Improving Multiple File Transfers Using SCTP Multistreaming*

Sourabh Ladha, Paul D. Amer
Protocol Engineering Lab
Computer and Information Sciences
University of Delaware
{ladha, amer}@cis.udel.edu

Abstract

We identify overheads associated with FTP, attributed to separate TCP connections for data and control, non-persistence of the data connections, and the sequential nature of command exchanges. We argue that solutions to avoid these overheads using TCP place an undue burden on the application. Instead we propose modifying FTP to use SCTP and its multistreaming service. FTP over SCTP avoids the identified overheads in the current FTP protocol without introducing complexity at the application, while still remaining "TCP-friendly". We implemented FTP over SCTP in three ways: (1) simply replacing TCP calls with SCTP calls, thus using one SCTP association for control and one SCTP association for each data transfer, (2) using a single multistreamed SCTP association for control and all data transfers, and (3) enhancing (2) with the addition of command pipelining. Our experiments compared these 3 variations with the classic FTP over TCP. Results indicate significant improvements in throughput for multiple file transfers with all three of our variations. The largest benefit occurs for (3) FTP over a single, pipelined, multistreamed SCTP association. More generally, this paper encourages the use of SCTP's innovative services to benefit existing and future application performance and presents the case for multistreaming.

1. Introduction

The past decade has witnessed an exponential growth of traffic in the Internet, with a proportionate increase in Hyper Text Transfer Protocol (HTTP) [BFF96] and decline in File Transfer Protocol (FTP) [PR85], both in terms of use and the amount of traffic. The decline in FTP traffic is chiefly attributed to the inflexible nature of its interface and inefficiency in its end-to-end delay performance. Over the years several FTP extensions have been proposed (e.g., [AOM98], [EH02], [HL97]), but few aim at reducing file transfer latency [Kin00, AO97]. FTP uses TCP to provide end-to-end reliability. In this paper, we identify reasons why modifying FTP to reduce latency overheads has been difficult, mainly due to TCP's semantics which constrain the FTP application. One result of these constraints has been that several FTP implementations aiming to enhance performance use parallel TCP connections to achieve better throughput. However, opening parallel TCP connections (whether for FTP or HTTP) is regarded as "TCP-unfriendly" [FF99] as this allows an application to gain an unfair share of bandwidth at the expense of other network flows, potentially sacrificing network stability. Moreover multiple parallel TCP connections consume more system resources than are necessary. This paper focuses on improving end-to-end FTP latency and throughput in a TCP-friendly manner.

Although FTP traffic has proportionately declined in the past decade, FTP still remains one of the most popular protocols for bulk data transfer on the Internet [MC00]. For example, Wuarchive [WUARCHIVE] serves as a file archive for a variety of files including mirrors of open source projects. Wuarchive statistics for the period of April 2002 to March 2003 indicate FTP accounting for 5207 Gigabytes of traffic, and HTTP accounting for 7285 Gigabytes of traffic. FTP is exclusively used in many of the mirroring software on the Internet, for various source repositories, for system backups and for file sharing. All these applications require transferring multiple files from one host to another.

In this paper we identify the overheads associated with the current FTP design. We present modifications to FTP to run over Stream Control Transmission Protocol (SCTP) [SXY+03] instead of TCP. SCTP is an IETF standards track transport layer protocol. Like TCP, SCTP provides an application with a full duplex, reliable transmission service. Unlike TCP, SCTP provides additional transport services. This paper focuses on the use of one such service: multistreaming. SCTP multistreaming logically divides an association into streams with each stream having its own delivery mechanism. All streams within a single association share the same congestion and flow control parameters. Multistreaming decouples data delivery and transmission, and in doing so prevents Head-of-Line (HOL) blocking.

This paper shows how SCTP multistreaming benefits FTP in reducing overhead, especially for multiple file transfers. We recommend two modifications to FTP which make more efficient use of the available bandwidth and system resources. We implemented these modifications in a FreeBSD environment, and carried out experiments to compare the current FTP over TCP design vs. our FTP over SCTP designs. Our results indicate dramatic improvements in transfer time and throughput for multiple file transfers under certain network conditions. Moreover, our modifications to FTP solve concerns that current FTP protocol faces with Network Address Translators (NAT) and firewalls in transferring IP addresses and port numbers in the payload data ([AOM98], [Tou02], [Bel94]).

The remainder of this paper is organized as follows. Section 2 summarizes FTP, focusing on features we changed in order to use SCTP. This section also details and quantifies the overheads

*Prepared through collaborative participation in the Communication and Network Consortium sponsored by the U.S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

in the current FTP over TCP design. Some security concerns in FTP over TCP have also been noted. Section 3 discusses possible solutions to eliminate these overheads while still using TCP as the transport. Section 4 introduces SCTP multistreaming. Section 5 presents our protocol changes in FTP to exploit using SCTP multistreaming, and a description of how these designs reduce the overheads. Section 6 presents the experimental results and discussion. Section 7 concludes the paper.

2. FTP Protocol Elements

This section presents briefly the elements of the FTP protocol and its sequence of operations. It then quantifies the overheads in FTP's operation.

2.1 FTP over TCP Design

FTP currently operates atop TCP's reliable, byte stream service. An FTP session consists of one control connection, and one or more data connections. The control connection is used for the exchange of commands and replies in simple ASCII format. Each command and reply typically consists of 20-40 bytes. The exchange of commands and replies over the control connection is periodic in nature triggered by user requests. A unique data connection is established for each file transfer or directory listing transfer, and is terminated after the transfer. The closing of the data connection indicates the End of File (EOF). Thus the number of data connections in an FTP session is equivalent to the number of transfers performed. Each data connection follows one of the two modes, active or passive depending on whether the server or client initiates the connection, respectively. In the active mode, the client sends a PORT command to the server indicating the IP address and the port number to which the server should establish the data connection. Extensions to FTP [AOM98] introduce additional commands (e.g., EPSV, EPRT) mainly for operations in the passive mode where the client opens the connection to the server. The passive mode also solves, to some extent, the problems FTP faces in interacting with NATs and firewalls [Bel94].

The common user service commands for file transfer are RETR, LIST, STOR, NLST, APPE [PR85]. One of the recent additions to the FTP command set proposed in [EH02] includes the SIZE command. The SIZE command requests the size of the remote file to be transferred, before the file is actually requested with a RETR command. The actual size is returned in a "213 reply". Knowing the file size can assist a receiver to determine the restart marker and the number of bytes left to be read under a restart condition, which may be caused by an end host crash or network failure.

FTP provides the retrieval of multiple files based on an expression given by the user, for example, using "*mget* * ". The files are transferred independently and no form of connection information is shared between each file's transfer. Each transfer requires the client to send PORT, SIZE and RETR (or equivalent) control commands. The total number of data connections consumed for a multiple file transfer request is (n+1): one to transfer of the name list of files, and one for each of the n file transfers. Figure 1 shows a timeline for multiple file retrieval, from the server to the client. The timeline shows the commands and replies exchanged, and the TCP connection establishment-teardown for the data transfer. (Not shown in Figure 1 is the TYPE command and its response. Moreover implementations may use extra commands exchanges prior to the data transfer. The time line is meant for the reader to understand the basic command exchanges in FTP and for comparison purposes to the modifications introduced later in this paper.) The solid and the dotted line in Figure 1 represent the transfer on the control and data connection, respectively. The dotted box represents operations repeated sequentially for each file transferred.

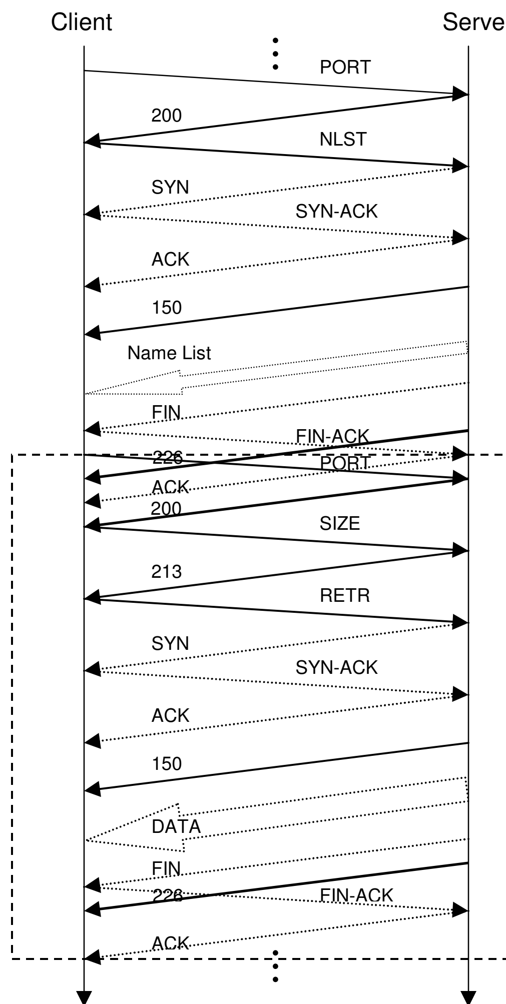


Figure 1: Current FTP over TCP

2.2 Inefficiencies in the current FTP design

FTP's current design includes a number of inefficiencies due to (1) separate control and data connection and (2) non-persistent data connection. Each is discussed in turn.

2.2.1 Distinct control and data connection

A. FTP's out-of-band control signaling approach has consequences in terms of end-to-end latency. Traffic on the control connection is periodic in nature, and hence this connection typically remains in the slow start phase of TCP congestion control [APS99]. The control connection is vulnerable

to timeouts because of the send-and-wait nature of control commands. (Also, insufficient packets are flowing to cause a TCP fast retransmit.) Thus, an operation (involving a single control command) will be subject to a timeout in the event of loss of either a command or its reply.

B. Since control and data flow on separate connections, an extra overhead of at least 1.5 Round Trip Time (RTT) is incurred for connection setup-teardown (1RTT for setup and 0.5 RTT for teardown). Moreover the end hosts create and maintain on average two Transport Control Blocks (TCBs) for each FTP session. This factor is negligible for clients, but may significantly degrade performance of busy servers that are subject to reduced throughput due to memory block lookups [FTY99].

C. Over the past years there have been considerable discussions on FTP's lack of security, some of them attributed to data connection information (IP address, port number) being transmitted in plain text in the PORT command on the control connection to assist the peer in establishing a data connection. Moreover, transferring IP addresses and port numbers in the protocol payload creates problem for Network Address Translators (NATs) and firewalls which must monitor and translate addressing information [AOM98, Tou02].

2.2.2 Non-persistence of the data connection

A. The non-persistence of the data connection causes connection setup overhead at least on the order of 1 RTT each time a file transfer or directory listing request is serviced (see Figure 1). Queuing delays can significantly increase the RTT [PM94]. To improve end-to-end delays, every attempt should be made to minimize the number of round trips.

B. Every new data connection causes a new probing of the congestion window (cwnd) during the connection's slow start phase. Each connection begins by probing for the available bandwidth before it reaches its steady state cwnd. Moreover, a loss early in the slow start phase, before the cwnd is large enough to allow for fast retransmit, will result in a timeout at the server. Figure 2 graphically shows the nature of this re-probing overhead in the event of three consecutive file transfers. The interval between the transfers indicates the time involved in terminating the previous connection, setting up a new connection, and transferring control commands. (The reader should be able to understand that this is a generic example and not an exact indication of cwnd evolution.)

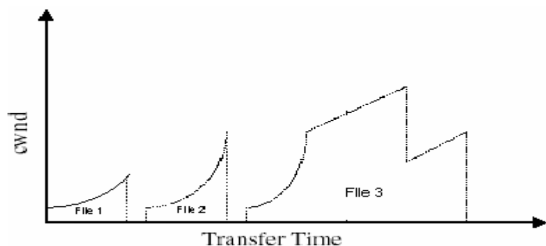


Figure 2: Expected cwnd evolution during a multiple file transfer in FTP over TCP

C. For each file transfer, a one RTT overhead is incurred for each exchange of the PORT command and its 200 reply (see Figure 1).

D. In the event of multiple small file transfers, the server ends up having many connections in the TCP TIME-WAIT state and hence maintain on average more than two TCBs per session. This per-connection memory load can adversely affect a server's connection rate and throughput [FTY99].

3. Possible solutions and drawbacks

We describe some of the possible solutions that try to avoid the above stated overheads while still using TCP as the underlying transport service. The drawbacks associated with each solution are presented.

A. Use a single persistent TCP connection for both control and data

Improvements: This approach avoids most overheads associated with FTP's current design listed in the previous section. The commands over the control connection can be pipelined (in the event of a multiple file transfer) to improve latency, and maintain the probed congestion window for subsequent transfers.

Drawbacks: TCP provides a byte-stream service and does not differentiate between the different types of data it transmits over the same connection. Using a single TCP connection requires the application to use markers to differentiate between control and data. This marking burden increases application layer complexity. Control and file data in an FTP session are logically different types of data, and conceptually are best kept logically, if not physically, separate. Additionally, using a single connection risks Head-of-Line (HOL) blocking (HOL blocking is discussed more in Section 4).

B. Use two TCP connections: one for control, and one persistent data connection

Improvements: A persistent data connection eliminates the connection setup-teardown and command exchange overheads for every file transfer, and thus reduces round trips.

Drawbacks: Due to the sequential nature of commands over the control connection, the data connection will remain idle in between transfers of a multiple files transfer. During this idle time, the data connection congestion window may reduce to as much as the initial default size, and later require TCP to reprobe for the available bandwidth. Moreover this approach suffers the overheads listed in Section 2.2.1.

C. Use two TCP connections: one for control, and one persistent data connection. Also use command pipelining on the control connection.

Improvements: A persistent data connection with command pipelining will maintain a steadier flow of data (i.e., higher throughput) over the data connection by letting subsequent transfers utilize the already probed bandwidth.

Drawbacks: This approach suffers from the overheads listed in Section 2.2.1.

D. Use one TCP connection for control, and 'n' parallel data connections

Improvements: Some FTP implementations achieve better throughput using parallel TCP connections for a multiple file transfer.

Drawbacks: This approach is not TCP-friendly [FF99] as it may allow an application to gain an unfair share of bandwidth and adversely affect the network's equilibrium [FF99, BFF96]. Moreover past research has shown that parallel TCP connections may suffer from aggressive congestion control resulting in a reduced throughput [FF99]. As such, this solution should not be considered.

Related Work: Apart from the above solutions, researchers in the past have suggested ways to overcome TCP's limitations in order to boost application performance (e.g. [Bra94], [BRS99]). For example, T/TCP [Bra94] reduced the connection setup/teardown overhead by allowing data to be transferred in the TCP connection setup phase. But due to a fundamental security flaw, T/TCP was removed from operating systems. Objectives (of aggregating transfers) have also been discussed for HTTP over the past years [PM94]. But while HTTP semantics allowed for persistent data connections and command pipelining, FTP semantics do not allow similar solutions without introducing changes to the application (see *A*. above).

Having summarized ways for improving FTP performance while still using TCP, we now consider improving FTP performance by using SCTP, an emerging IETF general purpose transport protocol [SXM+00].

4. SCTP Multistreaming

One of the innovative transport layer services that promises to improve application layer performance is SCTP multistreaming. A stream in an SCTP association is "A uni-directional logical channel established from one to another associated SCTP endpoint, within which all user messages are delivered in sequence except for those submitted to the unordered delivery service" [SXM+00].

Multistreaming within an SCTP association separates flows of logically different data into independent streams. This separation enhances application flexibility by allowing it to identify semantically different flows of data, and having the transport layer "manage" these flows (as the authors argue should be the responsibility of the transport layer, not the application layer). No longer must an application open multiple end-to-end connections to the same host simply to signify different semantic flows.

In Figure 3, Hosts A and B have a multistreamed association. In this example three streams go from A to B, and one stream goes from B to A. The number of streams in each direction is negotiated during SCTP's association establishment phase.

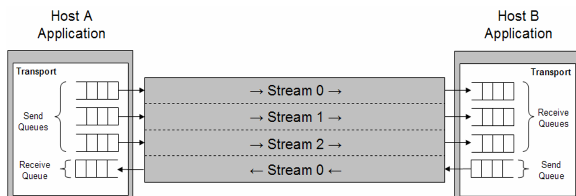


Figure 3: Use of streams within an SCTP association

Each stream has an independent delivery mechanism, thus allowing SCTP to differentiate between data delivery and reliable data transmission and avoid HOL blocking. Similar to TCP, SCTP uses a sequence number to order information. However, TCP sequences bytes, and SCTP sequences PDU's or "chunks". SCTP uses Transmission Sequence Numbers (TSN) for reliable transmission. The TSN is global over all streams. Each stream is uniquely identified by a Stream ID (SID) and has its own Stream Sequence Numbers (SSN). In TCP, when a sender transmits multiple TCP segments, and the first segment is lost, the later segments must wait in the receiver's queue until the first segment is retransmitted and arrives correctly. This HOL blocking delays the delivery of data to the application, which in signaling and some multimedia applications is unacceptable. In SCTP, however, if data on *stream 1* is lost, only *stream 1* may be blocked at the receiver while awaiting retransmissions. With streams being logically independent flows, the data on the remaining streams is deliverable to the application. The socket API extensions for SCTP [SXY+03] provide data structures and socket calls through which application can indicate or determine the stream number on which it intends to send or receive data.

5. FTP over SCTP Variants

In this section we propose three variants of FTP which use SCTP as the transport layer protocol. Each is discussed in turn.

5.1 FTP over SCTP

FTP over SCTP keeps the same semantics as the classic FTP over TCP. Thus, this FTP model uses one separate SCTP association for control, and a new SCTP association for each file transfer, directory listing, or file namelist. The changes to the classic implementation involved only changing the socket call parameters from IPPROTO_TCP to IPPROTO_SCTP in both the client and the server sources.

5.2 FTP over SCTP with multistreaming

In this second model, we use multistreaming to combine the FTP control and data connections in a single SCTP association. Only one SCTP association exists for the entire FTP session. First, an FTP client establishes an SCTP association with the server. During initialization, two streams are opened in each direction. The client and the server send control information (commands and replies) on their respective *stream 0*. Their respective data stream or *stream 1* is used to transfer data (files, directory listings, and file namelists). This approach maintains semantics for streams analogous to the control and data connections in FTP over TCP.

Recall that the data connection in FTP over TCP is non-persistent and the end of data transfer (EOF) is detected by the data connection's close. To detect EOF in our approach, we utilize the SIZE command [EH02]. The SIZE command is already widely used in FTP for the purpose of detecting restart markers. For directory listings, the end of data transfer is detected by using the information (number of bytes read by the *resvmsg*

call) provided to the application by the SCTP socket API [SXY+03].

In the event of a multiple file retrieval issued, the client sends out the request on outgoing *stream 0* and receives the data on incoming *steam 1* for each file in a sequential manner. Figure 4(a) shows the retrieval of multiple files using FTP over multistreamed SCTP. The outgoing stream for all messages and data has been identified. Data on *stream 1* is represented by dashed lines, and control messages on *stream 0* have been represented by solid lines. The dashed box on the timeline in Figure 4(a) indicates the operations that are repeated sequentially for each file to be transferred.

This approach has various advantages, and avoids most of the overheads described in Section 2.2. The number of round trips is reduced as: (1) a single connection (association in SCTP terminology) exists throughout the FTP session, hence repeated setup-teardown of each data connection is avoided, and (2) exchanging PORT commands for data connection information is not needed. The server load is reduced as the server maintains TCBs for at most half of the connections as required with FTP over TCP.

The drawback that this approach faces is similar to the drawbacks described in Section 2.2.2 (B). In the event of a multiple file transfer, each subsequent file transfer will not be able to utilize the prior probed available bandwidth. Before transmitting new data chunks, the sender calculates the cwnd based on the SCTP protocol parameter Max.Burst [SOA+03] as follows:

*if ((flightsize + Max.Burst*MTU) < cwnd)* (1)
*cwnd = flightsize + Max.Burst*MTU*

Since the next file transfer of *file i+1* cannot take place immediately (due to the exchange of control commands before each transfer (see Figure 4a)), all data sent by the server for *file i* gets acked, and reduces the flightsize at the server to zero. Thus in multiple file transfers, the server's cwnd may be reduced to *Max.Burst*MTU* ([SOA+03] recommends the value of the protocol parameter Max.Burst to be set to 4) before starting each subsequent file transfer.

5.3 FTP over SCTP with multistreaming and command pipelining

Finally in this third model we introduce command pipelining in our design from Section 5.2 to avoid unnecessary cwnd reduction for a multiple file transfer. In Section 5.2's model, the cwnd reduction between file transfers occurs because the SIZE and RETR commands for each subsequent file are sent only after the previous file has been received completely by the client.

In Figure 4(b), we present a solution which allows each subsequent transfer to utilize the probed value of congestion window from the prior transfer. Command pipelining ensures a continuous flow of data from the server to client throughout the execution of a multiple file transfer. As seen in Figure 4(b), after parsing the name list of the files, the client sends SIZE commands for all files at once. As soon as a reply for each SIZE command is received, the client sends out the RETR command for that file. Since the control stream is ordered, the replies for the SIZE and RETR commands will arrive in the same sequence as the commands.

By using SCTP multistreaming and pipelining, FTP views multiple file transfers as a single data cycle. Command pipelining aggregates all of the file transfers resulting in better management of the cwnd. This solution overcomes all of the drawbacks listed in Section 2.2, resulting in a more efficient utilization of the bandwidth.

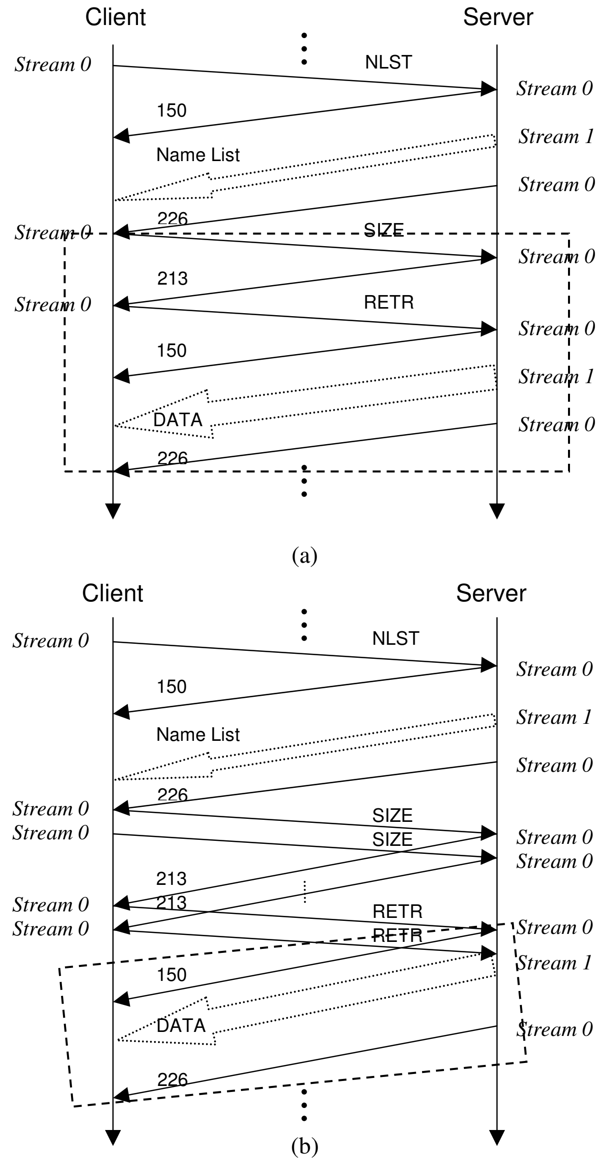


Figure 4: (a) FTP over multistreamed SCTP (b) FTP over multistreamed SCTP with command pipelining

6. Experimental results

We now report on our experimental study of FTP over TCP vs. FTP over SCTP. We focus only on experimental results, however we have also verified our results by simulations using ns version 2.1b8 [NS] and the SCTP patch developed within our Protocol Engineering Lab (PEL) at the University of Delaware. We measured the total transfer time observed for a multiple file transfer for a varied set of parameters.

- *Bandwidth-Propagation Delay (B-D) configuration*: Three path configurations were evaluated: (1Mbps, 35ms), (256Kbps, 125ms), (3Mbps, 1ms). Both the client to server and server to client paths share the same characteristics. In this section, we focus on the results of (1Mbps, 35ms) configuration. Results of the other two configurations have been described in Appendix A.
- *Packet Loss Ratio (PLR)*: The PLRs studied were (0, .01, .03, .06, and .1). Each value represents the loss ratio for both the client to server and the server to client paths experience the same loss rate. We used a uniform probability distribution to emulate packet loss. Certainly 10% loss represents an extreme case but we were interested in general trends as the loss rate increases. Moreover, higher loss rates may be of interest to wireless and military networks.
- *File sizes:* Although FTP is widely used for bulk data transfer, some applications (e.g., source updates) use FTP to transfer small files. To evaluate potential reduced overheads in a variety of these applications, we chose file sizes as (10K, 50K, 200K, 500K, and 1M).

Two sets of experiments were performed with different number of files transferred (10 and 100 files) to observe the effect of total transfer time on the number of files being transferred.

6.1 Experimental setup

We used *Netbed* [WLS+02] (an outgrowth of Emulab) which provides integrated access to experimental networks. Three nodes were used for each experiment, one for the FTP client and one for the FTP server. The third node acted as a router for shaping traffic between the client and server. The client and server nodes are 850MHz Intel Pentium III processors, and based on the Intel ISP1100 1U server platform. All three nodes run FreeBSD-4.6. The FreeBSD kernel implementation of SCTP available with the KAME Stack [KAME] was used on the client and server nodes. KAME is an evolving and experimental stack mainly targeted for IPv6/IPsec in BSD based operating systems. An updated snapshot of the stack (KAME snap kit) is released every week. We used the snap kit of 14th October, 2002. The router node runs *Dummynet* [Riz97] which simulates a drop tail router with a queue size of 50 packets, and specified bandwidth, propagation delay and packet loss ratio.

We implemented protocol changes by modifying the FTP client and server source code available with the FreeBSD 4.6 distribution. In our experiments, total transfer was measured using packet level traces as follows. The starting time was taken as the time the client sends out the first packet to the server following the user's *"mget"* command. The end time was the time the "226 control reply" from the server reached the client after the last file transfer. Each combination of parameters (3 B-D configurations x 5 PLR x 5 file sizes) was run multiple times to achieve a 90% confidence level for the total transfer time. *Tcpdump* [TCPDUMP] (version 3.7.1) was used to perform packet level traces. SCTP decoding functionality in *tcpdump* was developed in collaboration of UD's Protocol Engineering Lab and Temple University's Netlab. Our results compare four FTP variants:

(1) FTP over TCP: The current FTP protocol which uses a separate TCP connection for control, and a new TCP data connection for every file transfer, directory listing and name list. The TCP variant used was New-Reno.

(2) FTP over SCTP: The original FTP protocol design but using SCTP at the transport. See Section 5.1.

(3) FTP over multistreamed SCTP: This design, described in Section 5.2, uses a single SCTP association for both control and data. It uses multistreaming to assign one stream to control, and one stream to data. The SCTP association between the client and the server persists throughout the FTP session.

(4) FTP over multistreamed SCTP with command pipelining: Described in Section 5.3, this design adds command pipelining to FTP over multistreamed SCTP to ensure that the congestion window is not needlessly probed for each file transfer.

We have performed experiments involving single as well as multiple file transfer. Although the improvement of file transfers using SCTP multistreaming is also witnessed in single file transfers, we emphasize the results of experiments involving multiple file transfer for two reasons. First, the positive impact of multistreaming is more predominant in the event of multiple file transfers. Second, comparing variant (1) vs. variant (2) provides insight on single file transfer.

6.2 Results

Figure 5 shows the results obtained for (1Mbps, 35ms) bandwidth-delay configuration. Each graph represents the loss probabilities vs. total transfer time to retrieve 10 files (each the same size) using four different FTP variants. Figure 6 shows the same comparisons but with retrieval of 100 files.

6.2.1 Comparing (1) vs. (2)

Since variant (2) is simply a straightforward substitution of TCP calls with SCTP calls, any difference in performance must be attributed to SCTP's handling of data (i.e., congestion control, loss recovery) and not to its feature of multistreaming. Figure 5 shows that for small file transfers (see Figure 5(a) and 5(b)) (1) and (2) overall perform similarly. (2) performs worse than (1) at low loss rates (~ 0-3%) due to the fact that the per packet payload being carried by SCTP (1408 bytes) is less than TCP (1448 bytes) thus making the overhead associated with SCTP slightly more than TCP. (At the time experiments were performed, the SCTP fragmentation threshold for the FreeBSD implementation was 1408. This threshold has been increased recently thus reducing its effect on per packet overhead.) As the packet loss rate increases, (2) begins outperforming (1). We believe this reversal is due to SCTP's more robust loss recovery and congestion control mechanisms which outbalance the effects of per packet overheads. Details on the differences of congestion control mechanisms between SCTP and TCP can be found in [AAI02].

For small file transfers, the data connection in both (1) and (2) is dominated by the slow start phase of the congestion control. As the file size increases (see Figures 5 and 6: (c), (d) and (e)), the data connection's life time in both (1) and (2) is dominated by the congestion avoidance phase. Hence as the file size increases, both the scale and ratio of performance benefit seen by (2) as compared to (1) at loss rates (1-10%) increases. For example, at

3% loss rate the ratio of total transfer time taken by (1) to (2) is 0.92, 1.14, 1.29, 1.31, 1.56 for ten 10K, 50K, 200K, 500K and 1M multiple file transfers, respectively. This steady increase results because as the number of loss events generated increases proportionally with the size of file transfers, SCTP takes advantage over TCP on a per loss event basis eventually reducing latency by nearly or more than 50%. This improvement can be seen in Figure 6(e) at a 3% loss, (1) requires 2210 seconds to transfer 100 1M-files whereas (2) requires 1409 seconds.

As can be seen from Figures 5 and 6, as the number of file transfers increase from 10 to 100, the scale of performance of (2) as compared to (1) also increases. As the loss rate increases, more significant performance improvements can be seen. SCTP's significant outperformance of TCP at medium to high loss rate came as a surprise as it was widely understood that the congestion control mechanisms in TCP and SCTP are approximately the same. We have validated our results using simulations, and are currently investigating the effect of the subtle differences between the congestion control mechanisms in TCP and SCTP, which result in such significant difference observed in overall steady state performance (e.g., SCTP's congestion control semantics incorporate Limited Transmit [ABF01], Appropriate Byte Counting [All03], while the TCP implementation that is currently prevalent (and the TCP implementation used in our experiments) does not use such features.). The congestion control mechanisms in TCP are in the process of being fine tuned, a research task underway in the IETF [e.g., ABF01, All03]. Once the TCP extensions are included in TCP implementations, we expect (1) and (2) to perform similarly at different loss rates.

6.2.2 Comparing (3) and (4) vs. (2)

We now turn our discussion to the multistreamed FTP variants (3) and (4). We compare (3) and (4) with (2) and not with (1) because our main focus is to evaluate the effect of SCTP multistreaming and command pipelining on multiple file transfers.

As noted in Sections 2 and 5, using multistreaming and command pipelining (a) reduces the number of round trips in command exchanges and connection setup-teardown, and (b) maintains the probed value of the congestion window for subsequent transfers in a multiple file transfer. We hypothesized that the effect of (a) would remain fairly constant irrespective of the file sizes being transferred, and the effect of (b) would be more evident in transfer of small files and less in large files. For small files, non-persistent data connections would tend to remain in the slow start phase probing for available bandwidth, whereas the time spent in probing for available bandwidth for large file transfers would be relatively small as compared to the time spent in steady state congestion avoidance. However, we expected that the effects of both (a) and (b) would be directly proportional to the number of files being transferred.

In (3) we reduce the number of round trips but do not maintain the probed congestion window for subsequent transfers (see Section 5.2). As noted above this effect should have a constant scale as compared to (2). We can see from Figure 5 that the ratio of transfer time taken by (2) vs. (3) remains fairly constant ranging between 1.5 and 1.7. The small variance can be attributed to the losses (which result in timeouts) incurred by the extra round trips involved in (2). As noted above, the most significant impact of (4) as compared to (2) comes for short transfers. For example in Figure 6(a), at a 3% loss scenario (2) requires 103.3 seconds to transfer 100 files of size 10K each, as compared to (4) which takes only 19.8 seconds. From Figure 5, at 3% loss rate the ratio of total transfer time taken by (2) to (4) is 4.9, 4.1, 3.5, 3.1, and 2.1 for ten 10K, 50K, 200K, 500K and 1M file transfers, respectively. Thus this effect, which is also seen by comparing the ratio of (3) vs. (4), demarcates the benefits that multistreaming and command pipelining provide.

Moreover, it can be seen from Figures 5 and 6 that as the number of files to be transferred increase from 10 to 100, the performance gain by (4) as compared to (2) increases. This increase implies significant benefits to mirroring applications that use FTP (e.g., *fmirror*) which often have to mirror a large number of files from one server to the other.

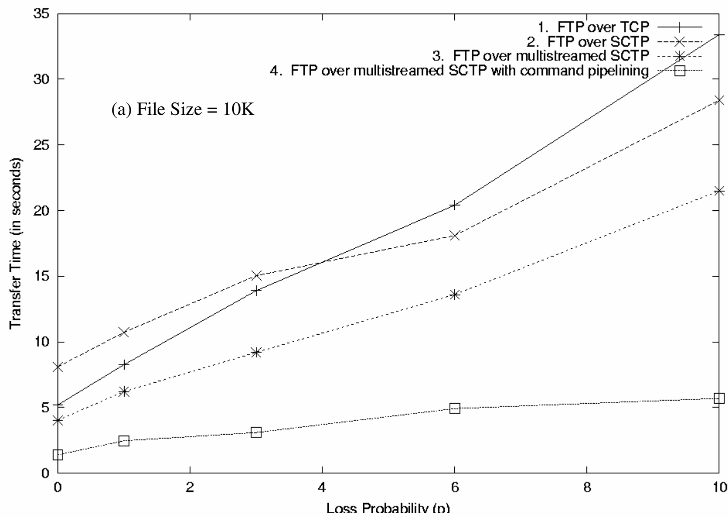
We would like to note that comparing (1) which is FTP over TCP-New Reno (the variant prevalent in the Internet) to (4) shows the tremendous impact that SCTP, multistreaming and command pipelining can have in FTP transfer time. From Figure 6(e), (1) takes 2210 seconds as compared to (4) which takes 948 seconds to transfer 100 1M-files at 3% loss. Also to note is that (3) and (4) perform consistently better as compared to either (1) or (2) irrespective of the loss rates.

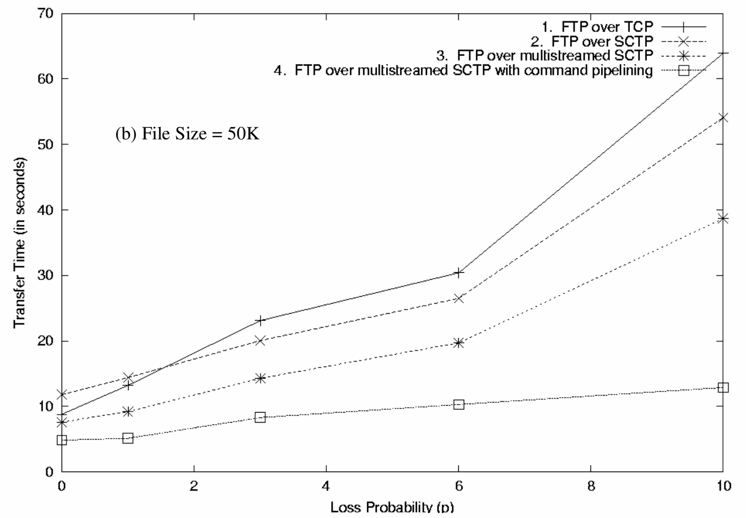
[The results of other bandwidth-delay configurations are included in Appendix A. Due to page limitations, they will not be included in the final paper if accepted.]

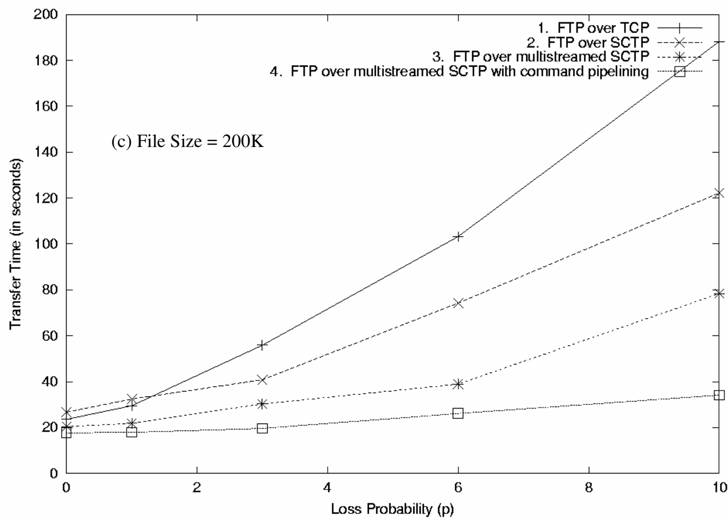
6.2.3 Summary

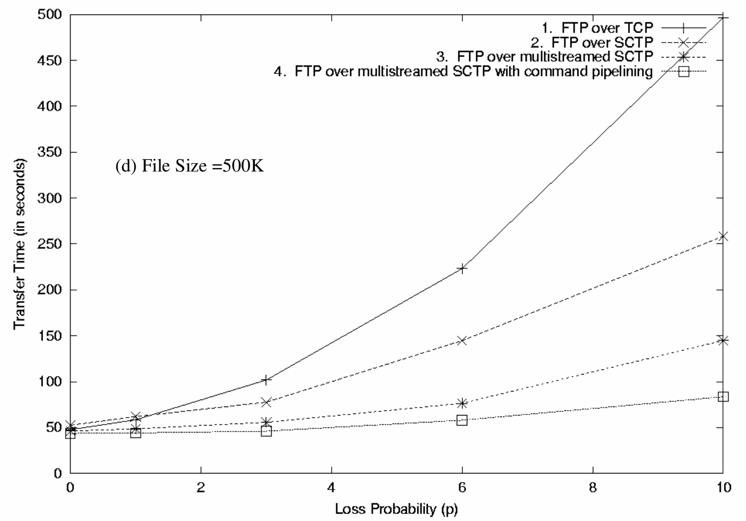
To summarize the results of our experiments:

- It is evident from the experimental results that (2) performs close to (1) at lower loss rates, and as the loss rate increases, (2) outperforms (1) significantly. For smaller loss rates, per packet overhead in (2) results in marginally lower performance as compared to (1). (This factor does not play into the latest implementation of SCTP.) Past research has shown that the congestion control semantics and loss recovery mechanisms in SCTP are robust as compared to TCP, which result in better steady state throughput at higher loss rates [AAI02].
- Exploiting SCTP multistreaming (in (3)) performs better by a steady scale factor of approximately 1.5 (in relation to file sizes) as compared to FTP over SCTP without multistreaming (in (2)). This gain can be attributed to the fact that multistreaming helps in reducing a constant number of round trips directly proportional to the number of files being transferred. The slight variance witnessed is due to the loss (and eventually timeouts) that these extra round trips can incur.
- Adding command pipelining to multistreaming in (4) further reduces total transfer time for a multiple file transfer. The effect of command pipelining is more predominant in small transfers due to the fact that short flows spend most of the time probing for the available bandwidth.
- The absolute scale of transfer time improvement in FTP over multistreamed SCTP with/without command pipelining is directly proportional to the number of files being transferred in a multiple file transfer request: more files transferred results in more relative savings in transfer time.









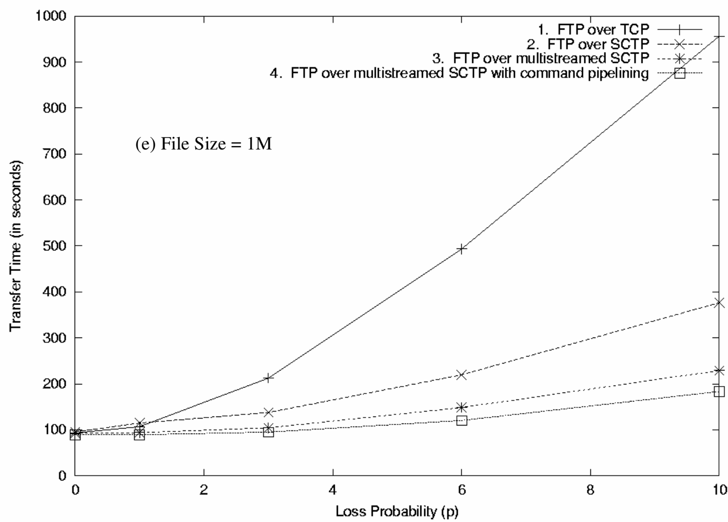
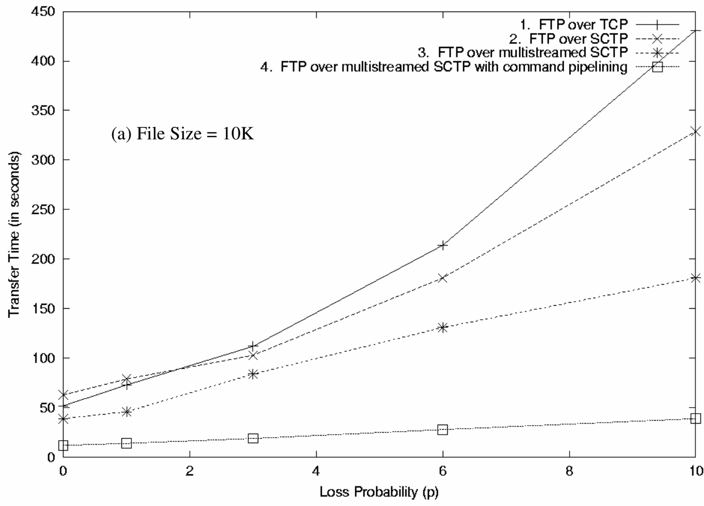
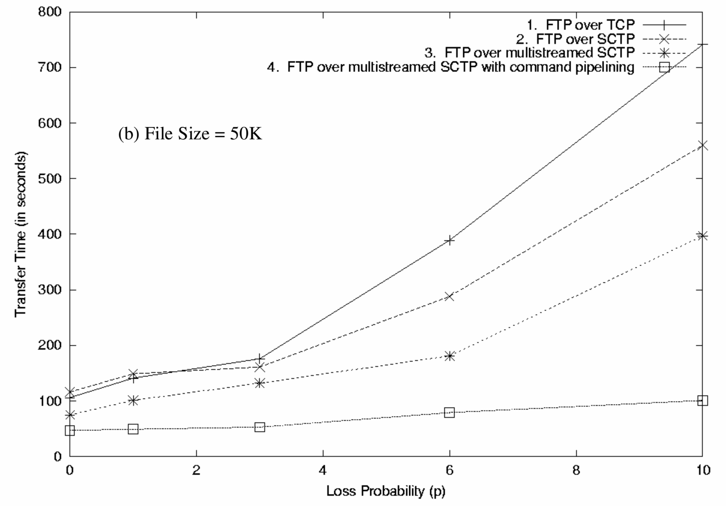
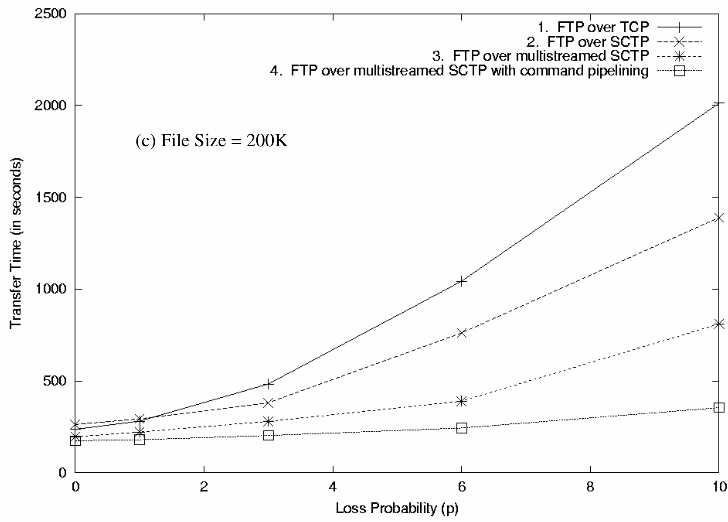
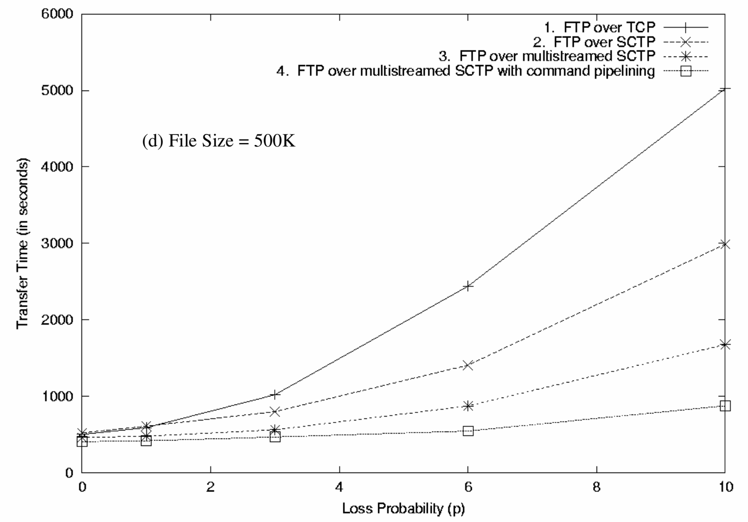


Figure 5: Transfer Time vs. Loss Probability for a multiple transfer of *10 files* (Bandwidth = 1Mbps Propagation Delay = 35ms)









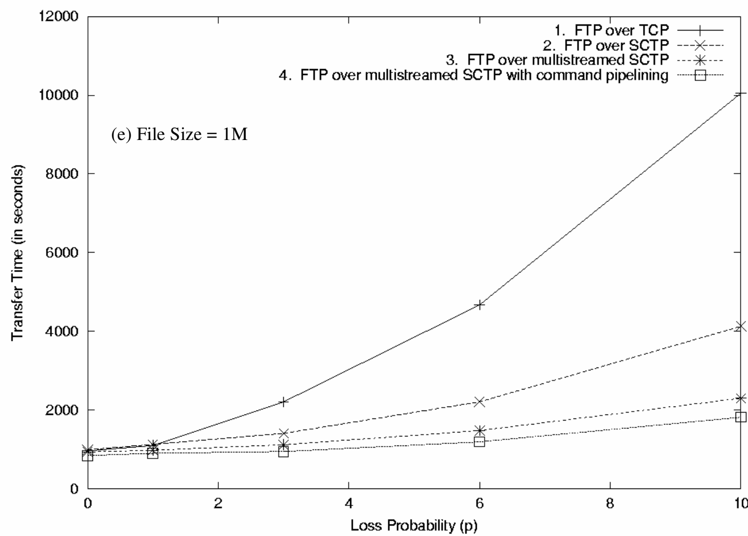


Figure 6: Transfer Time vs. Loss Probability for a multiple transfer of *100 files* (Bandwidth = 1Mbps Propagation Delay = 35ms)

7. Conclusions

Our experimental results confirm that modifying FTP to use SCTP multistreaming and command pipelining dramatically reduces latency of multiple file transfers. These features:

- reduce the number of connections by aggregating the control and data connections,
- reduce the number of round trips required for connection setup/teardown, and command exchange, and
- use the bandwidth more efficiently by preserving the congestion window between file transfers.

Apart from transfer time improvements, other advantages achieved by running FTP over SCTP (with multistreaming and/or command pipelining) instead of over TCP are:

- The number of connections a server must maintain is reduced. Quantifying server load and its effects on throughput is beyond the scope of this paper. The interested reader is pointed to [FTY99]. We however expect that by using either modification (3) or (4), servers will be able to serve at least twice the number of clients as compared to the current FTP over TCP design (assuming that the bottleneck for the number of simultaneous clients served is the number of TCBs reserved for the connections). This consideration may be of interest to busy servers who are constrained by the number of clients that can be served simultaneously.
- The number of packets exchanged between the client and the server is reduced, thus reducing the overall network load.
- Aggregating control and data connections into one SCTP multistreamed association solves concerns that current FTP protocol faces with Network Address Translators (NAT) and firewalls in transferring IP addresses and port numbers through the control connection [AOM98, Tou02].

The authors further argue that the benefits of SCTP's multistreaming can be exploited by other applications. SCTP's multistreaming provides a TCP-friendly mechanism for parallel transfers. Ongoing research at UD's PEL is investigating whether web transfers using HTTP can benefit from aggregation of multiple transfers in a single SCTP association.

Two limitations of this work which we plan to address in the future:

- We have used a uniform loss distribution model for emulating losses on the path. We are investigating a variation of *Dummynet* which can model burst losses.
- Recent additions to the TCP congestion control [e.g. ABF01, AF99] attempt to fine tune TCP's behavior to result in faster recovery from loss events, and fewer timeouts. An extension to our work could be to take such TCP fine tunings into consideration.

Acknowledgements

This paper significantly benefited from discussions with Janardhan Iyengar and Armando Caro. We thank Randall Stewart for providing support for the KAME stack implementation of SCTP. We thank Jay Lepreau and the support staff of Netbed (formerly known as Emulab), the Utah Network Emulation Testbed (which is primarily supported by NSF grant ANI-00-82493 and Cisco Systems) for making their facilities available for our experiments. A special thanks to Mike Hibler for helping set up nodes on Netbed. Finally, we thank the members of the Protocol Engineering Lab for helpful comments on an earlier draft of this paper.

References

[AAI02] R. Alamgir, M. Atiquzzaman, W. Ivancic, *Effect of Congestion Control on the Performance of TCP and SCTP over Satellite Networks*. Proc. NASA Earth Science Technology Conference, June 2002. Pasadena, CA.

[ABF01] M. Allman, H. Balakrishnan, S. Floyd, *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042, January 2001.

[AF99] M. Allman, A. Falk, *On the Effective Evaluation of TCP*. ACM Computer Communication Review, 29(5), October 1999.

[All03] M. Allman, *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465, February 2003.

[AO97] M. Allman, S. Ostermann, *Multiple Data Connection FTP Extensions*. Technical Report TR-19971, Ohio University Computer Science, February 1997.

[AOM98] M. Allman, S. Ostermann, C. Metz, *FTP extensions for NATS and firewalls*. RFC 2428, September 1998.

[APS99] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control. RFC 2581*, April 1999.

[Bra94] R. Braden, *T/TCP - TCP extensions for transactions functional specification*. RFC 1644, July 1994.

[BRS99] H. Balakrishnan, H. Rahul, S. Seshan, *An Integrated Congestion Management Architecture for Internet Hosts*. Proceedings SIGCOMM, September 1999.

[Bel94] S. Bellovin, *Firewall-Friendly FTP*. RFC 1579, February 1994.

[BFF96] T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*. RFC 1945, IETF, May 1996.

[BPS+98] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, R. Katz, *TCP Behavior of a Busy Internet Server: Analysis and Improvements*. Proc. IEEE Infocom, March 1998. San Francisco, CA.

[EH02] R. Elz, P. Hethmon, *Extensions to FTP*. draft-ietf-ftpext-mlst-16.txt, IETF Internet draft (work in progress), September 2002.

[FF99] S. Floyd, K. Fall, *Promoting the Use of End-to-End Congestion Control in the Internet*. IEEE/ACM Transactions on Networking, August 1999.

[FH99] S. Floyd, T. Henderson, *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 2582, April 1999.

[FTY99] T. Faber, J. Touch, W. Yue, *The TIME-WAIT State in TCP and Its Effect on Busy Servers*. Proceedings Infocom, March 1999. New York City, NY.

[HL97] M. Horowitz, S. Lunt, *FTP Security Extensions*. RFC 2228, October 1997.

[KAME] KAME Project, www.kame.net

[Kin00] J. King, *Parallel FTP Performance in a High-Bandwidth, High-Latency WAN*, SC2000, November 2000.

[MC00] S. McCreary, K. Clay, *Trends in Wide Area IP Traffic Patterns - A View from Ames Internet Exchange*. Proc. ITC, September 2000. Monterey, CA.

[NS] UC Berkeley, LBL, USC/ISI, and Xerox Parc. Ns-2 documentation and software, version 2.1b8. http://www.isi.edu/nsnam/ns.

[PM94] V. Padmanabhan, J. Mogul, *Improving HTTP latency*. Proc. 2nd International World Wide Web Conference, October 1994. Chicago, IL.

[PR85] J. Postel, J. Reynolds, *File Transfer Protocol (FTP)*. RFC 959, October 1985.

[Riz97] L. Rizzo, *Dummynet: a simple approach to the evaluation of network protocols*. ACM Computer Communication Review, 27(1):3141, January 1997.

[SOA+03] R. Stewart, L. Ong, I. Arias-Rodriguez, K. Poon, P. Conrad, A. Caro, M. Tuexen, *Stream Control Transmission Protocol (SCTP) Implementers Guide*.draft-ietf-tsvwg-sctpimpguide-08.txt, IETF Internet draft (work in progress), February 2003.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, *Stream Control Transmission Protocol*. RFC 2960, October 2000.

[SXY+03] R. Stewart, Q. Xie, L. Yarroll, J. Wood, K. Poon,, K. Fujita, M. Tuexen, *Sockets API Extensions for Stream Control Transmission Protocol (SCTP).* draft-ietf-tsvwg-sctpsocket-06.txt, IETF Internet draft (work in progress), February 2003.

[TCPDUMP] TCPDUMP public repository, http://www.tcpdump.org

[Tou02] J. Touch, *Those Pesky NATs*, IEEE Internet Computing, pp. 96, July/August 2002.

[WLS+02] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar. *An Integrated Experimental Environment for Distributed Systems and Networks*. Proc. 5th Symposium on Operating Systems Design and Implementation, December 2002. Boston, MA.

[WUARCHIVE] Usage Statistics for wuarchive, http://wuarchive.wustl.edu

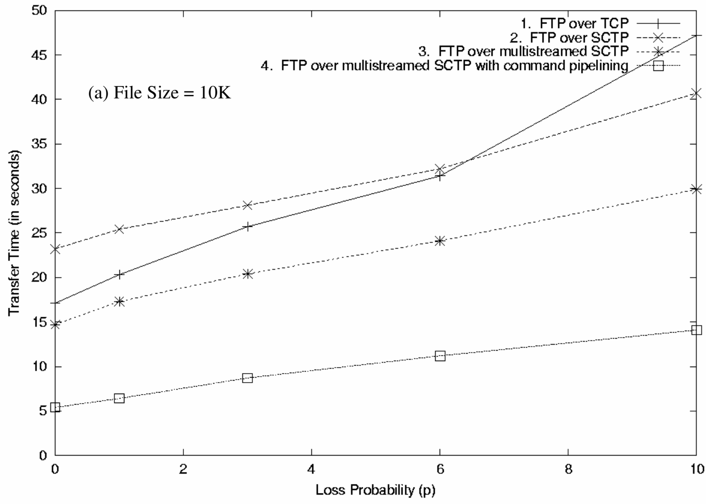
Appendix A

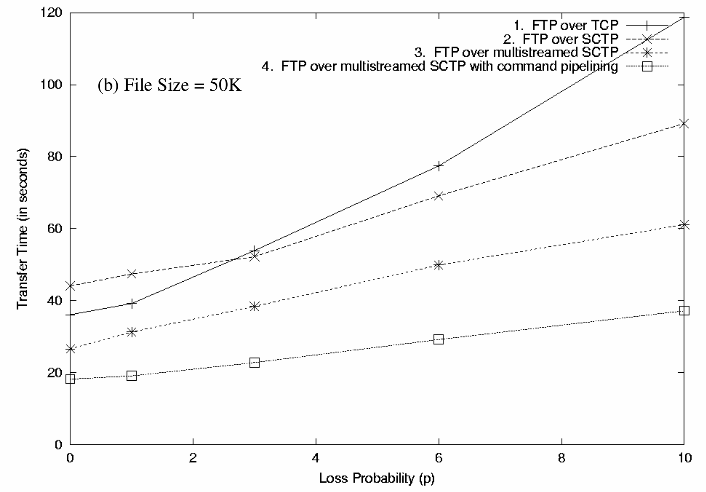
Figures 7 and 8 present results comparing the four FTP variants discussed earlier in this paper, in different bandwidth-delay configurations. The (256Kbps, 125ms) configurations represents an emulated satellite channel; whereas the (3Mbps, 1ms) configuration can be thought of as representing a Local Area Network (LAN) connectivity. The experimental setup used is the same as described in Section 6.1. The number of files transferred using a multiple file transfer is 10.

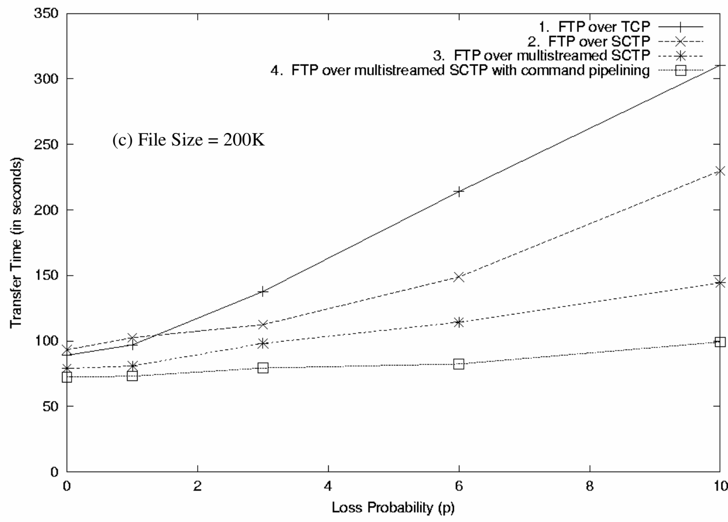
It can be seen from Figures 7 and 8 that the relative scale of improvement of (3) or (4) as compared to (1) remains fairly similar for different bandwidth-delay configurations. However, since the path with the higher *effective* RTT will result in lower throughput, the absolute difference in transfer time taken by (3) or (4) as compared to (1) will be large for such a configuration. This effect can be seen by comparing the total transfer time taken in corresponding graphs in Figures 6 and 7.

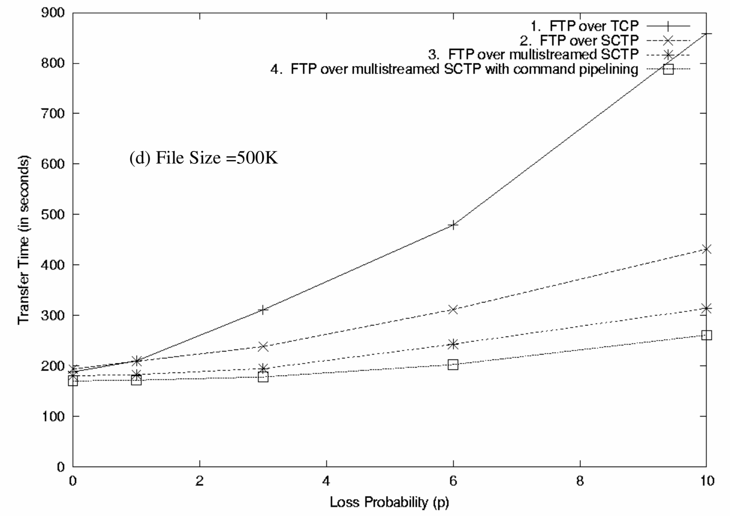
As seen earlier, (1) performs slightly better than (2) at low loss rates (0-3%), however (2) outperforms (1) significantly as the loss rate increases and as the size of the files being transferred increases. The outperformance of (2) can be again attributed to the better congestion control semantics in SCTP as compared to TCP.

Experimenting with different bandwidth-delay configurations, results in similar conclusions about the relation of file sizes and impact of multistreaming and command pipelining. Multiple file transfer of smaller file sizes (10K, 50K) using (3) or (4) results in significant relative improvements in throughput. As the loss rate increases, total transfer time taken by (4) increases much slowly as compared to (1) or (2). This robustness to loss can be derived from the congestion control principles in SCTP: since (4) aggregates all the files into a bulk data transfer (thus keeping the window fairly high), the number of losses detected by timeouts in (4) will be relatively very low as compared to the number of losses detected by four missing reports. However, in (1) and (2), the connection may spend substantial time in slow start (thus having smaller windows), and hence depend on timeouts for loss recovery.









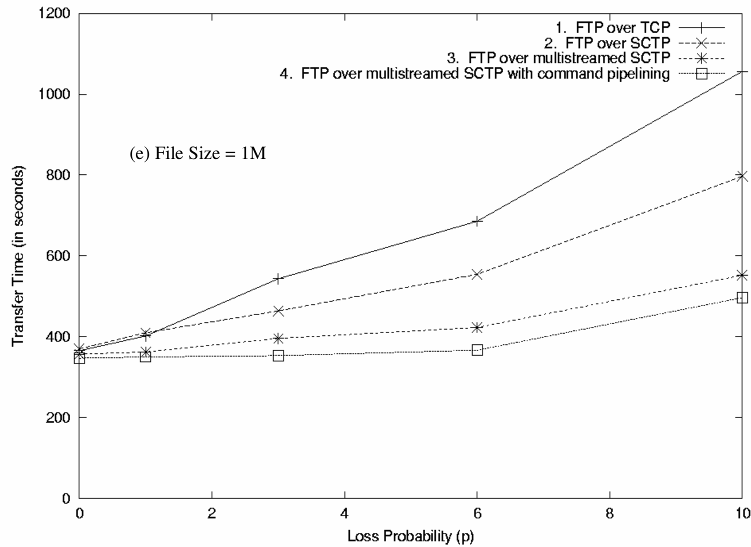
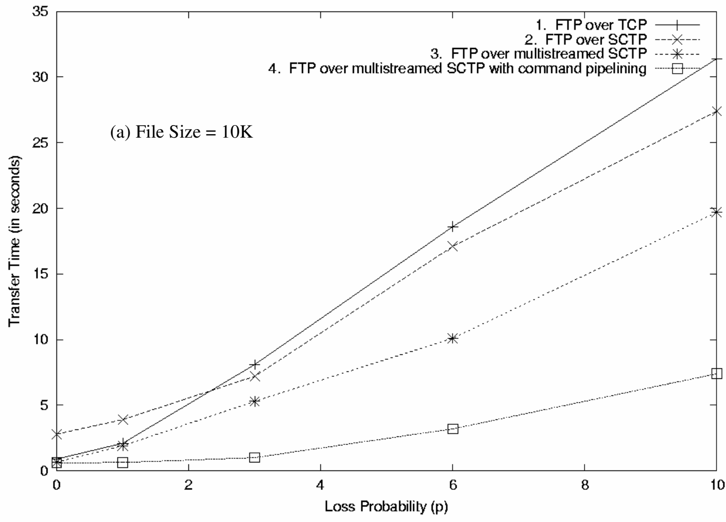
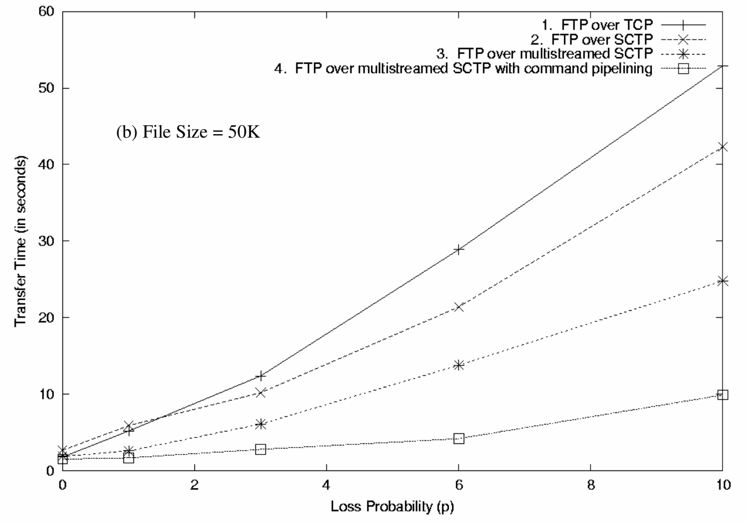
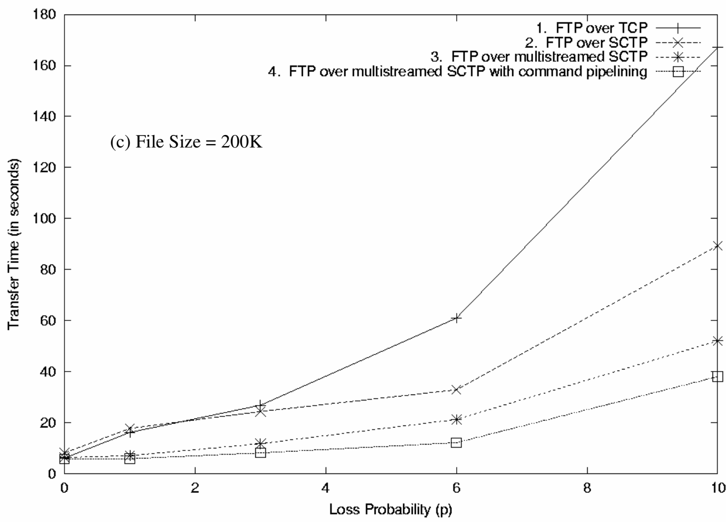
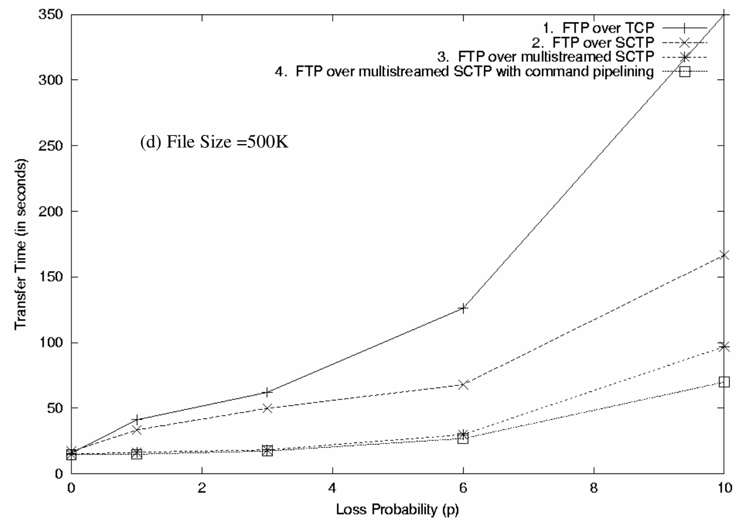


Figure 7: Transfer Time vs. Loss Probability for a multiple transfer of *10 files* (Bandwidth = 256Kbps Propagation Delay = 125ms)









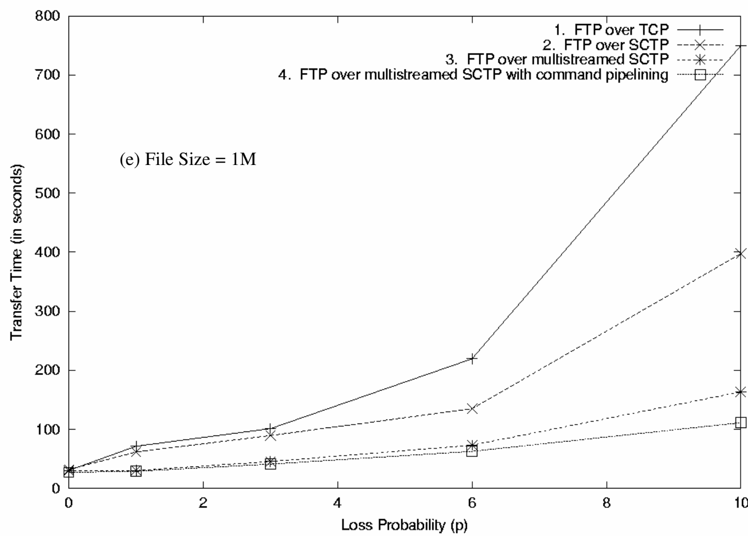


Figure 8: Transfer Time vs. Loss Probability for a multiple transfer of *10 files* (Bandwidth = 3Mbps Propagation Delay = 1ms)