

IMPROVING TCP PERFORMANCE OVER SATELLITE CHANNELS

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

Mark Allman

June, 1997

THIS THESIS ENTITLED
“IMPROVING TCP PERFORMANCE OVER SATELLITE CHANNELS”

by Mark Allman

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

This thesis would not have been possible without help from a number of people. Thanks to Shawn for enlightening conversations, clever ideas, patience and for providing a stimulating research environment. I would also like to thank Hans and Ohio University's Internetworking Research Group for many animated discussions and much support over the last few years. Special thanks to my officemates Boris, Brian, Chris, Doug and Rich for their assistance, conversations and for generally putting up with me. Also, I thank my family and many friends for their support over the years. This work would not have been possible without generous funding from NASA's Lewis Research Center and Ohio University's School of Electrical Engineering and Computer Science. Most importantly I thank my father, Wes, and Meredith for their encouragement and support.

DISCARD THIS PAGE

TABLE OF CONTENTS

	Page
LIST OF TABLES	v
LIST OF FIGURES	vi
1. INTRODUCTION	1
1.1 TCP Overview	1
1.1.1 Slow Start and Congestion Avoidance	4
1.1.2 Fast Retransmit and Fast Recovery	5
1.2 TCP Problems in the Satellite Environment	6
1.2.1 Problems Increasing the Sliding Window	7
1.2.2 Problems With TCP Steady-State Behavior	10
1.3 Possible Solutions	10
2. AN APPLICATION-LEVEL SOLUTION	12
2.1 XFTP Theory	12
2.2 Experimental Environment	13
2.2.1 NASA ACTS	14
2.2.2 Software Emulator	15
2.2.3 Hardware Emulator	16
2.3 Experimental Results	18
2.4 XFTP Lessons	22
2.4.1 Bigger Windows	22
2.4.2 Selective Acknowledgments	23
2.4.3 More Aggressive Slow Start	23
2.4.4 More Aggressive Congestion Avoidance	24
2.4.5 Slow Start Threshold Estimation	24

	Page
3. SLOW START MODIFICATIONS	25
3.1 Slow Start Modifications	25
3.2 Experiments With Larger Initial Window	27
3.3 Experiments With a New Window Increase Algorithm	29
3.4 Experiments with Larger Initial Windows and a Modified Window In- crease Algorithm	32
3.5 Future Work	33
4. CONCLUSIONS AND FUTURE WORK	35
4.1 Recommendations for TCP Extensions	35
4.1.1 Large Windows	35
4.1.2 Selective Acknowledgments	36
4.2 Recommendations for Future Modifications to TCP	36
4.2.1 Slow Start Modifications	36
4.2.2 Congestion Avoidance Modifications	37
4.2.3 Slow Start Threshold Estimation	37
4.2.4 New Loss Recovery Mechanisms	38
4.3 Conclusions	39
BIBLIOGRAPHY	40
 APPENDIX	
A. TCP CONGESTION CONTROL ALGORITHMS	44
B. FTP PROTOCOL MODIFICATIONS	49
B.1 Introduction	49
B.2 FTP Commands	49
B.2.1 MULT	49
B.2.2 MPRT	50
B.2.3 MPSV	50
B.2.4 Discussion	51
B.3 Enabling Multiple Data Connections	51
B.4 Dividing a File Across Multiple Connections	52
B.5 Recommended Limits	53

ABSTRACT 54

LIST OF TABLES

Table	Page
1.1 TCP Cumulative Acknowledgment Example	2
3.1 New Window Increase Algorithm Without Congestion	30
3.2 New Window Increase Algorithm With Congestion	31
A.1 Congestion Control Algorithm Example	45

LIST OF FIGURES

Figure	Page
1.1 Slow Start Comparison	8
1.2 Congestion Avoidance Comparison	9
2.1 NASA ACTS Satellite System	14
2.2 <i>ONE</i> Software Emulator Setup	15
2.3 A Comparison of NASA ACTS and <i>ONE</i>	16
2.4 Hardware Emulator Setup	17
2.5 A Comparison of NASA ACTS and the Hardware Emulator	17
2.6 XFTP Test Results	19
2.7 Modified XFTP Tests	21
3.1 Larger Initial Windows Without Congestion	28
3.2 Larger Initial Windows With Congestion	29
3.3 Larger Initial Windows and a Modified Window Increase Algorithm Without Congestion	33
3.4 Larger Initial Windows and a Modified Window Increase Algorithm With Congestion	34
B.1 XFTP Record Format	52

1. INTRODUCTION

The Transmission Control Protocol (TCP) [Pos81] [Com95] is the reliable connection-oriented transport protocol that a number of major Internet services use to communicate (e.g., SMTP [Pos82], NNTP [KL86], HTTP [BLFN96] [FJGFBL97], FTP [PR85]). TCP performs well in terrestrial networks, but does not make full use of the available bandwidth over satellite channels [Kru95]. The following is an overview of TCP, followed by an outline of specific problems discovered in the satellite environment. Finally, several proposed solutions are given followed by a brief outline of the research presented in this paper.

1.1 TCP Overview

TCP provides reliable segment delivery through a positive acknowledgment mechanism. Each data segment transmitted contains a sequence number indicating the position of the data in the transmission. The sequence number is determined by a base sequence number picked by the operating system and the relative position of the segment's first octet of data in the transmission. For example, assume a TCP sender with base sequence number X transmits 100 byte segments. The first segment will contain sequence number X and the second segment will contain sequence number $X + 100$. For simplicity, this paper treats the sequence number as a "segment number," rather than the standard "octet number." Including sequence numbers in segment headers allows the receiver to reassemble the stream of data if segments are re-ordered. Additionally, the receiver is able to report the segments that have arrived to the sender

by transmitting an acknowledgment (ACK) to the sender for each segment received. Each ACK contains the sequence number of the next in-order segment the receiver expects to arrive. Table 1.1 provides an example of TCP's acknowledgment behavior. The table shows that segment 1 triggers an ACK containing sequence number 2. This indicates that the next segment the receiver expects contains sequence number 2. The arrival of segments 2 and 3 trigger similar ACKs. Notice that segment 4 does not arrive at the receiver before segments 5 and 6. The receiver generates ACKs containing sequence number 4 in response to segments 5 and 6 because segment 4 is the next in-order segment the receiver expects to arrive. When segment 4 does arrive, the ACK triggered contains sequence number 7, indicating the next in-order segment expected should contain sequence number 7. If a sender does not receive an ACK for a given segment within a certain amount of time, the segment is *retransmitted*. The amount of time the sender waits for an ACK before retransmitting the data segment is the *retransmit timeout* (RTO). The RTO is a smoothed estimation of the round trip time (RTT) plus some variation. The exact details of calculating the RTO are given in [JK88] and [KP87].

Segment Received	ACK Sent
1	2
2	3
3	4
5	4
6	4
4	7

Table 1.1 TCP Cumulative ACK Example

This figure shows TCP's acknowledgment behavior from the receiver's perspective. Each segment received triggers the transmission of a cumulative ACK containing the sequence number of the next segment the receiver expects to arrive.

TCP is a *sliding window* protocol. A sliding window protocol allows the sender to transmit a given number of segments before receiving an ACK. When an ACK is received by the sender, the window “slides” to allow one more segment to be transmitted. Each TCP segment sent (data segments and ACKs) contains a *window advertisement*. The size of the window advertised by the receiver is the upper bound for the sender’s sliding window. The largest window standard TCP can advertise is 65,535 bytes due to the 16 bits allocated for the advertisement in the TCP header [Pos81].

TCP uses a set of *congestion control* algorithms [JK88] [Jac90b] [Ste97] that further control TCP’s sending behavior. These algorithms are important because they ensure that TCP will not transmit data at a rate that is inappropriate for the network resources available. If TCP’s transmission rate is too high, the intermediate routers in the network can be overwhelmed. If segments arrive at an intermediate router faster than the router can forward the segments, the segments will be queued for later processing. If a segment arrives at a router that has no memory to queue the segment, the segment will be discarded. Therefore, it is important for TCP to be able to adapt its sending rate to the network conditions to avoid segment loss.

When too many TCP connections are sending at an inappropriately high rate the network can suffer from *congestive collapse*. Congestive collapse is a state when segments are being injected into the network but very little useful work is being accomplished; most of the data segments or their corresponding ACKs are discarded by one of the intermediate routers in the network before reaching their destination. This causes the sender to retransmit the data, further aggravating the problem. Congestive collapse is discussed in more detail in [Nag84a], [Nag84b] and [FF97].

TCP’s congestion control algorithms attempt to prevent congestive collapse by detecting congestion and reducing the transmission rate accordingly. While these algorithms are very important, they can also have a negative impact on the performance of TCP over satellite channels [Kru95]. TCP’s four congestion control algorithms are

slow start, congestion avoidance, fast retransmit and fast recovery [JK88] [Jac90b] [Ste97]. The following is a brief outline of these algorithms. A detailed example of how these algorithms work together is given in appendix A.

1.1.1 Slow Start and Congestion Avoidance

The slow start and congestion avoidance algorithms [JK88] [Ste97] allow TCP to increase the data transmission rate without overwhelming the intermediate routers. To accomplish this, TCP senders use a variable called the *congestion window* (*cwnd*). TCP's congestion window is the size of the sliding window used by the sender and *cwnd* cannot exceed the size of the receiver's advertised window. Therefore, TCP cannot inject more than *cwnd* segments of unacknowledged data into the network.

The slow start algorithm is used to gradually increase the amount of unacknowledged data TCP injects into the network, by gradually increasing the size of the sliding window. Slow start is used at the beginning of a TCP connection and in certain instances after congestion as detected. The algorithm begins by initializing *cwnd* to 1 segment¹. For each ACK received, TCP increases the value of *cwnd* by 1 segment. For example, after the first ACK arrives, *cwnd* is incremented to 2 segments and TCP is able to transmit 2 new data segments. This algorithm provides exponential increase in the size of the sliding window. Slow start continues until either the size of *cwnd* reaches the *slow start threshold* (*ssthresh*) or segment loss is detected. The value of *ssthresh* is initialized to the size of the receiver's advertised window at the beginning of the connection. If TCP's RTO expires for a given segment, TCP retransmits the segment but also uses this as an indication of network congestion. In response to an RTO timeout, TCP reduces its sending rate by setting *ssthresh* to half of *cwnd*'s value and then setting *cwnd* to 1 segment. This triggers the slow start algorithm, which will stop when the value of *cwnd* meets or exceeds *ssthresh*

¹In practice, *cwnd* is measured in bytes, however, to simplify discussion we express it in terms of segments in this paper.

or another loss is detected. The new value of *ssthresh* places an upper bound on the slow start algorithm of half the sending rate when the loss was detected.

Congestion avoidance is the phase that follows slow start. In this phase the value of *cwnd* is greater than or equal to *ssthresh*. This algorithm increases *cwnd* at a slower rate than during slow start. For each segment ACKed during congestion avoidance, the congestion window is increased by $1/cwnd$ (unless this would make the value of *cwnd* greater than the receiver's advertised window). This adds roughly one segment to the value of *cwnd* every RTT. The congestion avoidance algorithm provides a linear increase in the size of TCP's sliding window. This mechanism is used to probe the network for additional capacity in a conservative manner.

1.1.2 Fast Retransmit and Fast Recovery

The fast retransmit and fast recovery algorithms [JK88] [Jac90b] [Ste97] allow TCP to detect and recover from segment drops more effectively than relying on the RTO. The RTO is a smoothed average of the RTT plus some variance. As defined in the TCP standard [Pos81], TCP retransmits a segment if the RTO expires before the segment is ACKed. This mechanism works well if the granularity of the timer that TCP uses is less than or equal to the RTO. However, the BSD Unix operating system's timer granularity is 500 ms [WS95] which is clearly not sufficient to trigger retransmissions on most terrestrial networks (with RTTs less than 500 ms). BSD's implementation of TCP is freely available and has been used as the basis of many other implementations of TCP. Our investigations show that timers with granularity of approximately 500 ms are prevalent.

Fast retransmit provides a way to retransmit a segment before the RTO expires. As discussed in section 1.1, TCP generates duplicate ACKs when segments arrive out-of-order. When a small number (usually 3) of duplicate ACKs arrive, TCP uses this as an indication that a segment has been lost, and it retransmits the appropriate segment. In addition, TCP uses the dropped segment as an indication of network

congestion and reduces the transmission rate. When a segment is retransmitted using fast retransmit the fast recovery algorithm is employed, as follows. The *cwnd* is reduced by half and *ssthresh* is set to the new value of *cwnd*. Each duplicate ACK received by the sender indicates that a segment has arrived at the receiver and is no longer in the network. TCP uses this knowledge to infer that the network can absorb another segment and artificially inflates *cwnd* by 1 segment. New data segments may be sent if the value of *cwnd* becomes greater than the number of unacknowledged segments in the network. Upon receipt of a non-duplicate ACK, TCP reduces *cwnd* by the amount it was artificially inflated (i.e., back to the value of *ssthresh*) and the congestion avoidance algorithm is used as described in section 1.1.1.

TCP reduces its transmission rate for each lost segment it detects, but the amount of the reduction depends on the way the segment loss was detected. When the retransmission is due to the expiration of the RTO, TCP cannot infer anything about the state of the network and therefore initiates slow start (up to half the transmission rate when the loss occurred). When the retransmission is triggered by the fast retransmit mechanism, TCP is receiving duplicate ACKs indicating that segments are still flowing between the sender and receiver. Therefore, the reduction of the transmission rate is not as large as when the sender is receiving no feedback. When retransmission is triggered by duplicate ACKs, TCP reduces the sending rate by half.

1.2 TCP Problems in the Satellite Environment

TCP's performance problems in the satellite environment have been outlined in the literature [Kru95]. The problems can be broken down into two broad areas: problems increasing the size of the sliding window and problems in steady-state behavior.

1.2.1 Problems Increasing the Sliding Window

As outlined above, TCP uses two algorithms to increase the size of the sliding window. The following two sections will outline the problems with slow start and congestion avoidance in the satellite environment.

1.2.1.1 Slow Start Over Satellite Channels

Kruse [Kru95] measured the RTT over the NASA ACTS [Bv91] satellite as approximately 560 ms. For comparison, we measured the RTT over the terrestrial Internet between Ohio University and the University of California at Berkeley as approximately 80 ms. Equation 1.1 gives the time it takes the slow start algorithm to reach a window size of W segments on a network with a RTT of R as defined by [JK88].

$$\text{slow start time} = R \log_2 W \quad (1.1)$$

Assuming 512 byte segments, a window size of 128 segments (maximum TCP window) and the RTTs given above, TCP running over the satellite network takes 3.92 seconds to increase $cwnd$ to the full advertised window. TCP running over the terrestrial network takes 560 ms to reach the same window size. While using the slow start algorithm, TCP can waste available bandwidth. Figure 1.1 shows a mathematical model of the total amount of data sent over both the terrestrial network and the satellite network as a function of time. In the time it takes TCP to achieve the advertised window (128 segments) in the satellite network, TCP is able to send roughly 22 times more data over the terrestrial network.

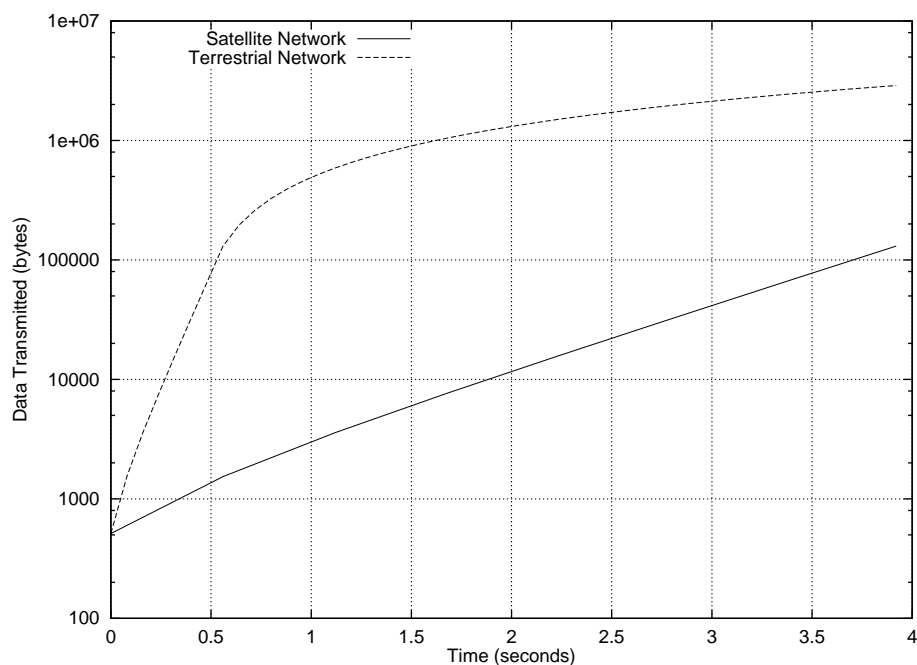


Figure 1.1 Slow Start Comparison

This figure shows a mathematical model of the total amount of data sent by TCP over a satellite network and a terrestrial network as a function of time. Both transfers start with a *cwnd* of 1 segment and increase the size of *cwnd* using the slow start algorithm. The amount of time shown represents the time it takes TCP over the satellite network to reach the advertised window (128 segments, in this case) using slow start.

1.2.1.2 Congestion Avoidance Over Satellite Channels

TCP uses congestion avoidance to probe the network for additional capacity after loss. Satellite channels increase the amount of time the congestion avoidance algorithm takes to increase *cwnd* when compared to terrestrial links. For example, when a TCP connection utilizing the maximum possible window (128 segments containing 512 bytes each) experiences a single segment drop, the value of *cwnd* is reduced to 64 segments. For each loss-free RTT the value of *cwnd* is increased by approximately 1 segment. Over the terrestrial network described above it takes 5.12 seconds to increase the value of *cwnd* size from 64 segments to 128 segments using congestion avoidance. Over the ACTS satellite channel the same increase takes 35.84 seconds. Each RTT

that TCP is not fully utilizing the available bandwidth represents lost throughput. Figure 1.2 presents a mathematical model illustrating the total amount of data the TCP congestion avoidance algorithm is able to send over both the satellite network and the terrestrial network as a function of time. The figure shows that when TCP is operating over a terrestrial network the increase to the advertised window is more rapid and therefore TCP is able to send almost 9 times more data over the terrestrial network in the same amount of time when compared to the satellite network.

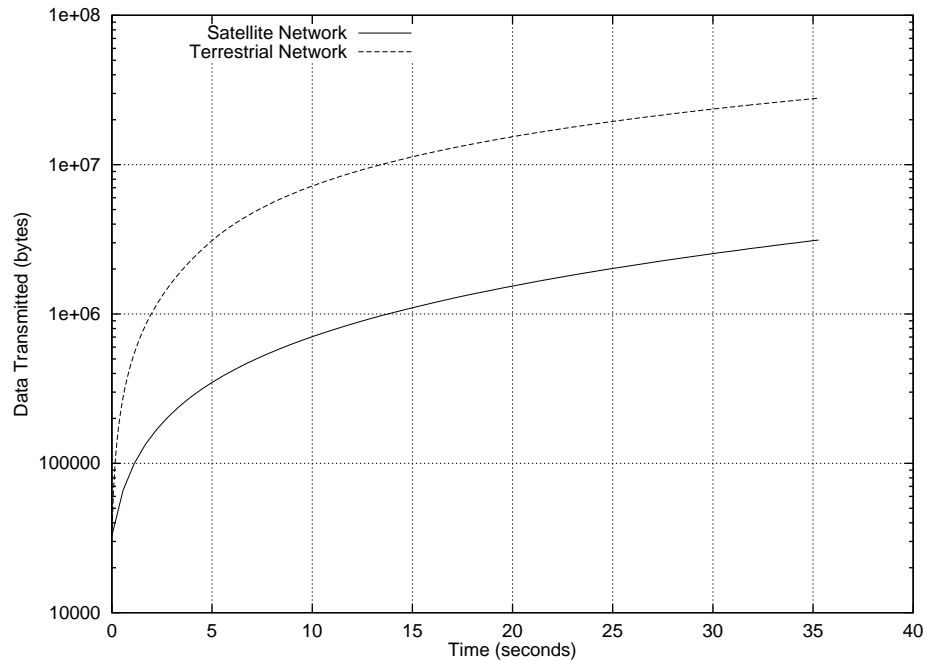


Figure 1.2 Congestion Avoidance Comparison

This figure shows a mathematical model of the total amount of data sent by TCP over a satellite network and a terrestrial network as a function of time. Both transfers start with a *cwnd* of 64 segments and increase the size of *cwnd* using the congestion avoidance algorithm until *cwnd* reaches the advertised window size (128 segments). The amount of time shown represents the time it takes TCP over the satellite network to reach the advertised window using congestion avoidance.

1.2.2 Problems With TCP Steady-State Behavior

The satellite channel studied in this paper has 1.536 Mbits/second (or 192,000 bytes/second) of capacity (T1 speed). An upper bound for TCP throughput is given in equation 1.2 [Pos81]. This equation assumes a loss-free network and a TCP that does not use any of the congestion control algorithms described in section 1.1.

$$\text{max throughput} = \frac{\text{receive window size}}{\text{round trip time}} \quad (1.2)$$

Therefore, when using the maximum TCP receive window of 65,535 bytes over a satellite channel (RTT of 560 ms) the upper bound on throughput is given by equation 1.3.

$$\text{max throughput} = \frac{65,535 \text{ bytes}}{560 \text{ ms}} \approx 117,027 \text{ bytes/second} \quad (1.3)$$

Clearly this upper bound on TCP throughput ensures that TCP will be unable to fully utilize the bandwidth provided by T1 satellite channels.

1.3 Possible Solutions

The problems outlined above can be mitigated using a number of approaches. One approach that has been studied is to break the TCP connection at the router before the satellite channel [BB95]. Using this mechanism, the router before the satellite channel accepts and ACKs all the segments sent to a machine across the satellite link. From the sender's perspective, the transfer is complete when the router ACKs all the segments. The router then sends the data to the host across the satellite channel. This makes the transfer "appear" much faster from the senders perspective. However, the ACKs received by the sender do not indicate that the data was received at the final destination and therefore, this mechanism breaks the end-to-end semantics of TCP. A similar idea is to make the router handle retransmissions for the sender [BSAK95].

Making the router repair all loss without notifying the TCP sender allows retransmission without making the sending TCP reduce the transmission rate. Researchers have also investigated compressing the TCP and IP headers in an effort to reduce the overhead, leaving more bandwidth for the data [Jac90a] [DENP96]. Finally, new queueing mechanisms, such as Random Early Detection [FJ93] [BCC⁺97], provide intermediate routers with a way to inform the TCP sender that congestion is occurring before the TCP sender sends an inappropriate amount of data. By giving TCP an early congestion indication, the router can force TCP to reduce the transmission rate before the TCP sender transmits an inappropriate amount of data forcing the router to discard a large number of segments. While potentially useful, these mechanisms are outside the scope of our investigation.

The first approach we studied was an application-level solution. We modified an FTP client and server to use multiple parallel TCP data connections to transfer a single file. While not a general purpose solution for improving utilization of satellite channels this version of FTP provided insights into the kinds of TCP mechanisms that may mitigate the problems introduced by satellite channels. XFTP and the lessons it provides are outlined in chapter 2. The second step of our work is to take the insights provided by XFTP and enhance the TCP protocol accordingly. One of the issues that XFTP illustrated was that using a larger initial sliding window enhances bandwidth utilization. We modified TCP to make the slow start algorithm more aggressive. The algorithm modifications and experimental results are outlined in chapter 3. Finally, chapter 4 provides conclusions, recommendations and outlines areas for future work.

2. AN APPLICATION-LEVEL SOLUTION

The first approach we used to better utilize the available bandwidth over satellite links was to employ multiple parallel TCP data connections. We altered an FTP client and server to use multiple parallel TCP data connections to transfer a single file. Our modified version of FTP is called *XFTP*¹. The changes to the FTP application-level protocol are outlined in appendix B and [AO97b]. This chapter is organized as follows. Section 2.1 outlines our application-level solution. Section 2.2 describes the various experimental environments we used to test XFTP. Section 2.3 presents the results of our experiments. Finally, section 2.4 relates the lessons of XFTP to mechanisms that could be used in TCP to help it better utilize the available capacity of satellite channels.

2.1 XFTP Theory

As outlined in equation 1.3 in the previous chapter, TCP is limited to a throughput of approximately 117,027 bytes/second over the satellite channel used in our experiments. This limit comes from the maximum TCP window size (65,535 bytes) and the delay imposed by satellite communications (560 ms RTT). When multiple TCP connections are used in parallel, an application can utilize a larger *effective TCP window*. Equation 2.1 shows that an application using 2 parallel TCP connections, each using a maximum window of 65,535 bytes, can fully utilize a satellite T1 link (192,000 bytes/second).

¹The name “XFTP” has been used by other researchers for experimental versions of FTP. Unless otherwise noted, “XFTP” refers to our modified version of FTP in this paper.

$$\text{max throughput} = \frac{2(65,535 \text{ bytes})}{560 \text{ ms}} \approx 234,054 \text{ bytes/second} \quad (2.1)$$

Our investigation found that the standard SunOS 4.1 FTP client and server use an advertised window of 24 KB², rather than the maximum window size of 65,535 bytes. Equation 2.2 shows that this smaller window further reduces the throughput FTP is able to achieve over satellite channels. Unless otherwise noted, the window size in all of our experiments is 24 KB so that valid comparisons with the standard SunOS FTP can be drawn.

$$\text{max throughput} = \frac{24,576 \text{ bytes}}{560 \text{ ms}} \approx 43,886 \text{ bytes/second} \quad (2.2)$$

Rearranging equation 1.2 gives equation 2.3 which defines the window size needed to fully utilize a channel with a given bandwidth and RTT.

$$\text{window size} = (\text{bandwidth})(\text{RTT}) \quad (2.3)$$

So, to fully utilize the capacity of the satellite circuit used in our study, a TCP window of 107,520 bytes is needed according to equation 2.4.

$$\text{window size} = (192,000 \text{ bytes/second})(560 \text{ ms}) = 107,520 \text{ bytes} \quad (2.4)$$

According to equation 2.5, XFTP must use at least 5 parallel data connections, each with a 24 KB window, to fully utilize a T1 satellite channel.

$$\left\lceil \frac{\text{effective window needed}}{1 \text{ connection window size}} \right\rceil = \left\lceil \frac{107,520 \text{ bytes}}{24,576 \text{ bytes}} \right\rceil = 5 \text{ connections} \quad (2.5)$$

2.2 Experimental Environment

We tested XFTP (and subsequent TCP modifications) in 3 environments. The satellite environment used for this research was the NASA ACTS satellite system

²In this paper, 1 KB = 1024 bytes.

[Bv91], which is in geosynchronous orbit. We also used a software emulator built at Ohio University in conjunction with this research and a commercially available hardware emulator to model the ACTS system when the satellite system was being used by other researchers. When conducting our tests, we did not share any of the 3 testbeds with competing network traffic. Each of these environments are discussed in detail below.

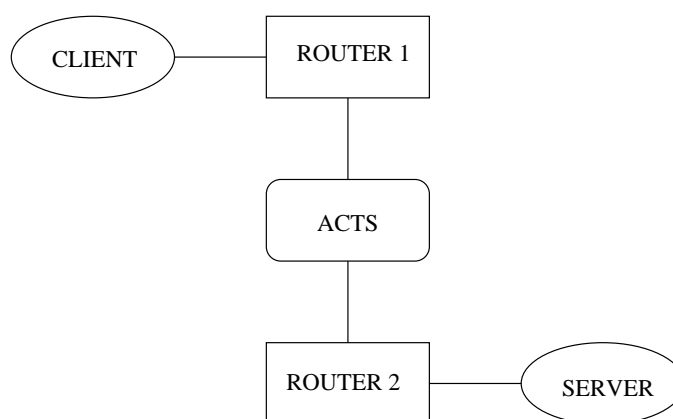


Figure 2.1 NASA ACTS Satellite System

This figure shows the network layout for our tests involving the NASA ACTS satellite system.

2.2.1 NASA ACTS

The network layout of the NASA ACTS testbed is shown in figure 2.1. Each physical network contains a Cisco 2514 router. The two routers are connected via the ACTS satellite. In this environment, the client and server are Sun IPX workstations running SunOS 4.1.3. We verified that the RTT was approximately 560 ms. The Cisco routers used in our experiments employed *drop-tail* queueing. When a router receives a segment to forward but is already in the process of transmitting another

segment, the incoming segment is placed in a queue for later processing. A router using *drop-tail* queuing discards incoming segments when the router has exhausted all available queue memory.

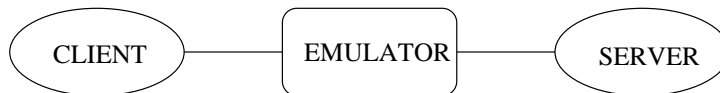


Figure 2.2 *ONE* Software Emulator Setup

This figure shows the network layout for all our tests involving the *ONE* software emulator.

2.2.2 Software Emulator

The software emulator used in our experiments is the Ohio Network Emulator (*ONE*) [ACO97]. The network layout for the experiments using *ONE* is shown in figure 2.2. The emulator runs on a Sun workstation running the Solaris operating system. The end points in this figure are a mixture of Sparc IPC and Intel 486 machines running NetBSD 1.1. The software emulator uses a drop-tail queuing strategy.

ONE passes segments between the two physical networks to which it is connected based on a number of user-configurable parameters. Each segment passing through the software emulator is subjected to 3 delays, as follows.

- **Transmission Delay:** This is the time it would take to transmit the segment on a network with the user-configured bandwidth.
- **Propagation Delay:** This is the user-configurable amount of time it takes a segment to travel the length of the channel being emulated.

- **Queue Delay:** This delay is determined by *ONE*. If a segment arrives while another is being serviced the new segment must sit in the queue and await transmission. The length of the queue determines the queue delay inserted by *ONE*.

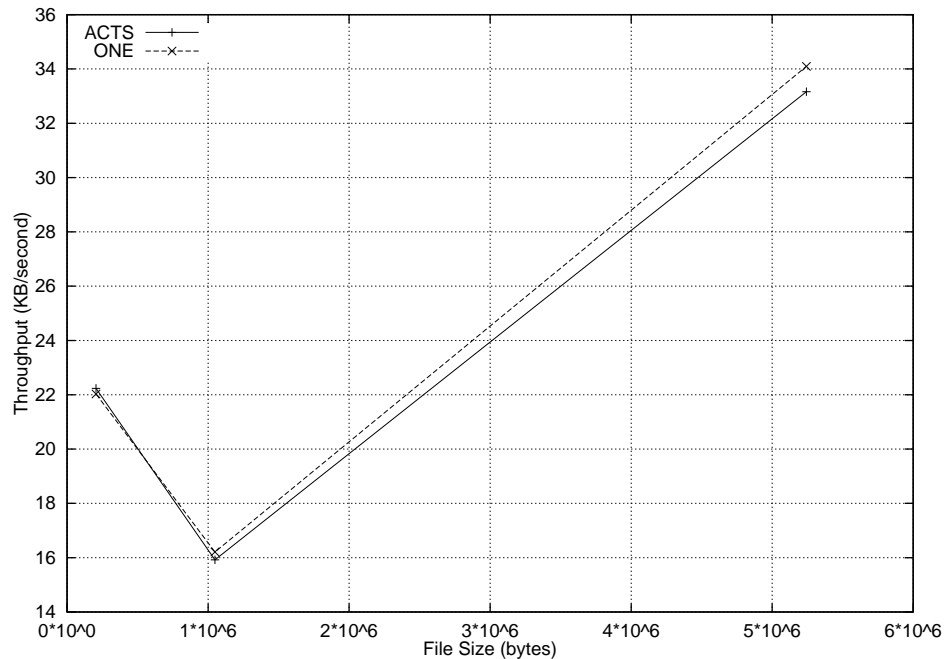


Figure 2.3 A Comparison of NASA ACTS and *ONE*

This figure shows the results of running identical tests using the NASA ACTS satellite system and the *ONE* software emulator.

ONE was configured to model the ACTS system and the same set of experiments run in both environments. As figure 2.3 shows, *ONE* accurately models the NASA ACTS satellite channel.

2.2.3 Hardware Emulator

The hardware emulator used in our experiments is the TestLink Data Link Simulator. The emulator runs on an Intel 486 machine equipped with two special purpose expansion cards that are designed to emulate the network configured by the user. The

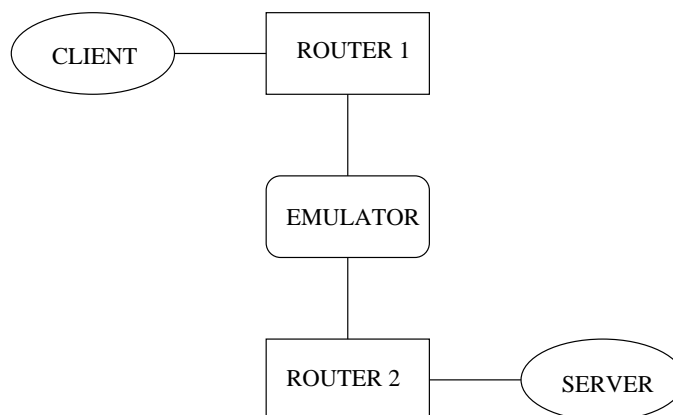


Figure 2.4 Hardware Emulator Setup

This figure shows the network layout for all tests involving the hardware emulator.

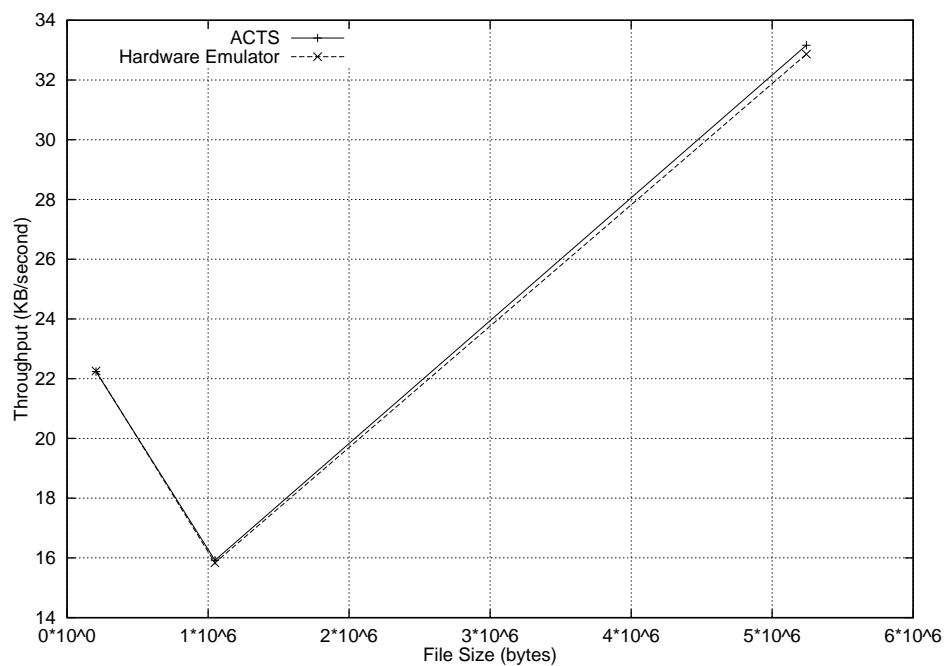


Figure 2.5 A Comparison of NASA ACTS and the Hardware Emulator

This figure shows the results of running identical tests using the NASA ACTS satellite system and the hardware emulator.

emulator is configured using a small software program. The network layout used in our experiments is illustrated in figure 2.4. As in the NASA ACTS experiments, Cisco 2514 routers employing drop-tail queueing are used. Furthermore, the mix of Sun IPC and Intel 486 NetBSD 1.1 machines used in the *ONE* testbed described above are again used as the end points for experiments utilizing the hardware emulator.

The hardware emulator limits the bandwidth between the routers by providing an appropriate clock pulse to the channel based on the user-configured bandwidth setting. The clock pulse determines the rate the routers can transmit segments. The queue delay is provided by the routers, just as in the ACTS system. The user-configured propagation delay is injected by the emulator delaying segments for the user-configured amount of time before forwarding them.

We configured the hardware testbed to model the ACTS environment and ran the same experiments in both environments. Figure 2.5 shows that the hardware emulator accurately models the ACTS satellite channel.

2.3 Experimental Results

Our first experiments showed that XFTP was able to utilize 84% of the available bandwidth over the satellite link by increasing the number of data connections used. Figure 2.6 presents the results of transferring a 5 MB³ file as a function of the number of data connections used. This figure illustrates a number of interesting findings.

As described in [Kru95] and predicted above in equation 2.2, using one TCP connection to transfer a file does not fully utilize the available bandwidth. The difference between the predicted throughput (equation 2.2) and the actual throughput (figure 2.6) can be explained by TCP/IP segment header overhead (approximately 8%).

³In this paper, 1 MB = 1024 * 1024 bytes = 1,048,576 bytes.

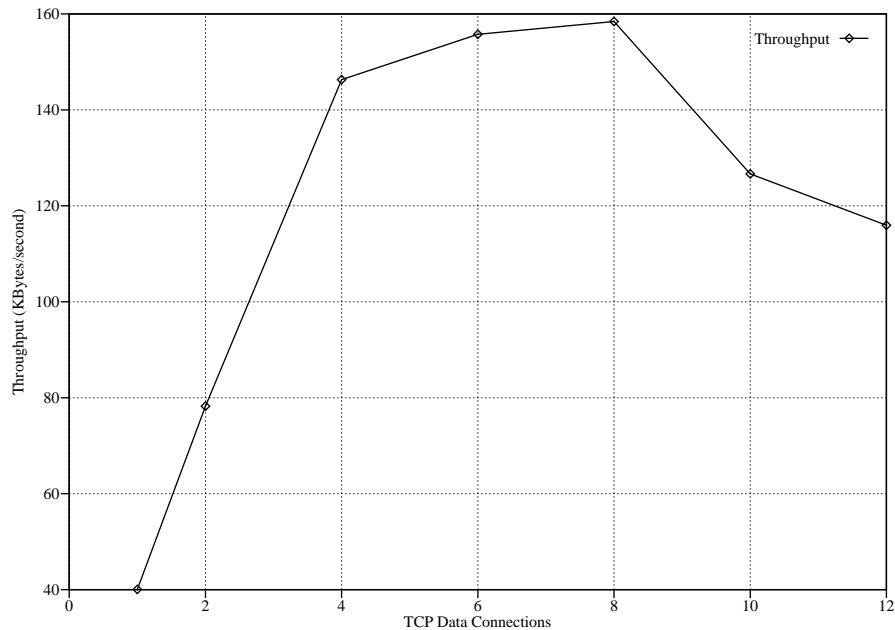


Figure 2.6 XFTP Test Results

This figure shows the throughput using XFTP to transfer a 5 MB file using the NASA ACTS satellite system plotted as a function of the number of parallel data connections employed.

Using 4 data connections gives XFTP an effective window of 98,304 bytes. This is approximately 91% of the window size needed to fully utilize the available bandwidth (according to equation 2.4). Using 4 data connections XFTP was able to utilize 87% of the available capacity after taking overhead into account.

As indicated in equation 2.5, using 4 data connections does not provide a large enough effective window to use the available capacity. However, 8 data connections provides an effective window of 196,608 bytes which is much larger than the 107,520 bytes needed. Yet, using 8 connections only yields a 84% utilization of the available bandwidth. Even when segment header overhead is taken into account, utilization is still only 92%.

Even though the required window size (107,520 bytes from equation 2.4) was exceeded, utilization continued to improve as connections were added through 8 connections. However, when 10 and 12 data connections were employed utilization dropped. Examination of network traces indicated a large loss event at the end of slow start on most of the TCP connections. This loss explains the reduction in throughput, since TCP must first waste time retransmitting lost data and then reduce the sending rate.

The cause of the loss can be directly explained by the slow start algorithm. Each data connection begins by exponentially increasing the sending rate every RTT until it fills the receive window or loss is detected. With multiple data connections doubling their sending rate every RTT, the intervening router is easily overwhelmed by the bursts of segments and must discard the segments it cannot queue. In the experiment shown in figure 2.6, the router was able to queue enough traffic to keep losses from occurring when 8 data connections were used by XFTP. However, when 10 connections were employed, the aggregate transmission rate overloaded the router's queue and a massive loss event ensued.

After these ACTS experiments, we were not able to explain why XFTP was only able to utilize 92% of the channel capacity by any TCP related problem. Upon profiling our XFTP client and server we found that transmitting and receiving were unnecessarily slow. XFTP writes and reads data in 8 KB chunks (see appendix B for details). The version of XFTP described above would write 8 KB at a time, even if the network was unable to accept the full 8 KB. When the operating system was unable to write the entire chunk, XFTP was forced to wait until the network could accept the rest. A similar problem was occurring on the receiving side of the connection. After the first round of tests, XFTP was modified to use asynchronous reading and writing calls. The modified version of XFTP transmits as much of the 8 KB chunk as possible and queues the rest for later transmission. This allows XFTP to transmit data on another data connection rather than waiting for the entire 8 KB chunk to be transmitted. A similar mechanism is employed on the receiving side of

the connection. In this way, XFTP is always writing or reading (unless all of the connections are unavailable). This optimization yields better network utilization, as shown in figure 2.7. Using 6–8 connections this version of XFTP was able to utilize approximately 90% of the available capacity. When segment header overhead is taken into account, XFTP is utilizing approximately 98% of the channel capacity.

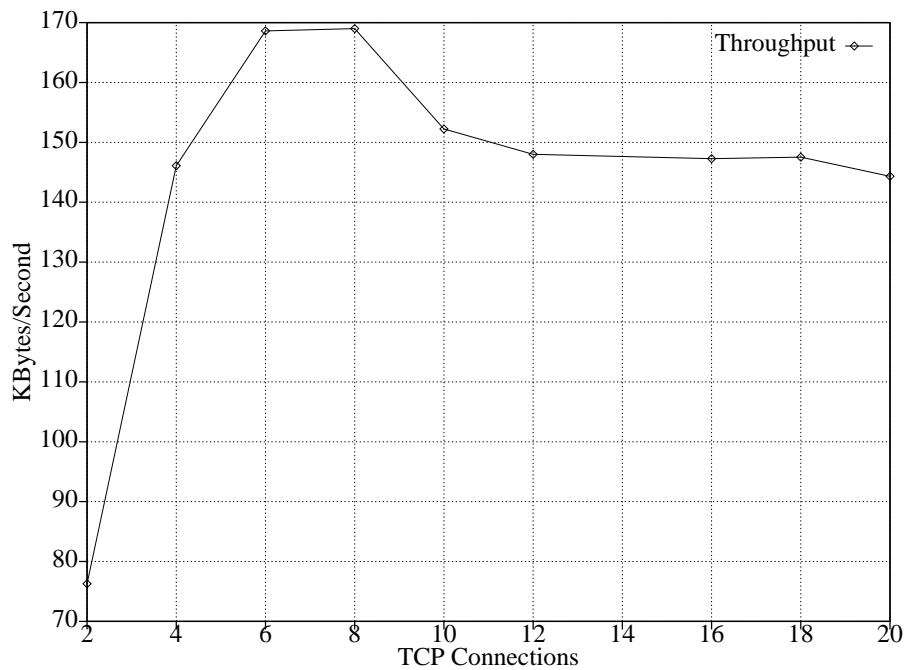


Figure 2.7 Modified XFTP Tests

This figure shows the throughput obtained using an asynchronous version of XFTP to transfer a 5 MB file across the hardware emulator plotted as a function of the number of parallel data connections employed.

2.4 XFTP Lessons

Studying XFTP's behavior has provided insight into TCP modifications that may improve performance over satellite channels. XFTP's use of N data connections generally makes it N times more aggressive than a single TCP connection. For example, slow start effectively begins by sending N segments rather than 1 segment. Similarly, congestion avoidance effectively adds N segments per RTT rather than just a single segment. Furthermore, XFTP is more aggressive in the face of dropped segments. For example, if each of N data connections are using a *cwnd* of m segments and a single segment is dropped on one of the connections, XFTP will reduce the *cwnd* to that given in equation 2.6 rather than reducing *cwnd* by half as a single TCP connection would. This reduction represents one of the N data connections reducing its *cwnd* by half, while the other $N - 1$ data connections continue using the same window size.

$$\left(\frac{2N - 1}{2N}\right) m \tag{2.6}$$

While XFTP is more aggressive than a single TCP connection it can provide insight into new mechanisms that will help TCP better utilize satellite channels. The lessons provided by XFTP include the need for larger windows, selective acknowledgments and possibly the need to make the standard TCP congestion control algorithms more aggressive.

2.4.1 Bigger Windows

The case for larger TCP windows is given in section 1.2.2. Furthermore, the XFTP experiments presented in 2.3 show that larger effective windows allow TCP to fully utilize the capacity of satellite channels. TCP options to provide larger windows have been defined [JB88] [BJZ90] [JBB92]. These extensions have been proven useful in networks with high latency [AHKO97] and in high bandwidth networks [VS95].

2.4.2 Selective Acknowledgments

XFTP shows that errors can be repaired better if TCP is given a better idea about which segments have been lost. Using N data connections allows TCP to detect up to N lost segments simultaneously. TCP is able to repair up to N lost segments using fast retransmit, as opposed to 1 lost segment when using a single TCP connection. This suggests that the TCP sender needs more information about which segments have arrived at the receiver. A *selective acknowledgment* (SACK) mechanism provides information about exactly which segments have arrived at the receiver. SACKs allow the TCP sender to retransmit only those segments that have been dropped by the network. A SACK is defined for TCP in [MMFR96]. SACKs have been proven useful in several research protocols (e.g., NETBLT [CLZ87], RDP [VHS84], VMTP [Che88]), as well as TCP [AHKO97] [FF96].

2.4.3 More Aggressive Slow Start

Our XFTP experiments show that one of the reasons TCP was not able to fully utilize the channel bandwidth even when using an adequate window size was the slow start algorithm. This has also been shown in companion research [AHKO97]. Adding more connections to an XFTP file transfer allows XFTP to better utilize the capacity. This can be explained by the increased rate at which XFTP increases the value of *cwnd*. XFTP effectively multiplies slow start by the number of connections being used. A single TCP connection can get this same effect by increasing the initial congestion window (currently 1 segment). The use of a larger initial window has been proposed recently [Flo97]. We have experimented with this more aggressive version of slow start and our results are presented in chapter 3.

2.4.4 More Aggressive Congestion Avoidance

XFTP also suggests that further study into the the congestion avoidance algorithm is needed. As outlined in section 1.2.1.2, congestion avoidance requires a large amount of time to increase the window size over satellite channels due to the long RTT. XFTP is able to multiply this increase by the number of connections being used (i.e., each data connection can increase the effective window by 1 segment per RTT). Future work in this area may include attempting to change the congestion avoidance algorithm to take the RTT into account when increasing *cwnd* rather than simply using the number of RTTs that have past since the loss occurred.

2.4.5 Slow Start Threshold Estimation

Finally, XFTP suggests that further study into choosing when TCP terminates the slow start algorithm is needed. Massive loss happens when XFTP utilizes too many data connections because the effective sliding window overwhelms the intervening router. TCP could avoid the majority of these losses, while maintaining good utilization, if it could terminate slow start just before the loss event occurs. The required window can be computed using equation 2.3. TCP already observes the RTT, but would need to estimate the available bandwidth of the bottleneck link to compute the required window. Once such an estimation is made the needed window should be assigned to the *ssthresh* variable (which determines when slow start is terminated).

3. SLOW START MODIFICATIONS

As discussed in section 2.4.3, XFTP is able to better utilize the available capacity of satellite links partly because it is able to increase the effective sliding window size more rapidly than a single TCP connection. As outlined in section 1.1.1, TCP begins each connection by using the slow start algorithm. This algorithm initializes *cwnd* to 1 segment. XFTP's use of N parallel data connections provides an effective initial *cwnd* of N segments. This increase in the effective initial size of *cwnd* is important because it reduces the time TCP spends in slow start, which reduces the amount of unused capacity (as discussed in section 1.2.1.1).

Section 3.1 outlines two slow start modifications that will reduce the amount of time TCP spends using the slow start algorithm. Section 3.2 presents results of tests using larger initial *cwnd* values. Section 3.3 presents results of tests employing our modified *cwnd* increase algorithm. Section 3.4 presents tests that employ both slow start modifications. Finally, section 3.5 outlines future work in this area.

3.1 Slow Start Modifications

We hypothesized that by increasing the initial size of *cwnd*, a single TCP connection may obtain channel utilization benefits similar to those obtained by XFTP. Floyd [Flo97] proposes allowing an initial *cwnd* of 4 segments (or 4380 bytes, whichever is smaller). This accelerated slow start behavior is only used at the beginning of a connection; if TCP reverts to slow start because the RTO expired, *cwnd* is re-initialized to 1 segment, as in standard TCP.

When an initial window of size W_I segments is used, TCP saves the amount of time normally associated with increasing the window to W_I segments ($\log_2 W_I$ RTTs as defined by [JK88]). Equation 3.1 is a generalized version of equation 1.1 that gives the amount of time needed to increase $cwnd$ from an initial $cwnd$ of W_I to an advertised window of W_A on a network with a RTT of R .

$$\text{slow start time} = R(\log_2 W_A - \log_2 W_I) \quad (3.1)$$

Clearly, equation 3.1 shows that for initial windows of more than 1 segment, the slow start time is reduced.

The second modification used to reduce the amount of time TCP spends using the slow start algorithm is to introduce a mechanism to counteract *delayed acknowledgments* as defined in RFC 1122 [Bra89]. Delayed ACKs allow TCP receivers to refrain from sending an ACK for each segment received. However, according to RFC 1122 an ACK must be generated for every other segment received. In addition, if a second segment does not arrive within a given timeout, the receiver must generate an ACK after the timeout (according to RFC 1122 the timeout must be no more than 500 ms). Since slow start increases the value of $cwnd$ by 1 segment for each ACK that arrives and delayed ACKs reduce the ACK arrival rate by roughly half, the rate at which $cwnd$ is increased is reduced. This reduction causes TCP to spend more time using the slow start algorithm to reach the advertised window. As discussed in section 1.2.1.1, the time spent in slow start can represent wasted bandwidth.

TCP can achieve the same increase rate regardless of how often the receiver generates ACKs if $cwnd$ is increased by the number of new segments an ACK covers, rather than increasing $cwnd$ by one segment for each ACK received. This rate of increase is consistent with that provided by the original definition of slow start [JK88], which predates the delayed ACK mechanism. Our implementation of this modified window increase algorithm takes place both at the beginning of the transfer and in slow start phases that follow the expiration of the RTO.

3.2 Experiments With Larger Initial Window

We used FTP to transfer files of varying sizes over the *ONE* testbed network, which was configured to model the ACTS satellite environment (as described in section 2.2.2). In the first set of tests, the receiver's advertised window was limited to ensure that no congestion loss occurred. Figure 3.1 presents the results of these tests, showing a plot of the throughput change between TCP with an initial *cwnd* of 1 segment (standard TCP) and TCP with larger initial *cwnd* values. As we hypothesized, increasing the initial value of *cwnd* reduced the total time of the transfer, thus increasing the throughput. This was most noticeable when transferring a small amount of data. For example, when sending 30 KB using a 32 segment initial window the throughput was improved by roughly 180%. Using the 4 segment initial window suggested by Floyd [Flo97] a 27% increase in throughput is shown for the 30 KB transfer. The throughput increase in short transfers can be explained by the close relationship between the total transfer time and the time saved by using a larger initial *cwnd* value. The throughput increase is not as large in the longer transfers because the total transfer time dominates the small number of RTTs saved by using a larger initial *cwnd* value. For instance, only a 3% increase in throughput is shown when using an initial *cwnd* value of 32 segments to transfer a 5 MB file. The suggested initial *cwnd* value of 4 segments did not degrade performance in any of the cases tested.

The tests outlined above were repeated using a larger advertised window, ensuring the TCP sender would overwhelm the intervening router queue (provided that the transfer is sufficiently long to allow TCP to increase the value of *cwnd* to the advertised window). Figure 3.2 presents the results of the second set of tests. As in the above tests, this figure illustrates the throughput change between standard TCP (with an initial *cwnd* of 1 segment) and TCP with a larger initial *cwnd* value. Figure 3.2 indicates that the shorter transfers (30 KB, 100 KB and 200 KB) performed almost

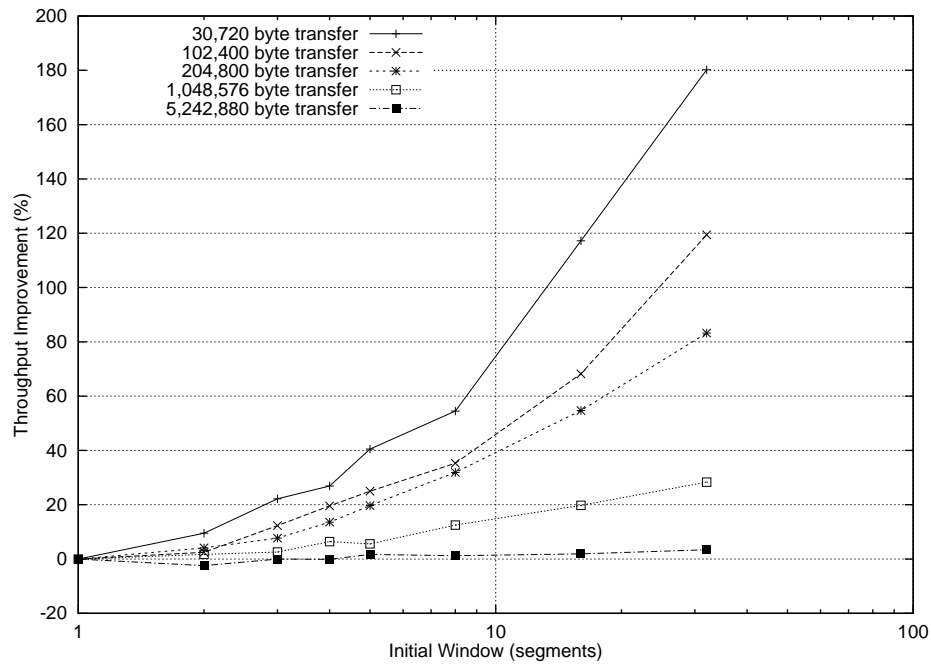


Figure 3.1 Larger Initial Windows Without Congestion

This figure shows the throughput improvement of modified TCP when compared to standard TCP (initial *cwnd* of 1 segment) plotted as a function of the initial *cwnd* size used. The advertised window in these tests ensured that TCP would not cause congestion loss by overwhelming the router queue.

identically to the transfers in the first set of tests. This indicates that the TCP sender completed the transfer before the value of *cwnd* was increased enough to overwhelm the intervening router queue. The 1 MB and 5 MB transfers show very little change in throughput across all initial *cwnd* values. Examination of network traces of the 1 MB and 5 MB transfers indicates that the loss pattern is similar across all initial *cwnd* values. The only difference we observed was the relative time at which the loss occurred. As the initial *cwnd* value increased, the relative time of the loss decreased. As in the first set of tests, using the suggested initial *cwnd* value (4 segments) did not degrade performance for any of the transfer sizes tested.

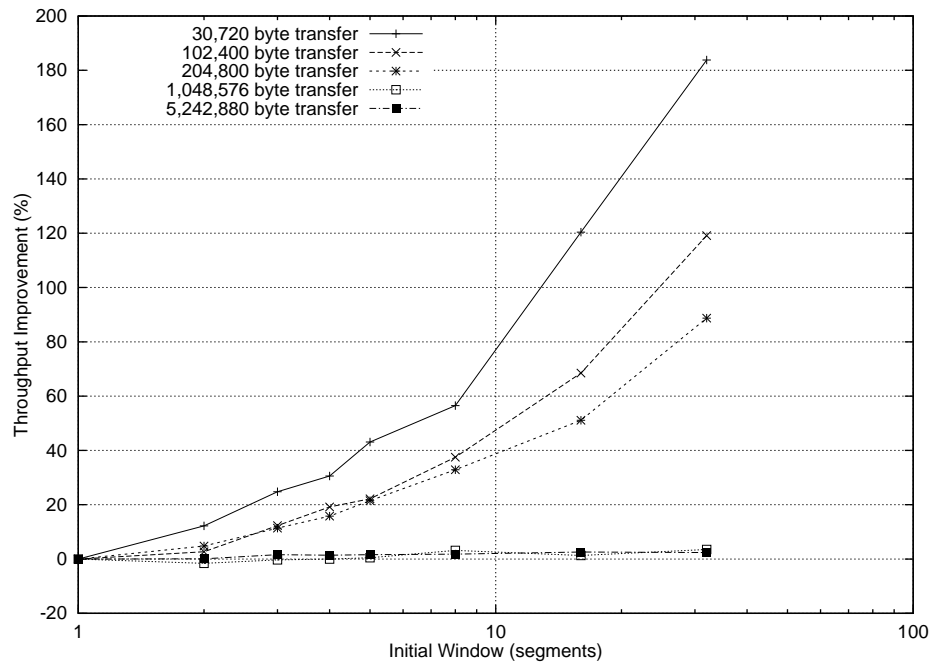


Figure 3.2 Larger Initial Windows With Congestion

This figure shows the throughput improvement of modified TCP when compared to standard TCP (initial *cwnd* of 1 segment) plotted as a function of the initial *cwnd* size used. The advertised window in these tests ensured that TCP would cause congestion loss by overwhelming the router queue, provided that the transfer was long enough to increase the size of *cwnd* to the advertised window.

3.3 Experiments With a New Window Increase Algorithm

To test the modified *cwnd* increase algorithm outlined in section 3.2, we used FTP to transfer files of various sizes using both the standard *cwnd* increase algorithm and our modified algorithm. As in the tests above, we used the *ONE* testbed for these tests. The first set of tests used a receive window that ensured a congestion-free environment. Table 3.1 presents the results of these tests. The table shows the throughput change between standard TCP and TCP employing our modified *cwnd* increase algorithm. Table 3.1 shows that all transfer sizes benefited from using our modified *cwnd* increase algorithm. The 100 KB and 200 KB transfers benefited the most from the new algorithm. Transfers of this size utilize the slow start algorithm for the entire transfer, so changes to slow start have a large impact on overall throughput.

The 30 KB transfer shows somewhat less improvement because it is too short to take full advantage of the modified algorithm. The 1 MB and 5 MB transfers show only minor improvements because slow start is only used during a small percentage of these transfers. Therefore, changes to slow start have little effect on the overall throughput of long transfers. Table 3.1 shows throughput gains using our modified *cwnd* increase algorithm across all transfer sizes tested.

File Size	Throughput Improvement (%)
30 KB	9.4
100 KB	16.9
200 KB	15.3
1 MB	8.5
5 MB	9.5

Table 3.1 New Window Increase Algorithm Without Congestion

This table shows the throughput improvement of TCP using our modified window increase algorithm when compared to standard TCP. The advertised window ensured that TCP would not cause congestion loss by overwhelming the router queue.

We repeated the above set of tests using a larger advertised window to ensure that the transmission rate will overwhelm the router queue (assuming the transfer is sufficiently long, so that the value of *cwnd* is increased to the advertised window size). The results of these tests are shown in table 3.2. As in the above set of tests, the throughput change between standard TCP and TCP using our modified *cwnd* increase algorithm is reported. The 30 KB and 100 KB transfers show throughput improvements of approximately 8% and 15% respectively when our *cwnd* algorithm is employed. These results are consistent with the first set of experiments involving our modified *cwnd* increase algorithm (table 3.1). This indicates that transfers of this size do not increase the value of *cwnd* enough to overwhelm the intervening router queue.

Our modified *cwnd* increase algorithm reduced the throughput of the 200 KB transfer by 36%. This throughput reduction indicates that when using our modified version of slow start, TCP is able to increase *cwnd* to the point at which the intervening router queue is overwhelmed. However, standard TCP is unable to increase *cwnd* enough to overwhelm the router and generate loss. This is not a flaw in our algorithm, as any version of TCP has a transfer size at which it will overwhelm the intermediate router. Our algorithm simply causes TCP to reach this “breaking point” for smaller transfers. The 1 MB and 5 MB transfers show little change in throughput. As in the last experiment, this is because TCP spends only a small percentage of the total transfer time using the slow start algorithm. Therefore, our modification of slow start have very little effect on the overall throughput.

File Size	Throughput Improvement (%)
30 KB	8.1
100 KB	14.6
200 KB	-36.2
1 MB	5.1
5 MB	3.7

Table 3.2 New Window Increase Algorithm With Congestion

This table shows the throughput improvement of TCP using our modified window increase algorithm when compared to standard TCP. The advertised window ensured that TCP would cause congestion loss by overwhelming the intermediate router, providing the transfer was long enough to increase *cwnd* to the advertised window size.

3.4 Experiments with Larger Initial Windows and a Modified Window Increase Algorithm

The following two sets of tests involve a version of slow start employing both larger initial values for *cwnd* and our modified *cwnd* increase algorithm. As with the previous tests outlined in this chapter, these tests were run using the *ONE* testbed. As in the above experiments, the first set of tests uses an advertised window that ensures no congestion loss. The results of these tests are presented in figure 3.3. This figure shows the throughput change between TCP using standard slow start and TCP using our modified slow start. As we hypothesized, figure 3.3 shows that when used together, the two modifications to slow start outlined above are able to increase the throughput more than either modification alone. Throughput is increased across all transfer sizes and all initial *cwnd* values.

The second set of experiments used an advertised window that ensured that congestion loss would occur provided the transfer was long enough to allow *cwnd* to reach the advertised window. Figure 3.4 presents the results of this set of tests. This figure plots the throughput change between standard TCP and TCP using our modifications. The 30 KB and 100 KB transfers were not able to increase *cwnd* enough to create loss. Therefore, the 30 KB and 100 KB transfers are nearly identical to those in the first set of tests (shown in figure 3.3). As predicted by tests outlined in section 3.3, the 200 KB transfers show reduced throughput when using both slow start modifications. As shown in table 3.2, using our *cwnd* increase algorithm caused loss in the 200 KB transfer while using the standard increase algorithm did not. Therefore, the 200 KB transfer showed a reduction in throughput when using both modifications to the slow start algorithm. The 1 MB and 5 MB transfers show little change when using both slow start modifications. This can be explained by the relatively small impact slow start has on the overall throughput of long transfers.

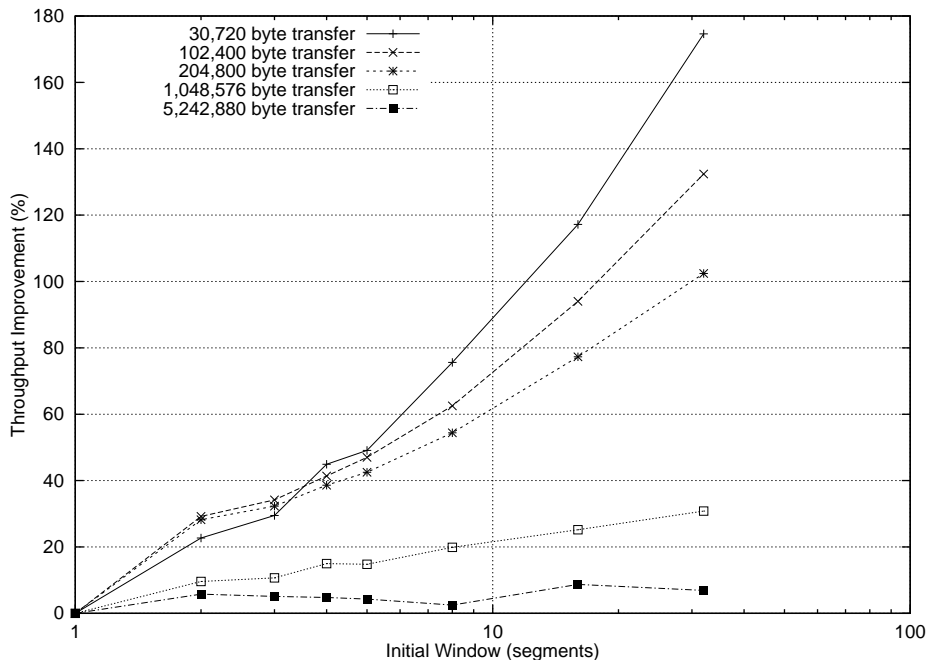


Figure 3.3 Larger Initial Windows and a Modified Window Increase Algorithm Without Congestion

This figure shows the throughput improvement achieved by our modified version of slow start when compared to standard TCP as a function of the initial *cwnd* size used. In addition to larger initial windows, our modified window increase algorithm is employed in all transfers except the standard TCP transfers. The advertised window in these tests ensured that TCP would not cause congestion loss by overwhelming the intermediate router.

3.5 Future Work

The test results presented above illustrate that the two slow start modifications outlined in section 3.1 increase overall throughput with little penalty over satellite channels. However, these modifications to slow start make TCP more aggressive and therefore further study is needed to ensure that they will not cause problems when competing traffic is present. In a companion study, we are testing these modifications over the global Internet [AHO97].

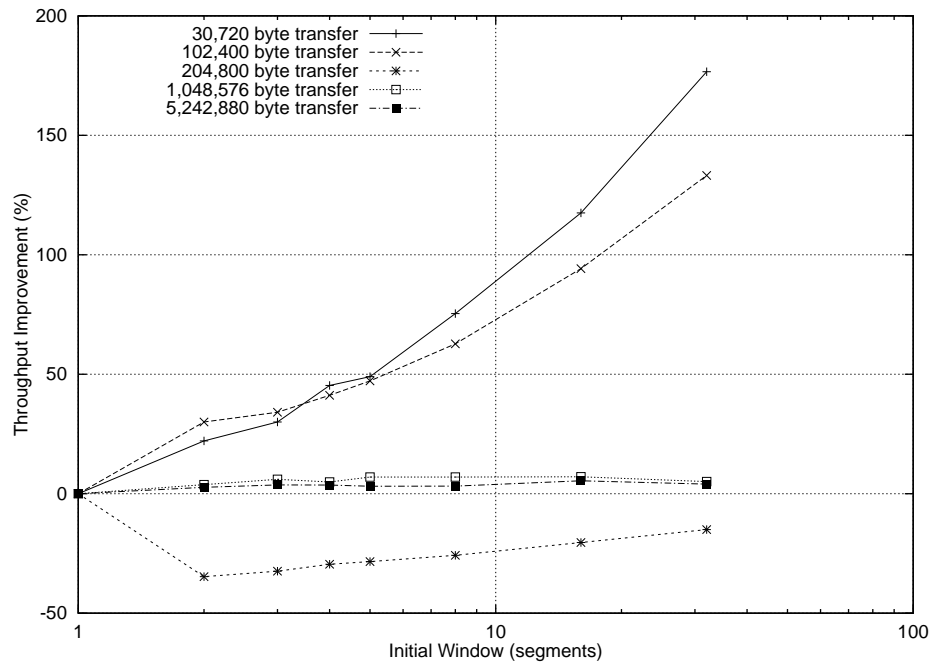


Figure 3.4 Larger Initial Windows and a Modified Window Increase Algorithm With Congestion

This figure shows the throughput improvement achieved by our modified version of slow start when compared to standard TCP as a function of the initial *cwnd* size used. In addition to larger initial windows, our modified window increase algorithm is employed in all transfers except the standard TCP transfers. The advertised window in these tests ensured that TCP would cause congestion loss by overwhelming the intermediate router, provided the transfer was long enough to increase the *cwnd* to the size of the advertised window.

4. CONCLUSIONS AND FUTURE WORK

We have developed a modified version of FTP that improves the utilization of satellite channels. However, we feel that our application-level solution, XFTP, is too aggressive to be used in general purpose networks. Furthermore, such a solution requires that modifications be made to all applications and servers in order to achieve full channel utilization over satellite links. However, the XFTP experiments presented in chapter 2 provided insights into TCP modifications that will help better utilize satellite circuits. The remainder of this chapter is divided as follows. Section 4.1 discusses the TCP mechanisms that have been thoroughly tested and we believe should be widely implemented. Section 4.2 discusses the TCP mechanisms that we believe are promising but need further study before they are widely implemented. Finally, section 4.3 provides final conclusions.

4.1 Recommendations for TCP Extensions

This section provides recommendations for TCP mechanisms that should be added to all general purpose TCP implementations. These mechanisms are needed to better the utilize satellite channels. Furthermore, these mechanisms have been proven to work well over the global Internet.

4.1.1 Large Windows

It has been shown in this paper (sections 1.2.2, 2.1 and 2.4.1) that for TCP to fully utilize the available capacity of satellite channels, large window TCP extensions [JBB92] are needed. Because the value of *cwnd* limits the amount of unacknowledged

data a TCP sender can inject into the network, advertising a window larger than permitted by standard TCP should not be harmful to the network. Larger advertised windows have been shown to work well in satellite networks [AHKO97] and over wide-area terrestrial networks [VS95].

4.1.2 Selective Acknowledgments

The need for SACKs has been shown in section 2.4.2. SACKs allow the receiver to report precisely which segments have arrived. This explicit information allows the sender to more effectively retransmit segments that have not been received. The sender is able to nearly eliminate retransmitting segments that have been successfully delivered because of using a poor retransmission strategy (i.e., using slow start to retransmit all segments that have not been cumulatively ACKed). In a companion study, the use of SACKs has been shown to be effective in the satellite environment [AHKO97]. In addition, SACKs have been shown to be beneficial in the face of varying amounts of loss in simulated terrestrial networks [FF96].

4.2 Recommendations for Future Modifications to TCP

This section outlines TCP mechanisms that have been proposed but not tested over satellite channels and in a shared network, such as the global Internet. These mechanisms show promise in helping TCP better utilize satellite channels as well as terrestrial links. Some of these TCP modifications are products of the work described in this paper, while others are products of other researchers and are included for completeness.

4.2.1 Slow Start Modifications

As shown in chapter 3, modifying the slow start algorithm can provide a throughput increase when transferring files over satellite channels. Furthermore, our tests show

little disadvantage in using these mechanisms in the satellite environment. However, these mechanisms increase TCP's aggressiveness and our tests did not include competing traffic. Therefore, further studies are needed to ensure these modifications do not negatively impact a shared network, such as the global Internet. We are currently involved in studying the impact of these slow start changes on the global Internet [AHO97].

4.2.2 Congestion Avoidance Modifications

As outlined in sections 1.2.1.2 and 2.4.4, TCP's congestion avoidance algorithm provides extremely slow *cwnd* increase over long-delay channels. Study into a version of congestion avoidance that is not biased against long-delay channels is needed. Such a mechanism should take into account both the number of RTTs that have passed, as well as the length of the RTT rather than simply the number of RTTs that have passed as the current mechanism uses.

4.2.3 Slow Start Threshold Estimation

Hoe [Hoe96] suggests using both the RTT and *packet-pair* bandwidth estimation [Kes91] to set the initial value of *ssthresh*. Packet-pair is a mechanism for determining the available bandwidth of the bottleneck link in the network between the sender and the receiver. Once the RTT and the available bandwidth are known, the appropriate window size can be computed using equation 2.3. Setting *ssthresh* to the appropriate window size for the current network conditions forces TCP to change from using the slow start algorithm to using the more conservative congestion avoidance algorithm before the slow start algorithm increases the value of *cwnd* to an inappropriate size causing a large number of segments to be dropped. Recent research has outlined problems with the packet-pair approach to bandwidth estimation and suggests that a receiver-based packet-pair mechanism may provide a better bandwidth estimate [Pax97]. Therefore, the estimation of *ssthresh* requires further study in both

satellite and terrestrial network environments before widespread deployment can be recommended.

4.2.4 New Loss Recovery Mechanisms

Improvements to the fast recovery algorithm that make data retransmission more effective have been suggested in the literature. Hoe [Hoe96] introduces a sender-side mechanisms called the “fast recovery phase.” This algorithm is used from the time a segment is retransmitted using the fast retransmit algorithm until all losses have been repaired. While in the fast recovery phase, TCP uses clues from incoming ACKs to determine which segments have not been delivered and therefore need retransmitted. This mechanism works better than standard TCP with fast recovery but not as well as TCP with SACKs [FF96]. An advantage of this mechanism over SACKs is that it only needs to be implemented on the sender-side of the TCP connection, rather than in both the sender and receiver TCP implementations as SACK requires.

TCP with SACK is explicitly given information about which segments have not arrived at the receiver. With this information, TCP can use more advanced retransmission techniques that decouple the decision of *what* to retransmit from the decision of *when* to retransmit. One such mechanism is the work by Fall and Floyd [FF96] that replaces the fast recovery algorithm. This algorithm is similar to that used by fast recovery, but less conservative. The algorithm attempts to determine the amount of outstanding data in the network and when this estimate becomes less than the congestion window, new data is transmitted. This algorithm takes advantage of the additional information provided by the SACKs to reduce the amount of time TCP uses retransmitting data without sending new data. This algorithm has been shown to work well in satellite networks [AHKO97] and in simulated terrestrial networks [FF96]. Another algorithm that uses SACKs and replaces fast recovery is *forward acknowledgments* (FACK) [MM96a] [MM96b]. FACK allows TCP to reduce the transmission rate gradually while retransmitting lost segments. Standard TCP

is unable to do this, because standard TCP senders do not know which segments need retransmitted. Further investigation is needed to determine which fast recovery replacement should be widely implemented.

4.3 Conclusions

Our application-level solution, as well as work done by other researchers, shows that TCP extensions providing larger windows and selective acknowledgments allow TCP to achieve good performance over satellite channels. Furthermore, our application-level solution suggested that changes to TCP's slow start and congestion avoidance algorithms could also improve performance. The experiments presented in this paper verified that modifications to slow start improved channel utilization. The extensions we tested, as well as the work cited above by other researchers, allow the TCP protocol to fully utilize long-delay satellite channels.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [ACO97] Mark Allman, Adam Caldwell, and Shawn Ostermann. ONE: The Ohio Network Emulator. Technical Report TR-19972, Ohio University Computer Science, June 1997.
- [AHKO97] Mark Allman, Chris Hayes, Hans Kruse, and Shawn Ostermann. TCP Performance Over Satellite Links. In *Proceedings of the 5th International Conference on Telecommunication Systems*, March 1997.
- [AHO97] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP Slow Start Modifications, 1997. In preparation.
- [AO97a] Mark Allman and Shawn Ostermann. FTP Extensions for Variable Protocol Specification, March 1997. Internet-Draft draft-allman-ftp-variable-04.txt (work in progress).
- [AO97b] Mark Allman and Shawn Ostermann. Multiple Data Connection FTP Extensions. Technical Report TR-19971, Ohio University Computer Science, February 1997.
- [BB95] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS)*, May 1995.
- [BCC⁺97] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, March 1997. Internet-Draft draft-irtf-e2e-queue-mgt-00.txt (work in progress).
- [BJZ90] Robert Braden, Van Jacobson, and Lixia Zhang. TCP Extension for High-Speed Paths, October 1990. RFC 1185.
- [BLFN96] Tim Berners-Lee, R. Fielding, and H. Nielsen. Hypertext Transfer Protocol – HTTP/1.0, May 1996. RFC 1945.

- [Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.
- [BSAK95] Hari Balakrishnan, Srinivasan Seshan, Etan Amir, and Randy Katz. Improving TCP/IP Performance Over Wireless Networks. In *ACM MobiCom*, November 1995.
- [Bv91] R. Bauer and T. vonDeak. Advanced Communications Technology Satellite (ACTS) and Experiments Program Descriptive Overview. Technical report, NASA Lewis Research Center, 1991.
- [Che88] David Cheriton. VMTP: Versatile Message Transaction Protocol, February 1988. RFC 1045.
- [CLZ87] David Clark, Mark Lambert, and Lixia Zhang. NETBLT: A High Throughput Transport Protocol. In *ACM SIGCOMM*, pages 353–359, August 1987.
- [Com95] Douglas E. Comer. *Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture*. Prentice Hall, 3rd edition, 1995.
- [DENP96] Mikael Degermark, Mathias Engan, Bjorn Nordgren, and Stephen Pink. Low-Loss TCP/IP Header Compression for Wireless Networks. In *ACM MobiCom*, November 1996.
- [FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, July 1996.
- [FF97] Sally Floyd and Kevin Fall. Router Mechanisms to Support End-to-End Congestion Control. Technical report, LBL, February 1997. Submitted to SIGCOMM.
- [FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.
- [FJGFBL97] R. Fielding, Jeffrey C. Mogul, Jim Gettys, H. Frystyk, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, January 1997. RFC 2068.
- [Flo97] Sally Floyd, February 1997. Note to end2end-interest mailing list.
- [Hoe96] Janey C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *ACM SIGCOMM*, August 1996.

- [Jac90a] Van Jacobson. Compressing TCP/IP Headers For Low-Speed Serial Links, February 1990. RFC 1144.
- [Jac90b] Van Jacobson. Modified TCP Congestion Avoidance Algorithm. Technical report, LBL, April 1990. Email to the end2end-interest mailing list. URL: <ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt>.
- [JB88] Van Jacobson and Robert Braden. TCP Extensions for Long-Delay Paths, October 1988. RFC 1072.
- [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 *ACM SIGCOMM*, 1988.
- [Kes91] Srinivasan Keshav. A Control Theoretic Approach to Flow Control. In *ACM SIGCOMM*, pages 3–15. SIGCOMM, ACM, September 1991.
- [KL86] B. Kantor and P. Lapsley. Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News, February 1986. RFC 977.
- [KP87] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *ACM SIGCOMM*, pages 2–7, August 1987.
- [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] 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/>.
- [MM96b] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, August 1996.
- [MMFR96] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.
- [Nag84a] John Nagle. Congestion Control in IP/TCP Internetworks, January 1984. RFC 896.

- [Nag84b] John Nagle. Congestion Control in IP/TCP Internetworks. *Computer Communication Review*, 14(4), October 1984.
- [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.
- [Pos82] Jon Postel. Simple Mail Transfer Protocol, August 1982. RFC 821.
- [PR85] Jon Postel and Joyce Reynolds. File Transfer Protocol (FTP), October 1985. RFC 959.
- [Ste97] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.
- [VHS84] D. Velten, Robert Hinden, and J. Sax. Reliable Data Protocol, July 1984. RFC 908.
- [VS95] Curtis Villamizar and Cheng Song. High Performance TCP in ANSNET. *Computer Communications Review*, 24(5):45–60, October 1995.
- [WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Volume II: The Implementation*. Addison-Wesley, 1995.

APPENDIX

A. TCP CONGESTION CONTROL ALGORITHMS

Table A.1 provides an example of TCP’s congestion control algorithms. In this example, the receiver’s advertised window is 10 segments and the transfer consists of 31 segments. The example is presented from the viewpoint of the sending host. Each event within a RTT is given its own row (and event number). A double horizontal line separates RTTs from one another. When an ACK arrives that is not triggered by the expected data segment the triggering segment is given in parenthesis after the sequence number contained in the ACK in the “Acknowledgment Received” column.

- **RTT: 1**

The value of *cwnd* is initialized to 1 segment and segment 1 is transmitted. The value of *ssthresh* is initialized to the receiver’s advertised window (10 segments).

- **RTT: 2**

The ACK generated by the receiver contains sequence number 2. This indicates that segment 1 has arrived (and segment 2 is expected next). Since *cwnd* is less than *ssthresh*, the slow start algorithm is in effect and *cwnd* is incremented by 1 segment. This increases *cwnd* to 2 segments and segments 2 and 3 are transmitted.

- **RTT: 3**

Slow start continues as ACKs covering segments 2 and 3 arrive. This leads to increasing *cwnd* to 4 segments and transmitting segments 4–7.

- **RTT: 4**

Again, slow start continues as ACKs covering segments 4–7 arrive. These ACKs

Event Number	Segment(s) Sent	Acknowledgment Received	<i>cwnd</i>	<i>ssthresh</i>	Unacknowledged Segments
1	1	–	1	10	1
2	2,3	2	2	10	2–3
3	4,5	3	3	10	3–5
4	6,7	4	4	10	4–7
5	8,9	5	5	10	5–9
6	10,11	6	6	10	6–11
7	12,13	7	7	10	7–13
8	14,15	8	8	10	8–15
9	16,17	9	9	10	9–17
10	18,19	10	10	10	10–19
11	20	11	10	10	11–20
12	–	11 (12)	10	10	11–20
13	–	11 (13)	10	10	11–20
14	11	11 (14)	8	5	11–20
15	–	11 (15)	9	5	11–20
16	–	11 (16)	10	5	11–20
17	–	11 (17)	10	5	11–20
18	–	11 (18)	10	5	11–20
19	–	11 (19)	10	5	11–20
20	–	11 (20)	10	5	11–20
21	21–25	21 (11)	5	5	21–25
22	26	22	≈ 5.2	5	22–26
23	27	23	≈ 5.4	5	23–27
24	28	24	≈ 5.6	5	24–28
25	29	25	≈ 5.8	5	25–29
26	30,31	26	≈ 6	5	26–31
27	–	27	6.17	5	27–31
28	–	28	6.33	5	28–31
29	–	29	6.49	5	29–31
30	–	29 (30)	6.64	5	29–31
31	–	29 (31)	6.79	5	29–31
32	29	–	1	≈ 3.4	29–31
33	–	32 (29)	2	≈ 3.4	–

Table A.1 Congestion Control Algorithm Example

This table shows how the four basic congestion control algorithms (slow start, congestion avoidance, fast retransmit and fast recovery) work as defined by Jacobson [JK88].

increase *cwnd* to 8 segments and 8 new segments are transmitted (segments 8–15).

- **RTT: 5**

Events 9 and 10 are continuations of slow start. ACKs covering segments 8 and 9 arrive. Each of these arrivals triggers an increase of *cwnd* by 1 segment and the transmission of 2 new segments. After these first two events, slow start ends, as *cwnd* has reached the receive window. At this point, each ACK will only trigger the transmission of 1 segment, as illustrated by event 11. In this event, the ACK covering segment 10 arrives, leaving 9 unacknowledged segments in the network. This causes a new segment (segment 20) to be transmitted.

Event 12 is the arrival of a duplicate ACK covering segment 10 (which was triggered by the arrival of segment 12 at the receiver). This is followed by the arrival of two additional duplicate ACKs covering segment 10 (event 14). When the third duplicate ACK arrives (triggered by the reception of segment 14), fast retransmit is used to retransmit segment 11. Furthermore, *ssthresh* is set to half of *cwnd*'s current value. *cwnd* is also halved, but fast recovery artificially inflates *cwnd* by the number of duplicate ACKs that have arrived (3). So, *cwnd* is set to 8 segments. The last event in this RTT is the arrival of a fourth duplicate ACK covering segment 10 (triggered by the receipt of segment 15). This causes *cwnd* to be artificially incremented by 1 segment, making it 9 segments. However, nothing can be transmitted because *cwnd* is still less than the number of unacknowledged segments in the network (10).

- **RTT: 6**

Event 16 is the arrival of the fifth duplicate ACK covering segment 10. This causes a 1 segment artificial increment of *cwnd* (now 10 segments). However, no new data can be sent, because there are still 10 unacknowledged segments in the network. Four more duplicate ACKs follow (triggered by segments 17–20).

These duplicate ACKs do not increase *cwnd* since *cwnd* has reached the size of the receive window. The last event that occurs in this RTT is the arrival of an ACK covering segment 20. This ACK was triggered by the arrival of the retransmission of segment 11. This causes *cwnd* to be reduced to its value prior to the artificial inflation (5 segments). Since there are no unacknowledged segments in the network, 5 new segments (segments 21–25) are transmitted.

- **RTT: 7**

This RTT shows ACKs for segments 21–24 arriving and one new data segment being sent for each of these ACKs. In addition, since TCP is now using the congestion avoidance algorithm *cwnd* is being incremented by $1/cwnd$ for each ACK received. These partial segments add up to a full segment by the end of the RTT. The last event of the RTT shows that *cwnd* is 6 segments. Therefore, in response to the ACK covering segment 25, TCP is able to transmit two segments (30 and 31).

- **RTT: 8**

Events 27–29 show the arrival of ACKs (covering segments 26–28). Since the transfer is only 31 segments long no new data is transmitted in response to these ACKs. However, *cwnd* is being incremented by a $1/cwnd$ for each ACK received. Events 30 and 31 show duplicate ACKs covering segment 28 arriving (triggered by segments 30 and 31). This indicates that segment 29 was dropped.

- **RTT: 9**

Since a third duplicate ACK never arrives the TCP sender is forced to wait for the RTO to expire. After the RTO expires, segment 29 is transmitted. Since this segment was resent due to the expiration of the RTO, *ssthresh* is set to half of the value of *cwnd* and *cwnd* is set to 1 segment.

- **RTT: 10**

This shows the arrival of the ACK covering segment 31. This ACK was triggered by the retransmission of segment 29. This ACK increments *cwnd* by 1 segment, since slow start is being used (since *cwnd* is less than *ssthresh*).

B. FTP PROTOCOL MODIFICATIONS

B.1 Introduction

This appendix outlines the modifications made to the FTP protocol [PR85] so that FTP is able to use multiple parallel TCP data connections to transfer a single file. These extensions include three new “FTP commands” (commands exchanged between the FTP client and the FTP server on the control connection), a new user mechanism for requesting multiple data connections, and a mechanism for dividing a file across multiple data connections.

B.2 FTP Commands

B.2.1 MULT

The MULT command is a question sent from the client to determine if the server supports the use of multiple data connections. This command has no arguments. If arguments are given, the server MUST return an error code of 500 (“command not understood”, as defined by [PR85]). If the server does not implement multiple connections, it will not understand the MULT command and therefore must return an error code of 500 (“command not understood”) according to [PR85]. If the server can accept multiple data connections, it MUST respond with a return code of 200 (“Command OK”). The string portion of the response SHOULD contain the maximum number of connections the server supports. If used, the format of the response string is:

```
X data connections available.
```

For example:

```
16 data connections available.
```

B.2.2 MPRT

The MPRT command MUST be used instead of the PORT command when multiple data connections are to be employed. The syntax of the MPRT command is an extension of the EPRT command, as defined by [AO97a]. The syntax of the MPRT command is:

```
MPRT <SPACE><net-prot>|<trans-prot>|<net-addr>|<trans-addr(s)><CRLF>
```

All fields except the last are unchanged from the definitions given in [AO97a], with one exception: if the <trans-prot> field is TCP, the <trans-addr(s)> can contain any number of TCP port numbers, separated by commas (“,”). For example:

```
MPRT IP4|TCP|132.235.34.34|2345,6789,1212
```

If the number of ports given exceeds the number of data connections supported by the server (as returned in response to the MULT command), the extra ports MUST be ignored (that is, the server MUST NOT open a data connection on these extra ports).

B.2.3 MPSV

The MPSV command MUST be used instead of the PASV command when multiple data connections are to be employed. The MPSV command is issued with no arguments. The text response to the MPSV command is similar to the response to EPSV defined in [AO97a]. The response MUST be:

```
(<net-prot>|<trans-prot>|<net-addr>|<trans-addr(s)>) \
<text indicating server is entering passive mode>
```

All fields within the parenthesis are unchanged from the response to EPSV defined in in [AO97a] except the <trans-addr(s)> field. If the transport protocol is TCP, the <trans-addr(s)> field can contain any number of TCP port number separated by commas (“,”). For example:

```
(IP4|TCP|132.235.34.34|3434,7865,9934) Entering passive mode.
```

If the number of ports listed in the MPSV response exceeds the maximum number of connections supported by the host issuing the MPSV command, the extra ports MUST be ignored. That is, no data connection should be opened on these ports.

B.2.4 Discussion

The MULT command is not required for correct operation of the system, but its use is RECOMMENDED before the MPRT command. In the case when the MULT command is not used and multiple data connections are not supported, an error code of 500 (“command not understood”) will be returned. In this case, the client and server must revert to using a single data connection. If the MULT command is used (and both client and server make use of the MULT response string in the recommended way) the host opening the data connections passively will not waste time and resources opening data connections that will be unused.

B.3 Enabling Multiple Data Connections

For the user to enable the use of multiple connections, a new mechanism must be added to the client. Once the user attempts to activate multiple connections using this mechanism, the client SHOULD send the MULT FTP command to the server. The client may allow the user to specify the number of connections to be used for the transfer. If such a choice is provided, the client SHOULD place a limit on the user’s choice (see section B.5). Furthermore, the client SHOULD NOT allow the user’s

choice to be greater than the number of data connections supported by the server (obtained from the response to the MULT command).

For example, a simple ASCII interface might provide a new “user command” (issued to the client by the user) called “multiple”. Without an argument, this command would trigger an attempt to activate multiple data connections using a default number of connections. With a numeric argument, the command would attempt to activate the number of data connections requested by the user (subject to the limits imposed by the client and server).

B.4 Dividing a File Across Multiple Connections

In order to transfer a file across multiple connections, the file MUST be broken up into 8192 byte chunks. Each chunk must be prepended with a 4 byte sequence number to form a record (as shown in figure B.1). The sequence numbers begin at 0 and increase by 1 for each subsequent record. The sequence number multiplied by the chunk size (8192 bytes) forms an offset from the beginning of the file to the location of the record’s data. The sequence number is necessary to reconstruct the file, as each record may be independently transmitted using any one of the established data connections.

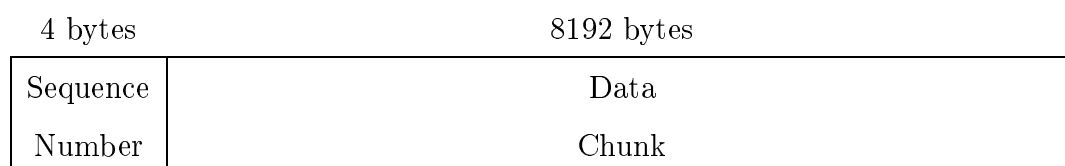


Figure B.1 XFTP Record Format

B.5 Recommended Limits

These extensions to FTP can allow a user to inadvertently flood the network with traffic. Therefore, the client and server SHOULD set limits on the number of data connections each is willing to support. The recommended limit for both the client and the server is 8 data connections. Furthermore, it is recommended that if a client employs a default number of connections, this default be 4. These limits will minimize the user's ability to inadvertently flood the network.

ALLMAN, MARK. M.S. June, 1997
Electrical Engineering

Improving TCP Performance Over Satellite Channels (64 pp.)

Director of Thesis: Shawn Ostermann

This thesis outlines performance problems with the Transmission Control Protocol (TCP) in networks containing geosynchronous satellite channels, as well as solutions to these problems. The first solution is a modification of the File Transfer Protocol (FTP). Standard FTP uses one TCP data connection to transfer a single file. However, our modifications allow FTP to employ multiple parallel TCP data connections to transfer a single file. Experiments have shown that this approach allows FTP to utilize the full capacity of the NASA ACTS satellite system. This application-level solution also provided insights into general TCP modifications that will allow all applications to fully utilize the available bandwidth provided by satellite channels. The insights provided by XFTP include the need for larger TCP windows, selective acknowledgments, changes to the slow start and congestion avoidance algorithms and the need for available bandwidth estimation. We altered the slow start algorithm to be more aggressive, as suggested by XFTP. Experiments using these slow start modifications show better utilization of satellite channels. Finally, this paper presents recommendations for which TCP mechanisms should be widely implemented and which mechanisms need further study.

Approved: _____