Issues of TCP with SACK

Sally Floyd[*]
floyd@ee.lbl.gov

VERY ROUGH DRAFT

March 9, 1996

1 Introduction

In this note we investigate several issues regarding the behavior of TCP with SACK, for the TCP SACK option as proposed in [MMFR95].

The first question that we address is the following: Will TCP with SACK be more aggressive in the presence of congestion than current TCP implementations, in a way that is damaging to the network?

A second, related question concerns how current TCP implementations fare in a congested environment competing against TCP implementations with SACK.

The final issue that we address is more specific, and concerns the limited number of SACK blocks in the SACK option format proposed in [MMFR95]. If a number of successive ACK packets are dropped in the network during a Fast Recovery period, the sender could be unaware that the receiver has already received a packet, and could retransmit that packet unnecessarily. In Section 3 we quantify the exposure of TCP with SACK to the unnecessary retransmission of packets. We show that this exposure is low, and is strictly less that the exposure of current TCP implementations.

For further background on our simulator and discussion of issues related to TCP with SACK, the reader is referred to [FF95].

2 Congestion control issues

2.1 Will TCP with SACK damage the network?

Certainly it would be possible to build "dangerous" implementations of TCP with SACK, just as it is possible to build dangerous implementations of TCP without SACK (dangerous, that is, to other users of the Internet). However, implementations of TCP with SACK should follow the underlying congestion control principles that have guided TCP implementations for the last eight years [Jac88].

That is, for every window of data from which one or more packets is dropped, the sender interprets this as an indication of congestion, and reduces the sender's congestion window by half. To further avoid overloading the network, outgoing data packets are effectively "clocked" by incoming ACK packets, and therefore obey a principle of "conservation of packets". With current TCP congestion control algorithms, an incoming ACK packet permits at most two outgoing data packets, even during a period of expansion such as a slow-start.

Implementations of TCP with SACK should also adhere to the current practices listed below. We argue that an implementation of TCP with SACK that follows these current practices and congestion control principles poses no danger to other traffic in the network. The behavior of TCP with SACK is identical to that of Reno TCP when at most one packet is dropped from one window of data. When multiple packets are dropped from one window of data, the behavior of TCP with SACK is similar to that of Reno TCP when only a single packet was dropped from the window. That is, TCP with SACK does not unnecessarily penalize the TCP connection when multiple packets are dropped from a single window of data. In all other respects, TCP with SACK behaves identically to Reno TCP.

1) Wait for three duplicate acknowledgements before retransmitting a packet. This delay provides robustness to packets reordered by the network. The sender might receive a single duplicate acknowledgement simply because two data packets were reordered in the network; in this case a retransmission by the sender is unnecessary.

2) Respond conservatively to sustained severe congestion. Both Tahoe and Reno TCP implementations respond more conservatively to sustained severe congestion than a simple halving of the congestion window each roundtrip time. For TCP with SACK implementations, this conservative response is provided by a possible wait for a retransmit timer to expire, followed by a slow-start, whenever a retransmitted packet is itself dropped. This conservative response is also provided by a sender's retransmission policy that, even during Fast Recovery, used incoming SACK packets to clock outgoing retransmitted data packets. This follows the current practice of Reno TCP of obey the principle of "conservation of packets" even during the Fast Recovery period.

In Tahoe implementations, a conservative response to sus-

[*]This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

tained congestion is provided by the use of slow-start for recovery for all congestion. In Reno implementations, when a number of packets are dropped in one window of data, or when a retransmitted packet is itself dropped, a conservative response to sustained congestion is provided by waiting for the retransmit timer to expire, followed by a slow-start.

3) Use retransmit timers to ensure the reliable delivery of data. In TCP with SACK, by using retransmit timers exactly as in done in Tahoe and Reno TCP, reliable delivery of data is ensured.

The behavior of TCP with SACK is in some respects easier to understand than the behavior of the TCP Tahoe and Reno algorithms. Unlike Tahoe TCP, with the complications of the slow-start and congestion avoidance phases, and Reno TCP, with the anomolous behavior that occurs when multiple packets are dropped from a window of data, the performance of TCP with SACK is more straightforward, easier to understand, and therefore easier to predict. TCP with SACK allows a fairly clean and straightforward implementation of the window decrease algorithm that has been used in the Internet for many years now, of reducing the sender's congestion window by half when one or more packets are dropped from the current window of data.

We will not necessarily understand every aspect of a modification to TCP before it is deployed in the network, However, TCP with SACK has been used for many years in simulations, and will receive more concentrated attention in the immediate future. It has received at least as much advance thought and discussion as have a number of other recent deployments that have changed traffic dynamics in the Internet (e.g., the WWW, unicast or multicast realtime traffic, reliable multicast traffic, etc.).

2.2 How will older TCP implementations fare against TCP with SACK?

Because TCP with SACK follows the same fundamental congestion control principles as do current implementations of TCP, connections using current TCP implementations will not be suddenly “shut out” of the network by having to compete against TCP with SACK. Nevertheless, current large-window TCP connections are handicapped in their achieveable throughput by the forced wait for a retransmit timer to expire when a number of packets are dropped in a window of data in Reno TCP [FF95], and by the requirement for a slow-start whenever a single packet is dropped in Tahoe TCP. The performance of large-window TCP [BBJ92] will remain handicapped until the SACK option is added to TCP.

Thus, if a large-window Tahoe or Reno TCP connection is competing for bandwidth against a large-window TCP with SACK connection, the TCP with SACK connection will receive the larger share of the link bandwidth. In the same way, if a Tahoe connection were to compete against a large-window Reno TCP connection in an environment without multiple drops for a single window of data (e.g., RED gateways [FJ93]), the Reno TCP connection would receive the larger share of the link bandwidth. And if a Tahoe connection were to compete against a Reno connection in a noisy environment with multiple packet drops in each congestion epoch, the Tahoe connection would receive the larger share of the link bandwidth.

Similarly, in the present Internet there are a number of other circumstances that would cause different TCP connections to receive different shares of the link bandwidth. These circumstances include different roundtrip times, different numbers of congested gateways, different TCP senders' clock granularities, different TCP receivers' policies with regards to delayed ACKs, and so on. It is not an argument against the deployment of TCP with SACK that TCP with SACK connections do not receive precisely the same bandwidth that a Tahoe or Reno TCP connection would receive in the same circumstances. However, there will certainly be performance incentives for high-bandwidth TCP connections to use TCP with SACK.

Because TCP with SACK follows the same fundamental congestion control principles as current implementations of TCP, “mice” (that is, short small-bandwidth TCP connections, of whatever TCP flavor) should not be unduely penalized by having to compete against “elephants” (that is, longer, large-bandwidth TCP connections) using TCP with SACK. (We intend to investigate this competition for bandwidth between mice and elephants in more detail in later work.)

3 The unnecessary retransmission of packets

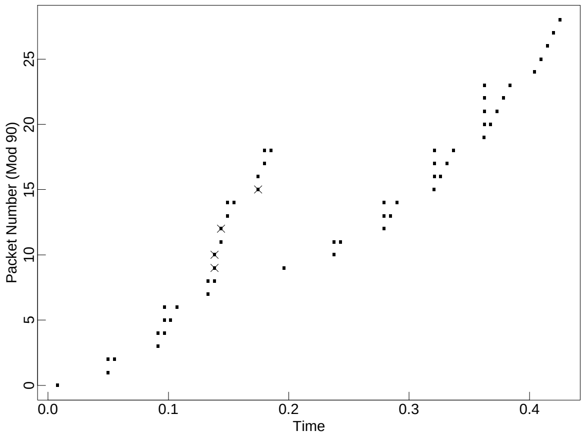
In this section we investigate the robustness of TCP with SACK with respect to the unnecessary retransmission of packets. We show that the exposure of TCP with SACK is strictly less than that of current implementations of TCP.

In the proposal [MMFR95], each SACK option packet can contain at least three SACK blocks, allowing each SACK block to be repeated at least three times in three successive ACK packets. However, if all of the ACK packets reporting a particular SACK block are dropped, then the sender will assume that the data in that SACK block has not been received, and will unnecessarily retransmit those segments.

In this section we show that the exposure of TCP with SACK in regard to the unnecessary retransmission of packets is strictly less than the exposure of current implementations of TCP.

3.1 Worst-case scenarios with Tahoe TCP

Figure 1 shows simulations with Tahoe TCP. The bottom example in Figure 1 uses a delayed-ACK receiver, and the top example does not. In Tahoe TCP, the sender will often unnecessarily retransmit packets when multiple packets are



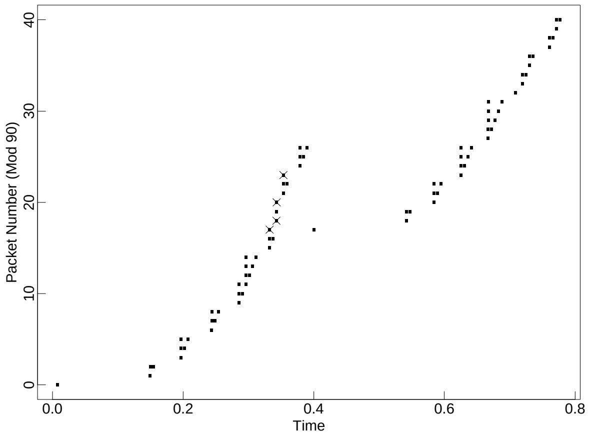


Figure 1: Duplicate packets retransmitted with Tahoe TCP.

dropped from a single window of data. For both simulations in Figure 1, four packets are dropped from a window of 10 packets, and the other six packets in the window are unnecessarily retransmitted by the sender.[1] The simulations were run by specifying to the simulator exactly which packets to drop at the gateway, rather than by laboriously scheduling competing traffic to produce the desired pattern of packet drops. This allows us to easily create worst-case scenarios.

Figure 1 was generated by tracing data packets entering and departing from the congested gateway. For each graph, the x-axis shows the packet arrival or departure time in seconds. The y-axis shows the packet number mod 90. Packets are numbered starting with packet 0. Each packet arrival and departure is marked by a dot on the graph. For example, a single packet passing through the gateway experiencing no appreciable queueing delay would generate two marks so close together on the graph as to appear as a single mark. Packets delayed at the gateway but not dropped will generate two colinear marks for a constant packet number spaced by the queueing delay. Packets dropped due to buffer overflow are indicated by an “X” on the graph for each packet dropped.

Because of the limitations of the cumulative acknowledgement, the sender in these simulations can find out about at most one additional dropped packet per roundtrip time. Consider the top simulation in Figure 1. At time 0.2 the sender receives three duplicate ACKs and initiates Fast Retransmit, retransmitting packet 9 and and entering slow-start. When the sender receives the ACK for packet 9, acknowledging all packets up to and including that packet, the sender increases its congestion window to two packets and retransmits packets 10 and 11. While packet 10 was needed at the receiver, packet 11 was not, and the retransmission of that packet was a waste of possibly-valuable link bandwidth. When the receiver receives the retransmitted packet 10, it sends an acknowledgement for all packets up to and including packet 11.

Continuing in this fashion, when the sender receives the acknowledgement for packet 11, it increases its congestion window from two to three, and retransmits packets 12, 13, and 14. While packet 12 was needed at the receiver, packets 13 and 14 were retransmitted unnecessarily.

With Tahoe implementations of TCP, when the congestion window contains at least six packets the sender might unnecessarily retransmit half of those packets. When the congestion window contains more than six packets, it is easy to construct scenarios where the sender unnecessarily retransmits more than half of the packets in that window. Note that behavior is similar with and without delayed acks.

3.2 Worst-case scenarios with Reno TCP

Figure 2 shows worst-case scenarios of unnecessarily retransmitted packets with Reno TCP.[2] The bottom example in Figure 2 uses a delayed-ACK receiver, and the top example does not. In both simulations, six packets are dropped from a window of eleven packets, and five packets from that window are unnecessarily retransmitted by the sender. In both simulations the sender has to wait for a retransmit timer to expire to recover from multiple packets dropped from a single window of data.

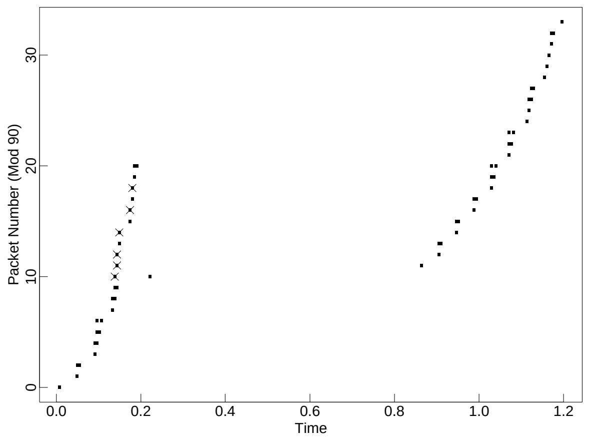
3.3 Worst-case scenarios with TCP with SACK

In contrast to these examples with Tahoe and Reno TCP, it is more difficult to construct scenarios with TCP with SACK where the sender unnecessarily retransmits packets. For this section we consider a SACK option packet with room for exactly three SACK blocks per SACK packet.

In this section we make the following assumptions about the sender's retransmit policies for TCP with SACK. First, the sender does not retransmit any packets until it has received three duplicate ACKs (that is, ACKs that don't ad-

[1]These simulations can be run on “ns” with the commands “ns dups.tcl tahoe1” and “ns dups.tcl tahoe2”, respectively. These simulations are run with “bug-fix” set to true, to avoid multiple fast retransmits from drops in a single window of data [Flo94].

[2]These simulations can be run on “ns” with the commands “ns dups.tcl reno2” and “ns dups.tcl reno1”, respectively.



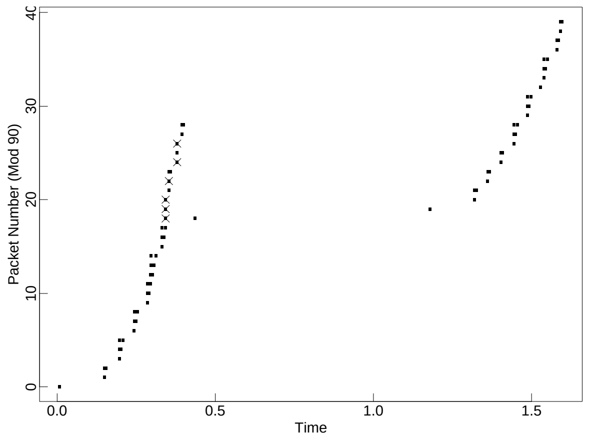


Figure 2: Duplicate packets retransmitted with Reno TCP.

vance the cumulative acknowledgement). Second, for every duplicate ACK received, the sender retransmits at most one data packet. These constraints limit the exposure of TCP with SACK with regards to the unnecessary retransmission of packets. The exposure of TCP with SACK is further limited by the redundancy in the SACK option that provides that SACK blocks appear in several successive SACK option packets.

Following a retransmit timeout, TCP with SACK uses the same slow-start procedure as does Reno or Tahoe TCP, and therefore has exactly the same exposure to unnecessarily-retransmitted packets. We show that in the absence of a retransmit timeout, the simpliest scenario that can produce a single unnecessarily-retransmitted packet for TCP with SACK requires a sender congestion size of at least 11 packets.

Why is a congestion window of at least 11 packets required? Without constructing a formal proof, here is the description:

1) At least one data packet has to have been dropped to cause a Fast Retransmit in the first place.

2) The sender has to successfully receive three dup ACKs to begin a Fast Retransmit. These account for three more of the 11 required data packets.

3) At least three SACK packets have to be dropped for

Data Pkt.	ACK packet	Eventual sender reaction
1	Normal ACK:1	Send 12 (normal)
2	(data packet lost)	
3	Dup Ack:1 3-3	No action
4	Dup Ack:1 3-4	No action
5	Dup Ack:1 3-5	Retransmit pkt. 2
6	Dup Ack:1 3-6	(ACK lost)
7	(data packet lost)	
8	Dup Ack:1 8-8 3-6	(ACK lost)
9	(data packet lost)	
10	Dup Ack:1 10-10 8-8 3-6	(ACK lost)
11	(data packet lost)	
12	Dup Ack:1 12-12 10-10 8-8	Retransmit pkt. 6

Figure 3: Worst-case example for TCP with SACK.

the sender to be unaware that the receiver has received a particular packet. These account for three more (seven so far) of the 11 required data packets.

4) For a data packet to be acknowledged in only three successive SACK packets, the data packet has to be followed by a dropped packet, followed by two singletons (that is, data packets both preceded and followed by dropped packets). These account for at least three more dropped data packets, for a total so far of 10 of the 11 required data packets.

5) Finally, because the first packet retransmitted by the sender is always a packet that has never yet been acknowledged, the sender must receives at least a fourth duplicate ACK in order to send the unnecessarily-retransmitted packet. This accounts for all 11 of the required packets.

So we have established that a congestion window of at least 11 packets is required for the sender to unnecessarily retransmit a packet, with at least four data packets dropped from that window of data. Given such a sequence of 11 data packets, a precise sequence of exactly three lost ACK packets is required in order for the sender to unnecessarily retransmit a packet.

In this example, because three successive SACK packets were dropped, the sender is unaware that the receiver has received packet 6. Given this sequence of data packets in the forward direction, this sequence of dropped SACK packets is the ONLY sequence of droppped SACK packets that will result in an unnecessarily-retransmitted data packet. Given a congestion window of at least eleven packets, what is the chance that the sequence of dropped data packets is one that, if accompanied by just the right sequence of dropped ACK packets, would permit the sender to unnecessarily retransmit a packet? And, given that pattern of dropped data packets, what is the chance that the pattern of dropped ACK packets is one that enables the sender to unnecessarily retransmit a packet? For example, in this particular case, there are $2^7 = 128$ patterns by which these seven SACK packets could or could not be dropped on the path from the receiver to the sender. What are the chances that the dropped packets will be in exactly this one of the 128 possible patterns?

While we have not attempted a complete analysis of the worst-case probabilities of unnecessarily retransmitted packets for TCP with SACK, we are convinced that the number of unnecessarily-retransmitted packets will be acceptably low, and that in any case the number of unnecessarily-retransmitted packets is strictly less that that in corresponding Tahoe and Reno implementations. We are aware of the throughput degradation that is possible when the TCP sender unnecessarily retransmits packets or cells (and have actually written an entire paper on the subject, in the context of TCP over ATM[RF95]).

4 Future research on congestion control algorithms

Clearly the use of the SACK option will open the way for research on modifications to the TCP congestion control algorithms, such as the FACK algorithm proposed by Matt Mathis and Jamshid Mahdavi [add reference], or potential proposals for the use of TCP with SACK over lossy links such as wireless or satellite links. Any such modifications to TCP's underlying congestion controls should, of course, receive the close attention and review that would be received by any other proposed modification to TCP's underlying congestion controls.

5 Previous research on TCP with SACK

In this section we summarize other published work investigating the performance of TCP with SACK.

Simulations in [Flo91] investigate the performance of TCP with SACK (using the Reduce-by-Half window decrease algorithm and the Increase-by-One window increase algorithm, in the terminology of that paper) in scenarios with a number of active TCP connections with a range of round-trip times and numbers of congested gateways. Figure 5 from [Flo91], for example, shows the performance of TCP with SACK in an environment with RED gateways. Figures 9-11 from [Flo91] considers the performance of TCP with SACK with a ranqe of gateway queueing disciplines, for RED, Random Drop, and Drop Tail gateways.

6 Conclusions

This note has addressed three possible concerns about the effects of TCP with SACK and shown that the behavior of TCP with SACK is unlikely to cause undesirable network effects.

7 Acknowledgements

I have included contributions and ideas from Kevin Fall, Matt Mathis, Jamshid Mahdavi, and Allyn Romanow. (That does not necessarily mean that everyone acknowledged agrees with everything in this document.)

References

[BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance. Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992. (Obsoletes RFC1185).

[FF95] K. Fall and S. Floyd. Comparisons of tahoe, reno, and sack tcp. Technical report, 1995. Available via http://www-nrg.ee.lbl.gov/nrg-papers.html.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug 1993. Available via http://www-nrg.ee.lbl.gov/nrg-papers.html.

[Flo91] Sally Floyd. Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic. *ACM Computer Communication Review*, 21(5):30–47, Oct 1991.

[Flo94] Sally Floyd. Tcp and successive fast retransmits. Technical report, 1994. Available via ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[Jac88] V. Jacobson. Congestion avoidance and control. *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 314–329, 1988. An updated version is available via ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[MMFR95] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. Tcp selective acknowledgement option. (Internet draft, work in progress), 1995.

[RF95] Allyn Romanow and Sally Floyd. Dynamics of tcp traffic over atm networks. *IEEE Journal on Selected Areas in Communications*, 13(4), 1995. Available via http://www-nrg.ee.lbl.gov/nrg-papers.html.