International Journal of Modern Engineering Research (IJMER)
www.ijmer.com Vol. 3, Issue. 6, Nov - Dec. 2013 pp-3403-3415 ISSN: 2249-6645

Implementation of Multiple FTP Application using SCTP Multistreaming

Srikanth Porika[1], B.Rajani [2], T.Bharath Manohar[3]
[1] *M.Tech, Dept of CSE,CMR COLLEGE OF ENGINEERING & TECHNOLOGY (Affiliated to JNTU Hyderabad),Hyderabad,Andhra Pradesh.India.*
[2] *Asst. Prof.,Dept of CSE , CMR COLLEGE, OF ENGINEERING & TECHNOLOGY (Affiliated to JNTU Hyderabad),Hyderabad,Andhra Pradesh.India.*
[3] *M.Tech, Dept of CSE,CMR COLLEGE OF ENGINEERING & TECHNOLOGY, (Affiliated to JNTU Hyderabad),Hyderabad.Andhra Pradesh.India.*

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 over TCP approach without introducing complexity at the application, and 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 command pipelining. Results comparing these 3 variations with the classic FTP over TCP indicate significant improvements in throughput for the transfer of multiple files by using multistreaming and command pipelining, with the largest benefit occurring for transferring multiple short files. More generally, this paper encourages the use of SCTP's innovative transport-layer services to improve existing and future application performance.*
Keywords: *SCTP, FTP Application,Multi-Streaming .*

I. INTRODUCTION

The past decade has witnessed an exponential growth of Internet traffic, 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.

Over the years, several FTP extensions have been proposed [AOM98, EH02, HL97], with a few efforts to improve performance by using parallel TCP connections [AO97, Kin00]. However, opening parallel TCP connections (whether for FTP or HTTP) is regarded as "TCP-unfriendly" [FF99] as it allows an application to gain an unfair share of bandwidth at the expense of other network flows, potentially sacrificing network stability. Our focus is to improve 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 Internet's software mirroring sites, for various source code repositories, for system backups, and for file sharing. All of these applications require transferring multiple files from one host to another.

In this paper we identify the overheads associated with the current FTP design mainly due to running over TCP, which constrains the FTP application. We present modifications to FTP to run over Stream Control Transmission Protocol (SCTP) RFC2960 [SXM+00] 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, in particular, 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 command pipelining and SCTP multistreaming benefit FTP in reducing overhead, especially for multiple file transfers. We recommend two modifications to FTP that 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 with lower transfer time and higher throughput for multiple file transfers particularly under lossy network conditions. Moreover, our modifications to FTP solve concerns current FTP protocol faces with NATs and firewalls in transferring IP addresses and port numbers in the payload data [AOM98, Bel94, Tou02].

This paper is organized as follows. Section 2 details and quantifies the overheads in the current FTP over TCP design. This section also discusses possible solutions to eliminate these overheads while still using TCP as the transport. Section 3 introduces SCTP multistreaming. Section 4 presents minor FTP changes needed to exploit SCTP multistreaming, and a description of how the new design reduces overhead. Section 5 presents the experimental results, and Section 6 concludes the paper.

II. MOTIVATION

2. Inefficiencies and possible solutions

2.1 Inefficiencies in FTP over TCP

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.

2.1.1 Distinct control and data connection

A. FTP's out-of-band control signaling approach has consequences in terms of end-to-end latency. In a multiple file transfer, traffic on the control connection tends to be send-and-wait, with no traffic transferred during the file transfer. This connection's congestion mechanism typically times out, and returns the connection to slow start for each new file to be transferred [APS99]. The control connection is particularly vulnerable to timeouts because too few packets are flowing to cause a TCP fast retransmit. 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. Attempts are needed to reduce the command exchange over the control connection.

B. With distinct connections, 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 impact busy servers that are subject to reduced throughput due to memory block lookups [FTY99]. TCB overheads may be reduced by using ensemble sharing [BS01, Tou97].

C. Over the past years, considerable discussion has taken place on FTP's lack of security, often 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 NATs and firewalls that must monitor and translate addressing information [AOM98, Tou02].

2.1.2 Non-persistence of the data connection

A.The non-persistence of a data connection for multiple files causes connection setup overhead at least on the order of 1 RTT for each file transfer or directory listing. (Traffic overhead also exists for connection teardown, but this traffic overlaps the control commands for the next operation.)

B.Every new data connection must initially probe the available bandwidth (via a congestion window (cwnd)) during a slow start phase, before the connection reaches its steady state cwnd. A loss early in the slow start phase, before the cwnd contains enough packets to allow for fast retransmit, will result in a timeout at the server. Figure 1 graphically shows the nature of this re-probing overhead in the event of three consecutive file transfers (over three different TCP connections). The interval between the transfers indicates the time involved in terminating the previous connection, transferring control commands, and setting up a new connection. (Note: Figure 1 represents a generic example.)

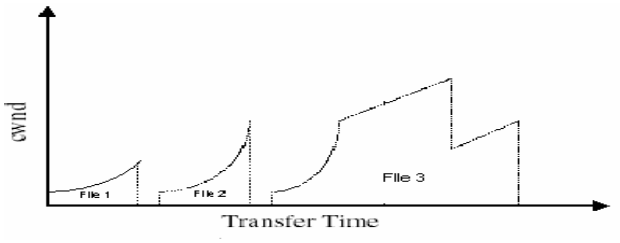


Figure 1: Expected cwnd evolution during a multiple file transfer in FTP over TCP

For each file transfer, at least one RTT overhead is incurred over the control connection for communicating the PORT command and its 200 reply.

D.In the event of multiple small file transfers, the server ends up having many connections in the TCP TIME-WAIT state and hence must 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].

2.2. 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 one persistent TCP connection for 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 transmitted over the same connection. Using a single TCP connection requires the application to use markers to differentiate between control and data, and the beginning/end of each file. 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 (discussed in Section 3).

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

Improvements: A persistent data connection eliminates the connection setup-teardown and command exchange overheads for every file transfer, thus reducing network traffic and the number of round trip delays.

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 little as the initial default size, and later require TCP to re-probe for the available bandwidth [HPF00]. Moreover this approach still suffers from the overhead listed in Section 2.1.1.

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

Improvements: Command pipelining allows for the immediate request of multiple files over the control connection rather than requiring $file_i$ is completely retrieved before $file_{i+1}$ is requested. 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 still suffers from the overhead listed in Section 2.1.1.

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

Improvements: Some FTP implementations do 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 [BFF96, FF99]. 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. This approach also suffers the overheads listed in Section 2.1.1.

Related Work: Apart from the above solutions, researchers in the past have suggested ways to overcome TCP's limitations and boost application performance [BS01, Tou97]. 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 could not succeed. Aggregating transfers has also been discussed for HTTP [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 the main objective of this paper - improving FTP performance by using SCTP, an emerging IETF general-purpose transport protocol [SXM+00]. We note that the TCP alternatives that incorporate temporal and ensemble sharing [Bra94, BS01, Tou97] are not discussed further in this paper; future work should evaluate such alternatives.

III. LITERATURE SURVEY

3. SCTP multistreaming

One innovative transport layer service 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 have 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.

Figure 2 shows Hosts A and B connected with a single multistreamed association. The number of streams in each direction is negotiated during SCTP's association establishment phase. In this example, three streams go from A to B, and one stream goes from B to A.

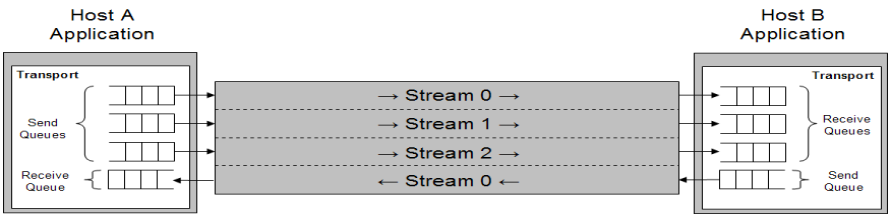


Figure 2: Use of streams within an SCTP association

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 and achieve reliability. However, where TCP sequences bytes, SCTP sequences transport layer protocol data units (PDUs) or "chunks" using Transmission Sequence Numbers (TSN). The TSN number space 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* can be blocked at the receiver while awaiting retransmissions. The logically independent data flows on remaining streams can be deliverable to the application. SCTP's socket API extensions [SXY+03] provide data structures and socket calls through which an application can indicate or determine the stream number on which it sends or receives data.

IV. SYSTEM ANALYSIS & DESIGN

4. FTP over SCTP variants

We consider three variations of FTP over SCTP to help identify the various gains of different features. Each is described in turn.

4.1 FTP over SCTP (SCTP-Naïve)

Our first variation named "SCTP-naïve" maintains the semantics of FTP over TCP. We name this approach "naïve" because it naïvely uses one persistent SCTP association for control, and a new non-persistent SCTP association is opened, used, and closed for each file transfer, directory listing, or file namelist, as is done in the current FTP over TCP approach. SCTP-naïve does not exploit any of SCTP's advantages; it is evaluated to measure the inherent performance differences between our TCP and SCTP implementations. If the basic TCP and SCTP implementations were the same, then the performance should be similar. The SCTP-naïve approach is not recommended in practice.

To derive SCTP-naïve, all socket calls in both the client and server in the FTP over TCP version (herein "TCP") were changed from using IPPROTO_TCP to IPPROTO_SCTP. The timing is shown in Figure 3 with solid lines representing PDUs traveling over the control association, and dotted lines representing PDUs traveling over new associations. The large dashed box represents the sequence of PDUs that must be iteratively transmitted for each file of the multiple file transfer.

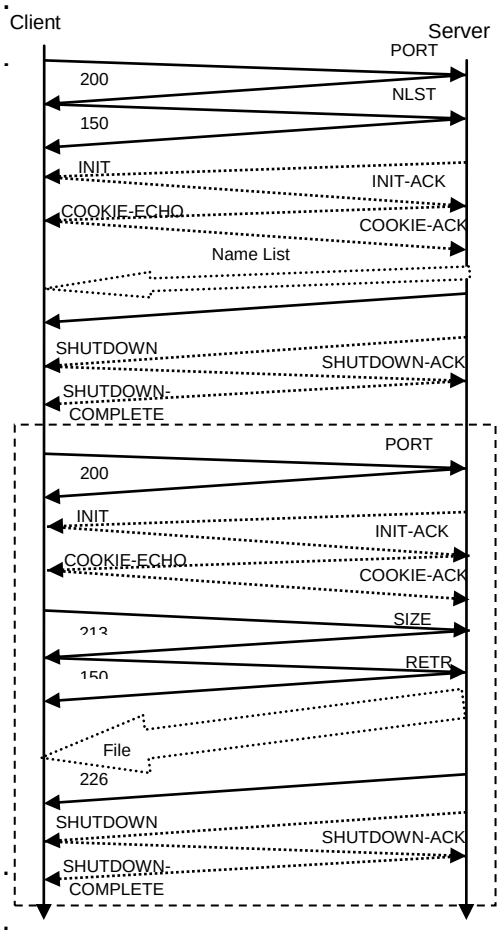


Figure 3: FTP over SCTP-Naïve

4.2 FTP over SCTP with multistreaming (SCTP-MS)

In "SCTP-MS", FTP control and data connections are combined over a single multistreamed SCTP association. That is, only one association exists for the entire multiple file FTP session. An FTP client establishes an SCTP association with the server with two streams opened in each direction. The client and the server send control information (commands and replies) on their respective *stream 0*. All data (files, directory listings, and file namelists) are transferred over their respective *stream 1*. This approach maintains semantics for streams analogous to 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 using one SCTP association, the SIZE command [EH02] is used. 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 number of bytes read by *recvmsg* call provided by the SCTP socket API [SXY+03].

For a multiple file retrieval, the client sends out requests on outgoing *stream 0* and receives the files sequentially on incoming *steam 1* (see Figure 4). Data on *stream 1* is represented by dashed lines, and control messages on *stream 0* are represented by solid lines. The dashed box on the timeline in Figure 4 indicates the operations that are repeated sequentially for each file to be transferred.

This approach avoids most of the overheads described in Section 2.1. 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 eliminated, and (2) exchanging PORT commands over the control connection for data connection information is unnecessary. The server load is reduced as the server maintains TCBs for at most half the connections required with FTP over TCP.

The drawback that this approach faces is similar to the drawbacks described in Section 2.1.2.B. For a multiple file transfer, each subsequent file transfer is unable 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:

$$\textit{if } ((\textit{flightsize} + \textit{Max.Burst*MTU}) < \textit{cwnd}) \quad (1)$$
$$\textit{cwnd} = \textit{flightsize} + \textit{Max.Burst*MTU}$$

Since the transfer of $file_{i+1}$ cannot take place immediately (due to the exchange of control commands before each transfer (see Figure 4)), all data sent by the server for $file_i$ gets acked, and the flightsize at the server reduces to zero. Thus in multiple file transfers, the server's cwnd reduces to *Max.Burst*MTU* before starting each subsequent file transfer ([SOA+03] recommends Max.Burst = 4).

4.3 FTP over SCTP with multistreaming and command pipelining (SCTP-MS-CP)

Finally, in "SCTP-MS-CP", SCTP-MS is extended with command pipelining (CP), similar to that defined in [PM94], to avoid unnecessary cwnd reduction between file transfers. In SCTP-MS, 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 5, we present a solution that 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. After parsing the name list of the files, the client sends SIZE commands for all files at once (which SCTP ends up bundling together in its SCTP-PDUs). As each reply for a SIZE command is received, the client immediately sends out the respective RETR command for that file. Since the control stream is ordered, SCTP guarantees the replies to the SIZE and RETR commands will arrive in proper sequence.

By using SCTP-MS-CP, 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 2.1.

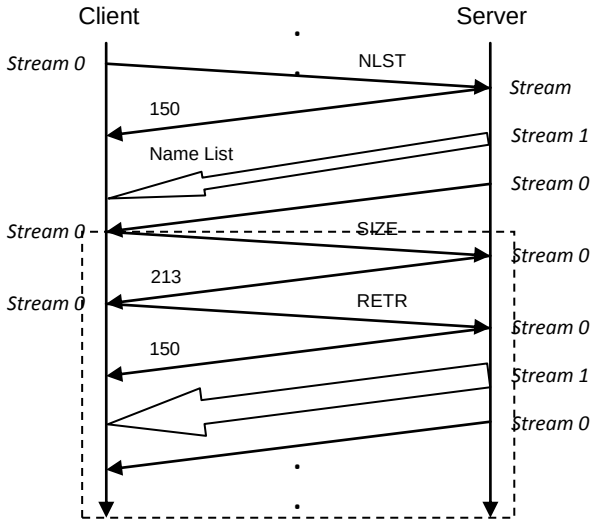


Figure 4: FTP over SCTP-MS

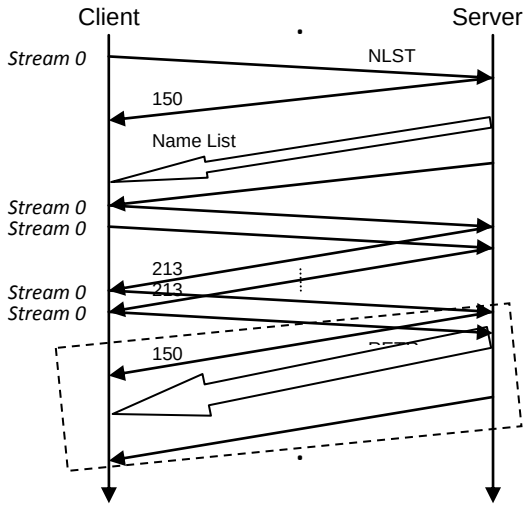


Figure 5: FTP over SCTP-MS-CP

V. RESULTS

To compare FTP over TCP vs. our three SCTP variations, we measured the total transfer time for a multiple file transfer for a varied set of parameters.

- *Bandwidth-Propagation Delay (B-D) configuration*: Three path configurations were evaluated: (3Mbps, 1ms), (1Mbps, 35ms), (256Kbps, 125ms). Both the client to server and server to client paths share the same characteristics. These configurations roughly represent an end-to-end connection on: a local network, U.S. coast-to-coast, and over a satellite, respectively.
- *Packet Loss Ratio (PLR)*: The PLRs studied were (0, .01, .03, .06, and .1). Loss was symmetric; each value represents the loss ratio for both the client to server and the server to client paths. We used a Bernoulli 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 are of serious interest in wireless and military networks.
- *File sizes:* We evaluated the potential overheads for a variety of file sizes: 10K, 50K, 200K, 500K, and 1M.

5.1 Experimental setup

Netbed [WLS+02] (an outgrowth of Emulab) was used to provide integrated access to experimental networks. Three nodes were used for each experiment: an FTP client, an FTP server, and an intermediate node running *Dummynet* [Riz97] to simulate a drop-tail router with a queue size of 500KB, and various bandwidths, propagation delays, and packet loss ratios. The router's queue was set large enough such that buffer overflow, i.e., loss due to congestion, did not occur. The client and server nodes were 850MHz Intel Pentium IIIs.

The client and the server nodes run FreeBSD-4.6. The dummynet router node runs FreeBSD-4.10. The FreeBSD kernel implementation of SCTP available with the KAME Stack [KAME] was used on the client and server nodes. SCTP patchlevel 24 released October 11, 2004 from www.sctp.org was used for the SCTP-MS and SCTP-MS-CP runs. Because of the timing of the experiments, patchlevel 25 released February 21, 2005 was used for SCTP-naïve. KAME is an evolving and experimental stack targeted for IPv6/IPsec in BSD-based operating systems.

In our previous published results [Lad04], Netlab's control connection was inadvertently used by SCTP end-hosts for retransmissions. SCTP is inherently multihomed, and without knowing it, our SCTP associations used Netlab's essentially error-free, no-delay control channel, thus biasing results in favor of SCTP. When rerunning the experiments, only the path thru the dummynet router was used.

We implemented protocol changes by modifying the FTP client and server source code available with the FreeBSD 4.6 distribution. Total transfer time was measured as follows. The starting time was when the "150 Opening" control reply from the server reached the client in response to the client's "NLST" request. The end time was when the server's "226 control reply" 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: "TCP" (the TCP variant used was New-Reno), "SCTP-naïve", "SCTP-MS", and "SCTP-MS-CP".

While we also performed experiments involving single (and multiple) file transfer, we only report the results of experiments involving multiple file transfers. Some minor improvement using SCTP multistreaming was witnessed in a single file transfer, but nothing significant. The major gains of multistreaming are more predominant when transferring multiple files. Additionally, comparing SCTP-naïve vs. TCP for multiple files provides insight on single file transfer.

5.2 Results

Figures 6, 7, and 8 (note: best viewed in color) show results obtained for our three bandwidth-delay configurations. Each graph displays the total time to transfer 100 same-size files for different loss probabilities using the four FTP variants.

5.2.1 TCP vs. SCTP-Naïve. Since SCTP-naïve is simply a straightforward substitution of TCP calls with SCTP calls, any performance difference must be attributed to the different ways our TCP and SCTP implementations handled connection/associate establishment and/or data transfer (i.e., congestion control, loss recovery). Congestion control differences between SCTP and TCP can be found in [AAI02] where the authors note that the congestion control semantics and loss recovery mechanisms in SCTP are robust, and result in better steady state throughput at higher loss rates in a satellite environment.

For all three B-D configurations and file sizes, TCP and SCTP-naïve performed almost identically at 0% loss. In only one case (the long delay satellite configuration with smallest 10K file size) was there a noticeable difference of SCTP being ~10% slower.

As loss was introduced and increased, however, the performance of these two methods clearly diverged. Interestingly, for the smallest file size (10KB), SCTP-naïve performed consistently worse than TCP, and for all other files sizes 50KB – 1MB, SCTP-naïve transferred multiple files consistently faster than TCP. And as the file size increased, so did SCTP-naïve's relative performance improvement.

We investigated many of the tcpdumps and discovered several differences between the two studied implementations that help explain this behavior.

Why TCP does better for short files - First, each SCTP-naïve association establishment uses a 4-leg handshake while TCP connects using 3 legs. (This added leg provides SCTP associations with better defense against DoS attacks [SX01].) SCTP's extra ½ RTT has significant impact; more so for short files. And as loss increases, SCTP incurs a greater chance (i.e., 4 to 3) that the establishment loses a leg, and requires a timeout before recover via retransmission. For newly established associations, this minimum timeout value is conservative (initially minRTO=3s; after the sender measures an RTT, minRTO=1s). Transferring a 10K file only involves ~7 PDUs, so for short transfers, a longer establishment time noticeably degrades SCTP-naïve performance. As file sizes increase, the establishment time becomes less a factor.

Second, an SCTP-naïve sender (and for that matter, all three SCTP variations) requires 4 missing reports before a fast retransmission, while a TCP sender fast retransmits on receipt of 3 dupacks. (Note: an SCTP missing report and a TCP dupack are analogous.) For short files, when the cwnd is often around size 3-4, TCP will be able to recover more often without a timeout via fast retransmit, while SCTP-naïve does not have sufficient PDUs in the pipe, and will require a timeout. As file sizes increase, this fast retransmit difference will not play as important a factor. (Note: in the latest SCTP design, only 3 missing reports will be required for a fast retransmit.) SCTP has Limited Transmit [ABF01], so this difference may not significant.

Why SCTP-naïve does better for longer files - SCTP-naïve's significantly better performance for longer files (increasingly as loss rates increased) initially came as a surprise as it was widely understood that the congestion control mechanisms in TCP and SCTP are approximately the same. The largest improvement is demonstrated in Figure 6's LAN connection transferring 100 – 1MB files at the highest 10% loss rate: SCTP-naïve is four times faster than TCP.

On analysis, we realize that the currently prevalent FreeBSD version of TCP (New-Reno) does not have three congestion control mechanisms included in our SCTP model: Limited Transmit [ABF01], Appropriate Byte Counting [All03], and Selective Acks [MMF96]. One advantage of an experimental protocol such as SCTP is its ability to include newer mechanisms much sooner than for TCP. Once these extensions are included in TCP implementations, we expect (1) and (2) to perform similarly at different loss rates.

In any case, our primary goal is NOT to focus on whether FTP over SCTP-naïve is better or worse than FTP over TCP. Such a comparison would require equivalent FreeBSD implementations, which was beyond the scope of this study. We focus on the gains from multistreaming and command pipelining using SCTP-naïve as a baseline to see if and how much these mechanisms benefit file transfer.

5.2.2 SCTP-MS vs. SCTP-Naïve. We first consider the impact of FTP using a transport layer with multistreaming by comparing SCTP-MS vs SCTP-naïve. In Figures 6-8, we observe that in a lossy environment, significant gains from multistreaming are evident; more so for (1) smaller file sizes vs. larger file sizes, and (2) the highest bandwidth - shortest delay connection (LAN) vs the lowest bandwidth - longest delay connection (satellite). No significant performance difference was observed for (3) 0% loss in the LAN environment, and (4) for all B-D configurations and loss levels when transferring large (1MB) files.

Regarding (1), for all B-D configurations with loss present, multistreaming in SCTP-MS transfers 10KB files in roughly 1/2 the time than without multistreaming (SCTP-naïve) consistently across all loss probabilities. The relative gains decrease to roughly 30-40% faster for 50KB files. SCTP-MS avoids the overhead to set up an additional association for every file, an overhead that is relatively more significant for smaller files.

Evidence of (2) is seen, for example, by comparing the 50KB file transfers and seeing that SCTP-MS for the 3Mbps-1ms link (Figure 6) is ~40% faster than SCTP-naïve, and for the 256Kbps-125ms link (Figure 8), SCTP-MS improves on SCTP-naïve by only ~20%. Because SCTP-naïve has at least 7 extra PDUs (4 for association establishment; 3 for shutdown), SCTP-naïve will experience more timeouts per file transfer than SCTP-MS when there is loss. These additional timeouts degrade SCTP-naïve 'relatively' more when the RTT is shorter because the sender uses a fixed minimum RTO value. When the RTT = 2ms in the LAN scenario (Figure 6), a timeout with minRTO results in ~500 idle RTTs,

whereas for the satellite scenario (Figure 8), only 3 RTTs are idle. Further evidence of a fixed minRTO degrading shorter RTT paths relatively more than longer RTT paths can be found in [IAS05].
Regarding (3), in a LAN environment (see Figure 6), while SCTP-naïve does require an extra establishment association for each file, this overhead delay is minimal because the extra RTTs are short. Only as loss is introduced does the performance between these two versions diverge significantly, because loss in any of the extra 4 legs needed for SCTP-naïve association establishment requires a timeout before recovery via retransmission is possible, and timeouts are relatively 'expensive' in terms of relative delay.

Regarding (4), once files become very large (1MB), the amount of time transferring the file dominates any extra time spent having to establish an association.
One unexpected result appears in Figure 8's satellite link scenario. For large files (500KB - 1MB), SCTP-MS is slightly slower than SCTP-naïve at certain loss rates. We investigated tcpdumps for several runs in detail, and found no protocol behavior to explain this minor inconsistency. We noted that in redoing our experiments, the SCTP-MS version used a slightly older patch (#24) than SCTP-naïve (#25), which could explain the minimal 1-2% difference.

5.2.3 SCTP-MS-CP vs. SCTP-MS. As explained in Section 4.3, when transferring multiple files at once, command pipelining (a) reduces round trips for command exchanges, and (b) maintains the probed value of the congestion window for subsequent transfers in a multiple file transfer. We note that command pipelining is not exploiting a new transport layer mechanism as is the case of using multistreaming. Conceptually, FTP over TCP could also be designed to pipeline the file retrieval commands over the control channel.
We hypothesized the effect of (a) would remain fairly constant irrespective of file sizes being transferred and loss rate, and the effect of (b) would be more prevalent in transferring smaller files. For small files, more time (relatively) is spent by SCTP-MS in slow start probing for available bandwidth compared to the amount of time spent probing in large file transfers. By avoiding this reprobing for each file, and spending more time in steady state congestion avoidance phase, SCTP-MS-CP gains should be more evident for smaller files.

Figures 6-8 confirm these hypotheses. For all three B-D configurations, command pipelining introduces clear performance improvements, more so for the smaller files. The most pronounced improvement is seen in Figure 6, where for 10KB files, SCTP-MS-CP transfer files as much as 8 times faster than SCTP-MS as loss increases above 2%. Even for transferring one hundred 200KB files, SCTP-MS-CP does 30% better than SCTP-MS. When the file size increases to 1MB, some gain using command pipelining is noticeable, but the majority of time spent in congestion avoidance (as opposed to slow start, and doing command exchanges) dominates the transfer time, making the gain of SCTP-MS-CP over SCTP-MS only minimally significant.

VI. CONCLUSION

Our experimental results confirm that modifying FTP to use SCTP multistreaming and command pipelining can dramatically benefit mirroring (e.g., *fmirror*) and other applications which transfer a large number of files from host to host. 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 documented in our performance experiments, other advantages of FTP over SCTP-MS-CP vs. FTP 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 SCTP-MS-CP, servers could serve at least twice the number of clients compared to the current FTP over TCP design when the bottleneck for the number of simultaneous clients served is the TCBs reserved for the connections. This result should be of interest to busy servers that are constrained by the number of simultaneous clients.
- The number of PDUs exchanged between client and server is reduced (e.g., by reducing the command exchanges and connection establishments/teardowns) thus reducing the overall network load.
- Aggregating control and data connections into one SCTP multistreamed association solves concerns that FTP over TCP faces with Network Address Translators (NAT) and firewalls in transferring IP addresses and port numbers through the control connection [AOM98, Tou02].

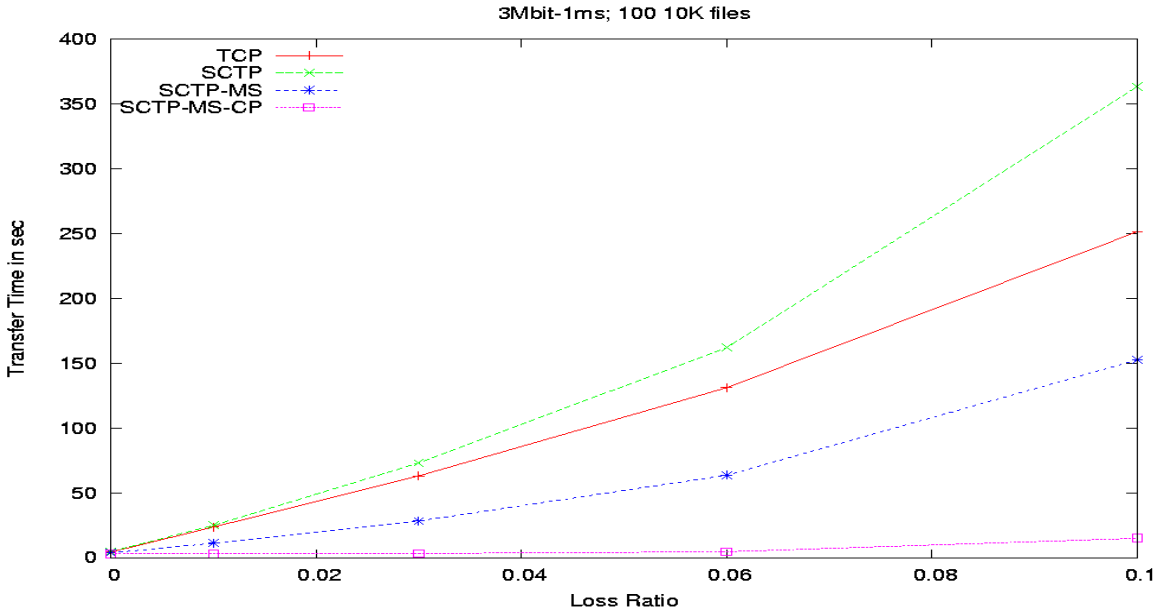
ACKNOWLEDGMENT

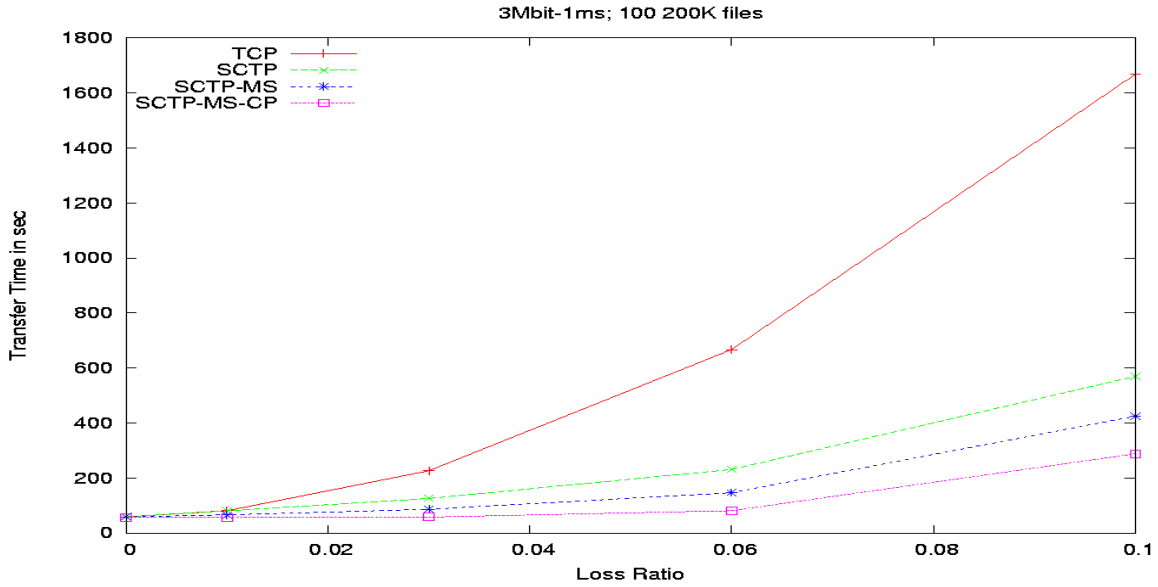
I would like to express my sincere thanks to my Guide and my Co-Authors for their consistence support and valuable suggestions.

REFERENCES

[1] R. Alamgir, M. Atiquzzaman, W. Ivancic, *Effect of Congestion Control on the Perf of TCP and SCTP over Satellite Nets*. Proc. NASA Earth Science Tech Conf, 6/02. Pasadena, CA.
[2] M. Allman, H. Balakrishnan, S. Floyd, *Enhancing TCP's Loss Recovery Using Limited Transmit*, RFC 3042, 1/01.
[3] M. Allman, A. Falk, *On the Effective Evaluation of TCP*. ACM CCR, 29(5), 10/99.
[4] M. Allman, *TCP Congestion Control with Appropriate Byte Counting (ABC)*, RFC3465, 2/03.

[5] R. Elz, P. Hethmon, *Extensions to FTP*. draft-ietf-ftpext-mlst-16.txt, IETF Internet draft (work in progress), 3/03.
[6] S. Floyd, K. Fall, *Promoting the Use of End-to-End Congestion Control in the Internet*. IEEE/ACM Trans on Networking, 8/99.
[7] S. Floyd, T. Henderson, *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC2582, 4/99.
[8] T. Faber, J. Touch, W. Yue, *The TIME-WAIT State in TCP and Its Effect on Busy Servers*. Proc Infocom, 3/99, NYC.
[9] S. Ladha, P. Amer, *Improving multiple file transfers using SCTP multistreaming*, 23rd IEEE Int'l Perf, Computing, and Comm Conf (IPCCC), Phoenix, 4/04, 513-22
[10] S. McCreary, K. Clay, *Trends in WAN IP Traffic Patterns - Ames Internet Exchange*.. ITC, 9/00. Monterey. [MMF96] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow. *TCP Selective Acknowledgment Options*, RFC2018, 10/96.
[11] L. Rizzo, *Dummynet: a simple approach to the evaluation of network protocols*. ACM CCR, 27(1):3141, 1/97.
[12] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, M. Tuexen, *SCTP Implementers Guide*, draft-ietf-tsvwg-sctpimpguide-13.txt (work in progress), 2/05
[13] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, *SCTP*, RFC2960, 10/00.
[14] R. Stewart, Q. Xie, L. Yarroll, J. Wood, K. Poon,, M. Tuexen, *Sockets API Extensions for SCTP*. draft-ietf-tsvwg-sctpsocket-10.txt, (work in progress), 2/05.
[15] B. White, et al. *An Integrated Experimental Environment for Dist'd Systems and Networks*. Proc. 5th Symp on OS Design and Implementation, 12/02. Boston.
[16] [WUARCHIVE] Usage Statistics for wuarchive, wuarchive.wustl.edu





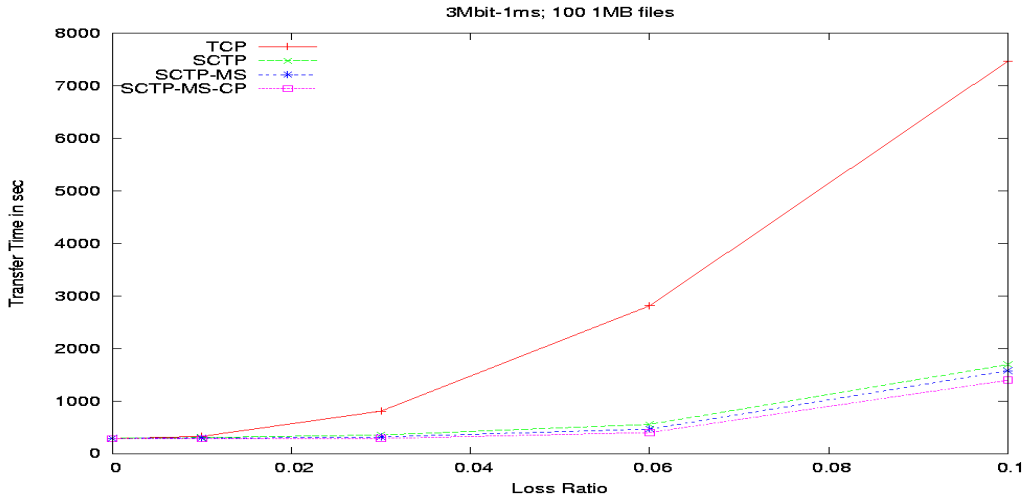
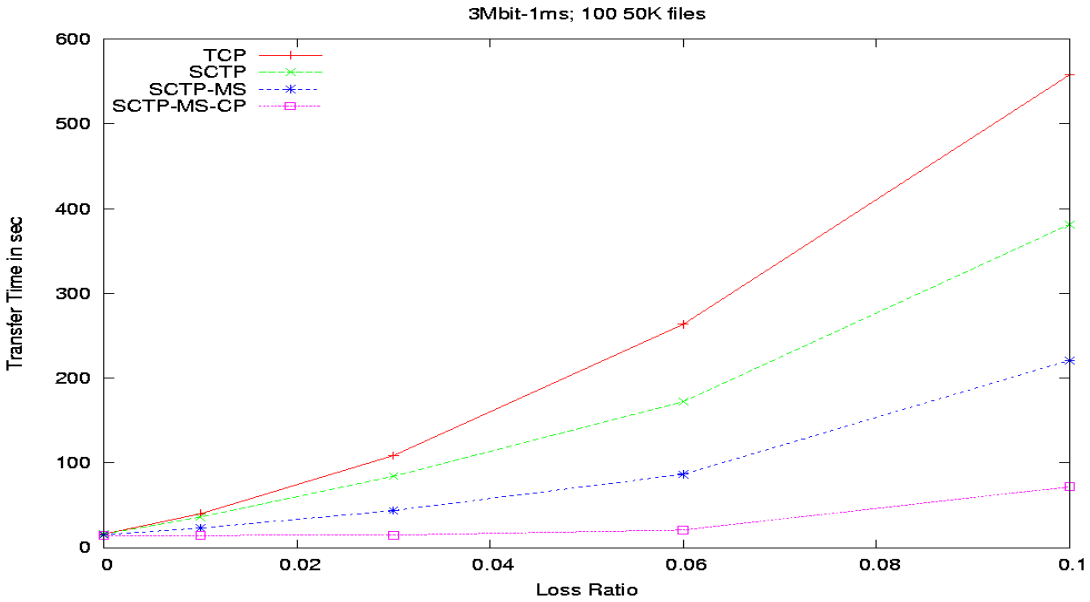
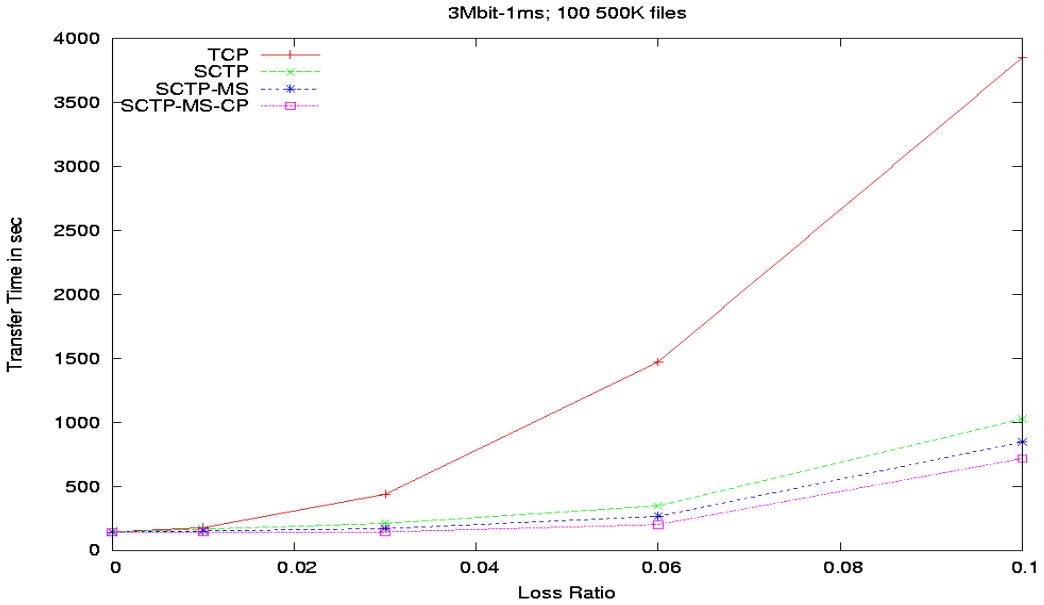
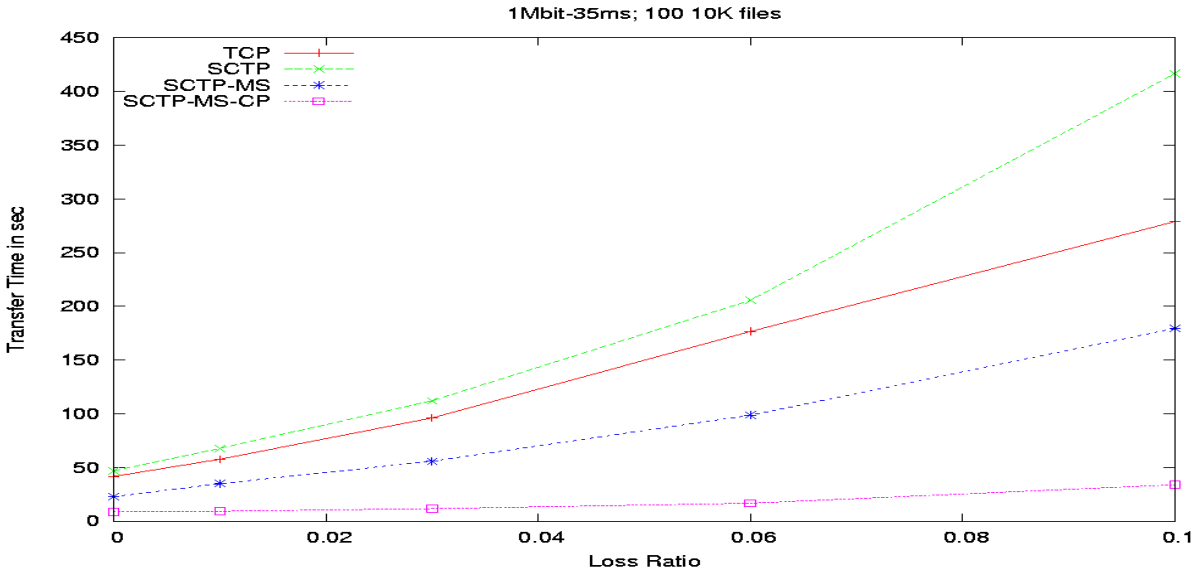
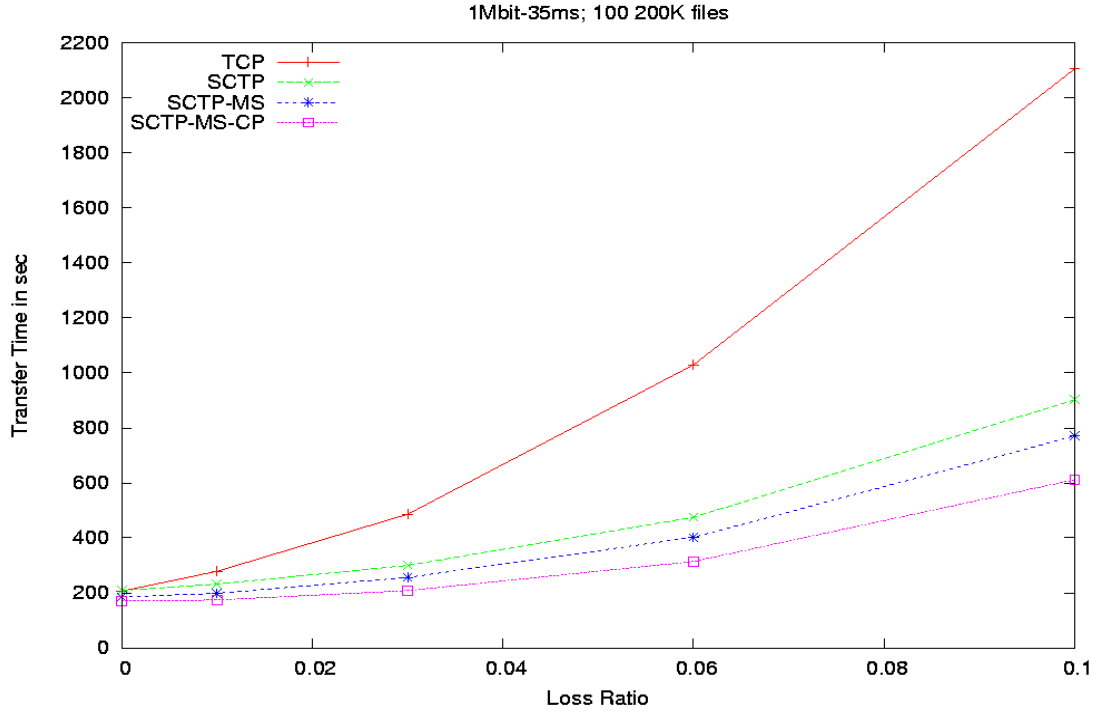


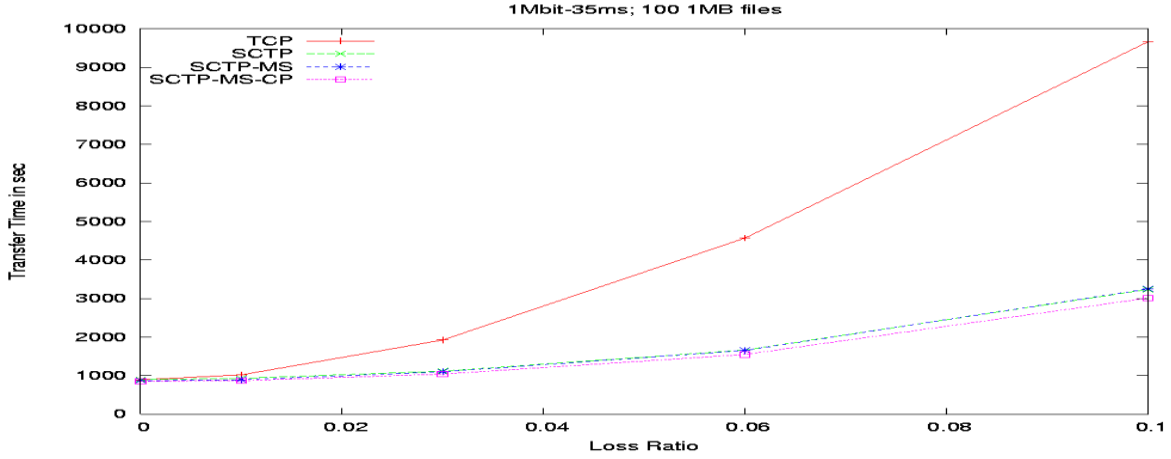
Figure 6: Transfer Time vs. Loss Ratio for a multiple transfer of 100 files on a LAN-like link (Bandwidth = 3Mbps, Propagation Delay = 1 ms)






1Mbit-35ms; 100 10K files
TCP
SCTP
SCTP-MS
SCTP-MS-CP
Transfer Time in sec
Loss Ratio
0
50
100
150
200
250
300
350
400
450
0
0.02
0.04
0.06
0.08
0.1


1Mbit-35ms; 100 200K files
TCP
SCTP
SCTP-MS
SCTP-MS-CP
Transfer Time in sec
Loss Ratio
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
0
0.02
0.04
0.06
0.08
0.1


1Mbit-35ms; 100 1MB files
TCP
SCTP
SCTP-MS
SCTP-MS-CP
Transfer Time in sec
Loss Ratio
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.02
0.04
0.06
0.08
0.1

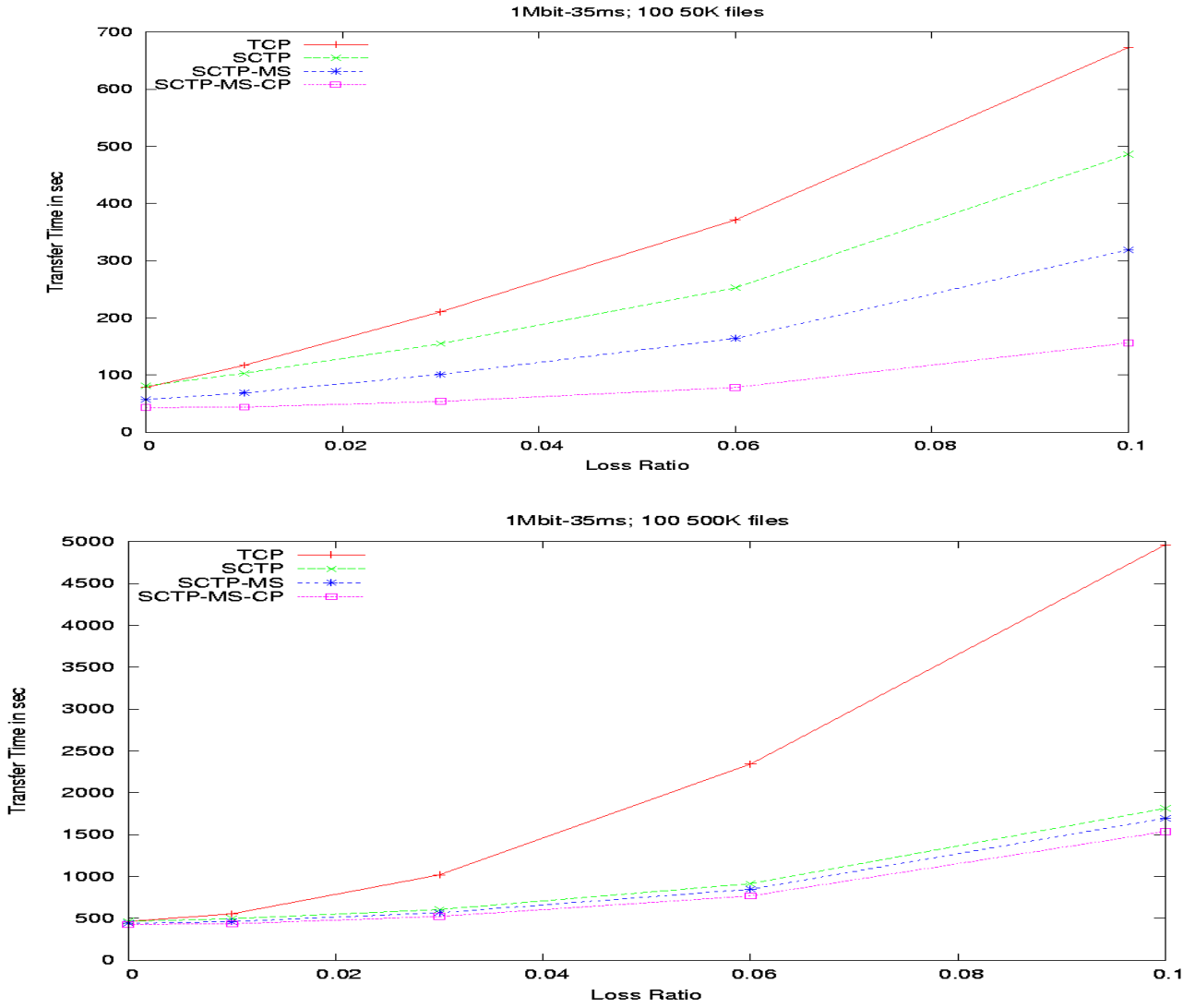
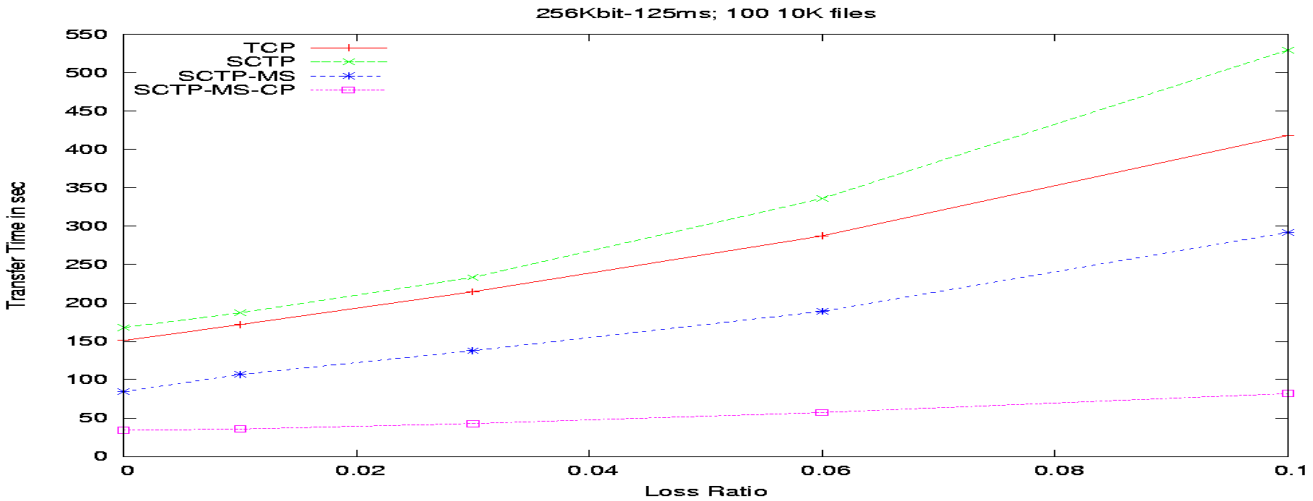


Figure 7: Transfer Time vs. Loss Ratio for a multiple transfer of 100 files on a US coast-to-coast-like link (Bandwidth = 1Mbps, Propagation Delay = 35 ms)



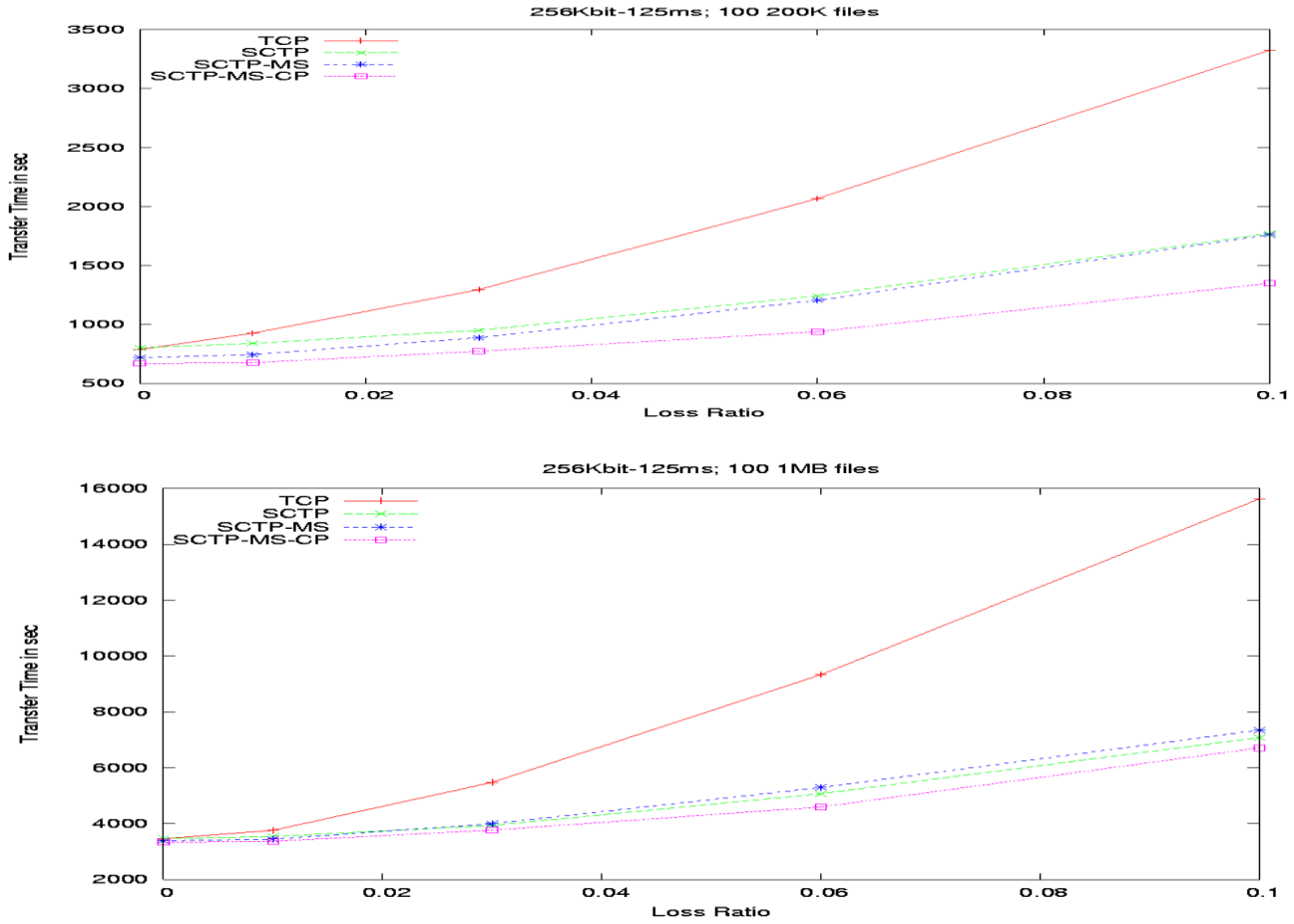


Figure 8: Transfer Time vs. Loss Ratio for a multiple transfer of 100 files on a satellite-like link

(Bandwidth = 256Kbps, Propagation Delay = 125 ms)