HTTP Page Transfer Rates over Geo-Stationary Satellite Links

Hans Kruse
J. Warren McClure School, Ohio University; hkruse1@ohiou.edu
Mark Allman
NASA Lewis Research Center/Sterling Software; mallman@lerc.nasa.gov
Jim Griner, Diepchi Tran
NASA Lewis Research Center; {jgriner, dtran}@lerc.nasa.gov

Introduction

The explosive growth of the Internet is being fueled by the popularity of the World Wide Web. A large amount of multimedia data is readily available on the web, making the physical distance between the information provider and the information user irrelevant. This development also produces pressure to extend the Internet, and with it the web, into geographic areas that have traditionally been far away from information sources. Rural areas, temporary installations, and developing nations all need Internet access. Geostationary satellites offer one possible avenue for providing this access.

Any Internet application operating over a geostationary satellite link must contend with round trip times (RTTs) in excess of 500ms. Protocol design decisions with minimal performance impact on low-delay links can have profound effects on satellite links [Kru95]. We present in this paper a comprehensive experimental performance study of the current HTTP protocol (HTTP 1.0) using the geostationary NASA Advanced Communications Technology Satellite (ACTS) system. We examine the time required for the retrieval of a number of different test web pages. In the experiment we evaluate the effect of using multiple TCP connections in the web browser (many browsers permit the user to control this setting). We also look at the impact of using persistent TCP connections, which most web servers allow, and the use of various TCP window sizes.

We begin the paper with a description of the network used for this experiment. We then detail the TCP protocol stack used for this study. To correctly evaluate the performance of HTTP, we felt that a TCP implementation close to the documented standard should be used. Therefore, while we start with the NetBSD UNIX implementation, we apply a number of fixes to the TCP stack. We also implement, for part of this study, the "4K" slow start modification that has been suggested for improved performance of web page transfers.

We then present the experimental data collected, and draw a number of conclusions from that data. We conclude with a number of suggestions for optimizing performance of satellite based web transfers, and suggest further research that should be carried out.

Experiment Configuration

Network Layout

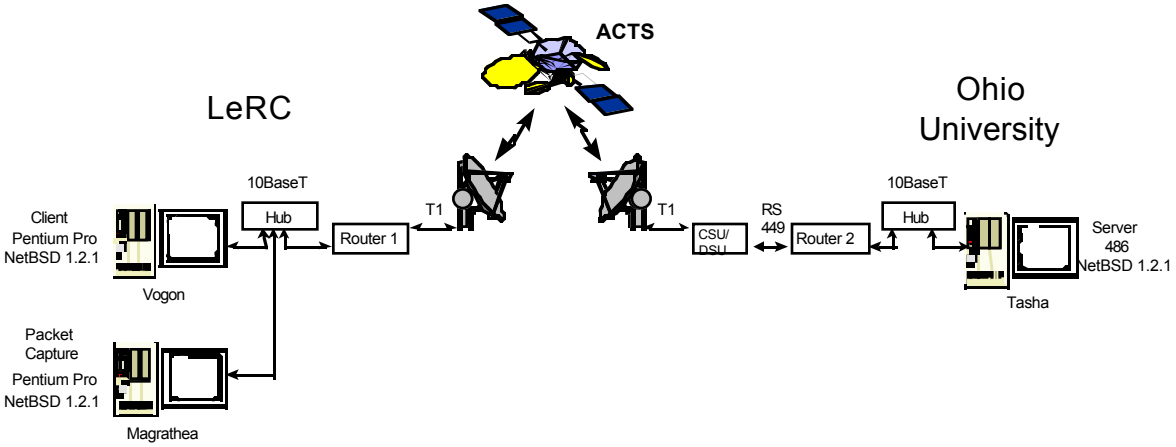

Figure 1: Network Topology

The network layout for these experiments is shown in Figure 1. The computers on the Lewis Research Center (LeRC) side of the testbed are Pentium Pro 180Mhz computers, running NetBSD 1.2.1. These two machines are connected to Router 1 via a 10BaseT hub. Router 1, using a built-in T1 (1.536Mbps) CSU/DSU, is connected to a VSAT earth station. A full duplex T1 link is setup between this earth station, through ACTS, to a similar earth station at Ohio University (OU). The second earth station is connected through a T1 CSU/DSU to a second router, via RS449. This router is connected through a 10BaseT hub to a 486 33Mhz computer at Ohio University, which is also running NetBSD 1.2.1.

The queue sizes in the routers have been adjusted to provide adequate buffering up to the T1 data rate of the circuit. These queue sizes have been shown to prevent packet loss in previous bulk-transfer experiments [All97b]. The satellite circuit provided by the ACTS system is essentially error-free.

Experiment Setup

The experimental setup consists of running an HTTP browser (Netscape 3.01) on the LeRC side of the satellite link, and an HTTP server (Apache 1.2b11) on the OU side. The configuration file settings for both the client and server can be found in Appendix A. In order to capture HTTP packets for later analysis, a second computer on the LeRC side is running the tcpdump packet capture program[1].

Test Pages

Four test web pages were accessed by the browser, each having a different number of elements and varying overall page size as shown in Table 1.

Page	# of Elements	Size (Kbytes)[2]	Next Page Delay (s)
/LeRC	4	48	30
/acts	9	99	30
/oufr	10	491	59
/Test	28	28	59

Table 1: Web pages

In order to automate the loading of pages, the first three pages are setup to load the following page after a preset length of time, while the last page loads the first. This manner of loading pages is an endless loop, with the time delay to the next page shown in Table 1. The times listed reflect a delay that is more than adequate to allow the given page to fully load, before the next page is requested.

The loading of subsequent pages is accomplished by using the HTML META tag "Refresh" at the beginning of the document. This command is used to make web browsers reload a page after a given length of time, in seconds. This META tag can also be used, as we have, to force a browser to load an alternate page after the refresh timer expires.

[1] Information on tcpdump may be found at "http://www-nrg.ee.lbl.gov/".

[2] In this paper, the unit Kbytes will be used to denote 1024 bytes.

Variables

The experiment described in this paper has 4 degrees of freedom. In the next section we describe the TCP implementations that are used in our experiments. The remaining three variables and their values are listed in Table 2:

Server KeepAlives	ON, OFF
Number of Connections	1, 4, 8, 16
Window Size (Kbytes)	8, 16, 64, 96

Table 2: Experiment Variables

KeepAlive:

KeepAlive is an extension to HTTP that allows multiple requests to be sent over a single TCP connection [Mog95]. This use of persistent connections was defined in HTTP/1.1 [FJGFBL97] and has been shown, in some cases, to reduce the load time of HTML documents by 42% [Hei97].

Number of Connections:

Number of connections is a parameter set within a web browser that determines the maximum number of simultaneous TCP connections that can be opened. This feature allows the browser to load multiple page elements in parallel.

Window Size:

The TCP window size determines the maximum effective data rate that can be obtained on a single TCP connection [Pos81] in a network with a given RTT. We control the window size by setting the default TCP window in the NetBSD kernel.

Data for each of the 32 variable combinations is collected for a sixteen minute time period, which requires a total of eight hours and thirty-two minutes to complete. These tests are repeated three times to catch anomalies in the data.

TCP Implementation in the NetBSD Kernel

TCP Reno

The TCP [Pos81] [Com95] implementation used for the experiments outlined in this paper is a slightly modified version of the implementation packaged with NetBSD (known as TCP

Reno). This implementation includes the standard congestion control algorithms slow start, congestion avoidance, fast retransmit, and fast recovery [JK88] [Jac90] [Ste97]. In addition, our implementation includes two bug fixes outlined in [All97a]. These bug fixes include correcting the initial window used by TCP and correcting a bug in the generation of acknowledgments (ACKs). The authors believe these changes make the TCP used in the following experiments compliant with all relevant standards. The following is a brief review of the congestion control mechanisms used by TCP Reno.

The slow start algorithm [JK88] is used at the beginning of the connection and after retransmission timeouts to gradually increase the amount of data being injected into the network. The congestion window (cwnd) is the upper bound on the amount of unacknowledged data the sender can inject into the network. Furthermore, the value of cwnd can never exceed the receivers advertised window. Slow start begins by initializing cwnd to one segment and transmitting a single segment. For each ACK received, the value of cwnd incremented by one segment and the appropriate number of segments are transmitted. Slow start is terminated when the value of cwnd reaches the advertised window or congestion is detected. While not standard, many implementations of TCP also perform slow start after the connection has been idle for a certain amount of time (usually 1 RTT). After being idle TCP assumes that the congestion window is out-of-date and may be inappropriately large for the current network conditions. Therefore, the conservative mechanism is to perform slow start again in order to obtain a new estimate of the appropriate congestion window.

While slow start is the way TCP initiates data flow when a new connection is established, congestion avoidance [JK88] is the way TCP slowly probes the network for additional capacity after congestion is detected. The congestion avoidance algorithm increases the congestion window by 1/cwnd for each ACK received. Therefore, the congestion avoidance algorithm can increase cwnd by at most one packet per round-trip time (RTT).

TCP's default mechanism for detecting lost packets is the retransmission timeout (RTO). When the RTO expires before the ACK for a given packet is received, that packet is retransmitted. TCP uses lost packets as a signal of network congestion. Therefore, after the RTO expires and the segment is retransmitted, the cwnd is set to 1 segment and the slow start algorithm is used to increase the cwnd to half the window size when congestion was detected.

TCP ACKs contain the sequence number of the last in-order segment that has arrived. Therefore, when a segment arrives out-of-order, a duplicate ACK covering the last in-order segment will be generated. The fast retransmit algorithm [JK88] allows TCP to assume that a

segment has been lost after receiving 3 duplicate ACKs, rather than waiting for the RTO to expire. After receiving 3 duplicate ACKs, the TCP sender retransmits the missing segment and again uses the dropped packet as a signal of congestion in the network. The fast recovery algorithm [Jac90] follows the fast retransmission of a packet and allows TCP to avoid entering slow start after a single loss. Fast recovery first reduces the cwnd by half. Fast recovery then artificially increments cwnd by one segment for each duplicate ACK received, as TCP can infer that one packet has left the network for each duplicate ACK received. When the congestion window permits, the sender will transmit new packets. When an ACK for new data (non-duplicate ACK) is received, the sender reduces cwnd by the amount it was artificially increased (back to half the window size when congestion was detected) and then uses congestion avoidance to send new segments.

4K TCP

The 4K initial window option is suggested by Floyd, Allman and Partridge in [FAP97]. This change increases the initial window from 1 segment to roughly 4K bytes. The suggested initial window size is given in equation 1:

$$\min (4*MSS, \max(2*MSS,4380 \text{ bytes})) \quad (1)$$

Where MSS is the current maximum segment size. As indicated by equation 1, the suggested initial window will contain at least two packets. Furthermore, the window can contain up to four packets, if their combined size is less than 4380 bytes. This increase in the size of the initial window is applied at the beginning of a transfer or by transfers beginning to send data after a long idle period. It would not apply to the window size used after a retransmit timeout. This larger initial window can reduce the transfer time up to 3 RTTs, which should reduce the time required to transmit short transactions, such as those used by the HTTP protocol [FAP97].

Experiment Results

To analyze the user perceived WWW page retrieval time, we captured network traffic on the receiver (browser) side of our testbed. The packet traces collected in this experiment were processed using the tcptrace[3] program to obtain time-stamps for HTTP GET operations and the

[3] Information on tcptrace may be found at “http://jarok.cs.ohiou.edu/”.

completion of the associated element transfer. These start and finish times for the various GET operations were then correlated into web page response times. We measure the page response time from the time of the earliest HTTP GET request to the receipt and acknowledgment of the last piece of data at the browser.

Multiple Connections

Figure 2 shows a typical set of transfer results. In this case the window size was 16K, and the server allowed persistent connections. The figure shows the response time, as defined above. Results for all four test pages are shown, with the number of simultaneous TCP connections set at 1, 4, 8, and 16.

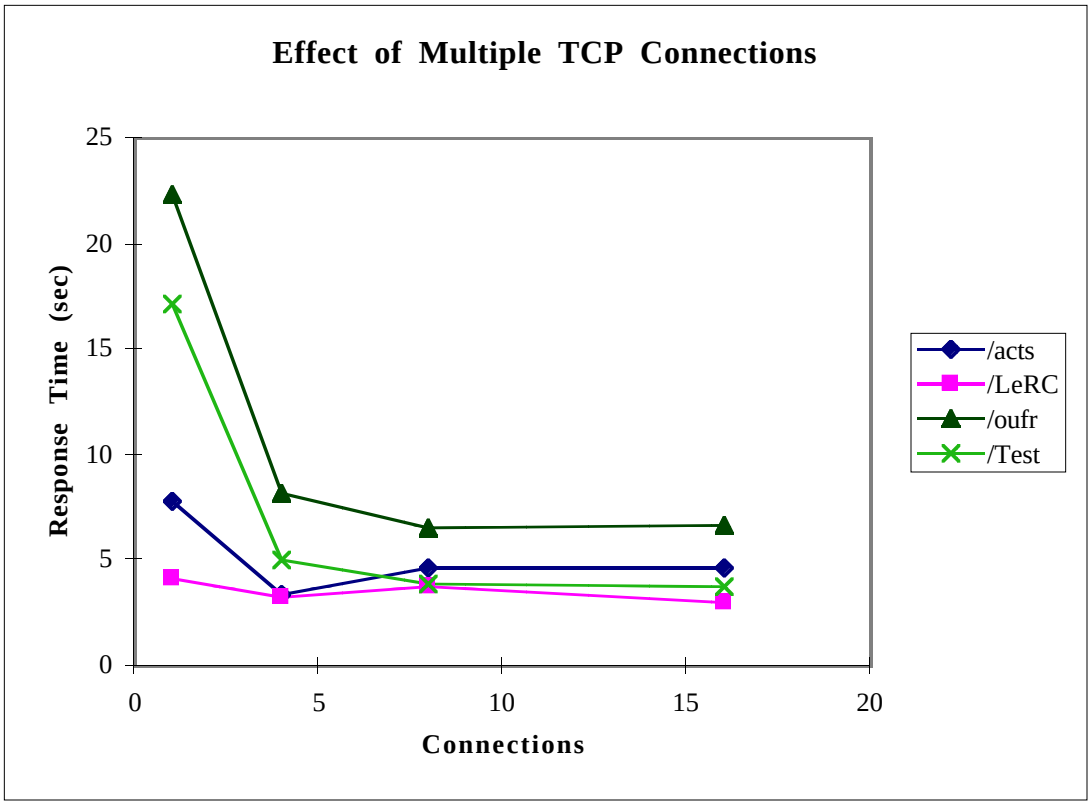

Figure 2: Effect of Multiple TCP Connections

For most of the test pages, little benefit is derived from using more than 4 connections. This is true even for the /Test page, which contains 28 separate elements. In the case of the /acts page, the response time actually becomes worse when more than 4 connections are used. We will return to this anomaly after introducing a simple "baseline" model in the next section.

Baseline Comparison

To better interpret the results above, we have constructed a simple model for the data transfer. We want to capture the minimum response time for each page as dictated by the basic mechanics of transmitting the data that makes up the page. Included in this baseline is the data rate of the channel, the TCP window size, and the number of TCP connections used. Not included are connection setup, TCP slowstart, details of the acknowledgment mechanisms, or the order in which HTTP requests page elements.

Since HTTP 1.0 does not permit "pipelining" a connection can be used for a new GET request only after the previous request on that connection is complete. As a consequence, each page element requires at least one Round Trip Time (RTT) to transfer. Since we do not attempt to model the slow-start algorithm or the use of a congestion window, we assume that up to a full TCP window can be transferred during a time span equal to the RTT plus the transmission time required to send the element at the T1 bit rate. If the size of the element exceeds the TCP window, one or more additional RTTs and transmission times may be required. Shown below are the comparisons of this simple model with the experiment, for the /oufr (Figure 3) and the /acts (Figure 4) pages. The same experiment settings as in figure 2 are used.

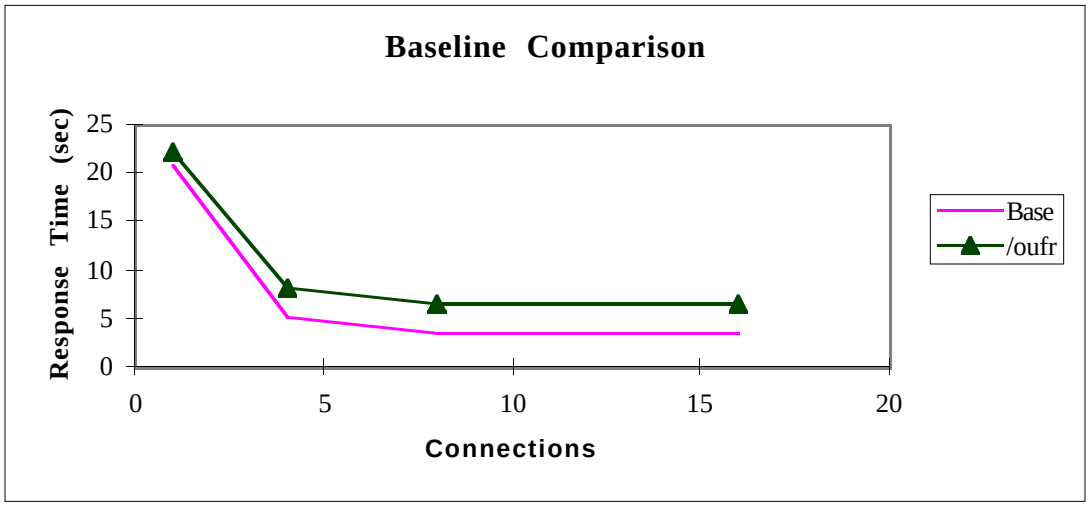

Figure 3: Comparison to the Baseline Model for the /oufr Page

The /oufr page consists mostly of large graphics elements. In this case, the single connection transfer proceeds almost precisely as predicted by the simple model. Transfers over multiple connections do not gain as much efficiency as the simple model indicates, due to the slow-start algorithm and the other effects not included in the model.

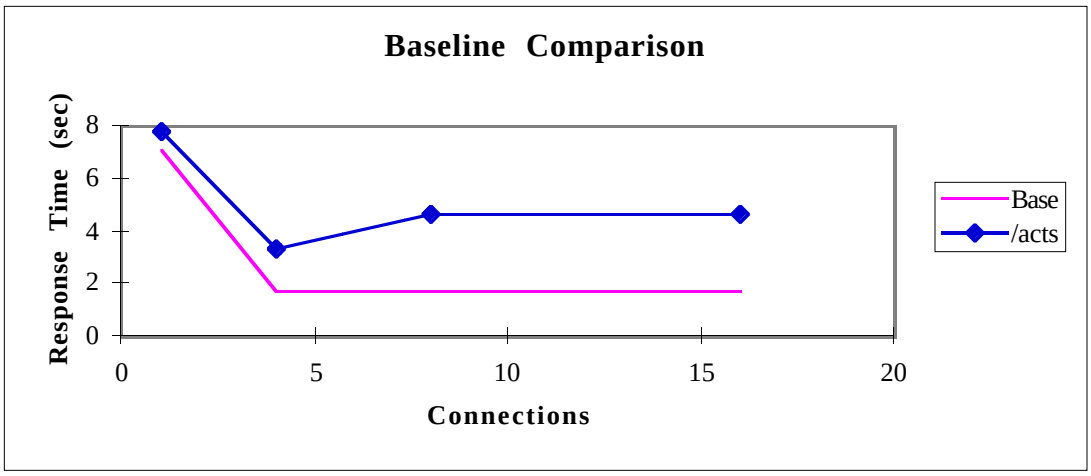

Figure 4: Comparison to the Baseline Model for the /acts Page

We have analyzed the details of retrievals for the /acts sample page to understand why the use of more than 4 connections actually hurts response time. When persistent connections are permitted by the web server, the browser always assumes that a previously opened connection can be used again. The server, on the other hand, will close connections based on resource constraints on the server machine and the time over which a connection was idle. In our server configuration, the server almost never closes a connection down when 4 or less persistent connection are in use. At 8 or 16 connections, however, there is a finite probability that a connection will be terminated by the server. When this happens, the browser will not realize this until it tries to use the closed connection. At that time the request will fail and the browser will request a new connection. The retrieval of the page element in question will require an additional 3 round-trip times due to the three-way handshake. In the case of the /acts page, there are only two elements which require significant time to transfer. The added retrieval time required for one of these elements then drives the response time for the entire page up.

It is surprising that the TCP slow-start algorithm (which is not included in the baseline model) does not appear to impact the response time. We have discovered that the retrieval of web pages over persistent connections has a particular sequence of TCP interactions which prevent the server from re-entering slow-start between element requests, and even between page requests. We discuss the details of this effect later in this section.

Persistent Connections

As indicated in the description of the experiment, we have examined the use of “persistent connections” [FJGFBL97]. Due to the large RTT on the satellite link, we expect that the use of

persistent connections will improve response time by saving the extra delay inherent in opening TCP connections with the three-way handshake. Figure 5 shows the difference in response times between persistent and non-persistent connections. The window size is 16Kbytes; positive numbers in the figure indicate that the persistent connection case was faster (lower response time) than the non-persistent case. Data for all pages is shown. The lines in the figure are labeled with the number of connections used. The differences in persistent/non-persistent response times is the same for 8 and 16 connections.

As one would expect, the response time savings from persistent connections is most pronounced for small numbers of connections, where connections have to be opened and closed sequentially multiple times. A comparison of the /LeRC and /Test pages shows this effect clearly. These two pages contain similar amounts of total data, but /Test contains a much larger number of elements. Especially for small numbers of connections, the savings for the /Test page are much more pronounced.

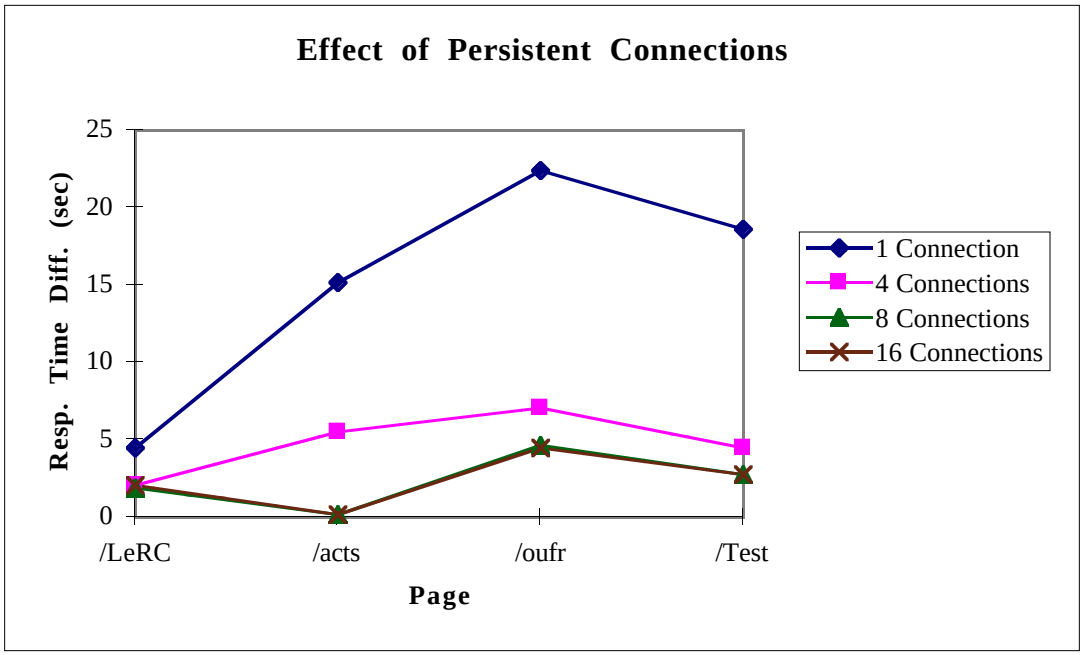

Figure 5: Response Time Improvements from Persistent Connections

On the other hand, the effect seen in figure 5 involves more than the time saved in opening connections. Note that the /acts and /oufr pages have similar numbers of elements, while /oufr contains a much larger amount of total data. The savings for /oufr are consistently larger than for the /acts page. We believe that the server avoids re-entering slow-start between pages when

persistent connections are used, which would account for the difference seen. We comment on this behavior in the next section.

Larger Initial Slow-Start Window

It has been suggested that web page transfers will benefit from the use of a larger initial window in the slow-start algorithm. We have incorporated the suggested 4K initial slow start window into the TCP Reno kernel used in these experiments. Figure 6 shows the difference between the response times with the normal slow start versus the 4K modification. Positive numbers indicate that the 4K modification reduced the response time. The experiment shown does not use persistent connections. As expected, the 4K modification is beneficial for pages which contain large graphics elements.

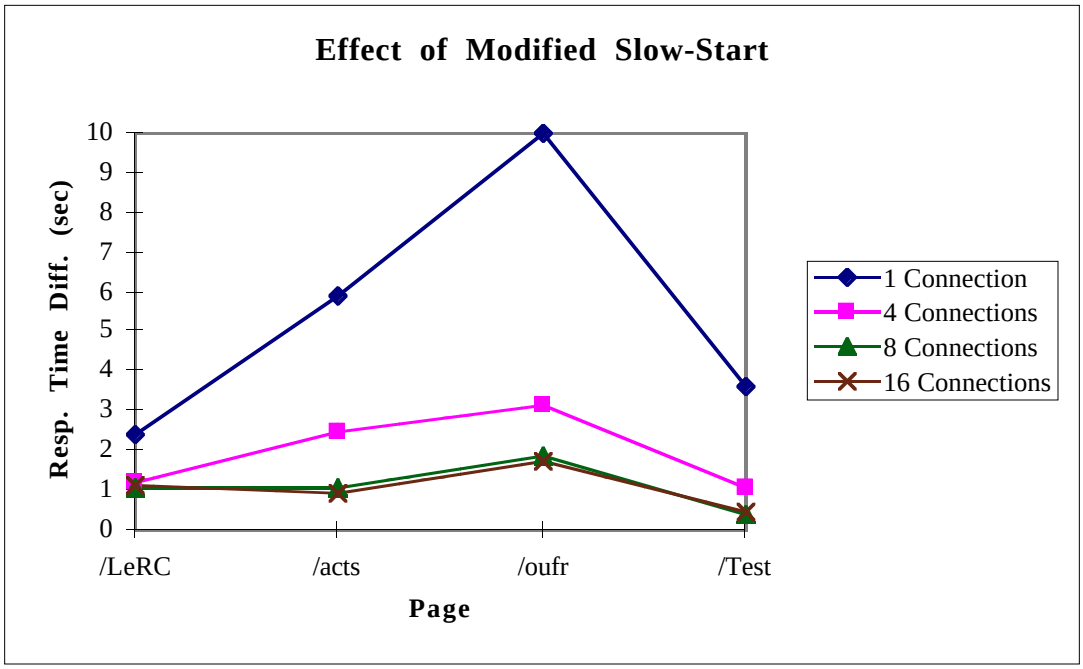

Figure 6: Response Time Improvements from Modified Slow-Start

Surprisingly, the experimental data show no difference between the modified and unmodified slow start if persistent connections are allowed. We assumed that the connections would enter an idle state between pages retrievals, or even between element retrievals. A detailed look at the page transfers shows, however, that TCP never re-enters slow start. After the transfer of a page, the conditions for an "idle" connection at the server are met. However, the server never sends further data unless a request from the client is received. The TCP segment that carries the next HTTP GET request resets the idle indication in the server TCP. Once the HTTP

server starts sending data, the idle condition is no longer set in TCP. The slow start modifications therefore have no measurable effect in the case of persistent connections, since the larger initial window is only used when the connection is first established, not after the TCP connection appears to have idled between page retrievals.

The Effect of Larger Windows

In bulk transfer of data, it has been demonstrated that large TCP window sizes are needed to produce effective use of a satellite channel [AHKO97], [All97b], [Hay97]. Figure 7 shows the results of various window sizes on the response times for the test pages in this experiment. The results shown are for a single persistent TCP connection, where the window size should matter most.

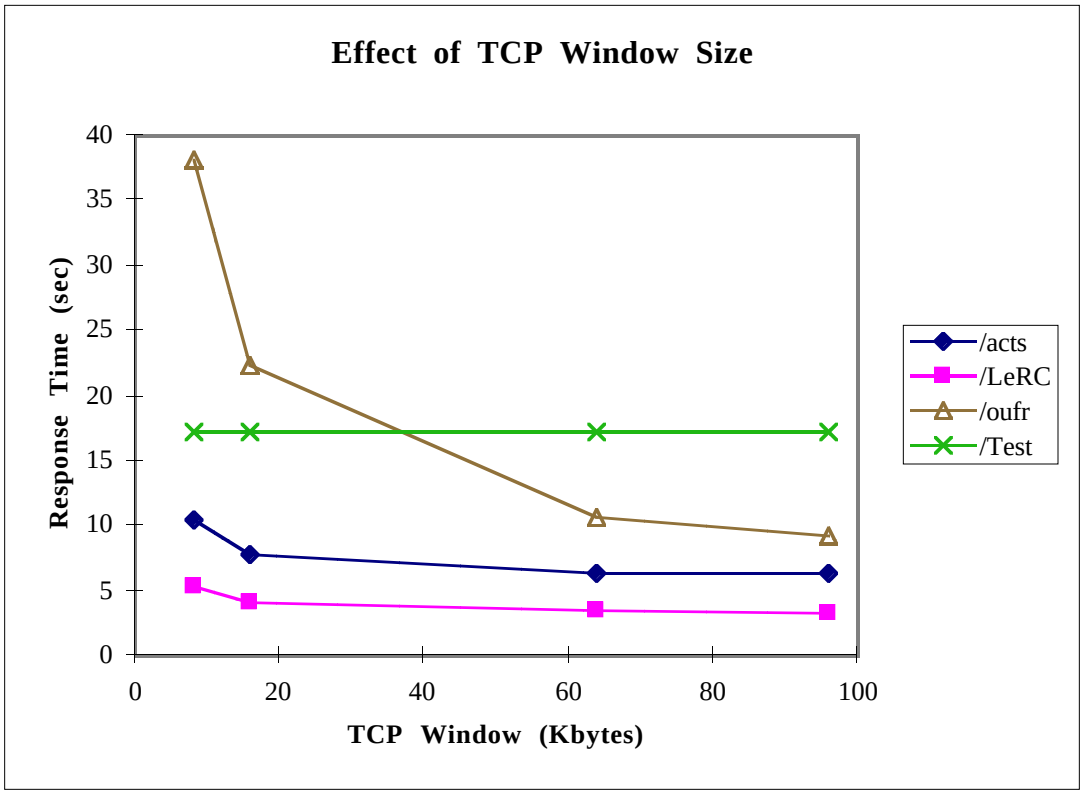

Figure 7: Effect of Large TCP Windows

The /Test page contains no elements above 3Kbytes; it is therefore not surprising that the TCP window size does not effect the response time for this page. The results indicate that for the other test pages, an increase of the TCP window from 8Kbytes to 16Kbytes is clearly beneficial, while TCP window sizes above 16Kbytes are useful only for pages that contain a

large amount of data. The /oufr page contains a total of 500kBytes of data, and its retrieval behaves more like a bulk data transfer.

Conclusions and Further Research

Our measurements clearly show that the use of persistent connections greatly improves information retrieval response time. This result also suggests that HTTP 1.1 may exhibit even further performance improvement in the satellite environment; we are in the process of extending our measurements to include HTTP 1.1. We have further shown that the increased initial slow start window is beneficial in the case of non-persistent connections. If connections are allowed to persist for a fairly long time, TCP does not re-enter slow start, and the slow start modification has no effect. For servers which allow persistent connections, but close idle connections after brief periods of inactivity, we expect some improvement from the use of the modified slow-start.

The fact that TCP does not re-enter slow start between page retrievals is disturbing due to the potential network impact. TCP adjusts the size of the congestion window based on network changes. After a long period of idle time, TCP assumes that its knowledge of the network conditions is no longer valid and therefore it uses slow start to find the proper window size. The experiments outlined in this paper show that the TCP implementation we used in our tests violates the spirit of this re-slow start-after-idle mechanism. Any incoming or outgoing packet resets the idle timer. So, an HTTP request resets the timer on the server, allowing the server to send data using the same window size employed before the idle period. Fixing this unintended problem in TCP will likely have a negative impact on HTTP transactions, as they are currently utilizing a large window after an idle period. However, not fixing this problem could lead to a large loss event caused by TCP sending at an inappropriate rate after the idle period (in which the network conditions have changed). A large loss event could lead to a lengthy recovery period leading to a longer total page transfer time.

The study presented here focuses on the specifics of how HTTP functions and interacts with the TCP stack. For that reason the tests presented here were conducted in a carefully isolated environment. Further experiments will need to include competing data flows on the satellite link to better simulate real web traffic. Since packet loss will be likely in that type of environment, it will then also become important to look into various TCP options which have been developed to better deal with lossy and/or congested links, such as selective acknowledgments. We have begun these types of studies in our test network. Together with the data presented in this paper, we hope to provide satellite operators, server administrators,

and browser manufacturers the information needed to build efficient World Wide Web networks that include satellite links.

Acknowledgments

The authors want to thank Dr. Shawn Ostermann, Ohio University, for his modification of tcptrace for the analysis of HTTP data transfers; these changes greatly simplified the data analysis for this experiment. We also thank Jason Pugsley, Ohio Northern University, for his assistance in configuring the test network and for various utilities which made this experiment easier for us. One of the authors (HK) gratefully acknowledges that his participation in this study is supported under NASA grant No. NCC3-430.

References:

[AHKO97] Mark Allman, Chris Hayes, Hans Kruse, Shawn Ostermann. TCP Performance Over Satellite Links. In Proceedings of the 5th International Conference on Telecommunication Systems, March 1997.

[All97a] M. Allman. “Fixing Two BSD TCP Bugs”. NASA Lewis Research Center Technical Report CR-204151, October 1997.

[All97b] Mark Allman. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.

[Com95] Douglas E. Comer. Internetworking with TCP/IP, Volume 1, Principles, Protocols, and Architecture. Prentice-Hall, 3rd Edition, 1995.

[FAP97] S. Floyd, M. Allman and C. Partidge. “Increasing TCP’s Initial Window”. Internet Draft. File: draft-floyd-incr-init-win-00.txt, July 1997.

[Hay97] Chris Hayes. Analyzing the Performance of New TCP Extensions Over Satellite Links. Master's thesis, Ohio University, August 1997.

[Hei97] John Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. Computer Communications Review, 27(2):65--73, April 1997.

[Kru95] Hans Kruse. Performance Of Common Data Communications Protocols Over Long Delay Links. In Proceedings of the 5th International Conference on Telecommunication Systems, March 1995.

[FJGFBL97] R. Fielding, J. C. Mogul, J. Gettys, H. Frystyk, T. Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1, January 1997. RFC 2068.

[JK88] V. Jacobson and M. Karels. “Congestion Avoidance and Control”. In ACM SIGCOMM Symposium on Communications Architecture and Protocol, August 1988.

[Jac90] V. Jacobson. “Modified TCP Congestion Avoidance Algorithm”. Technical report, April 1990.

[Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[Ste97] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

Appendix A

Server and Client Configuration File Settings

Apache Configuration

The following settings in the HTTP server can effect performance, but were not varied in these tests:

Timeout 300: This is the length of time the server will wait for each of the following conditions:

1. To receive a GET request
2. Between receiving or transmitting two data segments.

MaxKeepAliveRequests 100: If KeepAlives are on, this sets the maximum number of requests per connection.

KeepAliveTimeout 60: Once a request is received, and if KeepAlives are on, this is the maximum time the server will wait for another request before the connection is closed.

MinSpareServers 5, MaxSpareServers 10: These settings regulate the minimum and maximum number of idle server processes. If fewer than the set minimum are currently running, additional child processes are created at a maximum rate of one per second.

StartServers 5: Sets the number of processes to create on startup.

MaxClients 150: This setting regulates the maximum number of simultaneous requests that can be supported.

MaxRequestsPerChild 30: Limits the number of requests that a single child process is allowed to handle. The child process will end, after the limit is reached.

SendBufferSize 98304: This setting is required in order for apache to actually make use of a large TCP window advertised by the client machine.

Netscape Configuration

All caching has been turned off, by the following lines in the configuration file:

MEMORY_CACHE_SIZE: 0
DISK_CACHE_SIZE: 0
VERIFY_DOCUMENTS: 0
CACHE_SSL_PAGES: False

SOCKET_BUFFER_SIZE: 32: This sets the amount of memory allotted for network data transmissions, in kilobytes.