

ANALYZING THE PERFORMANCE OF NEW TCP EXTENSIONS OVER  
SATELLITE LINKS

A Thesis Presented To

The Faculty of the

Fritz J. and Dolores H. Russ  
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Christopher Hayes

August 1997

THIS THESIS ENTITLED  
“ANALYZING THE PERFORMANCE OF NEW TCP EXTENSIONS OVER  
SATELLITE LINKS”

by Christopher Hayes

has been approved

for the School of Electrical Engineering and Computer Science  
and the Russ College of Engineering and Technology

---

Shawn D. Ostermann  
Assistant Professor of Computer Science

---

Warren K. Wray, Dean  
Fritz J. and Dolores H. Russ  
College of Engineering and Technology

## ACKNOWLEDGMENTS

I'd like to acknowledge the assistance of Dr. Shawn Ostermann, who provided me with thoughtful insights, interesting challenges, and a wonderful working environment. I would also like to thank my friends in the Internetworking Research Group, who put up with many of my wild ideas. I thank Melissa for putting up with my long hours. For their generous funding, without which this research would not have been possible, I thank NASA and the School and Electrical Engineering and Computer Science. I also thank Mark for introducing me to the IRG and the field of computer networking. I especially thank my family, Oscar, Janet, Patsy, and Cheryl for all of their love and support through the years. Without my family this all would not have been possible.

DISCARD THIS PAGE

## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	vi
LIST OF FIGURES . . . . .	vii
1. INTRODUCTION . . . . .	1
1.1 Motivation for This Thesis . . . . .	1
1.2 Internetworking Overview . . . . .	1
2. TCP . . . . .	3
2.1 Acknowledgments . . . . .	3
2.2 Timers . . . . .	4
2.3 Congestion Control . . . . .	5
2.4 Slow Start . . . . .	5
2.5 Congestion Avoidance . . . . .	5
2.6 Fast Retransmit and Fast Recovery . . . . .	6
2.7 Reno . . . . .	7
2.8 TCP Extensions . . . . .	7
2.8.1 Big Windows . . . . .	7
2.8.2 New Reno . . . . .	8
2.8.3 TCP with Selective Acknowledgments . . . . .	9
2.8.4 Forward Acknowledgments . . . . .	11
2.9 Satellite Problems with Standard TCP . . . . .	12
2.9.1 Window Limitations . . . . .	13
2.9.2 Loss Recovery . . . . .	13
3. RESULTS . . . . .	16
3.1 TCP Versions Tested . . . . .	16
3.2 Experimental Setup . . . . .	17
3.2.1 Hardware Emulator . . . . .	17

	Page
3.3 Drop Experiments . . . . .	18
3.3.1 Single Drop . . . . .	20
3.3.2 Two Drops . . . . .	20
3.3.3 Four Drops . . . . .	27
3.4 File Transfer Experiments . . . . .	31
3.5 Non-Congestion Related Loss . . . . .	34
4. CONCLUSIONS . . . . .	38
4.1 Conclusions . . . . .	38
4.2 Further Research . . . . .	39
BIBLIOGRAPHY . . . . .	41
 <b>APPENDIX</b>	
A. HOW TO READ A TCP TIME SEQUENCE GRAPH . . . . .	43
ABSTRACT . . . . .	46

## LIST OF TABLES

Table	Page
2.1 ACK Example . . . . .	4
2.2 SACK Example . . . . .	10
2.3 FACK Example . . . . .	12
3.1 2 Drop Example . . . . .	21
3.2 4 Drop Example . . . . .	28
3.3 File Transfer Statistics . . . . .	33
3.4 Bit Error Tests Statistics . . . . .	36

## LIST OF FIGURES

Figure	Page
2.1 Window Scale Option . . . . .	7
2.2 Slow Start Comparisons . . . . .	14
3.1 Satellite Setup . . . . .	17
3.2 Hardware Setup . . . . .	18
3.3 ACTS vs Hardware Emulator . . . . .	19
3.4 Reno-1 drop . . . . .	21
3.5 FACK-1 drop . . . . .	22
3.6 Reno-2 drops . . . . .	23
3.7 New Reno-2 drops . . . . .	24
3.8 SACK-2 drops . . . . .	25
3.9 FACK-2 drops . . . . .	26
3.10 Reno-4 drops . . . . .	29
3.11 New Reno-4 drops . . . . .	30
3.12 SACK-4 drops . . . . .	31
3.13 FACK-4 drops . . . . .	32
3.14 Test Comparisons . . . . .	33
3.15 Bit Error Test Comparisons . . . . .	36
A.1 TCP Time Sequence Graph Example . . . . .	45

## 1. INTRODUCTION

This thesis investigates the performance of proposed *Transmission Control Protocol*(TCP)[Ste97][Com95] implementation changes over long delay high bandwidth satellite links. This thesis investigates the problems of standard Reno TCP and discusses the benefits of the proposed changes in a satellite environment.

### 1.1 Motivation for This Thesis

It has been shown in Kruse[Kru95] that long delay satellite channels suffer poor throughput using the Transmission Control Protocol, TCP. Performance is limited by the delay inherent in geosynchronous satellites and the probability of bit errors found in wireless systems[Kru95]. Several changes have been proposed that could help TCP performance over long delay paths. These changes include big windows and PAWS proposed in [JBB92], Selective Acknowledgments(SACK), proposed in [MMFR96a], Hoe's fast retransmit modifications [Hoe96], and Mathis and Mahdavi's Forward Acknowledgments(FAACK)[MMFR96b][MM96b].

### 1.2 Internetworking Overview

Communication protocols are often *layered* to provide added functionality. The delivery and control information for a protocol is contained in a *header* placed in the packet. A higher level protocol is *encapsulated* in the data portion of a lower level protocol. A single packet may contain several layers of protocol information, each playing a role in the delivery of the packet.

The *Internet Protocol*, IP, is the fundamental protocol for the Internet. IP is an unreliable and connectionless packet delivery protocol. IP is unreliable because the arrival of the data to the destination is not guaranteed and connectionless because each packet may take a different path to its destination, and the packets may arrive in any order. The delivery of packets is based on IP address assigned to a host computer.

## 2. TCP

TCP is a connection oriented sliding window protocol that is implemented on top of the *Internet Protocol*. TCP provides a reliable, byte streaming, full duplex connection[Com95]. TCP is connection oriented because two hosts must setup and establish a connection before any data is transmitted. Applications using TCP will receive the data in the correct order as sent by the sending host. A sliding window protocol places a fixed sized window over a sequence of segments and transmits all the segments that lie inside of the window[Com95]. Each segment has a sequence number to identify its order in the stream. A segment is unacknowledged if it has been transmitted and no acknowledgment has been received for the segment. The sender may transmit the window size in unacknowledged segments. Once an *acknowledgment*, ACK, has been received for a segment, the window will advance and new segments can be transmitted.

### 2.1 Acknowledgments

ACKs use the sequence numbers to report the next in-order segment the receiver expects to receive. These sequence numbers are based on the position of the segment in the byte stream. If segment  $X$  is 512 bytes long, the next sequence number will be  $X + 512$ . For simplicity, this paper will refer to the segment number rather than the sequence number in the byte stream. TCP has a cumulative acknowledgment scheme because each ACK covers all the in-order data received. If an out-of-order segment, a segment that does not contain the next expected sequence number, arrives, a duplicate ACK will be sent reporting the sequence number the receiver expected to receive. This indicates that a segment has arrived but cannot be used by the

receiver until the next in-order segment is received. Out-of-order packets are held in the receiver's buffer until used but may be discarded if memory is needed. Table 2.1 demonstrates how a receiver ACKs incoming segments using segment numbers.

Time Unit	Segment Received	ACK Sent	Comments
1	3	4	
2	4	5	
3	6	5	Duplicate ACK
4	7	5	Duplicate ACK
5	5	8	

Table 2.1 TCP Cumulative ACK Example

This figure shows TCP's cumulative acknowledgment scheme. Each segment the receiver receives generates a cumulative ACK reporting the next in-order sequence number the receiver expects to receive

## 2.2 Timers

TCP uses the incoming ACKs to calculate estimated round trip times[Com95]. TCP uses these round trip times to deduce the time period before retransmitting a segment. If a segment has not been received when this time period elapses, a *retransmission timeout*, RTO, occurs. The timeout causes the connection to enter slow start and the connection begins transmitting at the first unacknowledged segment. Our version of NetBSD uses a timer with a 500ms granularity.

### 2.3 Congestion Control

TCP's congestion control algorithms govern how TCP avoids and reacts to congestion. These algorithms also dictate the performance of TCP when loss occurs. The two major variables involved in TCP congestion control are the *congestion window*, *cwnd*, and the *slow start threshold*, *ssthresh*. When setting up a new connection, the *cwnd* is initialized to one *maximum segment size*, *MSS*, and the *ssthresh* is initialized to 64 KB<sup>1</sup>. These variables are used to control the amount of data sent by the congestion control algorithms, slow start, congestion avoidance, and fast recovery. Slow start increases the *cwnd*, fast recovery adjusts the *cwnd* in the event of loss, and congestion avoidance slowly increases the *cwnd*.

### 2.4 Slow Start

Slow start allows TCP to probe network conditions by slowly increasing the amount of data injected into the network. The slow start algorithm uses the *congestion window*, *cwnd*, to control the flow of data. The *cwnd* is initialized to one segment, usually 512 bytes. Slow start's principle is simple, for every acknowledgment received, add one segment to the *cwnd*. The sender can send the minimum of the congestion window or *ssthresh*. *Ssthresh* is initialized to the receiver's advertising window. When the *cwnd* is greater than or equal to *ssthresh* the connection enters the congestion avoidance phase. If the capacity of the link is reached before *cwnd* is greater than *ssthresh*, a gateway will signal congestion by dropping segments and TCP will enter the fast retransmit phase after three duplicate ACKs.

### 2.5 Congestion Avoidance

A bottleneck link can occur when a large pipe connects to a small pipe. Congestion occurs when the volume of incoming segments exceeds the buffer space of the

---

<sup>1</sup>b refers to a bit, B refers to a byte, KB refers to 1024 bytes, MB refers to 1024 KB, Mb refer to 1024 kilo bits

gateway. The gateway will continue to drop segments until buffer space is available. These dropped segments signal congestion to the TCP connection through duplicate ACKs or a *retransmission timeout*. When congestion occurs, the connection performs recovery then enters congestion avoidance. If a retransmission timeout occurs, *cwnd* is set to one MSS. When  $cwnd > ssthresh$  the slow start phase ends and congestion avoidance takes control. Congestion avoidance advances *cwnd* by

$$\text{segment size} * \frac{\text{segment size}}{\text{cwnd}}$$

While in the congestion avoidance phase, the *cwnd* will never be incremented more than one segment per *round trip time*, RTT, if all the segments in the window have been acknowledged[Com95]. This is a linear growth rate compared to slow start's exponential growth rate.

## 2.6 Fast Retransmit and Fast Recovery

A retransmission timeout detects a lost segment in TCP. When a timeout occurs, TCP will drop back into the slow start phase and retransmit any lost segments. This may cause the unnecessary retransmission of received out-of-order segments being held in the receiver's buffer. Jacobson[Jac90] proposed a fast retransmit phase to avoid the long wait to retransmit associated with the retransmit timer and the return to slow start. Fast retransmit occurs when three duplicate ACKs trigger the retransmission of a lost segment. Fast retransmit works under the principle that most out-of-order segments will appear after one or two segments. After three duplicate ACKs, it is safe to assume the segment has been lost. If the sender is receiving duplicate ACKs, then data is getting through the network and returning to slow start is unnecessary so fast recovery is performed. After three duplicate ACKs, the sender will set *ssthresh* to one half of the *cwnd* rounded down to closest multiple of the maximum segment size(MSS). *Cwnd* is set equal to *ssthresh* plus three MSS(one half the *cwnd* plus the three received segments that have left the network). The *cwnd* will be update by one MSS for each additional duplicate ACK that is received. If the *cwnd*

allows, the sender transmits a new segment into the network. These new segments will trigger additional ACKs which may allow for another fast retransmit phase if additional segments have been lost. When a new piece of data is acknowledged, the congestion window is set equal to `ssthresh`. The sender then enters the congestion avoidance phase.

## 2.7 Reno

TCP Reno is the base TCP for our experiments. Reno is defined as a TCP containing the algorithms outlined in RFC 2001[Ste97]. These algorithms include slow start, fast retransmit, fast recovery, and congestion avoidance.

## 2.8 TCP Extensions

### 2.8.1 Big Windows

The maximum window size for standard Reno TCP is 65535 bytes. A connection must use the window scale option defined in RFC 1323[JBB92] to increase the window size above 65535. The window scale option is a three byte option that may be sent in the SYN segment.

Kind	Length	shift.cnt
8	8	8
3	3	scale factor

Figure 2.1 Window Scale Option

Figure showing the layout of the window scale option.

The windows scale option serves a dual purpose, it signals that TCP is ready to both send and receive window scaling, and communicates the scale factor to apply to

the window. The scale factor is logarithmically encoded as an integer power of two. A power of two scaling factor allows the scaling to be easily implemented with binary shift operators. Both sides must send the window scale option for window scaling to be enabled.

In order to preserve the integrity of new segments on the network, the window scale option must not exceed  $2^{30}$  (shift.cnt=14). A data segment is considered old if the segment is not within  $2^{31}$  bytes of the left window edge [JBB92]. Any data considered old is discarded by the receiver. Since the maximum allowable window is  $2^{30}$ , standard TCP window  $65535 = 2^{16} - 1 + \text{max scale size} = 2^{14}$ , this allows a 1 GB window.

### 2.8.2 New Reno

New Reno is a proposal from Hoe [Hoe96] that requires only sender side changes which allows TCP to recover from multiple losses in a window of data. Our New Reno implementation is based on the implementation presented in the LBL ns simulator in Fall and Floyd [FF96].

After three duplicate ACKs, TCP performs standard fast retransmit, with the highest unacknowledged segment recorded in *snd\_high*. After the segment is retransmitted, the connection does not exit fast retransmit. The cwnd is increased by one maximum segment size for each duplicate ACK received. If the cwnd exceeds the amount of data left in the network, new segments are sent. A partial ACK is an ACK that does not cover all the segments through *snd\_high*. If a partial ACK is received, the sender assumes that segment is lost and retransmits the segment covered by the partial ACK. This is repeated until the fast retransmit phase exits, the ACK is greater than or equal to *snd\_high*, or a retransmission timeout occurs. When the fast retransmit phase ends, the inflated cwnd is set to ssthresh and TCP enters congestion avoidance. This scheme guarantees recovery of at least one segment per round trip time.

### 2.8.3 TCP with Selective Acknowledgments

TCP with Selective Acknowledgments(SACK), specified in RFC 2018[MMFR96a], is designed to provide information about the loss of multiple segments in a window of data. The data receiver uses the SACK option to inform the sender of all successfully received segments that have not been cumulatively acknowledged. The sender uses the SACKed information to retransmit only the segments that were lost.

The sender and the receiver must run an implementation of TCP with SACK to use the SACK option. SACK compatibility is verified in TCP's three way handshake with the Sack-Permitted option. The Sack-Permitted option maintains compatibility with older versions of TCP. TCP with SACK sends the Sack-Permitted in the SYN segment at the initialization of a connection. The receiver must answer with a Sack-Permitted option in the answering SYN of the three way handshake. If the Sack-Permitted option is not found in the answering SYN, the SACK option will not be used during the connection. If the answering SYN segment contains the Sack-Permitted option, then both connections are able to send SACK options on that connection.

The SACK option is sent by a data receiver to inform the data sender of non-contiguous blocks of data that have been queued in the receiver's receive buffer within the window. The SACK option should be included in all ACKs that do not acknowledge the highest sequence number received by the receiver. A SACK block must include the most recently received data in the first SACK block. The SACK block data allows the sender to retransmit only the lost segments in the fast retransmit and fast recovery phases of TCP. The TCP header is limited to 64 bytes, depending on the number of TCP options one, two, or three SACK blocks may fit in a segment.

#### 2.8.3.1 Retransmission Strategy

The listing of received outstanding data gathered from the receiver's SACK blocks is stored in the *scoreboard* data structure. The sender can use the *scoreboard* to locate segments that have been lost and need to be retransmitted.

Triggering Segment	ACK	First Block		2nd Block		3rd Block	
		Left Edge	Right Edge	Left Edge	Light Edge	Left Edge	Right Edge
3000	3500						
3500	(lost)						
4000	3500	4000	4500				
4500	(lost)						
5000	3500	5000	5500	4000	4500		
5500	(lost)						
6000	3500	6000	6500	5000	5500	4000	4500
6500	(lost)						

Table 2.2 SACK Example

Example of the SACK block information sent when multiple segments are lost.

A retransmission strategy using SACK information is outlined in Fall and Floyd [FF96]. This new strategy is designed to solve the performance problems associated with multiple losses in a window of data. If a connection has a single segment loss event, then standard Reno recovery algorithms are used. However, if TCP with SACK suffers a multiple segment drop in a single window of data, the new recovery scheme is used that uses the SACKed information maintained in the scoreboard. The sender will not exit the *fast recovery phase* until an acknowledgment is received that covers all the outstanding data present when *fast recovery* began. This new scheme maintains an estimated amount of data on the path in a *pipe* variable. *Pipe* is initialized to the *cwnd* and decremented by a segment size for each duplicate ACK that is received and two segments for each partial ACK that is received. *Pipe* is decremented twice for Partial ACKs because SACK assumes two segments have left the network, the dropped segment and the retransmitted segment. The sender is allowed to send new data or retransmit data if the *pipe* falls below *cwnd*. When

retransmitting data, the SACK sender will retransmit the next missing data segment maintained in the scoreboard. As in TCP Reno, a retransmission timer is maintained and if a retransmitted segment is lost, then the retransmission timer will expire and the connection will enter slow start.

#### 2.8.4 Forward Acknowledgments

TCP with forward acknowledgment (FACK) was proposed in Mathis[MM96a]. FACK was developed to make better use of the scoreboard information provided by selective acknowledgments. The current implementation of FACK is FACK with rate-halving[MM96b]. TCP with FACK separates the recovery algorithm from the retransmission algorithm. The rate-halving scheme was first proposed by Hoe[Hoe95]. Rate-halving is an attempt to better estimate the proper window size after a loss event caused by congestion. Also, FACK moves the fast retransmit threshold into the *scoreboard* and applies it to every hole in the scoreboard. A segment in the scoreboard must have three duplicate SACKs before the segment can be retransmitted. FACK also provides a new mechanism to detect the loss of a retransmitted segment.

The FACK algorithms engage after receiving three duplicate *SACK blocks*. The missing segment indicated by the scoreboard is retransmitted. The connection will perform rate-halving for one round trip time after the retransmission. In rate-halving, for every two ACKs, check the *scoreboard* for any hole that equals or exceeds the retransmit threshold and retransmit that segment. If no segment exceeds the retransmit threshold, then send new data if the advertising window allows. Transmitting a segment every two ACKs brings the window to half of the amount of data actually being held on the network,

$$snd\_wnd = (snd\_wnd - loss)/2. \quad (2.1)$$

During recovery, a retransmission loss is detected by advancing the *snd\_fack* variable. If *snd\_fack* advances beyond *snd\_next*, then later new data has arrived and the lost segment is retransmitted again. *Snd\_fack* tracks the SACK block information, if SACK

Time Unit	Segment Sent	ACK Received	Scoreboard Holes	Comments
1	10-21	10		10, 13, and 15 are lost
2		10	10	Duplicate ACK
3		10	10	Duplicate ACK
4	10	10	10 13	Duplicate ACK, Enter rate-halving
5		10	10 13 15	Duplicate ACK
6	13	10	10 13 15	Duplicate ACK
7		10	10 13 15	Duplicate ACK
8	15	10	10 13 15	Duplicate ACK
9		10	10 13 15	Duplicate ACK
10	22	10	10 13 15	Duplicate ACK
11	23	13	13 15	Exit rate-halving
12	24	15	15	
13		22		Exit recovery

Table 2.3 FACK TCP Example

This figure shows FACK TCP's rate-halving recovery scheme. For one RTT after the first retransmitted segment, a segment segment is retransmitted or a new segment is sent for every two duplicate ACKs received.

blocks are received for segments sent after the retransmission, then the retransmitted segment is assumed lost.

## 2.9 Satellite Problems with Standard TCP

A geosynchronous satellite must maintain a fixed position over the earth in order to maintain contact with its earth stations at all times. A satellite achieves a stationary orbit at 35,784 km above the earth where the satellite's period of rotation equals the earth's period of rotation[Sta94]. The long distance involved produces a 240 to 300ms

propagation delay between the transmission from the sending ground station, up to the satellite, and down to a receiving ground station. This high delay degrades TCP performance over satellites.

### 2.9.1 Window Limitations

As specified in Section 2.8.1, the maximum window size for standard TCP is 65535 bytes. A round trip time is the time interval between sending a segment and receiving an acknowledgment for it [JK88]. The maximum throughput of a TCP connection is bounded by the window size and round trip time defined by the formula in Postel [Pos81].

$$throughput_{max} = \frac{\text{receive buffer size}}{\text{round trip time}} \quad (2.2)$$

The average round trip time, RTT, for a satellite connection is approximately 585ms. Thus a typical geosynchronous satellite using a standard Reno TCP window is limited to throughput:

$$throughput_{max(satellite)} = \frac{64Kbytes}{585ms} \approx 112,000 \frac{bytes}{second} \approx 896,000 \frac{bits}{second}. \quad (2.3)$$

A full T1 satellite channel is 1,536,000 bits per second. Therefore a standard TCP connection with a window of 65535 bytes running at full throughput of 896,000 bps cannot fully utilize all of the available bandwidth of the satellite channel. RFC 1323 provides an option that allows a TCP connection to open its window beyond 65535 bytes. A window set to approximately 110 KB enables a connection to utilize the full bandwidth of the T1 satellite channel

$$throughput_{max(satellite)} = \frac{110Kbytes}{585ms} \approx 192,547 \frac{bytes}{second} \approx 1,540,376 \frac{bits}{second}. \quad (2.4)$$

### 2.9.2 Loss Recovery

Allman [All97] investigates the problems of satellite links in the slow start phase and shows that time spent in the slow start phase represents wasted capacity. The long delay of satellite links requires TCP to spend a longer period of time in slow start

than a terrestrial link. Figure 2.2 examines the performance difference of a satellite link compared to a terrestrial link with a 128 segment window and 512 byte segments.

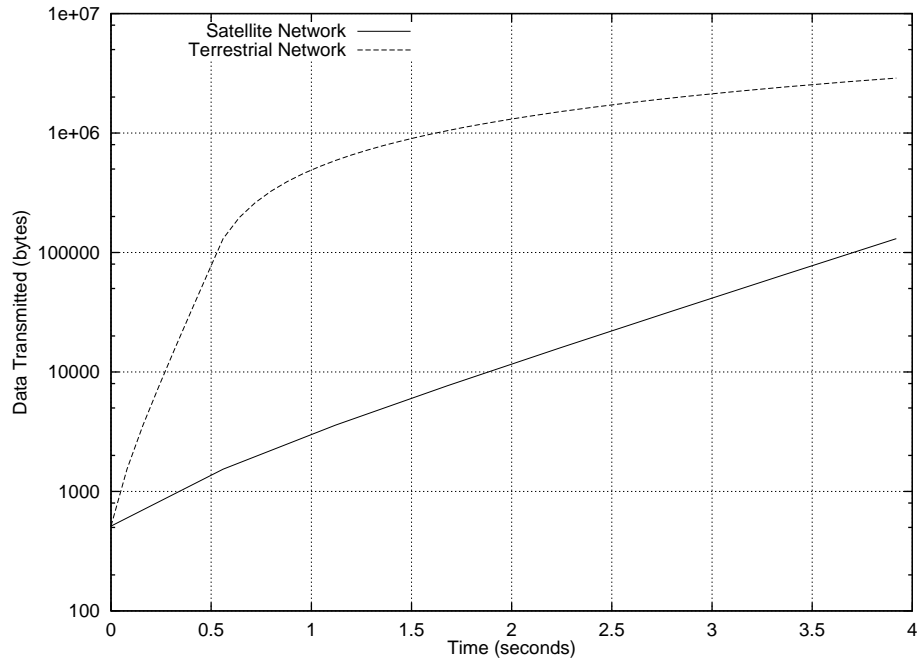


Figure 2.2 Slow Start Comparisons

This figure shows a mathematical model of the total amount of data sent over a satellite network and a terrestrial network as a function of time. Both connections start at one segment and use the slow start algorithm to increase the window size. The amount of time shown represents the time it takes the TCP using the satellite connection to reach the maximum window size of 128 segments.

Standard TCP suffers a performance problem when faced with multiple drops in a window of data. When multiple drops occur, three duplicate ACKs will trigger a fast retransmit. It will take one round trip time for the retransmitted segment to be ACKed. In that round trip time, all of the remaining segments will have left the network generating duplicate ACKs. When the ACK returns for the retransmitted segment, there are no longer enough duplicate ACKs remaining to trigger another fast retransmit. The connection is forced to wait for a timeout and return to slow start. Time spent in slow start represents wasted capacity. Figure 2.2 shows the amount of

capacity wasted in slow start over a satellite link compared to a terrestrial link. This wasted capacity makes returning to slow start undesirable in satellite channels.

The new TCP extensions are designed to solve the timeout problem when multiple segments are dropped in a window of data. Duplicate ACKs signal that data is still getting through the network. In response to a slightly congested network, TCP should back off by halving its window. This is the principle behind fast retransmit and fast recovery. The new extensions extend TCP to recover from multiple drops in a window of data. This is advantageous in the satellite environment where returning to slow start is costly to performance.

### 3. RESULTS

This research investigates the performance of new TCP extensions over satellite links. We conducted experiments to test the TCP extensions under various loss events to gain an understanding of how each extension performs. Traces were generated for each experiment to determine the amount of extra loss required to recover and the recovery window size. Throughput results for 200 KB, 1 MB, and 5 MB files were gathered using FTP over a simulated ACTS satellite network. These tests provide information on the performance of the new extensions under a realistic satellite environment with moderate congestion. The last section examines the performance of the new extensions when faced with non-congestion related loss. Such loss occurs when bit errors are introduced into the segments by the satellite link.

#### 3.1 TCP Versions Tested

The experiments described in this paper used the following TCP variants, as previously described in Chapter 2:

- Reno

Reno, as standardized in RFC 2001[Ste97], is the base TCP for this paper's experiments. Reno is the standard version of TCP found on most UNIX systems. Reno contains fast retransmit, fast recovery, congestion avoidance, and slow start.

- New Reno

New Reno is an extension of Reno that exits fast retransmit when the highest unacknowledged segment is ACKed. New Reno uses ACKs covering segments under the highest unacknowledged segment to trigger retransmits.

- SACK

SACK uses selective acknowledgments in its extensions of the fast retransmit and fast recovery algorithms. SACK keeps track of the amount of data on the network and retransmits or sends new data when the amount of data on the network is less than the congestion window.

- FACK

FACK uses the SACK information but is a departure from standard TCP. FACK separates the recovery and congestion control algorithms. FACK uses rate-halving to inject new or retransmitted data into the network. The amount of data sent in rate-halving determines the congestion window after recovery.

### 3.2 Experimental Setup

The experimental setup for the ACTS satellite systems consisted of two INTEL 486 PC's running the NetBSD 1.1 operating system. Each PC was connected to a Cisco 2500 router which routed the packets through the ACTS satellite system.



Figure 3.1 Satellite Setup

This figure shows the network setup for the experiments involving the ACTS satellite system.

#### 3.2.1 Hardware Emulator

Due to the high demand for satellite time, most of our tests were conducted using an emulator, the “Data Link Simulator” made by Testlink Corporation. The hardware emulator allowed us to emulate the delay and link capacity of the ACTS satellite channel. The test setup for the hardware emulator consisted of two INTEL

486 PCs running the NetBSD 1.1 operating system. The PC's were connected to Cisco 2500 routers via 10 Mb ethernet. The routers were connected to the Datalink emulator. The emulator allows the channel capacity and delay to be configured. The line capacity for our test was 768,000 bps(half T1), with a 290ms delay in each direction providing round trip times equivalent to the ACTS experiments. The Cisco routers were configured with 100 packet queues, 60 incoming and 40 outgoing packets. Figure 3.2 shows our test equipment layout.

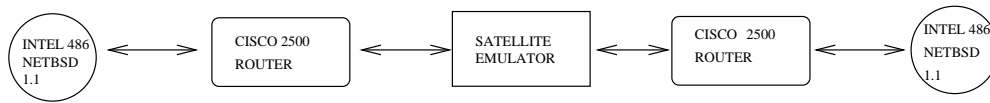


Figure 3.2 Hardware Setup

This figure shows the network setup for experiments involving the Data Link Simulator.

The lack of satellite time required most of our tests to be conducted over the emulator. The emulator must provide results close to the ACTS satellite system for the results to be useful. We transferred 200 KB, 1 MB, and 5 MB files over the emulator and the ACTS satellite system to compare the closeness of the results. We chose those file sizes because they were used in the various file size experiments and would provide a benchmark for the calibration of those experiments. We conducted the tests using the three files sizes with a TCP Reno connection. Figure 3.3 shows that the results acquired with the hardware emulator are very close to the results acquired over the ACTS satellite system. We concluded from the tests that the emulator provides a valid emulation of the ACTS satellite network for our experiments.

### 3.3 Drop Experiments

The first step in understanding if the new TCP extensions perform better is to understand how the new extensions work. We conducted the drop tests to gain a better understanding of how each new recovery mechanism worked under varying degrees of

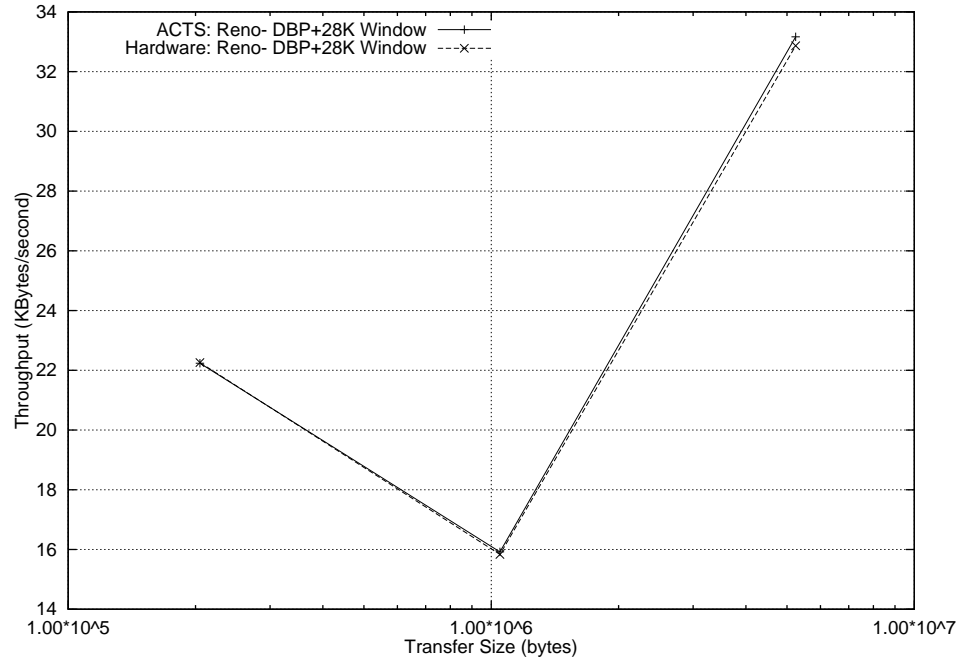


Figure 3.3 ACTS vs Hardware Emulator

This figure shows the emulator provides a valid emulation of the ACTS satellite network. We obtained the results by transferring 200 KB, 1 MB, and 5 MB files using the File Transfer Protocol, FTP, and Reno TCP.

loss. The effectiveness of Reno, New Reno, SACK, and FACK is investigated using one, two, and four drop tests. A one drop test is the optimal scenario for the Reno recovery algorithms. New Reno and SACK perform identical to Reno in a one drop scenario. The performance of FACK with rate-halving will be investigated under one drop as it does not perform fast retransmit and fast recovery.

A two drop test provides Reno with a minimal chance for recovery. This is the minimum number of multiple drops required to illustrate Reno's recovery problems when suffering multiple drops.

The four drop test gives Reno no chance for recovery and illustrates the smoother recovery of the new TCP extensions. All data for the drop experiments was collected using the `tcpdump`<sup>1</sup> packet capture utility. Time sequence graphs were generated

<sup>1</sup>Available at <http://www-nrg.ee.lbl.gov>

and analyzed with the `tcptrace`<sup>2</sup> utility. Appendix A describes how to read the time sequence graphs produced by `tcptrace`.

### 3.3.1 Single Drop

Figure 3.4 demonstrates the recovery of Reno from a single loss segment. Traces were not included for New Reno and SACK because they perform identical to Reno. Reno, SACK, and New Reno perform fast retransmit and fast recovery to recover from a single loss. Figure 3.5 illustrates FACKs recovery from a single loss when unable to inject new data into the network because the loss event occurred when the receiver's window is full, the receiver's window of unacknowledged data is on the network. FACK cannot inject any data into the network other than the single retransmitted segment so the `cwnd` is opened to one MSS. This returns the connection to slow start resulting in degraded performance.

### 3.3.2 Two Drops

The two drop experiments demonstrate the effectiveness of the new TCP extensions with a two segment loss event. The loss event will occur when the receiver's window is full, preventing the sender from injecting any new data into the network. Table 3.1 compares the number of losses, window sizes, recovery events needed for recovery.

Figure 3.6 illustrates the performance problems of Reno TCP with multiple drops in a window of data. The loss event occurs when the connection fills the bottleneck router queue forcing the router to drop segments. The out of order segments will generate duplicate ACKs. The third duplicate ACK triggers the fast retransmit of the first segment. The fast recovery algorithm inflates the congestion window but the sender has run up against the receiver's advertised window and cannot send new data. Without new data, the connection cannot generate enough duplicate ACKs to

---

<sup>2</sup>Available at <http://jarok.cs.ohiou.edu>

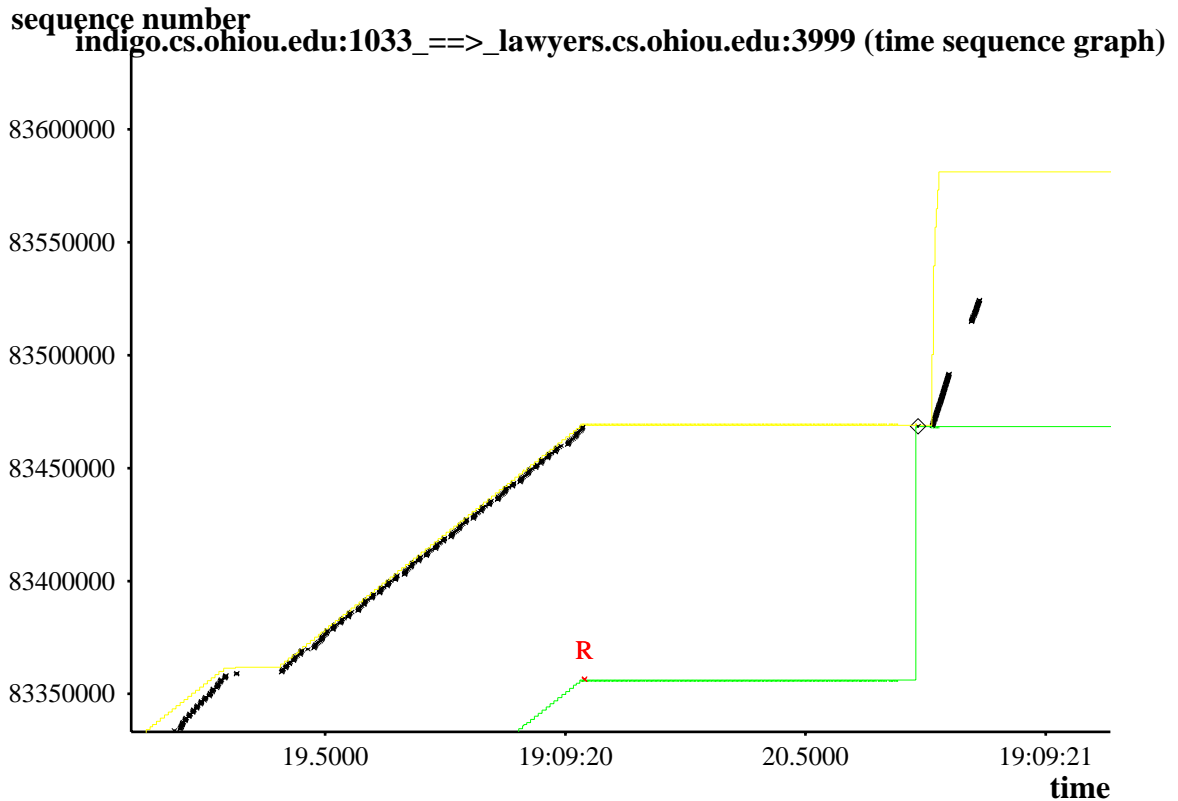


Figure 3.4 TCP Reno with 1 Drop

This time sequence graph shows TCP Reno's performance with a one drop loss event. Reno recovers from this loss event using Fast Retransmit. Upon Recovery, Reno halves the congestion window and enters congestion avoidance. SACK and New Reno perform identical to Reno under a one drop loss event.

TCP	Initial window in bytes	Recovery window in bytes	Rexmtd. segs.	Secondary loss event	Slow Start	Throughput in Bps
Reno	168 * 512	1 * 512	2	NO	YES	29018
New Reno	168 * 512	42 * 512	12	YES	NO	29957
SACK	168 * 512	42 * 512	6	YES	NO	36359
FAK	168 * 512	7 * 512	2	NO	NO	32591

Table 3.1 Algorithm Comparisons with Two Drops

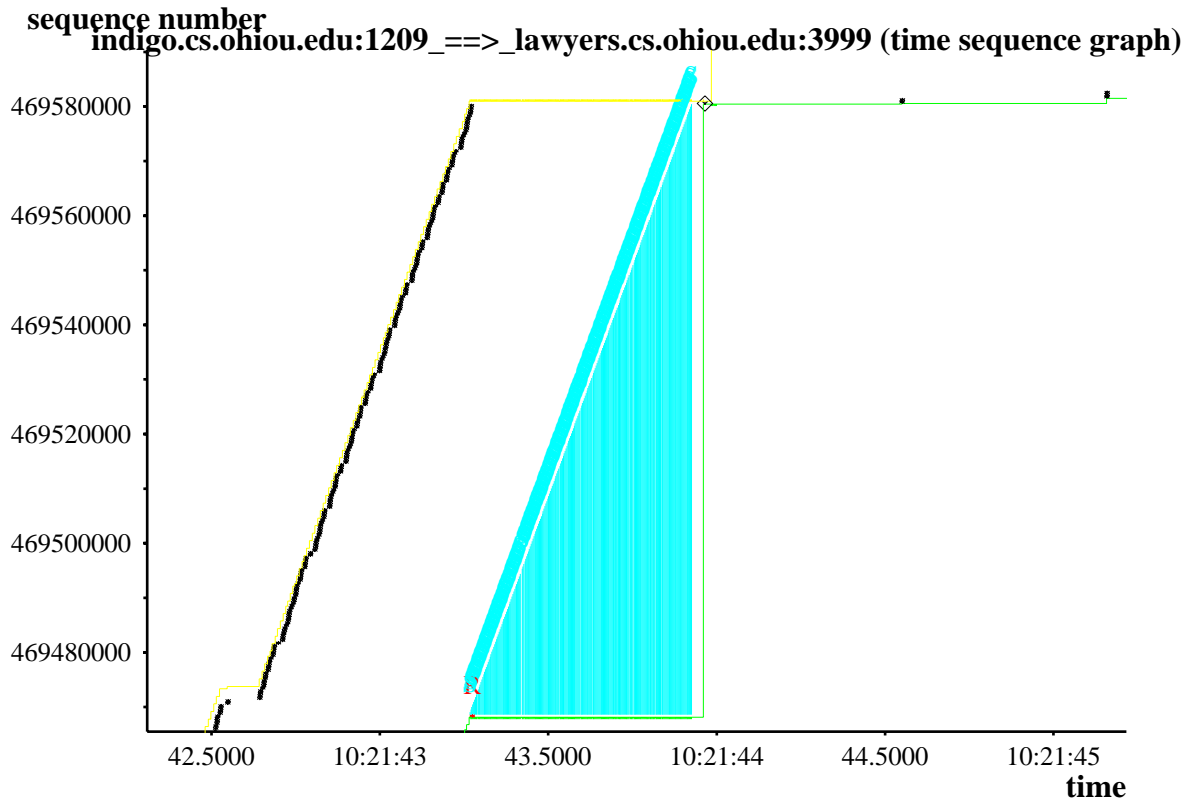


Figure 3.5 TCP with FACK with 1 Drop

This time sequence graph shows FACK TCP's performance with a one drop loss event. FACK recovers from the loss event using rate-halving. FACK cannot inject new data into the network during rate-halving because the receiver's advertised window of unacknowledged data is on the network. Upon Recovery, FACK's recovery congestion window is set to the one retransmitted segment and enters slow start.

cause the fast retransmit of the next lost segment and a retransmit timeout occurs returning the connection to slow start.

New Reno solves the performance problems of TCP Reno with a few sender side changes. Figure 3.7 illustrates the recovery mechanism of New Reno. The loss event generates three duplicate ACKs and triggers the fast retransmit of the first segment. The highest segment sent is also recorded in the variable *snd\_high*. The connection does not exit the fast retransmit phase until the ACK is greater than or equal to *snd\_high*. Every partial ACK, an ACK less than *snd\_high*, triggers the retransmission

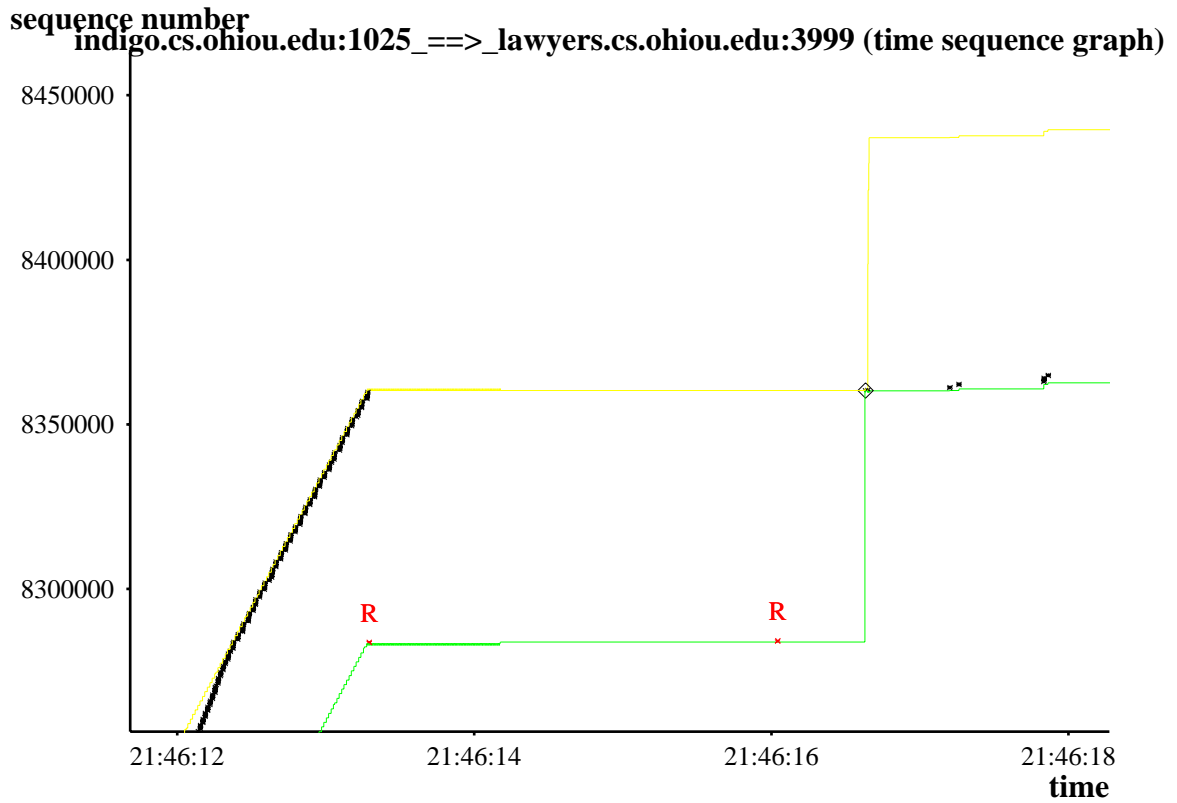


Figure 3.6 Standard TCP Reno with 2 Drops

This time sequence graph shows TCP Reno's performance with a two drop loss event. Reno cannot use fast retransmit to retransmit the second segment because all the duplicate ACKs have drained from the network. A retransmission timeout occurs and the TCP connection enters slow start.

of another segment. This guarantees the recovery of at least one packet per round trip time. When an ACK covering *snd\_high* returns the connection exits the fast retransmit phase and reduces *cwnd* by half. Using the new recovery mechanism, New Reno recovers from the first loss event in two RTTs. The connection then sends the full *cwnd* worth of data onto the network. This burst overruns the router buffer and causes another loss event. The *cwnd* will finally settle at  $\frac{\text{original cwnd}}{4}$  after the second loss event.

Figure 3.8 is an example of the Fall and Floyd[FF96] recovery mechanism that utilizes the SACK block information stored in the *scoreboard*. When three duplicate

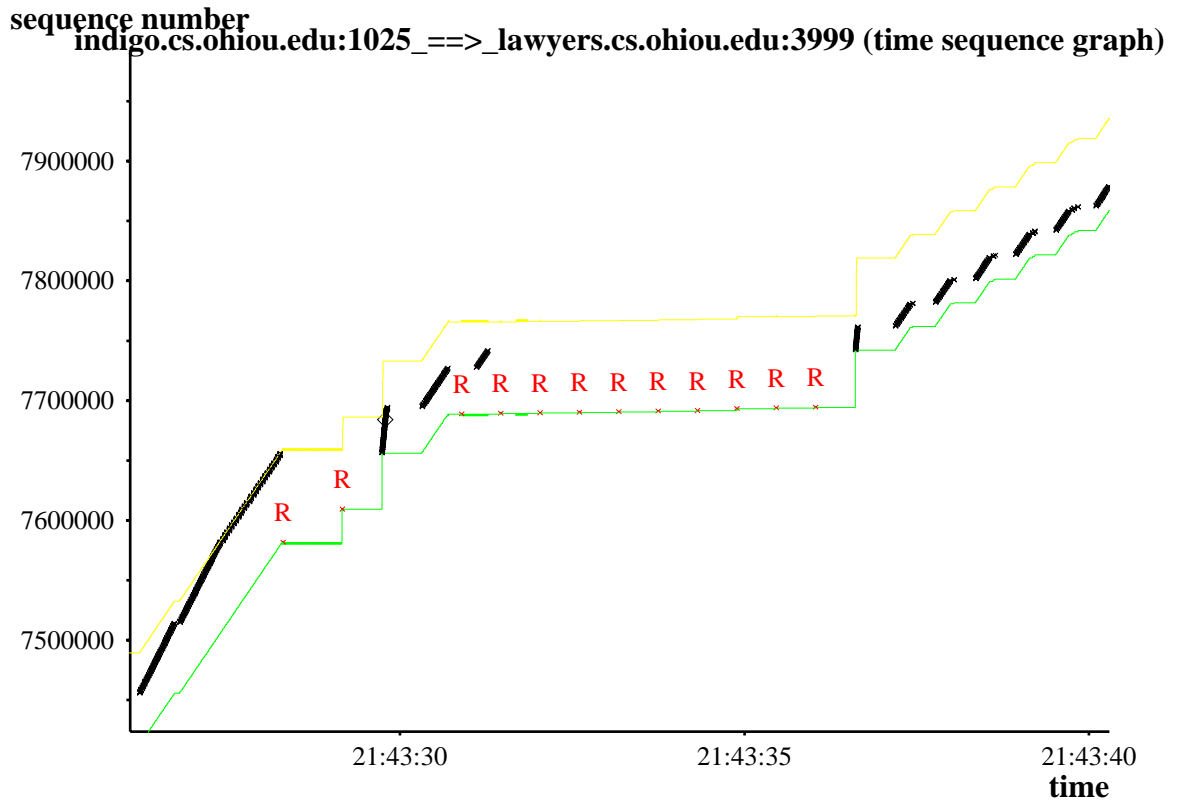


Figure 3.7 TCP New Reno with 2 Drops

This time sequence graph shows New Reno TCP's performance with a two drop loss event. New Reno does not exit the fast retransmit phase until the highest unacknowledged segment is ACKed. Partial ACKs trigger the retransmission of the lost segments. Upon recovery, New Reno sends a closely spaced burst of segments that the network cannot handle and a second loss event occurs.

ACKs are generated by out-of-order packets, the connection retransmits the lost segment indicated by the duplicate ACKs. The duplicate ACKs continue to arrive decrementing *pipe*. When *pipe* falls below *cwnd* the connection retransmits the first missing segment in the *scoreboard*. *Pipe* continues to decrement with each additional ACK causing the retransmission of the remaining lost segments in the *scoreboard*. When the connection exits fast recovery, a *cwnd* worth of data is burst onto the network over running the router queue causing a secondary loss event. The *cwnd* stabilizes at  $\frac{\text{original cwnd}}{4}$ .

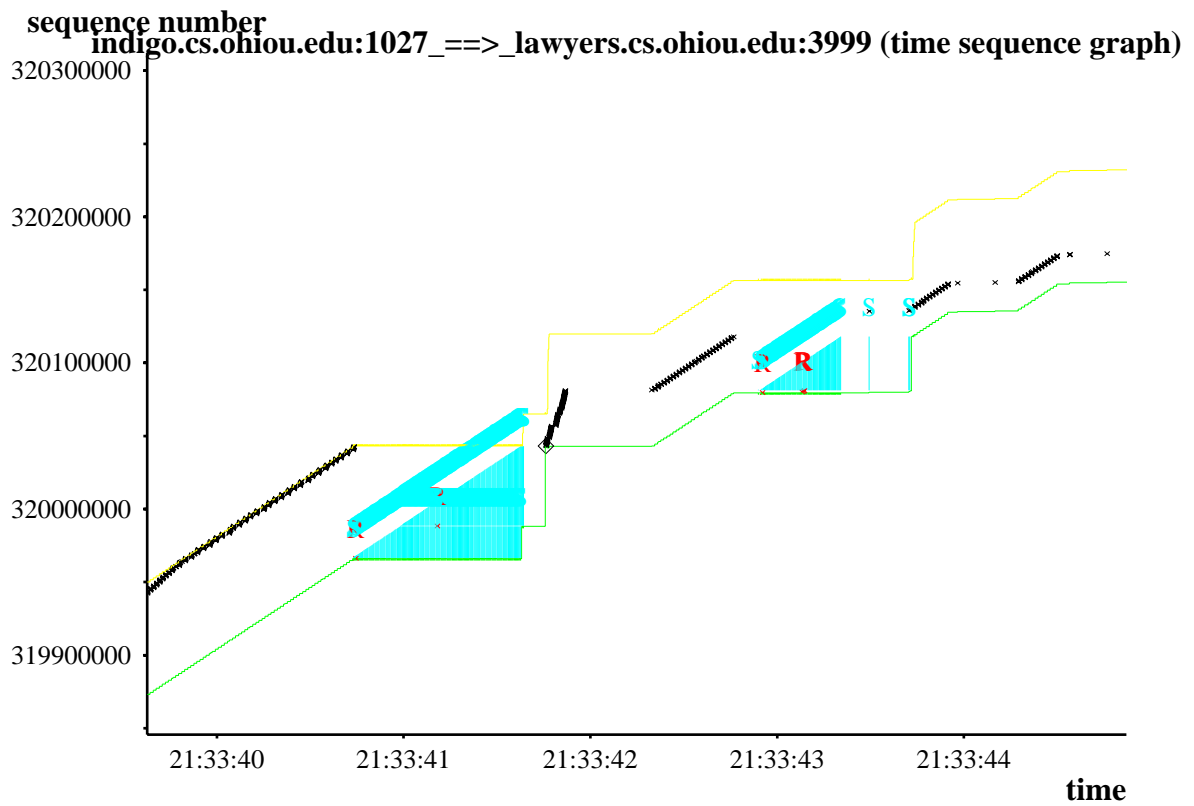


Figure 3.8 TCP with SACK with 2 Drops

This time sequence graph shows SACK TCP recovering from a two drop loss event. SACK retransmits the two lost segments using the SACK block information. Upon recovery, SACK sends a closely spaced burst of segments that the network cannot handle and a second loss event occurs.

TCP with forward acknowledgments as shown in Figure 3.9 provides a less aggressive recovery mechanism. FACK applies the fast retransmit algorithm to all missing segments stored in the *scoreboard*. The *scoreboard* tracks the number of SACK blocks received for a given segment. If the sender's *scoreboard* indicates the arrival of three duplicate SACK blocks, the missing segment is retransmitted. The connection enters the rate-halving phase for one RTT. During rate-halving, for every two duplicate ACKs, the connection checks the scoreboard for the next hole above the retransmit threshold. Two duplicate ACKs arrive and the connection retransmits the next segment. After the first round trip time, FACK enters a hold state and waits for the

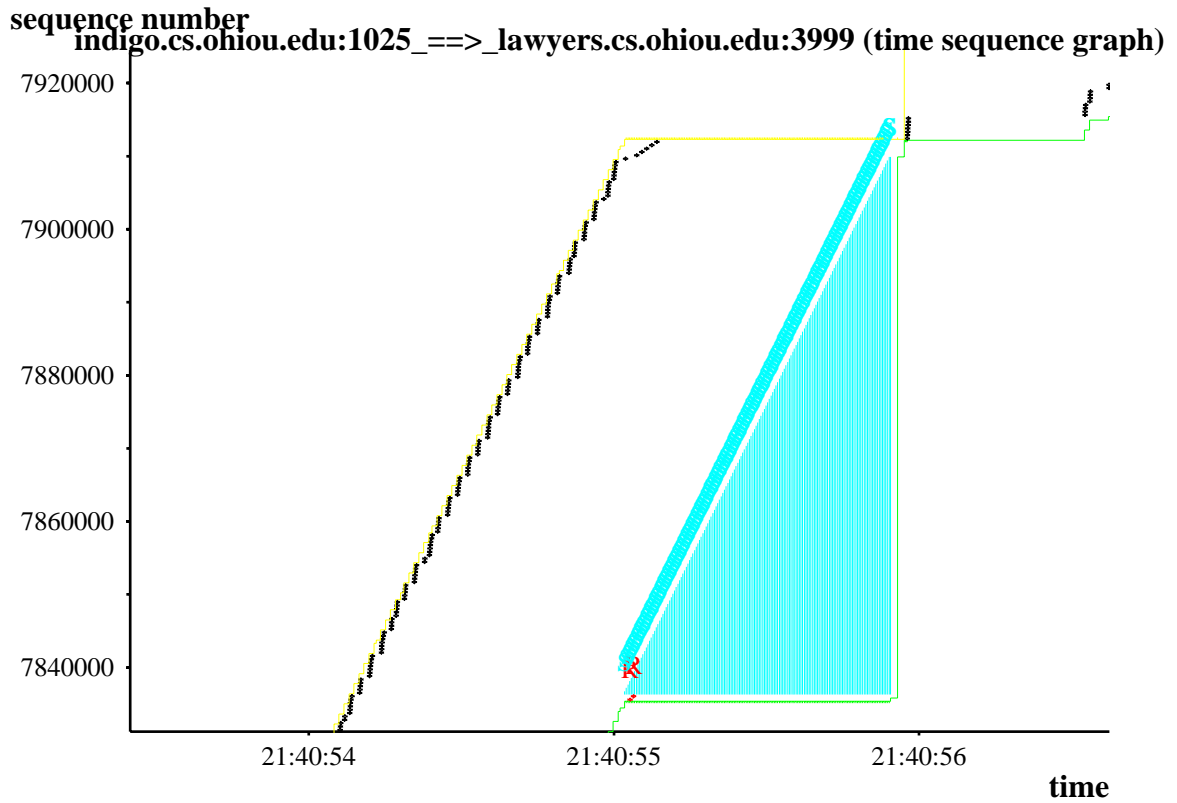


Figure 3.9 TCP with FACK with 2 Drops

This time sequence graph shows FACK TCP's performance with a two drop loss event. FACK recovers from the loss event using rate-halving. FACK can inject some new data into the network during rate-halving because the receiver's advertised window is not full. Upon Recovery, FACK's recovery congestion window is set to the eight segments sent during rate-halving and the connection enters slow start.

arrival of the ACK covering all outstanding data. When the ACK arrives for the second segment covering all outstanding data through  $snd\_fack$ , the recovery phase ends.  $Cwnd$  should be at  $cwnd = \frac{(cwnd-RHloss)}{2} - HSloss$ . This is the  $cwnd$  that FACK could obtain if FACK could inject data into the network. This connection is bound by the receiver's window during recovery and unable to inject new data into the network. This opens the  $cwnd$  to the amount of data that is transmitted during recovery. The recovery  $cwnd$  for this connection grew to eight segments because the loss event happened early in the window, the connection had not sent enough data

to fill the receiver's window and is able to send six new segments into the network before the receiver's window is full. Due to the conservative `cwnd`, FACK avoids the secondary loss event associated with SACK and New Reno.

From the two drop experiments, we conclude that while SACK increases the amount of loss needed to recover, its fast recovery time, larger recovery window, and avoidance of slow start provide better throughput than any of the other extensions. FACK relies on the ability to inject new data into the network. Because the receiver's window is full, FACK is unable to inject any new data into the network. The `cwnd` is opened only to the number of segments transmitted during recovery. This `cwnd` is very conservative and degrades performance. New Reno requires a longer time in recovery than SACK or FACK. A round trip time must be spent to recover each lost segment. The secondary loss event adds additional delay as New Reno must spend more time in recovery. New Reno still out performs Reno. Reno cannot recover from the loss and reenters slow start.

### 3.3.3 Four Drops

The four drop tests use the same setup as the two drop tests. The router buffering was adjusted to produce a four drop loss instead of only dropping two segments. The four drops all occur when a receiver's window worth of unacknowledged data is on the network so the sender is unable to inject new data into the network. Table 3.2 illustrates the effects of the recovery algorithms.

Reno cannot recover from the four drop loss event. Reno fast retransmits the first lost segment and enters fast recovery. The connection already has a receiver's window worth of data unacknowledged on the network and cannot send any new data. The connection cannot enter another fast retransmit phase and must wait for a retransmission timeout. Reno enters slow start and begins transmitting from the last ACK. Some of the transmitted segments are in the receiver's buffer triggering duplicate ACKs to be generated as the duplicate segments appear as out-of-order segments. This forces an unnecessary fast retransmit. The connection recovers and

TCP	Initial window in bytes	Recovery window in bytes	Rexmted. segs.	Secondary loss event	Slow Start	Throughput in Bps
Reno	168 * 512	1 * 512	6	NO	YES	17992
New Reno	168 * 512	42 * 512	14	YES	NO	29159
SACK	168 * 512	42 * 512	12	YES	NO	35352
FAK	168 * 512	4 * 512	4	NO	NO	31965

Table 3.2 Algorithm Comparisons with Four Drops

enters congestion avoidance with a cwnd of four. While in congestion avoidance, the cwnd can only open by at most one segment per round trip time leading to degraded performance.

New Reno retransmits the first segment and uses the partial ACKs to retransmit the other three lost segments. New Reno exits recovery when the ACK greater than *snd\_high* is received. New Reno halves the cwnd and sends the full cwnd worth of data onto the network which produces a secondary loss event. Another fast retransmit phase is entered which recovers the lost segments. Upon recovery, New Reno halves the cwnd again and sends the full cwnd onto the network but this time the burst is sufficiently small and does not overwhelm the network.

SACK retransmits the first segment and quickly retransmits the next three segments as the duplicate ACKs decrement *pipe*. SACK exits recovery when the highest unacknowledged segment is acknowledged. SACK halves the cwnd and bursts the full cwnd worth of data onto the network. The burst overwhelms the network producing another loss event. SACK enters another recovery phase and recovers the lost segments. Upon recovery SACK halves the cwnd again and bursts the full cwnd onto the network but this time the burst is sufficiently small and does not overwhelm the network.

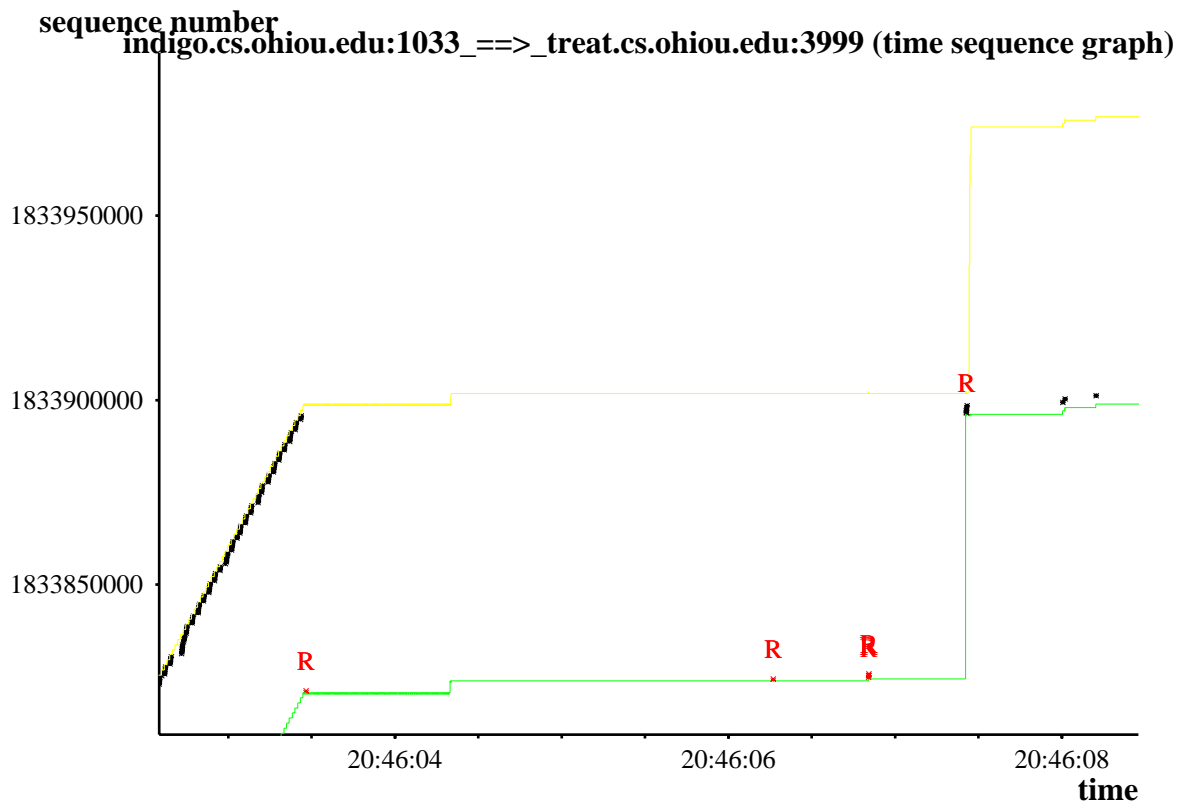


Figure 3.10 TCP Reno with 4 Drops

This time sequence graph shows TCP Reno's performance with a four drop loss event. Reno cannot use fast retransmit to retransmit the remaining segments because all the duplicate ACKs have drained from the network. A retransmission timeout occurs and the TCP connection enters slow start.

FAK receives three duplicate SACK blocks and retransmits the first segment. The connection enters the rate-halving phase and is supposed to send or retransmit a segment every two duplicate ACKs. The connection has a receiver's window worth of data unacknowledged on the network and cannot send any new segments. Rate-halving retransmits the four segments. The cwnd opens to the four segments sent during rate-halving. This conservative cwnd provides for degraded performance.

Reno cannot recover from four drops and enters slow start. Reno also retransmits segments already held by the receiver causing the connection to enter an unnecessary recovery phase. SACK and New Reno perform similar to the two drop events. Both

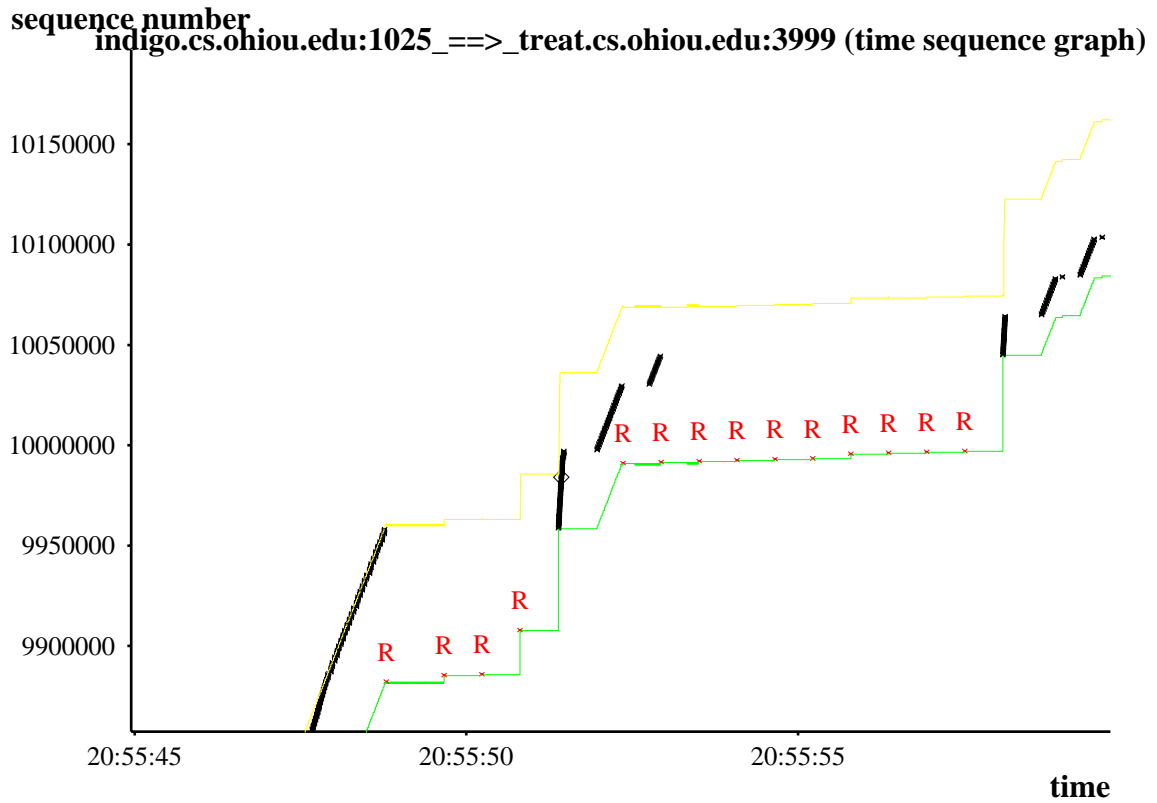


Figure 3.11 TCP New Reno with 4 Drops

This time sequence graph shows New Reno TCP's performance with a four drop loss event. New Reno does not exit the fast retransmit phase until the highest unacknowledged segment is ACKed. Partial ACKs trigger the retransmission of the lost segments. Upon recovery, New Reno sends a closely spaced burst of segments that the network cannot handle and a second loss event occurs.

suffering from another loss event after the initial recovery. FACK again suffers from its inability to inject new data into the network. The cwnd is only opened to four segment after recovery. Again with FACK, no secondary loss event occurs due to the conservative window.

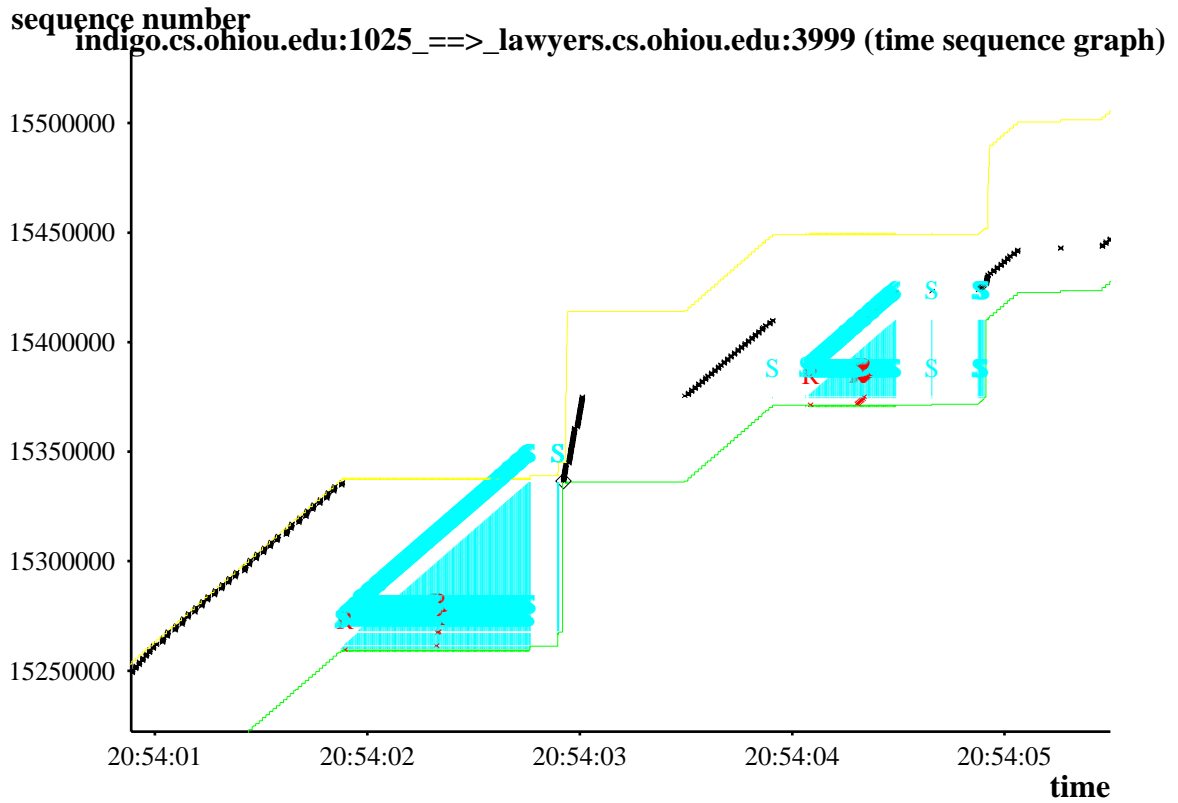


Figure 3.12 TCP with SACK with 4 Drops

This time sequence graph shows SACK TCP recovering from a four drop loss event. SACK retransmits the two lost segments using the SACK block information. Upon recovery, SACK sends a closely spaced burst of segments that the network cannot handle and a second loss event occurs.

### 3.4 File Transfer Experiments

The drop tests were designed to give a better understanding of how the mechanisms themselves worked. To investigate the new TCP extensions over realistic network conditions, we conducted file transfers of 200 KB, 1 MB, and 5 MB files. We transferred the files over a 768,000 bps link with 290ms delay in each direction using an FTP with adjustable window sizes. The window was adjusted to be slightly larger than the available router buffering. Each test was conducted 30 times for statistical accuracy. The 200 KB test does not exit slow start and has no loss event. The 1

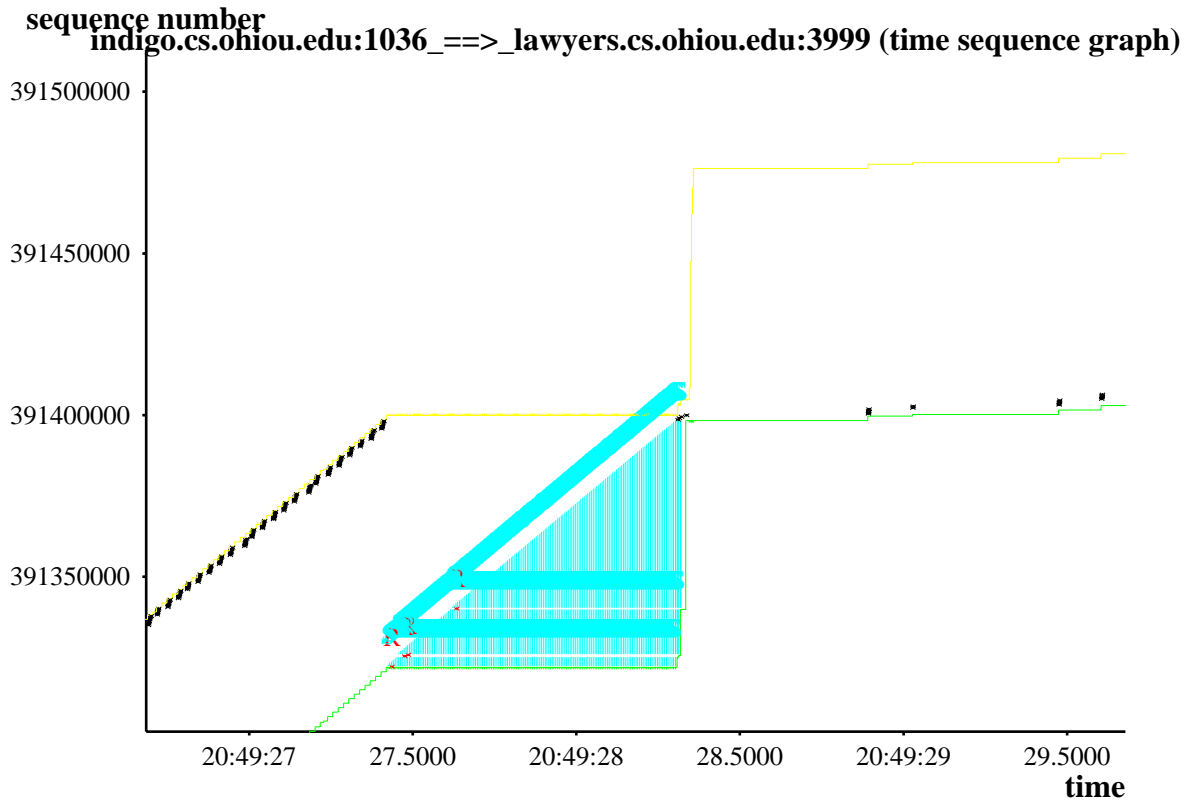


Figure 3.13 TCP with FACK with 4 Drops

This time sequence graph shows FACK TCP's performance with a four drop loss event. FACK recovers from this loss event using rate-halving. FACK cannot inject new data into the network during rate-halving because the receiver's advertised window of unacknowledged data is on the network. Upon Recovery, FACK's recovery congestion window is set to the four retransmitted segments and enters slow start.

MB test has a moderate loss event and a short recovery time. The 5 MB test has a moderate loss event and a longer recovery time.

For the 200 KB files, no loss event occurs. 200 KB files are not large enough to cause a congestion loss. Without loss, FACK, SACK, Reno, and New Reno have equal throughput. This is because the new TCP changes are only in the recovery mechanisms. Due to the capacity wasted during slow start, all of the TCP's performs at 27% of optimal.

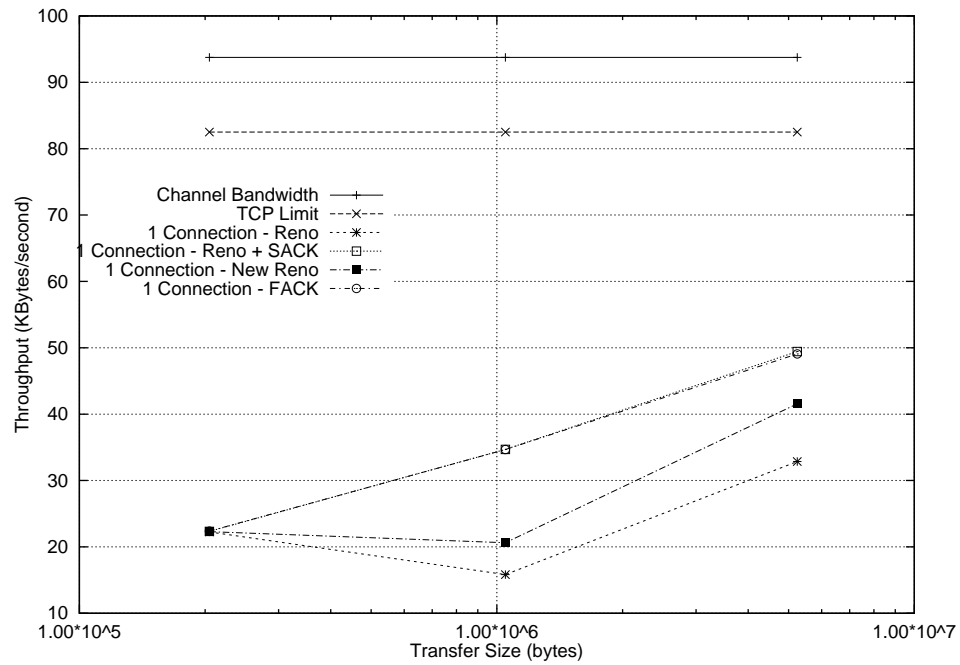


Figure 3.14 Test Comparisons of 200 KB, 1 MB, and 5 MB File Sizes  
 The 200 KB files are not large enough to generate loss. The 1 MB files has a moderate loss event and a short recovery time. The 5 MB file has a moderate loss event and a longer recovery time.

TCP	200 KB Tput in KBps	% Optimal	1 MB Tput	% Optimal	5 MB Tput	% Optimal	Optimal Tput
Reno	22.26	27%	15.83	19%	32.86	40%	82.5
New Reno	22.28	27%	20.62	25%	41.55	50%	82.5
SACK	22.34	27%	34.67	42%	49.43	60%	82.5
FAK	22.38	27%	34.66	42%	49.07	59%	82.5

Table 3.3 File Transfer Statistics for 200 KB, 1 MB, and 5 MB Transfers

For 1 MB files, TCP sends data faster than the router can send into the slower link. The router buffer begins to buffer segments until the router buffer is full. When the router buffer is full, the router drops incoming packets. The loss event drops multiple segments which causes performance problems in Reno. The loss event forces Reno back into slow start degrading Reno's performance. Reno performs at 19% of optimal. New Reno, performing at 25% of optimal, recovers better than Reno but slower than SACK or FACK. New Reno requires a round trip time to recover each lost segment. SACK and FACK require at most two round trip times if no further loss occurs during recovery. This accounts for the throughput difference between New Reno and SACK or FACK. SACK and FACK have almost identical throughput. Both perform at 42% of optimal. Normally SACK would perform better than FACK due to its larger recovery window, but the 1 MB transfer ends soon after recovery so the window size makes little difference.

For 5 MB files, the results are similar to the 1 MB files except that the larger file size provides a longer recovery period. Reno performs at 40% of optimal. SACK performs the best at 60% of optimal with FACK slightly lower at 59%. FACK's throughput is slightly lower because FACK sets a more conservative window than SACK after the loss event. Due to the long recovery period, FACK's conservative window only marginally effects the throughput. New Reno is between SACK and Reno performing at 50% of optimal.

### 3.5 Non-Congestion Related Loss

Non-congestion related loss occurs when bit errors are induced by the satellite link. These bit errors result in segments being discarded when they fail a checksum test at the receiving TCP[Com95]. Because the segment is corrupt, the valid recipient of a segment cannot be assumed and no information can be returned to the connection. To a TCP connection, all loss is assumed to be congestion related. The non-congestion related loss experiments examine if the new TCP extensions provide

better performance than standard TCP when encountering non-congestion related loss.

We conducted the non-congestion related loss tests by transferring a 5 MB file over a 1,544,000 bps(T1) link. We transferred the files using a 110KB window with 290ms delay in each direction using FTP. The window was adjusted to be slightly lower than the available router buffering to produce no congestion related loss. Each test was conducted 30 times for statistical accuracy.

The hardware emulator can be programmed to introduce a given number of bit errors at a deterministic interval. We chose to insert 4000 consecutive bit errors(500 bytes). An error burst of that size is likely to effect two adjoining segments. Glover[Glo96] reports that typical bit error rates for satellite networks are  $1 \times 10^{-7}$ . For our interval, we chose bit error rates of  $1 \times 10^{-7}$ ,  $1 \times 10^{-6}$ , and  $1 \times 10^{-5}$ . Those intervals give us a clear understanding of how TCP performs under typical to extremely lossy line conditions. Figure 3.15 and table 3.4 show the results of our tests.

For bit error rates of  $1 \times 10^{-7}$ , the hardware emulator introduced 4000 bit errors every 10,000,000 bits. At this error rate, we are destroying one or two segments out of 2400. This injects three loss events into a 5 MB transfer. 165 KB/s is the optimal transfer rate for this line speed. The new TCP extensions all provide better throughput than Reno at error rates of  $1 \times 10^{-7}$ . FACK and SACK obtain approximately 24% of optimal, New Reno obtains approximately 28%, of optimal, and Reno only achieves approximately 22% of optimal.

For bit error rates of  $1 \times 10^{-6}$ , the bit error rate is destroying one or two segments out of 240, or injecting 30 loss events into a 5 MB transfer. All of the TCP extensions perform at approximately 4% of optimal at this error rate. The new recovery algorithms make little difference at this high loss rate.

For bit error rates of  $1 \times 10^{-5}$ , the bit error rate is destroying one or two segments out of 24. All of the TCP extensions exhibited poor performance, obtaining approximately 1% of optimal. The new recovery algorithms make little difference at this high loss rate.

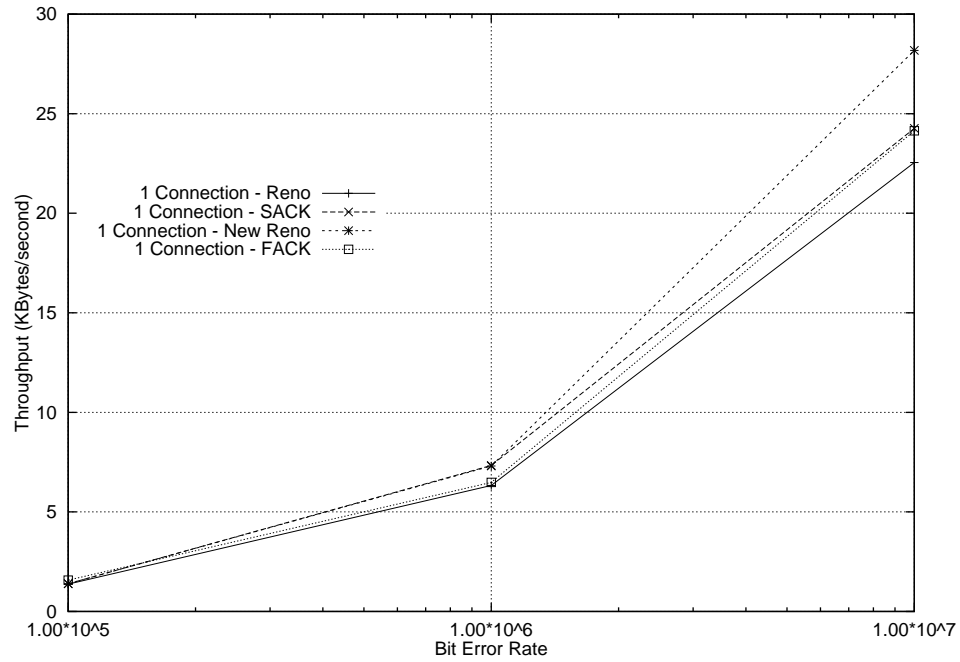


Figure 3.15 Test Comparisons of Various Bit Error Rates for 5 MB file Transfers  
 This graph compares the throughput of 5 MB files transferred over a connection with bit error rates of  $1 \times 10^{-5}$ ,  $1 \times 10^{-6}$ , and  $1 \times 10^{-7}$ .

TCP	Tput KBps $1x10^{-7}$	% Optimal	Tput $1x10^{-6}$	% Optimal	Tput $1x10^{-5}$	% Optimal	Optimal Tput
Reno	22.55	14%	6.32	4%	1.37	1%	165
New Reno	28.18	17%	7.30	4%	1.39	1%	165
SACK	24.26	15%	7.33	4%	1.40	1%	165
FACK	24.14	15%	6.48	4%	1.57	1%	165

Table 3.4 Bit Error Tests Statistics for 5 MB File Transfers

The new TCP extensions provide better performance only if the bit errors produce a multiple loss event. The bit errors at  $1x10^{-7}$  appear infrequent enough to look like light congestion and the connection recovers well. For bit errors at  $1x10^{-6}$  and  $1x10^{-5}$  the errors occur so frequently that the new recovery algorithms make little difference. All of the new extensions assume loss is generated by congestion not link errors. In all of the TCP extensions, the window is reduced after recovery. The new extensions provide no method to distinguish between link errors and congestion. TCP halves the congestion window in response to congestion after recovery. This leads to a unnecessary reduction of the window when encountering bit errors. The new extensions provide better recovery only because they can recover multiple drops. This is the only advantage they provide over Reno when facing non-congestion related loss.

## 4. CONCLUSIONS

### 4.1 Conclusions

Larger Windows and SACK provide necessary changes to improve TCP's performance over long delay links. The information provided by the SACK blocks allow for more efficient recovery mechanisms. All of the new recovery mechanisms provide greater throughput than Reno in the presence of loss. New Reno is only an optimization of standard Reno TCP. New Reno may be an interim solution to Reno recovery problems while RFC 2018 is in the standards process. It is important to note that users can benefit from the New Reno changes now. New Reno requires only sender side changes and does not require both sender and receiver to be modified.

To gain the benefits of selective acknowledgments both the sending and receiving TCP must agree to use it. It may take a long period of time to change every TCP implementation to use the SACK option. However, it is clear from our tests that both of the mechanisms that utilized the SACK information achieved higher throughput.

We recommend the Fall and Floyd recovery algorithm. The Fall and Floyd recovery algorithm is a conservative extension to fast retransmit and fast recovery. The Fall and Floyd recovery algorithm provides better throughput than FACK which is still an experimental TCP. This research concludes that FACK has obvious performance problems when the sender is unable to inject new data into the network during recovery. While Fall and Floyd is a simple extension to fast retransmit and Fast Recovery, FACK is a complicated algorithm relying on many bounding parameters. Despite FACK's complicated algorithms, FACK offers no performance gains over the Fall and Floyd SACK recovery algorithm.

FACK's conservative window does prevent a secondary loss event during recovery. This is worthy of further investigation. Our experiments indicate that a better recovery window estimation is needed in a long delay network. The long delay has the effect of generating a large burst after recovery that is too closely spaced for the routers to handle. This causes a secondary loss event which forces the TCP to spend unnecessary time in recovery. This research shows that a maxburst variable controlling the number of segments allowed on the network after recovery may be needed. This idea is discussed in Hoe[Hoe96]. Another solution could make use of the RTT. If the RTT is over 500ms(one timer tick) then we should decrease the window by another power of two. If the delay is less than 500ms we halve the window, else for each 500ms tick we divide the window by another power of two. So if the RTT was 1000ms we would divide the congestion window by eight. Note this scheme would only effect the cwnd not ssthresh. The connection would enter slow start and begin working back up to the proper window size. This would prevent a burst that the network cannot handle.

#### 4.2 Further Research

The new TCP extensions are centered around recovering losses. If you can avoid loss altogether you can achieve higher throughput. Keshav[Kes91],Hoe[Hoe96] and Paxson[Pax97] investigate new window estimation techniques that help prevent loss. These new techniques would estimate the amount of bandwidth available in the network and end slow start at that point. This would prevent the loss event that is found at the end of the slow start phase. Mathis[MSMO97] is also working on a mathematical approach to estimating the receiver's window.

Slow start modifications may be an area of improvement for TCP performance. Floyd[FAP97] suggests increasing the initial cwnd window from one segment to

$$cwnd = \min(\text{foursegments}, 4096\text{bytes}). \quad (4.1)$$

This would save up to three round trip times and allow the *cwnd* to open faster. This decreases the amount of time in slow start which is beneficial to satellite links. Another slow start modification would be to adjust the window increase algorithm. Most TCP's used delayed acknowledgments when ACKing segments[Bra89]. They can delay for a most two full sized segments before sending an ACK. This doubles the amount of time a connection spends in the slow start phase. We are investigating a window increase algorithm[AHO97] that would increase the window by

$$cwnd+ = \min(\text{newbytesacknowledged}, \text{twoMSS}). \quad (4.2)$$

This prevents the window from opening by more than two segments per ACK if the TCP is in violation of RFC 793[Pos81] and does not ACK for every two full sized segments. The window increase algorithm would reduce the amount of time in slow start for connections using delayed ACKs. Less time in slow start means less wasted capacity for the TCP connection.

## BIBLIOGRAPHY

## BIBLIOGRAPHY

- [AHO97] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP Slow Start Modifications, 1997. In preparation.
- [All97] Mark Allman. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.
- [Bra89] Robert Braden. Requirements for Internet Hosts - Communication Layers, October 1989. RFC 1122.
- [Com95] Douglas E. Comer. *Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture*. Prentice Hall, 3rd edition, 1995.
- [FAP97] Sally Floyd, Mark Allman, and Craig Partridge. Increasing TCP's Initial Window, July 1997. Internet-Draft draft-floyd-incr-init-win-00.txt.
- [FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, July 1996.
- [Glo96] Daniel Glover. Satellites and Networks. Technical report, NASA Lewis Research Center, 1996. URL: <http://sulu.lerc.nasa.gov/dglover/charactr.html>.
- [Hoe95] Janey Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, June 1995.
- [Hoe96] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 270–280, New York, August 1996. ACM Press.
- [Jac90] Van L. Jacobson. Fast Retransmit Note to End2end-interest mailing list, April 1990.
- [JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

- [JK88] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In *Proceedings, SIGCOMM '88 Workshop*, pages 314–329. ACM SIGCOMM, ACM Press, August 1988. Stanford, CA.
- [Kes91] S. Keshav. A Control-theoretic approach to flow control. In *ACM SIGCOMM*, pages 3–15, August 1991.
- [Kru95] Hans Kruse. Performance Of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination. In *3rd International Conference on Telecommunication Systems Modeling and Design*, 1995.
- [MM96a] M. Mathis and J. Malidavi. Forward acknowledgement: Refining TCP congestion control. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 281–292, New York, August 1996. ACM Press.
- [MM96b] Matt Mathis and Jamshid Mahdavi. TCP Rate-Halving with Bounding Parameters. Technical report, Pittsburgh Supercomputer Center, October 1996. URL: <http://www.psc.edu/networking/papers/FACKnotes/current/>.
- [MMFR96a] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options, oct 1996. RFC 2018.
- [MMFR96b] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and A. Romanow. TCP Selective Acknowledgment Options, October 1996. RFC 2018.
- [MSMO97] Matthew Mathis, Jeffery Semke, Jamshid Mahadvi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3):to appear, July 1997.
- [Pax97] Vern Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, September 1997. To appear.
- [Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.
- [Sta94] William Stallings. *Data and Computer Communications*. MacMillan, 4 edition, 1994.
- [Ste97] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

## APPENDIX

## A. HOW TO READ A TCP TIME SEQUENCE GRAPH

A TCP time sequence graph is a graphical representation of a TCP connection. TCP connections are full duplex, allowing the transfer of data in both directions simultaneously. A time sequence graph shows only one side of the full duplex connection. This allows the viewer to separate the actions of the sending and receiving hosts. Our tests used the File Transfer Protocol, FTP, as the sending agent. In our tests, the FTP transfers were uni-directional, receiving no data from the receiving host other than acknowledgments. Figure A.1 is an example time sequence graph.

Our graphs consists of data flowing from the sender to the receiver. The receiver sends acknowledgments in response to the incoming data. Each TCP segment has a sequence number to represent its order in the data stream. The y-axis of the time sequence graph is the TCP sequence number. The x-axis is the time index of the connection in seconds. As the sequence numbers increase, the graph will increase diagonally along the page. The graph grows from the lower left to upper right of the graph.

A data segment is represented by a vertical line with arrows at each end. The length of the line is determined by the size of the segment in bytes. The vertical placement of the line on the page represents the position of those bytes in the data stream.

A retransmitted data segment is a vertical line with an arrow on both sides topped with an R. Notice the vertical ticks along the bottom horizontal line. These vertical ticks represent a duplicate acknowledgment received by the sender. If selective acknowledgments, SACK, blocks are present in a connection, they will be represented by a vertical line topped with an S. The height of the SACK blocks vertical line is the length of the segments covered by the SACK information.

The horizontal line above the segments represents the upper limit of the receiver's window. The amount of space available in the receiving window is represented by the space between the bottom horizontal line and the top horizontal line. During the connection, the sender is not supposed to have more than the receiving window worth of data on the network. Graphically this bounds all of the segments between the two horizontal lines.

Examine the top edge of the receiving window, the top horizontal line in the graph. A vertical line represents the advance of the receiving window in the sequence space. The vertical line is a TCP window update indicating the receiver has removed some segments from its receiving buffer.

Examine the bottom edge of the receiving window, which is the bottom horizontal line. An advancing of the window is indicated by a vertical line. The window is advanced when a received segment is acknowledged by the receiver. When the sender receives an acknowledgment the window is advanced. The vertical line represents the segments acknowledged up to and including its position in the byte stream. The second vertical line around 21:33:42.75 represents a four segment acknowledgment covering segments 2-5.

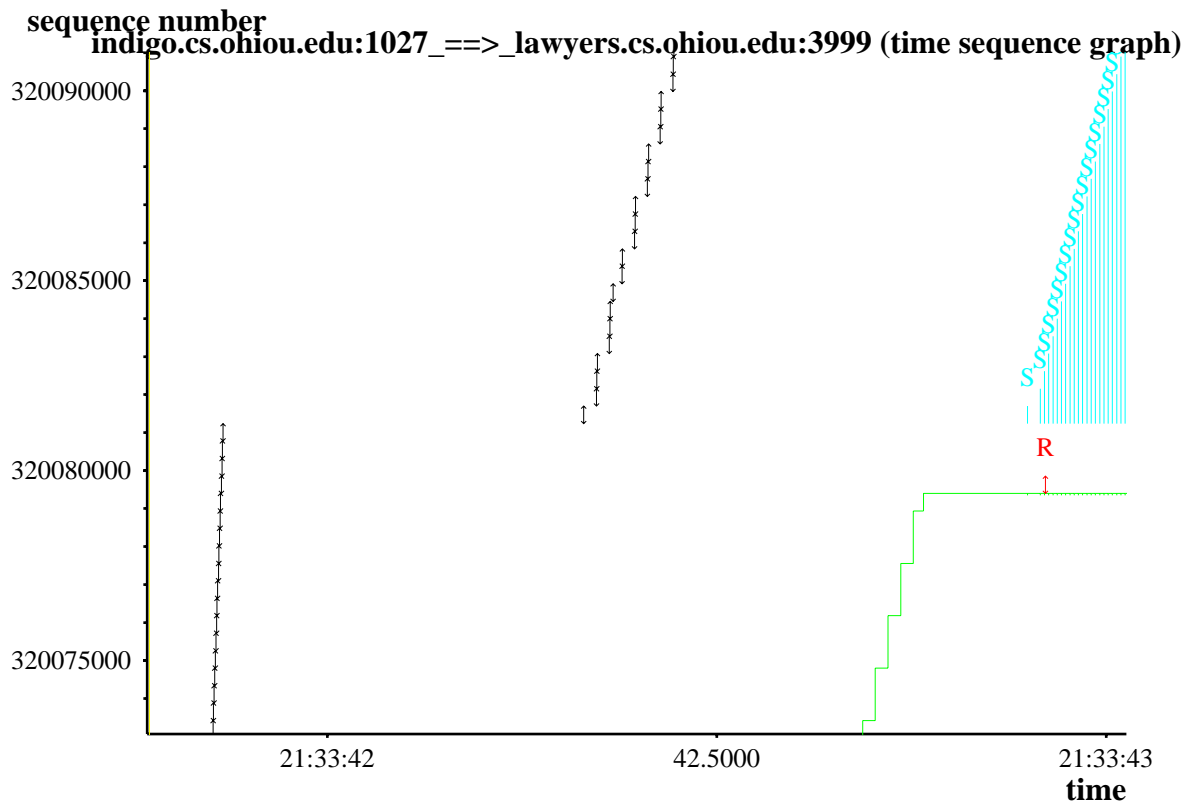


Figure A.1 TCP Time Sequence Graph Example  
Example time sequence graph generated using tcptrace.

Hayes, Christopher. M.S. August, 1997  
Electrical Engineering

Analyzing the Performance of New TCP Extensions over Satellite Links (44 pp.)

Director of Thesis: Dr. Shawn Ostermann

It has been shown in Kruse that long delay satellite channels suffer poor throughput using the Transmission Control Protocol, TCP. Performance is limited by the delay inherent in geosynchronous satellites. Several changes have been proposed that could help TCP performance over long delay paths. These changes include big windows and PAWS proposed in RFC 1323, Selective Acknowledgments(SACK), proposed in RFC 2018, Hoe's fast retransmit modifications, and Mathis and Mahdavi's Forward Acknowledgments(FACK). This thesis examines the performance of the new TCP extensions over long delay links. Performance is examined using one, two, and three drop loss events and transfers of various file sizes. This thesis also investigates the performance of the new TCP extensions when encountering varying degrees of bit errors in the satellite channel.

Approved: \_\_\_\_\_