DATA TRANSFER EFFICIENCY OVER SATELLITE CIRCUITS USING A MULTI-SOCKET EXTENSION TO THE FILE TRANSFER PROTOCOL (FTP)

Mark Allman, Shawn Ostermann
School of Electrical Engineering and Computer Science
Hans Kruse[†]
J. Warren McClure School of Communication Systems Management
Ohio University
Athens, OH 45701

ABSTRACT

In several experiments using NASA's Advanced Communications Technology Satellite (ACTS), investigators have reported disappointing throughput using the TCP/IP protocol suite over 1.536Mbit/sec (T1) satellite circuits. A detailed analysis of FTP file transfers reveals that both the TCP window size, and the TCP "Slow Start" algorithm contribute to the observed limits in throughput.

While it is tempting to approach a solution to this issue by raising the TCP window size, there are several issues which can not be addressed in that way: (1) In order to raise the TCP window size sufficiently to allow full utilization of a T1 circuit, the TCP Extended Window Option is required[4]. Commercial implementation of this option in major operating systems has not been completed. (2) The use of very large windows may actually hurt throughput if a moderate bit error rate is present on the satellite channel. (3) A change in the window size does not address the effect of the TCP Slow Start algorithm. We therefore propose an application-layer solution by adding an option to the standard FTP which uses multiple data connections. The use of multiple TCP connections allows the effective utilization of the channel bandwidth without an increase in the TCP window size. A similar approach has been suggested by Long et al for the transfer of specialized image databases, both via the Internet and over satellite links [7].

In this paper we summarize the experimental and theoretical analysis of the throughput limit imposed by TCP on the satellite circuit. We then discuss in detail the implementation of a multi-socket FTP (XFTP) client and server. XFTP has been tested using the ACTS system. We present results from these runs and discuss the interaction between the multi-socket application and the TCP/IP network, especially the queues in the IP routers. Our results show that a careful choice of the number of connections, or sockets, must be made. Too few connections result in wasted bandwidth, while too many connections lead to dropped packets due to queue overflows in the router; in this case the overall throughput is reduced. The optimal choice of the number of connections leads to a better than 90% utilization of the satellite circuit.

Finally, we discuss a preliminary set of tests on a link with non-zero bit error rates. XFTP shows promising performance under these circumstances, suggesting the possibility that a multi-socket application may be less effected by bit errors than a single, large-window TCP connection.

[†] One of the authors (HK) gratefully acknowledges that part of this work was made possible through his receipt of a NASA Summer Faculty Fellowship at the NASA Lewis Research Center.

BACKGROUND

About a year ago, we conducted a series of experiments to verify the performance reported for TCP over the ACTS system, and to determine the root causes for the fact that the satellite channel could not be fully utilized[6]. This section summarizes these results.

Motivation

Two factors combine to motivate a study of satellite transmissions of TCP/IP at this time. First, satellite communications continue to play an important role in business communication networks. Today's increasingly complex applications demand ever more bandwidth and sophisticated connectivity. In rural areas of the US, this connectivity can be achieved quickly and economically through satellite links, at least until the terrestrial network has a chance to catch up. Outside the developed countries, satellite communications may be the long-term solution to increasing communications needs. Finally, communications to mobile stations such as trucks, ships, or airplanes demand satellite based solutions.

The second factor is the unexpected longevity of today's "legacy" protocol stacks. While the Internet represents a large TCP/IP installed base, it has always been assumed that OSI-compliant protocols would replace TCP/IP during the migration of the Internet to commercial use. Instead it appears that commercial users are choosing to implement TCP/IP networks, and to connect to the Internet in its present form.

It seems therefore very likely that TCP/IP will not only remain prevalent in the corporate network, but that it will have to be used over satellite links as the network is extended into areas without adequate terrestrial infrastructure.

Preliminary Studies

The earlier studies were conducted at the ACTS Master Control Station (MCS), using Traffic Terminals #1 and #2. We used the same experimental setup as the current work; we therefore refer to the "Experimental Configuration" section below for details. Figure 1 shows the throughput obtained using the "stock" FTP implementation contained in the Sun OS 4.1.3 operating system.

We interpret the "saturation" in throughput, i.e. the fact that the throughput over the satellite circuit does not increase with increasing bandwidth, as an indication that the window size in TCP (which was 24KBytes[1] in this case) limits the throughput[3-5]. The fact that throughput is file-size dependent suggests the influence of the TCP "slow-start" algorithm[11] . Using actual packet traces captured during the file transfer, we can demonstrate the slow-start as shown in figures 2 and 3.

Using standard modeling techniques, we can attempt to predict the combined impact of the slow-start and the limited window size. The details of this model can be found in ref [6]. Figure 4 shows a comparison of the model for a 24KByte window size with the experimental results. The agreement between the experiment and the model, while not perfect, leads us to believe the combined effects of the slow-start algorithm and the window size limit are sufficient to explain the observed throughput limitations.

[1]In this paper we will use the term KBytes to represent 1024 bytes. In contrast, the term kbits will stand for 1000 bits, and kbytes for 1000 bytes.

Extensions to FTP

One of our design goals was that our modified versions of the FTP client and server software, referred to as *XFTP*, must be backward compatible with standard FTP software. To allow this, we needed to make some extensions to the protocol used by FTP [9]. Our enhanced version of FTP is based on the 4.4BSD Unix source code and has been compiled and run on Sun workstations running various flavors of SunOS. To understand our enhancements to FTP, one must first understand how FTP transfers files across a network.

The FTP file transfer protocol uses the TCP [8] transport protocol to communicate across networks. The user interface to FTP is through an FTP client application [1], which may have either a textual or graphical interface. An FTP client communicates with an FTP server at a remote machine to transfer files. The FTP server runs, by default, on TCP port 21.

When a user wants to transfer a file, he starts an FTP client and directs it to contact the FTP server on a given machine at port 21. The resulting TCP connection between the client and server is called the *control connection* and is used to send commands and result codes between the client and server applications. The control connection, however, is never used to transfer files; a separate TCP connection is created for each file transfer.

It is important to differentiate between the *user commands* issued by the user to the FTP client application and the *FTP commands* exchanged between the client and server applications[2]. In common, text-based client interfaces, the user types user commands such as `GET, PUT, BIN`, and `CD` to control the FTP client. The FTP client transforms these user commands into FTP commands sent to the server application. The enhancements that we made to the FTP client and server application to support multiple-connection file transfer required the addition of new user commands as well as new FTP commands.

How FTP Transfers Files

In the standard FTP implementation, the client and server software use the following sequence of actions to transfer a file (as shown in figure 5).

1) The client chooses an unused, local port number.
2) The client uses the "`PORT`" FTP command to inform the FTP server what port number to use for the next file transfer.
3) The server application acknowledges its willingness to use the port number by returning a "`PORT command successful`" message.
4) The client (in the case of file retrieval) tells the server which file it wants by using a "`RETR file`" FTP command.
5) The server creates a new TCP connection using the local port requested by the client, above.
6) The FTP server writes the requested file onto the new TCP connection and then closes the connection.
7) The server returns a "file complete" message to the client, once again using the original control connection.

Extended FTP Commands

For the FTP client and server to transfer a file using multiple TCP connections, we extended the sequence of actions in the following ways (as shown in figure 6):

[2] Unfortunately, the official FTP specification in RFC 959 refers to the tokens in both of these control languages as *commands*. We will use the terms *user commands* and *FTP commands* to distinguish between the two languages for clarity.

1) A new FTP command, "`MULT n`", was introduced. This command is a request from an FTP client to an FTP server to use *n* connections for file transfer rather than one. If the server has been extended to support multiple connections, then it accepts the request. If the server is a standard implementation, then it won't understand the request and will reject the command, in which case the client will use a single connection for file transfer.
2) A new FTP command, "`MPRT port1, port2, ...  portn`" was introduced. This new command is used by the client application in place of the "`PORT port`" command. The MPRT command, however, allows the client to specify *n* ports rather than just 1, where *n* was previously agreed upon by the client and server using the MULT command.

If either the client application or the server application is a standard application, then the MULT command will not be exchanged or accepted and data transfer takes place using a single connection, making our extended scheme backward compatible with current implementations of FTP.

Extended User Commands

Our enhanced client application, *xftp*, provides a single new user command, `MULT`. Before initiating a file transfer using multiple connections, the user must use the command "`MULT n`", where *n* is the number of connections that should be used to transfer files. The default value of 1 will be used if the user does not request multiple data connections. The `MULT` user command results in an exchange of FTP commands between the client and server applications to agree on the value of *n*.

Note that our current implementation requires the user to specify the number *n*, the number of concurrent data connections to use. An important research issue is to determine how the client and server can choose an appropriate value of *n* without the user's input. In a future version of the client interface, the user will simply type `MULT`, requesting that the client and server applications should attempts to use an "optimal" number of connections to transfer files. An appropriate definition of "optimal", in this context, remains an open issue.

Dividing a File across Multiple Connections

When multiple TCP connections are available to transfer a single file, the file must be divided into pieces that can sent over the various connections. This process is sometimes called *file striping*, after the commonly-used practice of *disk striping*, in which a single file is stored on multiple physical disks to increase throughput. File striping, in the context of TCP connections, however, must be carefully designed to avoid problems. A simple, obvious approach is to divide the file, *F*, into *n* blocks, $F_1 ... F_n$. Block F_i is then transferred using connection C_i.

This simple approach can perform badly, however, when the TCP connections do not all exhibit the same throughput. Because each of the connections is competing for the same resources, they may each perform differently. A primary cause of throughput variance is network congestion. The TCP protocol adapts to network congestion by *backing off*, or decreasing the rate at which it inserts information into the network [2]. Those connections that observe congestion will slow down, whereas connections that aren't affected by congestion will speed up and consume more network resources. As a result, if each of *n* connections transfers the same amount of data, some of the connections will take longer to complete. In the worst case, *n-1* of the connections have each transferred their blocks and one connection requires additional time to finish, which is clearly inefficient.

To optimize the transfer rate of a single file using n connections, each with a different throughput, an application cannot determine the division of the file into blocks beforehand. Our solution to this problem divides a single file into m records (where $m >> n$). Each record is then sent over a single connection. The sender of the file, which can be either the FTP client or server, reads the file from local storage one record at a time and sends each record over whichever connection has resources available to accept it. By dividing a file in this way, it is possible to keep each of the connections busy until the entire file has been transferred, even if the connections do not all transfer the same amount of data.

An application can easily determine if a TCP connection can accept more data. The TCP protocol maintains a *sending buffer* of a fixed size. As data from the sending buffer is sent across the network (and acknowledged by the receiver), TCP makes more space available in the sending buffer. An application can determine if a connection has available resources by checking the size of the sending buffer.

As a file's records arrive at the receiver (which may be either the client or the server), the application must put the records back into their proper order to recreate the file being transferred. To accomplish this, each record must contain an identification field that tells the receiver where the record belongs relative to the start of the file. Because these identification fields must also be sent over the network, they impose an additional overhead on the file transfer. In the current implementation, a file is divided into 8k records. Each record contains a 4-byte offset field that specifies the byte number of the first byte in the record. These values represent an added transfer overhead of $4/8096$ bytes $\approx .05\%$

Handling Multiple Connections

Adding multiple connections to FTP's file transfer model required that we greatly expand the complexity of FTP's connection management software. When using a single connection to transfer a file, FTP's connection handling algorithm could be fairly simple, as shown in figure 7.

The addition of multiple connections requires a more sophisticated I/O processing algorithm because the software must choose which TCP connection to use at each step. The primary problem is that access to the TCP connection can *block*, meaning that the calling application is forced to wait because the requested operation could not be immediately complete because of a lack of resources [10]. For example, if an application tries to read 1000 bytes from TCP and the TCP connection does not have 1000 bytes currently available, then the caller is blocked. When trying to optimize data transfer across many TCP connections, an FTP client or server that blocks is prevented from performing other useful work.

The most elegant solution to this problem would be to write a multi-threaded [10] version of the FTP client and server. Each thread could manage a single connection and blocking a single thread would not diminish performance. Unfortunately, multi-threaded applications cannot currently be used on many platforms. A more expensive alternative would be to use multiple processes rather than multiple threads. The overhead required for interprocess communication and context switching, not to mention the extra computer resources required, made this on unattractive alternative.

Our enhanced FTP client and server use a sophisticated system of asynchronous I/O and application-level buffering to prevent blocking. All access to the TCP connections is accomplished using non-blocking system calls. Because non-blocking read and write system calls have the option of accepting (or returning) less data than was requested, however, application-level buffering is required to hold the remaining data for a particular record until it can be accepted. The main processing loop used by our enhanced FTP applications is shown in figure 8.

EXPERIMENTAL STUDIES

Experimental Configuration

All tests reported here were conducted in the ACTS visitor center facility. The equipment diagram is shown in figure 9. Test files of various sizes were created on “pongo.lerc.nasa.gov”; the file contents consists of random ASCII characters (including non-printable characters). The XFTP client runs on “pongo...”. In all tests “PUT” commands are used to send the test file to the XFTP server on “perdita.lerc.nasa.gov”. On the server side, a “null” device is used to receive the file to avoid the cost of transferring the file back to disk. Ethernet packets can be captured on the network that connects “pongo...” to its associated router (“actsrtr2.lerc.nasa.gov”). Packet tracing is implemented using the Etherpeek™ software package running on a separate hardware platform (a Macintosh Centris).

Each of the two routers in the system have two hardware interfaces, one ethernet and one serial RS449. The RS449 interface is clocked off the CSUs, which in turn take clock information from the ACTS Traffic Terminals. The nominal data rate on the serial link is 1.536Mbits/sec[3] For test purposes, and when the satellite link is not active, the CSUs can be connected in a “back-to-back” bypass as shown in figure 5.

For most tests the ACTS Traffic Terminals #1 (TT1) and #2 (TT2) have been used. These terminals are part of the Master Control Station (MCS). They share the large-diameter reflector used for all MCS functions, and therefore operate error-free under almost all weather conditions. For tests on links with non-zero bit error rates, traffic terminal 1 or 2 was replaced by an ACTS T1 VSAT terminal (Traffic Terminal #3). The RF feed on this terminal had to be de-polarized to produce the desired bit error rates.

At the bottom of figure 9 we show the theoretical limits on the transmission rates between the workstations and the routers, and between the two routers. The rates in KBytes/sec also include the overhead appropriate for TCP transmission using 512byte packets. In the case of Ethernet, each packet carries a 58 byte header (Ethernet, IP, and TCP). Between the routers, a modified HDLC frame structure is used; the overhead here is 46 bytes. The Ethernet on the sending (left) side may be able to operate at the physical speed since it contains only two transmitting devices. The receiving Ethernet also contains a gateway to the Internet (not shown in figure 9), and will therefore not be able to run at line speed.

All file transfers were timed using time-stamp system calls within the XFTP code. All performance figures reported in this paper were obtained this way. For selected file transfers we have also recorded the packet headers of all packets seen on the “sending” Ethernet segment. These packet traces are used to evaluate the details of the file transfer operation; while we will refer to a few of these results, a detailed discussion of the trace file analysis is beyond the scope of this paper.

Results

Our original measurements using FTP showed that a single data connection could yield a throughput of about 40KBytes/sec, or approximately 24% of the available capacity[4]. Using the

---

[3]Mbits/sec equal 10^6 bits per second.

[4]Refer to figure 5 and the discussion of the overhead computations in the text.

model described in ref [6], we obtain reasonable agreement between the measurements and the model predictions. The model is centered around the fact that FTP is forced to send data in discrete "blocks", since the TCP window for the data connection fills up long before an acknowledgment can be received, given the round-trip time of approximately 590msec. The size of the data blocks is time-dependent due to the slow-start algorithm, which initially allows only one TCP segment to be sent. After that, each acknowledgment will double the available window size up to the full TCP window (which can be up to 64KBytes without the extended window option; in practice most operating systems limit the TCP window to a lower value). If this maximum window size is less than the product of the round-trip delay and the data rate of the channel (taking into account the packet overhead), there will still be periods of time when the channel is unused, because the window has been exhausted, and the next acknowledgment is still in transit. For a T1 channel, this will occur for windows less than about 100KBytes.

The multi-socket XFTP still shows that same behavior. In this case, however, the multiple data connections combine to create an "effective" window size, equal to the product of the number of data connections and the window size allocated to each connection. For example, with a 24KBytes TCP window and 4 connections, the effective window size will be 96KBytes, very close to the window needed to saturate the channel. The slow-start will allow <u>each connection</u> to begin by transmitting one TCP segment. The initial effective window is therefore equal to the number of data connections; this window will double in size with each <u>group</u> of acknowledgements (one per connection). By extending the model in this way, and taking into account the limit on the number of segments that can traverse the channel in a given time interval, we can attempt to predict the performance of the multi-socket XFTP.

Figure 10 shows both the experimental and the model results. All tests were conducted using a TCP window of 24KBytes. The model shows the expected near-linear performance improvement up to 4 data connections. Full channel speed cannot be reached due the time lost in the slow-start sequence. The model predicts a small performance increase past 4 data connections since the channel can be saturated a little sooner with more connections.

The experimental results agree in general with these predictions, with a number of important differences. The performance increase from 1 data connection to 4 data connections is not as high as the model predicts. By the time we reach 8 data connections (the effective window in that case is about twice what should be required to saturate the channel), the experimental and the modeled throughput agree again. At that point, the throughput represents a 91% utilization of the T1 link.

In the experiment, attempts to operate with more than 8 data connections lead to packet loss in the router queues, with large performance penalties due to packet re-transmission. Two router queues are involved in this case, both in "actsrtr2.lerc.nasa.gov". The router maintains an input queue for packets from the Ethernet segment, and an output queue for packets waiting for transmission over the ACTS link. Both of these queues will fill up if the effective window size allows data to be sent faster than the T1 channel rate. The queue lengths are adjustable within the limits of total router memory available. In general, long queues are not desirable since they add to the overall round-trip delay. For our measurements, we decided to allocate all available[5] router memory to these two queues in

[5]Some buffer memory is needed for the T1 input queue and the Ethernet output queue. Since both of these queues are at transition points from lower to higher data rates, only a small amount of memory is needed for each one.

equal parts. Note that no adjustments were needed on “actsrtr2.lerc.nasa.gov”, since the return traffic of acknowledgment segments cannot exceed the T1 rate at which data is received by the server. A detailed analysis of the queue behavior in the router, and the XFTP performance in the area beyond an effective window of 100KBytes is still in progress.

To determine the validity of the effective window concept, we conducted tests using several different settings for the TCP window and the number of data connections. The results are summarized in figure 11, showing throughput as a function of the effective window size, for transfers of 5MByte files. In this case, the achievable throughput is clearly dependent only on the effective window size. There are no observable performance penalties for the use of a large number of data connections, compared to the use of large TCP windows. Note that slow-start has a very small effect for files of this size. Large numbers of data connections may have a slight advantage for smaller files since the slow-start throughput penalty can be overcome a little sooner.

All results reported to this point were obtained on an error-free link. In figure 12 we present preliminary results obtained during test runs on a T1 VSAT link which was not error-free. The RF feed of the VSAT was adjusted to weaken both the receive and the transmit signal. The resulting link had a bit error rate between 5×10^{-6} and 1×10^{-5} in the direction that the data was flowing, and between 1×10^{-7} and 5×10^{-7} in the reverse direction. The effective window size was held constant at 192KBytes for all three data points. Contrary to the error-free case, the configuration using a large number of independent data connections shows a much better performance than cases where fewer connections with larger TCP windows were used.

CONCLUSIONS

Our preliminary experiments clearly show that it is possible to achieve throughput as high as 90% across a satellite link running at 1.544Mbps with an unmodified TCP implementation.

We have shown that, by combining several TCP connections in parallel, it is possible to achieve effective window sizes that allow very efficient file transfer over long, narrow channels such as satellite links. Newer standards for TCP allow applications to specify sending and receiving buffer sizes that result in higher throughput across satellite links. These versions of TCP, however, have been slow in reaching end users. Historically, it is also unlikely that many installed computers will be upgraded to newer versions of TCP. More important than the lack of new TCP implementations, however, is that multiple TCP connections may work better than a single TCP connection with a huge buffer size. Preliminary experiments over links with mild error rates suggest that multiple TCP connections achieve a higher aggregate throughput in the presence of errors.

Our exhanced version of the FTP client and server, XFTP, is completely backward compatible with existing FTP implementations. XFTP allows a user to transfer a single file over a long narrow channel at throughput levels well above those achievable with conventional FTP implementations. Determining the optimal number of connections to use for file transfer, however, is still an open issue. We are currently working on several theoretical models that may allow us to build an application that can choose a near optimal number of connections for a particular file transfer without user intervention.

Our prototype software was built into an FTP client and server. Our algorithms could be adapted for use in other network applications, with obvious examples being HTTP (used by world wide web applications), and NNTP (used by network news).

REFERENCES

[1] Comer, D.E., *Internetworking with TCP/IP Volume I, Principles, Protocols, and Architecture*, 1988, Prentice Hall.

[2] Jacobson, V. *Congestion Avoidance and Control*. in *Proceedings ACM SIGCOMM '88*. 1988. .

[3] Jacobson, V. and R. Braden. *TCP Extensions for Long-Delay Paths*. LBL 1988; Internet RFC 1072.

[4] Jacobson, V., R. Braden, and D. Borman. *TCP Extensions for High Performance*. LBL 1992; Internet RFC 1323.

[5] Jacobson, V., R. Braden, and L. Zhang. *TCP Extension for High-Speed Paths*. LBL 1990; Internet RFC 1185.

[6] Kruse, H. *Performance of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination*. in *3rd International Conference on Telecommunication Systems Modeling and Design*. 1995. Nashville, TN.

[7] Long, L.R., L.E. Berman, and G.R. Thoma. *Client/Server Design for Fast Retrieval of Large Images on the Internet*. in *Proceedings of the Eighth IEEE Symposium on Computer-Based Medical Systems (CBMS '95)*. 1995. Lubbock, TX.

[8] Postel, J. *TRANSMISSION CONTROL PROTOCOL*. ISI 1981; Internet RFC 793.

[9] Postel, J. and J. Reynolds. *FILE TRANSFER PROTOCOL (FTP)*. ISI 1985; Internet RFC 959.

[10] Silberschatz and Galvin, *Operating System Concepts*, 4th ed. 1994, Addison Wesley.

[11] Stevens, W.R., *TCP/IP Illustrated: The Protocols*, Vol. 1. 1994, Addison-Wesley.

FIGURES

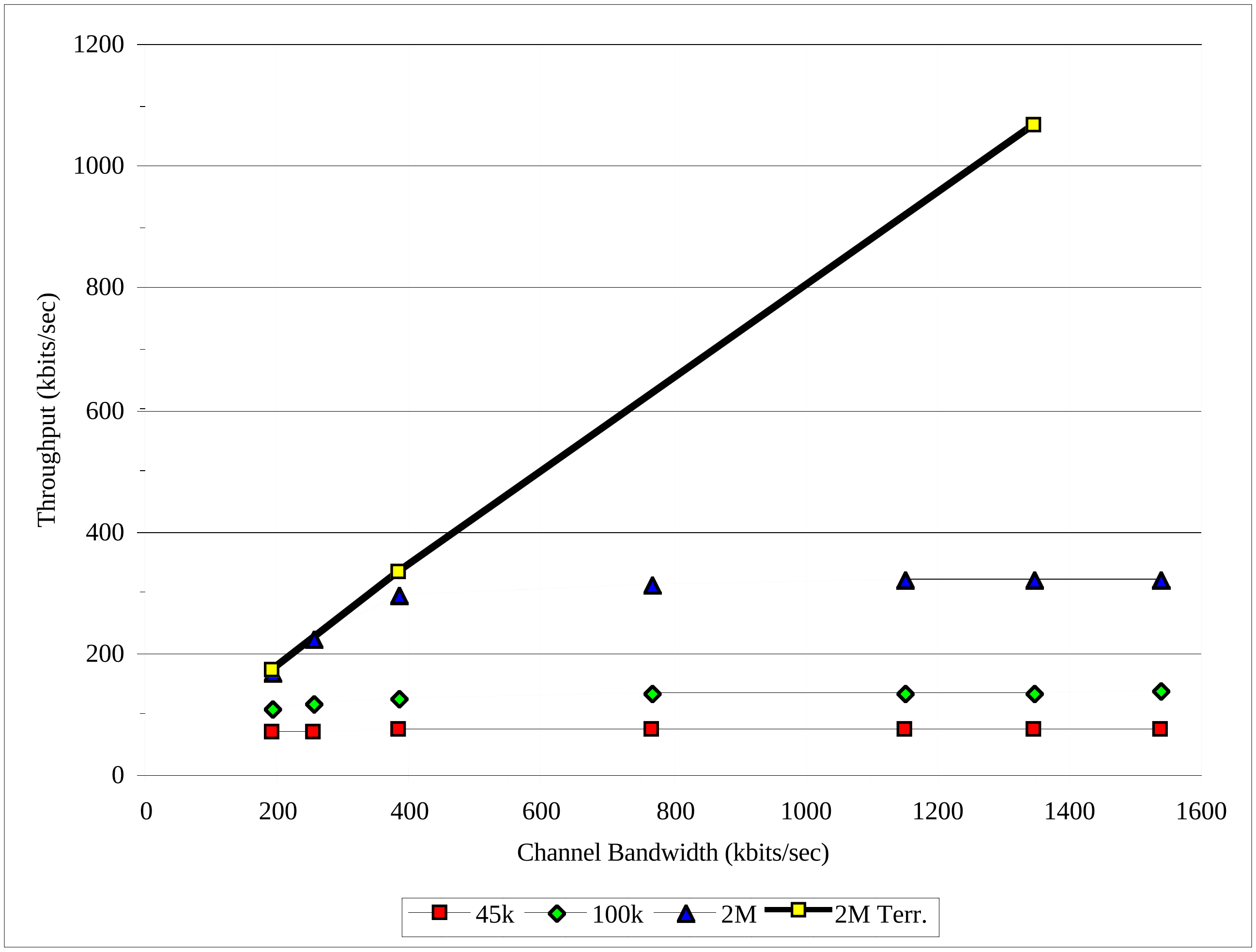

Figure 1 — Measured throughput, defined as the ratio of file size to the file transfer time, for the transfer of different size files at different channel data rates. The “2M Terr.” line is the terrestrial baseline case, all other measurements were made over the satellite channel. This figure that the satellite channel’s bandwidth cannot be fully utilized. As the bandwidth of the satellite channel is increased, throughput initially also increases, but then levels off. The bandwidth at which throughput levels off, and the maximum observed throughput are both dependent on the size of the file being transferred.

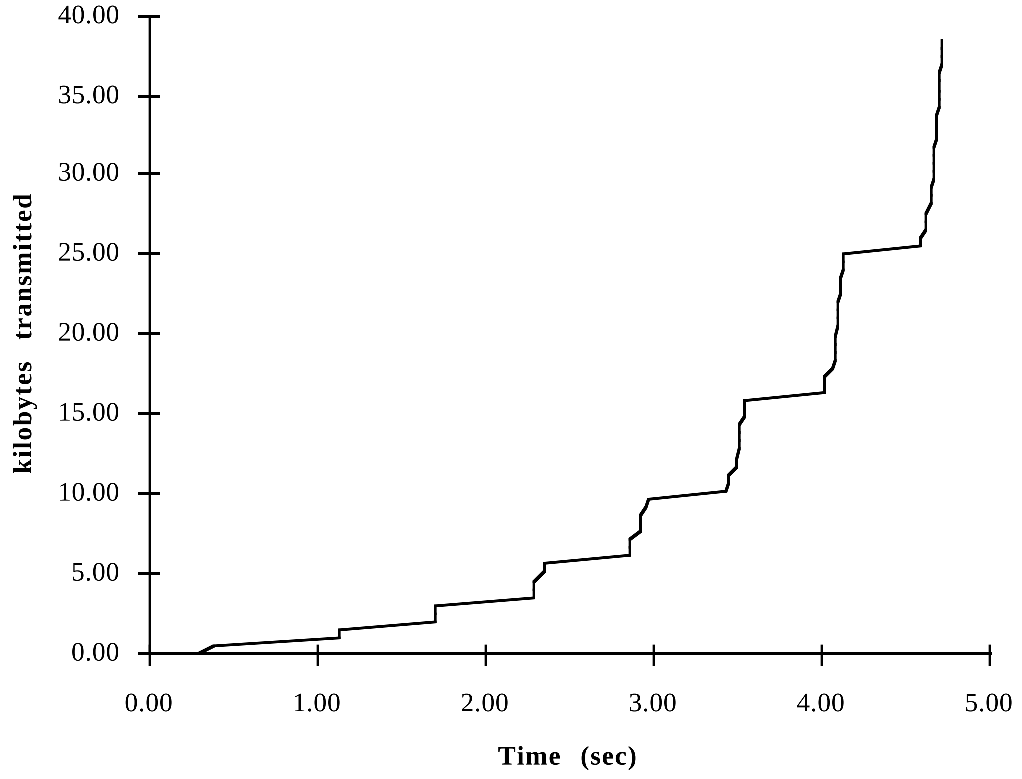

Figure 2 — Amount of data transferred as a function of elapsed time from the start of the file transfer. The data is being sent on a 1.344Mbits/sec satellite channel. The data shown here covers the first 5 seconds of the file transfer. The near-horizontal portions of the graph represent times when the window is exhausted, and the sender is waiting for an acknowledgment. The amount of data transferred between wait periods increases as the slow-start algorithm allows the window to open up.

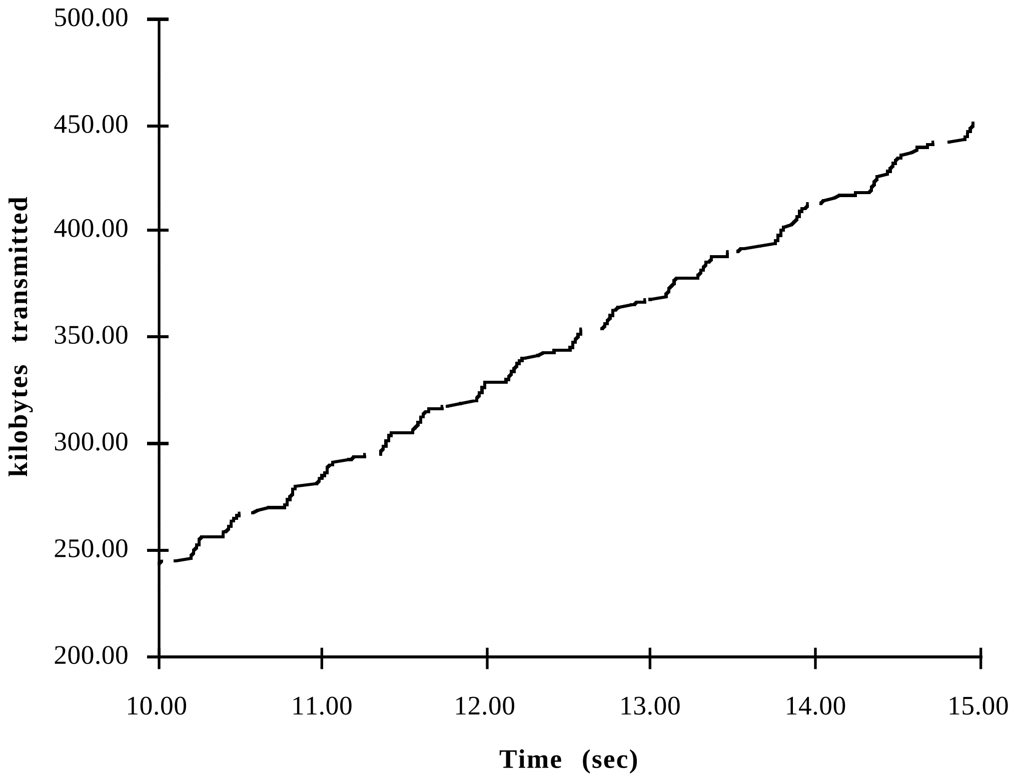

Figure 3 — Same as figure 2, but at a later time in the file transfer. Note that the horizontal scale covers a much larger range than in figure 2. The wait periods are still visible, but not as pronounced. The amount of data transmitted between wait periods is roughly equal to the window size.

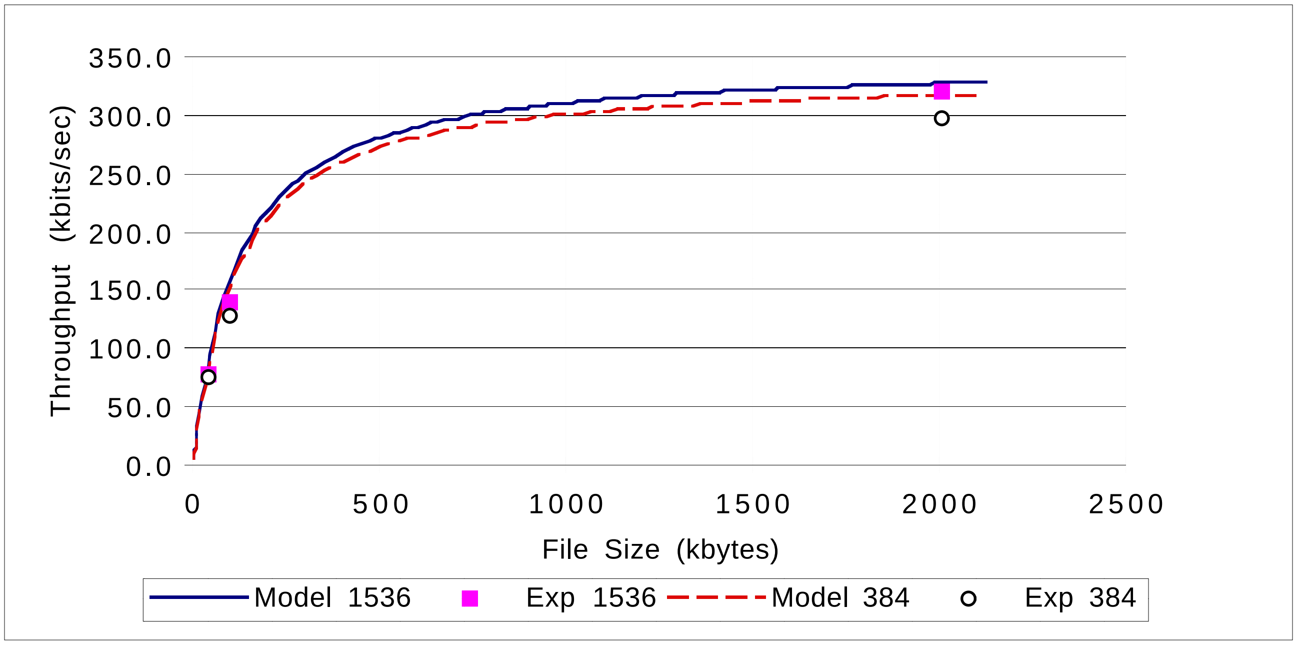

Figure 4 — Comparison of the slow-start model and experimental results obtained at channel speeds of 1.536Mbits/sec (solid line and filled squares), and 384kbits/sec, for different file sizes. The experimental results are based on the unmodified FTP, with a window size of 24KBytes. Agreement between the model and the experiment is generally good; the model tends to predict somewhat higher throughput than is observed for a given file size and channel bandwidth.

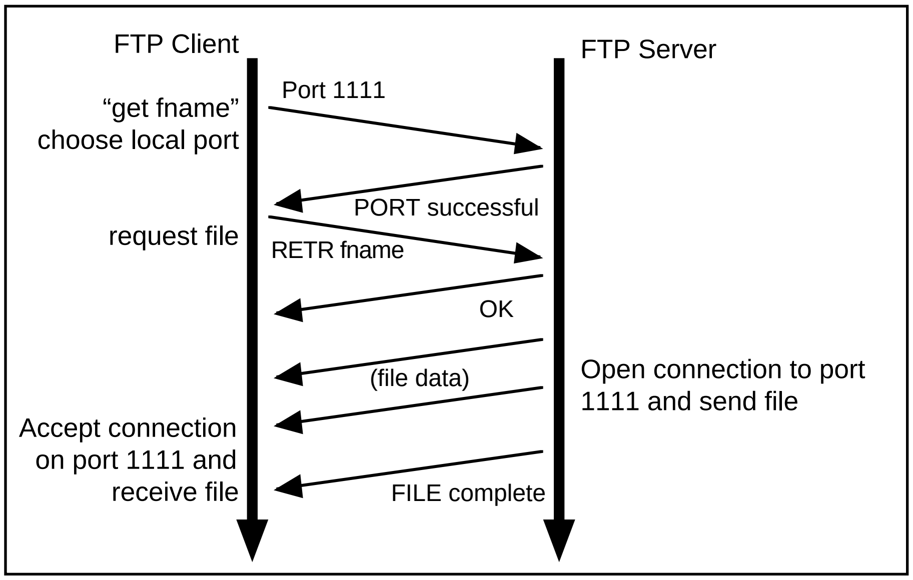

Figure 5 — Normal FTP Action Sequence. This figure shows the normal sequence of actions that takes place when a user retrieves a file fname using standard FTP.

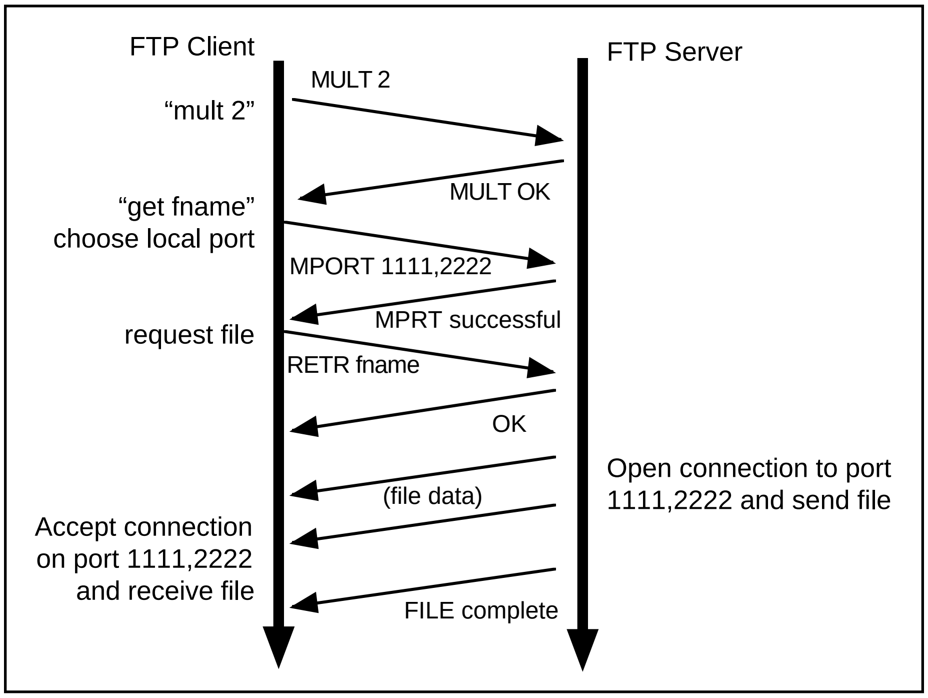

Figure 6 — Extended FTP Action Sequence. This figure shows the sequence of actions that takes place when a user retrieves a file using enhanced FTP. The file is transferred using TCP connections on ports 1111 and 2222.

```
Repeat while not eof(file) {
     Read a buffer from the file
     Write the buffer to the connection
}

Repeat while not eof(tcp) {
     Read a buffer from the connection
     Write the buffer to the file
}
```

Figure 7 — FTP Main File Transfer Algorithms. This figure shows the simple logic used for FTP's main file transfer input and output routines.

```
while (not done) {
     FD_SET(wmask, interested_file_descriptors);
     select(wmask);  /* wait for available connection */
     for (each usable file descriptor fd) {
          len = write(fd,buffer);
          if (len < sizeof(buffer))
               bufferup(the_rest)
     }
}
```

Figure 8 — XFTP Input Processing. This code fragment shows the algorithm used for writing the multiple TCP connections. Input processing is similar.

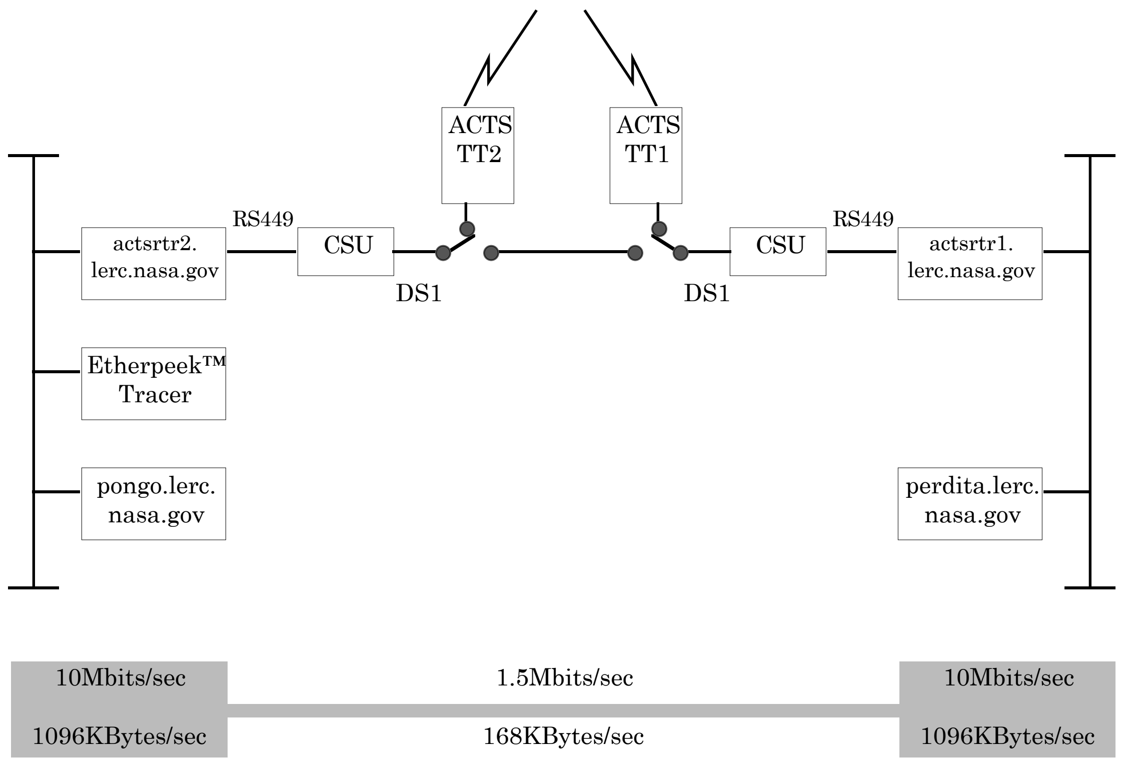

Figure 9 — Block diagram of the experiment configuration. “pongo.lerc.nasa.gov” and “perdita.lerc.nasa.gov” are Sun IPX workstations, the tracing software runs on a Macintosh Centris System; “actsrtr1.lerc.nasa.gov” and “actsrtr2.lerc.nasa.gov” are Cisco Systems 2500 series routers. ACTS traffic terminals #1 and #2 (TT1 and TT2) are shown; in some experiments the T1 VSAT terminal #3 was used in place of either TT1 or TT2.
Below the equipment diagram is an indication of the maximum line speeds on the ethernet segments and the router connection. Traffic between the workstations and their associated routers travels over 10Mbits/sec channels. Taking overhead into account, the theoretical user data transfer rate on these channels is 1096KBytes.sec. The connection between “actsrtr1.lerc.nasa.gov” and “actsrtr2.lerc.nasa.gov” is made at 1.536Mbits/sec unless otherwise noted. At that bit rate, and taking into account overhead, the maximum user data transfer rate is 168KBytes/sec.

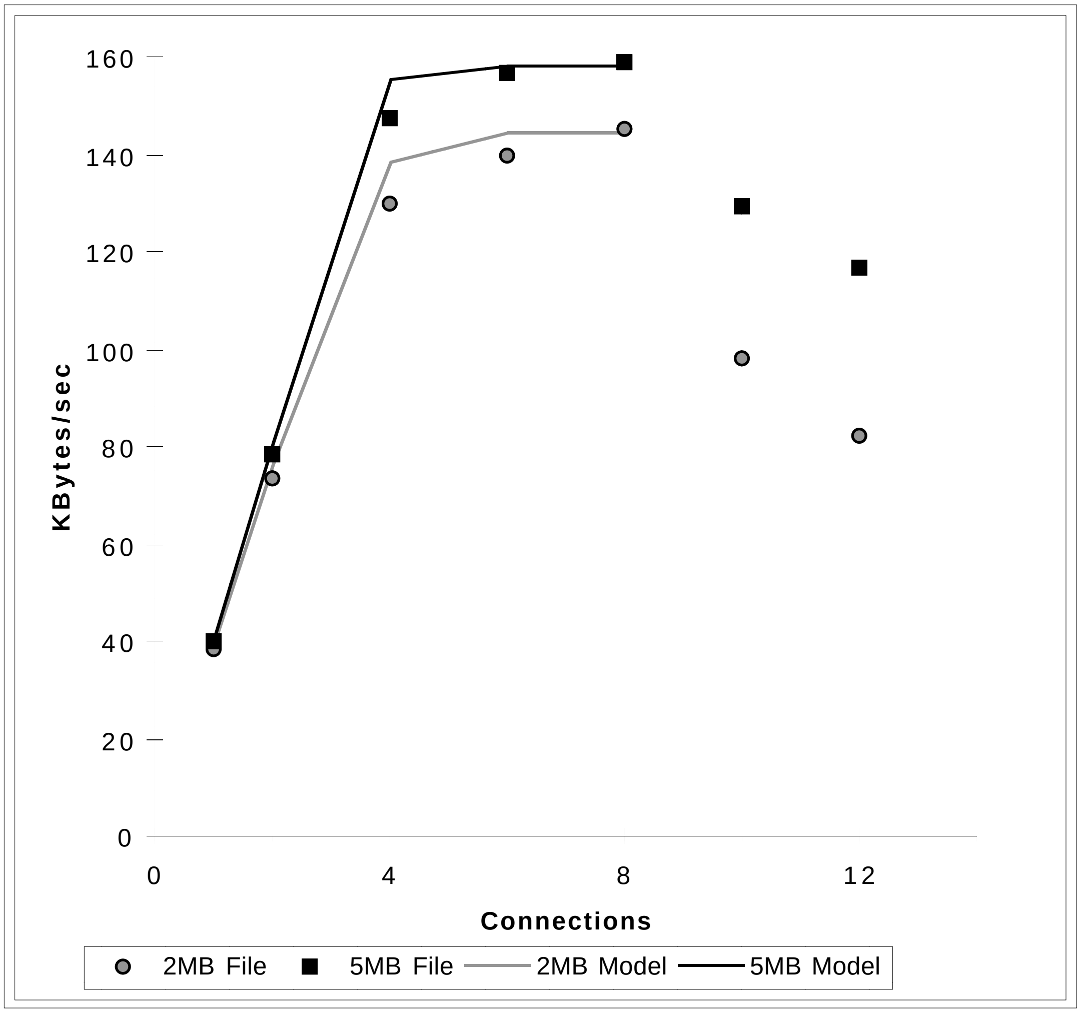

Figure 10 — Performance of 2Mbyte and 5Mbyte file transfers using varying numbers of data connections. The symbols represent experimental measurements, while the solid lines are the result of a model calculation including slow-start, TCP window size, and multi-socket operation.

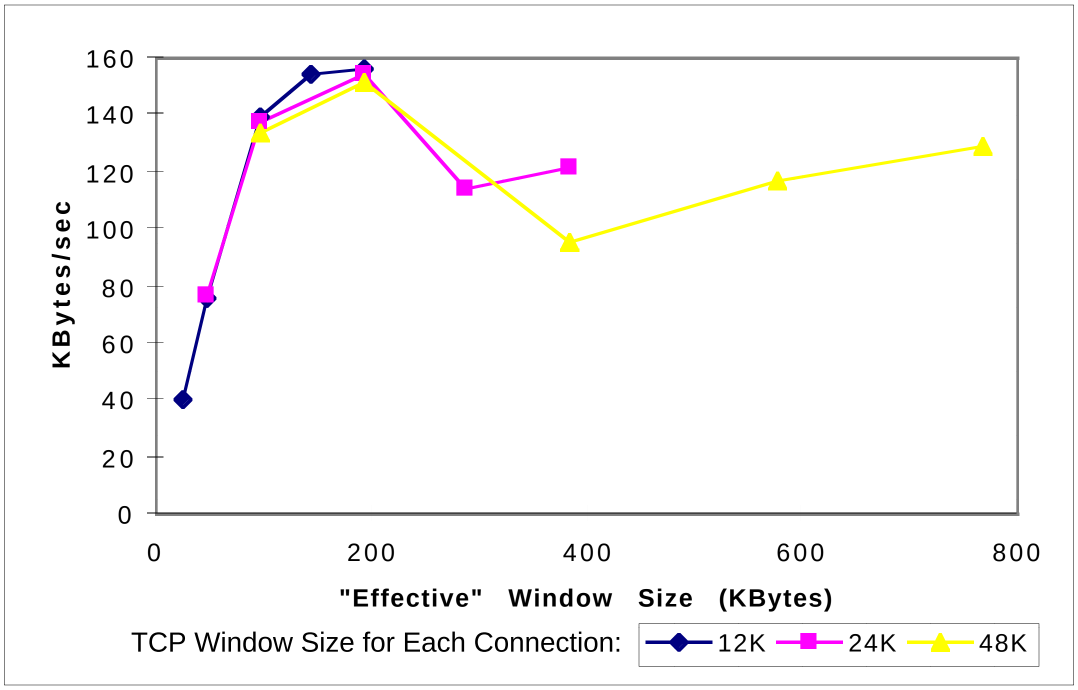

Figure 11 — A comparison of 5MByte file transfers with different TCP windows and different numbers of data connections. The effective window size is the product of the TCP window size and the number of data connections.

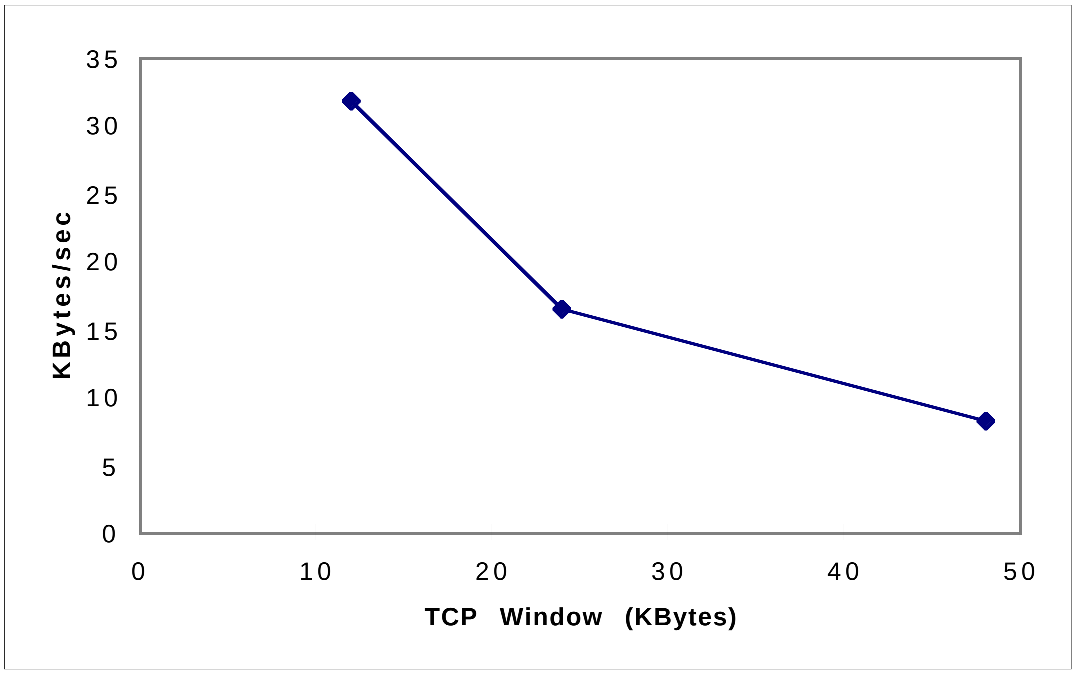

Figure 12 — Throughput on a link with non-zero bit error rates. During these tests, bit error rates were measured between 5×10^{-6} and 1×10^{-5}. The number of connection was adjusted for each of the TCP window sizes used to keep the effective window size constant from test to test.