ELSEVIER

Computer Communications 24 (2001) 1525–1539

computer communications

www.elsevier.com/locate/comcom

Balanced packet discard for improving TCP performance in ATM networks

Reuven Cohen*, Yaniv Hamo

Computer Science Department, Technion — Israel Institute of Technology, 32000 Haifa, Israel

Received 14 March 2000; revised 10 January 2001; accepted 11 January 2001

Abstract

TCP suffers from low performance over asynchronous transfer mode (ATM) networks. This is mainly because during phases of congestion, ATM drops cells without taking into account the effect this has on the upper layer protocols. Two main algorithms, called partial packet discard (PPD) and early packet discard (EPD), were proposed in the past for improving TCP performance. However, they address one aspect of the problem that has only small effect on the final performance. In this paper we propose an enhanced method for packet discard, called balanced packet discard (BPD) that improves TCP performance dramatically on congested networks and guarantees fairness among multiple connections. We will show that BPD increases TCP throughput by more than 25% compared to EPD/PPD.

Keywords: TCP performance; Early packet discard; ATM networks

1. Introduction

With the increased popularity of the Internet, TCP has become the most commonly used transport protocol. It is unlikely that this will change in the near future, because of the enormous amount of software that was written for it. For this reason, it is essential that asynchronous transfer mode (ATM), which is likely to play a major role as a layer-2 protocol in the Internet, will provide good support for TCP. However, the interface between TCP and ATM introduces performance degradation, mainly due to the following reasons:

- *The timeouts problem*. When a congested ATM switch drops a single cell from a TCP segment, the entire segment is corrupted. This results in an upper layer packet loss rate that is much larger than the ATM cell loss rate. Consequently TCP often loses several segments from the same window, and can usually recover from that only after a timeout that significantly reduces its performance.
- *The corrupted packets problem*. The rest of the cells belonging to a corrupted packet continue to travel over the network, consuming bandwidth and buffer space unnecessarily, because they will be discarded at the destination anyway. Furthermore, the transmission of these useless cells in times of congestion may cause other packets to lose their cells. The loss of cells belonging to other packets, due to resource consumption by cells of a corrupted packet, reduces the network performance significantly.

Two mechanisms have been proposed in the literature to address the problem of corrupted packets: *Partial Packet Discard* [1] (PPD) and *Early Packet Discard* [2] (EPD). In PPD, when a cell has been discarded, the rest of the cells belonging to the same packet are intentionally discarded by the same ATM switch. Still, some bandwidth and network resources are wasted while delivering the leading cells of the packet, which were stored in the buffer prior to the loss. EPD is intended to save this bandwidth, by identifying in advance every packet that is likely to encounter a loss, and discarding all the cells of such a packet. These mechanisms do improve performance, but they suffer from low fairness [3,4], and their contribution is negligible when the network is not heavily congested. These mechanisms improve the performance of *every* packet-based transport protocol that runs over ATM, but fail to fulfill the specific requirements of the most widely used protocol — TCP.

In this work we present a new algorithm called Balanced Packet Discard (BPD), designed specifically to improve the

* Corresponding author.
E-mail address: rcohen@cs.technion.ac.il (R. Cohen).

PII: S0140-3664(01)00315-2

performance of TCP over ATM by minimizing the timeouts problem. BPD takes into account TCP considerations when deciding whether or not to discard a packet. It therefore provides a much better support for TCP connections than EPD or PPD. In some cases the gain in throughput is 100% higher than when EPD and PPD are employed. We study the behavior of TCP over ATM networks, in two representative topologies. The study is conducted using the NS simulator from LBL [5] with ATM extensions.

The rest of this paper is organized as follows. Section 2 discusses the timeouts problem, introduces relevant aspects of TCP and provides a detailed analysis of the behavior of fast-retransmit in the presence of multiple segment losses. Section 3 discusses the corrupted packets problem, presents the existing schemes, PPD and EPD. It introduces the simulation environment, describes the tested topologies and provides simulation results for EPD and PPD. Section 4 presents the new algorithm, BPD, in details. Section 5 presents simulation results for BPD. Section 6 gives a brief overview of the new TCP-SACK extension and discusses the effects of combining BPD with TCP-SACK, along with simulation results. Finally, Section 7 concludes the paper.

2. The timeouts problem

2.1. TCP background

Current TCP implementations [6–8] contain a number of algorithms aimed at controlling network congestion, and recovering from segment losses. These algorithms include slow-start, congestion-avoidance, fast-retransmit and fast-recovery. Together they define the congestion window, *cwnd*, as an estimation of the maximum number of segments that can be sent back-to-back without overloading the network. The TCP sender never sends more than the minimum of *cwnd* and the receiver advertised window.

The TCP sender operates in one of two modes: *slow-start* or *congestion-avoidance*. The main difference between these two modes is the rate of increasing *cwnd*. The sender determines its mode based on the value of *cwnd* and a threshold value called *ssthresh*. As long as *cwnd* is smaller than *ssthresh*, the sender operates in slow-start mode. When *ssthresh* is reached, the sender switches to congestion-avoidance. During slow-start, the sender starts with a congestion window of one segment, and increments it by one segment with every acknowledgment received. Assuming an ack is sent for every received data segment, this results in doubling *cwnd* every round trip time (RTT). In contrast, in congestion-avoidance mode, the sender increases the value of *cwnd* by 1/*cwnd* for every acknowledgment received. This is approximately equivalent to a linear increase of one segment every RTT.

When a segment is lost, subsequent segments are received out of order. An out of order segment triggers an acknowledgment carrying the same sequence number as a previous acknowledgment. Such an ack is therefore referred to as a *duplicate ack*. The sender attributes the first and second duplicate acks to a possible out of order routing. However, when the third duplicate ack is received, the sender assumes a segment has been lost. It then invokes the fast-retransmit procedure which sets *ssthresh* to *cwnd*/2, shrinks *cwnd* to *ssthresh*, and immediately retransmits the missing segment. After transmitting the missing segment, fast-recovery takes over. The value of *cwnd* is set to *ssthresh* + 3, and is increased by 1 segment for each additional duplicate ack received. An ack that acknowledges the retransmitted segment sets *cwnd* to *ssthresh*, thus putting the sender in congestion avoidance mode, and defining the end of fast-recovery. Fig. 1 illustrates the fast-retransmit and fast recovery mechanisms.

The scenario considered in the previous paragraph is the fastest way to recover from a segment loss. However, fast-retransmit cannot be invoked when *cwnd* is smaller than 5 segments, or when the TCP connection suffers from a loss of several segments during the same sending window. In these cases there are not enough segments in the pipe to trigger three duplicate acks for every lost segment. Fig. 2 illustrates one possible scenario, in which two segments are lost from a window of 8 segments. There are enough duplicate acks to trigger fast-retransmit for the first segment. However, because of the window being small, the sender cannot send enough new segments in order to trigger 3 duplicate acks for the second lost segment. This prevents TCP from recovering using fast-retransmit.

In order to detect a loss even in this case, the sender maintains a retransmission timer. This timer is restarted when TCP sends a data segment. When the timer expires,

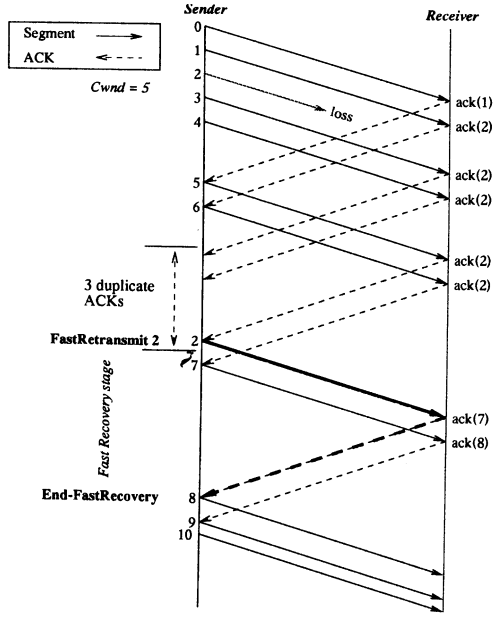

Fig. 1. An example of fast-retransmit.

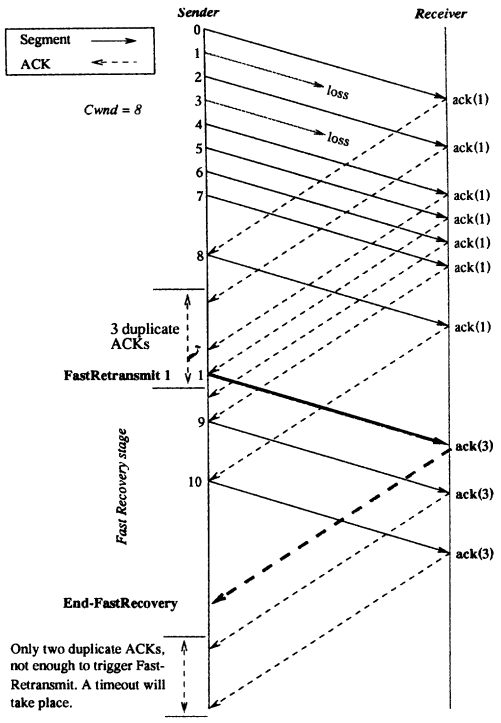

Fig. 2. Fast-retransmit cannot be invoked for the loss of the second segment.

the oldest segment for which an ack has not been received is retransmitted. In addition the sender shrinks *cwnd* to 1, enters slow-start and sets *ssthersh* to half the value *cwnd* had when the loss was detected. Timeouts have a significant negative impact on the throughput, both because the sender waits idle for the timer to expire, and because it operates with a non-optimal window for a few round trips after the timeout takes place.

The retransmission timeout value, called RTO, is computed dynamically, based on round trip measurements the sender performs throughout its operation. Due to conservative estimation of the round trip time, and the usage of a coarse granularity timer, typically 500 ms, TCP often ends up with timeout values that are much longer than the actual RTT. Fig. 3 shows the average value of RTO versus the actual RTT for several hundreds TCP connections we have studied by simulation. Since typical transfer times range from tens of milliseconds to several seconds, one timeout is sufficient to significantly degrade the performance. Later we will see that when more than three segments are lost from the same window, fast-retransmit cannot be triggered for the fourth loss, and a timeout takes place. With small window sizes, even less than three segment losses can force timeout. Therefore, TCP performance can be significantly improved by minimizing the probability for multiple losses from the same window.

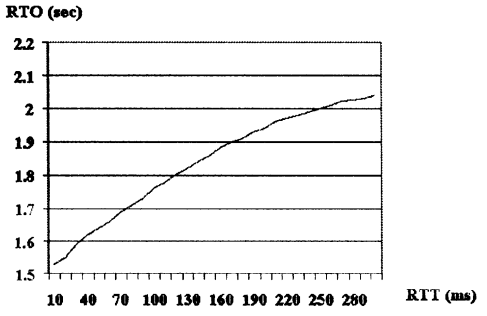

Fig. 3. RTO as a function of the actual RTT.

2.2. Analysis of fast-retransmit

In the following we analyze the cases where TCP needs timeouts in order to recover from losses. Throughout the discussion, we assume that the sender uses the maximum window size allowed by *cwnd*. Consider the case of N lost segments, from a window of size W. Following the assumption that TCP works at maximum speed, W is also the maximum window size. Let the segments in the window be numbered 0, 1,…,$(W-1)$. Let the group of *lost* segments be $G_L = \{n_1, n_2, \ldots, n_N\}$, where $|G_L| = N$ and $n_1, n_2, \ldots, n_N$ are the indices of the lost segments.

Segments 0,1,…,$(n_1 - 1)$ arrive at the receiver, and generate n_1 normal (i.e. not duplicate) acks. These normal acks advance the window to $[n_1,\ldots,n_1 + W - 1]$. The last segment sent is $n_1 + W - 1$. Now $W - N$ duplicate acks for segment n_1 arrive. Because three duplicate acks are needed to fast-retransmit segment n_1, we require that

$$W - N \geq 3 \tag{1}$$

will hold in order to avoid a timeout.

After the sender receives three duplicate acks on n_1, it retransmits the lost segment, shrinks the congestion window size to $W/2 + 3$, and enters the fast-recovery phase. During this phase, $W - N - 3$ duplicate acks on n_1 arrive. Each one increases *cwnd* by one segment. At this stage, new segments may be transmitted because the increased value of *cwnd* allows it. We now compute the number of new segments that can be transmitted during this phase. Upon entering fast-recovery, the window contains segments $[n_1, \ldots, n_1 + W/2 + 2]$, and the last sent segment is $n_1 + W - 1$. Therefore $n_1 + W/2 + 2 + W - N - 3 - (n_1 + W - 1) = W/2 - N$ new segments will be sent. After the receipt of a new ack, that is ack(n_2), fast-recovery ends. The value of *cwnd* is set to $W/2$, and the window embraces segments $[n_2, \ldots, n_2 + W/2 - 1]$. New segments can theoritically be transmitted now. The last segment sent so far is $n_1 + W - 1 + W/2 - N$. New segments will be transmitted after exiting fast-recovery only if $n_2 + W/2 - 1 > n_1 + W - 1 + W/2 - N$. That is, only if $n_2 - n_1 > W - N$, which is impossible for $N > 2$ ($n_2 - n_1$ cannot be $W - N + 1$ because when $N > 2$ we also have n_3 inside the window and after n_2). Fast-retransmit is therefore ended with $W/2 - N$ new sent segments. These new segments, and only them, will trigger

duplicate acks for n_2. Therefore,

$$\frac{W}{2} - N \geq 3 \tag{2}$$

must hold in order to fast-retransmit segment n_2. This implies that if only two segments are lost within a window, the window must be greater than or equal to 10 in order to avoid any timeout.

So far, the position of the lost segments within the window does not play any role in determining whether or not a fast-retransmit is invoked. After three duplicate acks on n_2, the lost segment is retransmitted, the congestion window size shrinks to $W/4 + 3$, and the fast-recovery phase is entered. During this phase, $W/2 - N - 3$ duplicate acks on n_2 arrive. Each one of them increases *cwnd* by one segment. We next find the number of new segments transmitted during this phase. Upon entering fast-recovery, the window embraces segments $[n_2, \ldots, n_2 + W/4 + 2]$, and the last segment sent is $3W/2 + n_1 - N - 1$. Hence $n_2 + W/4 + 2 + W/2 - N - 3 - (3W/2 + n_1 - N - 1) = n_2 - n_1 - 3W/4$ new segments are sent. After the receipt of a new ack, that is, ack(n_3), fast-recovery is ended. The value of *cwnd* is set to $W/4$ and the window contains segments $[n_3, \ldots, n_3 + W/4 - 1]$. The last segment sent so far is $3W/4 + n_2 - N - 1$. New segments will be transmitted after exiting fast-recovery only if $n_3 + W/4 - 1 > 3W/4 + n_2 - N - 1$. That is, only if $n_3 - n_2 > W/2 - N$, which is impossible under the constraint $n_2 - n_1 - 3W/4 > 0$ (positive number of sent segments). The fast-retransmit state is therefore ended with $n_2 - n_1 - 3W/4$ new sent segments. These new segments, and only them, trigger duplicate acks for n_3. The conclusion is that

$$n_2 - n_1 \geq \frac{3W}{4} + 3 \tag{3}$$

must hold in order to fast-retransmit segment n_3. This implies that the first loss and the second loss must be spaced far enough from each other, in order for the third lost segment to be fast-retransmitted. Fig. 4 demonstrates the above equations by an example of three losses from a window of 24 segments. Similar analysis for the case of four losses yields the condition

$$n_3 - n_1 \geq \frac{11W}{8} - N + 3 \tag{4}$$

which can never hold with the previous constraints ($n_2 - n_1 \geq 3W/4 + 3$). This implies that if TCP loses four segments from the same window, a timeout is inevitable, regardless of the position of the lost segments within the window, or the size of the window.

To conclude, TCP can recover using fast-retransmit with high probability only if two segments or less are lost within the same window. With small window sizes, even such a loss might result in a timeout. However, for long transfer and reasonable loss rates, *cwnd* is most of the time not so small. Thus, keeping the number of losses from the same window below three is sufficient in order to minimize the probability for a timeout, with the understanding that three losses are also acceptable, if the first two are spaced far enough. A conservative approach would be to minimize the probability that two segments are lost from a consecutive group of $3W/4 + 3$ segments. In this case TCP loses one segment every $3W/4 + 3$ segments on the average, which is a loss of at most two segments per window, a loss TCP can handle without significant performance degradation. This value also ensures that if something happens and two close segments are lost, no timeout will occur, because the previous two segments are spaced for enough. Minimizing the probability for two losses within a consecutive group of $W/2$ can also ensure no timeouts, if it is strictly kept. Our proposed algorithm, BPD, provides a way to space the losses far enough from each other.

3. The corrupted packets problem

Since TCP is a packet based protocol while ATM is a cell based network, TCP/IP segments must be fragmented into several ATM cells. For example, a typical TCP segment is of 536 bytes. With the additional TCP/IP headers, we get an IP datagram of 576 bytes. This datagram is fragmented by ATM to 12 AAL5 ATM cells. A special bit in the ATM header of an AAL5 cell, called ATM-layer-user-to-user (AUU), is set to 1 for the last cell of the packet. Hence the ATM layer can distinguish one packet from another.

At times of congestion, the buffers at the ATM switch are overflown. When the buffer is full, the switch must drop any incoming cell. Because the switch may drop only part of the cells of a packet, we might see cells of corrupted packets traveling across the network. When such cells are received by the destination host, they are discarded by the segmentation and reassembly (SAR) unit at the end of the unsuccessful reassembly process. Therefore, the network resources consumed by these cells are wasted, and other packets that could have used these resources might also be dropped because of congestion.

An improved cell discarding scheme, called PPD was proposed in [1]. According to this scheme, when a switch drops one cell, it continues to drop all subsequent cells belonging to the same packet, except the last one with the AUU bit set to 1. PPD requires the switch to keep state information for each crossing VC. This information indicates which VC uses AAL5, and whether to drop or forward the next cell of such a VC. This scheme practically chops the tail of a damaged packet, and in this way saves the network resources otherwise needed for transferring this tail. However, the head of the corrupted packet is still transmitted.

EPD [2], aims at saving the network resources consumed by the head cells of corrupted packets. EPD introduces a threshold on the buffer of the ATM switch, called *EPD-threshold*. The switch is said to be in an overflow danger

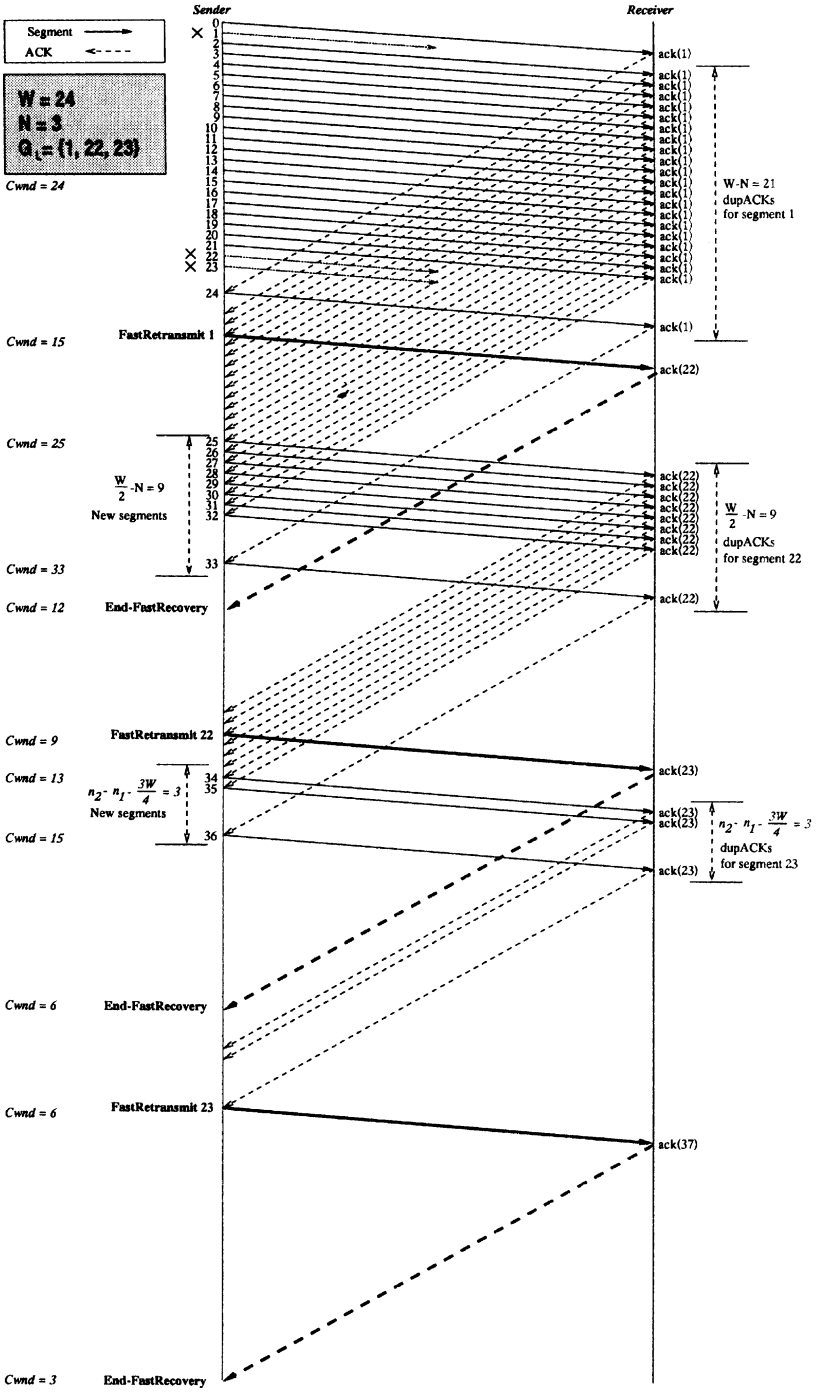

Fig. 4. Fast-retransmit in the presence of multiple losses.

whenever the threshold is exceeded. EPD-threshold can be defined by an absolute number of cells or by percentage of the total buffer capacity. In [9] it is suggested to define the threshold by examining the slope of the buffer occupancy curve. When the buffer occupancy reaches the EPD-threshold, the switch discards the first cell and all subsequent cells of every received packet. In this way the switch imitates a packet-based switch, which drops full packets. Since EPD is usually used in conjunction with PPD, for the rest of this paper the term EPD means EPD with PPD. In terms of implementation, EPD only drops the first cell of the packet, and then PPD drops all subsequent cells, except the last one. EPD requires the switch to hold an EPD-threshold for every buffer, and to efficiently monitor the occupied buffer size.

First, we examine the performance and fairness of TCP over ATM assuming the two existing discard algorithms

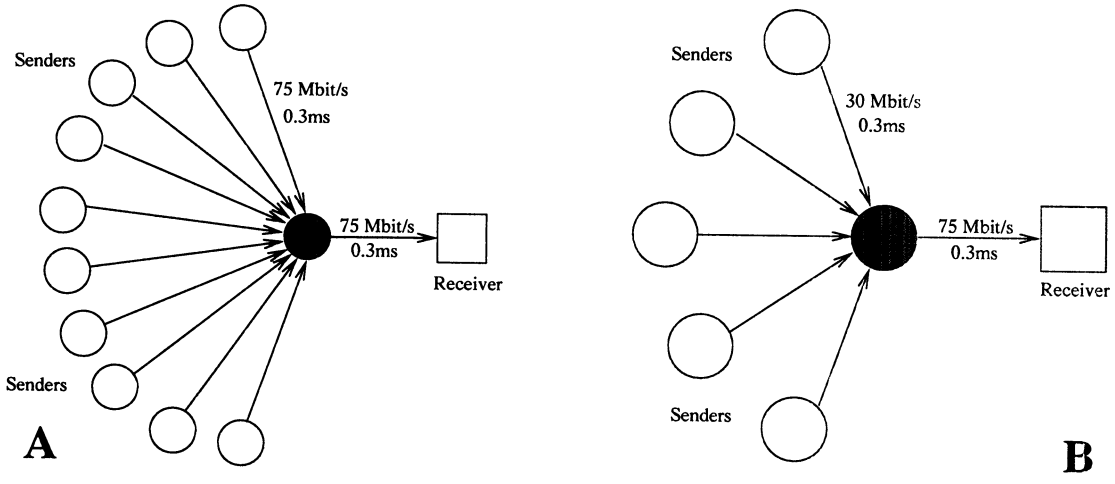

Fig. 5. The considered network topologies. topology A (left) and topology B (right).

PPD and EPD. We show that these algorithms eliminate the problem of corrupted packets, but fail to prevent timeouts and thus fail to significantly improve the performance of TCP.

All simulations were performed using a modified version of NS 1.4 from LBL [5]. The TCP flavor used is Reno. TCP maximum segment size is set to 536 Bytes, which yields IP datagram of 576 Bytes (20 Bytes of TCP header + 20 Bytes of IP header). Maximum window size is 64 KB. We have considered two network topologies for our simulations. In both topologies there is one bottleneck switch, colored gray. This congested switch is assumed to have a buffer of 1200 cells, and it implements some of the discussed packet discard mechanisms. The two simulated network topologies, called topology A and topology B, are shown in Fig. 5. Topology A presents 10 incoming links, each of 75 Mbit/s with a propagation delay of 0.3 ms, and one outgoing link of 75 Mbit/s with the same delay. This topology is practically the same as considered in [2]. The gray switch in this example is much more congested than the one in topology B. However, it is enough for this system to have one active TCP connection, namely a connection that does not wait for a timeout, in order to reach maximum throughput. This means that we can lose many cells, have 90% of the connections inactive, and still achieve maximum throughput. Throughput degradation in this case is caused mainly by cells of corrupted packets, that are nevertheless still transmitted, rather than from TCP idle times. Hence, in this case, low throughput can be avoided by means of EPD.

In topology B the congested switch has five incoming links, each of 30 Mbit/s with a propagation delay of 0.3 ms, and one outgoing link of 75 Mbit/s with the same delay. It is clear that the gray switch might be congested, because it might receive cells at a rate of 150 Mbit/s, while it can transmit cells at a rate of only 75 Mbit/s. Assuming that a TCP connection runs on each of the five VCs, it is enough that three connections wait for a timeout to degrade the total throughput. In such case only two connections remain active, and they can generate together only 60 Mbit/s.

We start with topology A. Similarly to [2], we define the *effective throughput* or *goodput* as the throughput that is "good" in terms of the application layer. That is, the effective throughput does not include the various headers (in our case, ATM, TCP and IP headers) nor the cells that are part of retransmitted or incomplete packets. Fig. 6 plots the effective throughput, or *goodput*, as a function of the EPD-threshold. The threshold is given as a percentage of the total buffer size. For instance, 0.4 means that the switch invokes EPD when the buffer is more than 40% full.

The "Plain ATM" curve represents the case where no special cell discard policy is enforceable. Hence, for this curve and for the PPD curve, the threshold plays no role. We see that PPD has better performance than Plain ATM, and that EPD only slightly improves this performance, with the increase of the EPD-threshold (recall that by EPD we refer to the implementation of EPD and PPD). Increasing

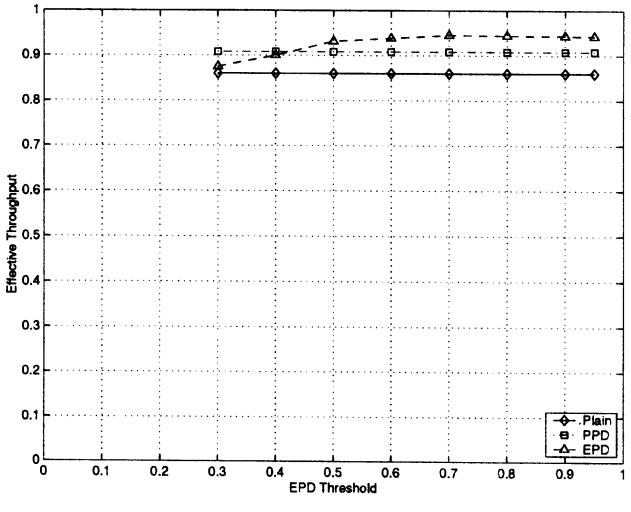

Fig. 6. Plain ATM, PPD and EPD in topology A.

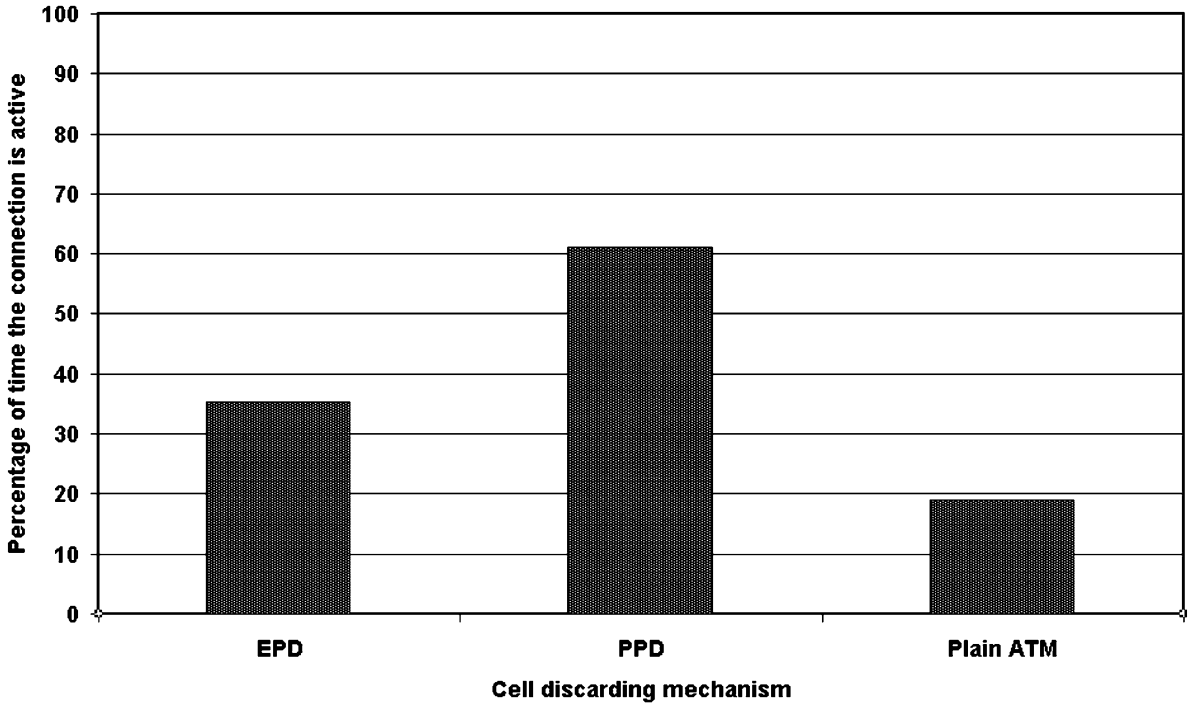

Fig. 7. Proportion of the time the connections are active in topology A.

the EPD-threshold above 70% in our case has negative effect because the switch has not enough excess space for storing the body of every packet for which the first cell is not discarded.

In overall, this configuration reaches fairly high throughout. The results are consistent with those presented in [2]. The reason for the high throughput is that one active connection is enough to keep the congested link highly utilized. In other words, if many TCP connections are not active because their senders are waiting for their timeout to expire, the link is still being utilized by the other connections, and the total throughput is not affected. Note, however, that for smaller buffer sizes [2] shows that PPD and EPD significantly improve the effective throughput.

For each discarding scheme, Fig. 7 depicts the total percent of time the TCP connections are active, i.e. transmitting data and not waiting for a timeout. Because of the early packet discard performed by EPD, it sometimes drops a packet when PPD would not have, thus causing more timeouts at the sender. This is the reason why under PPD the connections are more active than under EPD. However, in this topology, there is almost no correlation between the percentage of time the connections are active, and the overall throughput, and indeed, EPD reaches higher throughput than PPD even though its TCP connections are less active.

TCP timeouts have a minor influence on the overall throughput of the system in topology A because there are many competitors over the congested link. Hence, if some connections are idle, awaiting timeouts, other connections use their "bandwidth share". In order to examine how the different discard mechanisms really perform, we must test them in scenarios where the overall throughput is affected by timeouts. Such cases are characterized by an outgoing link whose bandwidth can be fully utilized only if most of the connections are active, as in topology B. In this topology it is enough to have three idle connections in order to reduce the maximum possible throughput to $60/75 = 0.8$. Therefore, the conventional EPD/PPD schemes cannot significantly improve the performance. Fig. 8 depicts the measured effective throughput as a function of the EPD-threshold for topology B. The EPD-threshold is given again as percentage of the total buffer size. The curves are

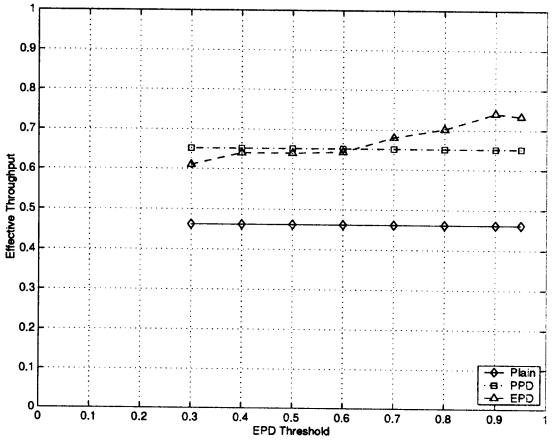

Fig. 8. Plain ATM, PPD and EPD in topology B.

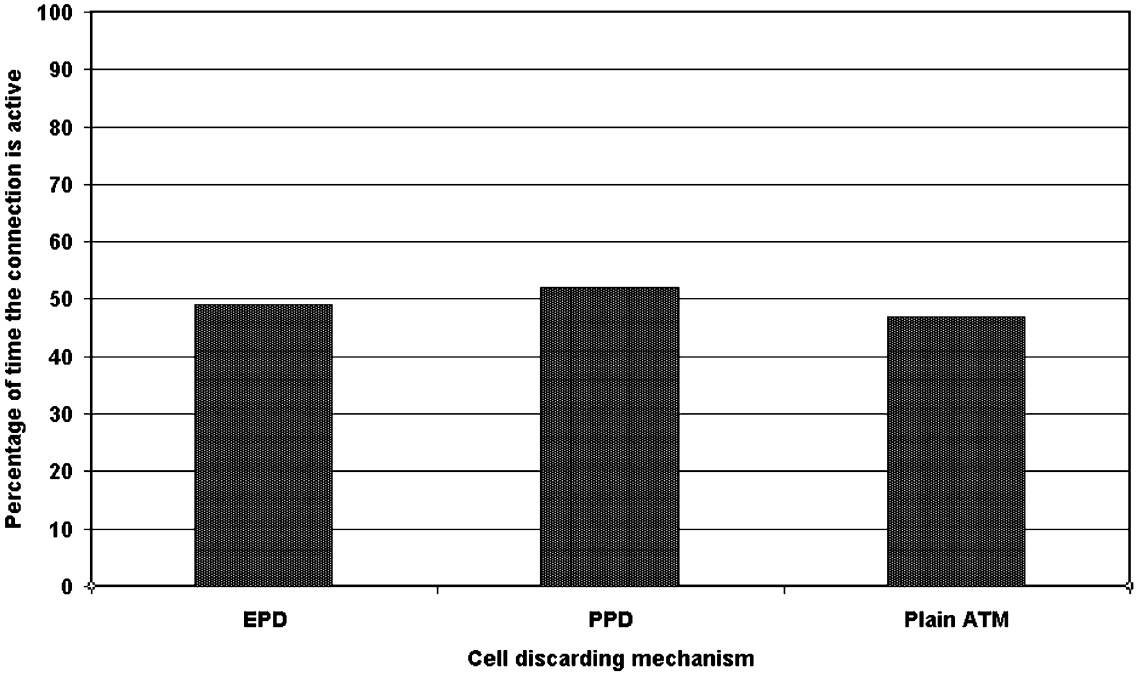

Fig. 9. Proportion of the time of connections are active in topology B.

similar to those presented for topology A, but they are much lower. EPD is again the better scheme, but here it achieves a maximum throughput of only 0.75. Fig. 9 presents the percentage of the time during which the connections are active for each discarding scheme. We see that with EPD the connections are idle for 49% of time.

TCP timeouts are the main cause for low throughput in configurations like topology B. We now present a discard scheme that significantly decreases the number of TCP timeouts.

4. Balanced packet discard

As explained in Section 2 and shown in the simulations results for PPD and EPD, TCP timeouts cause significant throughput degradation. When a segment is dropped from a TCP connection, the sender does not get an ack for it, so it cannot advance its sending window. After sending all the data in the sending window, and without being able to advance the window, the sender remains idle until the lost segment timer expires. Then, the sender retransmits the first segment in the window for which an ack was not received, and continues with a window of one segment in slow-start mode. The sender may recover from a segment loss much faster using the fast-retransmit mechanism, where the third duplicate ack triggers retransmission. However, as explained in Section 2.2, when the sender window is smaller than 5 segments or when multiple segments from the same window are lost, there are often not enough duplicate acks to trigger fast-retransmit.

In order to minimize the number of timeouts, BPD aims at preventing multiple segment losses from the same window. In this way, it increases the probability of loss recovery using fast-retransmit. BPD can be viewed as an extension of EPD, the EPD-threshold is called *Lower-Threshold (LT)* in BPD terms, and an additional, higher, threshold is added, called *Upper-Threshold (UT)*. When a switch drops a cell from VC *i*, because the buffer occupancy exceeds the lower-threshold or the buffer overflows, it drops all subsequent cells of the same packet (PPD), and marks this VC as "damaged". A damaged VC is a VC that has "recently" lost a packet. This VC is granted a higher priority over the non-damaged VCs in the sense that it is subject to the upper-threshold for some *recovery period*. The flow chart of the algorithm is presented in Fig. 10. To see how this scheme significantly reduces the probability of multiple segment losses from the *same* window, consider a case of an EPD switch with a 50 KB buffer and an EPD-threshold of 50%. This buffer has an "excess buffer capacity" [2] of 25 KB. This means that as long as the switch is not overloaded, these 25 KB are not in use. However, when the buffer is congested, i.e. contains more than 25 KB awaiting for transmission, and some VC has two incoming packets, EPD would drop both packets. In contrast, with BPD the second packet of the VC has high probability not to be discarded, because it will be subject to UT which is higher than 50%. The immediate effect of BPD is that a damaged connection is given the chance to recover from the first loss, without immediate additional packet losses. The recovery period is the period of time during which damaged VCs enjoy a higher threshold than the non-damaged VCs. In our study,

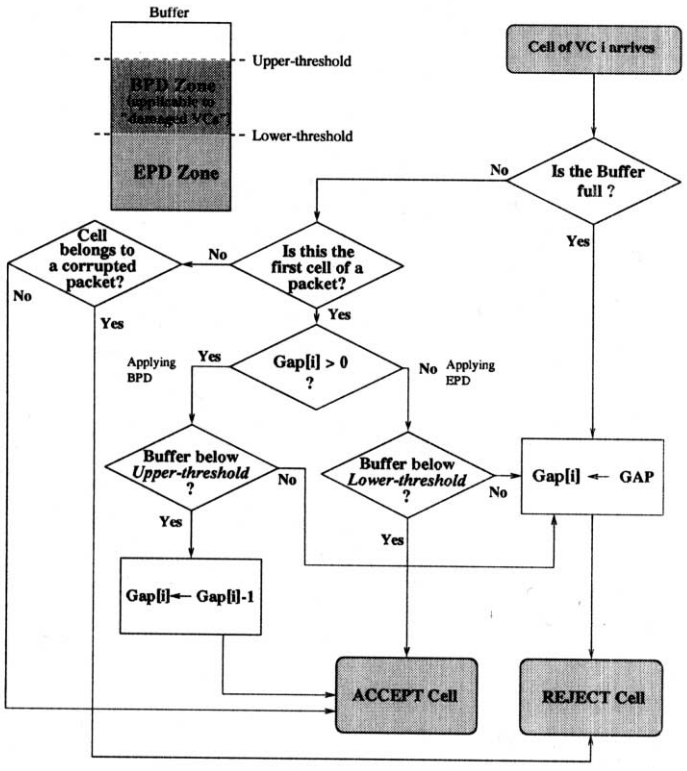

Fig. 10. The BPD algorithm. GAP is the desired recovery period in packet units. Gap[i] holds the number of packets left until the recovery period is over.

we represent this period in packet units. A recovery period of 5 packets means that after a VC loses a packet, it is subject to UT for the subsequent 5 packets of this VC. From implementation point of view, the switch initializes a counter per VC (called Gap[i] for VC *i*) at the beginning of the recovery period, and decrements it by one for every end of packet cell. When this counter reaches zero, the VC is considered regular again.

Let B be the capacity in cells of the space between LT and UT (see Fig. 11). This space is mainly available for the damaged VCs, for recovery purpose.

Let N be the number of VCs, G be the recovery period (in packets), and M be the number of cells in each packet. In the worst case, all the VCs may lose a packet at the same time, so all of them may enter the recovery period and need to be protected by BPD from additional losses. In this case, $NGM < B$ must hold. This imposes an upper bound on the length of the recovery period, during which a damaged VC is protected. Statistical considerations can be taken into account in order to increase the recovery period beyond B/NM. However, there is no practical need to make it longer than the sending window. In reality, even a shorter recovery period significantly decreases the probability for a timeout.

5. Simulation results for BPD

Fig. 12 plots the effective throughput as a function of the

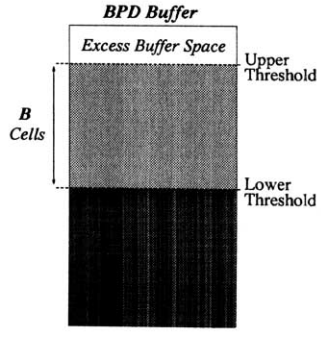

Fig. 11. A diagram of the space distribution in a buffer of a BPD switch.

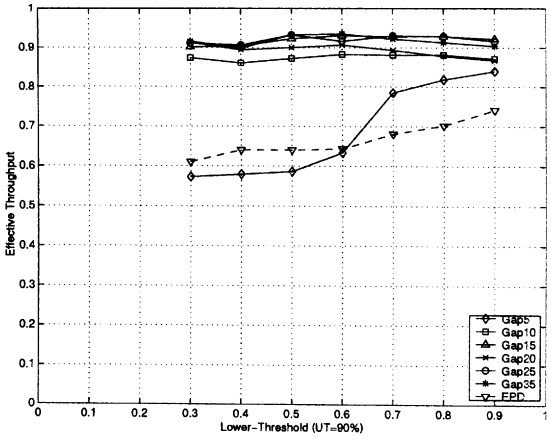

Fig. 12. BPD in topology B.

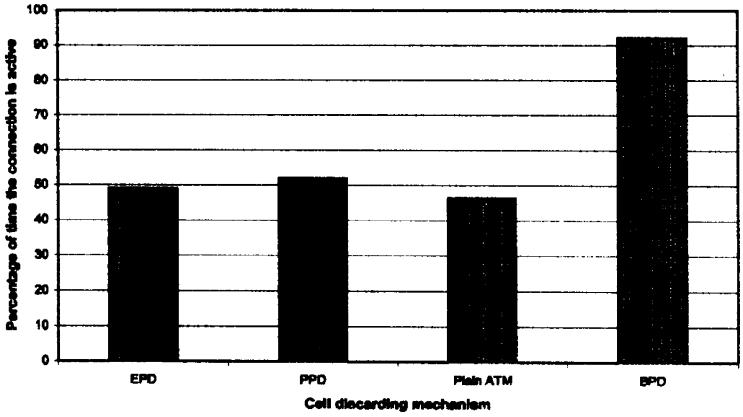

Fig. 13. Proportion of the time the connections are active in topology B with BPD (Gap = 25).

LT as achieved by BPD in topology B. A smaller LT gives a larger excess buffer capacity, and the ability to lengthen the recovery period. There are six graphs, for several lengths of the recovery period *Gap*: 5, 10, 15, 20, 25 and 35 packets. Buffer size is 1200 cells (100 packets).

There is a general increase in performance with the increase of LT. This is because a larger LT provides more usable buffer space. However, if we take a look at some of the larger *Gap* values, 35 for instance, we see that from a certain point the performance degrades as the LT increase. This is because a larger LT provide *less* excess buffer capacity. The small excess buffer capacity is not enough to hold 35 segments of each VC that has lost one packet, so the buffer fills up until UT is reached, and the switch starts discarding packets indiscriminately.

In general, BPD reaches high throughput in this configuration, and is significantly better than EPD. EPD reaches a maximum throughput of 0.741, while BPD reached 0.932. This is an improvement of more than 25%. The reason for this improvement can be easily seen from Fig. 13: with BPD the connections are active 92% of the time, while with EPD they are active only 49% of the time. This indicates that BPD eliminates most of the timeouts.

Recall that in topology A timeouts almost never affect the throughput. Hence, for this topology BPD introduces only a minor improvement. Fig. 14 plots the effective throughput of the network in topology A and BPD is used, as a function of LT. Although the results are very good, there is no significant improvement over EPD.

In topology B when BPD is being applied and the system stablizes, each TCP connection loses one segment every *Gap* segments. Therefore, *Gap* practically determines the maximum number of segments lost from the same window. In a first glance it seems impossible to have such a guarantee without an additional buffer. However, TCP is an adaptive protocol that "senses" the network and adjusts its sending rate accordingly. When TCP loses a segment, it transmits the next segments at a lower rate. BPD will help *Gap* segments to be queued, allowing the congestion window of TCP to be opened, and then if the buffer is full, another segment is lost, causing *cwnd* to decrease and so on. After the system stablizes, TCP loses segments periodically, but almost never require a timeout in order to recover. Every loss is recovered using fast-retransmit and the rate of the connection is reduced by half.

Fig. 15 demonstrates the above discussion. It plots *cwnd* of a TCP connection over an ATM network with BPD, as a function of the time. As can be seen, at the beginning the connection has no knowledge of the network capacity, and its congestion window expands exponentially without proportion to the network capacity. This results in multiple losses that in most of the cases, cause a timeout. In the example shown in Fig. 15, a timeout does not take place, but nevertheless segments are lost, and fast-retransmit is triggered. After it recovers, TCP sends new segments at a lower rate, and after the stabilization point it loses one segment every 25–26 segments which is the *Gap* that was used (this can be clearly seen in higher resolution, in Fig. 16).

After a timeout *cwnd* decreases to one segment, so we can determine the number of timeouts by counting the points at the bottom of the graph. In the case of BPD, we conclude that no timeout has occurred, because *cwnd* never reaches 1.

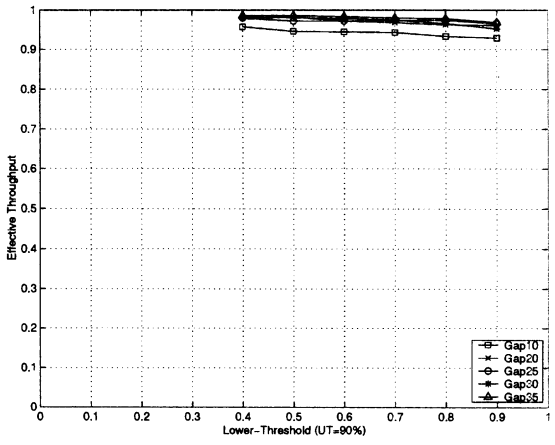

Fig. 14. BPD in topology A.

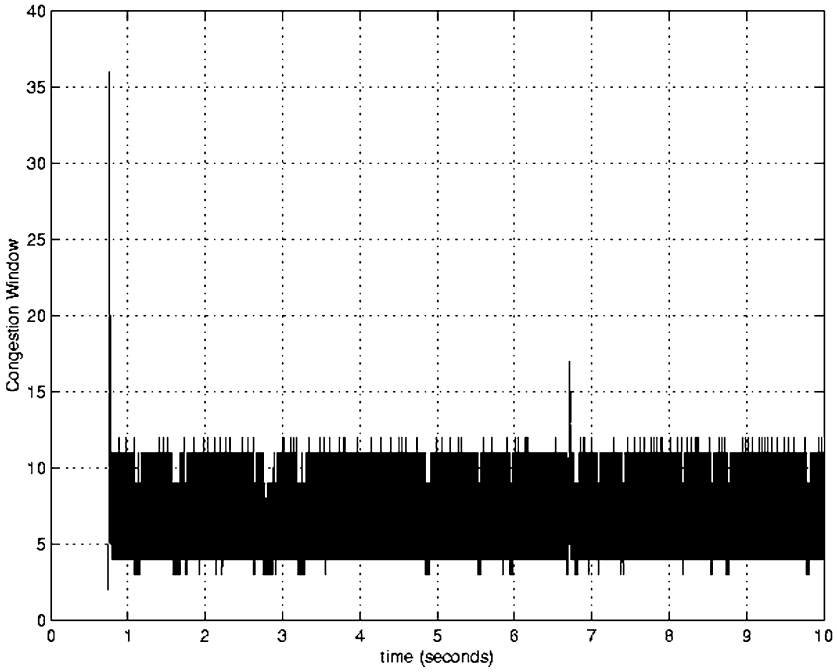

Fig. 15. Congestion Window with BPD over a long period of time. LT = 40%, UT = 90%, Gap = 25.

With EPD (Fig. 17) we count 18 times where *cwnd* reaches 1, meaning 18 timeouts during a period of 10 s. Although *cwnd* reaches lower peeks when applying BPD than when applying EPD, it maintains *cwnd* in a reasonable size for a lengthy period of time. With EPD *cwnd* reaches higher peeks, but suffers from timeouts, so the average *cwnd* size is significantly lower than with BPD.

A key parameter of BPD is the excess buffer space, namely the buffer space above LT. This space is normally not used when BPD is inactive. When a new cell arrives, and the buffer is above LT, this cell is accepted only if it is a part of a packet whose leading cells have already been accepted. Hence, each connection may usually have no more than one packet in the excess buffer space. When using LT of 50%, like in Ref. [2], there is still much space left. BPD makes use of that space to hold packets of VCs in their recovery period. Fig. 18 depicts the amount of used buffer space during the simulation time with an EPD-threshold of 50% in the switch that implements EPD, and an LT of 50% in the switch that implements BPD. We can see that EPD keeps the buffer constantly below the threshold, with only minor breaks to the excess buffer space. These breaks are tails of packets whose leading cells were accepted before the buffer was congested. BPD, on the other hand, uses this space more often and more intensively. This is because VCs in their recovery period consume buffer space of several packets.

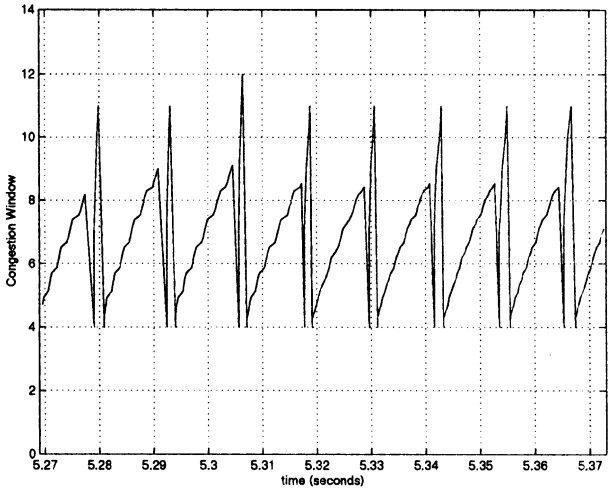

Fig. 16. Congestion Window with BPD over a short period of time. LT = 40%, UT = 90%, Gap = 25.

6. BPD in the presence of TCP-SACK

6.1. TCP-SACK overview

TCP-SACK [10] is an extension to TCP that makes it behave more like a Selective Repeat protocol, rather than a Go-Back-N protocol. TCP-SACK can recover from multiple segment losses from the same window without a timeout, using a selective retransmission of the lost segments during the fast-recovery phase. The main idea behind TCP-SACK is that the receiver informs the sender of non-contiguous blocks of data that have been received and queued [10]. TCP-SACK makes use of the options field in the TCP header of the returning ack segments, and stores there pairs of sequence numbers, each represent a single non-contiguous block. The TCP header can hold up to 4 blocks in such a representation. If the Time-Stamp option

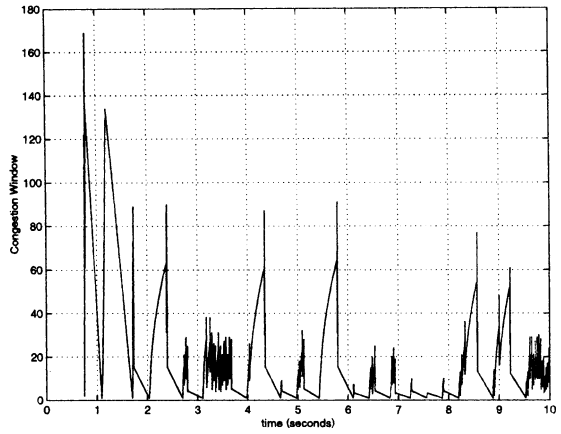

Fig. 17. Congestion Window with EPD. EPD-threshold = 90%.

is used (for the PAWS [11] algorithm, to prevent wrapping of sequence numbers in high transfer rate networks), SACK has room for only 3 blocks.

The SACK receiver must indicate in every ack the block to which the segment that has triggered this ack belongs. Hence, it would be enough to report just one block in the SACK option in order to allow the sender to build an exact map of the receiver's queue. Still, three blocks are used in order to provide some redundancy in the case of ack loss. The SACK option does not require the receiver to hold the segments reported in a SACK option but not yet acknowledged. This implies that the sender must not discard data from its buffer until it was acked.

The RFC of TCP-SACK [10] does not explicitly specify the SACK sender behavior. In [12] the authors propose the following scheme for the sender. The sender keeps an additional bit for every segment in its queue, called SACK-bit. When the server receives an ack with SACK info, it turns that bit on for every segment within the boundaries of the reported blocks. The trigger for retransmission remains 3 duplicate acks. Fast-recovery phase is exited when an ack for all the data that was outstanding upon initiating fast-retransmit is received, as proposed for TCP-Reno in Ref. [13]. This is in contrast to what TCP-Reno does: it exits fast-recovery immediately after an ack for the retransmitted segment is received. Upon entering fast-retransmit, the sender initializes a variable called *pipe* to the value of the current window size minus 3. This variable reflects the number of outstanding segments in the pipe between the sender and the receiver. In TCP-Reno, *cwnd* was used for this purpose. However, because TCP-SACK may *retransmit* several old segments during fast-recovery, rather than transmitting only new segments, *cwnd* can no longer track the number of outstanding segments. Because it no longer aims at estimating the number of outstanding segments, but only serves for congestion control, upon entering fast-retransmit *cwnd* is shrunk to *cwnd*/2 and not to *cwnd*/2 + 3 as in TCP-Reno. The congestion window grows or shrinks exactly as with TCP-Reno. However, the requirements for sending a segment are modified. With TCP-Reno the sender could retransmit a segment if $highest\_ask + cwnd > last\_sent$. In such a case a segment is transmitted from the right edge of the window. With SACK, the sender can transmit a segment if $pipe < cwnd$, and then the transmitted segment is chosen from the SACK info. If no such segment exists, i.e. the SACK bit is turned on for all segments, the sender may send a new segment. The *pipe* variable only determines

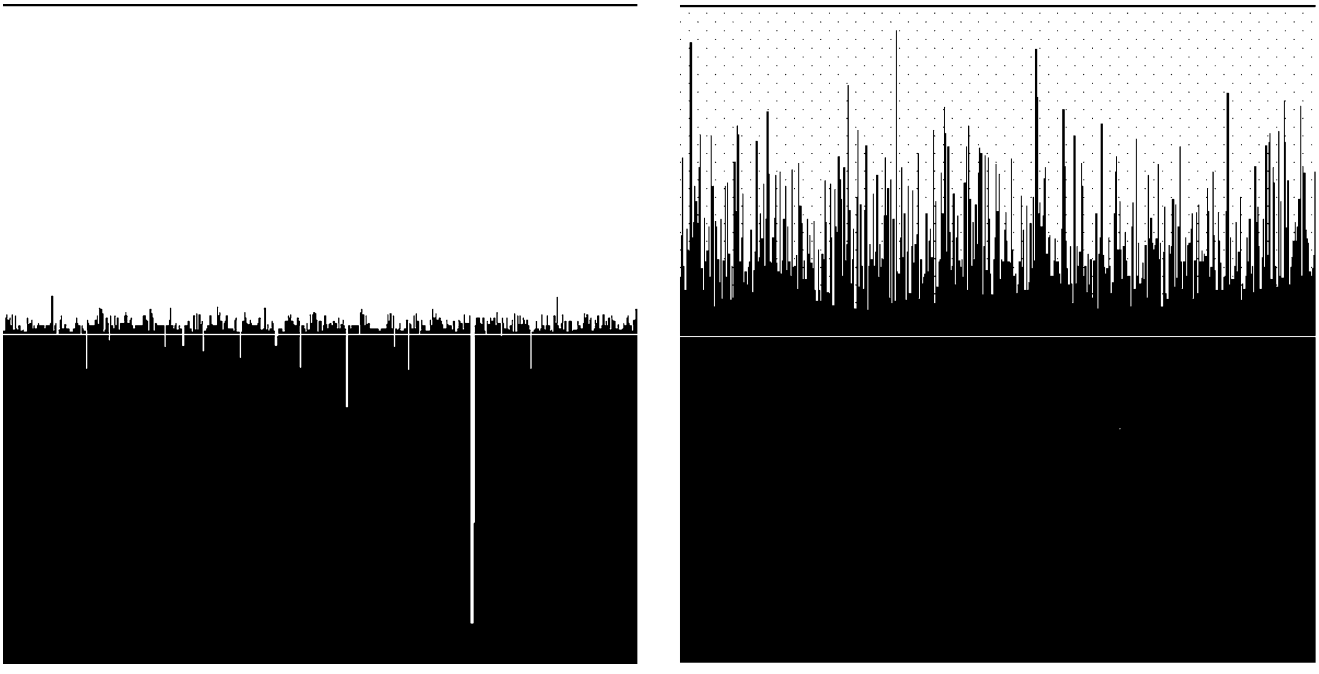

Fig. 18. Occupied buffer space of EPD (left) and BPD (right). EPD-threshold is 50% when EPD is used, and LT is 50% when BPD is used.

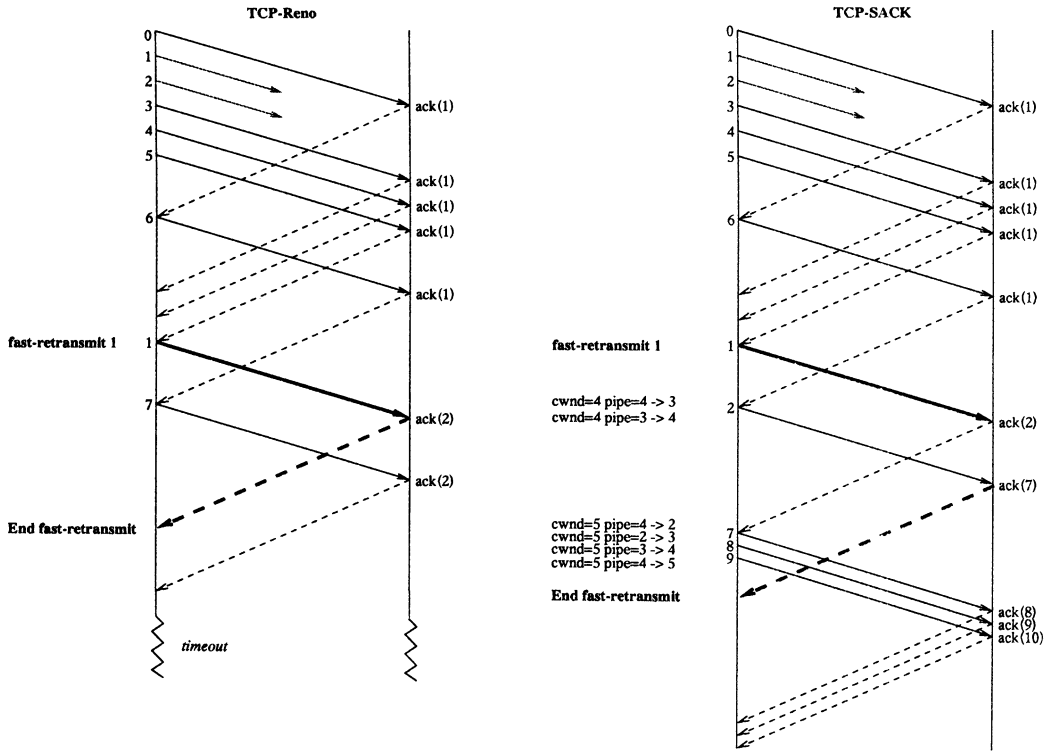

Fig. 19. TCP-SACK can recover from multiple losses without a timeout.

when the sender may send data, rather than *when* and *what* to send as *cwnd* does in traditional TCP implementations.

The value of *pipe* is incremented by one when the sender either sends a new segment or retransmits an old one. It is decremented when the sender receives a dup ack with a SACK option reporting that a new segment has been received and queued by the receiver. The SACK sender has a special treatment for a partial ack, namely an ack received during fast-recovery that advances the Acknowledgement Number field, but does not take the sender out of fast-recovery. For such an ack, the sender decrements *pipe* by two segments, rather than only one, for the following reason: when fast-retransmit is initiated, *pipe* is initialized to $cwnd - 3$, following the assumption that only one segment was lost. However, if several segments have been lost, *pipe* holds a larger value than the actual number of segments in the pipe. Each partial ack represents a segment that was lost before fast-retransmit began, which the initial value of *pipe* did not take into account, as well as segment that has just left the pipe. Fig. 19 demonstrates how TCP-SACK can recover from multiple losses without a timeout, while under the same conditions, TCP-Reno requires a timeout in order to recover.

6.2. BPD in the presence of TCP-SACK

TCP-SACK dramatically improves the performance of TCP by addressing the timeouts problem, as does BPD. In a typical scenario that was examined in Ref. [14], TCP-SACK reduced the number of timeouts from 50 to 2. Hence, BPD cannot significantly improve the throughout of TCP if SACK is used. However, BPD can still contribute to the overall performance of the network if some of the connections use TCP-Reno while other use TCP-SACK. Simulations in Ref. [14] show that when two TCP-Reno connections share the same link, they receive an even share of the bandwidth more or less. However, when one TCP-SACK connection and one TCP-Reno connection share the same link, TCP-SACK gets almost 100% more bandwidth than Reno. This results in low fairness among the connections. If BPD is employed in the ATM layer, then the bandwidth is shared among all the connections, SACK and non-SACK, more fairly as shown in the following.

In order to examine the effect that BPD has when SACK is being used, we consider a simple model, consisting of two connections, *connection* 1 and *connection* 2, that share the same buffer at the congested switch, and the same outgoing link. Both incoming links to the congested (gray) switch are of 45 Mbit/s, and the outgoing link is of 75 Mbit/s, introducing possible congestion. Connection 1 and connection 2 will be either Reno or SACK. The gray switch has a buffer of 400 cells, and is capable of performing BPD. The simulated network topology is shown in Fig. 21.

We first examine the results with BPD turned off. We check the throughput when both connections are Reno, and when connection 1 is SACK and connection 2 is Reno. The fairness index is computed as suggested in Ref. [15]: $\left[\sum x_i\right]^2 / n \sum x_i^2$, where n is the number of connections,

	Connection	TCP Version	Number of Timeouts	Throughput	Fairness	Overall throughput
(a)	Connection 1	Reno	24	0.454	0.995	0.584
	Connection 2	Reno	22	0.517		
	Connection 1	SACK	3	0.910	0.702	0.662
	Connection 2	Reno	34	0.193		
(b)	Connection 1	Reno	0	0.8122	0.999	0.98
	Connection 2	Reno	0	0.8216		
	Connection 1	SACK	0	0.8109	0.999	0.98
	Connection 2	Reno	0	0.8228		

Fig. 20. (a) Performance without BPD, (b) performance with BPD.

and x_i is the throughput of the $i'th$ connection in Mbit/s. A fairness index of 1 indicates a perfect fairness, where each connection gets the same share. In a less fair situation, the fairness index will be smaller than 1.

We first examine the results when BPD is not used. As can be seen from Fig. 20 (a), when two Reno connections are competing with each other, they receive more or less the same share. The fairness in this case is good, but both connections suffer from many timeouts and thus low throughput. When we replace one Reno connection with a SACK connection, we see that the SACK connection has huge advantage over the Reno connection in terms of throughput. It experiences only three timeouts, and reaches fairly high throughput, as opposed to Reno that suffers from 34(!) timeouts. Reno cannot compete against SACK in preventing timeouts, so it is left behind with much lower throughput. This results in a very low fairness index, and in low overall throughput.

When applying BPD, we expect that the number of timeouts will decrease dramatically, allowing Reno to compete with SACK, and the overall throughput to increase. We set the following BPD parameters: lower-threshold = 0.4, upper-threshold = 0.9, $Gap = 20$. Fig. 20 (b) summarizes the results. Indeed, BPD eliminates all the timeouts in both cases. Without timeouts, Reno competes with SACK as equal, and both get almost the same bandwidth share. Moreover, the overall utilized bandwidth is almost 100%.

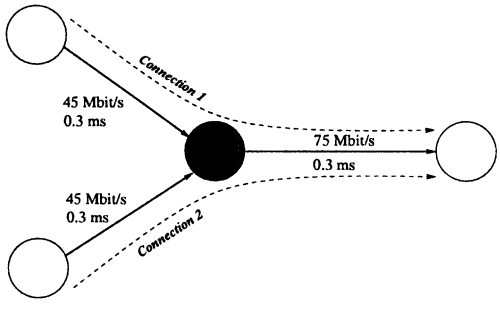

Fig. 21. The topology used to test BPD in a mixed environment of Reno and SACK TCP connections.

7. Conclusions

The paper has shown that the poor throughput TCP achieves in ATM networks can be attributed to two main reasons: transmission of useless cells from corrupted packets, and timeouts. The first problem has been fixed by previously proposed discard policies called PPD and EPD. However, in many cases, these schemes have little effect on the throughput, which remains low because of the timeouts problem. A new discard policy, called BPD, was introduced. BPD spaces segments losses far from one another, resulting in almost no timeouts. BPD significantly improves performance compared to EPD + PPD and achieves almost the maximum possible throughput.

The paper has shown that in contrast to the standard TCP-Reno, the new TCP option, called SACK, enables to achieve maximum throughput even without BPD. However, without BPD when a TCP-Reno and a TCP-SACK share the same links, the SACK connections get much more throughput and the overall bandwidth is not fully utilized. In the presence of BPD, the two connections get equal share of the bandwidth because BPD eliminates the advantage of SACK over Reno. Moreover, the overall bandwidth is fully utilized by all the connections.

References

[1] J.A. Grenville, M.A. Keith, Packet reassembly during cell loss, IEEE Network 7 (5) (1993) 26–34.
[2] A. Romanow, S. Floyd, The dynamics of TCP traffic over ATM Networks, in: Proceedings, SIGCOMM Conference, London, UK, August 1994, pp. 79–88.
[3] H. Tzeng, C. Ikeda, H. Li, K. Siu, H. Suzuki, TCP Performance over ABR and UBR Services in ATM, in: IPCCC'96, March 1996.
[4] R. Goyal, Performance of TCP over UBR + , October 1996.
[5] S. McCanne, S. Floyd, ns-Network Simulator.
[6] V. Jacobson, Congestion avoidance and control, ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, vol. 18, no. 4, August1988, pp. 314–329.
[7] W. Stevens, TCP Slow Start, Congestion Avoidance, Fast-Retransmit, and Fast Recovery Algorithms, RFC-2001, January 1997.

[8] V. Paxson, M. Allman, W. Stevens, TCP Congestion Control, RFC-2581, April 1999.
[9] Jonathan Turner, Maintaining high throughput during overload in ATM switches, in: Infocom, San Fransisco, California, March 1996, pp. 287–295.
[10] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgement Options, RFC-2018, October 1996.
[11] V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, RFC-1323, May 1992.
[12] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, ACM Computer Communication Review 26 (3) (1996) 5–21.
[13] J.C. Hoe, Improving the Start-up Behavior of a Congestion Control Scheme for TCP, Sigcomm'961996, pp. 270–280.
[14] R. Bruyeron, B. Hemon, Experimentations with TCP selective acknowledgement, ACM Computer Communication Review 28 (2) (1998).
[15] R. Jain, D. Chiu, W. Hawe, A quantative measure of fairness and discrimination for resource allocation in shared computer systems, Tech. Rep. TR-301, DEC Research Report, September 1984.