Congestion Control: SCTP vs TCP*

Armando L. Caro Jr., Keyur Shah, Janardhan R. Iyengar, Paul D. Amer
Protocol Engineering Lab
Computer and Information Sciences
University of Delaware
{acaro, shah, iyengar, amer}@cis.udel.edu

Randall R. Stewart
Cisco Systems Inc.
rrs@cisco.com

Abstract

We characterize an inefficiency in the current specification of SCTP's congestion control, which degrades performance (more than necessary to be "TCP-friendly") when there are multiple packet losses in a single window. We present an SCTP variant, called *New-Reno SCTP*, which introduces three modifications. First, a Fast Recovery mechanism, similar to that of New-Reno TCP, is included to avoid multiple congestion window (cwnd) reductions in a single round-trip time. Second, we introduce a new policy which restricts the cwnd from being increased during Fast Recovery, thus ensuring that the newly introduced Fast Recovery mechanism maintains conservative behavior. Third, we modify SCTP's HTNA (Highest TSN Newly Acked) algorithm to ensure that Fast Retransmits are not unnecessarily delayed. We show that New-Reno SCTP performs better, while at the same time, still conforms to AIMD principles. Also, we compare these two variants of SCTP with New-Reno TCP, SACK TCP, and FACK TCP under six different loss scenarios. Our results show that New-Reno SCTP is as robust as TCP FACK, and more robust than the others.

1 Introduction

Until now, there have been two general purpose transport protocols widely used for applications over IP networks: UDP and TCP. Each provides a set of services that cater to certain classes of

*Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U.S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

applications. However, the services provided by TCP and UDP are disjoint, and together do not satisfy ideally the needs of all network applications. The Stream Control Transmission Protocol (SCTP), designed to bridge the gap between UDP and TCP, addresses shortcomings of both.

SCTP was originally developed to carry telephony signalling messages over IP networks for telecommunications and e-commerce systems. With continued work, SCTP evolved into a general purpose transport protocol that includes advanced delivery options. Like TCP, SCTP provides a reliable full-duplex connection, called an *assocation*. However, within an SCTP association, multistreaming allows for independent delivery among streams, which reduces the risk of head-of-line blocking. SCTP supports multihoming to provide redundancy at the path level, thus increasing association survivability in the case of a network path failure. Finally, SCTP's four-way handshake for association establishment makes it resistant to SYN attacks, and hence increases overall security.

SCTP employs congestion control algorithms similar to those used in TCP. Although SCTP's congestion control mechanisms are expected to be more robust to loss and provide better performance, we have discovered a flaw in SCTP's current mechanisms (which we refer to as *Original SCTP*[1]). Original SCTP suffers when there are multiple packet losses in a single window. Under these conditions, Original SCTP improperly reduces the congestion window (cwnd) for each detected loss.

We introduce an SCTP variant, called *New-Reno SCTP*, which introduces into SCTP a Fast Recovery mechanism similar to that of New-Reno TCP. We show that New-Reno SCTP performs better than Original SCTP and TCP variants, and importantly, continues to conform to AIMD [4, 7] principles (making it "TCP-friendly"). To illustrate the advantages of New-Reno SCTP, we present simulations with detailed explanations comparing Original SCTP and New-Reno SCTP for two loss scenarios. Also, we compare these two variants of SCTP with New-Reno TCP, SACK TCP, and FACK TCP under six different loss scenarios.

In Section 2 we describe the congestion control and packet retransmission algorithms for Original SCTP and New-Reno SCTP. We assume the reader is familiar with New-Reno TCP [], SACK TCP [], and FACK TCP []. Section 3 explains the details of the simulation environment used to produce our results. Important features of these three congestion control approaches are highlighted in the paper in comparison with the SCTP variants. Section 4 provides detailed simulations results for Original SCTP and New-Reno SCTP under two simulation scenarios. In Section 5, we compare the performance of the two SCTP variants with New-Reno TCP, SACK TCP, and FACK TCP. Section 6 concludes the paper.

[1] We refer to Original SCTP as specified in RFC2960 [14] and version 7 of the SCTP Implementer's Guide [12].

2 SCTP Variants

2.1 Original SCTP

SCTP is defined in RFC2960 [14] with certain changes and additions included in the SCTP Implementer's Guide [12]. SCTP uses a SACK-based ack scheme similar to SACK TCP. SCTP's congestion control algorithms are based on RFC2581 [1], with some subtle differences in the actual mechanisms. Like TCP, SCTP uses three control variables: rwnd, cwnd, and ssthresh. SCTP introduces an additional variable, partial_bytes_acked (pba), to calculate cwnd growth during the Congestion Avoidance phase.[2]

SCTP performs Slow Start when cwnd $\leq$ ssthresh. In contrast, TCP implementations are given the freedom to choose between Slow Start or Congestion Avoidance when cwnd = ssthresh. Incidentally, ns-2 chooses Congestion Avoidance when these variables are equal.

During Slow Start, the cwnd is increased only if two conditions hold true: (1) the incoming ack advances the cumulative ack point (cumack), and (2) the full cwnd was in use before the ack arrived. If so, the cwnd is incremented by min(newly acked data, MTU), where "newly acked" data includes any data not previously acked. If an ack is a duplicate ack (i.e., the ack does not advance the cumack, but includes selective acks), then the newly acked data reduces the amount of outstanding data. Hence, the unchanged value of cwnd now allows new data to be sent.

SCTP performs Congestion Avoidance when cwnd $>$ ssthresh. During Congestion Avoidance, the goal is to increase the cwnd by one MTU every RTT. SCTP uses pba to facilitate this mechanism. Initially, pba is set to zero. When a SACK that advances the cumack arrives, pba is incremented by the number of bytes newly acked in the SACK, as determined by the cumulative and selective ack feedback. The cwnd is increased by one MTU when an ack arrives and the following two conditions hold true: (1) pba $\geq$ cwnd, and (2) the full cwnd was in use before the ack arrived. In addition to incrementing the cwnd, pba is reset to pba $-$ cwnd. When all of the data sent by the sender has been acked by the receiver, pba is reset to zero.

As in all variants of TCP, SCTP uses two mechanisms to detect loss: retransmission timeout and Fast Retransmit. SCTP's Fast Retransmit algorithm is slightly different from TCP's. First, Fast Retransmit is triggered by four "missing reports" (via SACKs) instead of three "duplicate acks" as with TCP. Also, the missing reports are counted following the *HTNA* (Highest TSN Newly Acked) algorithm. This algorithm attempts to handle stray packets due to reordering as follows. For each incoming ack, the highest *TSN* (Transmission Sequence Number) being newly acked becomes the frame of reference for incrementing missing reports. Only TSNs prior to this reference point can have their missing report incremented with the current ack [12].

To avoid bursts, SCTP uses a Congestion Window Validation algorithm similar to the one described in [6]. Any time the sender has new data to send, the maxburst parameter (recommended to be 4) is first applied as follows [12]:

[2] Since SCTP supports multihoming, the sender maintains cwnd, ssthresh, and pba per destination. For simplicity, we assume single-homed hosts.

```
if((outstanding + maxburst*MTU) < cwnd)
    cwnd = outstanding + maxburst*MTU
```

This algorithm automatically handles scenarios where the sender's cwnd is stale due to idle or application-limited behavior which does not use its full cwnd. The sender does not need to periodically adjust the cwnd for such scenarios as originally suggested in RFC2960 [14].

2.2 New-Reno SCTP

Original SCTP does not include a Fast Recovery mechanism, as found in TCP. The SCTP authors state that "because cwnd in SCTP indirectly bounds the number of outstanding TSN's, the effect of TCP fast-recovery is achieved automatically with no adjustment to the congestion control window size" [14]. The authors were correct that Fast Recovery is not needed to clock out new data packets while the sender is recovering from a Fast Retransmit. However, Original SCTP suffers when there are multiple losses in a single window of data. Each loss reduces the cwnd by half, as experienced by Reno TCP [5].[3] SCTP should avoid performing multiple cwnd reductions per window, as do New-Reno TCP, SACK TCP, and FACK TCP.

In this paper, we introduce an SCTP variant, called New-Reno SCTP, which includes recover state. The Fast Recovery period begins when a Fast Retransmit is triggered. Analogous to the variable `recover` in New-Reno TCP, New-Reno SCTP stores the highest outstanding TSN to mark the end of the Fast Recovery period. Hence, when an ack acknowledges all TSNs up to and including this stored TSN, Fast Recovery is exited. During the Fast Recovery period, subsequently detected lost packets are Fast Retransmitted without reduction of cwnd.

Original SCTP increases the cwnd for any ack that introduces a new cumack, including partial acks. Although the concept of a partial ack does not exist in Original SCTP, with the introduction of a recovery period, a partial ack in SCTP is defined as in New-Reno TCP. New-Reno SCTP introduces a new policy which avoids increasing the cwnd for partial acks. Since Original SCTP had a cwnd reduction for each loss (even if they occured in the same window), this policy was not necessary. But with New-Reno SCTP, this policy is necessary to avoid increasing cwnd during multiple loss scenarios.

SCTP's Fast Recovery mechanism differs from New-Reno TCP, SACK TCP, and FACK TCP in that subsequent losses in the same window continue to be triggered for Fast Retransmit by four missing reports. While in Fast Recovery, New-Reno TCP, SACK TCP, and FACK TCP trigger subsequent Fast Retransmits by a single missing report (or partial ack in the case of New-Reno TCP). The assumption is that once a loss is detected, missing reports in the same window are more likely due to loss rather than reordering. However, in scenarios where loss is combined with reordering, Fast Retransmits triggered by a single missing report may cause spurious retransmissions. Hence, SCTP's Fast Recovery is more conservative, but is more robust to such scenarios.

[3] SCTP does not suffer as badly as Reno TCP in the face of multiple losses in a window. Reno TCP almost always suffers a retransmission timeout when there are three or more packet drops [5].

New-Reno SCTP also introduces a modified HTNA algorithm for Fast Retransmit. The original HTNA algorithm sometimes delays the triggering of Fast Retransmit. The problem occurs when there are multiple losses in a window. The retransmission of the first lost packet triggers an ack which advances the cumack, but does not ack the remaining lost packets. Upon the arrival of this ack, the sender recognizes the retransmitted TSN as the highest TSN newly acked at this time. The remaining lost packets have higher TSNs; thus, according to the current HTNA algorithm, the remaining lost packets will not have their missing reports incremented. However, this ack should increment the missing reports, because this ack does not correspond to a reordered packet; instead, it corresponds to a retransmitted packet. The modified HTNA algorithm solves this problem. Any time the sender is in Fast Recovery, an ack that advances the cumack is always used to increment missing reports.

In summary, New-Reno SCTP differs from Original SCTP in the three following ways. First, a Fast Recovery mechanism, similar to that of New-Reno TCP, is included to avoid multiple congestion window (cwnd) reductions in a single round-trip time. Second, we introduce a new policy which restricts the cwnd from being increased during Fast Recovery, thus ensuring that the newly introduced Fast Recovery mechanism maintains conservative behavior. Third, we modify SCTP's HTNA (Highest TSN Newly Acked) algorithm to ensure that Fast Retransmits are not unnecessarily delayed.

3 Simulations

All our simulations were done using ns-2 [2], which supports the flavors of TCP in this paper. SCTP, however, is available as a third party module developed by the Protocol Engineering Lab at the University of Delaware [3]. The simulations presented were produced using the same scripts used in [5] with modifications to include the two SCTP variants and additional loss scenarios.

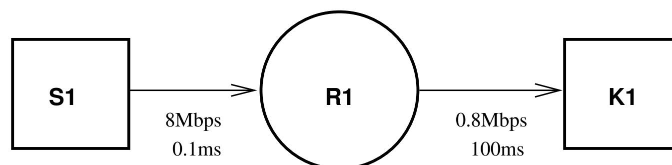

Figure 1: Simulation Topology

As show in Figure 1, we use the same network topology used in [5]. The topology simply consists of a sending and receiving host (labelled S1 and K1, respectively) connected via a drop-tail router (R1). The bandwidth and delay of the links are indicated in the figure. Each simulation consists of only a single TCP connection or SCTP association between S1 and K1. The loss pattern in each of the scenarios is specified explicitly using a "droplist" in ns-2. For simplicity and to be consistent with the results presented in [5], all simulations consist of one-way traffic and delayed acks are not used. Also, the link characteristics, buffer sizes, and loss pattern are not intended to be realistic. They provide simple scenarios which illustrate the congestion control algorithms of

TCP and SCTP.

4 Original vs New-Reno SCTP

We present detailed simulations results for Original SCTP[4] and New-Reno SCTP under two simulation scenarios: one and two drop scenarios. We chose the same loss scenarios for one and two drops that Fall and Floyd used in [5]. Also, it should be noted that though the congestion control variables for SCTP are manipulated in bytes and multiples of the MTU, our detailed analysis is done in packets for simplicity.

We present the results by graphing the packets which enter and depart R1 over the course of the simulation. The x-axis depicts the simulation time in seconds, while the y-axis represents the packet number modulo 50 (packet numbering begins at 1). A data packet arrival is indicated on the graph as a ■, whereas a □ marks the departure of a data packet. Any horizontal space between these two marks represents the queuing delay the packet experiences at R1. Packets dropped upon arrival at R1 due to buffer overflow are marked with an $\times$ on the graph. The acks in the reverse direction are marked with a ○ at the time they arrive at R1. Only the cumulative ack is represented on the graph; selective ack information is not conveyed explicitly and must be inferred from the graph. Also, we assume that each packet corresponds to a single TSN.

4.1 One Packet Loss

Figures 2 shows the behavior of Original SCTP and New-Reno SCTP with one dropped packet. Let us first consider **Original SCTP**. Packets 1-14 are sent successfully, and in the process, the cwnd increases exponentially from 1 to 15 according to the Slow Start algorithm. Since packet 15 is lost, packets 16-29 arrive at the receiver and trigger fourteen duplicate acks for packet 14. The first three duplicate acks for packet 14 decrease the number of outstanding packets by one each. Thus, packets 30-32 are clocked out.

Upon receiving the fourth duplicate ack for packet 14 (which serves as the fourth missing report for packet 15), the sender Fast Retransmits packet 15. As in TCP's Fast Retransmit algorithm, the sender reduces the cwnd and ssthresh, but Orignal SCTP reduces them to 7.5. Also, Original SCTP does not have a Fast Recovery mode. Since the cwnd in SCTP represents only how much the sender can send and not which packets can be sent, a Fast Recovery algorithm is not needed to keep the ack clock going [14].

The fifth duplicate ack for packet 14 reduces the number of outstanding packets from 15 to 14. Each additional duplicate ack for packet 14 received similarly decreases the number of outstanding packets by one. Since the number of outstanding bytes may exceed cwnd by at most $\mathrm{MTU}-1$

[4] Although we refer to Original SCTP as the specification in RFC2960 [14] and changes up to version 7 of the SCTP Implementer's Guide [12], the ns-2 implemenation of SCTP only includes up to version 4 [3]. The differences do not affect the results in this paper.

bytes [12], the eleventh duplicate ack reduces the oustanding packets to 7, and begins to clock out new packets once again. The last seven duplicate acks for packet 14 allows the sender to transmit packets 33-39.

Upon receiving the ack for packet 32, the cwnd and ssthresh are 7.5, and hence, the sender is in Slow Start mode. In response to the ack for packet 32, the sender increases the cwnd to 8.5 and transmits packets 40-41. Finally, the sender continues transmitting in Congestion Avoidance mode.

As show in Figure 2, **New-Reno SCTP** shows no difference from Original SCTP with one packet drop (as expected).

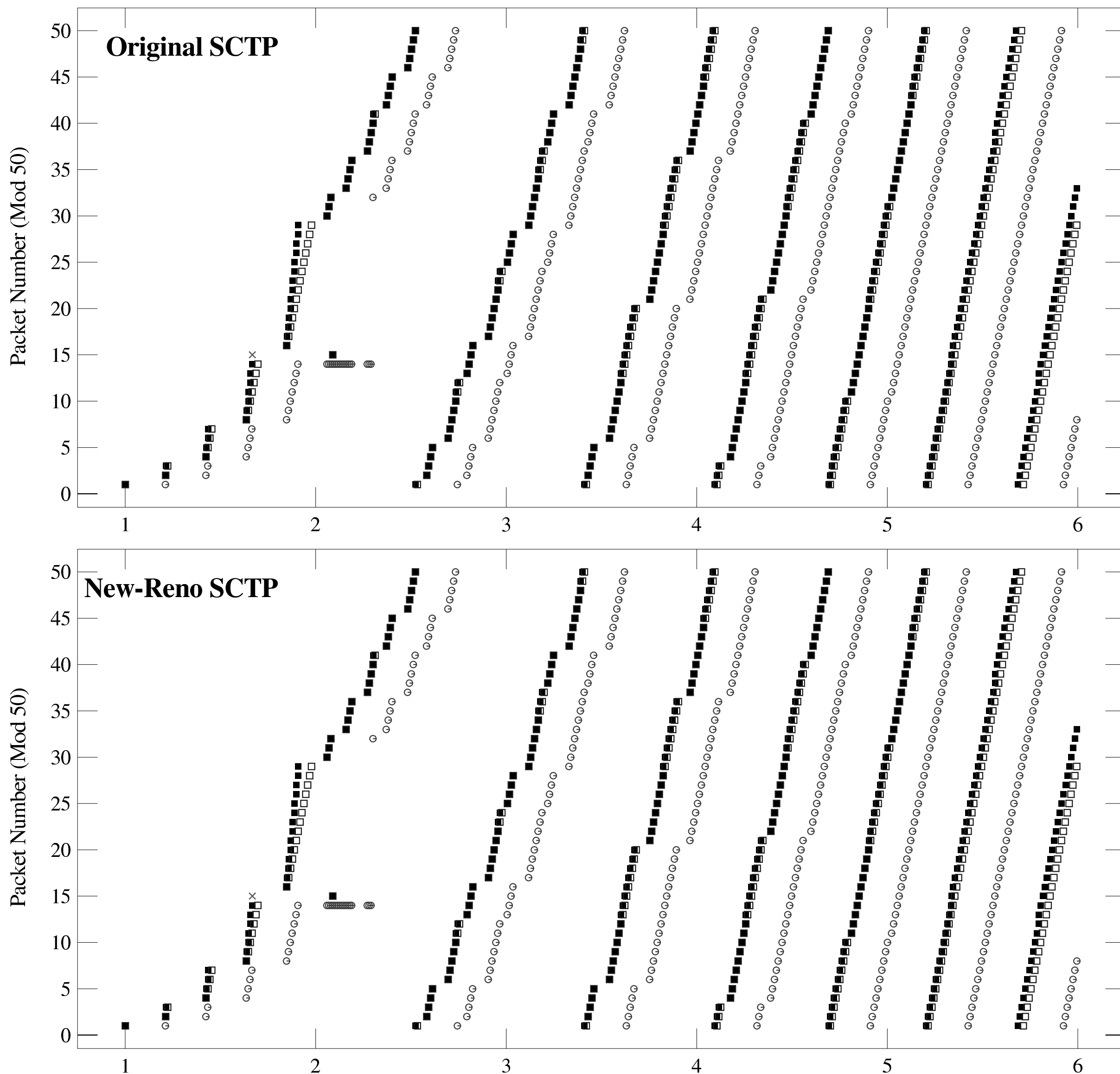

Figure 2: SCTP simulations with one dropped packet

4.2 Two Packet Losses

Figures 3 shows the behavior of Original SCTP and New-Reno SCTP with two dropped packets. **Original SCTP** initially behaves the same as explained earlier with one drop, except that there is one less duplicate ack for packet 14. Hence, all the duplicate acks for packet 14 allow the sender to transmit only 30-38. The last three duplicate acks for packet 14 contain selective acks for packets 30-32, and hence count as missing reports for packet 29.

The next ack is for packet 28, and corresponds to the arrival of the retransmitted packet 15. Following the HTNA algorithm (see Section 2.1), this ack's newest newly acked packet is 15, which does not count as a missing report for packet 29. This is a flaw with the current HTNA algorithm which is addressed in New-Reno SCTP. Since the sender is in Slow Start mode, the cwnd is increased to 8.5 in response to this ack. The sender now transmits packets 39-40, and moves into Congestion Avoidance mode with nine packets outstanding.

The first duplicate ack for packet 28 counts as the fourth missing report for packet 29, and triggers the Fast Retransmit of packet 29. As a result, the cwnd is cut to 4.25.

Upon receiving the fifth duplicate ack for packet 28, the number of outstanding packets decreases to four. Thus, the sender transmits packet 41. Likewise, the next three duplicate acks for packet 28 allow the sender to send packets 42-44.

Finally, an ack for packet 40 arrives. Since the cwnd and ssthresh are 4.25, the sender is in Slow Start mode and increases the cwnd to 5.25. The sender then transmits packets 45-46 and continues transmitting in Congestion Avoidance mode.

New-Reno SCTP behaves the same as Original SCTP until the first ack for packet 28. Due to the modified HTNA algorithm, this ack now triggers the Fast Retransmit of packet 29. Note that previously with Original SCTP, this ack did not trigger the Fast Retransmit of packet 29 because it did not increment packet 29's missing report. Also, to contrast Original SCTP's behavior, New-Reno SCTP does not reduce the cwnd for this Fast Retransmit since the sender is in Fast Recovery mode. Hence, the cwnd remains at 7.5. Since there are only seven outstanding packets at this point, the sender may transmit packet 39. The next six duplicate acks for packet 28 clock out packets 40-45.

The sender exits Fast Recovery mode when the ack for packet 38 arrives. The sender is in Slow Start mode and increases the cwnd to 8.5. Then, the sender transmits packets 46-47 and continues transmitting in Congestion Avoidance mode. Note that New-Reno SCTP exits Fast Recovery sooner, with a larger cwnd, and with more packets transmitted than Original SCTP. These differences lead to significant improvement in performance of New-Reno SCTP over Original SCTP (see Section 5).

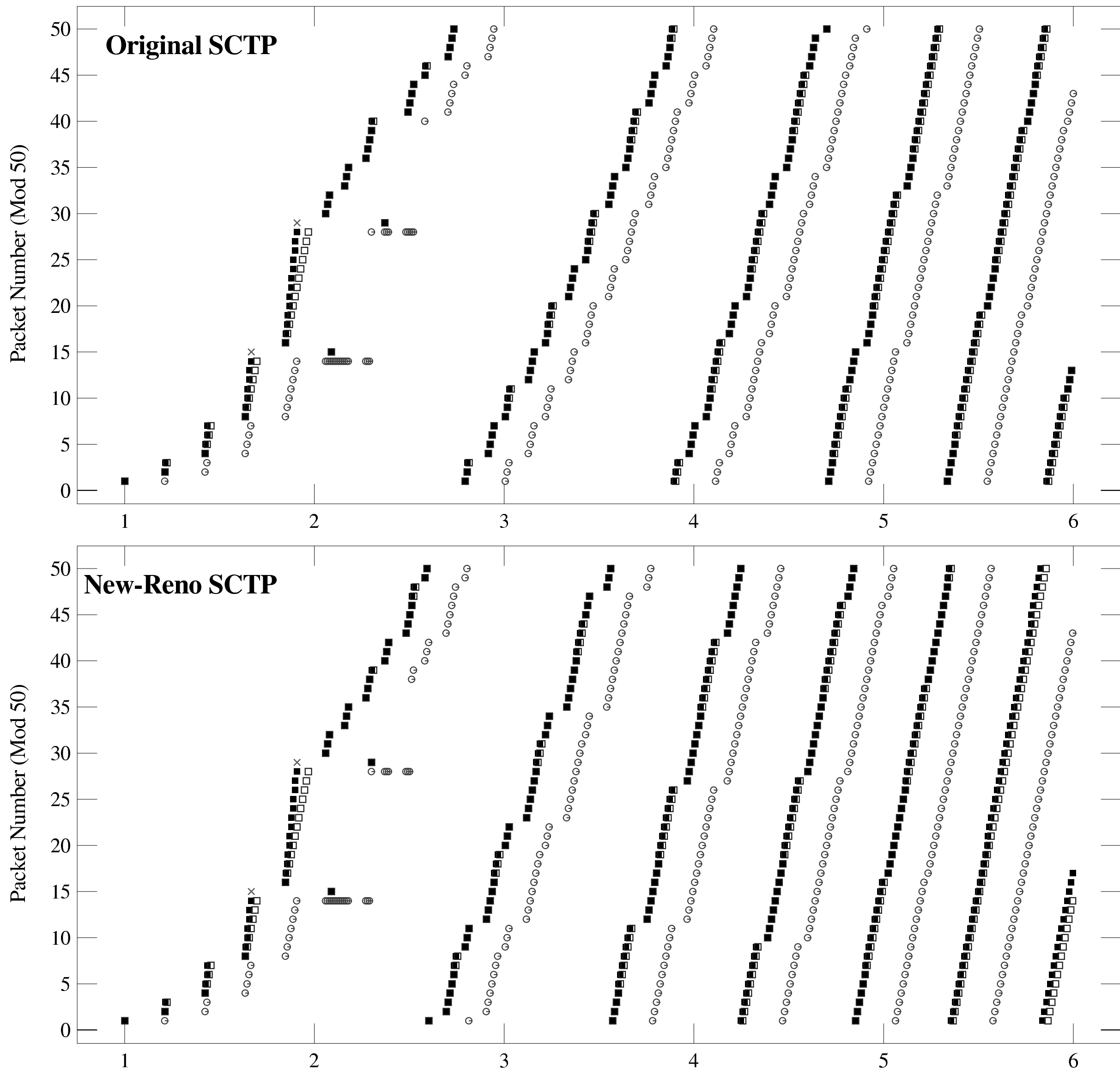

Figure 3: SCTP simulations with two dropped packets

5 Comparisons with TCP Variants

We present final simulation results for New-Reno TCP, SACK TCP, FACK TCP, Original SCTP, and New-Reno SCTP under six different loss conditions. As explained in Section 2.1, the ns-2 implementation of TCP performs Congestion Avoidance when cwnd = ssthresh. SCTP, however, is specified to perform Slow Start when these variables are equal. To eliminate the effect of this difference, we also present results for a protocol labelled "New-Reno SCTP*". New-Reno SCTP* is equivalent to New-Reno SCTP, but differs in that Congestion Avoidance is performed when cwnd = ssthresh. New-Reno SCTP* is included only for the purpose of fairly comparing with the

TCP-variants; we are not proposing this change be made to SCTP. We include New-Reno TCP in our comparisons, because it is the most widely deployed version of TCP [11], and it is the best peforming non-SACK variant of TCP [5]. Since SCTP uses selective acks, two SACK-based TCP variants are included as well: SACK TCP [5, 10] and FACK TCP [8].[5] Although SACK TCP is more conservative and does not perform as well as FACK TCP, we included it for completeness.

(@@@ Also New-Reno SCTP** is just like New-Reno SCTP* with one difference. The cwnd is incremented/decremented in whole MTUs. Since the ns-2 implementation of TCP only deals with packets and not bytes, a cwnd of 15 is cut in half to 7 when Fast Retransmit is triggered. However, since the ns-2 implementation of SCTP deals bytes, the cwnd is cut to 7.5, which allows the sender to have 8 outstanding packets. New-Reno SCTP** is an attempt to make the comparison even MORE fair by making SCTP cut the cwnd to 7. We still have to decide whether to either keep these * and ** versions in the final paper.)

We present results for six loss scenarios: zero to five dropped packets. The zero drop scenario is included to show that all the protocols succeed in transmitting the same number of packets when there are no losses. The other drop scenarios (shown in Figure 4) are included to compare the robustness of the protocols in handling multiple losses in a single window. The one to four drop scenarios are the same as presented in [5].

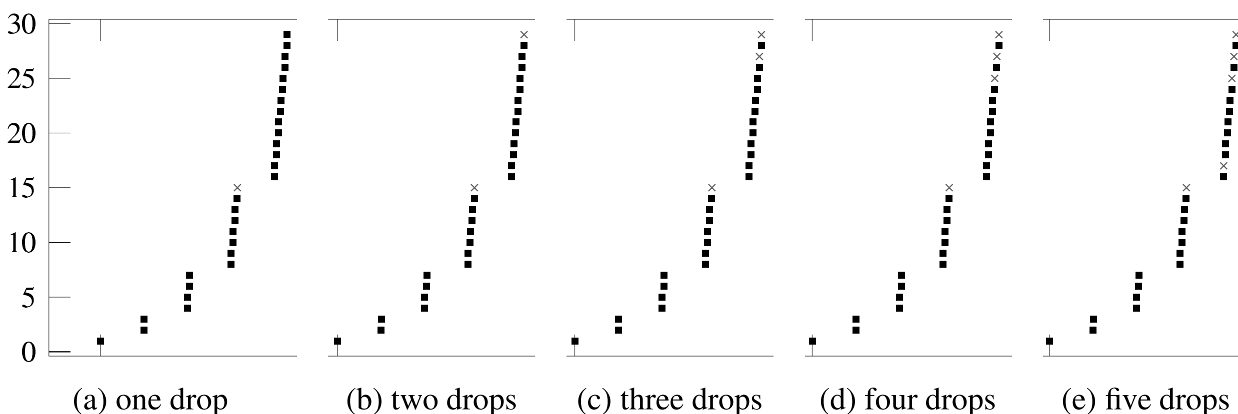

Figure 4: Drop scenarios

Figure 5 plots the number of data packets forwarded by the router (R1 in Figure 1) for each of the six loss scenarios. As expected, New-Reno SCTP significantly outperforms Original SCTP as the number of drops in a window increases. Since the modifications introduced in New-Reno SCTP come into effect only when there are multiple losses in a window, New-Reno SCTP and Original SCTP perform identically for the zero and one drop scenarios. Also, since Original SCTP is the only protocol shown in the graph which suffers multiple cwnd reductions in a window, it performs worse than the others.

We consider the results for New-Reno SCTP* to fairly compare New-Reno SCTP and the TCP

[5]The most recent version of FACK TCP is Rate-Halving TCP [9], which is supposed to provide even better performance. However, we learned through an email conversation with Matt Mathis that there are a couple of issues which currently make it unsuitable for deployment. Hence, we do not include Rate-Halving TCP in our comparisons.

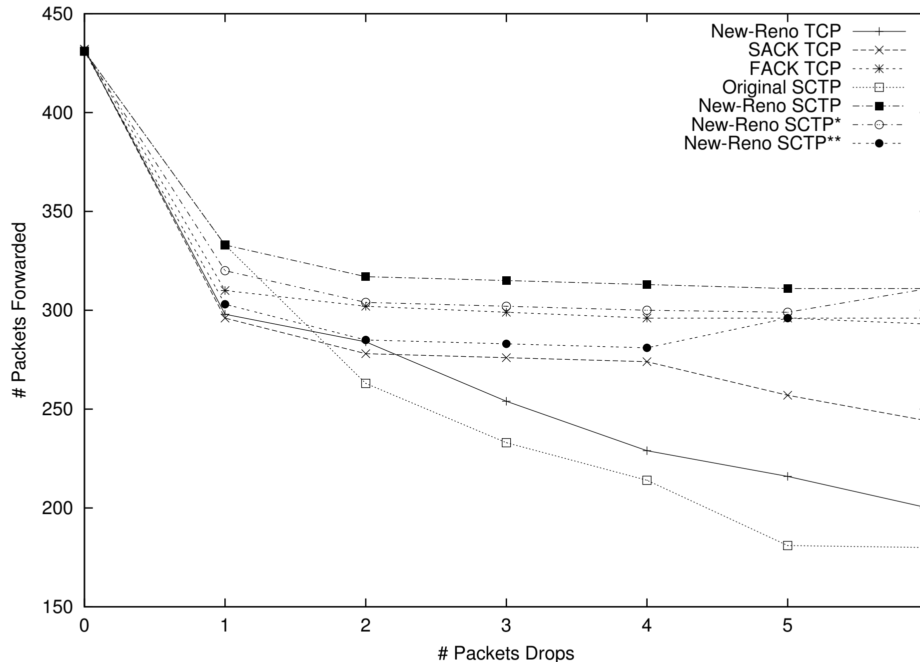

Figure 5: Simulation comparisons of protocols

variants. Without SACK information, New-Reno TCP can retransmit at most one dropped packet per round-trip time. Hence, New-Reno TCP performs worse than the SACK supporting protocols: SACK TCP, FACK TCP, and New-Reno SCTP*.

@@@ from this point on, these are just notes for the authors...

we need to decide whether we show these * and ** versions of SCTP. the * version performs closely with FACK, while the ** performs closely with SACK. we expected SCTP to perform as well as SACK, but just slightly better. also, the ** version is the most fair comparison (i think). FACK is known to be more aggressive than SACK, however we didn't expect it to be in the cases we have shown... especially the one drop. the FACK paper says that one drop should perform identical to SACK. also, it says that FACK shouldn't perform significantly better than SACK unless more than half the window is lost. but we have seen the ns-2 implemenation implement a behavior which is not in the paper at all. we have mailed the fack authors about it, but have received no response. so we'll just put into the paper exactly what ns-2 is doing with fack and say that we are discussing with the fack authors whether the behavior is correct. anyway, the behavior is as follows. when there is a fast retransmit, FACK reduces the cwnd to 1. Then for each ack (including dup acks) that arrives, the cwnd is increased. Since the ssthresh = 7, the first 6 acks increase the cwnd by 1. Then the others increase the cwnd by 1/cwnd. By the time fast recovery is exited, the cwnd is nearly 8. SACK doesn't increase the cwnd at all until after fast recovery is exited, so the cwnd is 7 upon exiting. New-Reno SCTP** increases the pba once for each acked fast retransmit; when exiting fast recovery, the cwnd=7 and pba is equal to the number of fast retransmits triggered. we need to think about what is a fair way to compare these and good way to explain the differences in a concise manner without confusing the reader.

make a statment that we are not concluding that the results represent how the protocols will perform for any drop scenario of the given number of losses. we are only showing how they perform in these scenarios.

the anomalous behavior seen with the New-Reno SCTP* (6 drops and up) and New-Reno SCTP** (5 drops and up) is explained by how pba is incremented. pba is incremented everytime a new cumack arrives. each time a new cumack arrives, the newly acked data is added to pba. so what ends up happening is that if the number of drops is 1 or 2 greater than the cwnd/2, then performance goes up. how? well u need to compare graphs, but this is what happens... if pba is high enough when fast recovery is exited, then it is possible for the cwnd to be incremented in the same window that fast recovery is exited in. if that happens, you get the anomalous behavior.

Let's take the ** case with 4 drops. The cwnd drops to floor(15/2)=7 when fast recovery begins (pba=0). Each new cumack in fast recovery increases the pba by 1. The ack for packet 35 brings the sender out of fast recovery and brings the pba to 5 (it acks packets 29 and 35). The next ack is for packet 36 and is the last ack for the current window. This ack brings pba to 6, but since cwnd=7, the cwnd is not incremented in this window.

Now let's take the ** case with 5 drops. Again the cwnd=7 when entering fast recovery. This time the ack for packet 34 brings the sender out of fast recovery and brings the pba to 6. There are two more acks in this window: the acks for packets 35 and 36. The ack for packet 35 brings the pba to 7, which increases the cwnd to 8. Thus, the anomalous behavior begins here!

This sounds bad, but really isn't a big deal because we are talking about independent losses. i.e., no burst losses. it is unlikely in a window of 15 to have more than 5 drops which are completely isolated. it may be more likely to have 3 bursts of 2 drops. anyway, this behavior only happens when congestion avoidance is done when cwnd = ssthresh. sctp does not do this anyway, so this anomalous behavior is not of concern. however, we should not leave it in the paper because it will draw attention.

6 Conclusion

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC2581, Internet Engineering Task Force (IETF), April 1999.

[2] UC Berkeley, LBL, USC/ISI, and Xerox Parc. ns-2 documentation and software, Version 2.1b8, 2001. http://www.isi.edu/nsnam/ns.

[3] A. Caro and J. Iyengar. ns-2 SCTP module, Version 3.2, December 2002. http://pel.cis.udel.edu.

[4] D. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17(1):1–14, June 1989.

[5] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. In *ACM Computer Communications Review*, pages 5–21, July 1996.

[6] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC2681, Internet Engineering Task Force (IETF), June 2000.

[7] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM 1988*, August 1988.

[8] M. Mathis and J. Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM 1996*, pages 281–291, Stanford, CA, August 1996.

[9] M. Mathis and J. Mahdavi. TCP Rate-Halving with Bounding Parameters. Technical report, December 1997. http://www.psc.edu/networking/papers/FACKnotes/current/.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC2018, Internet Engineering Task Force (IETF), October 1996.

[11] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[12] R. Stewart, L. Ong, I. Arias-Rodriguez, K. Poon, P. Conrad, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Implementer's Guide. draft-ietf-tsvwg-sctpimpguide-07.txt, Internet Draft (work in progress), Internet Engineering Task Force (IETF), October 2002.

[13] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison Wesley, New York, NY, 2001.

[14] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Proposed standard, RFC2960, Internet Engineering Task Force (IETF), October 2000.