Experimentation and Modeling of HTTP Over Satellite Channels *

Hans Kruse
J. Warren McClure School, Ohio University
hkruse1@ohiou.edu

Mark Allman
NASA Glenn Research Center/BBN Technologies
mallman@grc.nasa.gov

Jim Griner, Diepchi Tran
NASA Glenn Research Center
{jgriner,dtran}@grc.nasa.gov

Abstract

This paper investigates the performance of various versions of the HyperText Transfer Protocol (HTTP) over a geosynchronous satellite link. Both HTTP/1.0, the currently popular form of the protocol, and HTTP/1.1, the recently standardized form of HTTP, are studied. Next, we quantify the impact of a moderate bit-error rate on the performance of HTTP. Finally, we expand the mathematical model of HTTP presented in [HOT97] to encompasses a wider range of HTTP behavior. We show this model accurately predicts HTTP throughput by comparing it with HTTP transfers made over a satellite channel.

1 Introduction

This paper presents a detailed performance evaluation of World-Wide Web (WWW) page retrievals over a network path with a satellite component. The delay imposed by the geosynchronous (GEO) satellite channel used in our investigation highlights the importance of using several mechanisms included in the standard HyperText Transfer Protocol (HTTP) [BLFN96, FGM+97]. NASA's Advanced Communication Technology Satellite system was used to conduct the tests presented in this paper.

The first goal of this paper is to document experimental results for WWW page retrieval over a satellite network using a number of popular versions of the HTTP protocol. We tested both HTTP/1.0 and HTTP/1.1 in various configurations. The results of these tests illustrate the importance of using the latest version of HTTP (version 1.1) with the pipelining option in satellite networks. Additionally, we quantify the impact of using multiple concurrent HTTP connections on the transfer time. Finally, we investigate the performance impact of using a larger initial TCP congestion window [AFP98] in HTTP transfers over long-delay channels.

Even though the satellite channel used in our experiments employed strong forward error correction (FEC), as recommended by [AGS99], some of the transfers experienced non-negligible bit-error rates (BER), leading to segment loss. TCP interprets all loss as an indication of network congestion and reduces the sending rate accordingly [Jac88, Ste97, APS99]. Therefore, a segment lost due to corruption leads to an unnecessary reduction in TCP's sending rate and often a reduction in performance (especially for short transfers, such as WWW pages). This paper contributes a preliminary evaluation of the impact of non-negligible BERs on HTTP performance.

Finally, we extend the model of HTTP presented in [HOT97] to analyze reasons behind the observed performance. In addition, the algorithms associated with TCP, which is used to reliably deliver

*This is a preprint of an article accepted for publication in the International Journal of Satellite Communications.

HTTP data, have a direct impact on the transfer of WWW pages and therefore modeling the behavior of TCP is equally important.

The remainder of this paper is organized as follows. Section 2 discusses the TCP algorithms in the version of TCP used in our experiments. Section 3 discusses the differences between HTTP/1.0 and HTTP/1.1. Section 4 outlines the setup of our experiments. Section 5 defines the model developed to characterize WWW transfers. Section 6 outlines the results of our experiments. Finally, section 7 presents our conclusions and outlines future work in this area.

2 TCP Versions

HTTP traffic is reliably delivered by the Transmission Control Protocol (TCP) [Pos81]. The algorithms employed by the TCP implementation can have a large impact on the performance of HTTP. In this paper, we concentrate on the standard version of TCP (known as *TCP Reno*) [Ste97, APS99]. Specifically, the implementation of TCP contained in NetBSD 1.2.1 is the basis for this study. NetBSD's version of TCP includes the congestion control algorithms slow start, congestion avoidance, fast retransmit and fast recovery algorithms, originally outlined by Jacobson [Jac88, Jac90]. The reader is assumed to be familiar with these algorithms. Additionally, our TCP implementation contained two bug fixes that were found during the course of this study and briefly outlined below [All97a]. Finally, we investigated the impact of TCP using a larger initial congestion window [AFP98], which is also briefly summarized below. Since our experiments were conducted over a non-congested network path, the selective acknowledgment (SACK) TCP option [MMFR96] would have provided little impact on the main results presented in this paper.

2.1 TCP Reno Bug Fixes

In the course of running the tests presented in this paper, we found two bugs in the NetBSD TCP implementation[1] [All97a, PAD+99]. The first bug is the use of a two segment initial congestion window (*cwnd*) when a sending TCP is opened passively. The bug occurs because the acknowledgment (ACK) for the SYN segment used to setup the connection incorrectly increments *cwnd* by one segment[2]. Therefore, data transmission begins by sending 2 segments rather than the standard 1 segment[3]. The version of TCP used in the tests presented in this paper always uses a 1 segment initial congestion window, unless otherwise noted.

The second bug involves TCP's *delayed acknowledgment* mechanism. RFC 1122 [Bra89] suggests that a TCP receiver refrain from generating an ACK for each incoming segment. However, an ACK must be transmitted for every second full-sized segment. If a second full-sized segment is not received within a given timeout, the receiver must generate an ACK (this timeout must be no more than 500 ms according to [Bra89], however is implemented as a 200 ms heartbeat timer in NetBSD). NetBSD uses the delayed ACK mechanism. The bug in NetBSD is that the receiver does not correctly determine whether a segment is "full-sized" and therefore often ACKs every third segment, rather than every second segment. When TCP uses options, such as timestamps [JBB92] (which were used in our tests), the amount of data transfered in a given packet is reduced by the amount of space used by the options. However, this is not taken into account by the algorithm that determines when an ACK must be generated. Therefore, to obtain 2 "full-sized" segments worth of data, three segments must be received. Generating fewer ACKs hinders the growth of the congestion window, and therefore the performance TCP and HTTP are able to attain [Pax97, PAD+99, All98]. Our fixes corrected this problem so that ACKs are correctly generated for every second full-sized data segment that arrives.

2.2 Larger Initial Window

In some of the experiments outlined in this paper, we tested an experimental TCP modification that increases the size of the initial congestion window from 1 segment to roughly 4 KB[4] [AFP98]. Specifically, the initial window (W_i) is set according to equation 1.

$$W_i = min(4 \cdot MSS, max(2 \cdot MSS, 4380)) \qquad (1)$$

According to this equation, the number of segments in the initial window can vary from 2–4 depending

[1] These bugs in NetBSD were traced back to the BSD 4.4 Lite distribution. Therefore, we suspect that NetBSD is not the only operating system that contains these bugs.

[2] In the NetBSD TCP implementation, *cwnd* is maintained in bytes. However, to simplify the discussion we discuss the algorithms in terms of segments in this paper.

[3] Since the time these tests were conducted, the IETF has approved the use of a 1 or 2 segment initial congestion window. Therefore, this can no longer be described as a bug.

[4] In this paper, 1 KB = 1024 bytes.

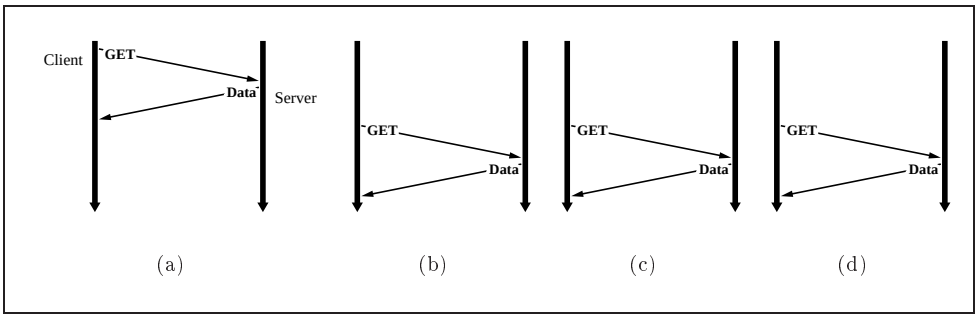

Figure 1: HTTP/1.0 – Non-persistent Connections

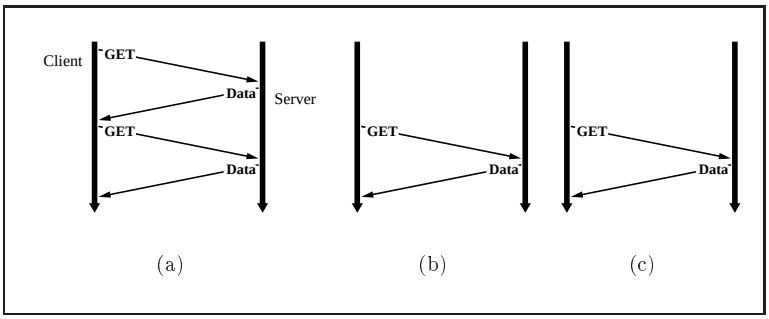

Figure 2: HTTP/1.0 – Persistent Connections

on their size. In our experimental environment using the above equation yields an initial *cwnd* of 3 segments, each consisting of 1460 bytes (unless a particular transfer is less than 4380 bytes long).

3 HTTP

Several versions of HTTP are currently used in servers and browsers on the Internet. HTTP/1.0 calls for a separate TCP connection to be established to retrieve each HTTP object (HTML page, image, Java script, etc.). This method of retrieving pages causes many short TCP connections to be established between the client and the server to retrieve a given WWW page. This has been shown to be an inefficient means of page retrieval [Mog95, THO96, Spe97]. In addition, WWW browsers often open multiple simultaneous TCP connections to download WWW objects in parallel. Figure 1 illustrates the interactions between a common HTTP/1.0 WWW client and an HTTP server when loading an initial HTML page that contains 3 inline images. In this case, the base HTML page is transfered over connection a, followed by 3 concurrent connections (denoted b, c and d) that are used to transfer the 3 inline images.

To overcome the inefficiencies of establishing many short TCP connections, an approach for using persistent connections in the form of HTTP/1.0 "Keep-Alive" requests has been implemented in some WWW clients and servers. The Keep-Alive extension to HTTP/1.0 allows multiple requests to be sent over a single persistent TCP connection [Mog95]. However, the use of persistent connections does not preclude the use of multiple parallel TCP connections in many browsers. Figure 2 illustrates the interactions between a client and server, using HTTP/1.0 with the Keep-Alive option. This figure differs from figure 1 in that the first connection (denoted a) is used to transfer the original HTML page plus one of the 3 inline images.

The use of persistent connections was formally defined in HTTP/1.1 [FGM+97] and has been shown, in some cases, to reduce the load time of web pages by 42% [Hei97]. HTTP/1.1 does not use multiple connections, but rather loads each element in turn on the same TCP connection. Figure 3 illustrates HTTP/1.1 loading an HTML page and 3 inline images. This figure illustrates that after each response is received from the server, the client issues the next request. This leads to a lull of 1 round-trip time (RTT) between subsequent object transfers. Note that HTTP/1.1 (with persis-

tent connections) acts very similar to HTTP/1.0 with keepalives. For the experiments presented in this paper we consider the two cases to be identical.

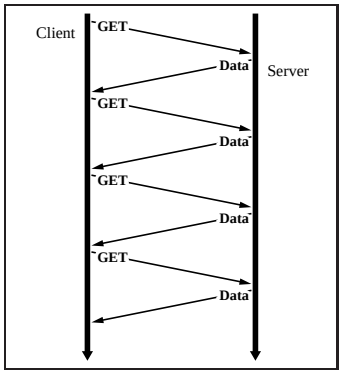

Figure 3: HTTP/1.1 without Pipelining

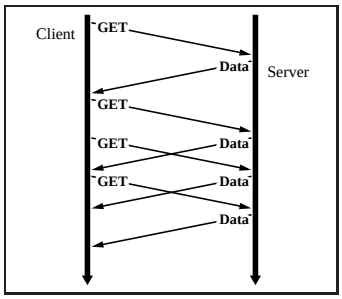

Figure 4: HTTP/1.1 with Pipelining

HTTP/1.1 with *pipelining* is also defined in [FGM+97]. Pipelining allows an HTTP client to send multiple requests while receiving responses from the server. This mechanism eliminates the lull between each object retrieval caused by the transmission of a new request. Figure 4 illustrates HTTP/1.1 with pipelining loading an HTML page followed by 3 images. In this case, the HTML page is loaded and then 3 requests are sent to the server immediately. The responses are sent back to the client in the order they were requested.

4 Experimental Configuration

4.1 Network Topology

The satellite channel used in our experiments was NASA's Advanced Communications Technology Satellite (ACTS) [Bv91]. ACTS, a geostationary satellite, operates in the Ka-band, has a 30 GHz uplink and a 20 GHz downlink, with data rates ranging from 64 Kbps to 622 Mbps. For our experiments, T1 (1.536 Mbps) VSAT terminals are utilized at both NASA's Lewis Research Center[5] (LeRC) and Ohio University (OU). Since ACTS operates in the Ka-band, which is known to be strongly effected by rain-fade events, a strong forward error correction algorithm, deployed upon detection of a rain-fade, is used. Therefore, the circuit used in these experiments had a very low (but not necessarily zero) bit-error rate (BER). Several of our tests did experience corruption-based losses. The impact of these losses is quantified in section 6.

The network layout used is shown in figure 5. The hosts on the LeRC side of the testbed are Pentium Pro 180 Mhz computers, running NetBSD 1.2.1. These two machines are connected to router 1. Using a built-in T1 (1.536 Mbps) CSU/DSU, router 1 is connected to a VSAT earth station. A full duplex T1 link is setup between this earth station, through the ACTS satellite, to a similar earth station at Ohio University (OU). The second earth station is connected through a T1 CSU/DSU to router 2, via RS 449. Router 2 is connected to a 486/33 Mhz computer at Ohio University, which is also running NetBSD 1.2.1.

The queue size in the routers have been adjusted to provide adequate buffering (roughly twice the size of TCP's advertised window in these experiments). The queue size used has been shown to prevent segment loss in previous bulk-transfer experiments over ACTS [All97b].

4.2 Experimental Setup

The experimental setup consists of running an HTTP browser on the LeRC side of the network, and an HTTP server on the OU side. The client application was Netscape's *Navigator*, version 3.01, for the HTTP/1.0 tests and W3C's *WebBot*[6] for the HTTP/1.1 tests. The WWW server used in all tests was Apache version 1.2b11. To analyze the behavior of HTTP, all segments were captured on the LeRC (client) side of the network using *tcpdump*[7]. The transfers were traced on the client side of the network so that the transfer times were accurate from the point of view of a WWW user. Four test web pages were requested by the browser, each

[5] NASA's Lewis Research Center has been renamed the Glenn Research Center since these experiments were performed. However, the experiments presented were performed before the re-naming and thus we have retained the old name for this paper.

[6] *WebBot* is available from `http://www.w3.org/Robot/`.

[7] *tcpdump* is available from `http://www-nrg.ee.lbl.gov`.

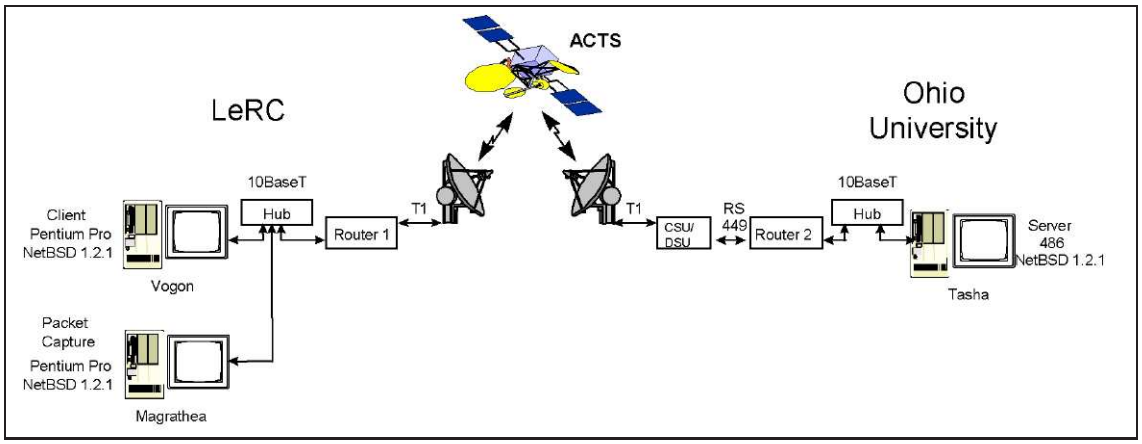

Figure 5: Network Topology

Page	Number of Elements	Page Size (KB)
LeRC	4	48
acts	9	99
oufr	10	491
Test	28	28

Table 1: Web Page Details

having a different number of elements and overall page size as shown in table 1.

This paper presents several different HTTP scenarios, as follows.

- HTTP/1.0 with Keep-Alives. Using HTTP/1.0 we present results using a single active connection, as well as tests in which 4 parallel connections were employed. Note that when using only a single connection, this scenario is identical to the HTTP/1.1 without pipelining scenario, as outlined above.

- HTTP/1.1 with pipelining. Under this scenario, only a single active connection was employed.

- The above two items were tested with the standard initial window (1 segment) and with the larger initial congestion window specified in equation 1 (3 segments in our environment).

The advertised window size used in all the experiments presented was 96 KB (roughly the delay-bandwidth product of the network path). However, the TCP connections were never limited by the advertised window. Each HTTP scenario outlined above was repeated for approximately 48 minutes (representing multiple retrievals of each page given in table 1). The packet traces collected during each run were post-processed using *tcptrace*'s[8] HTTP module, as well as several scripts.

5 HTTP Model

5.1 Modeling the TCP Congestion Window

We restrict our examination to the case where the RTT is large compared to the transmission time of a TCP segment. In such a situation, it is well known that the TCP slow start algorithm produces isolated *flights* of segments, followed by an idle period until the first acknowledgment for the flight returns. Also note that we are restricting our analysis to the case when the network path is congestion free and free of bit errors. While this is somewhat artificial we believe that the model provides a good basis for understanding the various protocol interactions before considering the case when network congestion is introduced or non-zero bit error probabilities are encountered.

We define the congestion window, *cwnd(i)* as the maximum number of segments in the *i-th* flight. The starting value, *cwnd(0)*, depends on the operating system implementation. The design of TCP assumes that *cwnd(0)=1*, which NetBSD uses. As discussed above, we also use *cwnd(0)=3* in some of the experiments presented.

As discussed by Heidemann [HOT97], *cwnd(i)* depends on *cwnd(i-1)* and the number of acknowl-

[8] *tcptrace* is available from `http://jarok.cs.ohiou.edu/`

edgments received for the flight *i-1*. A general formula for *cwnd(i)* is given in equation 2.

$$cwnd(i) = cwnd(i\text{-}1) + f(cwnd(i\text{-}1)) \quad (2)$$

Where *f()* represents the number of ACKs that arrive at the TCP sender in response to flight *i-1*. The definition of f depends on whether the receiver generates delayed ACKs. Equation 3 defines f for the case when each segment is ACKed, while equation 4 specifics f when the receiver generates delayed ACKs.

$$f(x) = x \quad (3)$$

$$f(x) \approx \left\lceil \frac{x}{2} \right\rceil \quad (4)$$

Note that equations 3 and 4 are upper bounds and the actual value of the function may be reduced by ACK loss (which was negligible in our experiments). Additionally, the function for delayed ACKs (equation 4) assumes that the RTT is larger than the delayed ACK timer. This estimate further neglects the residual delay between the receipt of an odd segment, and the firing of the delayed ACK timer.

Equation 2 does not have a general closed-form solution for the function defined in equation 4. However, equation 2 can, of course, easily be solved numerically. Examining the numeric results shows that the increase in *cwnd* is almost exactly exponential after the first 2–3 RTTs. We therefore arrive at useful approximation given in equation 5.

$$\begin{aligned} cwnd(i) &\approx cwnd(0) \cdot A^i \\ A &= \begin{cases} 2 & \textit{for equation 3} \\ 1.57 & \textit{for equation 4} \end{cases} \end{aligned} \quad (5)$$

Also, it is often of interest to determine the number of RTTs needed to open the congestion window to the steady-state window size (which, in the absence of congestion, is determined by the advertised window or the bandwidth-delay product and the bottleneck buffer capacity). If the desired window size for a single connection is W_s segments, equation 6 yields the number of RTTs required to open *cwnd*.

$$i' = \frac{ln(W_s) - ln(cwnd(0))}{ln(A)} \quad (6)$$

The notation *i'* is used to indicate that the result of equation 6 is not an integer, while in equation 5, i is assumed to be integer.

5.2 Multiple Connections

WWW browsers using HTTP/1.0 tend to open multiple TCP connections to retrieve the elements of a web page (as outlined in section 3). The analysis of the previous section applies to each connection. However, in the case of multiple parallel connections, we want the sum of all congestion windows to reach the capacity of the network path. A lower bound on the time required to reach this point can be found by assuming that all connections increase *cwnd* equally. Assume we have M connections and a target aggregate congestion window of $W_A = W_s \cdot M$ segments. Inserting into equation 6 yields equation 7, which is the number of RTTs needed to reach an effective window of W_A using M parallel TCP connections.

$$i' = \frac{ln(W_A) - ln(M) - ln(cwnd(0))}{ln(A)} \quad (7)$$

Note that the use of multiple concurrent connections reduces the time required for slow start ramp-up in the same way as an increased initial window, at the level of detail in this analysis. However, operating system effects may cause the two scenarios to differ in performance in an actual implementation. In the following sections we will use the slow start acceleration value, S_a, as defined in equation 8. S_a is the effective initial window across the M concurrent connections.

$$S_a = M \cdot cwnd(0) \quad (8)$$

5.3 Page Transfer Modeling

5.3.1 Modeling HTTP 1.1

Due to the use of a single TCP connection, the HTTP/1.1 with pipelining implementation used in these experiments is easily modeled in a straightforward manner. At the time the request for a WWW page is made, the browser issues an HTTP `GET` command for the base HTML document. One RTT later, the first base document data will be received. The browser then issues further `GET` commands for each page element referenced in the base document. These `GET` commands can be generated as soon as the reference is received by the browser, without waiting for the current data transfer from the server to be completed. This behavior is illustrated in figure 6, which shows the sequence of element retrieval requests for an HTTP/1.1 transfer of the *oufr* WWW page. For each element, the bottom of the vertical line shows the time the element was requested by the browser, while the top

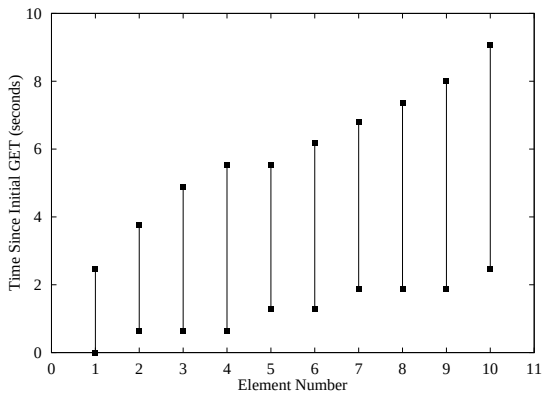

Figure 6: HTTP/1.1 with Pipelining Page Retrieval

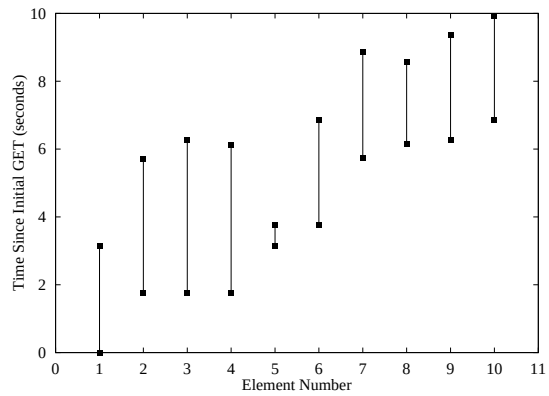

Figure 7: HTTP/1.0, 4 Connection Page Retrieval

of each line shows when the element was completely received.

Figure 6 shows that all `GET` requests were issued while the first element was being received. For our analysis, we assume that after the second `GET` request the server always has data to send. Under this assumption, the page retrieval can be modeled as a bulk data transfer of the combined page elements, taking slow start into account as described in section 5.1, plus the one additional round trip time required to request the initial base HTML element.

It should be noted that the HTTP/1.1 specification does allow the use of two concurrent TCP connections. For implementations which use the two connection approach, a more detailed model will be needed.

5.3.2 Distribution of Pages Over Multiple Connections (HTTP/1.0)

Since HTTP/1.0 retrieves page elements using multiple TCP connections, it requires a more complex model to predict page retrieval times. We must account for two major differences between HTTP/1.0 and HTTP/1.1:

- To model HTTP/1.0, we must determine which page elements will be retrieved on which connection. Since the distribution of element requests over the TCP connections is controlled by the browser implementation, there is no single correct answer to this question. We describe below a heuristic that will, in most cases, result in the most efficient allocation of page elements to connections. This will give us a lower bound on the page retrieval time.
- Once we determine which page elements will be transfered on which connection, we can model the data flow on each of these connections as described in section 5.1, with one exception. Since HTTP/1.0 does not permit pipelining, a gap of one round trip time is required between page elements to allow the browser to submit the next `GET` request.

Figure 7 shows the sequence of element retrievals for a 4 connection HTTP/1.0 request of the *oufr* page. We use this figure to illustrate an anomaly. We expected `GET` requests 2–4 to follow the base request by two RTTs (one RTT to get the document, and a second RTT to open the additional three TCP connections). Instead the figure clearly shows a delay of three RTTs. We have determined that in this case the initial `GET` request is split over two packets, which are spaced apart by one RTT due to slow start. We have not attempted to determine if the browser or the operating system cause this split. However, we have taken this additional round trip time into account in the model.

5.3.3 The Distribution Algorithm

A WWW page can be described as a collection of *n+1* elements, $E_0; E_i, i = 1, n$. The element E_0 is the "base" HTML document that must be retrieved first. The remaining n elements are referenced in the base document. These n objects are retrieved in an arbitrary order based on algorithms in the browser. The intent of the model described here is to estimate the "best" retrieval order (i.e., the ordering of elements E_1 through E_n that results in the shortest retrieval time, while using the smallest number of connections). The second part of the

algorithm ensures that we will use new connections only if they do reduce the overall transfer time.

Without loss of generality, we assume that the elements 1–n are ordered such that $size(E_i) \leq size(E_{i+1})$. The function *size(E)* will be defined later, however it basically represents the size of the given element. We intend to assign each element to a connection from the set $\{C_k, k = 1, M\}$. Finally, the amount of data carried by a connection is defined as:

$$Length(C_k) = \sum_{i=1}^{n} size(E_i) \cdot x_{ik} \tag{9}$$

where x_{ik}=1 if element i is assigned to connection k, and 0 otherwise. The algorithm is defined as follows:

1. Assign element E_1 to connection C_1. Set index $j = 2$. Set index $n = 2$.

2. Attempt to place element j, starting with connection n: If $Length(C_n) + size(E_j) \leq Length(C_{n-1})$, assign element j to connection n and increment j. If there are more elements to place, repeat step 2, otherwise exit. If element j was not placed by the above test, increment n and repeat step 2 (if there are connections left). Otherwise, continue to the next step.

3. Reorder connections such that $Length(C_k) \geq Length(C_{k+1})$.

4. Assign element j to connection M (the connection with the smallest length). Reorder the connections again based on the definition in step 3. Increment j and set $n = 2$. If there are elements left to place, go to step 2, otherwise exit.

The justification of the algorithm assumes that the *size(E)* function depends only on properties of the element, *E*. In our case that is not entirely true, as we will discuss later. Step 1 is based on the fact that elements cannot be divided. The *size ()* of the largest element therefore represents the smallest possible transfer time. Next, step 2 places elements into connections only if the placement does not increase the currently required transfer time. Finally, step 3 operates on the largest remaining element. Since step 2 could not place the element, we know that an increase in the total transfer time is needed. The increase due to this element is minimized by (a) placing the element after it has been determined that the transfer time must be increased and (b) placing the element on the connection with the smallest *Length()* value.

The algorithm defined above requires a definition for the size of an element. This size must depend only on the properties of the element itself and any global constants that apply to all connections. We use the following definition to determine *size(E_j)*:

Find the smallest integer I which satisfies equation 10:

$$\sum_{i=0}^{I} A^i \geq \frac{bytes(E_j)}{cwnd(0)} \tag{10}$$

where *bytes(E)* represents the actual length of the element in bytes and *cwnd(0)* is the initial TCP window in bytes, as defined above. The constant A is defined in equation 5. Therefore,

$$size(E_j) = (I + 1) \cdot RTT + \frac{bytes(E_j)}{B} \tag{11}$$

where RTT is the round-trip time between the client and server in seconds and B is the bandwidth of the channel in bytes per second.

The definition of size given in equation 11 takes into account the time required to transmit the element, as well as the impact of slow start. Note that this is an estimate of the size of the given element. The *size()* function will not be used to actually compute the overall transfer time. Once the placement of all elements onto connections is complete, we correctly compute the transfer time using slow start across all elements. The fact that the current congestion window encountered by an element on a particular connection depends on the previous transfers on that connection is not captured in the *size()* function. We have not evaluated this approximation systematically. However, for the web pages used in our experiments, we have computed the transfer times based on retrieving elements in ascending order of size (as opposed to the descending order assumed above). These calculations did not show a significant difference in retrieval times.

6 Results

6.1 Experimental HTTP Comparisons

Figure 8 summarizes the performance of various versions of HTTP collected as described in section 4. We first compare two different configurations

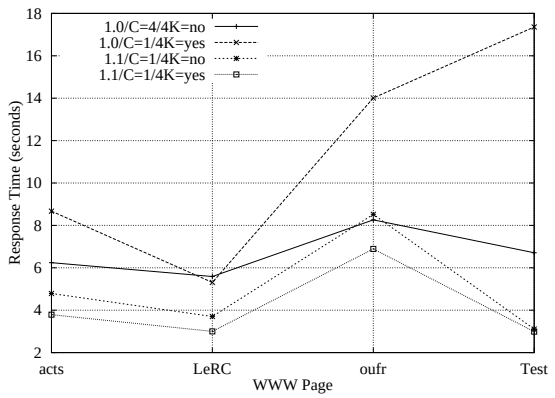

Figure 8: HTTP Comparison

of HTTP/1.0. In one case, we use a single TCP connection with a larger initial *cwnd* and the other case we use the standard initial *cwnd*, but allow the browser to use 4 parallel TCP connections. Again note that when using a single HTTP/1.0 connection with keepalives the network behavior is the same as when using HTTP/1.1 without pipelining. When using a single connection with a larger initial congestion window or 4 parallel connection we use roughly the same aggregate initial *cwnd*. Figure 8 shows that the use of multiple connections results in lower response times for all but the *LeRC* page, which consists of 4 page elements (the base HTML and 3 images). The sequential retrieval of these elements over one TCP connection requires a total of three extra RTTs because one idle RTT is inserted between the retrieval of each page element after the base HTML has been transfered. This idle period is the time required to send the request and start receiving the response. Using four connections all images can be retrieved in parallel without paying a 1 RTT penalty for each element after the base HTML. When concurrent connections are used to load the *LeRC* page the response time is dominated by one large image. Without a larger initial *cwnd*, the single-connection case should run roughly one RTT slower than the four connection case, which our experimental data confirms. In figure 8, the one connection HTTP/1.0 transfer has been accelerated by the use of a larger initial *cwnd* which saves approximately 1 RTT while increasing the TCP congestion window. As a result, the data points for these two cases basically overlap in the figure.

The remaining three WWW pages (all but *LeRC*) have a larger number of page elements. Therefore, the savings from the use of the multiple connections outweighs the benefit of using a larger initial *cwnd* on a single connection. This difference is most dramatic for the *Test* page which contains the largest number of elements, and therefore is subjected to a large number of idle periods when using a single connection, as discussed above. The *acts* and the *oufr* pages both have smaller images and a similar number of elements. The *acts* page is dominated by a small number of large images and therefore the more efficient slow start phase in the single connection case keeps the single connection response time closer to the multiple connection response time than shown for the *oufr* page.

HTTP/1.1, even without a larger initial *cwnd*, achieves equal or better performance than HTTP/1.0 with 4 concurrent connections in all cases presented in figure 8. The performance gain is most apparent for the pages where the response times are dominated by the long RTT (all but the *oufr* page, which has a large amount of data). The larger initial window improves response time by roughly 1–1.5 RTTs for those pages where any sustained data flows are required. The *Test* page contains enough data to lead one to expect a performance gain from using a larger initial *cwnd*, however the data does not show such a performance increase. We speculate that browser or server effects related to the processing of a large number of simultaneous `GET` requests may cause a non-network related performance problem in this case.

Page	Corrupted Transfers (%)
LeRC	7
acts	13
oufr	41
Test	2

Table 2: Frequency of Bit-Errors

6.2 The Impact of BER on HTTP

Table 2 shows the percentage of HTTP/1.1 transfers that experienced bit-errors on the satellite channel during our experiments. The table shows that the larger WWW pages (e.g., *oufr*) were more likely to experience bit-errors while pages with smaller elements were not as likely to experience corruption, due to the slow start algorithm. As discussed above, slow start ensures that the transfer will have significant idle periods. Bit-errors on the channel during these idle periods do not impact TCP performance.

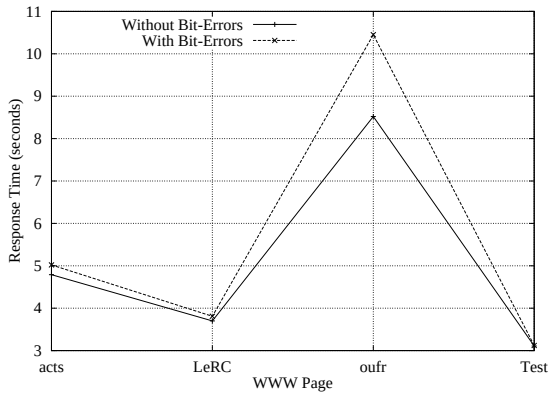

Figure 9: Impact of Non-Zero Bit-Error Rate on HTTP/1.1 Transfers

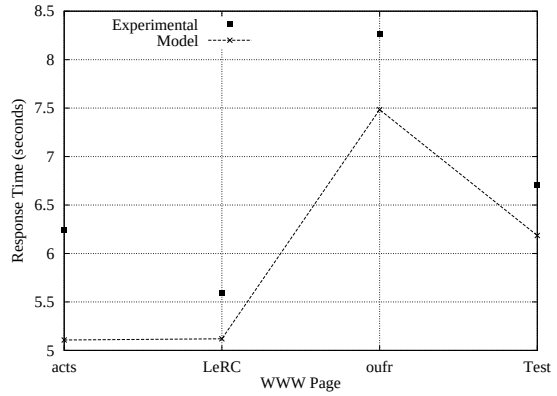

Figure 10: Model Comparison – HTTP/1.0, 4 Connections, No 4K Initial Window. Note this case is the same as the HTTP/1.1 without pipelining case.

Bit-errors (hopefully!) cause TCP's checksum to fail, which causes the segment to be silently discarded. TCP's congestion control algorithms interpret all segment loss as network congestion and decrease *cwnd* accordingly. Therefore, bit-errors hurt TCP performance by acting as false indications of network congestion. Figure 9 shows the impact of a non-negligible bit-error rate on HTTP/1.1 transfers. As shown, the impact on transfer time when the runs that experienced bit-errors are included in the analysis is fairly minimal. The small impact is due to the nature of HTTP traffic. The transfers are generally short and the performance dominated by the slow start algorithm. So, reducing *cwnd* from an already small value does not cause a major impact on performance. The *oufr* page has the most data and therefore is able to increase *cwnd* more than the other pages. Therefore, as expected, bit-errors hurt the performance of transfering this page more than the other 3 pages.

6.3 HTTP Model Comparison

Figures 10–13 compare the same experimental results shown in figure 8 to the model described in section 5. Figures 10 and 11 address the case of HTTP/1.0. We find that the model agrees reasonably well with the experimental results. Due to the unknown order in which the browser retrieves page elements, the model is designed to produce a lower bound on response time. Within this inherent limitation, we can predict the page response times quite well. There is no indication of systematic deviations between the experimental data and the model, indicating that all major factors influencing performance have been incorporated.

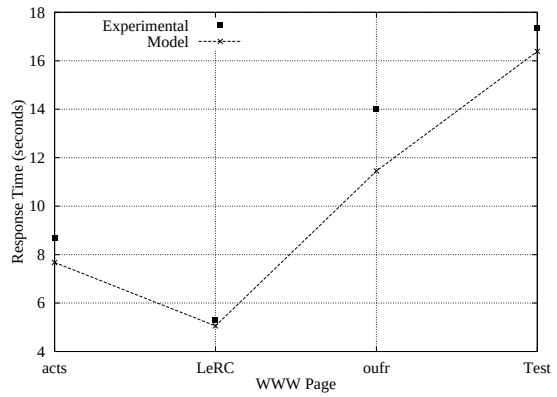

Figure 11: Model Comparison – HTTP/1.0, 1 Connection, 4K Initial Window.

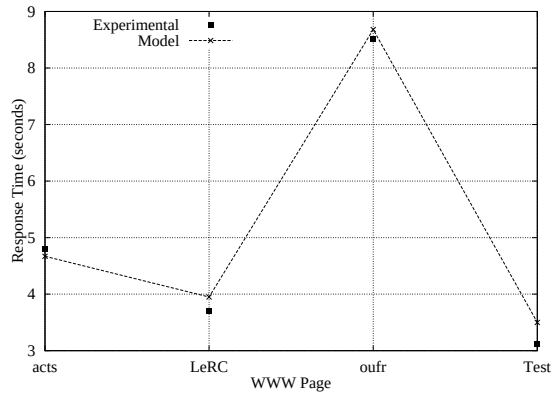

Figure 12: Model Comparison – HTTP/1.1 with pipelining, 1 Connection, No 4K Initial Window.

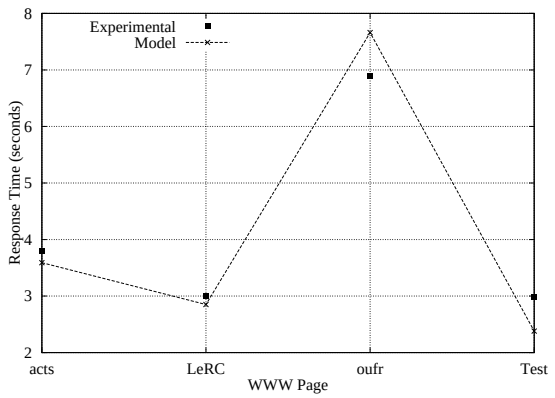

Figure 13: Model Comparison – HTTP/1.1 with pipelining, 1 Connection, 4K Initial Window

Figures 12 and 13 compare the HTTP/1.1 experiments to the model. Due to its reliance on a single TCP connection, modeling HTTP/1.1 is much less complex than HTTP/1.0, and therefore the agreement between the experimental data and the model is excellent. The model does not capture the transfer of the *Test* page as accurately as the other pages. As discussed above, we suspect that effects not related to HTTP or TCP play a role in this case.

There is also an underestimate of the experimental result by the model for the *oufr* page in figure 13. This page is sensitive to the growth of *cwnd*, and it is likely that a network effect is at work here. It is possible that the strict delayed acknowledgment discipline assumed in the model is not followed exactly by the actual TCP transfer, or that the application is causing the transmission of segment in an unexpected way. Further investigation is needed to resolve this case.

7 Conclusions and Future Work

In this paper we have analyzed the behavior of various popular methods of accessing WWW pages. The following are some of our key conclusions.

- We have shown that using a larger initial *cwnd* in conjunction with HTTP/1.1 with pipelining provides the best performance of all the HTTP versions tested across several WWW page scenarios in long-delay satellite networks.

- Our experiments have shown that pipelining is an important mechanism in long-delay environments.

- We have shown that using multiple parallel connections improves transfer time more than using a larger initial window when using HTTP/1.0. However, the negative aspects of using multiple concurrent connections [FF98, BPS+98] were not shown in our tests due to the lack of competing traffic.

- Our results indicate that links with low, but non-zero, BER will mainly effect WWW page retrievals with large page elements, i.e. those most resembling bulk transfers.

- We have extended the HTTP model presented in [HOT97] to accurately predict transfer time of most common forms of transfering WWW pages in use today.

Future work in this area includes quantifying the different HTTP mechanisms in a more realistic scenario with competing traffic and congestion-based loss. Additionally, further study of HTTP (and TCP) in real satellite environments with higher BERs is also needed.

Acknowledgments

The authors thank the following people for their contributions to this work. Shawn Ostermann wrote a module for *tcptrace* that helped us analyze the data presented in this paper. Jason Pugsley provided help running our tests. Eric Helvey helped us debug our NetBSD kernels. Joseph Ishac developed an early version of the connection packing algorithm outlined in section 5.3.3. Finally, the IJSC reviewers provided valuable feedback on this paper. Our thanks to all.

References

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[AGS99] Mark Allman, Dan Glover, and Luis Sanchez. Enhancing TCP Over Satellite Channels Using Standard Mechanisms, January 1999. RFC 2488.

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

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

[All98] Mark Allman. On the Generation and Use of TCP Acknowledgments. *Computer Communication Review*, 28(5), October 1998.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[BLFN96] Tim Berners-Lee, R. Fielding, and H. Nielsen. Hypertext Transfer Protocol – HTTP/1.0, May 1996. RFC 1945.

[BPS+98] Hari Balakrishnan, Venkata Padmanabhan, Srinivasan Seshan, Mark Stemm, and Randy Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *IEEE InfoCom*, March 1998.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[Bv91] R. Bauer and T. vonDeak. Advanced Communications Technology Satellite (ACTS) and Experiments Program Descriptive Overview. Technical report, NASA Lewis Reseach Center, 1991.

[FF98] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. Technical report, LBL, February 1998.

[FGM+97] R. Fielding, Jim Gettys, Jeffrey C. Mogul, H. Frystyk, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, January 1997. RFC 2068.

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

[HOT97] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the Performance of HTTP Over Several Transport Protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–630, October 1997.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[Jac90] Van Jacobson. Modified TCP Congestion Avoidance Algorithm, April 1990. Email to the end2end-interest mailing list. URL: ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[Mog95] Jeffrey C. Mogul. The Case for Persistent-Connection HTTP. In *ACM SIGCOMM*, pages 299–313, 1995.

[PAD+99] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, March 1999. RFC 2525.

[Pax97] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

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

[Spe97] Simon Spero. Analysis of HTTP Performance Problems, 1997. http://sunsite.unc.edu/mdma-release/http-prob.html.

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

[THO96] Joe Touch, John Heidemann, and Katia Obraczka. Analysis of HTTP Performance. Technical report, Information Sciences Institute, August 1996. http://www.isi.edu/lsam/publications/http-perf/.