# High Speed Internet Access Through Unidirectional Geostationary Satellite Channels

Ina Minei Reuven Cohen
Computer Science Department
The Technion
Haifa 32000, Israel
e-mail: {aminei,rcohen}@cs.technion.ac.il

**Abstract**

One of the proposed solutions for increasing the speed of Internet access is to connect the home user to a direct satellite channel, at a speed 20 times faster than that of an average telephone modem. Communication over satellite links is often characterized by sporadic high bit-error rates and burst losses. This is especially true when working in the Ka band, where weather conditions greatly affect link availability. Under such conditions, the TCP protocol that is predominantly used by data applications, degrades dramatically in performance. Using simulations, this paper studies the performance of TCP under different network conditions. Several modifications, that take advantage of the special properties of the satellite channel, are proposed and a new sender algorithm which can efficiently handle burst losses is presented. The main attractiveness of the proposed new sender algorithm is that it can be implemented only at the satellite ground station, rather than at every server in the world.

# 1 Introduction

The Internet is becoming an important source of information for an ever growing community of users. When a home user connects to the Internet through a telephone modem, the speed of the received information is less than 56Kb/sec, due to the modem constraints. Therefore, for large amounts of information the transfer time is too long.

Developments in satellite technology allow home users to receive data directly from a geostationary satellite channel at a rate 20 times faster than of an average telephone modem. Connectivity of a home user to the Internet is achieved using a configuration similar to that depicted in Figure 1. In this configuration, the home PC accesses an Internet server over a slow modem link and requests data. Rather than sending the data back over the slow modem link, the server sends the data through the Internet to a satellite ground station. The ground station, also known as an uplink, then forwards the data over a fast satellite channel to the home user. Such a configuration is already commercially available.

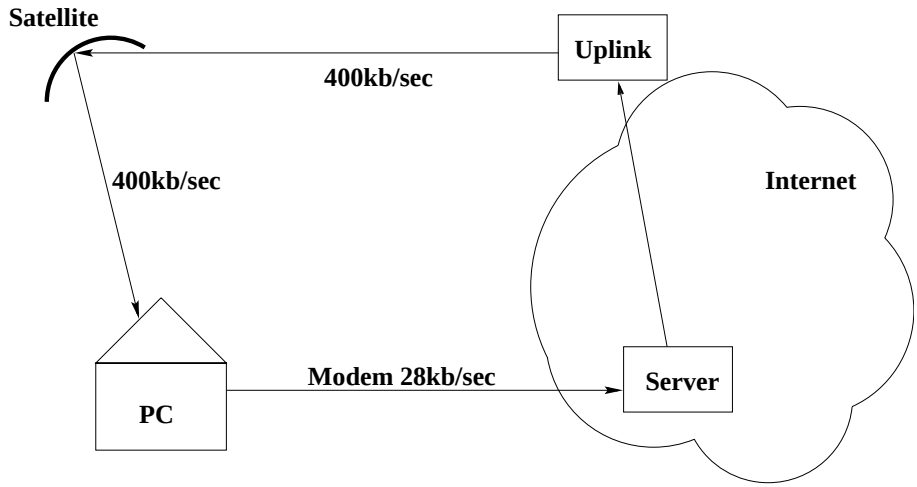


Figure 1: Internet access through a satellite channel

A common way to characterize the performance of an access network is in terms of the throughput observed by the applications running above the TCP layer. The throughput achieved depends on three factors. First, on the bandwidth available for data and acknowledgments. Second, on the packet loss rate, and finally on the specific TCP implementation. Running TCP on a network with a satellite link poses two problems:

- Burst losses: the satellite channel may suffer sporadic burst losses, especially during bad weather conditions. TCP does not perform well under burst losses. The mechanism for efficient recovery of lost packets, called fast-retransmit, fails when several consecutive packets are lost, drastically affecting the throughput.

- A long round trip delay: the long propagation delay over the satellite channel, about 250ms, introduces high TCP timeout values, which lead to low throughput when lost packets have to be retransmitted.

This work studies the performance of TCP on networks that contain a satellite channel. The study is conducted using the TCP *ns* simulator from LBL [6]. Based on the observations made, we propose to split the TCP connection at the satellite uplink node into two separate connections: a satellite connection and a terrestrial connection. Then, on the satellite connection, we propose to use a smaller number of duplicate acknowledgments to trigger fast-retransmit, and a window of constant size. Finally, we propose a new TCP sender algorithm, which handles more efficiently burst losses in a satellite channel. The proposed algorithm is tailored for a TCP connection over a satellite link, where the window is large and losses are caused by media errors rather than by router congestion. This is mainly because the algorithm reacts to packet losses by sending bursts of retransmissions. In a heavily loaded network such an approach aggravates the congestion and increases the loss. A major attractiveness of all the proposed enhancements is that they affect only the TCP implementation at the uplink. No modification is needed at the client PCs or at any remote server.

The rest of this paper is organized as follows. Section 2 introduces a brief description of the TCP functions relevant to this work. Section 3 characterizes the performance of TCP under different network conditions. Section 4 examines the benefit of splitting the TCP connection at the uplink, and discusses several mechanisms for taking advantage of the unique satellite channel properties in order to improve performance. Section 5 presents a new TCP sender algorithm, better adapted to handle bursts of error losses in a satellite link, to be implemented at the uplink node. Finally, section 6 concludes the paper.

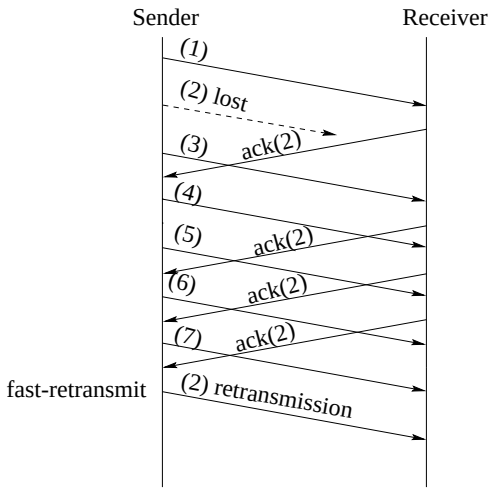


Figure 2: Fast-retransmit

## 2 TCP Background

Current TCP implementations [5, 9] contain a number of algorithms aimed at controlling network congestion. These algorithms include slow-start, congestion-avoidance, fast-retransmit and fast-recovery. Together they define the congestion window, *cwnd*, as an estimate of the maximum number of packets that can be sent without overloading the network. The TCP sender never sends more than the minimum of *cwnd* and the receiver's advertised window.

The TCP sender operates in one of two modes: slow-start or congestion-avoidance. The main difference between these modes is the rate of increasing *cwnd*. The sender determines its mode based on the values of *cwnd*. As long as *cwnd* is smaller than a threshold value, called *ssthresh*, the sender works 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 packet and grows it by one with every acknowledgment received. Assuming an acknowledgment (ACK) is sent for every data packet received, which is not the case when the receiver uses delayed ACKs [9], this results in doubling *cwnd* every round trip. In contrast, in congestion-avoidance mode, the sender increases the value of *cwnd* by $1/cwnd$ for every data packet received, which is approximately equivalent to an increase of one packet every round trip time (RTT), yielding linear growth.

When a packet is lost, the subsequent packets are received by the receiver out of order. An out of order packet triggers an acknowledgment carrying the same sequence number as a previous

acknowledgment. Such an ACK is therefore referred to as a duplicate ACK (dupACK). The sender attributes the first and second dupACKs to a possible out of order routing. However, when the third dupACK is received, the sender assumes a packet has been lost. It then invokes the fast-retransmit procedure by immediately retransmitting the missing packet, as depicted in Figure 2. After receiving an ACK for the retransmitted packet, the sender performs a procedure called fast-recovery by shrinking *cwnd* to half and entering congestion avoidance mode.

The fast-retransmit mechanism is not always triggered when a packet is lost. As a simple example, consider the case where the window size is only 4, in which case only two dupACKs can be received. To detect a packet loss even in this case, the sender maintains a retransmission timer. This timer is set when TCP sends data, but only if the timer is not currently enabled. The retransmission timer is turned off when an ACK for all outstanding packets is received. If only part of the data is acknowledged, the timer is restarted. When the timer expires, the oldest packet for which an ACK has not been received is retransmitted. The time the sender is idle, waiting for the timer to expire, is called a timeout. After a timeout, the sender retransmits the lost packet, shrinks *cwnd* to 1 and enters slow-start. In addition, it sets *ssthresh* to half the value *cwnd* had when the loss was detected. Thus, timeouts have a significant impact on throughput both because they introduce a period of idle time and because they shrink the window.

The timeout value is calculated dynamically, based on the round trip time measurements the sender performs throughout its operation. The timeout, RTO is calculated as: $RTO = average(round\_trip\_time) + 4 \times mean\_deviation(round\_trip\_time)$ . One of the problems with the calculation of the timeout value is the fact that TCP uses a coarse granularity timer, with 500ms delay between successive invocations of the timer routine, to time the round trip delay. A long round trip delay is more frequently timed as 1 clock tick than a short round trip delay, thus yielding a larger timeout value. For example, our simulations show that in a network with 50ms actual round trip delay, the mean RTO measured is 1.6 seconds, whereas in the considered satellite network, with 290ms actual round trip delay, the mean RTO measured is slightly more than 2 seconds. (The exact details of the average and mean deviation calculation, as well as the implementation of this formula using only integer arithmetic are found in [8].)

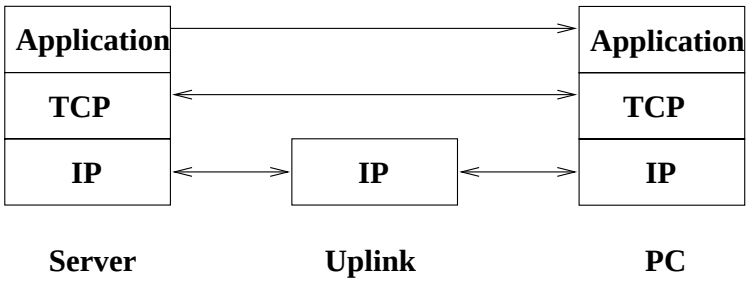


Figure 3: Protocol layer for a direct connection between the server and the PC.

# 3 The Performance of a Direct Satellite TCP Connection

## 3.1 Simulation Model

In this section we simulate the performance of TCP in the following network configuration. The satellite channel between the uplink station and the PC is of 400Kb/sec with 250ms delay. These characteristics were chosen in accordance to the data published for commercial systems. Since we want to concentrate on the satellite channel performance, we assume that the terrestrial channel between the server and the uplink station, which is actually a path through routers in the Internet, is not a bottleneck. Hence, its bandwidth is also assumed to be 400Kb/sec, and its delay is modeled as 40ms, according to typical Internet delays (80ms RTT). The mixed modem/Internet channel between the PC and the server is 28,800b/sec, for reasons that will be explained below. The delay of this channel is modeled as 40ms, and represents the propagation delay between the PC and the server and the transmission delay and other delays encountered at the Internet Service Provider.

Throughout this work we refer to a transparent satellite system, i.e. one that simply forwards the data received from the uplink, without any processing. Furthermore, we assume a dedicated satellite channel, where no congestion is possible. The effect of sharing the link between multiple users is orthogonal to the addressed issues and is therefore not covered in the paper.

We start with the simple case, where a single direct TCP connection is established between the server and the PC, as shown in Figure 3. In this configuration, the uplink is simply a router on the way. The data is forwarded from the server to the PC through the Internet and the satellite link. The ACKs are returned from the PC to the server over a mixed modem/Internet link.

Throughout the paper, the network traffic is assumed to be large FTP transfers. In keeping

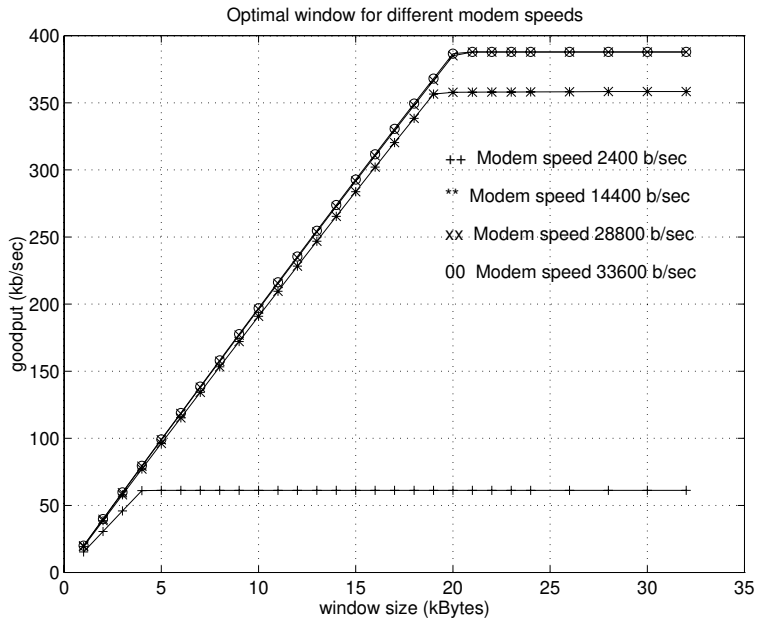


Figure 4: Throughput achieved for various modem speeds and window sizes, for a 5MB file.

with most commercial implementations, the receivers are assumed to implement delayed ACKs. The length of a data packet is 512 bytes and of an ACK packet is 40 bytes.

Since our main goal is to study the effect of packet loss on throughput, all the parameters used in the simulations are set to values that maximize throughput in the absence of loss. As depicted in Figure 4 the optimal choice from throughput considerations is a modem speed of 28,800b/sec and a window size of 21KB.

The size of the transfered file, 5MB, is chosen to minimize the effect of slow-start on the throughput. As shown in Figure 5, smaller files get lower throughput due to the low speed of the connection during slow-start, whereas for larger files the throughput improvement is negligible. The throughput of a 5MB file transfer is 387Kb/sec, namely 96% of the theoretical maximum.

### 3.2 The Effect of Acknowledgment Losses

Figure 6 shows that ACK losses only slightly affect the throughput. This is because TCP uses a cumulative acknowledgment strategy. Thus, an ACK loss is compensated for by the arrival of any subsequent ACK, and only rarely results in a data packet retransmission.

The small throughput degradation associated with ACK losses can be attributed to the following reasons. First, ACK losses slow down the transmitter, because the increase in the window size and

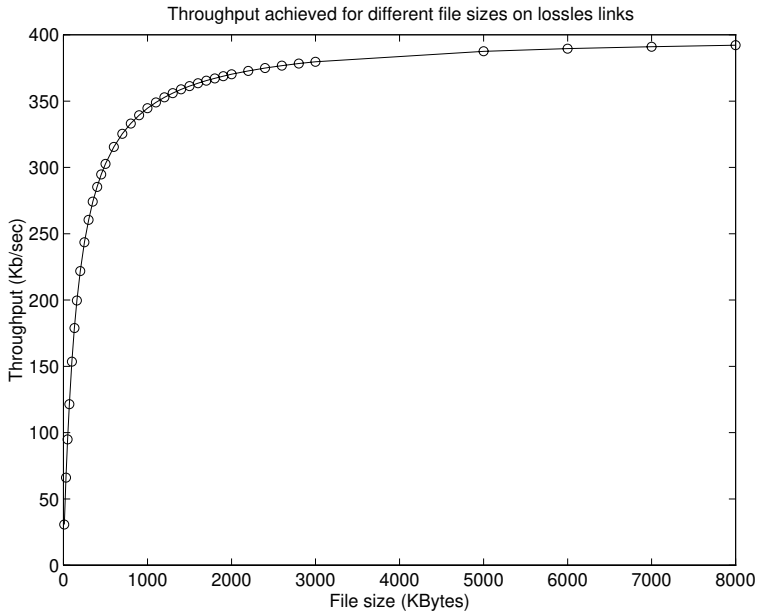


Figure 5: Throughput achieved when transferring files of different sizes over lossless links.

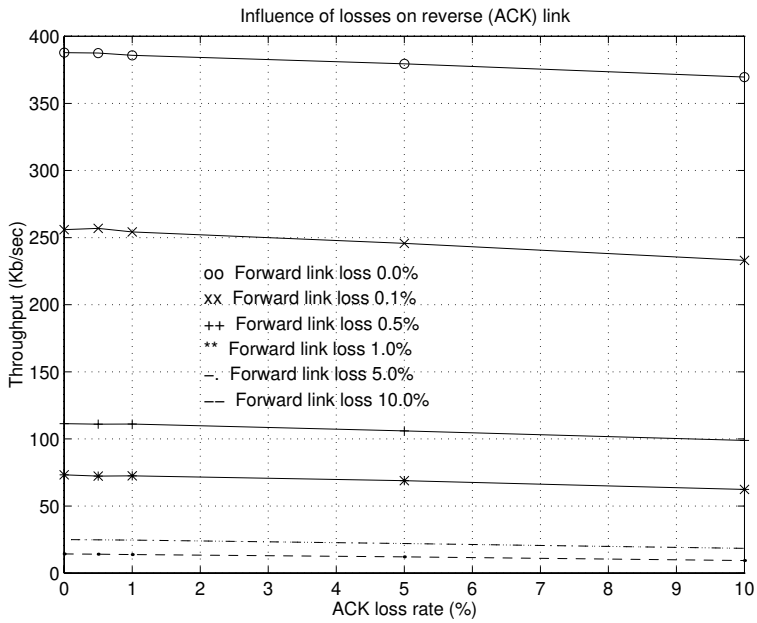


Figure 6: Throughput degradation in the presence of losses on the reverse (acknowledgment) link.

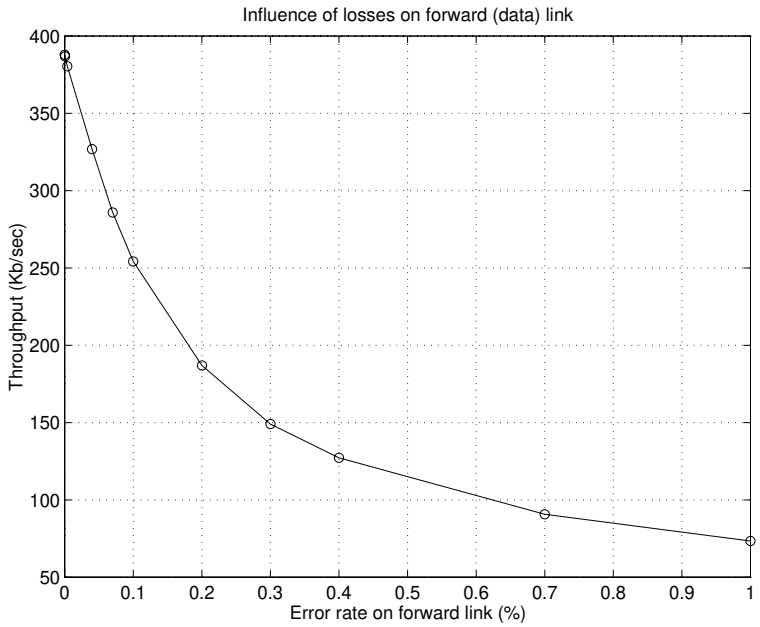


Figure 7: Throughput degradation in the presence of losses on the forward (data) link.

the opening of the window depend on the receipt of ACKs. Second, ACK losses may cause timeouts, especially when the window is small. As mentioned in section 2 timeouts have a significant impact on throughput. Finally, ACK losses affect the transmission of new data packets during fast-retransmit mode because in fast-retransmit a new packet is sent only when an acknowledgment is received.

The simulation shows that the effect of ACK loss on throughput is more severe when the data link is lossy. These results are explained by the effect of ACK losses during fast-retransmit.

### 3.3 The Effect of Data Packet Losses

The effect of data packet losses is much more severe than of ACK losses. Figure 7 shows the dramatic reduction in throughput caused by data packet loss for a 5MB data transfer, when the acknowledgment link is lossless. A loss rate as low as 0.04% causes a 15% drop in throughput, and a 1% loss rate slashes the throughput by 81%.

There are two main reasons for the dramatic throughput degradation. The first and most important is timeouts. As explained in Section 2, the sender is idle for about 2 seconds waiting for the timer to expire, and operates below its optimum speed a few round trip times afterwards during slow-start mode. The probability of a timeout increases with the packet loss rate because

for high loss rates, the probability of losing several packets in the same window, which usually leads to timeouts, increases. Also, after a timeout, the sender works with a window smaller than 4 for 2 round trip times, and during this period even a single packet loss may cause a timeout.

The second reason for throughput degradation is the effect that a packet loss has on the window size. After any retransmission, whether following a timeout or following fast-retransmit, the sender shrinks its transmission window to one or to half its original size, respectively. Thus, following a loss, the sender operates below its optimum speed for a few round-trip times. If losses occur at the time the window is growing back towards its optimal size, they lower the window yet again. Moreover, if a loss occurs while the window grows in slow-start, the growth rate turns from exponential to linear, and it takes even longer for the window to reach the optimal value.

# 4 Improving Performance Using a Split Connection

When a direct connection is used, a packet lost anywhere on the path from source to destination has to be retransmitted by the source and be routed all the way to the destination. In [2], the benefits of using a TCP proxy to split the TCP connection into two separate connections is discussed in the context of a hybrid fiber coaxial network. In this section we study the same approach in a satellite network, by placing a TCP gateway at the uplink station, as depicted in Figure 8. Consequently, two TCP connections are created. One connection is between the server and the uplink over the wired network. The other connection is between the uplink and the PC. The forward (data) channel of this connection is the satellite link, whereas the reverse (ACK) channel consists of the modem link and an Internet path. One advantage of the split connection approach is that it separates the losses on the satellite link from the losses on the Internet, allowing local recovery of lost packets. Another advantage of the split connection approach is that it allows tailoring the TCP implementation on each of the connections to best suit the characteristics of the underlying channel.

In our simulation model, the only parameter affected by this change is the window size, which is now calculated separately for each of the two TCP connections. The optimal window size for the connection between the server and the uplink is 5KB (10 packets), and for the connection between the uplink and the PC is 16KB (32 packets) .

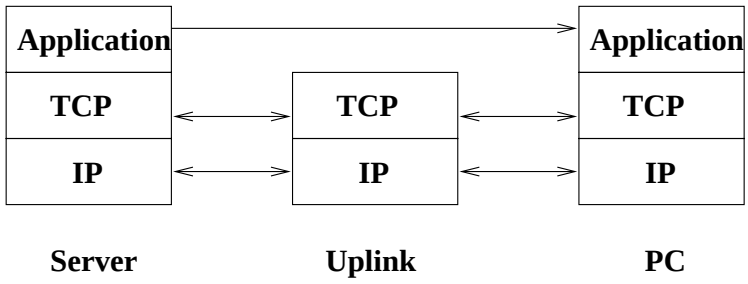


Figure 8: Protocol layer for a split connection configuration.

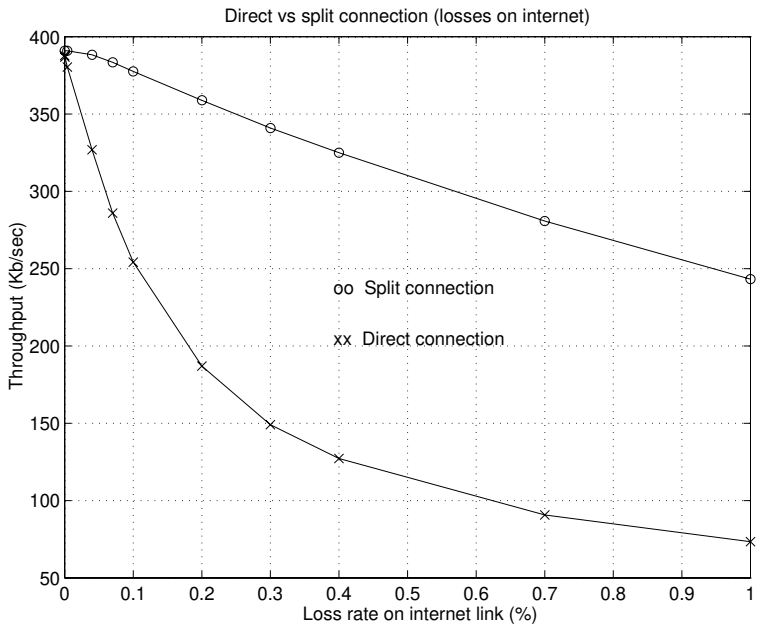


Figure 9: Throughput comparison between a direct and split connection, when all losses are on the Internet link.

### 4.1 Performance Improvement of a Split Connection with Data Loss

Figure 9 shows that in the case where losses occur only on the Internet link, TCP performance in the split connection configuration is much more robust than in the direct connection configuration. For a direct connection with 1% data loss rate, the throughput degrades by 81% from 387Kb/sec to just 73Kb/sec, whereas for a split connection, the degradation of throughput is of just 37%, from 390Kb/sec to 243Kb/sec. This happens because no retransmissions are made over the long delay satellite link and a lost packet is recovered much faster than in the direct connection model. Moreover, due to the lower RTT on the Internet connection, growing the window back to its optimal size is also faster.

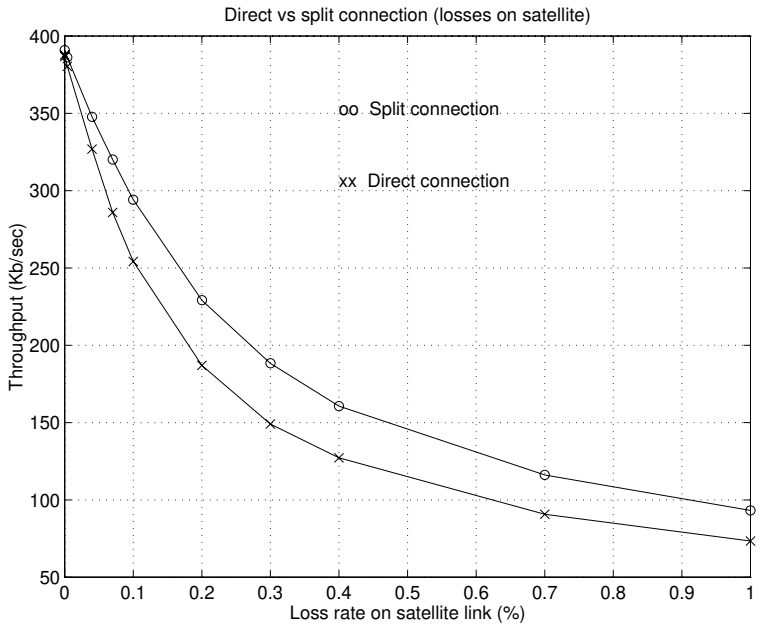


Figure 10: Throughput comparison between a direct and split connection, when all losses are on the satellite link.

For the same reasons, throughput improvement is not expected to be dramatical when losses occur on the satellite link, as confirmed by Figure 10.

## 4.2 Taking Advantage of the Satellite Channel Specific Properties

As already indicated, besides the improvement in performance, the other advantage of splitting the TCP connection is that the satellite channel is isolated from the rest of the Internet. This channel has two unique properties which differentiate it from the rest of the Internet. The first property is that packets sent on the satellite channel cannot be routed out of order. The second property is that congestion is not possible and therefore the only reason for packet losses is transmission errors. Both properties are attributable to the fact that there does not exist any router on the channel between the uplink station and the home PC.

In [1] it is observed that on a link where in-sequence delivery of packets is guaranteed, the receipt of the first dupACK is a clear sign that a data packet has been lost and fast-retransmit can be immediately triggered, without waiting for two additional dupACKs. This scheme, referred to in [1] as "super fast-retransmit", accelerates the recovery of fast-retransmitted packets and helps avoiding timeouts in many cases where the sender can receive a single dupACK instead of three.

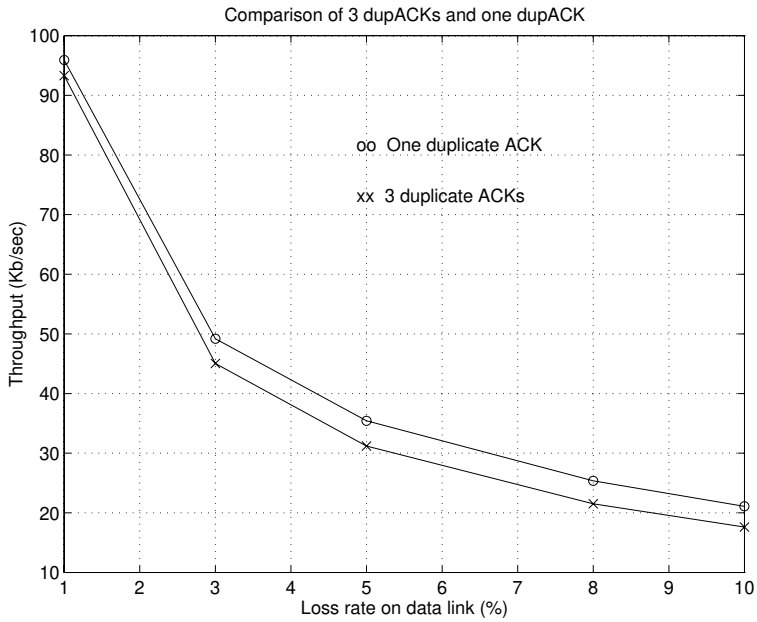


Figure 11: Comparison of fast-retransmit and super fast-retransmit on the satellite portion of a split connection.

In particular, if fast-retransmit is invoked only after 3 dupACKs then timeouts always occur when the window is of 4 packets or less, since in this case there are not enough data packets to trigger 3 dupACKs. Moreover, when multiple losses occur in the same window, fast-retransmit can usually recover the first lost packet but not the others, as will be explained in section 5.1.

Figure 11 shows that running super fast-retransmit on the satellite portion of the split connection is worthwhile only for very high data loss rates. For instance, an improvement of 19% for a 10% data loss rate, and of 13% for a 5% data loss rate. For loss rates below 1% on the data link, the performance of the two algorithms is comparable. This is because at low error rates, there is a small probability of losing one or more packets in the exact configuration that does cause a timeout when using fast-retransmit, but does not cause a timeout when using super fast-retransmit.

The second property of the satellite channel in the split connection configuration is that congestion is not possible. Thus, there is no need to slow down the transmitter (by shrinking its window) after a packet is lost. Moreover, the sender does not have to probe for the network capacity using slow-start. Hence, the sender can proceed using a fixed window size, which is optimized to the delay-bandwidth product of the satellite channel. The throughput increases since the slow

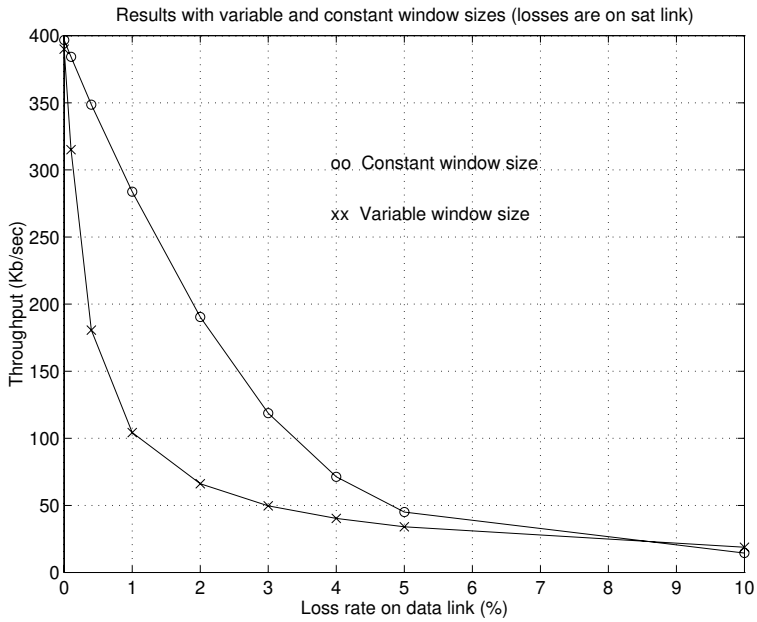


Figure 12: Throughput achieved with constant and variable window sizes when losses occur on the satellite link.

operation of the sender at the beginning of the connection and after packet loss detection is avoided.

Figure 12 shows the improvement achieved when transferring a 5MB file using a constant optimal window over a lossy satellite link. The throughput is improved by about a factor of 2 for 0.4% packet loss rate and by about a factor of 3 for 1% loss rate. For losses of more than 5% the improvement achieved decreases, and at 10% packet loss rate the performance of the constant window and variable window are comparable. This is because for very high loss rates, the time wasted during timeouts becomes the dominant factor influencing throughput.

# 5 A New TCP Sender Algorithm for Burst Loss Recovery

Communication over satellite links is often characterized by sporadic high bit-error rates and burst losses. This is especially true when working in the Ka band, where weather conditions greatly affect link availability. The effect of errors on the channel throughput was discussed in Section 3 and 4. The present section focuses on burst losses, which cause retransmission timeouts and significant throughput degradation. We first analyze the behavior of TCP-Reno in the presence of burst losses and then present a modified version for the TCP sender algorithm at the uplink in the split connection configuration, that can recover a loss of multiple packets in a burst without a

timeout. This version is suitable only for satellite channels, as explained later.

### 5.1 TCP-Reno in the Presence of Burst Losses

The current implementation of TCP, TCP-Reno [8] assumes a packet has been lost and therefore has to be retransmitted if: (1) a timeout occurs; (2) 3 dupACKs are received. In the case of a timeout, the sender shrinks its window to 1 and initiates slow-start. This recovery procedure reduces the sender throughput significantly because the sender stalls while waiting for the timer to expire, and transmits well below the maximum capacity during slow-start. To reduce the probability of having timeouts, the TCP sender uses the fast-retransmit algorithm.

The sender is said to enter "fast-retransmit mode" upon receiving 3 duplicate acknowledgments. Upon entering fast-retransmit, the sender retransmits the missing packet and sets its window to half its original value $W$ plus 3, $\frac{W}{2}+3$. The sender exits fast-retransmit mode when an acknowledgment for the retransmitted packet is received.

In order to prevent a burst of packets from being transmitted when the ACK for the retransmitted packet is received, the sender increases its window by one with each ACK it receives while in fast-retransmit mode. When the number of outstanding packets becomes smaller than the window size, the sender injects a new packet into the network after every ACK received. This sequence of events is analyzed below and depicted in Figure 13.

In the following discussion we assume the sender uses its window fully. Thus, upon entering fast-retransmit mode, the number of outstanding packets is equal to the size $W$ of the window before entering fast-retransmit minus the 3 packets whose reception triggered the fast-retransmit mode. The total number of dupACKs that can be received during fast-retransmit is therefore $W - number\_of\_lost\_packets - 3$, but only $\frac{W}{2}-3$ of them increase the window from its value upon entering fast-retransmit ($\frac{W}{2}+3$) to its old value $W$. The remaining $(W - number\_of\_lost\_packets - 3)-(\frac{W}{2}-3) = \frac{W}{2} - number\_of\_lost\_packets$ dupACKs trigger the transmission of new data packets. Thus, $(\frac{W}{2} - number\_of\_lost\_packets)$ new packets enter the network between the time the window size becomes greater than $W$ and the time the acknowledgment for the retransmitted packet is received. Note that in fast-retransmit mode an ACK is generated for every received packet, even if the receiver implements "delayed ACK". When the ACK for the retransmitted packet finally

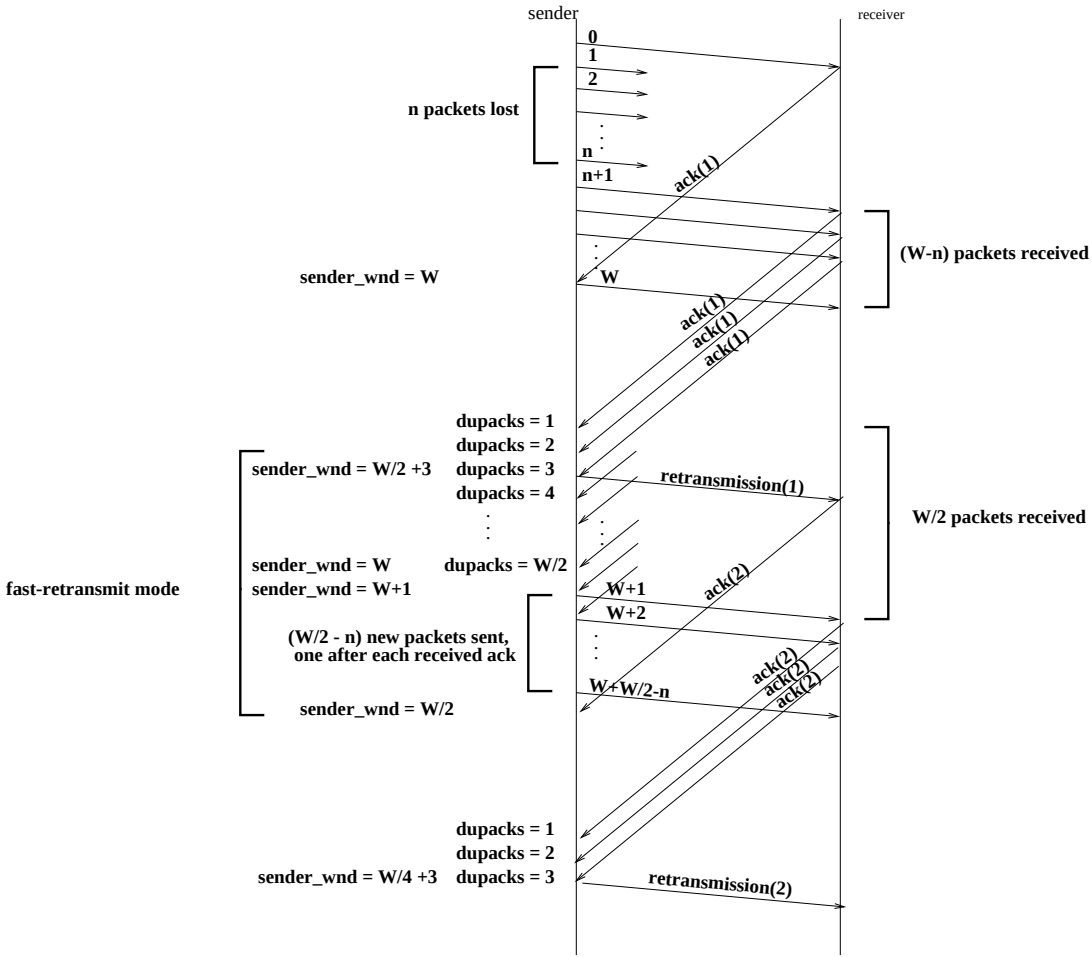


Figure 13: TCP-Reno in the presence of burst losses.

arrives, the window value is set to $\frac{W}{2}$.

If a single packet is lost in a window, it is recovered by fast-retransmit only if the window size is larger than 4. Otherwise, the 3 dupACKs needed for fast-retransmit cannot be generated, and the packet is recovered only after a timeout.

If two consecutive packets are lost in a window of size $W > 5$, the first packet is recovered by fast-retransmit. The second packet can be recovered by fast-retransmit only if more than 3 new packets are sent after the retransmission of the first lost packet, as can be seen in Figure 13. This yields the requirement $\frac{W}{2} - 2 \geq 3$. Solving for $W$ we see that a timeout must take place if the initial window is smaller than 10.

If three or more consecutive packets are lost a timeout must occur. After the recovery of the first lost packet, the window size is $\frac{W}{2}$. When the loss of the second packet is detected, the window size is set to $\frac{W}{4} + 3$. Therefore, at most $\frac{W}{4} - number\_of\_lost\_packets$ dupACKs will arrive after the window reaches its old value $\frac{W}{2}$. In this case, however, no new packet is injected into the network, because the number of outstanding packets $W + \frac{W}{2} - number\_of\_lost\_packets - 1$ is greater than the maximum value the window can reach $\frac{W}{2} + \frac{W}{4} - number\_of\_lost\_packets$. Since no new packets can be sent, the 3 dupACKs needed to trigger fast-retransmit cannot be generated, and a timeout occurs.

### 5.2 Related Work: How Other TCP Versions Handle Burst Losses

A modified version of TCP, called TCP new-Reno, is presented in [4]. This version aims at avoiding timeouts when multiple packets are lost in the same window by changing the behavior of the TCP sender during fast-retransmit as follows. The protocol defines a "fast-retransmit phase" as the time between the receipt of 3 dupACKs and the time when an ACK arrives for *all* the packets that were outstanding when the phase started. (This is in contrast to the regular Reno implementation, where the fast-retransmit mode lasts until the acknowledgment for the retransmitted packet arrives.)

When multiple packets are lost in the same window, the retransmission of the first lost packet triggers a so called *partial ACK*, i.e. an ACK that acknowledges some but not all the packets that were outstanding at the start of fast-retransmit. A partial ACK is treated as a signal that the packet whose sequence number is indicated has been lost and should be retransmitted. As an

example, consider the case where 2 consecutive packets, 1 and 2, are lost in a window of size 6. The loss of packet 1 is detected when 3 dupACKs bearing this sequence number are received, causing the fast-retransmit phase to be entered. The packet is retransmitted and is acknowledged one RTT later by an ACK carrying the sequence number of the next packet the receiver expects to receive, namely packet number 2. This is a partial ACK, indicating that packet 2 has also been lost. Hence, packet 2 is immediately retransmitted and no timeout occurs. This pattern is repeated if several packets are lost, implying that TCP new-Reno recovers one lost packet during each RTT.

Another approach for recovering from multiple losses is by using selective acknowledgments (SACKs), as described in [7]. With SACKs, the receiver can inform the sender of up to 3 non-contiguous blocks of data that have been received and queued. The TCP sender can deduce from this information the sequence numbers of all the lost packets and timeouts are avoided. A simulation study in [3] demonstrates the strength of TCP-SACK. However, its main drawback is that it requires changes at both the sender and receiver TCP. This is in contrast to TCP new-Reno and to the algorithm proposed in this paper, which only change the sender algorithm and therefore can be gradually deployed without affecting the software of the home PCs.

### 5.3 The New Sender Algorithm

We propose a change to the TCP sender algorithm, that enables the sender to recover from a burst of $N$ losses during a period of $2 \times RTT$, instead of $N \times RTT$ for TCP new-Reno or $M \times RTT + timeout + \log(N - M)$ for TCP Reno ($M < N$ is the number of packets that can be recovered using fast-retransmit; its value can be found by the analysis given in section 5.1). As explained below, the algorithm performs well only in channels where losses occur due to bursts of errors rather than due to congestion. In particular, it performs better than TCP new-Reno in a satellite channel, where RTT is relatively large.

Like the algorithm proposed in [4], our algorithm also uses the notions of fast-retransmit phase and of partial ACKs, defined as follows:

**Definition 1** *The fast-retransmit phase starts after 3 dupACKs have been received and ends when an ACK arrives for all the packets that were outstanding when the phase started.* [1] *During the*

[1] As already shown, fast-retransmit can be invoked in the considered satellite channel after the first dupACK. However, since the contribution of this change to the proposed protocol is negligible, we ignored this possibility in

*fast-retransmit phase the window is not slided (moved right).*

**Definition 2** *A partial ACK is an ACK that acknowledges some but not all the packets that were outstanding at the start of the fast-retransmit phase.*

The proposed algorithm tries to estimate the number of lost packets from the number of ACKs the sender receives. The considered lost packets are then retransmitted before 3 dupACKs are received for each of them.

The algorithm is demonstrated in Figure 14 and explained in the following. When a loss is detected following the receipt of 3 dupACKs, the sender enters the fast-retransmit phase. It retransmits the missing packet and remembers the number of outstanding packets.

After retransmitting the lost packet, a dupACK is received for every packet the receiver receives. This is the case even if the receiver implements the delayed ACK policy, because this policy is disabled when the receiver detects a missing packet. If the acknowledgment for the retransmitted packet does not confirm the receipt of all the packets that were outstanding at the retransmission time, the sender estimates the number of dropped packets as: $number\_of\_dropped\_packets = number\_of\_outstanding\_packets - number\_of\_duplicate\_acks$. For instance, in Figure 14(a) there are 8 outstanding packets upon entering fast-retransmit, and the number of dupACKs received is 4, so the sender deduces that 4 packets were dropped. Since the first dropped packet has already been recovered, the number of packets still needed to be retransmitted is $N = number\_of\_dropped\_packets - 1$. *The sender assumes all losses occured in a single burst*, and therefore retransmits $N$ consecutive packets starting with the first unacknowledged packet. In Figure 14(a), 3 consecutive packets are retransmitted, starting with packet number 2.

The burst of retransmissions triggers a series of partial acknowledgments, such as ack(3) and ack(4) in Figure 14(a). The reception of a single partial acknowledgment does not imply that the packet following the acknowledged one has to be retransmitted, because this packet might have already been sent in the retransmission burst. This is for example the case for packet 4 in Figure 14(a). However, if a *duplicate* partial ACK arrives, the sender can deduce that there is another "hole" in the original packet sequence. To cover this additional hole, another burst is scheduled for retransmission, starting with the packet following the last acknowledged packet. This

the following.

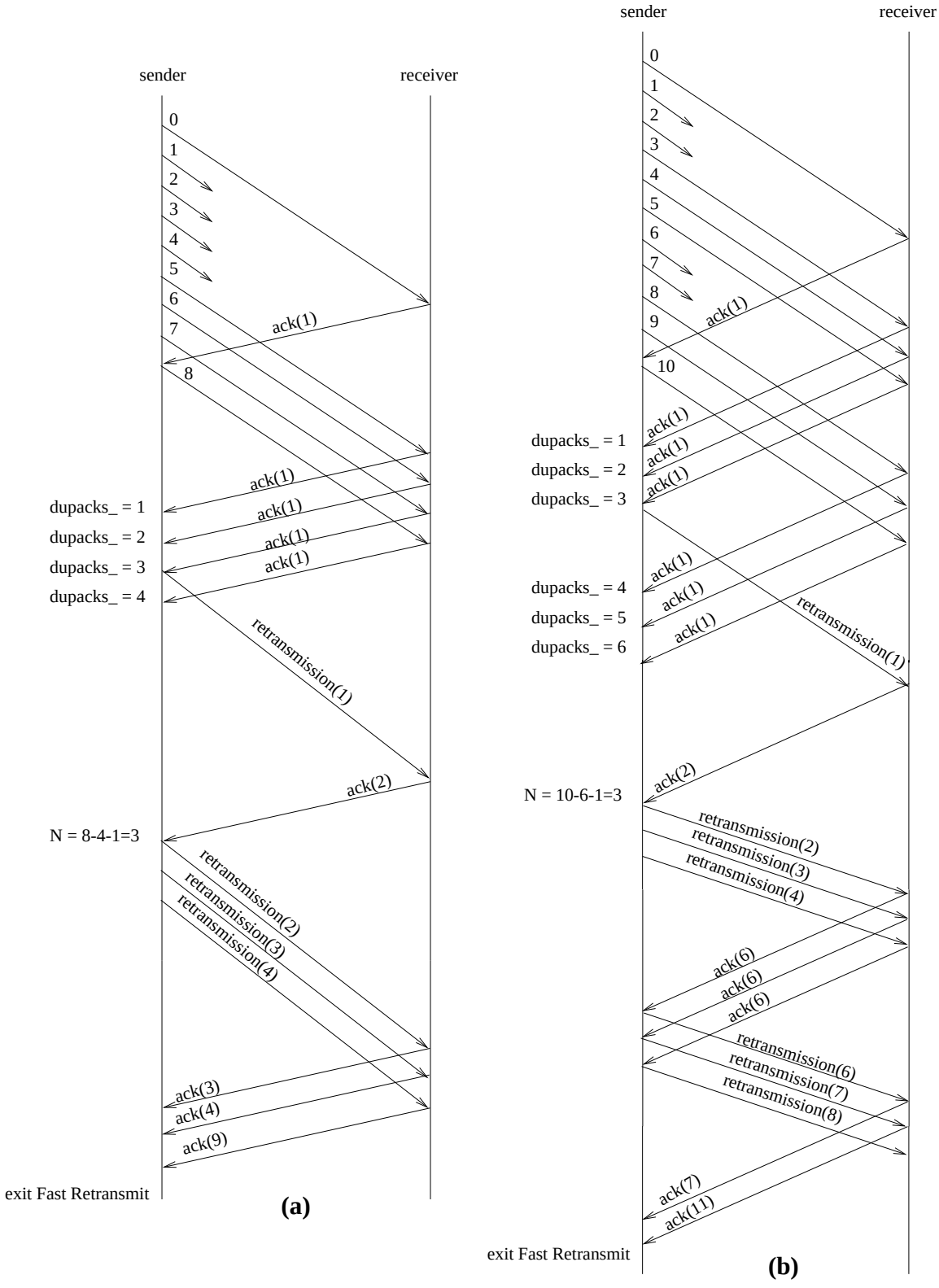


Figure 14: Algorithm execution for one (a) and two (b) holes in the packet sequence.

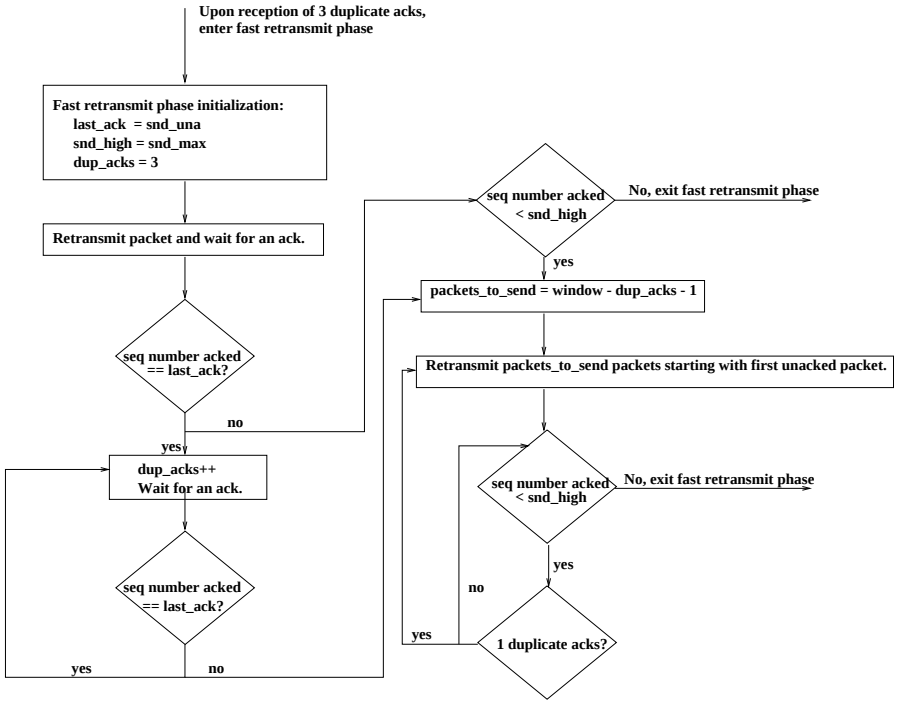


Figure 15: The proposed algorithm.

continues until the first non-partial ACK arrives and the fast-retransmit phase is therefore exited. This sequence of events is depicted in Figure 14(b), where the first dupACK carrying the sequence number 6 causes the retransmission of a burst of 3 data packets starting with data packet number 6, namely packets 6, 7 and 8.

A formal description of the algorithm is given in Figure 15.

If one of the packets retransmitted during the fast-retransmit phase is lost, it will be recovered either following a timeout, as in the case of losing the first packet recovered by the fast-retransmit phase (e.g. packet 1 in Figure 14(a) ) or at the cost of unnecessary retransmissions. A simple example for the second case would be the loss of retransmission(2) in Figure 14(a). In that event, retransmission(3) would trigger ack(2) and retransmission(4) would trigger a duplicate ack(2). The duplicate ACK would trigger the retransmission of a burst of 3 packets starting with packet number 2, causing an unnecessary retransmission of packets 3 and 4.

The proposed algorithm is tailored for a TCP connection over a satellite link, where the window is large and losses are caused by media errors rather than by router congestion. This is mainly because the algorithm reacts to packet losses by sending bursts of retransmissions. In a heavily

loaded network such an approach aggravates the congestion and increases the loss.

Since no congestion is possible on the satellite channel, the algorithm does not have to implement slow-start and congestion-avoidance, as explained in 4.2. Thus a window of constant size can be used. In particular, the sender does not halve the window upon exiting the fast-retransmit phase. In order to keep the algorithm simple the window is not inflated during the fast-retransmit phase, when dupACKs arrive. This may cause a burst of packets to be sent when exiting fast-retransmit, but will not create a problem on a satellite link as there exist no intermediate routers that can be overflowed by a quick burst of packets.

At this point is should be noted that the proposed algorithm can also be used with slow-start and congestion-avoidance (i.e. with variable window size instead of constant window size). The proposed algorithm does not interfere with either the slow-start or the congestion-avoidance algorithms. The only change that must be introduced to implement congestion-avoidance is to cut the window to half upon exiting the fast-retransmit phase, and to subsequently increase it back towards its original value at a rate of 1 packet each RTT.

If retransmitted packets are not lost and if the number of missing packets is correctly estimated, there will be no timeout during the fast-retransmit phase for the following reason. When there are multiple holes, the algorithm attributes all missing packets to a single burst of losses. and retransmits more packets than needed for each hole. For example, if there are two holes in the packet sequence, of sizes $n_1$ and $n_2$ respectively, then $(n_1 + n_2 - 1)$ packets are resent in each burst of retransmissions. The first burst is sent after recovering the first packet of the first hole, causing unnecessary retransmission of $n_2$ packets. In Figure 14(b) the first burst is sent after recovering packet 1, causing unnecessary retransmission of two packets (3 and 4). These unnecessary retransmissions trigger $n_2$ duplicate acknowledgments (the two duplicate ack(6)). The minimum size of a hole is one packet, so the minimum value of $n_2$ is 1. Thus, at least one dupACK will be sent by the receiver. Since the retransmission during a fast-retransmit phase is performed following a single duplicate acknowledgment, there is no shortage of acknowledgments and timeouts do not occur.

A false invocation of the fast-retransmit phase can happen if 3 or more dupACKs are received

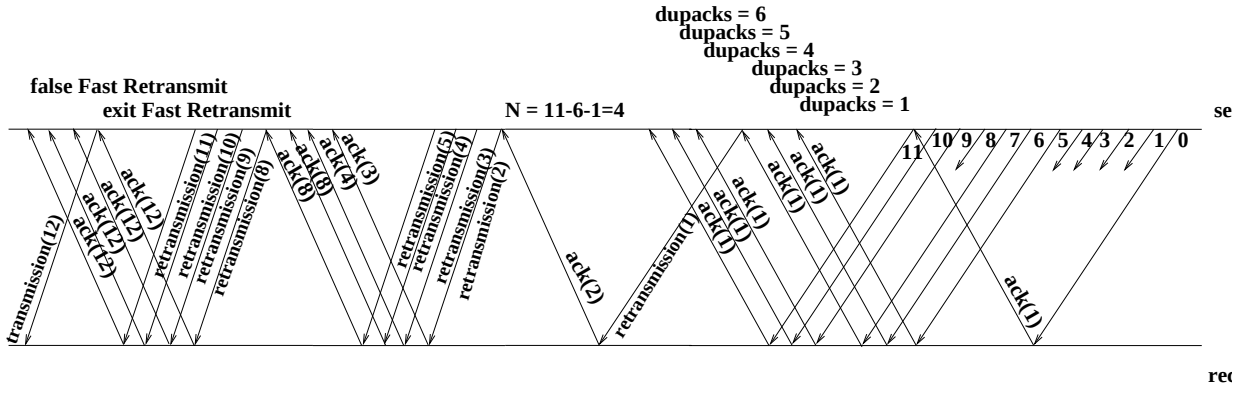


Figure 16: False invocation of fast-retransmit.

when in fact no packet has been lost. This can happen upon exiting a previous fast-retransmit phase, if the number of packets retransmitted in each burst exceeds by 3 the size of the last hole. In this case, the sender may receive 3 dupACKs for the packet beyond the highest sent before entering the fast-retransmit phase (TCP variable *snd_high* ). This packet is sent immediately after TCP exits the fast-retransmit phase. The 3 dupACKs are triggered by packets sent during the fast-retransmit phase. They will cause TCP to assume the packet was lost and falsely reenter fast-retransmit. This rare sequence of events is depicted in Figure 16. Four packets are retransmitted in each burst. For the last hole, packets number 8, 9, 10 and 11 are retransmitted. When packet 8 reaches the destination, all the packets that were outstanding at the start of the retransmission phase have been recovered, and the sender exits the fast-retransmit phase. As a result, the sender starts sending new packets, starting with sequence number 12. However, the receipt of packets 9, 10 and 11 trigger 3 dupACKs for packet 12, so a new fast-retransmit phase is falsely reentered.

### 5.4 Performance Analysis of the Proposed Algorithm

In the case of one lost burst per window, the algorithm can recover all lost packets in $2 \times RTT$ time: one RTT for the first packet and one for the rest, as shown if Figure 14(a). When there are $n$ loss bursts in one window, the recovery time is $(n+1) \times RTT$: one RTT for the first packet and one for every hole, as shown in Figure 14(b) . In any case, the recovery time depends on the number of lost bursts rather than on the number of lost packets, as in [4]. E.g., for $n$ holes of length $m$ the recovery time is $(n+1) \times RTT$ versus $n \times m \times RTT$ in the algorithm proposed in [4] or a timeout

and several RTTs for TCP Reno.

The above analysis assumes each burst of retransmissions recovers packets lost in a single burst only. However, when multiple short bursts of losses are spaced tightly together, retransmissions for one burst may in fact recover packets lost in several bursts. For example, suppose the following 6 packets are lost: 1, 2, 4, 5, 7, 8. The fast-retransmit phase is entered to recover packet 1. A partial ACK arrives for packet number 2, and 5 consecutive packets are therefore sent starting with packet number 2: these are packets 2, 3, 4, 5 and 6. Thus, packets from two holes are recovered together and the time is shorter than the bound calculated above. The algorithm pays for these savings by falsely retransmitting packets 3 and 6.

As illustrated in the previous example, the improvement in the time needed for recovery from burst losses is sometimes achieved at the expense of additional bandwidth, due to false retransmissions. Since the algorithm reacts to all losses by resending a burst of packets, if there is more than one burst of lost packets in the same window, there will be some unnecessary retransmissions, such as those of packets 3 and 6 in the example above. The worst case scenario occurs when there are multiple single-packet losses in the same window. The upper bound on the number of unnecessary retransmissions in the presence of $n$ single-packet losses is $(n-1)^2 - (n-1) = (n-1) \times (n-2)$, obtained by the following analysis. After retransmitting the first packet the sender deduces from the number of returning ACKs that there are $(n-1)$ missing packets and retransmits these packets in a burst. If this pattern repeats for each missing packet, $(n-1)^2$ packets are retransmitted instead of only $(n-1)$.

The performance improvement achieved when using the proposed algorithm over links where losses occur in bursts is shown in Figure 17. The burstiness of the losses is modeled by increasing the probability of a packet to be lost if the previous packet was lost, by a factor called "loss burstiness factor", up to some maximum threshold.

The graph shows that the proposed algorithm achieves substantial throughput improvements in the presence of burst losses. The proposed algorithm performs 3 times better than TCP Reno and about 2.8 times better than TCP new-Reno. It even performs better than TCP SACK by about 50%. The dramatical improvement is mainly due to the fact that the proposed algorithm does not

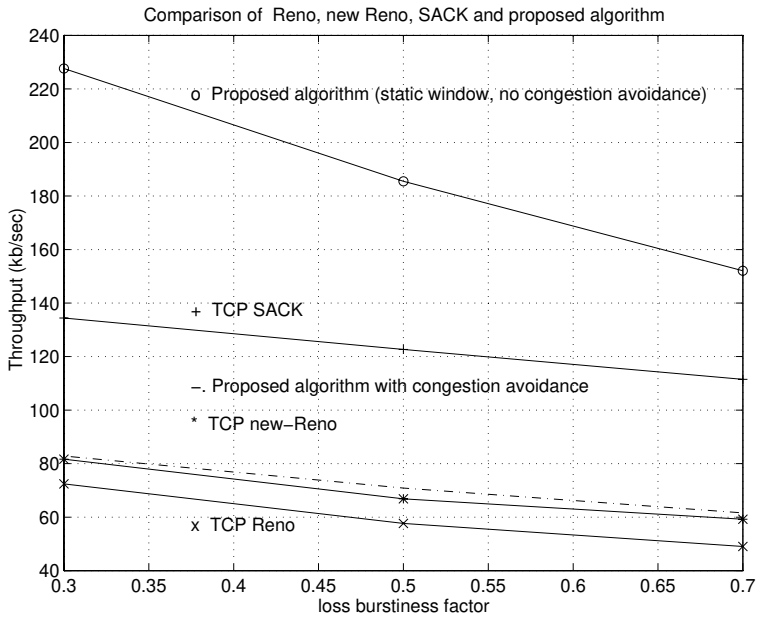


Figure 17: Throughput over links with various loss burstiness factors.

implement congestion-avoidance.

When congestion-avoidance is implemented by the proposed algorithm, the improvement is less substantial: a 25% improvement over Reno for a loss burstiness factor of 0.7, 22% for 0.5 and 14% for 0.3. When congestion-avoidance is implemented, the algorithm performs well below SACK, since SACK has complete knowledge regarding the packets that have arrived at the destination. The algorithm also performs only slightly better than TCP new-Reno under this loss model, although it shortens the recovery time for a burst of $m$ packets from $m \times RTT$ to $2 \times RTT$. This is because in this model the typical burst length is not very long. For example, for a link with a loss burstiness factor of 0.3 the probability that a 4 packet loss burst will follow the loss of a packet is less than 1% ($0.3^4 = 0.0081$).

To see the advantage of the proposed algorithm with congestion avoidance over TCP new-Reno, a link with constant probability of losing a burst of $n$ packets is used. Thus, longer bursts can be encountered and the advantage of the proposed approach, when congestion avoidance is implemented is more apparent. For example, for bursts of length 7, the proposed algorithm performs about 23% better than both TCP Reno and TCP new-Reno. For bursts of length 5 the improvement is of 36% over Reno and 16% over new-Reno. Finally, for bursts of length 3, the proposed algorithm

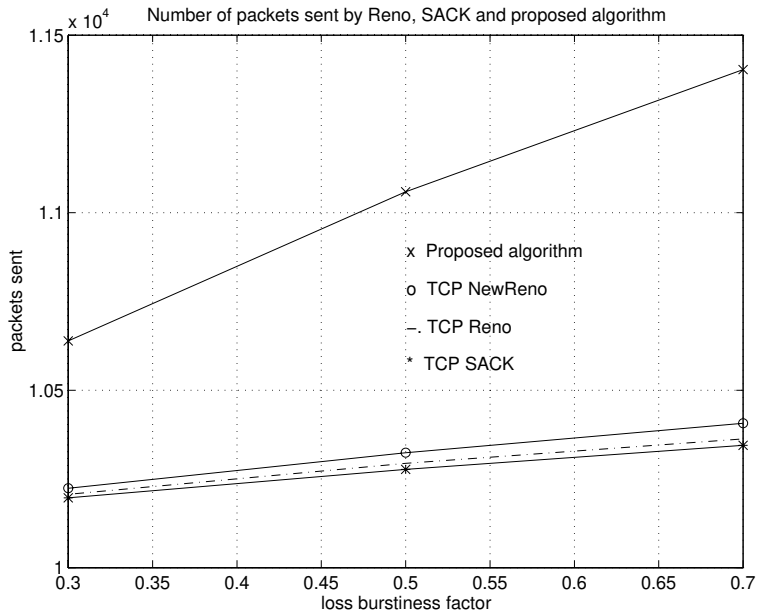


Figure 18: Bandwidth consumed by the compared algorithms.

achieves an increase of 45% over Reno and 5% over new-Reno.

Figure 18 shows a comparison of the bandwidth consumed by each algorithm during the transmission of a 10000 packet file in the presence of burst losses. The proposed algorithm uses less than 10% more bandwidth than TCP-Reno, and TCP-SACK achieves the best results.

# 6 Conclusions

In this paper, we have studied the performance of TCP in a network configuration where home users are connected to the Internet through a satellite channel. Using simulations, we have shown that a direct TCP connection is very sensitive to packet loss.

We have presented several methods for improving the performance of TCP in the considered configuration. All these methods heavily rely on the basic principle of splitting the TCP connection at the uplink, and creating two separate connections: one over the satellite channel and the other over the Internet. The implementation of the TCP protocol over the satellite channel can be tailored to best suit the characteristics of the underlying link. Due to the fact that no intermediate routers are present on the satellite link, the satellite channel is associated with two unique properties: (1) out of order routing of packets is not possible, and (2) congestion is not possible. Taking advantage

of these two properties, we have proposed two modifications to the TCP implementation: (1) using a smaller number of duplicate acknowledgments to trigger fast-retransmit and (2) using a window of constant size throughout the protocol operation.

We have also introduced a new sender algorithm, designed specifically for the satellite uplink node, in the split TCP configuration, which can recover losses occuring in a burst. The new algorithm estimates the number of lost packets and retransmits them before receiving explicit notification of their loss. The proposed algorithm achieves a threefold improvement in TCP performance and requires only changes to the TCP implementation at the satellite uplink node, leaving the receiver and server implementations unaltered.

## References


[1] R. Cohen and Srinivas Ramanathan. Tuning TCP for high performance in hybrid fiber coaxial broadband access networks. *IEEE/ACM Transactions on Networking*, 6(1):15–29, February 1998.

[2] R. Cohen and Srinivas Ramanathan. Using proxies to enhance TCP performance over hybrid fiber coaxial networks. *Computer Communication*, 20(16):1502–1518, January 1998.

[3] K. Fall and S. Floyd. Comparison of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[4] J. Hoe. Improving the start-up behaviour of a congestion control scheme for TCP. In *SIGCOMM*, 1996.

[5] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[6] S. McCanne and S. Floyd. Ns (network simulator). $http://www-nrg.ee.lbl.gov/ns$.

[7] J. Mahdavi S. Floyd, M. Mathis and A. Romanow. TCP selective acknowledgements options. RFC-2018, April 1996.

[8] W. Stevens. *TCP/IP Illustrated, volume 2*. Addison-Wesley, first edition, 1994.

[9] W. Stevens. TCP slow start, congestion avoidance, fast retransmit and fast recovery. RFC-2001, January 1997.