolation between connections. The accepted burst of ACKs from an unfiltered connection results in an increase in the RTT of the other connections. A Round Robin (RR) scheduling is required for such isolation. The same weight is associated to 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. With a RR scheduler, ACKs from filtered connections no longer need to wait after ACKs from unfiltered connections.

We implement the different filtering schemes we cited above into the `ns` simulator [17] 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 TCP 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 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 the results in Figure 11 where we show the performance as a function of N for five schemes: the no-filtering scheme ($\delta = +\infty$), the classical filtering scheme ($\delta = 1$), the delayed filtering scheme with all the connections grouped into one flow, the per-connection delayed filtering scheme with FIFO and with Round Robin scheduling.

The no-filtering scheme gives the worst performance due to the long queue of ACKs that builds up in the reverse buffer. Classical filtering solves this problem and improves the performance but it is too aggressive so it does not give the best performance especially for small number of connections. For large number of connections, the bandwidth share of a connection becomes

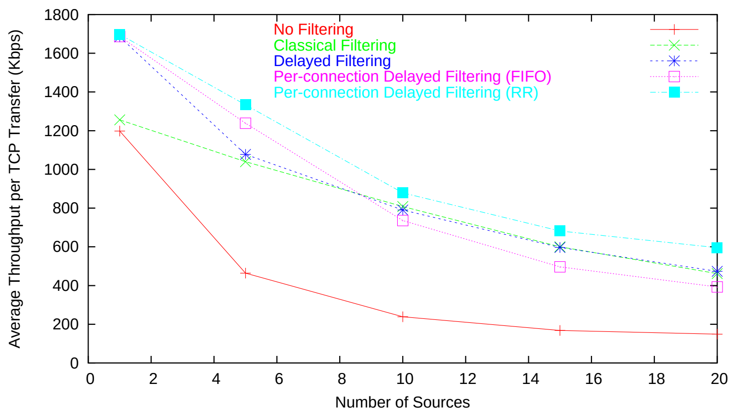

Figure 11: Case of multiple connections and a large buffer

small and classical filtering gives close performance to the best scheme, the per-connection filtering with Round Robin scheduling. Single-flow delayed filtering is no other than classical filtering beyond a small number of connections. With single-flow delayed filtering, ACKs are filtered in the same manner as with classical filtering once the slow channel is fully utilized. The grouping of all the connections into one flow is only used for the purpose of measuring the utilization of the slow channel. 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 ones.

4.2 Case of a small buffer

In the case of large reverse buffer, ACK filtering is sufficient to achieve good performance. An arriving ACK is always able to find a place in the reverse buffer. The question we ask here is, given a certain filtering scheme, 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 only used 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 blocs. The first bloc which is the filtering bloc tries to eliminate the unnecessary information from the flow of ACKs. The filtered flow of ACKs is then sent into the second bloc 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 filtering scheme, and we compare the performance under a small buffer assumption.

For classical filtering ($\delta = 1$), we use the simple drop tail policy. The buffer space is fairly shared between the different connections (one ACK per connection) and we do not 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 [19] use the content of the buffer to get an idea on bandwidth sharing. The same drop tail policy is used in case of single-flow delayed filtering when ACKs are filtered (full utilization of the reverse channel). When ACKs are not

filtered, we use the Longest Queue Drop policy (LQD) described in [19]. The oldest ACK of the connection with the longest queue is dropped. For the per-connection delayed filtering scheme, we profit in the drop procedure from the information available for filtering. The oldest ACK of the connection with the highest rate is dropped. 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 in Figure 12 as a function of the number of sources. 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. 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 Round Robin scheduling gives again the best performance since it guarantees the best isolation between flows. Single-flow delayed filtering gives better performance than classical filtering in case of small number of connections but it converges to classical filtering when N increases. 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 the case of no-filtering. However, as the number of connections exceeds the buffer size, the reverse buffer starts to overflow and new connections are no longer 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 mechanism needs to track.

5 Conclusions

We studied in this paper the gain from 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 a TCP connection, which requires the arrival of the maximum possible amount of ACKs to quickly increase its congestion window. First, we studied a static filtering scheme, which delays the filtering of ACKs until the length of the queue in the reverse buffer reaches a certain fixed value. Second, we presented different delayed filtering schemes that use the rate of ACKs, or the presence rate of ACKs in the reverse buffer, as an indication to start the filtering of ACKs. We found that per-connection ACK filtering with Round Robin scheduling gives the best performance. We also saw that there is no need to delay filtering beyond a certain number of TCP connections, since all schemes converge to the classical filtering strategy studied in the literature.

References

[1] M. Allman et al., " Ongoing TCP Research Related to Satellites", *RFC 2760*, February 2000.

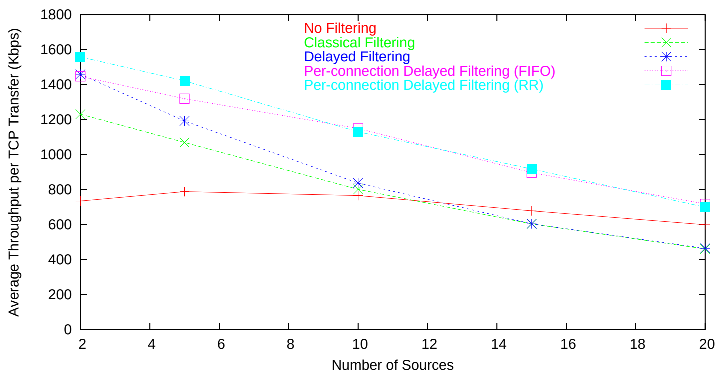

Figure 12: Case of multiple connections and a small buffer

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

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

[4] C. Barakat, "Performance Evaluation of TCP Congestion Control", *Ph.D. thesis*, University of Nice - Sophia Antipolis, France, April 2001.

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

[6] C. Barakat, E. Altman, and W. Dabbous, "On TCP Performance in a Heterogeneous Network : A Survey", *IEEE Communications Magazine*, *IEEE Communications Magazine*, vol. 38, no. 1, pp. 40-46, January 2000.

[7] Neal Cardwell, Stefan Savage, and Tom Anderson, "Modeling TCP Latency", *IEEE INFOCOM*, March 2000.

[8] D. Clark and W. Fang, "Explicit Allocation of Best Effort Packet Delivery Service", *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, pp. 362-373, August 1998.

[9] R. Durst, G. Miller, and E. Travis, "TCP Extensions for Space Communications", *ACM MOBICOM*, November 1996.

[10] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", *ACM Computer Communication Review*, vol. 26, no. 3, pp. 5-21, July 1996.

[11] S. Floyd and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397-413, August 1993.

[12] V. Jacobson, "Congestion avoidance and control", *ACM SIGCOMM*, August 1988.

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

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

[15] 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*, vol. 5, no. 3, pp. 336-350, Jun. 1997.

[16] 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*, April 1997.

[17] The LBNL Network Simulator, *ns-2*, http://www.isi.edu/nsnam/ns/

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

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