Experimentations with TCP Selective Acknowledgment

Renaud Bruyeron, Bruno Hemon, Lixia Zhang

UCLA Computer Science Department
{bruyeron, bruno, lixia}@cs.ucla.edu

Abstract

This paper reports our experimentation results with TCP Selective Acknowledgments (TCP-Sack), which became an Internet Proposed Standard protocol recently. To understand the performance impact of TCP-Sack deployment, in this study we examined the following issues:

- How much performance improvement TCP-Sack may bring over the current TCP implementation, TCP-Reno. We conducted experiments both on a lab testbed and over the Internet, under various conditions of link delay and losses.
- In particular, how well TCP-Sack may perform over long delay paths that include satellite links.
- What is the performance impact of TCP connections with Sack options on those connections without Sack when the two types running in parallel, if there is any.

1 Introduction

Up until now the common TCP implementation with the latest Performance tuning has been TCP-Reno, which uses *Cumulative Acknowledgments*. Recently, the TCP Selective Acknowledgment option (TCP-Sack) became an Internet Proposed Standard protocol, and is described in the RFC 2018 [8]. TCP-Sack aims at improving TCP's ability to recover from multiple losses within the same congestion control window.

With the current TCP implementation (TCP-Reno), the sender can quickly recover from an isolated packet loss using the fast retransmit and fast recovery mechanisms [13]. When multiple packets are lost, however, the sender times out in most cases and enter the *slow-start* phase ([7],[13]). To minimize false-alarms, the retransmission timeout often takes long, and when combined with the following *Slow-start* phase which reduces the congestion window size to one data segment, can be particularly damaging to TCP's throughput.

To allow incremental deployment, the two ends of a TCP connection performs a Sack negotiation during the connection setup. A TCP implementation that supports Sack appends a **SackOk** option in its SYN packet. If both ends advertise Sack option, then Sack will be used for data transmission in both Directions. If any one end does not advertise Sack, the other end shall not attempt to use Sack.

To understand how TCP-Sack works, let us look at the receiver's end first. If a packet is dropped, it creates a *hole* in the receiver's window of incoming data. When the receiver receives the data segment right *after* the hole, it sends a duplicate ACK for the segment *before* the hole, as what TCP-Reno does. In addition, TCP-Sack also adds to the ACK packet an option field that contains a pair of sequence numbers which describe the block of data that was received out-of-order. Given the maximum size of the TCP option field (40 bytes), an ACK packet can carry at most 4 Sack blocks. This means that 4 *holes* in the same congestion window can be advertised in one ACK packet. Practically, because of the increas-

ing use of the *timestamp* option described in [6], the number of Sack blocks carried by each ACK is limited to 3.

A TCP sender uses Sack options to build a table of all the correctly received data segments, thus it knows exactly which parts are missing. It can retransmit them together during a recovery phase. In TCP-Reno, a Duplicate ACK means that the receiver received data out of order and was sending ACKs for the left edge of the window. In TCP-Sack, a duplicate ACK carries the *same* information, in addition it also carries information on what other data segments that have been received out-of-order. The sender starts the data recovery mechanism after receiving 3 duplicate ACKs (as suggested in the RFC [8]).

1.1 Previous work

TCP Sack was first described in RFC 1072, and became an Internet Proposed Standard protocol in RFC 2018. In [4], Sally Floyd addressed several issues in the behavior and performance of TCP-Sack. In [3], Kevin Fall and Sally Floyd used simulation to compare the performance of TCP-Tahoe, TCP-Reno, and TCP-Sack.

Our work on TCP-Sack is an extension to the above, as we performed real experiments on our testbed and over the Internet to verify and confirm previous results.

1.2 Performance Issues

Our first goal was to measure the throughput improvement TCP-Sack may bring over TCP-Reno. We conducted test runs over both our own testbed in UCLA's Internet Research Lab and two multi-hop paths over the Internet.

Our second goal was to evaluate the benefits of TCP-Sack over long delay links such as a satellite link (typically with a round trip time of 500 msec.). In the case of long delay links, the time-out and slow-start phase proves especially harmful to TCP's throughput. Our aim was to measure TCP-Sack throughput under controlled packet losses, and compare the results with the throughput of TCP-Reno, as well as with the theoretical maximum achievable throughput.

Our last, but not the least, goal was to understand the impact of TCP-Sack on competing TCP connections without Sack. Since TCP-Sack connections achieve higher throughput than TCP-Reno connections, one concern is whether TCP-Sack makes TCP-Reno connections starve when the latter compete for bandwidth against the former.

1.3 Implementation of TCP-Sack

We used a TCP-Sack implementation by Elliot Yan of USC, which in term is ported to FreeBSD from PSC's Sack code for NetBSD [1], which itself is a port of Sally Floyd's Sack code for the NS simulator. Yan's implementation contains several flavors of TCP that can be selected for each connection at run-time using a socket option, making it easy to run TCP connections with or without Sack at the same time on the same host.

Certain coupling and interaction exist between TCP's congestion control algorithms and the Selective Acknowledgment processing and generation, which must be compliant with the specification [8]. How much of the current Congestion Algorithms should be altered to benefit from Selective Acknowledgments remains an open research question. Several Modifications to TCP's congestion control algorithm have been proposed recently ([9], [2], [1]).
The congestion control algorithm in FreeBSD implementation is described in [3]. It features the *pipe* algorithm from Sally Floyd's NS code, basically an adaptation of TCP-Reno to the new Sack capability. Therefore, it is considered fairly conservative. More aggressive congestion control algorithms were proposed, most notably *Forward Acknowledgment* in [9] (which is explicitly designed to work together with Sack) and *TCP-Vegas* ([2]). In this study, we used TCP-Sack with the *Pipe* algorithm.

[1] http://www.psc.edu/networking/tcp.html

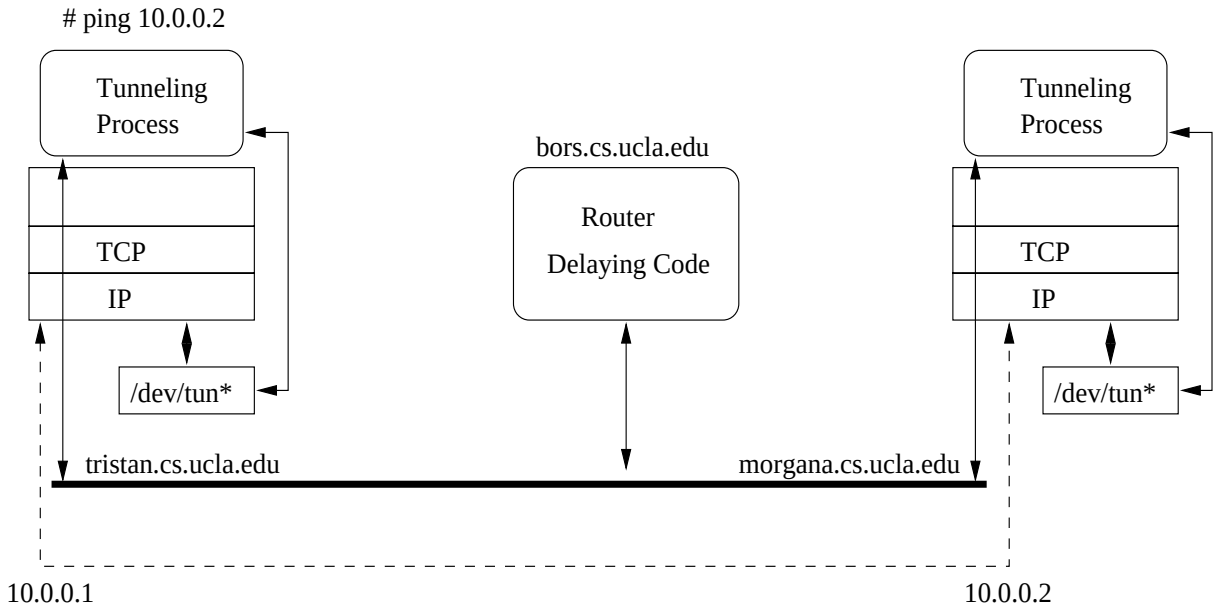

Figure 1: *Testbed – Tunneling to emulate delay*

1.4 Testbed

The testbed consisted of two PCs running FreeBSD 2.1.5 connected by a Sun UltraSparc workstation which behaves as a controllable IP router that can introduce defined link delay and packet loss patterns. To avoid modifying the Solaris kernel we implemented the above router functions in the user space on the Ultrasparc, and to make this user level manipulation transparent to the end-to-end TCP connections we used IP-in-IP tunneling. FreeBSD kernel supports a generic tunneling interface, which is used in our testbed implementation to build an IP tunnel between the Ultrasparc router and each of the two end PC's.

IP-Tunneling in FreeBSD

The tunnel device in FreeBSD is a pseudo-device that can be binded to a source and a destination address (see Fig. 2, the source address on the figure is 10.0.0.1, and the destination address is 10.0.0.2). When IP sends a packet to the destination address (*e.g* 10.0.0.2), the kernel captures the packet and hands it to the tunnel device instead of the network interface. The packets are queued in the tunnel device, and can be unqueued by a user process. Similarly, a user process can write a packet into the tunnel device. If the packet bears the source address of the tunnel (10.0.0.1 in Fig. 2), the kernel will unqueue it and move it to the IP layer, as if the packet was an incoming packet.

Delay Emulation

Fig. 1 shows how the tunneling comes into play in our testbed. Our user process listens both to the tunnel device and to a prearranged UDP port. Whatever is read from the tunnel device is sent to the router through the UDP socket (IP-in-UDP encapsulation), and whatever comes from the router through the UDP port is written to the tunnel device. The trick is invisible to the end-to-end TCP connections. The client program runs on every PCs, and can handle several tunnel interfaces, e.g. it can handle several

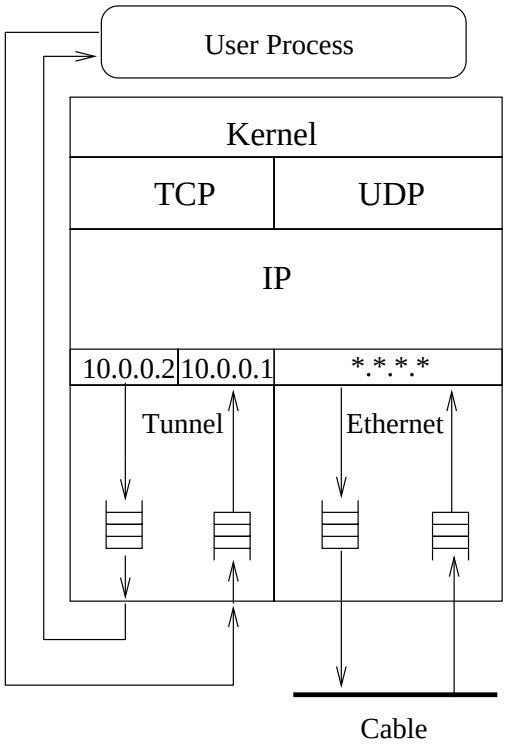

Figure 2: *Tunneling in FreeBSD*

tunnel destinations. The server program runs on the UltraSparc, and listens to a prearranged UDP port. All incoming packets are queued in a FIFO queue. The delay in the queue is defined by a command-line parameter. We added the possibility to have a separate queue for each way. We also added a leaky-bucket algorithm[2] to limit the available bandwidth.

Each end host 'sees' a virtual link between 10.0.0.1 and 10.0.0.2. The characteristics of this virtual link depend on the settings of the FIFO queue on the server process on the router, and on the settings of the user processes on both hosts. The server takes care of the delay and the bandwidth limitations, while the artificial losses were introduced in the user processes.

1.5 Measurements, instrumentation

To gather data on our experiments, we added instrumentation code in the FreeBSD kernel to plot the values of *cwnd* and *ssthresh* for a given connection. These plots are very interesting in that they show congestion events and *slow-starts*.
We also hacked LBNL's *tcpdump*[3] to make it compliant with the new specifications of TCP-Sack described in [8]. To process the output of *tcpdump*, we used both Shawn Ostermann's *tcptrace* [11] and our own AWK scripts.

1.6 Tests on the real Internet

In order to confirm the results from our testbed, we ran similar experiments on the real Internet. We had two partners for this project, the Pittsburgh Supercomputing Center and the NASA's Goddard Space Flight Center. PSC provided us with a NetBSD machine running an experimental kernel with PSC's implementation of TCP-Sack. At NASA, we used

[2] Packets entering the FIFO queue are time-stamped. The output of the queue is limited to X byte/sec by setting the time to wait before sending next packet to $\frac{P}{X}$ where P is the size of the previous packet.

[3] *tcpdump* is available at **ftp.ee.lbl.gov**. Our patch can be found at **http://irl.cs.ucla.edu/sack.html**.

a host running an experimental implementation of TCP-Sack on SunOS.

2 Comparison of TCP-Sack and TCP-Reno

In the first phase of our project, we studied the behavior of TCP-Sack and compared it to TCP-Reno. We focused on the congestion window and the throughput. We started with some tests in our lab and then moved on to the Internet to confirm our results.

2.1 Over the Testbed

2.1.1 Description of the tests

During these experiments, we studied the behavior of TCP-Sack or TCP-Reno, alone, facing the same delay, bandwidth and loss conditions. The simulation involved a FTP-like transfer of data during typically 5 minutes. Using tcpdump and our instrumentation code, we gathered data on the congestion window size and the throughput.

The link delay was 25 ms and the bandwidth was limited to 2 Mb/s. We used different packet loss probabilities. The buffer space at the sender and at the receiver was set to allow a maximum window size of 16 KB. The packets were 1400 bytes long. One advantage of using our tunnel was that the conditions were perfectly reproductible between each run.

There were two sets of experiments, with two different loss patterns: isolated drops or bursty drops.

- For isolated losses, a single packet is dropped with a given probability (between 0% and 9%).
- For bursty losses, a burst of packets is dropped in a row, with a given burst loss probability. The number of packets in the burst is constant, typically three. Therefore the packet loss probability is equal to the burst loss probability multiplied by the number of packets in the burst. In the following sections, we will refer to P as the packet loss probability, P' as the burst loss probability and b as the number of packets in a burst.

2.1.2 Isolated losses

We present in this section typical results from our tests with isolated packet losses. The following table (Fig. 3) shows the throughput of TCP-Reno or TCP-Sack when the packet loss probability is between 0% to 9%.

	TCP-Reno	TCP Sack	Sack/Reno
$P = 0\%$	184 KB/s	184 KB/s	1.00
$P = 1\%$	132 KB/s	133 KB/s	1.01
$P = 3\%$	60 KB/s	67 KB/s	1.12
$P = 9\%$	19 KB/s	19 KB/s	1.00

Figure 3: *TCP-Reno and TCP-Sack with isolated drops – separate runs of 5 minutes (50ms RTT, 2Mb/s)*

First, if there is no loss (P=0%), TCP-Reno and TCP-Sack have exactly the same behavior and have the same throughput. The congestion window opens during the initial slow-start until it reaches its maximum and then remains at its maximum. The throughput of 184 kB/s is the maximum achievable with our test parameters.

If the loss probability is low (P=1%) and if the losses are isolated, only one packet loss per window is likely to occur. In this case, TCP-Reno recovers using fast retransmit and fast recovery. TCP-Sack does not bring any improvement.

If the loss probability is P=3%, the probability of having multiple lost packets per window is greater. TCP-Sack can recover most of the time from multiple losses within the same window, while TCP-Reno very often experiences a time-out followed by a slow-start.

Fig. 4 and 5 show some plots of the congestion window size (*cwnd*) and the slow-start threshold (*ssthresh*) for TCP-Reno and TCP-Sack during an interval of 50 seconds in the P=3%, 50ms RTT and 2 Mb/s case. A slow-start event is characterized by a 'falling edge' on the congestion window plot (*cwnd* is suddenly set to 1). On Fig. 4 and Fig. 5, each of these 'falling edges' represents a slow-start (this is confirmed by the numeric data from the kernel logs). When comparing the two plots, one can notice that TCP-Sack does not show as many of these events as

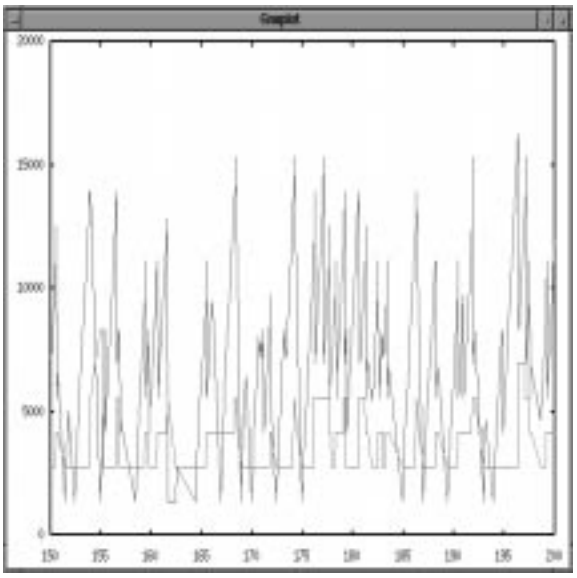

Figure 4: *TCP-Reno – Congestion window – P=3%, 50ms RTT, 2 Mb/s*

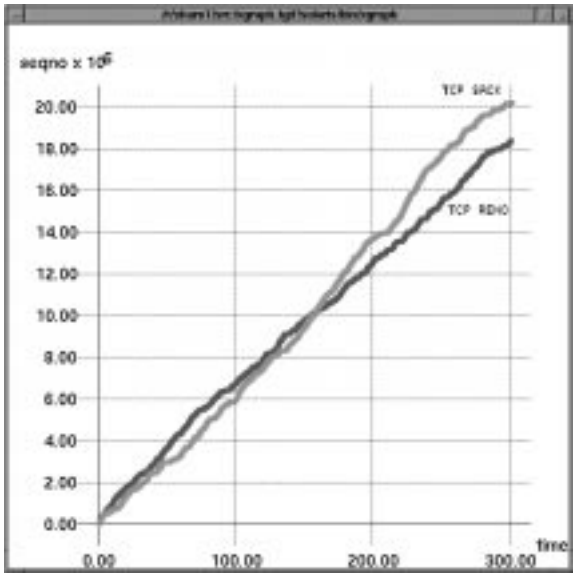

Figure 6: *Compared Throughput of TCP-Sack and TCP-Reno – P=3%, 50ms RTT, 2Mb/s*

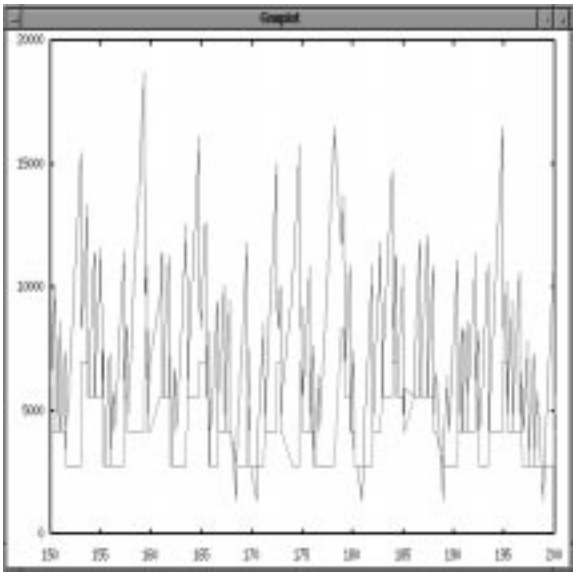

Figure 5: *TCP-Sack – Congestion window – P=3%, 50ms RTT, 2 Mb/s*

TCP-Reno (5 against 16).

As a result, the average congestion window size is larger with TCP-Sack and the throughput of TCP-Sack is better. See Fig. 6 for the throughput plot for TCP-Reno and TCP-Sack during this typical experiment. Note that even if the tests were run one after another, the plots are on the same figure. We see in this case that TCP-Sack improved the throughput by 12%.

Finally, if the loss probability is P=9%, both TCP-Reno and TCP-Sack show poor performance. The main reason is that, according to RFC 2018 [8], TCP-Sack has to time-out when a retransmitted packet is lost again. TCP-Sack will also have to time-out if a lot of acknowledgments are lost. Therefore, if the loss probability is high, retransmitted packets get dropped and TCP-Sack cannot avoid a time-out/slow-start. The result is that both TCP-Reno and TCP-Sack have a low throughput in case of high loss probability.

2.1.3 Bursty losses

When congestion occurs in the real Internet, several packets are likely to be lost. Therefore, we decided to repeat the previous experiments with 'bursty losses'. The loss distribution is very simple : a 'drop event' happen with a given probability, and at each drop event, a fixed number of packets is dropped. In the following sections, we will note P' the probability of a drop event, and b the number of packets in a burst. In this section, we have $b = 3$.

The following table (Fig. 7) shows the throughput of TCP-Reno and TCP-Sack, with $b = 3$ and P' set to 1%, 2% or 3%.

	TCP-Reno	TCP-Sack	Sack/Reno
$P' = 1\%$ $b = 3$	60 KB/s	98 KB/s	1.63
$P' = 2\%$ $b = 3$	29 KB/s	50 KB/s	1.72
$P' = 3\%$ $b = 3$	23 kB/s	24 kB/s	1.04

Figure 7: *Comparison of TCP-Reno and TCP-Sack with bursty losses*

The first obvious result is that TCP-Sack improved the throughput by 60% to 70%, when the drop event probability is 1% or 2%. In the next paragraphs, we review the results for a drop event probability of 1%. The same comments apply for a burst loss probability of 2%.

See Fig. 8 and 9 for the *cwnd* and *ssthresh* plots for TCP-Reno and TCP-Sack during an interval of 50 seconds. See Fig. 10 for the throughput plots.

We can see on the *cwnd* plots that the number of time-outs and slow-starts is higher with TCP-Reno (see Section 2.1.2 to understand what a slow-start looks like on these plots). While TCP-Reno times-out and slow-starts often and regularly (approximately 3 times every 5 seconds (Fig. 8)), we can see that TCP-Sack is able to keep its window open during 30 seconds without any slow-start (Fig. 9).

Fig. 10 shows the difference in term of throughput between TCP-Reno and TCP-Sack. The final throughput of TCP-Sack is 63% higher than the

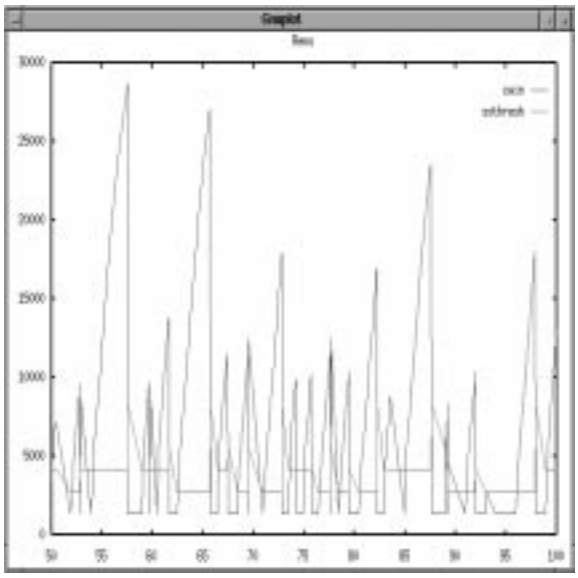

Figure 8: *cwnd and ssthresh for TCP-Reno – P'=1%, b=3, 50ms RTT, 2Mb/s*

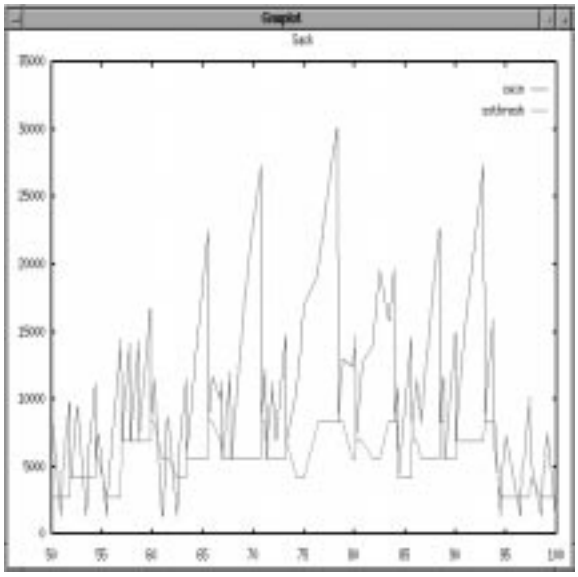

Figure 9: *cwnd and ssthresh for TCP-Sack – P'=1%, b=3, 50ms RTT, 2Mb/s*

throughput of TCP-Reno.

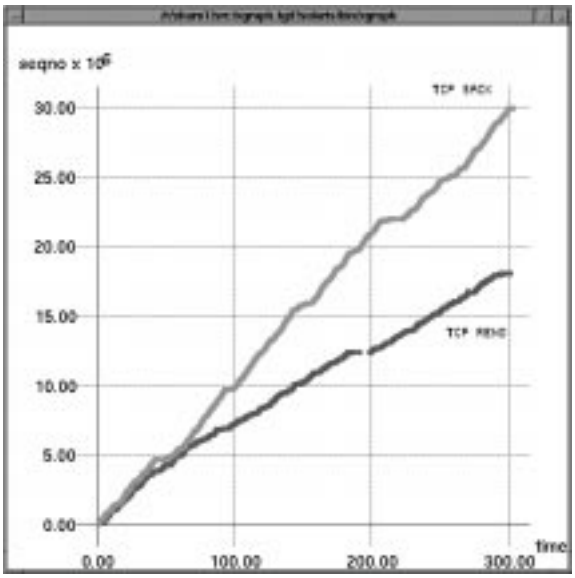

Figure 10: *Time-sequence diagram – P'=1% – b=3*

We illustrate now the different behaviors of TCP-Reno and TCP-Sack after a drop event ($b = 3$). Fig. 11 and 12 represent the packet sequence numbers and the acknowledgement sequence numbers with respect to time, for TCP-Reno and TCP-Sack respectively. Both plots represent an interval of time of 1.5 second centered on the drop event.

For TCP-Reno, we can see on Fig. 11 that the drop event occurs at $t = 122.93s$. Just after the loss, several duplicate acknowledgements are received. After the third duplicate acknowledment has been received, the first missing packet is retransmitted at $t = 123.07s$, using the fast retransmit mechanism. After the fast retransmit, if the sending window is not exhausted, more packets can be sent. We see that at $t = 123.09s$ a new packet is sent. Up to this point, TCP-Reno has fast-retransmitted one missing packet and sent new packets until the sending window is exhausted. But now, the sending window is exhausted, no more acknowledgements are received and some packets are still missing. TCP-Reno has to time-out and enter a slow-start phase. At $t = 123.91s$, the second missing packet is retransmitted. At $t = 124.07$ an acknowledgement is received. The window size is now 2 packets. The two next packets are now retransmitted. The first one was the third missing packet and the second one an unnecessary retransmission. At $t = 124.14s$, an acknowledgement for all the sent data is received. The window size is now three packets and TCP-Reno can now send new data. Recovering from this drop event involved 900ms of idle time and a slow-start.

For TCP-Sack, we can see on Fig. 12 that the drop event occurs at $t = 123.43$. After the loss, several duplicate acknowledgements are received. After the third duplicate acknowledgement, the three missing packets are retransmitted at $t = 123.55s$, using fast retransmit. After this fast retransmit, if the sending window is not exhausted, more data can be sent. In this case the sending window was exhausted. At $t = 123.63$, an acknowledgement for all the sent data is received and some new data can be sent. We see that in this case, no time-out and slow-start occured. All the data has been correctly recoverered using fast retransmit.

The two previous scenarios illustrate clearly the benefit of using TCP-Sack when the losses are bursty. We can note the interesting fact that TCP-Reno has the same throughput (60 KB/s) for isolated losses with a packet loss probability of 3% and for bursty losses with a burst loss probability of 1% and a burst size of three packets. This means that for TCP-Reno, three consecutive drops and three separate drops look and feel the same. For TCP-Sack, on the other hand, the size of the drop event does matter to the recovery mechanism.

When the burst loss probability is higher (P'=3%), then both TCP-Reno and TCP-Sack have the same poor performance. The reason is the same as in the isolated losses case. When the loss probability is too high, The window size remains small and some time-outs and slow-starts can not be avoided, even with TCP-Sack.

2.1.4 Conclusions

We can draw some conclusions from these experiments. First, it was observed that TCP-Sack is useful for a given range of packet loss probability. If there

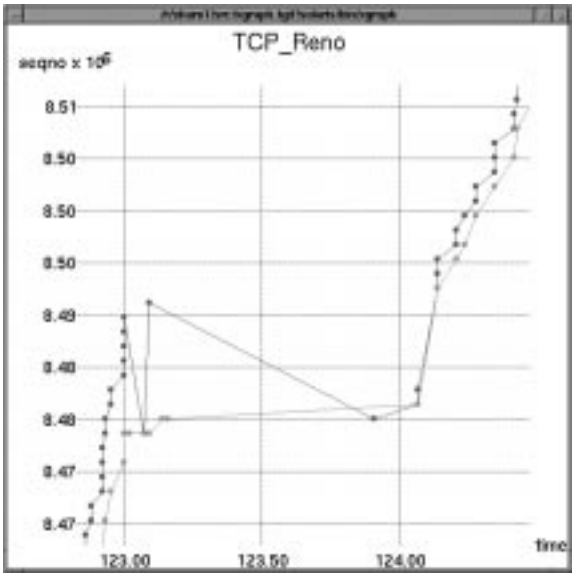

Figure 11: *Close-up on a bursty drop event – TCP-Reno – P'=1% – b=3*

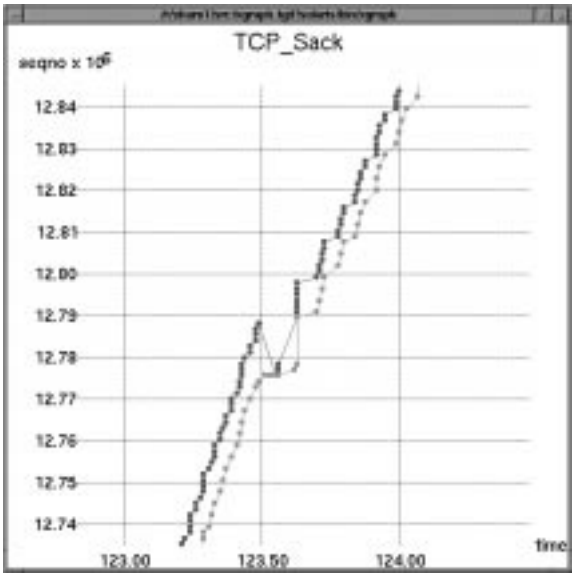

Figure 12: *Close-up on a bursty drop event – TCP-Sack – P'=1% – b=3*

is no loss or if the loss probability is very low, then TCP-Reno can recover using fast retransmit and fast recovery. If the loss probability is very high, even TCP-Sack is subject to some time-outs and slow-starts and can not keep a large window. In both cases, there is no real benefit of using TCP-Sack.

Then, it was observed that for a drop rate between 2% and 4%, TCP-Sack improved the throughput by a significant amount. This improvement is really important when the packet losses are bursty. We illustrated in a detailed analysis the difference in behavior of TCP-Reno and TCP-Sack after a loss of a burst of packets. While TCP-Reno can only recover from an isolated loss using fast retransmit, TCP-Sack can recover from a burst loss without any time out or slow-start. As a result, we observed a throughput improvement of 60% to 70% with TCP-Sack when the packet losses were bursty.

The results presented in this section are typical of the results we got during our numerous experiments.

2.2 Experiments over the Internet

There are two main reasons to try to validate our results by running experiments on a real network. First, the topology of our testbed is very simple (one hop): it is interesting and useful to know what the picture is on a multi-hop path. The second reason is that we used fixed loss distributions to get the bottom line. On a real multi-hop path, the congestion patterns are very complex and difficult to model: we want to know what the results are in this case. The main problem with real-life experiments is that the network conditions are not reproductible from one test to another.

2.2.1 Between UCLA and the Pittsburgh SuperComputing Center

In this set of experiments, we used one host in our lab at UCLA and one host at the Pittsburgh SuperComputing Center. The path is 14 hops long, with an average round-trip time of 65 ms and a bottleneck link of 6 Mb/s (Van Jacobson's pathchar to measure the path). The packet size is 496 bytes. The buffer

space at the sender and at the receiver was set up to 65535 bytes (approximately 132 packets).

One important difference with our testbed is that depending on the time of the day the traffic conditions change dramatically. Therefore, we show here three typical experiments, with typical (average) results.

For each one of our tests, we established an ftp connection between UCLA and PSC for 2 minutes, first with TCP-Reno and after with TCP-Sack. For each test three plots were generated. The first plot is the usual time-sequence diagram for TCP-Reno and TCP-Sack. Even if the two connections were run one after another, the throughputs of TCP-Reno and TCP-Sack appear on the same graph for easy comparison. The second and third plots show the congestion window size for TCP-Reno and TCP-Sack respectively.

The first experiment (Fig. 13 and 14) was performed at 12 pm (pacific time). The Internet traffic at this time is heavy. The second experiment (Fig. 15 and 16) was performed at 4 pm (pacific time) when the traffic is 'average'. The third test (Fig. 17 and 18) was performed at 8 pm (pacific time) when the Internet traffic is light.

The first interesting thing to notice is the behavior of the congestion window (Fig. 14, 16 and 18). Every time a time-out followed by a slow-start occurs, the congestion window is reduced to 1 (falling edge on the diagram). We can estimate the number of time-outs and slow-starts by counting the number of falling edges. For example, we can see on Fig. 14 that TCP-Reno timed-out approximately 50 times in 2 minutes while TCP-Sack timed-out only twice. In all our test, TCP-Reno timed-out more often than TCP-Sack.

Since TCP-Sack is able to avoid some time-outs and slow-starts, it can keep a larger average congestion window. It is especially true when the Internet traffic is heavy. We can see on Fig. 14 and 16 that TCP-Sack timed-out respectively 2 and 3 times in 2 minutes, while TCP-Reno timed-out 50 and 25 times. The average congestion window is therefore larger with TCP-Sack. As a result, the throughput of TCP-Sack is higher than the throughput of TCP-Reno. When the Internet traffic is light, TCP-Reno can recover from congestion using fast retransmit and fast recovery and there is less benefit using TCP-Sack. Looking at the tcpdump traces of our tests, we were able to figure out what the loss pattern is during the experiments. As we expected, the losses are very bursty. We observed that the average burst length is 20 to 30 packets. The maximum window size is set to 65535 bytes and the packets are 496 bytes long. Therefore, the maximum window size is set to approximately 132 packets. This very bursty loss pattern explains why TCP-Sack is more efficient than TCP-Reno.

The following table (Fig. 19) shows the throughput of TCP-Reno and TCP-Sack at these three typical times of the day.

Figure 19: *UCLA to PSC – Throughput comparison*

	Fig. 13 & 14	Fig. 15 & 16	Fig. 17 & 18
Reno	63 KB/s	104 KB/s	221 KB/s
Sack	81 KB/s	132 KB/s	257 KB/s
Sack/Reno	1.29	1.27	1.16

The first comment is that there is more benefit using TCP-Sack when the Internet traffic is average or heavy. This was expected since TCP-Reno is able to recover from one single isolated loss. In this case, TCP-Sack does not do better than TCP-Reno. We found a throughput improvement of 15% with TCP-Sack during a period of light Internet traffic.

We found a throughput improvement of approximately 30% with TCP-Sack during periods of average or heavy Internet traffic. We think that such an improvement of 30% is significant.

There are two interesting conclusions to these experiments. First, the bursty loss pattern observed during our tests over the real Internet was what we expected. The behavior of both TCP-Reno and TCP-Sack in the face of congestion was also what we expected. While TCP-Reno had to time-out and slow-start after congestion, TCP-Sack was able to recover smoothly. The second interesting point is that we observed a throughput improvement of 30% with TCP-Sack when facing heavy or average Internet traffic.

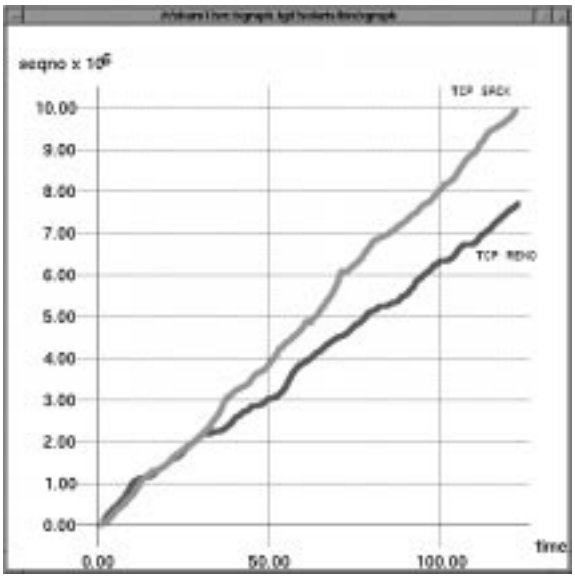

Figure 13: *UCLA to PSC – Heavy Traffic Conditions (12 pm p.t.) – Time-sequence*

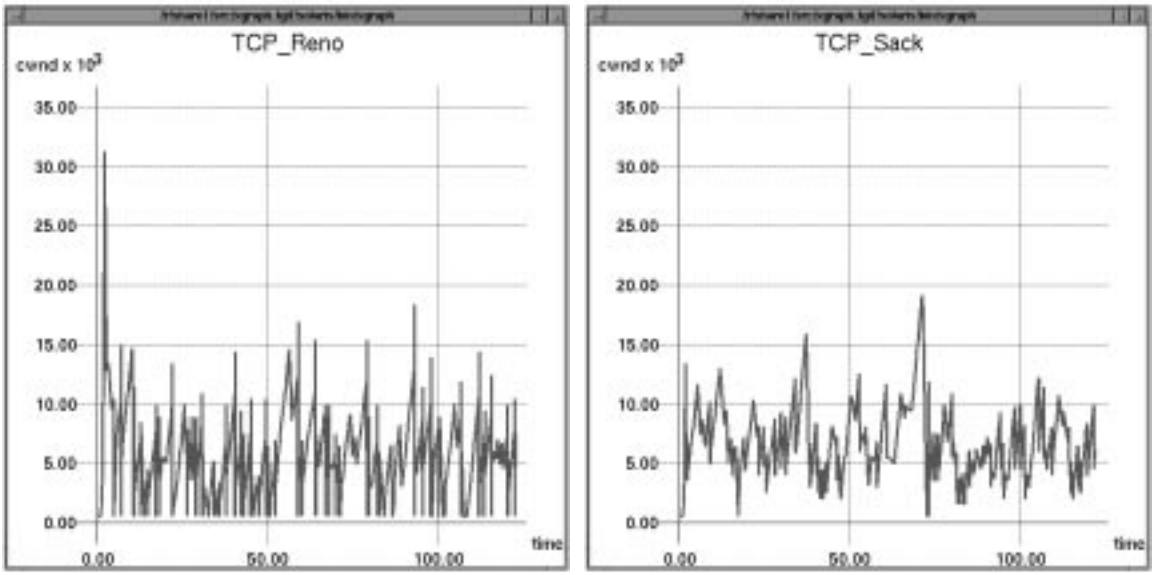

Figure 14: *UCLA to PSC – Heavy Traffic Conditions (12 pm p.t.) – cwnd*

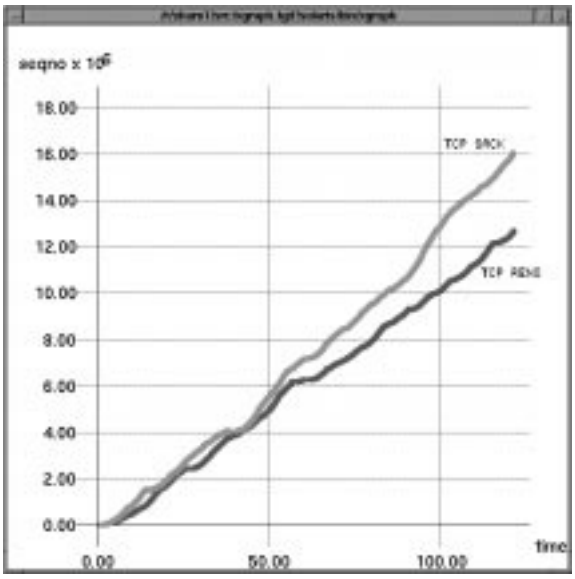

Figure 15: *UCLA to PSC – Average Traffic Conditions (4 pm p.t.) – Time-sequence diagram*

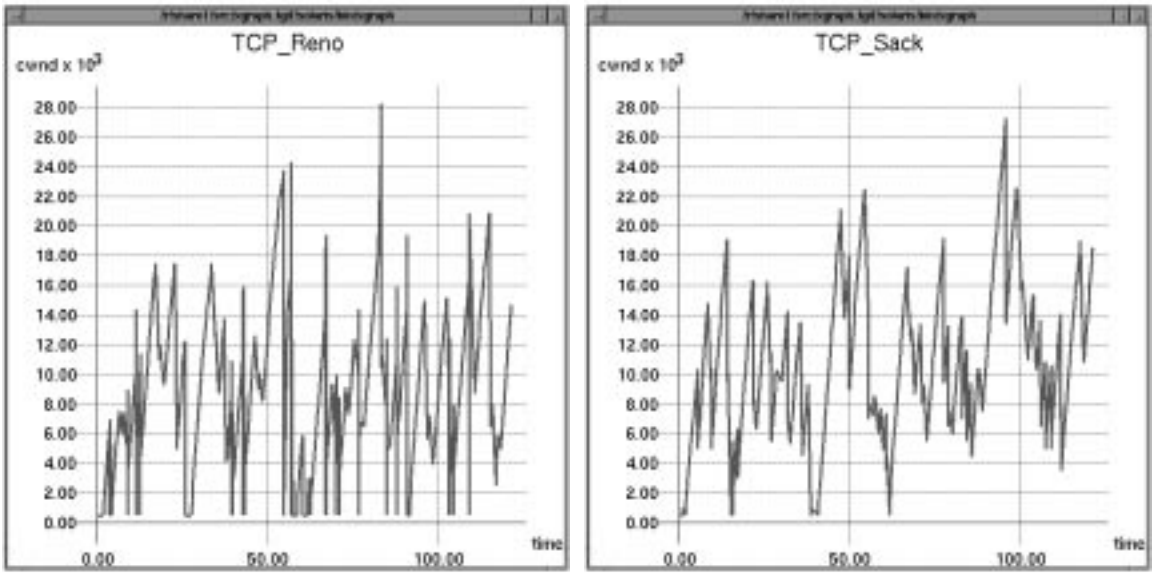

Figure 16: *UCLA to PSC – Average Traffic Conditions (4 pm p.t.) – cwnd*

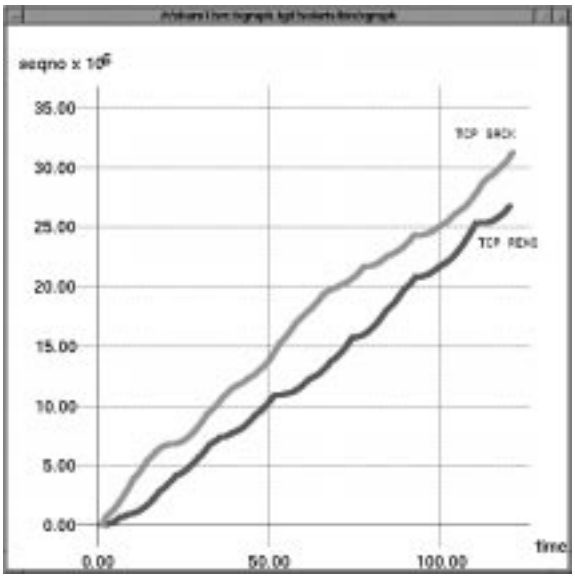

Figure 17: *UCLA to PSC – Light Traffic Conditions (8 pm p.t.) – Time-sequence diagram*

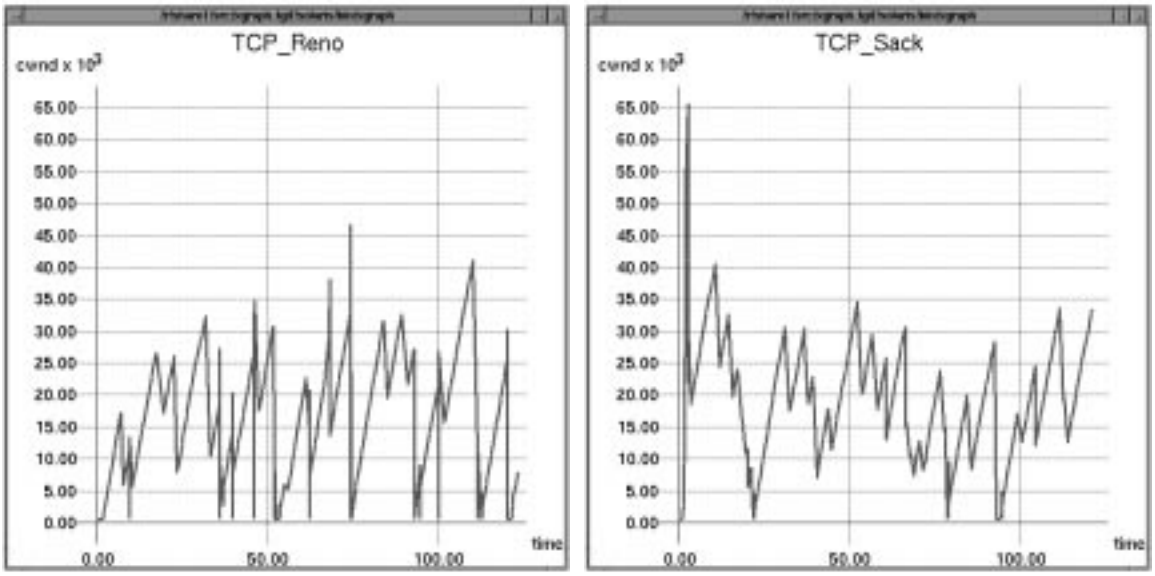

Figure 18: *UCLA to PSC – Light Traffic Conditions (8 pm p.t.) – cwnd*

2.2.2 Between UCLA and the NASA's Goddard Space Flight Center

We tried to run the same experiments between UCLA and NASA's Goddard Space Flight Center. Due to technical problems with the experimental SunOS kernel we used at GSFC, we did not apply the same methodology and could not run as many tests between UCLA and the NASA lab as in the previous section. Here, we show some typical results.

The path between the two hosts is 16 hop long. Using pathchar, we measured a round-trip time of 80 ms and a bottleneck of 8.4 Mb/s. We could not use our own traffic generator for these experiments and we used instead a regular FTP program to transfer one big file. The file was 29 MB long and the duration of the transfer was between 2 and 3 minutes.

See Fig. 20 and 21 for the results of one typical test. Fig. 20 shows the time-sequence diagram of TCP-Reno and TCP-Sack. The two tests were run one after the other but we show both results on the same plot. Fig. 21 is a table which compares the throughput of TCP-Reno and TCP-Sack.

Because of technical problems, we could not use the instrumentation code to plot *cwnd* and *ssthresh*. Still, it is possible to see on the time-sequence diagrams that more time-outs and slow-starts occur with TCP-Reno. A time-out is an accident in the time-sequence diagram, like a bump (see the close-up on Fig. 20). The number of time-outs and slow-starts with TCP-Sack is smaller, and the transfer appears to be much smoother and steadier.

We found a throughput improvement of 46% when using TCP-Sack. This improvement is a typical value that was confirmed by our other tests. Almost 50% of improvement for an FTP transfer is very significant.

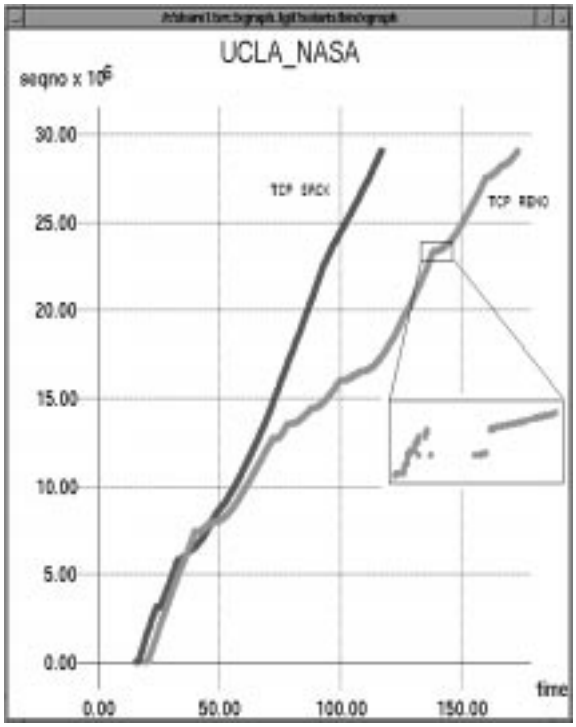

Figure 20: *NASA to UCLA – 29 MB FTP – close-up on a time-out event*

2.2.3 Conclusion

Thanks to our two partners, the Pittsburgh SuperComputing Center and NASA's Goddard Space Flight Center, we have been able to compare the performance of TCP-Reno and TCP-Sack over a real path on Internet. All our tests confirmed the same conclusions. We first observed that the real loss patterns are bursty as we suspected. Then we had confirmation that TCP-Sack is able to avoid most time-outs and slow-starts, keeping a larger congestion window than TCP-Reno. Finally we observed some important throughput improvement with TCP-Sack, especially when facing heavy traffic.

We found that TCP-Sack could improve the throughput by 30% to 45% over two different multi-hop paths over the real Internet. What is interesting is that these experiments were done in real conditions. We believe that a such a big throughput improvement meets the expectations placed on TCP-Sack.

Our last conclusion is that these experiments confirm the results on our testbed.

Figure 21: *NASA to UCLA – Throughput comparison*

	Fig. 20
Reno	183 kB/s
Sack	268 kB/s
Sack/Reno	1.46

3 Long delay links

One goal of our project was to study the behavior of TCP-Sack over long delay links, when used together with the Window Scale option defined in [6].
Our focus was mainly on the 500ms range, since this is a typical value for a Geosynchronous satellite. We ran several experiments with different loss rates and different scenarios.
We set the buffer sizes to 256 KB on both sides of the virtual link, to take advantage of the large $delay - bandwidth$ product. The bandwidth setting varied between 2 Mb/s and full 10BT bandwidth (10 Mb/s). Note that when limiting the bandwidth using the leaky bucket algorithm, the average delay is impacted by the modified behavior of the FIFO queue in the delaying server (see Section 1.4).

3.1 Preliminary results

Using the full bandwidth available, and with a long RTT, we are able to generate some congestion on the virtual link. Indeed, with buffers set to 256 KB on both PCs, with a rate of 10 Mb/s, the link can get congested in a realistic way (several drops per window, bursty drops, etc). The congestion is caused by buffer overflows on the router (the maximum UDP buffer size on Solaris is limited to 64 KB, hence this problem). The delay was 80ms, and Sack does 100% better than Reno. The time-sequence diagram can be seen in Fig. 22, and the *cwin* and *ssthresh* plots can be seen in Fig. 23.
This experiment gives the bottom line for further refinement of the experiments. The striking difference between the congestion behavior of Reno and Sack is very obvious when looking at the time-*cwin* diagrams. In this case (80ms, full 10BT bandwidth, congestion loss), Sack is able to avoid almost all the $slow - starts$, hence the huge throughput increase.

3.2 Artificial losses

In this experiment, we set the round trip time to 500ms, and we limited the bandwidth to 2 Mb/s. All the drops are artificially introduced. The drop rate is $P' = 0.18\%$ with $b = 3$. This case is extremely favorable to TCP-Sack. The time-sequence diagram is at Fig. 25.

Figure 24: *0.15% drop rate, burstiness 3*

	Throughput
TCP-Reno	32 KB/s
TCP-Sack	55 KB/s

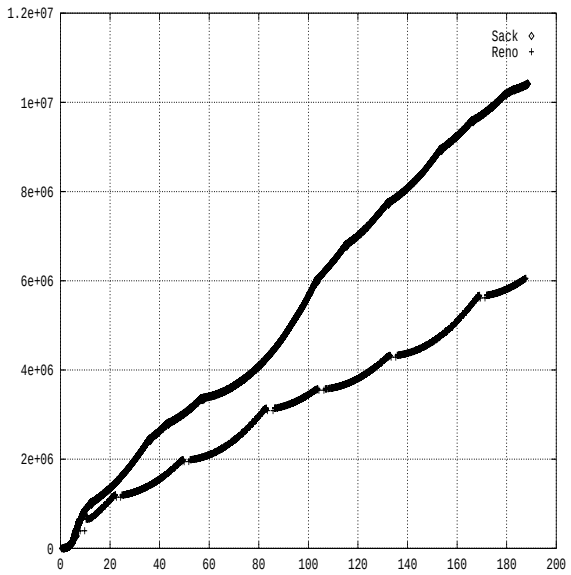

Figure 25: *Time-Sequence diagram — Sack vs. Reno — Link: 500ms RTT, 2 Mb/s, 0.18% drop rate, burstiness 3*

3.3 Sack and Reno vs. Theoretical bandwidth

Next, we focused on the link utilization. In [10], Mathis et al. defined a macroscopic behavior of

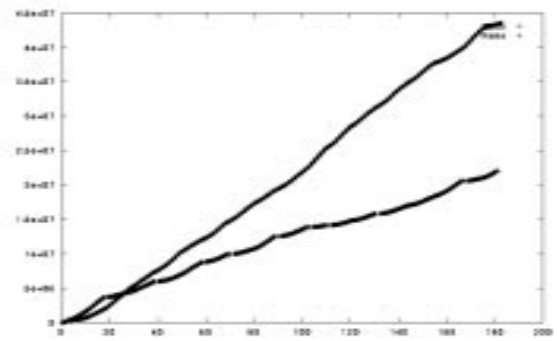

Figure 22: *Preliminary test: Sack & Reno with Congestion-like type of loss*

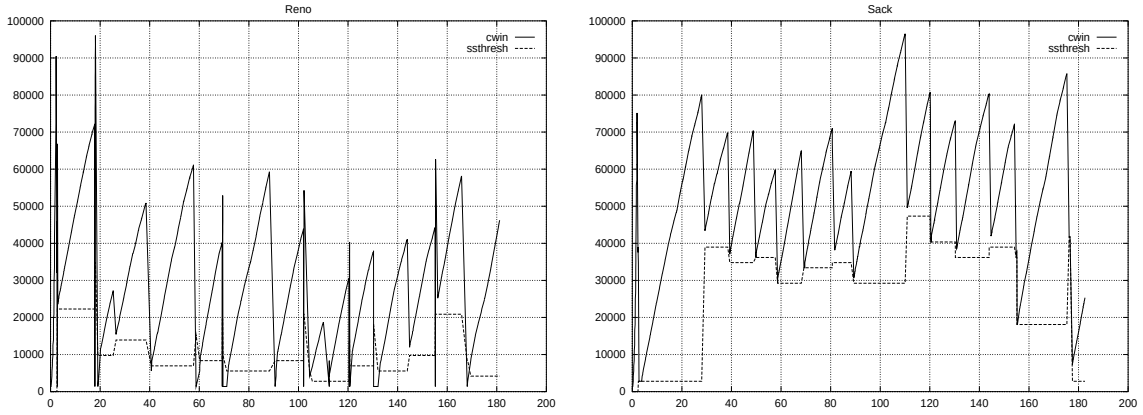

Figure 23: *Preliminary test: time-cwin and time-ssthresh diagrams*

the congestion avoidance algorithm. It starts with the hypothesis that the current congestion algorithm (halving the congestion window when there is a congestion signal, *e.g* a drop) is necessary in any TCP implementation. Given this hypothesis, the model gives an ideal throughput that can be achieved under a given (moderate) drop rate, and a given delay. The assumption behind the formula is that the connection does not experience any time-outs, and that the effects of the start-up phase are diluted over a very long connection. The formula for the achievable throughput is

$$BW = \frac{MSS * C}{RTT\sqrt{p}} \quad (1)$$

where MSS is the segment size, RTT is the round trip time, and p is the drop rate. C is a constant that depends on the ACK strategy and several other

factors. As one can see in this equation, the throughput achievable is proportional to the inverse of RTT. That is, the longer the delay, the smaller the throughput. Why ? Isn't the *wscale* option supposed to address the issue of long-delay ?
The reason is that with long delay, the linear expansion of a large *cwin* is very long. Therefore, reaching the optimum *cwin* (*i.e* the optimum throughput) takes a long time. This is why one can expect the link utilization to be low for long delay.

We compared the performance of TCP-Sack and TCP-Reno against the model, and the results back the conclusion of [10] that TCP-Sack is indeed closer to the ideal congestion behavior. We also focused on the throughput increase brought by TCP-Sack when the burstiness of the drops is higher than one.
In this experiment, we set the link propagation delay to 250ms (RTT=500ms). We used large windows (256 KB of available queue length) and artificial losses introduced at the end points. For the model, we took $C = \sqrt{\frac{3}{2}}$ since the ACKs in our implementation are not delayed (see [10]).

Drop Rates	Sack	Reno
0.05	73 %	69 %
0.1	87 %	77 %
0.15	88 %	70.5 %
0.18	82 %	66.5 %
0.2	76 %	62 %
0.3	81 %	73 %
0.4	90 %	66 %
0.5	85 %	68 %
0.8	86 %	68 %
1	87.5 %	64.5 %

Figure 26: *Sack & Reno vs. Theoretical Throughput*

In Fig. 26, we show the performance of Sack and Reno compared to the throughput according to the model from [10]. 100% would mean that TCP achieved the best possible throughput using the Congestion Algorithm. The reason why Sack and Reno do not achieve 100 % is because of time-outs and because of the start-up phase. TCP-Sack is closer to the model because there are less time-outs. The plot of this table can be seen in Fig. 28.

Drop Rates	Sack	Reno
0.05	37 %	34 %
0.1	28.5 %	25 %
0.15	25.7 %	20.5 %
0.18	22 %	17.7 %
0.2	19 %	15.6 %
0.3	16.7 %	15 %
0.4	16 %	11.7 %
0.5	13.5 %	10.7 %
0.8	10.7 %	8.5 %
1	9.7 %	7 %

Figure 27: *Link Utilization for Sack and Reno*

In Fig. 27, one can see the link utilization achieved by TCP-Sack and TCP-Reno under the same conditions. It of course depends strongly on the state of the link. It is very low. The reason is because every time a drop occur, *cwin* is divided by 2. Therefore, at this point, the throughput is divided by 2. Then, linearly expanding *cwin* back to a large value takes a long time because of the delay, and because of the large size of the window (see sec. 3.4).

The key to the better behavior of TCP-Sack is its ability to avoid slow-starts. Slow-starts are especially damaging to the throughput over long-delay links. To time-out, TCP-Sack must face heavy congestion: loose a string of ACKs, or loose a retransmitted packet.

3.4 Issue of Initial *ssthresh* setting

When we first had a look at Fig. 27, we did not understand the shape of the plot 'Throughput Improvement of Sack over Reno', and especially why the plot for *burstiness* = 3 has this shape: why is the improvement of Sack maximum for $p = 0.4\%$, and why is it so low for small drop rates ?

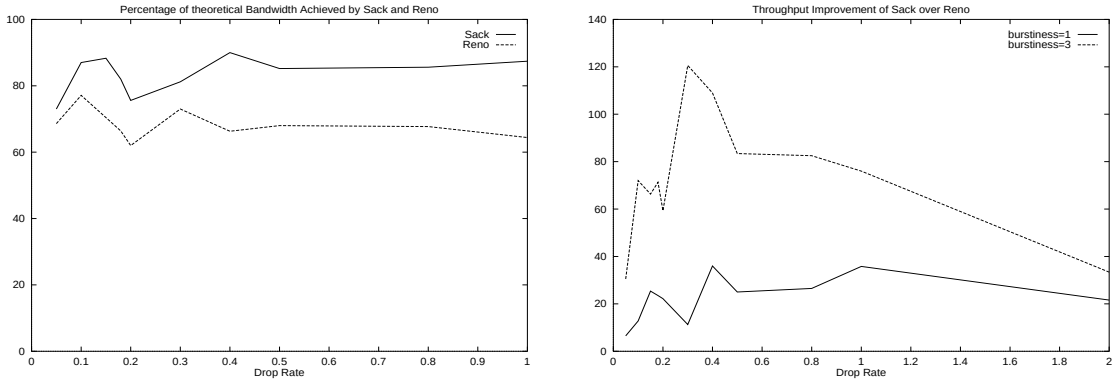

Figure 28: *TCP-Sack and TCP-Reno against Congestion Avoidance Model – on the left, how close to the model TCP-Sack and TCP-Reno are – on the right, performance improvement of TCP-Sack over TCP-Reno, as a function of P' and b*

The culprit seems to be the default initial setting of *ssthresh* when a TCP connection is started.
In most implementations of TCP, *ssthresh* is initially set to the maximum window size. If the *wscale* option is used, the maximum possible window size is 4 GB. This initial value lets TCP start with a *slow-start* that can be interrupted only by congestion signals, at which point *ssthresh* is set to a more reasonable value (*cwin*/2 when the congestion signal occurs).
Unfortunately, if several congestion signals occur, *cwin* is divided by two several times. If the congestion signal is very bursty, then *ssthresh* is likely to be set to a very small value (a few packets).
This is extremely costly in the case of long-delay (linear increase of *cwin* takes a long time) and large windows. The large delay and high bandwidth cause the *slow-start* phase to shoot up very high, and then the connection usually experiences a burst of loss, which resets *ssthresh* to a very low value (sometimes a few packets).

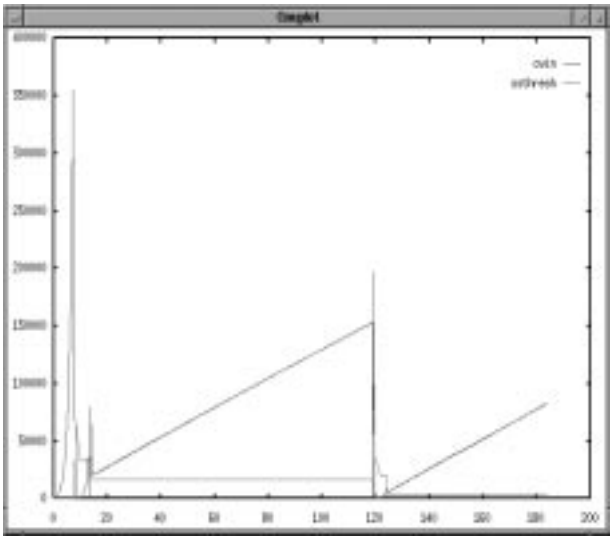

Figure 29: *Effect of high initial setting of ssthresh on cwin expansion over long-delay with large window – cwnd collapses to a very low value after 8s because of a slow-start phase that lasts too long*

To illustrate this effect, we show in Fig. 29 a plot of *cwin*/*ssthresh* for a connection over the virtual link with the following settings: bandwidth = 5Mbps, RTT = 500ms, buffers = 512KB, no artificial losses. The drops are caused by congestion at the UDP inter-

face in the router (the UDP buffers on the UltraSparc were set to 64KB, the maximum value allowed by Solaris).
On this plot, we see the *slow-start* phase shoots up very high, then we see a burst of drops, and *ssthresh* collapses to a low value (around 15KB). We can see that the equilibrium for the window size seems to be around 130KB, and it takes about 85s for *cwin* to expand to this value. Therefore, the start-up behavior of TCP in this case resulted in a poor estimate of *ssthresh*.

This affects both TCP-Sack and TCP-Reno, hence the lower improvements of Sack over Reno with small drop rates. Why does it affect the results with small drop rate ? Because with higher drop rate, an artificial drop is more likely to occur during *slow-start* and interrupts it before it shoots too high and generates a burst of congestion drops. With higher drop rates, the *slow-start* phase results in a better estimate of *ssthresh*.
This problem is described in [5], and several solutions are being reviewed.

3.5 Conclusions over long delay performance

The key to high performance when facing high delay-bandwidth product links is the use of the *wscale* option. Being able to "fill the pipe" during the transfer is what really matters.
Yet, the long delay makes the *slow-start* algorithm extremely costly, since it takes several RTT to get the ack-based clock running again. What TCP-Sack does much better than TCP-Reno is recover from multiple loss within one window of data. TCP-Reno needs one RTT per packet drop to recover, while TCP-Sack is able to recover from several losses in one single RTT: the most obvious benefit, as one can see in Fig. 23, is that TCP-Sack avoids some of the *slow-starts*.

4 Negative impact of TCP-Sack on TCP-Reno

TCP-Sack is now a proposed Internet standard and is soon to be deployed over the Internet. This deployment will be incremental. TCP-Sack is expected to be in most cases more efficient than TCP-Reno and one key issue is to know what will be the behavior of TCP-Reno when it is competing against the more efficient TCP-Sack. It would not be fair to have all the bandwith taken by TCP-Sack, while TCP-Reno connections are starving. The purpose of the last part of our experiments is to study the impact of TCP-Sack on TCP-Reno.

In the following tests, we are not interested in the benefit of TCP-Sack but in knowing what happens to a TCP-Reno connection when it is competing against a TCP-Sack connection. One typical test features 2 ftp connections at the same time, one with TCP-Sack and one with TCP-Reno. Then the same experiment is performed with 2 ftp connections both using TCP-Reno. The point is to compare the performance of the lone TCP-Reno of the first test with the two TCP-Reno in the second run.

4.1 Experiments with a few flows

4.1.1 First experiments

This experiment is over our testbed. First, 2 TCP-Reno connections share the link for 5 minutes. Then we run in the same conditions 1 TCP-Reno and 1 TCP-Sack connections at the same time.

The link delay is set to 20 ms, the burst loss probability is 1% and the burst size 3 packets long ($b = 3$). We saw earlier that TCP-Sack significantly outperforms TCP-Reno under these conditions. The question is: would the throughput of TCP-Reno drop when it is competing against TCP-Sack ?

See Fig. 30 for the time-sequence plots of these 2 tests.

See the following table (Fig. 31) for a summary of the throughput of each connection.

In the first test, the 2 TCP-Reno connections have approximately the same throughput (10% difference). We will refer to the one with the highest throughput

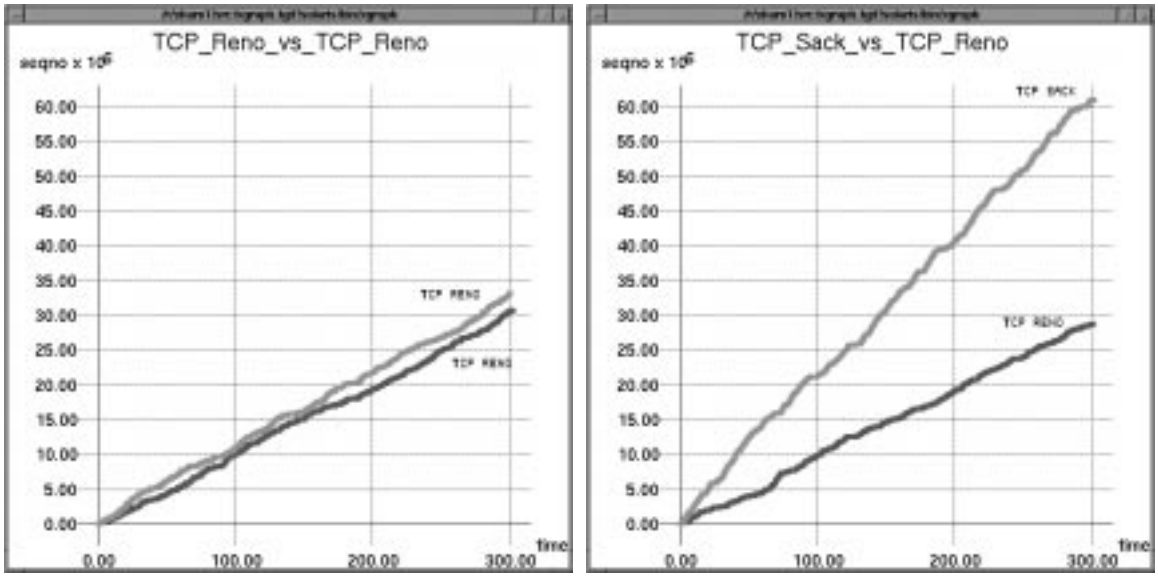

Figure 30: *link delay=20 ms, burst loss probability=1%, burst length=3 – Comparison of TCP-Sack vs. TCP-Reno and TCP-Reno vs. TCP-Reno*

Figure 31: *Throughput table for Fig. 30*

TCP-Reno vs. TCP-Reno	TCP-Sack vs. TCP-Reno
101 kB/s (Reno)	95.5 kB/s (Reno)
110 kB/s (Reno)	202 kB/s (Sack)

as the best TCP-Reno. The difference may seem significant between two identical algorithm, but it is actually a constant feature of long tests (the odds of an unlucky succession a drop events are greater). What is important is that the slope of the two plots are equal on average.

We can first notice that the throughput of TCP-Sack in the second test is really higher than the throughput of the best TCP-Reno in the first test. The throughput improvement with TCP-Sack is roughly 84%.

But if we compare the throughput of the TCP-Reno with the lowest throughput in the first test and the throughput of TCP-Reno in the second test, we can see that this throughput decreased by only 5.4%. As we said earlier, a 10% difference between two identical algorithms is not significant. Therefore a 5.4% difference here is not significant.

If we compare the aggregated throughput of the 2 connections in the first and second test, we see that it increased by 41%. That means that TCP-Sack uses more efficiently the bandwith than TCP-Reno. Some bandwith which was wasted in the first test is now used by TCP-Sack in the second test.

4.1.2 Second example

In this experiment, two TCP-Reno connections start together at t=0s. At t=90s, a third connection starts. At t=270s, the third connection terminates. At t=360s, the first two connections terminate. We repeated this test twice in the same conditions, once with TCP-Reno as the third connection and once with TCP-Sack. The focus is on what is happening to the two first connections when a third flow enters the competition for bandwidth.

The link delay is set to 50ms and the bandwith is set to 10 Mb/s

See Fig. 32 for time-sequence plots.

See Fig. 33 and 34 for throughput tables.

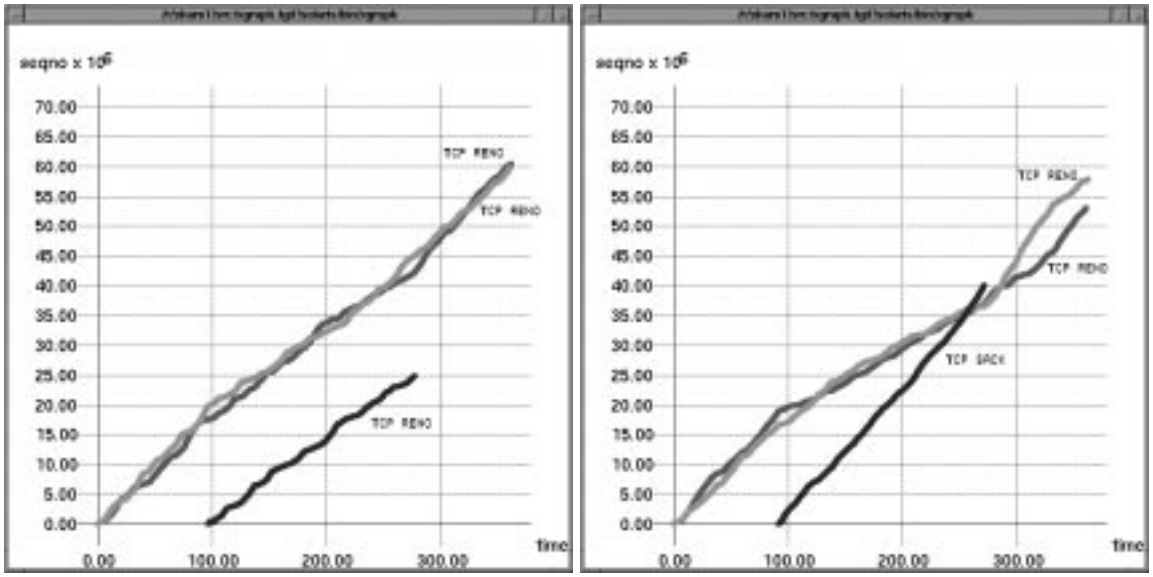

Figure 32: *Second example (link delay=50ms, bandwith=10MB/s)*

Figure 33: *Second example – 1st test*

	TCP-Reno1	TCP-Reno2	TCP-Reno
t=90s	190 kB/s	193 kB/s	
t=270s	153 kB/s	162 kB/s	140 kB/s
t=360s	167 kB/s	166 kB/s	

Figure 34: *Second example – 2nd test*

	TCP-Reno1	TCP-Reno2	TCP-Sack
t=90s	180 kB/s	204 kB/s	
t=270s	136 kB/s	139 kB/s	222 kB/s
t=360s	160 kB/s	147 kB/s	

The throughput of the third connection, starting at t=90s, improved by 58% with TCP-Sack. But, if we compare the throughput of the 2 TCP-Reno connections, it decreased by 11% and 14% at t=270s and by 4% and 11% at t=360s.

If we compare our first and second test, we can see that TCP-Sack is really more efficient than TCP-Reno. We can also see that TCP-Sack has an impact on the two other TCP-Reno connections. The two TCP-Reno connections are less efficient when they are competing against the more efficient TCP-Sack. But this effect is limited in its magnitude, compared to the throughput improvement provided by TCP-Sack.

4.2 Experiments with multiple flows

In Section 4.1, the focus was on experiments with only a few connections. In this part, we focus on the impact of TCP-Sack when there are multiple flows.

4.2.1 Methodology

All the following experiments were performed over our testbed, with a link delay of 10 ms, a link bandwith of 4 Mbps, a burst loss probability of 1and this is exactly what we need in order to study the impact of TCP-Sack on TCP-Reno.

In this experiment, N TCP-Reno connections compete in parallel for 90 seconds. Then, in the same conditions, $N-1$ TCP-Reno compete with 1 TCP-Sack connections. To compare the two experiments and what their results mean, one can imagine that the 'best' TCP-Reno is replaced by a TCP-Sack. The

meaningful information here is what is the impact on the $N-1$ other flows.

4.2.2 Results

The values of N range from 3 to 10. 4 useful variables R, r, S and s are used in the computations:

Figure 35: *R, r, S, s for N ranging from 3 to 10*

N	R	r	S	s
3	266 kB/s	172 kB/s	312 kB/s	170 kB/s
4	319 kB/s	234 kB/s	349 kB/s	234 kB/s
5	352 kB/s	274 kB/s	393 kB/s	270 kB/s
6	378 kB/s	311 kB/s	399 kB/s	304 kB/s
7	400 kB/s	335 kB/s	419 kB/s	324 kB/s
8	413 kB/s	355 kB/s	420 kB/s	344 kB/s
9	418 kB/s	365 kB/s	421 kB/s	358 kB/s
10	419 kB/s	367 kB/s	423 kB/s	362 kB/s

(See Fig. 36 for a plot of R,S,r and s in function of N.)

- R is the aggregated throughput of the N connections, in the case (N TCP-Reno).
- r is the aggregated throughput of the $N-1$ connections with the lowest throughputs, in the case (N TCP-Reno).
- S is the aggregated throughput of the N connections, in the case ($N-1$ TCP-Reno + 1 TCP-Sack).
- s is the aggregated throughput of the $N-1$ connections with the lowest throughputs, in the case ($N-1$ TCP-Reno + 1 TCP-Sack).

Therefore, $R-r$ is the highest throughput, in the case (N TCP-Reno). $S-s$ is the highest throughput, in the case ($N-1$ TCP-Reno + 1 TCP-Sack).

4.2.3 Analysis of the results

If we compare $S-s$ to $R-r$, we can find the **throughput improvement for the best TCP-Reno** when it is replaced by TCP-Sack. $ratio1 = ((S-s)-(R-r))/(R-r)$ quantifies this improvement.

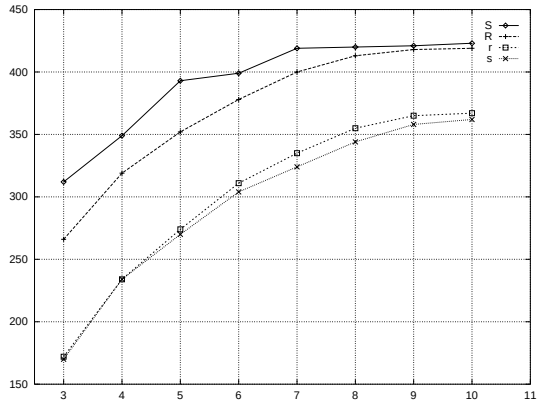

Figure 36: *R, r, S, s for N ranging from 3 to 10*

Figure 37: *ratio1, ratio2, ratio3*

N	ratio1	ratio2	ratio3
3	51.0%	19.7%	-1.16%
4	35.3%	16.6%	-0.00%
5	57.7%	27.7%	-1.46%
6	41.8%	17.2%	-2.25%
7	46.2%	19.0%	-3.28%
8	33.3%	8.05%	-3.10%
9	18.9%	3.66%	-1.92%
10	17.4%	4.94%	-1.37%

If we compare S to R, we can find what is happening to the aggregated throughput of our N connections after changing the best TCP-Reno into TCP-Sack. An interesting variable to measure the gain in aggregated bandwith is : $ratio2 = (S-R)/(B-R)$, with B being the maximum link bandwith (500 kB/s in our case). $B-R$ is the unused bandwith when all connections are TCP-Reno and $ratio2$ is **the proportion of this unused bandwith which is now used** when there is one TCP-Sack.

If we compare s to r, we can find **how much the total throughput of the other $N-1$ TCP-Reno is decreasing when the best TCP-Reno is changed into TCP-Sack**. $ratio3 = (s-r)/r$ quantifies this negative impact of TCP-Sack.

The first interesting thing is that the throughput

improvement for the best TCP-Reno when it is replaced by TCP-Sack is quite high. For N ranging from 3 to 8, this improvement ($ratio1$) ranges from 30% to almost 60%. For N equal to 9 or 10, this improvement is a little less than 20%.

$ratio2$ is also relatively high for N ranging from 3 to 7, being between 15% and 20%. That means that TCP-Sack uses more efficiently the bandwith than TCP-Reno. TCP-Sack is able to use some bandwith which was wasted by TCP-Reno and therefore when one TCP-Reno is replaced by one TCP-Sack, the total aggregated bandwith increases. When the number of connections N is high (9 or 10), filling completely the link capacity, $ratio2$ gets smaller because the amount of wasted is smaller. This explains too why $ratio1$ is smaller for high value of N.

Finally, $ratio3$ is small with a maximum value of approximately 3%. This means that the impact of TCP-Sack on the other $N-1$ TCP-Reno's is not significant.

The conclusion of these experiments is that even when TCP-Sack outperforms TCP-Reno by a large amount, the negative impact on the other competing connections is small. It can be explained by the fact that TCP-Sack uses more efficiently the bandwith and increases the total aggregated bandwith.

5 Conclusion

In this report of our work, we presented the experimental results of our study of TCP-Sack performance, and how it compares to the very common TCP-Reno.
We believe our study is the first to combine experiments on a testbed and experiments over a real path of the Internet.
It backs initial expectations that TCP-Sack is able to recover from multiple losses within one window of data without necessarily timing out. TCP-Sack *slow-starts* less often than TCP-Reno, and it is therefore closer to the ideal TCP Congestion Avoidance behavior [10].

The improvement in throughput between UCLA and the two test hosts at NASA's GSFC and PSC ranged between 15% and 45%. On the testbed (using virtual links and emulating a long-delay link), we had throughput improvements ranging from 10% to 120%, mainly depending on the congestion pattern.
The other issue was the possible unfairness of TCP-Sack when facing congestion. It was feared that TCP-Sack might take bandwidth away from non-Sack TCP stacks. Our experiments show that the implementation of TCP-Sack we used is not too aggressive: it makes a better use of the bandwidth wasted by TCP-Reno, but the competing TCP stacks are not affected by TCP-Sack.

We believe that Selective Acknowledgment is first and foremost a data recovery mechanism, and that it should be clearly separated from the Congestion Avoidance and Control algorithms. In TCP-Reno, the data recovery and congestion control algorithms were very tightly bound. Modifying each of them was very difficult. Data Recovery using Selective Acknowledgments is more robust, and less dependant on the Congestion Algorithms.

As a side effect, the current Congestion Algorithms (that are very much like Reno, even in the implementation we used) are perhaps a bit conservative when associated with Sack. We believe that Sack should not be viewed as Congestion Control Algorithm, but solely as a Data Recovery algorithm. We believe that there is room for improvements in the Congestion Control Algorithms, as illustrated in [9].

6 Acknowledgements

We would like to thank the following people for their help and support throughout this project: Lixia Zhang, our advisor, for her insight and support. All members of the UCLA's Internet Research Laboratory, and especially Scott Michel for his help in setting up the testbed. Elliot L. Yan of USC, for letting us use his port of Sack to FreeBSD. Kalyan Kidambi of NASA's Goddard Space Flight Center, and Jeff Semke of PSC, for providing us with two test hosts to conduct tests over the Internet. Shawn Ostermann for modifying *TcpTrace* for packets without

Link Layer Headers.

References

[1] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of tcp vegas: Emulation and experiment. Technical report, USC, 1995. http://excalibur.usc.edu/research/vegas/doc/.

[2] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. *Proceedings of ACM SIGCOMM '94*, pages 24–35, May 1994. ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps.

[3] Kevin Fall et al. Simulation-based comparisons of reno, tahoe, and sack tcp. *Computer communications Review*, July 1996.

[4] Sally Floyd. Issues of tcp with sack. Technical report, LBL Network Group, January 1996.

[5] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for tcp. *Proceedings of ACM SIGCOMM '96*, August 1996.

[6] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. Technical Report RFC 1323, IETF, 1992.

[7] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM'88*, 1988.

[8] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgement options. Technical Report RFC 2018, IETF, 1996.

[9] M. Mathis and M. Mahdavi. Forward acknowledgment: Refining tcp congestion control. *Computer Communication Review*, 26(4), 1996.

[10] M. Mathis, M. Mahdavi, J. Semke, and T. Ott. The macroscopic behaviour of the tcp congestion avoidance algorithm. *Computer Communications Review*, 1997.

[11] Shawn Ostermann. *TcpTrace, a Trace Analyser*. http://jarok.cs.ohiou.edu/software/tcptrace.

[12] Luigi Rizzo. Issues on the implementation of selective acknowledgement for tcp. Technical report, Universita di Pisa, 1996.

[13] W. Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, IETF, January 1997.