IMPROVING THE PERFORMANCE OF TCP OVER SATELLITE CHANNELS

by

Spiros Philopoulos

A Thesis presented to the University of Manitoba in
partial fulfillment of the requirements for the degree of
Master of Science
in the
Department of Electrical and Computer Engineering

Winnipeg, Manitoba

June, 2001

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Improving the Performance of TCP Over Satellite Channels

BY

Spiros Philopoulos

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ABSTRACT

Reliable transport protocols such as the TCP transport layer protocol have been designed and tuned for traditional wired networks. TCP performs relatively well on such wired networks by adapting to end-to-end delay and to losses caused by congestion. However, in the case of wireless links, such as satellite channels, due to the special attributes of satellite links, such as high bit error rates, large propagation delays etc., TCP performance degrades significantly. The work described in this thesis was motivated by the need for improving the performance of TCP over satellite channels. In this thesis three connection-splitting architectures based on the use of transparent proxies are implemented and tested. Namely, a single-TCP-interproxy-connection connection-splitting architecture, a multiple-TCP-interproxy-connection connection-splitting architecture and a UDP-interproxy-connection connection-splitting architecture are implemented. The throughput increases obtained by the latter two architectures were very significant with the UDP connection-splitting scheme even yielding a throughput improvement of an order of magnitude in some cases.

ACKNOWLEDGEMENTS

I would like to begin by thanking Dr. K. Ferens for proposing this topic and helping me throughout the course of this thesis. I would also like to thank Dr. R. D. McLeod for his advice and help in completing this thesis. Many thanks also go out to Guy Jonatschick, Arindam Mitra, Thuraiappah Vaseeharan and especially to Dr. Muthucumaru Maheswaran for providing me with some of the resources necessary to complete this thesis.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Reliable transport protocols such as the TCP transport layer protocol have been designed and tuned for traditional wired networks. TCP performs relatively well on such wired networks by adapting to end-to-end delay and to losses caused by congestion. However, in the case of wireless links, such as satellite channels, due to the special characteristics of satellite links, such as high bit error rates, large propagation delays etc., TCP performance degrades significantly. The following is a brief overview of the TCP protocol, followed by an outline of the problems encountered by TCP in a satellite environment.

1.1 Overview of TCP Protocol

TCP is a networking protocol that provides reliable connections between two applications. TCP provides congestion control so as to not send data to the receiver faster than the receiver can receive data, and also provides flow control so as to not congest the network. Data is sent by TCP by segmenting a byte stream of data into a number of segments up to a maximum segment size, called *MSS*, typically 536 or 1460 bytes in size. In order for data to be exchanged between two endpoints first a connection must be established between them and this is done by performing the so called *three-way handshake.* The three-way handshake consists of having the initiator of the connection send a packet/segment with the *SYN* flag set to the receiver to be, which receiver if it decides to accept the connection request will respond by sending a segment with the *SYN* and *ACK* flags set, acknowledging and accepting the connection request. Then the connection setup will be completed by the initiator by sending an *ACK* segment (i.e. *ACK* flag set) back to the receiver. This connection setup procedure is referred to as the *three-way handshake* and takes an approximate amount of time of one *round trip time (RTT)*, which is the amount of time for a segment to traverse the network path in both directions.

Once the connection setup is complete the two endpoints can exchange data and every packet sent must be acknowledged by the receiver. In older TCP implementations acknowledgments were sent for every segment transmitted, however for years now TCP implementations implement the so-called *delayed ACK* whereby acknowledgments are coalesced for up to two segments. In TCP, acknowledgments are cumulative rather than selective, meaning that TCP will not acknowledge a particular segment if there are segments preceding it that have not been received yet.

Flow control and congestion control are made possible by the use of what are called *sliding windows* and are defined in terms of *MSS* segments in size. The size of the sending window determines how many segments can be sent by the transmitter so as to not congest the receiver (congestion control) or the network (flow control). The two windows that exist in TCP are the *receiver-advertised window* -advertised in every packet sent by the receiver- and the *congestion window*. The *receiver-advertised window* value reflects the approximate amount of data that the receiver is capable of receiving without being congested i.e. without having its receive buffers overflow. To what value the *receiver-advertised window* is set to, e.g. whether it is exactly the amount of free space in the receive buffers or some other way of calculation, varies depending on the particular TCP implementation. The *congestion window* on the other hand, is the window used by the transmitter and it represents the amount of data the sender can transmit before receiving an acknowledgement. The actual window used by the sender though when transmitting data is not the *congestion window* but the so-called *effective window* which is defined as the minimum of the *congestion window* and the *receiver-advertised window*. Since in well designed TCP implementations the *receiver-advertised window* is typically set to a fixed size of 64KB, the *congestion window* determines the flow of data from the sender.

While TCP manages congestion control via the use of sliding windows, there is also the issue of flow control i.e. not congesting the other network resources along a given network path e.g. routers. If TCP's transmission rates are too high then intermediate routers along the network path will be overwhelmed, leading to buffer overflow and eventually causing the routers to discard packets. Therefore, TCP must adjust its transmission rate to the network conditions in

order to avoid segment loss and thus retransmissions. If a large number of TCP connections are sending data at rates inappropriately high then the network might suffer from *congestive collapse.* In a state of *congestive collapse* most of the data segments sent, and even their corresponding acknowledgments, will be lost forcing segment retransmission making the situation even worse, bringing the network to a virtual halt. The flow control mechanisms employed by TCP are the *Retransmission Time-Out (RTO)* mechanism and the *Fast Recovery & Fast Retransmission* mechanism. Before proceeding to the explanation of these two flow control mechanisms it is necessary to first explain how the *congestion window* evolves.

There are two ways by which the *congestion window* evolves: a) *slow start* and b) *congestion avoidance*. In *slow start* the *congestion window* starts off with a small size of typically one *MSS* and gradually increases in size by one *MSS* for every acknowledgment received, resulting in an exponential *congestion window* growth rate. TCP uses *slow start* whenever it is not sure how fast to send data, as for example at the beginning of a TCP session, after congestion in some cases and after idle periods. Once the *congestion window* size reaches the value of the *slow start threshold*-(*sstresh*), *slow start* mode terminates and *congestion avoidance* mode commences. The *slow start threshold* is initially set to the value of the *receiver-advertised window* at the start of the connection. In *congestion avoidance* mode, the *congestion window* increases at a slower rate than in *slow start* mode, by increasing by 1/(*congestion window*), unless the window size is equal to the *receiver-advertised window* size, for every acknowledgement received thus resulting in a linear sliding window growth rate of approximately one segment every *RTT*. The linear window growth rate reflects the fact that *congestion avoidance* mode is used to conservatively probe for additional bandwidth after *slow start* mode [Allm97].

Returning to the *Retransmission Time-Out* and *Fast Recovery & Fast Retransmission* mechanisms, both of these mechanisms detect packet loss and retransmit the lost segments(s). With the *Retransmission Time-Out* mechanism, every segment sent by TCP has a timer associated with that segment. If an acknowledgement for that segment is not received within a certain amount of time, the timer expires (a timeout occurs) and the sender responds by

retransmitting the segment and entering into *slow start mode* reducing its *congestion window* size to one. This has the effect of severely reducing the sender's transmission data rate, and also resets the *slow start threshold* to half of the current *congestion window* size (before the *congestion window* is set to one of course). The timeout value associated with each TCP segment is a smoothed estimation of the *round trip time-(RTT)* plus some variation. The second method is based on receiving multiple duplicate acknowledgments (with the receiver sending the same acknowledgment for every segment received after the lost one), typically three, which leads the sender to believe that the segment after the last acknowledged segment has been lost (with the TCP segments after the lost one not being able to be acknowledged since TCP uses cumulative acknowledgments). In response the sender will retransmit the packet believed to be lost, referred to as a *fast retransmit*, and will halve the *congestion window* as opposed to setting it to one *MSS* as with *slow start*, and the *slow start threshold* is set to the new *congestion window* size. This is referred to as *fast recovery*. As can be easily understood the effects of *fast retransmission* and *fast recovery* on the TCP sending rate are are significantly less adverse than those of the *Retransmission Time-Out* mechanism. When a retransmission occurs due to a timeout expiry TCP cannot infer anything about the state of the network and thus reduces its *congestion window* to one and enters *slow start* in order to not aggravate a potentially serious congestion problem in the network. In the case of *fast retransmission* though, TCP does receive duplicate acknowledgements for TCP segments that were transmitted after the lost segment thus indicating that there continues to be a flow of data from sender to receiver and therefore the sending rate reduction does not have to be as severe [Allm97].

1.2 Satellite Link Characteristics and their Adverse Effects on TCP

1.2.1 Satellite Link Characteristics

Communication satellite systems can be categorized into three categories based on their orbit: Geosynchronous (GSO), Medium Earth Orbit (MEO) and Low Earth Orbit (LEO). Geosynchronous satellites are located at a geostationary orbit with an altitude of approximately 36,000 km. Due to this large altitude the propagation delay for these satellites is large compared

to that of terrestrial links, or any other type of link for that matter, varying in range typically from 240 msec to 270 msec for a signal to travel twice that distance of 36,000 km i.e. one hop (from ground-station to satellite to ground-station). Thus the *Round Trip Time (RTT)* for a GSO satellite link alone approaches 500 msec on average, not including the delay of the rest of the network. For LEO satellites the delay is significantly smaller ranging from a few msec to as much as 80 msec. The *large propagation delays* are one of the important characteristics of satellite networks that distinguish them from terrestrial networks. A second fundamental characteristic of satellite channels is the *higher bit error rates (BER)* that they exhibit which are typically in the range of one error in 10^7 bits or 1×10^{-7}, and for some older satellite systems the *BER* even reaches levels of 1×10^{-5}. Other fundamental characteristics of satellite channels include *bandwidth asymmetry*, *variable RTTs* and *interminent connectivity* [RFC2488]. *Bandwidth asymmetry* is exhibited in asymmetric satellite networks that where built, usually due to cost reasons, with one direction of the link having a higher capacity than the reverse direction link. For example outgoing traffic might use the satellite channel but with the return link using a slow terrestrial link for example, such as a modem channel. Regarding *variable RTTs*, in some satellite environments, such as LEO satellite systems, the propagation delay will vary over time. However it is still uncertain whether variable RTTs have an adverse affect on TCP performance. Regarding the last fundamental characteristic mentioned, *interminent connectivity* is exhibited in non-GSO satellite systems where handoffs are required, which if not performed will lead to increased packet loss.

Summarizing, the fundamental characteristics of satellite systems that distinguish them from terrestrial networks are:

i) Large propagation delays (in GSO systems to a greater extent)

ii) High bit error rates

iii) Bandwidth asymmetry

iv) Variable RTTs (in LEO satellite systems to a greater extent)

v) Interminent connectivity leading to greater packet loss (in non-GSO systems)

The two satellite channel characteristics with the largest adverse effect on TCP performance are the *large propagation delay* exhibited in GSO satellite networks and the *high bit error rates*. Since these two characteristics are those that most adversely affect TCP, the focus in this thesis will be on those, along with bandwidth asymmetry, and on GSO satellites as GSO satellites exhibit these characteristics most.

1.2.2 Adverse Effects of Propagation Delay

Two adverse effects of the large propagation delays in satellite networks on TCP performance are: sliding window growth problems and underutilization of available bandwidth [Allm97] [Krus95]. Sliding window growth problems are exhibited in both the *slow start* and *congestion avoidance* phases. [JaKa88] defines that the time needed in *slow start* mode for the sliding window to reach a size of W segments given a RTT of R is:

$$Slow\ \ Start\ \ Time = R \cdot \log_2 W \qquad (1.1)$$

Therefore, for a given effective sliding window of W segments, we have:

$$\frac{Satellite\ \ Slow\ \ Start\ \ Time}{Terrestrial\ \ Slow\ \ Start\ \ Time} = \frac{R_{SAT} \cdot \log_2 W}{R_{TER} \cdot \log_2 W} = \frac{RTT_{SAT}}{RTT_{TER}} \qquad (1.2)$$

For typical RTT values the ratio in (1.2) exceeds the value of 5 i.e. in a satellite environment in *slow start* mode it takes the sliding window much longer to reach its maximum size. Fig. 1.1 shows, based on the mathematical model of equation (1.1), the time required by a satellite network and a terrestrial network to complete the *slow start* mode at the start of a TCP connection, i.e. starting from a *congestion window* of size one up to the typical *sstresh* of 64KB, using typical RTT values and using a segment size of 1KB (i.e. W= 64 segments), assuming lossless operation. Many short flow (i.e. small size) data transfers can complete without ever having attained a window large enough for optimal link utilization. Similarly, in the

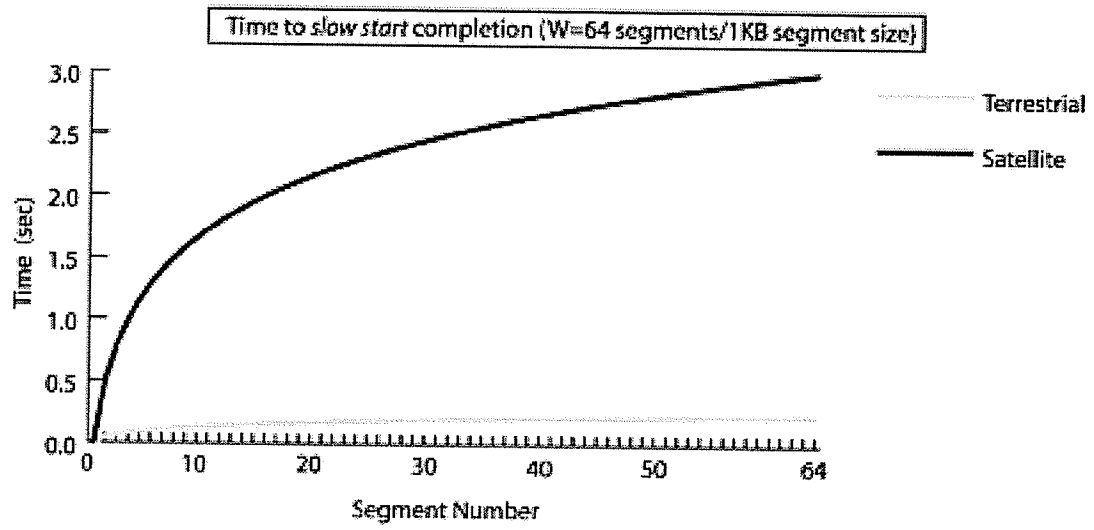

Fig. 1.1. Slow start mode comparison (mathematical models) for satellite and terrestrial networks.

congestion avoidance phase it is obvious that the growth rate of the sliding window in a satellite network will be much smaller than in a terrestrial network since as we know the congestion window increases in *congestion avoidance* by approximately one segment every *RTT* with the *RTT* being much larger in the satellite case. The time to increase the *congestion window* size by W segments in *congestion avoidance* is simply $RTT \cdot W$. Using this simple mathematical model Fig. 1.2 compares the times required in *congestion avoidance* mode for both terrestrial and satellite networks to increase the *congestion window* from 32 to 64 segments. Again, as in Fig. 1.1, the satellite network with its much larger *RTT* takes significantly longer to increase the *congestion window* size.

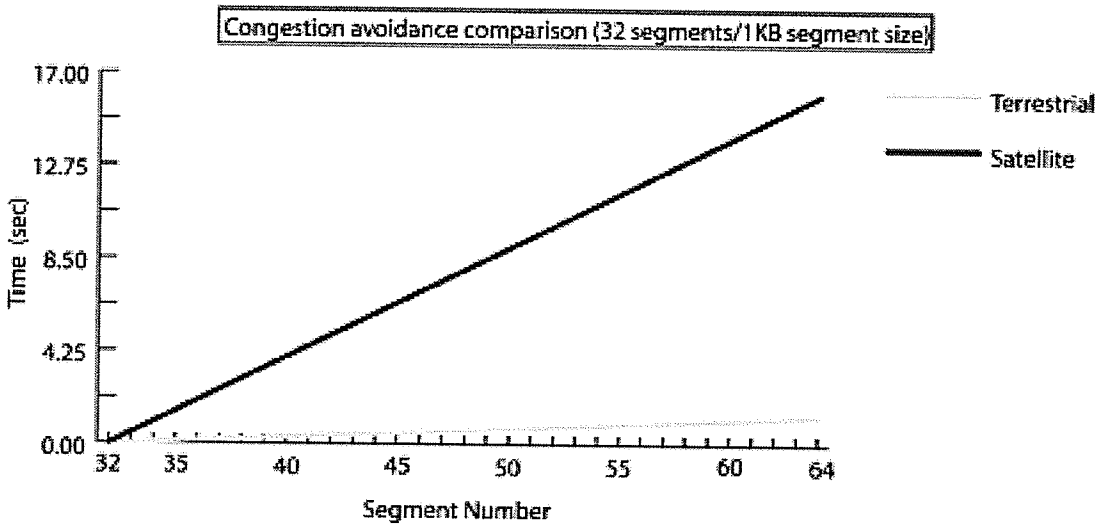

Fig. 1.2. Congestion avoidance comparison (mathematical models) for satellite and terrestrial networks.

The second adverse effect of the large propagation delays in satellite environments, as mentioned before, is the significant *underutilization of available bandwidth.* As given in the original RFC 793 that defined TCP, the maximum throughput of a TCP connection is

$$Max\ \ Throughput = \frac{Max\ \ \text{Re}ceiver\ \ Window\ \ Size}{RTT} \qquad (1.3)$$

therefore for a typical GSO satellite link *RTT* of 500 msec and the maximum possible receiver advertised window in TCP of 64 KB, the maximum throughput for TCP over a satellite link is:

$$Max\ \ Throughput = \frac{65{,}535\ \ bytes}{500 \times 10^{-3}\ \text{sec}} = 131{,}070\ \ bytes/\text{sec}$$

This maximum throughput is well below the bandwidth capabilities of many satellite channels which can range from compressed phone circuits of a few kb/s to very large bandwidths. Compared to for example the throughput of a T1 rate satellite channel of 1.536 Mbits/sec (192,000 bytes/sec) there is only a 68% maximum utilization of bandwidth. For other higher bandwidth satellite channels the bandwidth utilization reaches of course even lower

unacceptable levels.

1.2.3 Adverse Effects of High Bit Error Rate

Another fundamental characteristic of satellites that has a significant effect on the performance of TCP is the relatively *high bit error rate (BER)*. This aside from leading to performance degradation because of needed retransmissions, performance is further reduced since TCP assumes every loss is due to congestion, thus unnecessarily resorting to congestion mitigation in many instances. In the case that a segment loss due to corruption, and not congestion, results in a timeout, TCP enters *slow start* mode, the *congestion window* is reset to a size of one, having a detrimental effect on the TCP sending rate, and the *slow start threshold* is set to half of the current *congestion window* value (before being reset to one of course), thus reducing the rate at which the *congestion window* will grow since *slow start* with its exponential window growth rate will terminate earlier with the reduced *sstresh* value. In the case that a segment loss due to corruption results in a *fast retransmission* and *fast recovery*, the effect on the TCP sending rate is less adverse than in the case of a timeout but still significant. The *congestion window* is set to half its current size, thus halving the sending rate, and the *slow start threshold* is set to half of the *congestion window* value (before it is reset), thus reducing the rate at which the *congestion window* will grow when TCP is in *slow start* mode as explained previously.

1.2.4 Adverse Effects of Bandwidth Asymmetry

Bandwidth asymmetry is another property of some satellite channels that adversely affects TCP performance due to the fact that TCP relies on feedback in the form of cumulative acknowledgments in order to perform flow and congestion control properly. When the bandwidth asymmetry between the forward and reverse link channels exceeds a certain threshold then the reverse acknowledgment channel will become congested because of the number of acknowledgments that must be transmitted in order to acknowledge the data flowing in the forward direction link. This reverse link congestion will disrupt the flow of

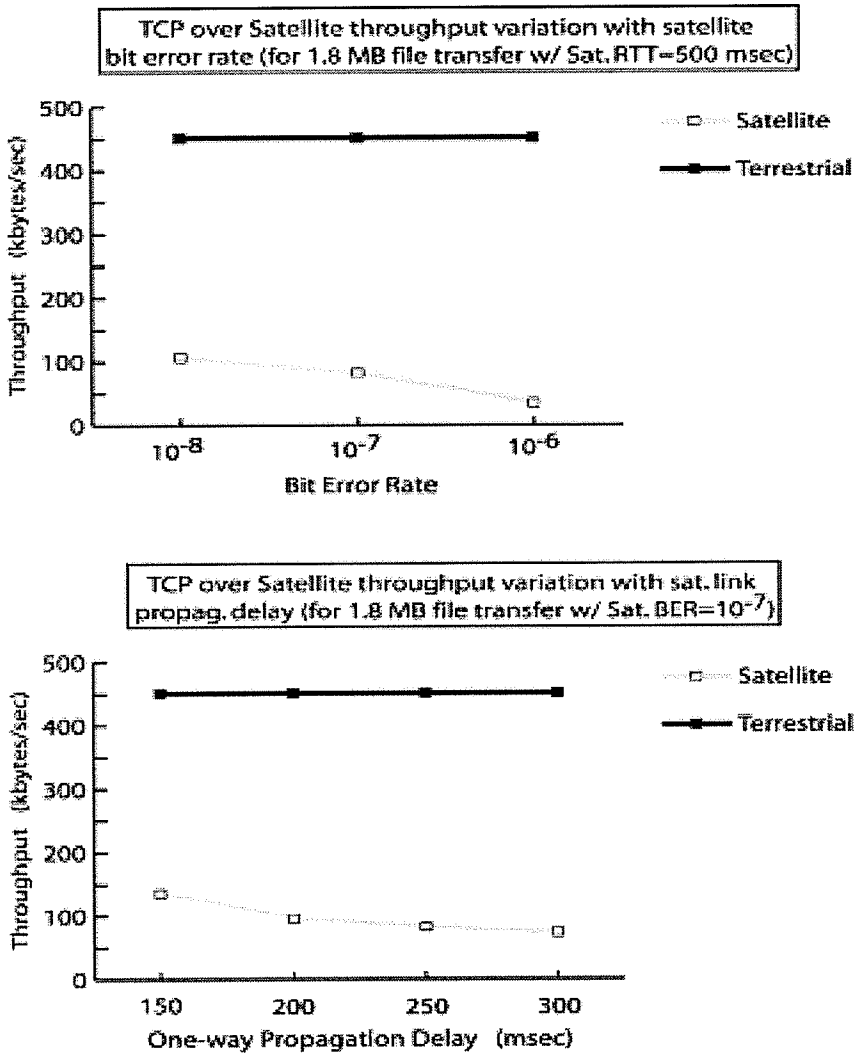

Fig. 1.3 & 1.4. Variation of TCP-over-Satellite throughput with varying satellite link BER and propagation delay.

acknowledgments to the transmitter presenting hindrances in maintaining proper flow and congestion control on the sender's part, ultimately resulting in lost acknowledgments and thus inducing unnecessary timeouts, retransmissions and sending rate reduction/congestion mitigation.

1.2.5 Adverse Effects of Satellite on TCP performance: Experimental Verification

Fig. 1.3 and 1.4 show the actual variation of TCP-over-satellite throughput with varying satellite link BER and propagation delay, compared to TCP performance without the satellite link, obtained from experiments performed for a 1.8 MB file transfer. Fig. 1.3 more specifically, shows the effect of increasing the satellite channel bit error rate on TCP performance, with the error rate varying from 10^{-6} to 10^{-8} while using a typical satellite link *RTT* of 500 msec. This is compared to TCP performance when omitting the satellite link. The very significant adverse affect of satellite channel attributes on TCP performance is obvious, with the TCP throughput when the satellite link is included being less than 25% of that when the satellite channel is not present, and decreasing as the *BER* increases. Similar results are obtained, as shown in Fig. 1.4, when varying the satellite channel one-way propagation delay from 150 msec to 300 msec, while using a typical *BER* of 10^{-7}. The tests performed to obtain the aforementioned results, where done using a proprietary TCP-based file transfer protocol implemented by the author of this thesis [Phil00], and to emulate the satellite link, the NIST Net 2.0.10 link emulation package [NIST00] was used. More details on these and the test configuration can be found in Chapter 4 and the aforementioned references.

CHAPTER II

BACKGROUND THEORY

2.1 Background Literature

The topic of TCP performance over satellite channels has been studied quite extensively the past few years, producing a plethora of literature and proposed solutions on this topic. Below a list of many of the most important proposals is given, most of which have already been documented in RFCs by the IETF[RFC2488] [RFC2760].

2.1.1 TCP for Transactions

[RFC1644] As we know TCP uses a connection setup procedure to establish a connection between two endpoints with the typical setup being the so-called three-way handshake. The connection setup procedure takes between 1 and 1.5 RTTs (Round Trip Times) to complete. During this period there is no transfer of data which is a small inefficiency of the TCP protocol which though is quite significant for short flow TCP connections such as HTTP connections for example. This can be mitigated by implementing TCP extensions for translations (T/TCP), as specified in RFC 1644 "T/TCP - TCP Extensions for Transactions" in 1994, which allows the transfer of data during connection setup by including data in the connection setup packets. It should be noted that while T/TCP was recommended in the RFC 1644 (1994), it was also part of the original TCP specification in RFC 793 in 1981. The implementation of T/TCP requires modification of both the transmitter's and receiver's stacks and is considered safe to be implemented in a shared network environment, such as the Internet, from a congestion control point of view. However there are some security issues that arise by sending data in the initial setup packets.

2.1.2 Slow start

2.1.2.1 Larger Initial Congestion Windows

[RFC2414] TCP uses the *slow start* algorithm at the beginning of a new connection, after long idle periods and after congestion has been detected. *Slow start* grows the window at an exponential rate however always starts from an initial window size of one. Starting from such a small initial window size is inefficient particularly for short flow transfers such as WWW traffic. The problem is further amplified by the use of Delayed ACKs whereby an ACK is not sent by the receiver until two full-sized segments have been received or a 500 msec (maximum) timer has expired leading to a significant delay until the window grows to a size of two. Starting from a larger size would improve efficiency during the startup phase of a connection and an optional initial window size of two has already been proposed by RFC 2581. Experiments have also been done with initial widow sizes of 3 and 4 which showed improved performance although a small increase in dropped packets was also observed. Implementing initial window sizes of 2 to 4 is considered safe from a congestion perspective and would require modification of the transmitter's stack only.

2.1.2.2 Delayed ACKs after *slow start*

Today's TCP stacks implement the Delayed ACK mechanism whereby an ACK is not sent by the receiver until two full-sized segments have been received or a max 500 msec timer has expired. One option is to implement Delayed ACKs only after *slow start* so that the ACK rate increases thus resulting in faster congestion window growth. Experiments have shown an increase in throughput by implementing Delayed ACKs only after *slow start*, however the loss rate also increases somewhat due to the faster window growth. This could also be a problem for networks that have asymmetric links (in terms of bandwidth) where the reverse link (carrying the ACK stream) would be further congested and the Delayed ACKs after *slow start* mechanism would also conflict with ACK rate reduction measures such as ACK congestion control and ACK filtering that will be presented later on. The implementation of this mechanism

requires only modification of the receiver stack and there would have to be some way by which the receiver can detect if the transmitter is currently in *slow start* or not. This *slow start* detection could be done by using a heuristic or having the transmitter indicate to the receiver that it is in *slow start* (which would require modifications on the sender's side also to implement).

2.1.2.3 Terminating *slow start*

[JHoe96] One of the more important issues with TCP is the *slow start* threshold value *sstresh*, i.e. the value used to determine when *slow start* should cease for *congestion avoidance* to take over. Choosing an appropriate *sstresh* value can eliminate the problem of having *slow start* not end soon enough leading to congestion of the link and dropped packets. Currently *sstresh* is set to the size of the receiver advertised window initially and to half the current congestion window size after congestion. One algorithm proposed to determine more appropriate *sstresh* values is the packet-pair algorithm in combination with the measured round-trip time RTT proposed by [JHoe96]. The algorithm estimates the link bandwidth by observing the spacing of the ACKs in the reverse link, and together with the estimated RTT is used to calculate the bandwidth-delay product which is used as the *sstresh* value. The implementation of this mechanism requires modifications to the transmitter's stack only and can be safely implemented in shared networks. A problem however is the correct estimation of the available bandwidth and more so in the case of asymmetric networks where the bandwidths of the forward and reverse links differ and thus the bandwidth estimated from the reverse link is erroneous and not representative at all of the forward link bandwidth.

2.1.2.4 Byte Counting

[Allm97] [Allm98] A method that can be used to implement congestion window growth that is independent of the ACK mechanism used by the receiver stack is *byte counting*. With *byte counting* instead of basing the growth of the congestion window on the number of ACKs received (e.g. in *slow start* the congestion window grows by one for each ACK received) the window growth is based on the number of bytes of acknowledged data. Two *byte counting*

algorithms that have been proposed are the *Unlimited Byte Counting (UBC)* algorithm and the *Limited Byte Counting (LBC)* algorithm. With the UBC algorithm, the congestion window grows by the amount of bytes of acknowledged data, while the LBC algorithm limits the congestion window growth to two segments. Studies [Allm98] have shown that LBC *byte counting* leads to throughput improvement while also slightly increasing the packet drop rate.

2.1.2.5 More Conservative Congestion Control

Because of the relatively high bit error rates over satellite channels many segments lost due to corruption will be falsely assumed to have been lost because of congestion, resulting in the sender going into *slow start* unnecessarily many times and consequently degrading performance. One possible method to mitigate this is to use more conservative congestion control by not employing *slow start* after assumed congestion but rather have the congestion window be reduced to half its current value (as is done in Fast Recovery) as opposed to a size of just one as happens with *slow start.* If congestion is still perceived to be continuing then the sliding window can be halved again and so on and so forth.

2.1.3 Congestion Avoidance

[SLSC98] It has been observed that during the *congestion avoidance* phase there is unfair sharing of bandwidth when multiple connections of varying RTTs exist over the same bottleneck link. More specifically, connections with large RTT values, such as those that traverse a satellite link somewhere along their path, have their congestion window grow too slowly. A proposed remedy is to have the window growth rate during *congestion avoidance* be increased so that it is larger than one segment per RTT. Two such proposed algorithms are the *Increase-by-K* and *Constant Rate* policies.

With the *Constant Rate* algorithm the congestion window will grow at a rate larger than one segment per RTT, with the rate depending on the RTT value. Two issues with this method are the correct estimation of the RTT and the proper selection of the constant growth rate.

The *Increase-by-K* algorithm adds K segments (instead of one) to the congestion window per RTT, for connections over a given RTT. The value of K and the RTT threshold values are still open issues. It has been shown that small values of K used when there is a relatively small number of connections over a bottleneck link improves throughput and fairness. However both of the above policies can lead to increase of the number of dropped segments and are not recommended for implementation in a shared network environment at this time since they violate the *congestion avoidance* algorithm outlined in RFC 2581.

2.1.4 Loss Recovery

2.1.4.1 Non-conservative Fast Recovery

Forms of fast retransmit have been implemented in previous versions of TCP such as Tahoe TCP and Reno TCP whereby upon receiving K duplicate acknowledgments (typically K=3) a lost segment is retransmitted without waiting for the retransmission timeout to occur. However, the recovery after a fast retransmit was either conservative (Reno TCP) or non-existent (Tahoe TCP). In newer TCP algorithms, such as NewReno TCP, an aggressive fast recovery is implemented whereby upon retransmitting a first lost segment (after receiving K duplicate ACKs), to fast retransmit a second lost segment K duplicate ACKs are not required to be received but rather just a single ACK. Therefore in cases of multiple lost segments algorithms with more aggressive fast recovery, such as NewReno TCP, have a smaller probability of resorting to a coarse timeout and thus resulting in better throughput in some cases. However studies [FaFl96] have shown that in some instances it is more advantageous to resort to a RTO timeout to recover from multiple lost segments. Implementing this type of more aggressive fast recovery requires modifications to the transmitter stack only and is allowed by RFC 2581 ("TCP Congestion Control", 1999). It has been observed that fast recovery algorithms also tend to create bursts after loss recovery and thus some sort of burst suppression algorithm, such as *Sender Adaption (SA)*, which limits the number of segments transmitted, may be useful.

2.1.4.2 Selective Acknowledgments

[FaFl96] With this type of mechanism instead of using the classical TCP cumulative acknowledgments, *selective ACKs (SACKs)* are used to identify which packets have arrived at the receiver thus giving the ability to the transmitter to know which specific segments to retransmit. A slight variation of this would be to use instead an explicit loss notification mechanism. One selective acknowledgments algorithm that has been studied and proposed is the *Fast Recovery SACK* algorithm, which is an extension to a normal fast recovery algorithm, whereby the receiver sends selective acknowledgment information to the transmitter. After a fast retransmit and after reducing the congestion window to half its size, the sender maintains a variable called a "pipe", which is an estimate of the number of outstanding TCP segments. The "pipe" variable is increased by one segment for each duplicate ACK that is received containing SACK information, and is decremented by one for each segment sent out (new or retransmitted). When the value of the "pipe" variable is lower than that of the congestion window size then a segment can be transmitted. This transmitted segment will be a new segment unless the selective acknowledgment information from the receiver indicates that there are retransmissions left to be carried out. The SACK algorithm has been shown in studies [FaFl96] to improve performance for the cases of 1 to 4 segments lost in a window of data and particularly in satellite environments but in some cases can lead to throughput degradation by causing bursts at the end of loss recovery. This algorithm requires modifications to both the sender's and receiver's stacks and is considered safe to be used in shared networks and has been allowed by RFC 2581 ("TCP Congestion Control", 1999).

2.1.4.3 Detection/Notification of Losses due to Congestion or Corruption

[RFC2481] [Floy94] The way TCP currently operates is to assume that every lost segment is due to congestion. While this may work well for the case of wired networks, in the case of satellite links and wireless networks in general this is significant cause for throughput degradation due to the fact that satellite (and wireless) links exhibit relatively high bit-error rates (BER) in the range from 10^{-7} to 10^{-5}. As a result a large number of the losses that are due to

corruption, and not congestion, will result in congestion mitigation taking place with the use of *slow start*, resulting in degraded performance. What is necessary is for better mechanisms to be put into place to detect if a loss is due to congestion or corruption, with just retransmission being required in the latter case. Such mechanisms are divided into two separate general categories: a) *Explicit Congestion Notification* where the sender is informed of congestion by either the receiver or an intermediate router and b) *Corruption Loss Detection.*

There are two types of Explicit Congestion Notification which are *Backward Explicit Congestion Notification (BECN)* and *Forward Explicit Congestion Notification (FECN).* With BECN a router will inform the sender of congestion, e.g. through the use of an ICMP "Source Quench" message. With FECN, routers will mark packets using a tag when congestion is imminent but will still forward the segment. The receiver upon getting the tagged segments will notify the transmitter. FECN requires modifications to both the sender's and receiver's stack and that routers tag segments. BECN requires modifications only at the transmitter.

With regards to Corruption Loss Detection, the receiver or a router that detect a corrupt packet could notify the sender of this so as to not assume the segment was lot due to congestion. However this is not a reliable method since the source address listed in the packet might also be corrupted and thus this method should not be used. A variation of this method that is more reliable is for a router to maintain a cache of recent destinations and when corruption is detected above a certain threshold level have the router send a corruption-experienced ICMP message to all destinations listed in the destinations cache. Then each destination would inform its respective transmitter through a TCP option of experienced corruption and each sender would refrain for some period from implementing congestion mitigation.

2.1.5 Multiple Parallel Connections

As mentioned previously at the beginning of this thesis, one of the problems with TCP connections over satellite links is the significant underutilization of satellite link bandwidth due

to the maximum window size limit of 64 KB and the large RTT values for GSO satellites. One way to mitigate this is to use multiple simultaneous TCP connections. With N connections the effective initial congestion window size is N, the window increases by N segments per RTT during *congestion avoidance* and in general the use of N simultaneous connections makes the sender N times more aggressive. Simulations have shown improved performance when utilizing multiple connections. This is an application layer modification requiring no stack migration. However this increased aggressiveness can lead to congestive collapse and it is not considered safe to use multiple connections in a shared network and should be limited for the time being to private networks only. To a certain extent the advantages that would be provided by multiple connections can be provided by using other methods that have been proposed in this report, such as larger initial window sizes. It should be noted that the method of utilizing multiple connections is already being used today by Web browsers, which typically use 4 simultaneous connections.

2.1.6 Header Compression

One method used to improve the efficiency of data transmission is to perform header compression of TCP and IP headers. Header compression is based on the idea that since a large portion of the information contained in the TCP and IP headers remains the same, or changes infrequently, or changes in a predictable matter during the life of a given connection, this information need not be transmitted in every header. Header compression algorithms have been proposed by [RFC2507], [DENP97] and [RFC1144]. According to these methods full TCP and IP headers are sent at the beginning including a session identifier that will be used to reference that particular connection. Later packets will simply contain the session identifier and any needed information that can't be derived from the full TCP/IP header sent initially which will be used as a template to derive all information that isn't contained in the compressed headers.

Header compression reduces overhead and can be of significant benefit for data

transmissions that have relatively small data payloads such as Telnet sessions and to a lesser extent WWW sessions. Some header compression methods can reduce TCP/IP header size from the typical 40 bytes down to 5 bytes in general and even 3 bytes for common cases. Header compression also has the added benefit of leading to a reduced rate of packet corruption due to the smaller packet sizes.

Header compression can be applied either end-to-end between the communicating endpoints or be applied transparently between the routers at the two edges of a satellite link. In the first case of end-to-end, header compression must be applied below the IP layer and be implemented at every intermediate router used in the connection and potentially also at the two communicating hosts. By implementing compression below the IP layer, header compression is transparent to routing by passing uncompressed headers to the IP layer. A more transparent way of implementing header compression is by performing compression only over a given satellite channel between the two routers (referred to from here on as *pivot-points*) at the edges of the satellite link. In the latter, case the compression should also performed below the IP layer and since compression is limited to over just the satellite link it requires modifications only at the two pivot-point routers and not at all routers along the path. In simulations, header compression has proven to be beneficial for cases with small packets sizes (larger overhead) and medium to low bandwidth links and/or links with relatively high bit error rates (such as is the case for satellite channels).

2.1.7 Sharing of TCP State Information

[RFC1379] [BaRS99] [RFC2140] There are many parameters in TCP which are set to initial values and later on modified during the duration of the connection such as the initial congestion window size for example. Changes have been proposed to these initial parameter values, however it is difficult to find one value that is suitable for all cases and environments. One proposition is to have the sharing of state information across TCP connections in the same environment. A characteristic example that shows the benefit of state information sharing is the

case of the initial congestion window size, where instead of starting from an initial window size of one the congestion window size could initially be set to the sustained "steady-state" window size used in previous connections over the same path. Sharing of state information could also be extended to the case of parameters that do not use preset initial values such as the round-trip time, Max Segment Size (MSS) etc. Sharing of state information could occur between connections originating from the same host or between hosts in the same subnet. Sharing state information across connections requires modifications to the sender's stack and possibly to receiver stacks also.

2.1.8 ACK Rate Control

In asymmetric networks where the bandwidth of the forward link is significantly higher than that of the return link, if the difference in links speeds is high enough then congestion will be exhibited in the return ACK link due to the high sending rate of the transmitter which will result in a large number of ACKs being generated and sent out by the receiver. In the case of 1500 byte data segments, given the typical 40 byte ACK packets (standard TCP/IP header size), congestion in the return link will be exhibited for bandwidth asymmetries of approximately 75:1, in the case that Delayed-ACKs are used, and 37:1 if ACKs are generated for each segment. One way to mitigate this problem is to implement rate control which will limit the sending rate of the transmitter, resulting in a reduced ACK rate. Such sender-rate limiting techniques are rate-to-window translation schemes where the receiver advertised window is modified by a router, thus forcing an upper limit on the transmitter sending rate (since the window used by the sender has as a maximum upper bound the receiver advertised window). However such sender-rate limiting techniques also lead to underutilization of the available bandwidth thus degrading throughput. A more efficient way would be to simply control the ACK rate directly rather than indirectly. Two such algorithms are *ACK Filtering* and *ACK Congestion Control.*

2.1.8.1 ACK Congestion Control

[KaVR98] With the ACK Congestion Control (ACC) algorithm, if a router along the reverse link path observes congestion then it will notify the sender of this. Then the sender will notify the receiver of the congestion in the reverse link, and the receiver will adjust its ACK rate accordingly using multiplicative backoff. When the receiver is not notified of congestion anymore then it resumes its normal ACKing policy. A problem with ACC is that since the ACKs are fewer and each ACK packet acknowledges more data, this will result in erratic sender window growth and thus the sender traffic becomes bursty. To mitigate this sender rate adaptation has to be implemented which will limit the max number of segments sent out regardless of window size. ACC requires modifications to both the transmitter and receiving TCP stacks as well as to routers. A variant of the above method would be to omit notification of the sender and receiver and simply have the router that detects congestion on the reverse link to perform a filtering/compounding of ACKs itself (e.g. if one ACK acknowledges up to N, and a second ACK acknowledges up to $N+k$, then these two ACKs cold be compounded into one ACK for $N+k$). However this still presents the same problem of causing bursty traffic and requiring sender adaptation. Furthermore because of the change to the ACKing policy it is recommended that ACC not be implemented in shared networks.

2.1.8.2 ACK Filtering

[BaPK97] A second algorithm for ACK rate control is to implement ACK Filtering. With this policy a router maintains a queue of ACKs that have passed through, and for each new ACK that comes the router scans the ACK queue to determine if the new ACK is redundant. If the ACK is redundant it is discarded. Thus ACK Filtering, as opposed to ACC, does not result in bursty sender traffic. Studies [BaPK97] however have shown performance to fall when using ACK Filtering alone and results in increased performance only when used in combination with ACK Reconstruction. With ACK Reconstruction, the router observes the ACK streams looking for large gaps. If a gap is observed then it constructs and inserts a new ACK(s) to "smooth out" the ACK stream. In studies [BaPK97] it has been shown that ACK

Filtering offers significantly better performance than ACC. Both ACK Filtering and ACK Reconstruction require only router modification and thus have the advantage of being applied transparently.

2.1.9 Error Recovery

One promising method to mitigate the problem of high error rates over satellite links is to implement reliability at the link layer over a given satellite channel. Caching of link layer frames would take place at the pivot-points (e.g. ground-station hardware) of a satellite link and any frames that where not received at the receiving pivot-point would be retransmitted by the sender pivot-point. The reliability would be based on an acknowledgment mechanism implemented at the link layer which would be based most likely on selective acknowledgments. A scheme like this has been proposed in [BaSa98]. The error recovery could be expanded further if for the case of packets that where not received at all (and thus not cached) at the first pivot-point, the pivot-point could trigger a fast retransmission from the TCP sender by sending multiple duplicate ACKs to receive the lost segment(s). Similar experimental techniques have been implemented in wireless systems where the performance increase was significant for higher error rates, in some cases more than doubling receiver throughput. This can be complemented with the use of error correction codes such as Reed-Solomon codes, BCH codes etc.

2.2 Vandermonde Matrix based Erasure Codes

2.2.1 Introduction to Erasure Codes

For years the telecommunications industry has used Forward Error Correction (FEC) techniques. FEC techniques try to prevent losses by producing redundant information from the data that it is to be transmitted, and appending this redundant data to the information that was to be originally transmitted. This redundant information is used to be able to reconstruct data that

has been corrupted during transmission, rather than just discard the data.

FEC techniques are generally based on error detection and error correction codes. Error detection codes, as their name implies, are used to detect data that has been corrupted. A well known example of an error correction code is the Cyclic Redundancy Check (CRC) code which is commonly used, for example by the IEEE 802.3 (Ethernet) link layer protocol. In the case of computer/networking communications, error detection codes are typically used at various layers, starting from the lower link layer protocols up to the higher layers such as at the transport layer where for example TCP and UDP implement checksums to verify the validity of received data segments. However use of error detection codes alone only offers us the ability to detect and discard corrupt packets. With the use of error correcting codes, such as Reed-Solomon codes, we can also recover and reconstruct corrupted data based on the redundant information included. While, as mentioned before, error detection codes have been and are widely used in networking communications, error correction codes have not been widely used. In the case of networking communications, the use of so-called *erasure codes* is of greater benefit rather than the use of error correction codes. Erasure codes are used not to reconstruct from corrupted data but rather to allow us to recover from *erasures* i.e. missing data. The reason for this is that in networking communications due to the widespread use of error detection codes at the various protocol layers which will result in any corrupted data being discarded, we deal with lost packets at the higher layers (e.g. application layer) and not corrupted packets.

Despite the apparent benefit and suitability of the use of erasure codes for networking communications and the general consensus of their usefulness [Rizz97], FEC techniques and erasure codes more specifically have not been used in Internet protocols. One reason for this is likely the fact the FEC techniques used in other areas, such as the telecommunications industry, have focused on error correction codes operating on relatively short strings of bits and running on dedicated hardware; while networking communications require erasure codes operating on relatively large packets of data that must be implemented efficiently in software [Rizz97].

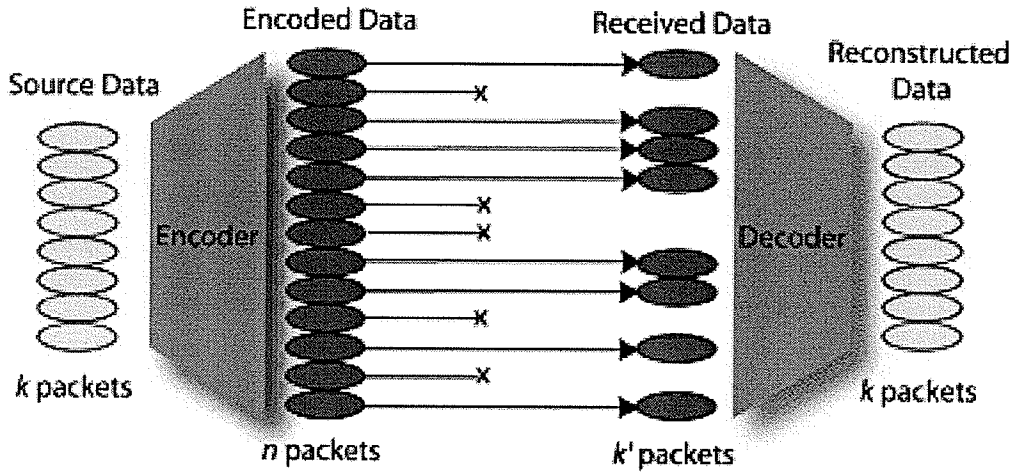

Fig. 2.1. Graphical representation of erasure code encoding/decoding process.

2.2.2 Linear Block Erasure Codes

Linear block codes are a category of erasure codes that are relatively simple and allow for an efficient implementation thus making them suitable for network communications applications. Linear block codes are named as such as they can be analyzed using the properties of linear algebra.

The concept behind erasure codes is that k blocks of source data can be encoded to produce n blocks that contain redundant information, with $n>k$, such that if any k blocks of data are received at the receiver the original k blocks of source data can be reconstructed. Such a code is called an *(n, k)* code. Fig. 2.1 provides a graphical representation of the encoding/decoding process of erasure codes. This differs from simply transmitting duplicate packets as erasure codes provide significantly greater efficiency and resiliency against losses. Regarding resiliency against losses, this is because in the case of transmitting duplicates it is required that at least one copy of each packet survive, while this is not the case with erasure codes as any k packets of the n transmitted, that are received suffice to reconstruct all source data. As for efficiency, in terms

of amount of redundant data produced and transmitted, sending duplicates requires that at least two copies of every packet are transmitted, which is not the case for erasure codes. To explain this better, let us consider wanting to send k packets of data. If using the duplicates method, at least $2 \times k$ packets must be sent to include any redundancy for each source data packet. If using an erasure code on the other hand, transmitting a total of even only $k+1$ packets provides some degree of redundancy/loss resilience for each packet of source data.

Let $\underline{x} = x_0, \ldots, x_{k-2}, x_{k-1}$ be the source data, i.e. each x_i is a packet of data (bit string) consisting of the same number of bits for all i. Given an appropriate $n \times k$ matrix G, a linear block erasure code *(n, k)* can be represented as:

$$\underline{y} = G \cdot \underline{x} \qquad (2.1)$$

where $\underline{y}$ is a $n \times 1$ vector representing the n produced encoded packets to be transmitted. As dictated by the erasure code properties, any k of the n transmitted packets/blocks that are received successfully will allow for the reconstruction of the original k blocks of data $\underline{x}$. More analytically:

$$\underline{y}' = G' \cdot \underline{x} \Rightarrow \underline{x} = G'^{-1} \cdot \underline{y}' \qquad (2.2)$$

where $\underline{y}'$ represents the subset of k packets of the number of packets (elements of $\underline{y}$) that were received successfully, and G' is the subset of rows of G that correspond to the elements of $\underline{y}'$ [Rizz97]. Fig. 2.2 gives a graphical representation of the encoding/decoding process in matrix form. For reconstruction of the source data additional information is also required that will have to be transmitted along with the encoded data, such as the number of source packets k, the

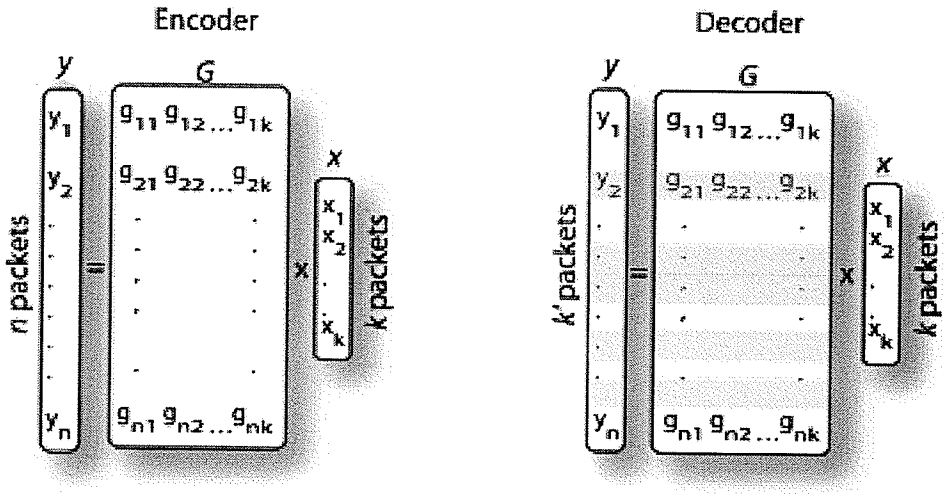

Fig. 2.2. Graphical representation of the encoding/decoding process in matrix form.

identity of each packet of data, the packet size etc. Additional overhead is incurred by the required precision for computations. More specifically, if packet x_i is represented using b bits and each element g_{ij} of G is represented using b' bits, then to represent each element y_i of $\underline{y}$, $b + b' + \lceil \log_2 k \rceil$ bits are required i.e. an overhead of $b' + \lceil \log_2 k \rceil$ bits for every packet in order that there be no loss of precision. Another very important issue is the size of the bit strings used to represent the data packets i.e. the elements of $\underline{x}$ and $\underline{y}$. More specifically, while in telecommunications each block of data is a small number of bits, in networking communications each packet of data is typically in the thousands of bits, e.g. a typical packet size of 1KB corresponds to 8,192 bits. Of course such a large number of bits can not be represented by the typical data structures available in programming languages with an integer for example being only 32 bits and a long integer being only 64 bits (typical size values). Therefore the issue is how to represent such large bit strings in order to perform the required matrix operations as specified above in (2.1) and (2.2) in a computationaly efficient manner.

In order to get around these problems *finite fields* must be used. A *field*, roughly speaking, is a set in which we can operate as on normal integers by adding, subtracting,

multiplying and dividing with the distinct difference that fields are closed under the operations of addition and multiplication, i.e. the result of summation and multiplication of elements of a field are also elements of the same field. *Finite fields* are fields that consist of a finite number of elements and most properties of linear algebra apply to finite fields. The usefulness of finite fields for erasure code implementation lies in the closure property of finite fields, meaning that no additional bits are required to represent the results of operations. Thus for erasure code implementation, the data elements can be mapped to field elements, with operations being performed on those field elements producing results that do not require additional bits for representation because of the closure property of fields, and then finally remap the resulting field elements to data elements [Rizz97].

Fields with p elements where p is prime are called *prime fields* or *GF(p)*, where *GF* stands for Galois fields. *GF(p)* is simply the set of integers from *1* to *p-1* under the operations of addition and multiplication modulo p. Fields with $q=p^r$ elements, where p prime, are called *extension fields* or *GF(p^r)*. Extension fields are of more use than prime fields for erasure codes because for p=2, operations on extension fields can become relatively simple also resulting in efficient implementations. The operations of interest to us are addition and multiplication as required by the matrix operations in equations (2.1) and (2.2). For extension fields, summation and subtraction become the same operation (bit-by-bit sum modulo 2) which is simply implemented with an XOR.

A property of extension fields is that the for every extension field there is at least one special element of the field, α, whose powers generate all the non-zero elements of that field, and the powers repeat with a period of *q-1* i.e. $\alpha^{q-1} = \alpha^0 = 1$. Thus, every non-zero element of the extension field x can be represented as $x = \alpha^{k_x}$. Since $x = \alpha^{\log_\alpha x}$, k_x can simply be considered to be $\log_\alpha x$. Therefore, for multiplication we have that:

$$x \cdot y = \alpha^{k_x} \cdot \alpha^{k_y} = \alpha^{(k_x + k_y) \bmod (q-1)} \qquad (2.3)$$

For more details on extension fields, and so as to not go beyond the scope of this thesis, one is referred to [Rizz97].

2.2.3 An Erasure Code based on Vandermonde Matrices

In section 2.2.2 it was stated that given an appropriate $n \times k$ matrix G, a linear block erasure code *(n, k)* can be represented as:

$$\underline{y} = G \cdot \underline{x} \qquad (2.1)$$

G is called the *generator* matrix of the code and in order for the properties of the erasure code to hold G must posses certain properties. Since G is an $n \times k$ matrix with rank k, any subset of k encoded blocks should convey information on all k source data blocks. Therefore, each column of G must have at most *k-1* zero elements per column. A simple and effective way to generate a generator matrix is by using matrices known as *Vandermonde* matrices which are of the form:

$$\begin{bmatrix} 1 & x_1 & . & . & x_1^{N-1} \\ 1 & x_2 & . & . & x_2^{N-1} \\ . & . & & & . \\ . & . & & & . \\ 1 & x_N & . & . & x_N^{N-1} \end{bmatrix} \qquad (2.4)$$

using as $x_i's$ elements of the *GF(p^r)* extension field. The determinant of Vandermonde matrices is given by:

$$\prod_{i,j=1\ldots k,\ i<j} (x_j - x_i) \qquad (2.5)$$

If all $x_i's$ are different then the determinant is non zero and thus the generator matrix is

invertible which is necessary for reconstructing the original source data according to equation (2.2). If all x_i are non-zero and the period q (as defined in section 2.2.2) is greater than the number of encoded packets n then up to $q-1$ rows can be constructed thus satisfying the properties for the generator matrix [Rizz97].

CHAPTER III

SYSTEM DESCRIPTION

3.1 Performance Enhancing Proxies

TCP was designed for general purpose links and thus is not optimal for specific types of links, and satellite links in particular, as analyzed in section 1.2. While enhancements and modifications to the TCP protocol might mitigate problems to a certain extent there are various issues:

- some problems are still not addressed by the plethora of TCP modification proposals, such as the high penalty of congestion mitigation techniques for packet losses on satellite links. Also, they may not provide such a substantial performance increase [BhBB99].

- this is a non-transparent solution and it would take years to get approved by standards bodies and implemented in the field.

- TCP stack/protocol modifications would make the protocol link-specific thus making it unsuitable for other types of links. A major design decision behind TCP was a simpler protocol design that ignored link characteristics. This of course was known that it would lead to suboptimal performance but was considered an acceptable tradeoff [BhBB99].

- would increase complexity of TCP implementations. Particularly not suitable for smaller embedded type devices [BhBB99].

Proxies can be used to separate links or groups of links with highly dissimilar attributes. These proxies can take advantage of their "knowledge" of the characteristics of the particular links and try to achieve closer-to-optimal performance, while isolating end hosts from these

details. Essentially isolating the end hosts from these dissimilar, in attributes, links that adversely affect end-to-end performance. This is also a transparent type of solution that requires no modifications to end-to-end protocols that can be readily implemented by placing the proxies in between the two communicating end hosts. The most suitable position for placement of proxies would be at the edges of the link to be "isolated" (in our case the satellite link) so as to intercept all the network traffic that is to traverse the satellite link, and would likely be integrated with the ground-station hardware. No TCP stack modifications are required at the two endpoints of the communication and this is of great importance and benefit. Performance enhancement is obtained at the expense of increasing somewhat the complexity in the network rather than at the end-user point [BhBB99]. Thus, this is a type of solution architecture that can be implemented much more readily and in a significantly more timely manner.

Two types of proxies architectures (commonly referred to as Performance Enhancing Proxies - PEPs) have been proposed in literature:

a) TCP spoofing proxies

b) TCP connection-splitting proxies

3.1.1 TCP Spoofing Proxy Scheme

TCP spoofing proxies (Fig. 3.1) would be placed at the edges of the satellite link, so as to be able to monitor all traffic that is to traverse the satellite link. A proxy would "spoof" the sending TCP end host by monitoring TCP segments sent by the transmitting TCP host and based on that traffic locally generate and send appropriately spaced TCP acknowledgments to the sending end host. This would result in maintaining a stable and open sliding window at the transmitting host giving it the illusion of a short path delay. The data sent by the sending TCP end host would route normally to the destined TCP end host, however acknowledgments send by the receiver would be intercepted (filtered) by the proxy as the acknowledgments have already been sent by the sender-side proxy as explained before (Fig. 3.1).

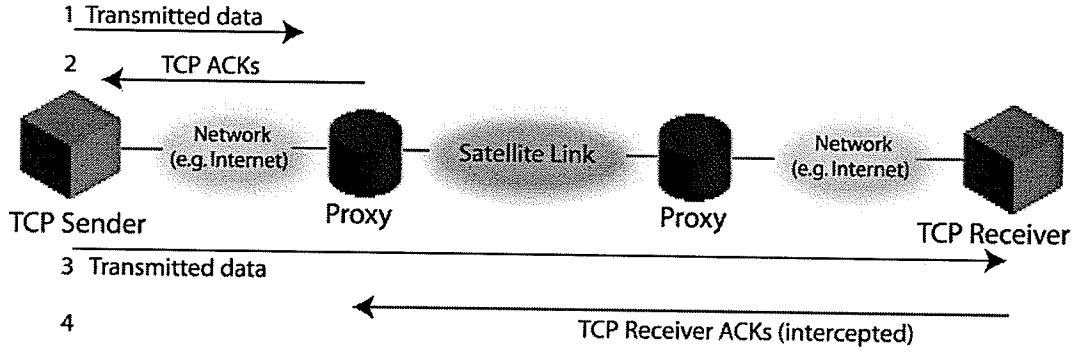

Fig. 3.1. TCP spoofing proxy architecture.

TCP spoofing will result in hiding the large satellite link propagation delay from the TCP end hosts and giving the illusion of a smaller delay path and thus accelerating sliding window growth and increasing bandwidth utilization at the end hosts since as stated in section 1.2.2

$$Max\ \ Throughput = \frac{Max\ \ \text{Re}ceiver\ \ Window\ \ Size}{RTT} \qquad (1.3)$$

Additionally, because each proxy acknowledges all data sent by its corresponding end host, they must also be responsible for the reliable delivery of that data to its destination and thus must cache the data and perform any necessary retransmissions. As a result the end hosts are also essentially isolated from the relatively high satellite channel bit error rate. On the other hand, TCP spoofing does nothing to mitigate the effects of bandwidth asymmetry.

Despite the apparent benefits there are also problems associated with the use of a TCP spoofing scheme. One problem is that symmetric paths are required in order to use TCP spoofing proxies i.e. that data and acknowledgments must flow along the same path through the proxies, otherwise acknowledgment filtering will not be possible. A second problem is the inability to work with encrypted IP datagrams as the proxies will not be able to read TCP headers to obtain the required information [PaSh97].

3.1.2 TCP Connection-splitting Proxy Scheme

The other type of performance enhancing proxy scheme is that of TCP connection-splitting proxies (also referred to as the Cascading TCP proxy scheme). This is the scheme the author's own variations and specific implementations of which will be studied and implemented in this thesis. TCP connection-splitting proxies would be placed at the edges of a satellite link (so that all traffic to traverse the satellite link will go through the proxies) with the purpose of transparently splitting the end-to-end TCP connection between two end hosts into three separate connections as shown in Fig. 3.2. Two TCP connections would exist between the end hosts and their respective connection-splitting proxies (i.e. the proxy on each end-host's side of the satellite link). The third connection would be between the two connection-splitting proxies. The connections between the end hosts an their respective proxies must be TCP connections as the end hosts use TCP. The middle connection segment between the two proxies however does not need to be TCP and another transport layer protocol can be chosen. As mentioned, the connection splitting is transparent to the two TCP end hosts.

The connection-splitting proxy on the side of the end host that is transmitting data at a particular instance, will intercept the sent TCP traffic and send acknowledgments back to the sending TCP host impersonating the intended receiver end host i.e. the source IP address used in the acknowledgments will be that of the intended receiver. The data intercepted by the proxy, let us call it the sender-side proxy, will be forwarded to the proxy on the other side of the satellite link (receiver-side proxy) using whatever transport-layer protocol has been implemented for that connection segment. The receiver-side gateway will then transmit the received data to the intended receiver end host, in this case not impersonating the original sending host though. Thus the receiving host will "think" that the entire TCP session was originally initiated by the receiver-side proxy. The receiving end host will acknowledge all received data by sending ACKs

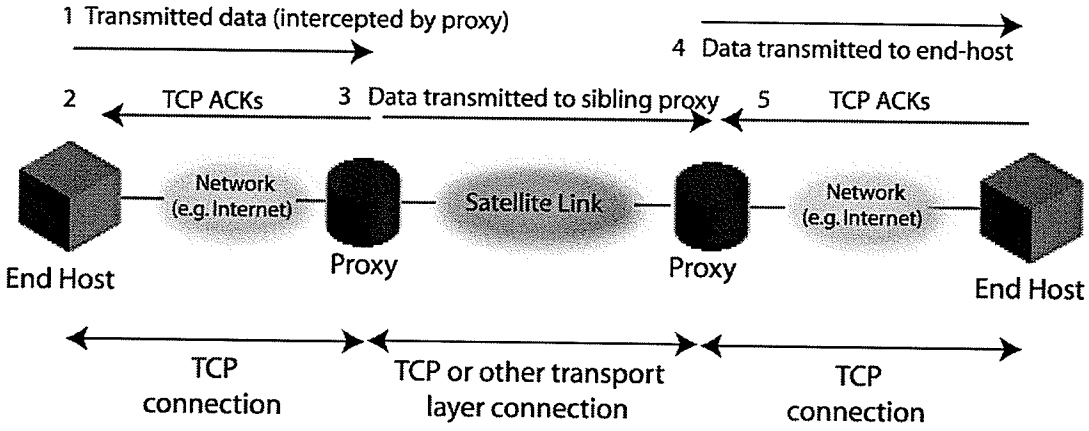

Fig. 3.2. TCP connection-splitting proxy architecture.

to the receiver-side proxy (Fig.3.2). Any data transmitted by the former receiving end host will be sent to, and acknowledged by, its proxy and the data will again be forwarded to the sibling proxy. The sibling proxy will then transmit this data to the former sending host impersonating, as before when acknowledging, the former receiver. From the aforementioned it can be seen how the end-to-end TCP session is broken into three separate connections.

The advantages of using the connection-splitting scheme and how it mitigates the adverse effects of satellite channel attributes on TCP performance, will be analyzed in detail for each connection-splitting architecture examined later on as the advantages vary depending on the type of implementation of the scheme.

As with TCP spoofing, the split-connection proxy scheme cannot work when encrypted IP datagrams are used [PaSh97]. However it does not require the use of symmetric paths, as TCP spoofing did, as each connection is terminated and there is no filtering of acknowledgments.

3.1.3 Terminology

A more consistent and coherent terminology that will be used from here on is that of *client*, *client-side proxy*, *server-side proxy* and *server*. *Client* is the initiator of a given TCP session and *server* is the intended receiver of the TCP session initiation request. *Client-side proxy* is the connection-splitting proxy on the *client* side of the satellite link and *receiver-side proxy* is naturally the proxy on the *server* side of the satellite link. Both of the proxies will be sender-side and receiver-side proxies at different times as the client and server respectively alternate between transmitting and receiving data. Also the terms proxy and gateway may be used interchangeably.

3.2 Single-TCP-Connection Connection-Splitting Performance Enhancing Proxies

The single-TCP-connection connection-splitting proxy scheme is a connection-splitting scheme, where the protocol used for the second connection segment to handle communication between the two gateways is the TCP transport layer protocol. One TCP connection is used between the two gateways for each end host TCP session. Fig. 3.3 shows a timing diagram for the connection-splitting procedure from initial connection setup to two-way data transfer and session close-down. As depicted in Fig. 3.3, when the client end host initiates a TCP session by sending a SYN segment (i.e. a TCP segment with the SYN flag set indicating a TCP connection setup request), the client-side proxy detects this session initiation request and responds with a SYN-ACK segment. The connection setup is completed with the client sending an ACK segment. Upon completion of the connection-setup between client and client-side proxy, the client-side proxy will then make a TCP connection-setup request to its sibling proxy, the server-side proxy, going through the same TCP three-way handshake procedure as detailed before. Finally, after the TCP connection-setup between the two proxies has completed, the server-side proxy will similarly send a TCP connection-setup request to the other end host (the server), and once this last connection is setup the 3-segment virtual connection between the two end hosts will exist.

As depicted in Fig. 3.3, exchange of data between the two hosts occurs in a similar way. Data sent by the client host will be intercepted by the client-side proxy which will acknowledge it. This data is then transmitted over the satellite link using the TCP connection established between the two gateways. Then of course the data received at the server-side proxy will be sent to the server host over the TCP connection setup between them. The proxy will not perform any host/address impersonation since, as mentioned before and repeated so as to avoid any confusion, the TCP connection was setup between the two and the server host believes all data to originate from the server-side proxy. For data originating from the server end host, the procedure is the same with the only difference that the data transmitted by the server host does not need to be intercepted by the server-side proxy as it is sent to it directly.

Session tear-down is dissimilar to session setup. Because the proxies are protocol-agnostic, i.e. they are not customized to interoperate with any specific protocol, this means that they also do not know when a particular connection session between two end hosts will terminate. Thus, the proxies rely on the client and/or server end hosts to terminate the session. More specifically, when an end host notifies that it is closing the TCP connection, by sending a FIN TCP segment, the connection between the host and its proxy terminates. The other end host might also terminate the connection with its proxy or not. In any case, the closing of at least one TCP connection between a host and its corresponding proxy will result in the TCP connection between the proxies to be closed. This subsequently will cause the shutting-down of the TCP connection between the other end host and its proxy, if not already done so.

One fact that should be noted regarding the implementation of all connection-splitting proxy schemes in this thesis is that while normally the client-side proxy must intercept all information sent by the client end host to the server end host (i.e. TCP connection setup requests, data and connection tear-down requests), having the client-side proxy respond by impersonating the server end host, the client end host actually addresses all data to the client-side proxy. Doing this is necessary due to implementation constraints imposed by the available resources. More specifically, having the client-side proxy transparently intercept all information sent by the client

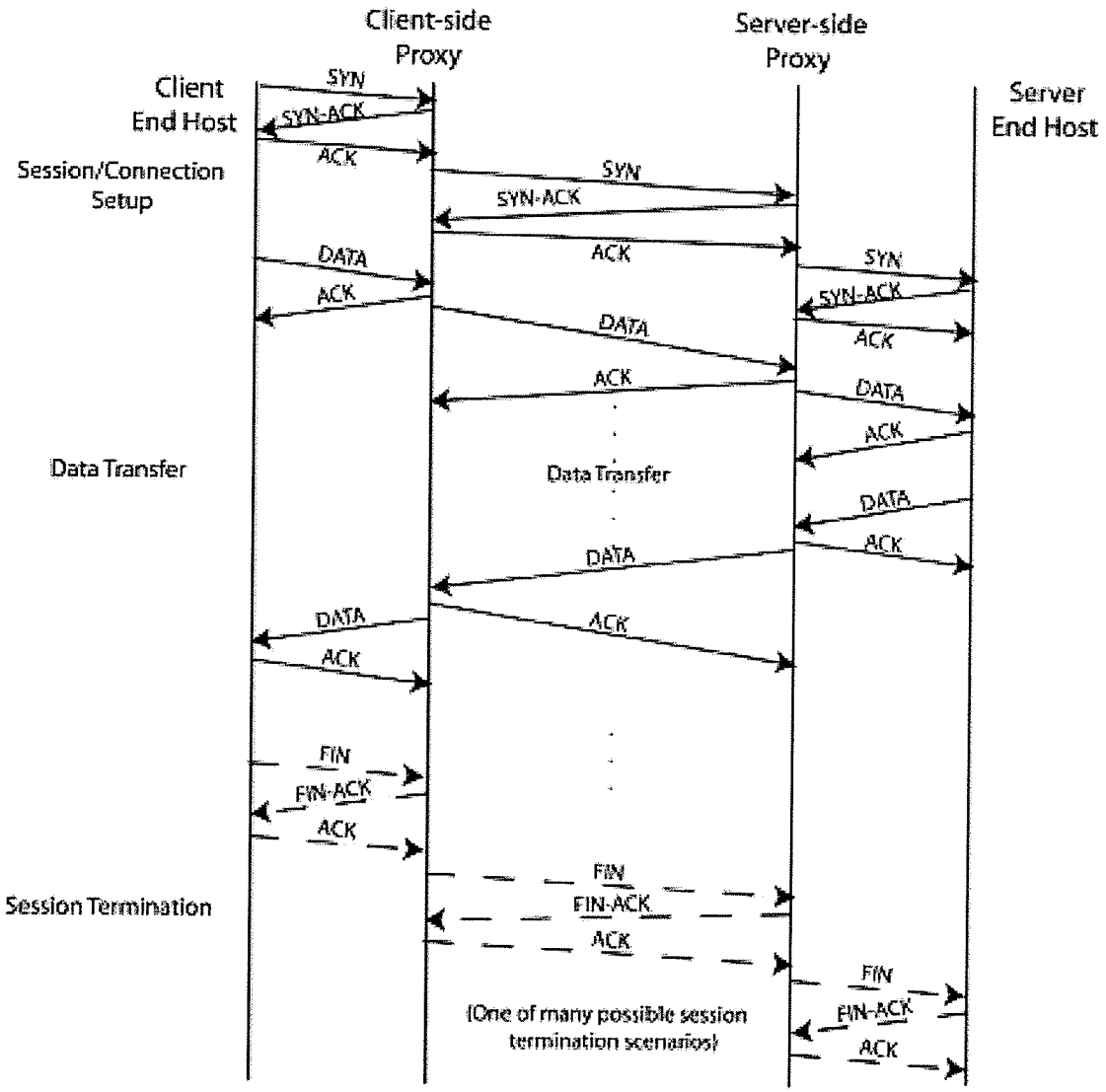

Fig. 3.3. Timing diagram for single-connection TCP connection-splitting proxy architecture.

to the server end host would require the use of either a network interface card operating in promiscuous mode or having the proxies operate on workstations that function as routers for the network. The first was not possible due to security concerns and performance issues associated with promiscuous mode NIC operation, and the second was not feasible. This small, needed modification in the operation of the connection-splitting scheme should not affect the performance of the connection-splitting proxy architectures in any significant way at all.

Another consequence of this is that port information is not maintained (in order to contact the server at the intended port address) as the client addresses the proxy directly rather than the server.

Single-TCP-connection connection-splitting will result in hiding the large satellite link propagation delay from the TCP stacks of the end hosts. This results in giving the illusion of a smaller delay path and thus accelerating sliding window growth at the end hosts as the acknowledgments are coming at a faster rate. Additionally, increasing bandwidth utilization at the end hosts is possible for the same reason. The end hosts are also isolated from the relatively high satellite channel bit error rate as any data sent by the hosts has been acknowledged and retransmissions required because of losses over the satellite channel are the responsibility of the proxies. This will allow the sliding windows of the end hosts to grow at a more normal rate rather than have unnecessarily deployed congestion mitigation techniques reduce them. However the "hiding" of the large propagation delay and the high bit error rate of the satellite channel from the end hosts, and the benefits that this incurs, is offset to a certain extent by the fact that both the satellite channel propagation delay and high bit error rate will still affect the transmission of data from one proxy to the other as they also use a TCP connection that will of course be affected by these factors. Finally, single-TCP-connection connection-splitting does not reduce the number of acknowledgments that will traverse the satellite link and thus does not

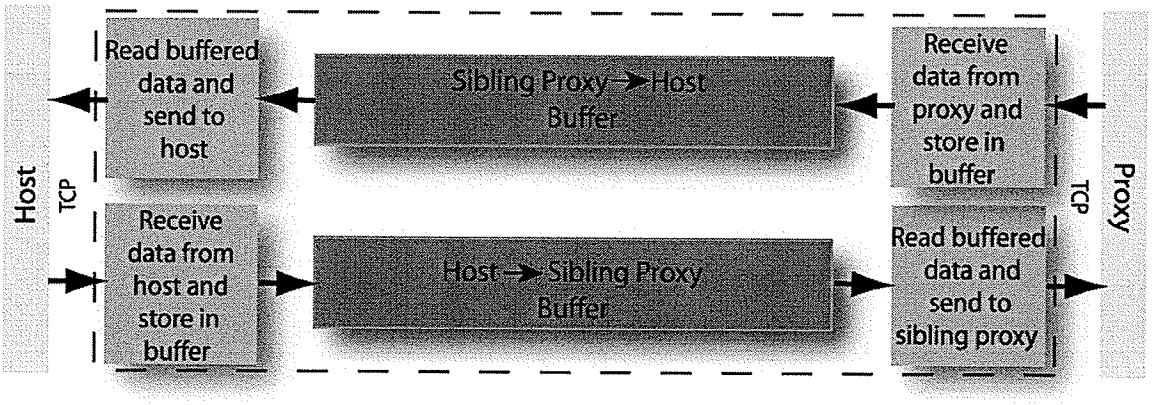

Fig. 3.4. Program structure abstract model for single-connection TCP connection-splitting proxy architecture.

mitigate the effects of bandwidth asymmetry.

The proxies are implemented as concurrent servers with the ability to handle multiple sessions between any number of end host pairs. For each session between a pair of hosts, a child process is created to handle the given session. Fig. 3.4 shows an abstract model of the program structure that depicts the operation of the proxies. Each child process that is created uses a multi-threaded architecture. More analytically, each child process employs four threads that are used to receive and transmit data: a pair of threads are used to transmit and receive data to/from the one end host and the other pair of threads is used to transmit and receive data to/from the sibling proxy on the other edge of the satellite link. Data received from either the host or sibling proxy is placed in the corresponding buffers, while data to be transmitted is obtained from the same respective buffers where it was placed by the receiving threads. There is of course a variable delay due to buffering at the proxy and it is important to utilize buffers large enough so as to prevent congestion at the proxy and consequently a large delay that will adversely affect performance.

During the lifetime of the session a "back-pressure" algorithm is used to implement flow control across the three connections. In other words, it is not possible for the ingress data rate into the proxy to be larger than that of the egress data rate otherwise there will be congestion at the proxy, leading to performance degradation. The "back-pressure" algorithm takes care of not having the ingress data rate significantly exceed the egress. This "back-pressure" algorithm is not implemented explicitly but is an inherent property of using multiple TCP connections in series. More analytically, as a receiver of data (whether it be a proxy or an end host) starts getting congested it will start reading data from the network at a slower rate thus resulting in the system network receive buffers having reduced free space. Therefore, leading to a smaller receiver advertised window which will lead to the slower growth of the senders sliding window and consequently to a reduced rate of transmission. The buffers are necessary in order to compensate for any small or temporary significant differentials in the ingress and egress data rates at the proxies. The proxies were implemented in the C programming language and run on the Sun Solaris operating system.

3.3 Multiple-TCP-Connection Connection-Splitting Performance Enhancing Proxies

The multi-TCP-connection connection-splitting proxy scheme is similar to that described in the previous section with the difference that multiple TCP connections are being used between the two proxies for each end host TCP session. One must note that this is significantly different than simply using multiple TCP connections between the two end hosts for two reasons: a) using multiple connections end-to-end would require new versions of all applications and thus is not a transparent solution b) many, including the IETF through its RFCs, do not consider it safe to use multiple connections across a shared network, such as the Internet, for fear of congestive collapse. With the multi-connection proxy solution, the use of multiple connections is limited to just the satellite link and not over any shared network segment. Fig. 3.5 shows a timing diagram for the multi-connection connection-splitting architecture from initial connection setup to session termination. When the client end host initiates a TCP session by sending a SYN segment, the client-side proxy detects this connection setup request and responds with a SYN-ACK segment, and the three-way handshake is completed with the client sending an ACK segment. Upon completion of the connection-setup between client and client-side proxy, the client-side proxy will then make a TCP connection-setup request to its sibling proxy going through the same TCP three-way handshake procedure as detailed before. Once the first TCP connection between the two proxies has been established, the client-side proxy will use this connection to transmit to the server-side proxy the total number of TCP connections to be opened between the two for the given session. Following this, the advertised number of connections is opened between the two gateways. The number of TCP connections to be shared between the two proxies for a given session is a configurable value. Finally, after the TCP connection-setup between the two proxies has completed, the server-side proxy will similarly establish a TCP connection with the server end host, completing the session setup between the two end hosts.

Exchange of data between the two hosts occurs in a similar way. Data sent by the client host will be received by the client-side proxy which will acknowledge it. This data is then

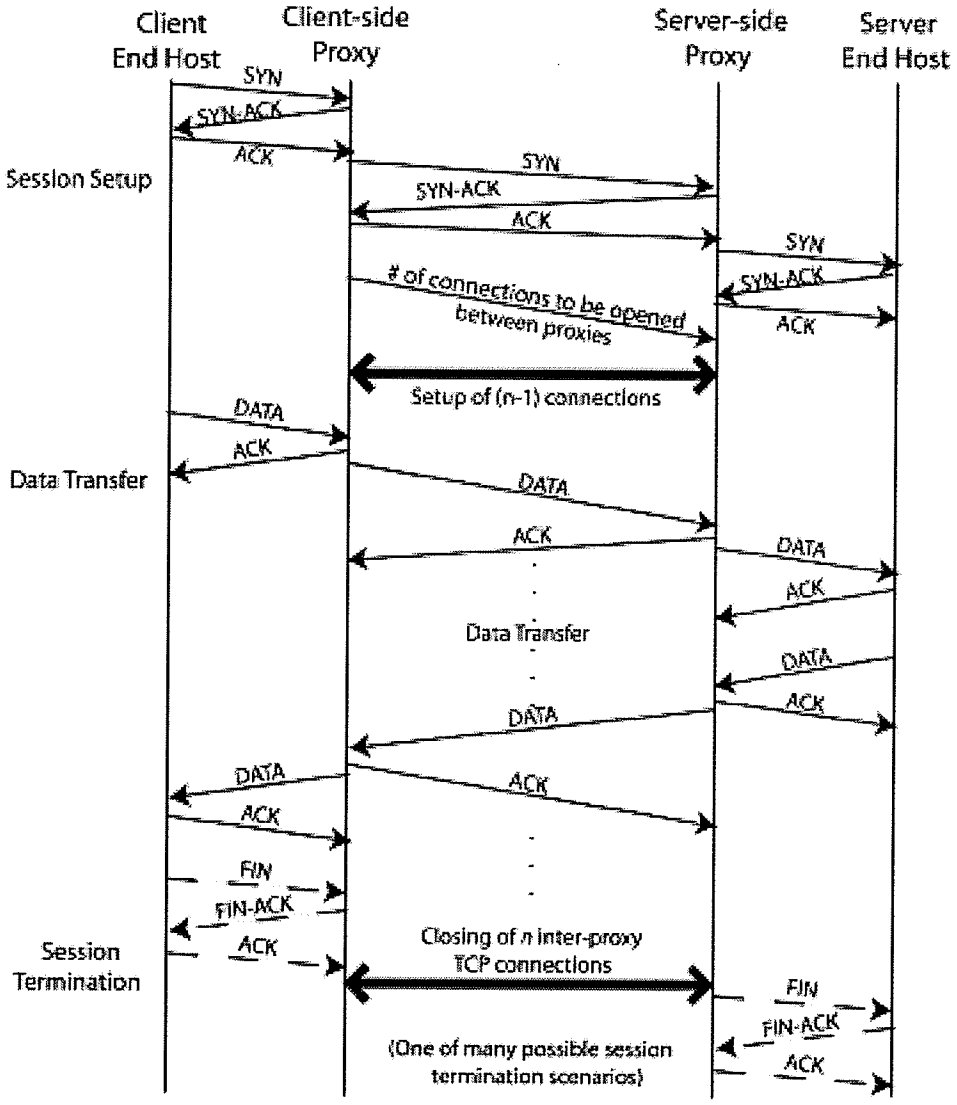

Fig. 3.5. Timing diagram for multi-connection TCP connection-splitting proxy architecture.

transmitted over the satellite link to the sibling proxy. The multiple TCP connections between the two proxies are utilized by transmitting each packet of data over the multiple connections in a round-robin manner, i.e. one packet is sent over a given connection, the next packet through the next TCP connection and so on and so forth. More on this will be explained later on when examining proxy operation below. The data received at the server-side proxy will be then sent to the server host over the TCP connection setup between them. Data transmission from server to

client follows the exact same procedure.

Session tear-down is identical to that for single-TCP-connection connection-splitting, with either (or both) end hosts closing their TCP connection to their proxy, causing the proxies to close the multiple TCP connections between them. This will subsequently lead to the closing of the TCP connection between the other host-proxy pair, if not done so already.

Multi-TCP-connection connection-splitting, as with the single-connection scheme, will result in hiding the large satellite link propagation delay and bit error rate from the TCP stacks of the end hosts. The first, results in giving the illusion of a smaller delay path and thus accelerating sliding window growth at the end hosts. The latter allows for increased bandwidth utilization and improved sliding window growth, as explained in section 3.2. However, as also applies and was mentioned for the single-TCP-connection scheme, the hiding of the large propagation delay and the high bit error rate of the satellite channel from the end hosts is offset to a certain extent by the fact that both the satellite channel propagation delay and high bit error rate will still affect the transmission of data from one proxy to the other as they also use a TCP connection that will of course be affected by these factors. The multiple TCP connections are used to mitigate these adverse effects. By using N connections between the proxies we are N times more aggressive, having essentially an initial sliding window of size N as opposed to one, with a N times larger effective sliding window. In addition, it is more resilient to packet losses since from a given loss only one sliding window of the N will be affected by the congestion mitigation policies. Finally, multi-TCP-connection connection-splitting does not reduce the number of acknowledgments and thus does not counter the effects of bandwidth asymmetry.

The proxies are implemented as concurrent servers with the ability to handle multiple sessions. The program structure is quite different from that of the single-TCP-connection proxies, an abstract model of which showing proxy operation is shown in Fig. 3.6, given that they must now handle multiple TCP connections for each session. Each packet of data received

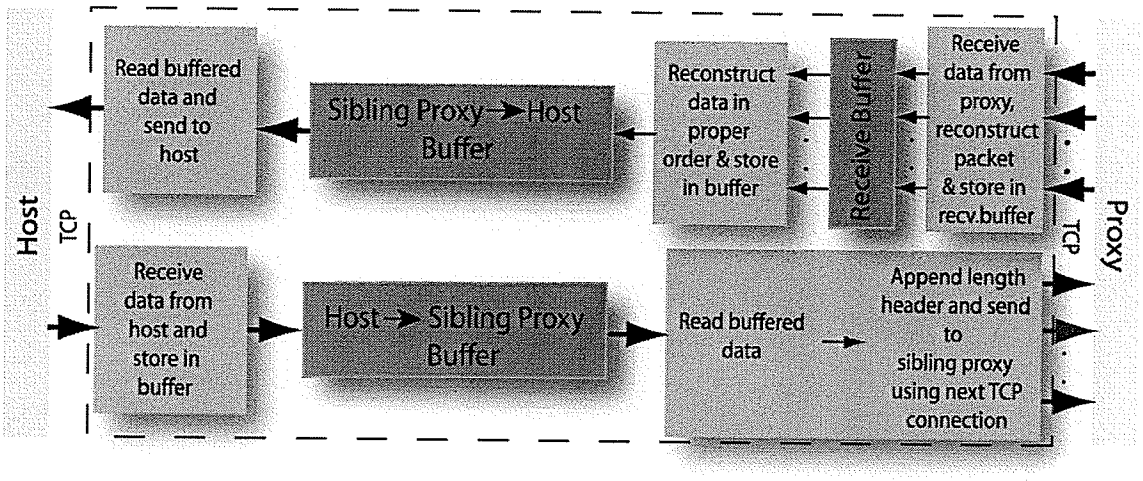

Fig. 3.6. Program structure abstract model for multi-connection TCP connection-splitting proxy architecture.

by a proxy is transmitted over the multiple connections in a round-robin manner, i.e. one packet is sent over a given connection, the next packet through the next connection etc. At the receiving proxy this data must be collected from the multiple connections and reconstructed in the correct order. In order for this to be possible, because of the byte stream property of TCP (i.e. the fact that packet/segment boundaries may not be maintained), packet boundaries must be known so as to be able to reconstruct the data from the multiple connections correctly. This is done by appending to the beginning of every transmitted packet a 2-byte header containing the length of the packet, from which the receiver is able to distinguish between individual packets (as when transmitted) and thus reconstruct the data in proper order. The data is collected from each individual connection by a separate thread which also performs the above aforementioned tasks, and once a full packet has been obtained it is copied to the receive buffer, as shown in Fig. 3.6. Data placed in the receive buffer is collected according to the order of transmission and placed in order in the single-column buffer from where data is read and sent out to the end host. Data received from an end host goes through the reverse procedure, by being placed in a buffer by the host-side receiving thread, from where the packet is read and then the length header is appended. Finally, the packet is transmitted over the appropriately selected TCP connection.

During the lifetime of the session the "back-pressure" algorithm is in effect, used to implement flow control among the three connection segments. The multi-connection proxies were implemented in C and operate on the Sun Solaris operating system.

3.4 Single/Multi-UDP-Connection Connection-Splitting Performance Enhancing Proxies

This new connection-splitting proxy scheme follows the same concept as the connection-splitting schemes examined in the previous sections with the difference that UDP is used to implement a protocol for the exchange of data between the proxies as opposed to TCP, whether it be one or more connections for each session handled. The timing diagram for the UDP connection-splitting scheme detailing the procedure from initial connection setup to session termination is given in Fig. 3.7. A TCP connection is first established between the client end host and the client-side proxy in a manner identical to that detailed for the previously examined schemes. Upon completion of the connection-setup between client and client-side proxy, the client-side proxy will then initially establish a TCP connection with the server-side proxy. This TCP connection is used by the client-side proxy to transmit to the server-side proxy the total number of UDP connections to be opened between the two for a given session, and this information will be used by the server-side proxy to open the appropriate number of UDP sockets. The TCP connection is not closed at this point as it will be used to transmit congestion control information as will be detailed later on. The host-to-host virtual connection is then completed by having a TCP connection established between the server-side proxy and the server end host.

Any data sent by an end host will be received by its corresponding proxy which will acknowledge it. This data is then transmitted over the satellite channel to the sibling proxy. The multiple UDP connections between the two proxies are utilized by transmitting each packet of data over the multiple connections in a round-robin manner as with the multi-connection TCP scenario. The data received by the sibling proxy will be then sent to the other end host over the

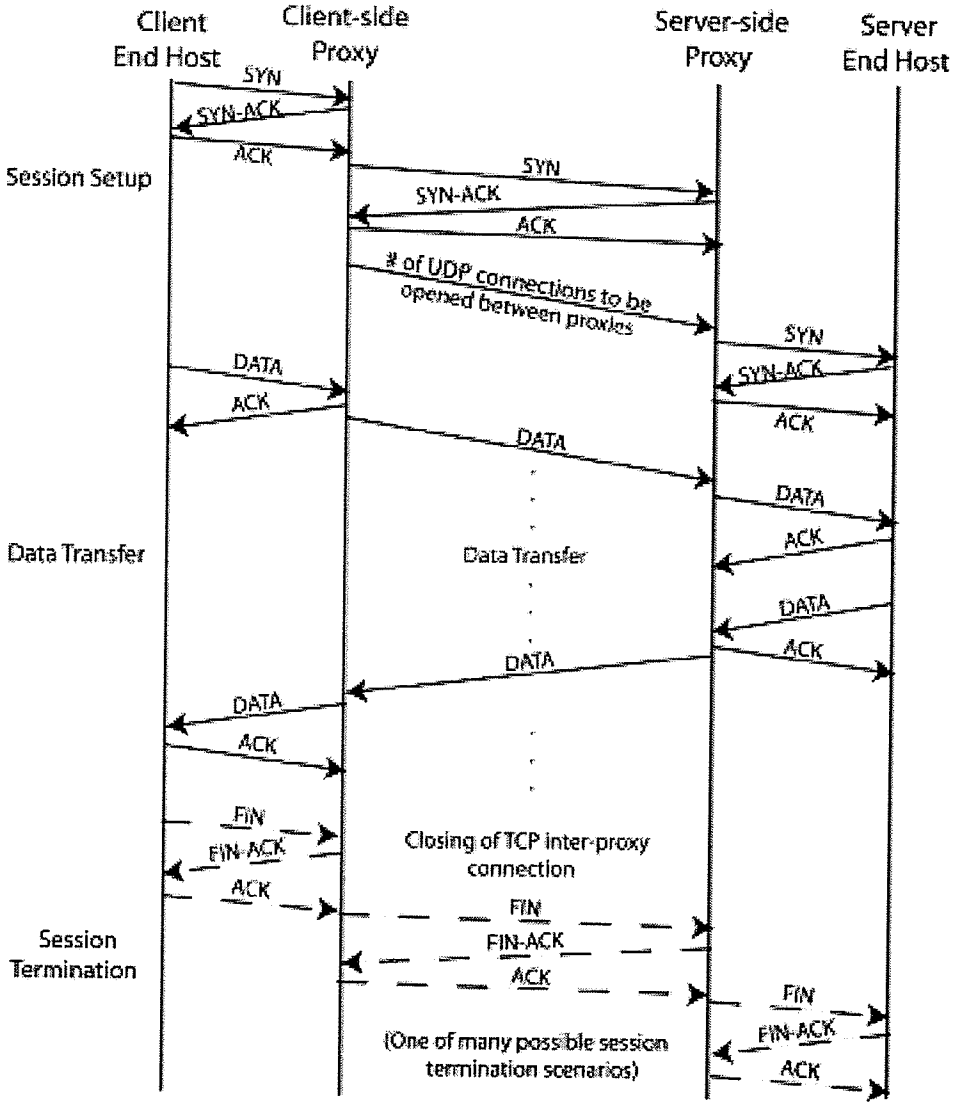

Fig. 3.7. Timing diagram for UDP connection-splitting proxy architecture.

TCP connection established between them.

Session tear-down takes place with either end host closing its TCP connection with its proxy, causing the proxies to close the single TCP connection between them (UDP connections are not closed as there are no UDP connections given that UDP is non-connection oriented datagram protocol). This leads to the close-down of the TCP connection between the other

proxy and end host (if not already done so).

UDP connection-splitting, as with both previously examined schemes, will result in hiding the large satellite link propagation delay and bit error rate from the TCP stacks of the end hosts. The significant difference though with the two previous TCP schemes, is that by using UDP for inter-proxy data exchange, the large propagation delay and the high bit error rate of the satellite channel will affect the transmission of data from one proxy to the other to a much lesser extent than it will the TCP schemes. This is because UDP does not use a sliding window for transmitting data, who's growth depends on the arrival of (highly delayed) acknowledgments, and who's size will be affected by packet losses due to high satellite bit error rate. This makes the UDP approach significantly more resilient to the adversely affecting satellite channel attributes. One drawback however of using UDP is that there is no built-in congestion control and no mechanism to ensure reliable delivery of data, as TCP provides, for the communication between the two proxies. Thus a congestion control mechanism must be implemented in addition to a mechanism for ensuring reliable data transfer.

For reliable data transfer use of erasure codes is made. Erasure codes, as examined in section 2.2, can be used to encode data and by using redundancy allow for the full reconstruction of all source data from less packets than the total amount of packets transmitted. Thus, by using a sufficient degree of redundancy it is possible to achieve reliable transmission of data over a lossy link without the need of an acknowledgment mechanism. The erasure code used is a computationaly efficient Vandermonde matrix based erasure code described in [Rizz97], developed, and with the source code written by, the author of the aforementioned paper.

Congestion control is implemented by transmitting *RNR* (Receive Not Ready) messages to the sibling proxy when a proxy's buffers occupancy percentage exceeds a certain threshold value. The RNR message will result in the proxy that receives the RNR message to cease transmitting temporarily. After an RNR message has been transmitted, once the proxy's buffers

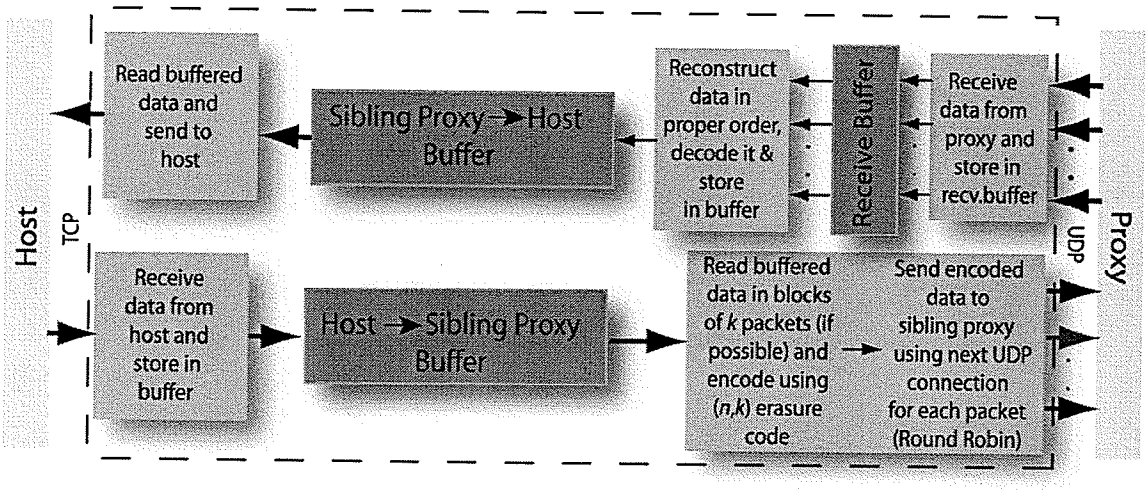

Fig. 3.8. Program structure abstract model for UDP connection-splitting proxy architecture.

occupancy percentage falls below a threshold value, a *RR* (Receive Ready) message is transmitted to the other proxy allowing it to commence transmitting again. The threshold value used for triggering a RR message transmission is smaller than the threshold value used for triggering a RNR message transmission so as to introduce a hysteresis and thus avoid a "ping-pong" effect of continuously transmitting in alternating order RNR and RR messages. The RNR and RR messages are transmitted over the TCP connection that was established between the proxies to transmit the connection number at the beginning of the session in order to insure the guaranteed delivery of the congestion control messages.

Another advantage of the UDP scheme is that because of the lack of an acknowledgment mechanism, there are of course no acknowledgments at all and the only traffic in the reverse direction channel are occasional RNR and RR messages. This leads to a very effective mitigation of bandwidth asymmetry over the satellite link segment.

Flow control is not implemented for the inter-proxy connection(s) as it is not required since the path between the two proxies is a simple point-to-point connection over a single "bent-pipe" satellite link [BhBB99].

An abstract model of the program structure for the UDP connection-splitting proxies is shown in Fig. 3.8 and depicts how the proxies operate. Data received from an end host is placed in a buffer by the host-side receiving thread. A separate thread is used to collect a block of packets from the buffer, which are then encoded using the erasure code. For a *(n, k)* code, k source packets will be encoded to produce a block with a total of n packets and any k will suffice for source packet reconstruction at the receiver. Collecting and encoding multiple packets at a time is done because using blocks of just one packet results in the erasure code just producing duplicate. This would also be inefficient as the ratio of total encoded packets to source packets will be at least 2, while better ratios can be achieved using multiple packets at a time. The number k of packets collected and encoded, as well as n, can be varied. The proxy will try to use blocks of k packets, however if not enough packets are available within a specified amount of time it will proceed with less so as to avoid delays. It is necessary along with the encoded packets to also transmit additional information needed to reconstruct the data at the receiving proxy. This information must include the total number of encoded packets of a packet block and the number of source packets which will indicate the number of packets needed to reconstruct the original data at the receiving proxy. Also, since there will be some lost packets, each encoded packet must carry an identifier, a sequence number, so as to be able to identify each individual packet and derive how many and which packets have been lost which is also information needed in reconstructing the data.

An 8-byte header containing all this and additional needed information is appended to every encoded packet transmitted and is shown in Fig. 3.9. The first and last bytes of the header are beginning and end-of-header indicator bytes. The second and third bytes of the header indicate the number of source data packets and the total number of encoded packets of the packet block of which the current packet is a member of. The fourth byte of the header contains the sequence number while the next two bytes contain the length of the current packet (including the header). Finally, the seventh byte indicates the position of the packet within its packet block.

After a block of packets have been encoded, the header containing all the necessary

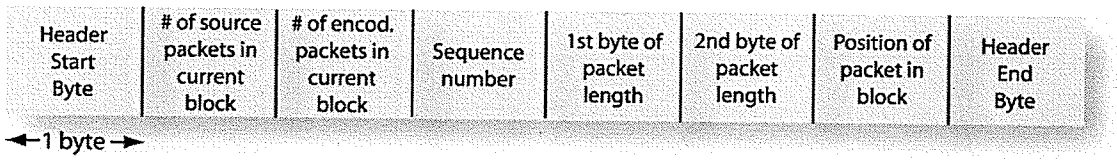

Fig. 3.9. UDP proxy architecture packet header.

information is appended to each encoded packet and the individual packets are transmitted over the single connection. Or if multiple UDP connections exist, the packets are transmitted in a round-robin manner over the different connections just as with the multi-TCP-connection scheme. In the UDP connection-splitting scheme there is no need for packet reconstruction as with the previous two TCP-based proxy architectures as, unlike TCP, UDP is a datagram and not a byte stream protocol and thus all transmitted segments will be received exactly as transmitted. The packets received by the other sibling proxy will have their sequence number checked in order for the receiving thread to determine if there have been any lost packets and if so will set a skip flag in the appropriate receiving buffer cells indicating that these packets have not been received. The received packet will be written in the receive buffer in the cell that corresponds to its sequence number. A separate handling thread will read the packets from the receive buffer, ignoring buffer entries that have the skip flag set. Once enough packets from a packet block have been obtained (for a *(n, k)* code, packet blocks will consist of n packets and any k suffice for source packet reconstruction) the packets are sent for erasure code decoding and the original source packets are obtained. Fig. 3.10 depicts the (simplified) flow of execution from reception of packets to erasure decoding. The reconstructed source data packets are then written to the single column buffer, as shown in Fig. 3.8, from where they will be of course read and sent out to the end host.

During the lifetime of the session the "back-pressure" algorithm is in effect, implementing flow control across the three connection segments. The UDP connection-splitting proxies were implemented in C and operate on the Sun Solaris operating system.

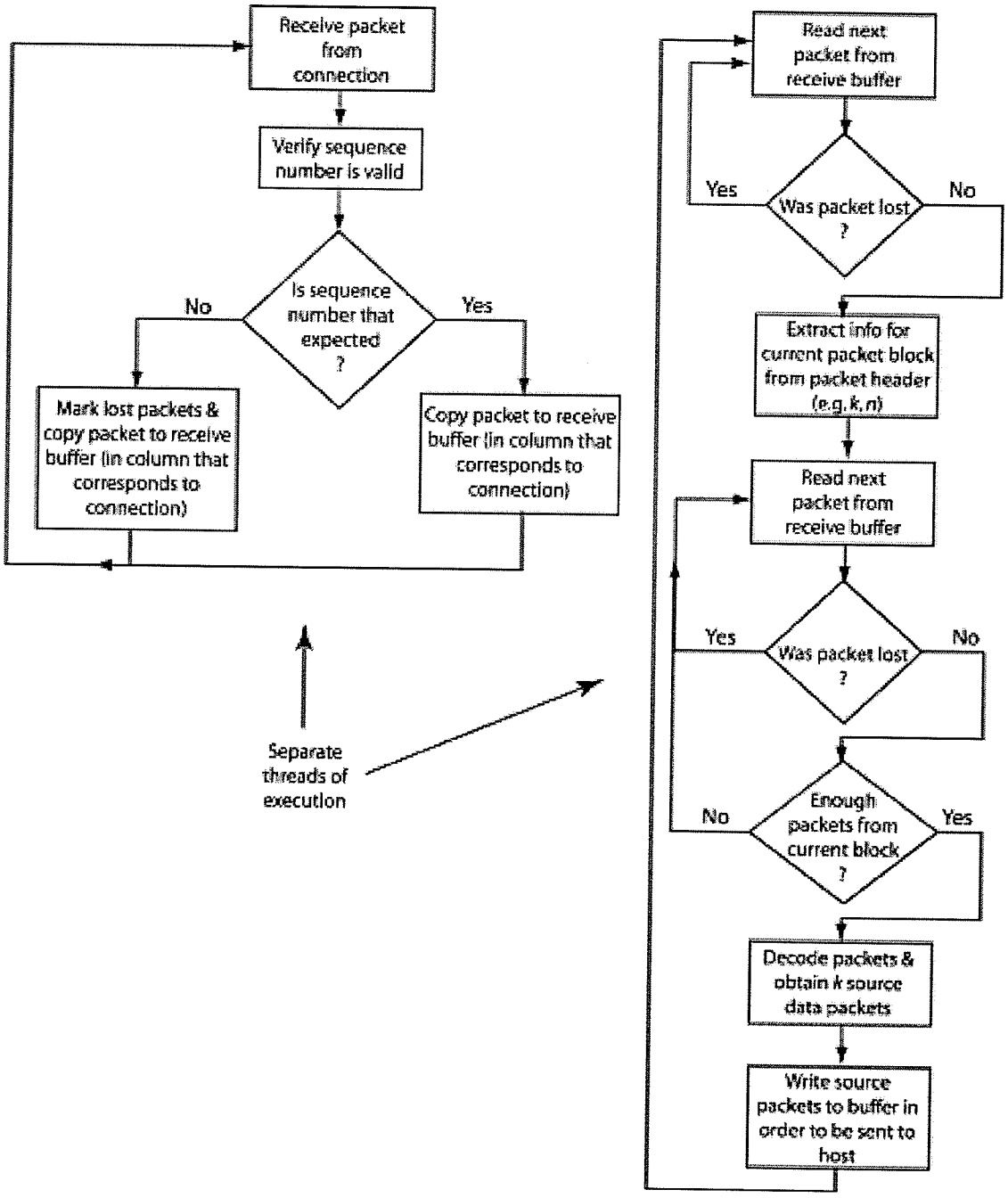

Fig. 3.10. Simplified flow of execution from reception of packets to erasure decoding.

CHAPTER IV

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Test Configuration and Methodology

The test configuration used for conducting all experiments is shown in Fig. 4.1. The current implementations of the connection-splitting proxies run on the Sun Solaris 5.8 operating system, with Sun Ultra 10 workstations being used as the hardware platform. The same software and hardware platform is used to run the client and server applications. The client and server applications used implement a proprietary file transfer protocol implemented by the author of this thesis [Phil00]. This file transfer protocol uses TCP to transfer files in blocks, with 10^6 byte blocks being used for the experiments conducted and analyzed in this chapter.

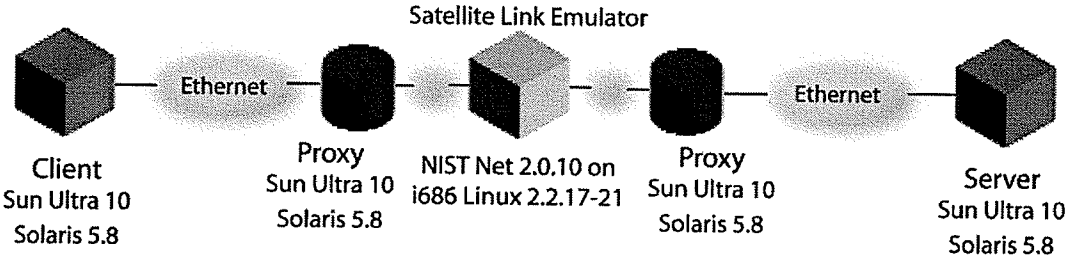

Fig. 4.1. Test configuration.

To emulate the satellite link, the NIST Net 2.0.10 link emulation package [NIST00] was used running on the Linux operating system (kernel version 2.2.17-21) on Intel i686 hardware. The setup was configured such that the satellite link-emulating Linux workstation was the default gateway for all data exchange between the two proxies. The NIST Net 2.0.10 link emulation package was used to implement a 255 msec one-way link delay and a 10^{-7} and 10^{-6} bit error rate, corresponding to a typical GEO satellite link one-way propagation delay and bit

error rates respectively. A typical GEO satellite channel bit error rate is closer to 10^{-7}, with the 10^{-6} bit error rate value also being tested in order to identify the effects of larger bit error rates on throughput performance.

All workstations used in the experiments were on the same LAN connected via 100 Mbps Fast Ethernet links, with the server application being tuned to implement a small time delay in order to limit server (and thus overall) data throughput to approximately 500 KB/sec. This was done in order to reflect more realistic throughputs typical of Internet and remote connections as opposed to the very high LAN speeds. Performance tests were executed measuring the time required for the transfer of a 3.8×10^6 byte file. Please note that the notation of MB used in the graphs and tables in this chapter refers to "millions of bytes" and not MBytes as in 2^{20} bytes. A second note is that all UDP proxy scheme tests use a (4, 2) erasure code unless explicitly stated otherwise.

4.2 Obtained Results and Analysis

Fig.4.2 shows the results obtained from multiple tests of each scheme, for the transfer of a 3.8 MB file utilizing an emulated satellite link with a 255 msec one-way delay and an 10^{-7} bit error rate. It compares the performance of the reference case of *straight TCP* (i.e. direct TCP connection between client and server with no proxies) versus that of the single-connection TCP proxy scheme and the single-connection UDP proxy scheme using a (4, 2) erasure code. The results show a very significant performance advantage of the UDP proxy scheme, which seems to outperform straight TCP by a factor of over four. The single-connection TCP proxy scheme on the other hand does not perform too well and seems to perform at the levels of straight TCP. The results show a rather significant variation in the transfer times for straight TCP and the single-connection TCP proxy scenario. On the other hand, the UDP proxy scheme seems to be very consistent in regards to the download times. This variation in transfer times for straight TCP and the single-connection TCP proxy scheme is likely due to the effects of the bit error

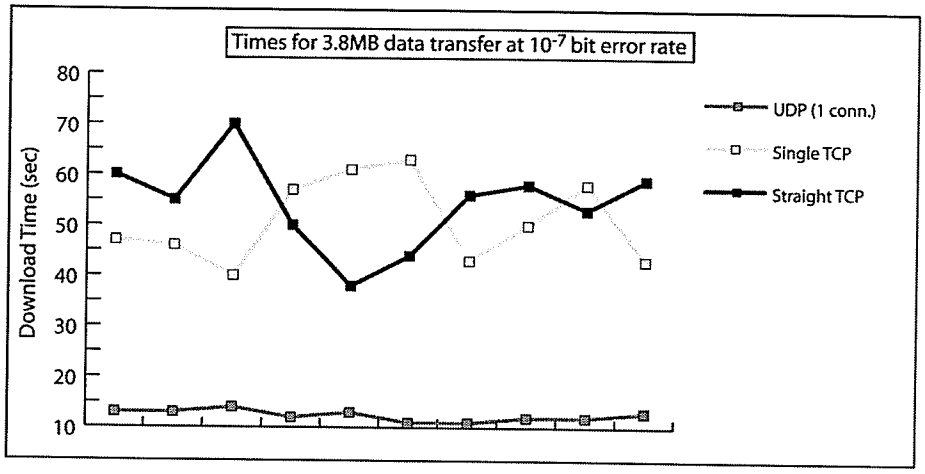

Fig. 4.2. Transfer times for 3.8 MB file at 10^{-7} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs Single-connection UDP proxy scheme.

rate (i.e. packet loss) on the growth of the TCP sliding window.

It is believed that the exceptional performance of the UDP proxy scheme is due to the fact that because UDP, unlike TCP, does not use a sliding window it is largely unaffected by the large link propagation delay and relatively high bit error rate. Large propagation delay and high bit error rate lead to, as explained analytically in Chapter 1, slower sliding window growth due to large delay before ACKs arrive, frequent shrinking of the sliding window due to congestion mitigation mechanisms being used every time there is packet loss, bandwidth underutilization due to a small maximum sliding window and large link delay etc. While the TCP proxy scheme does theoretically isolate the end host TCP stacks from the satellite link's large delay and bit error rate, the TCP connection used by the proxies for transfer of data between them is affected to the fullest extent by these satellite link attributes with the analogous adverse effects on performance, thus decreasing end-to-end throughput. With the UDP proxy scenario on the other hand, packet losses will not cause it to reduce the sending rate of the proxies, nor are acknowledgments expected in order to increase the transmission rate between proxies, at the same time isolating the TCP end hosts from the adversely affecting satellite channel attributes.

Fig. 4.3 depicts the download times for the same set of conditions (i.e. file size, link delay

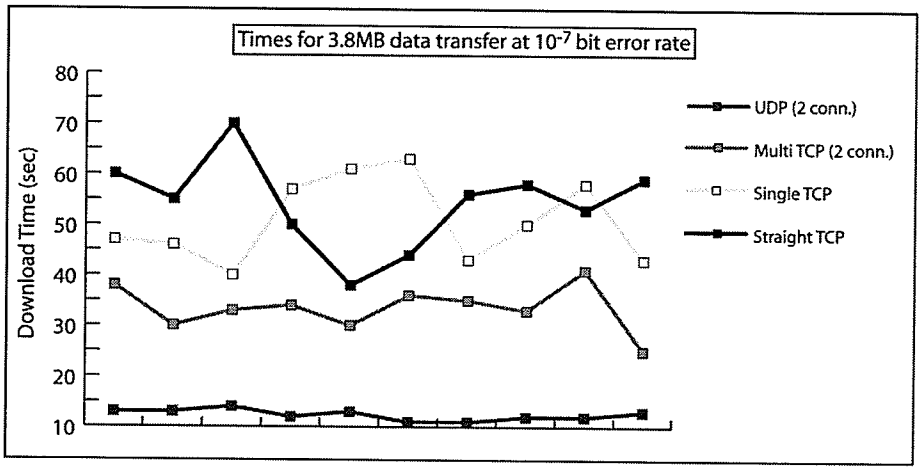

Fig. 4.3. Transfer times for 3.8 MB file at 10^{-7} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs 2-connection TCP & UDP proxy schemes.

and error rate) comparing the 2-interproxy-connection TCP and UDP schemes versus straight TCP and the single-connection TCP proxy scheme. It is observed that the UDP proxy scheme performance is at the same levels as with the single-connection UDP proxy scheme examined previously far outperforming straight TCP. While the 2-connection TCP proxy scheme also shows a distinct performance advantage over straight TCP as opposed to the single connection case. Fig. 4.4 shows the results obtained for using 3 inter-proxy connections for both the UDP and TCP proxy schemes. Again the UDP proxy times remain at the same levels as with a single and two connections and far lower than any of the other times. The fact that the use of 2 or 3 inter-proxy UDP connections did not increase performance further is in all likelihood due to the fact that the server throughput is not enough to saturate a single UDP connection and thus increasing the number of UDP inter-proxy connections further will not yield any performance improvement. Performance of the 3-connection TCP proxy scheme is at the same levels as that of the 2-connection TCP proxy scheme, with both yielding a significant improvement in performance over straight TCP and the single-connection TCP proxy case. It is apparent that the use of multiple inter-proxy TCP connections helps improve the performance by being more aggressive. An interesting fact is that the use of 3 TCP connections provides only a marginal improvement in download times compared to 2 connections, despite still being significantly

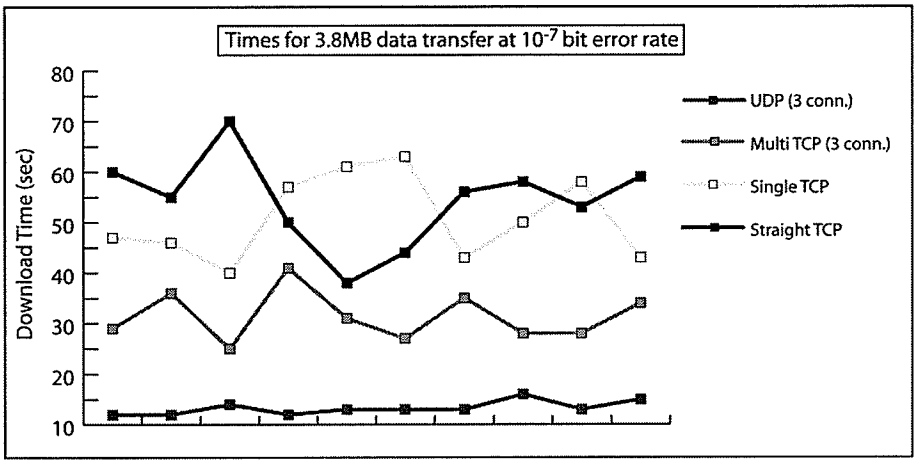

Fig. 4.4. Transfer times for 3.8 MB file at 10^{-7} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs 3-connection TCP & UDP proxy schemes.

higher than the UDP proxy architecture download times i.e. despite there being more room for higher end-to-end aggregate throughput rates. From the above one draws the conclusion that the TCP proxy architecture has reached its upper bound due to the limitations of the TCP protocol and how adversely it is affected by the satellite channel attributes.

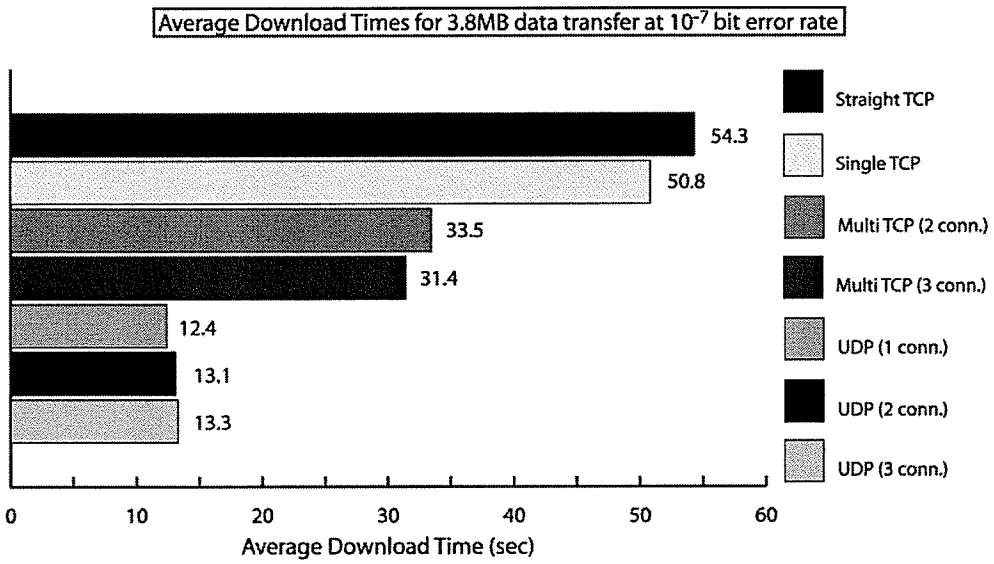

Fig. 4.5. Average transfer times for 3.8 MB file at 10^{-7} bit error rate.

Fig. 4.5 summarizes the performance of all schemes giving the average transfer times while Table 4.1 lists a summary of the average transfer times and the percent improvement in throughput.

Connec-tions	Straight TCP	Single connection TCP	Throughput increase (%)	Multi-connection TCP	Throughput increase (%)	UDP	Throughput increase (%)
1	54.3 sec	50.8 sec	6.9 %			12.4 sec	338 %
2	(54.3 sec)			33.5 sec	63 %	13.1 sec	315 %
3	(54.3 sec)			31.4 sec	73 %	13.3 sec	308 %

Table 4.1 Average transfer times and the percent improvement in throughput for 10^{-7} bit error rate.

The experiments were performed again for all cases, but using this time an increased satellite link bit error rate of 10^{-6} in order to observe the effects of an increased error rate on the performance of all schemes. The experiments are exactly the same as those performed before for the lower bit error rate with the difference that the multi-connection TCP proxy scheme was also tested for the case of 4 inter-proxy connections as there seemed to be a noticeable

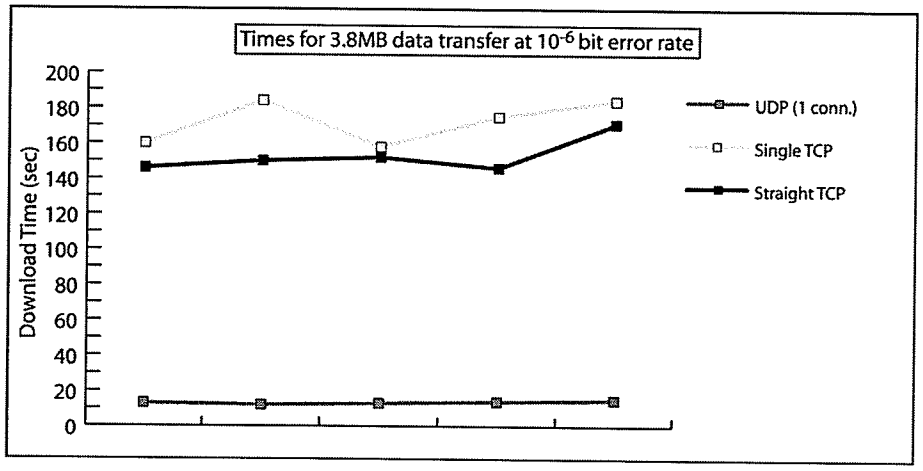

Fig. 4.6. Transfer times for 3.8 MB file at 10^{-6} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs Single-connection UDP proxy scheme.

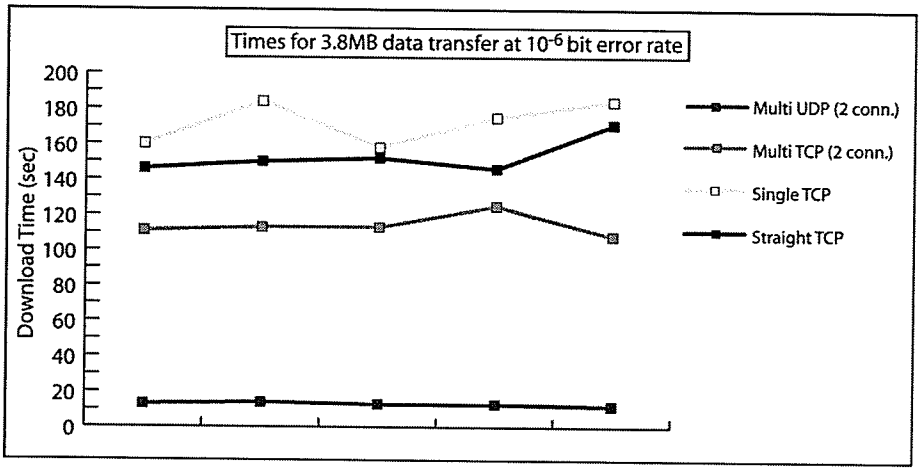

Fig. 4.7. Transfer times for 3.8 MB file at 10^{-6} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs 2-connection TCP & UDP proxy schemes.

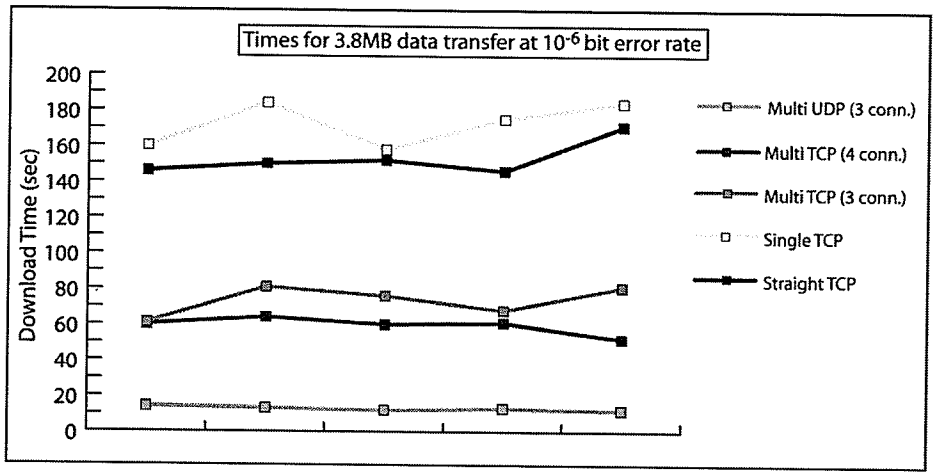

Fig. 4.8. Transfer times for 3.8 MB file at 10^{-6} bit error rate for Straight TCP vs Single-connection TCP proxy scheme vs 3-connection UDP proxy scheme and 3 & 4 connection TCP proxy scheme.

improvement this time with increasing the connection number. The obtained results are shown in Fig. 4.6, Fig. 4.7 and Fig. 4.8.

A concise summary of the obtained results is given in Fig. 4.9 and Table 4.2 which show

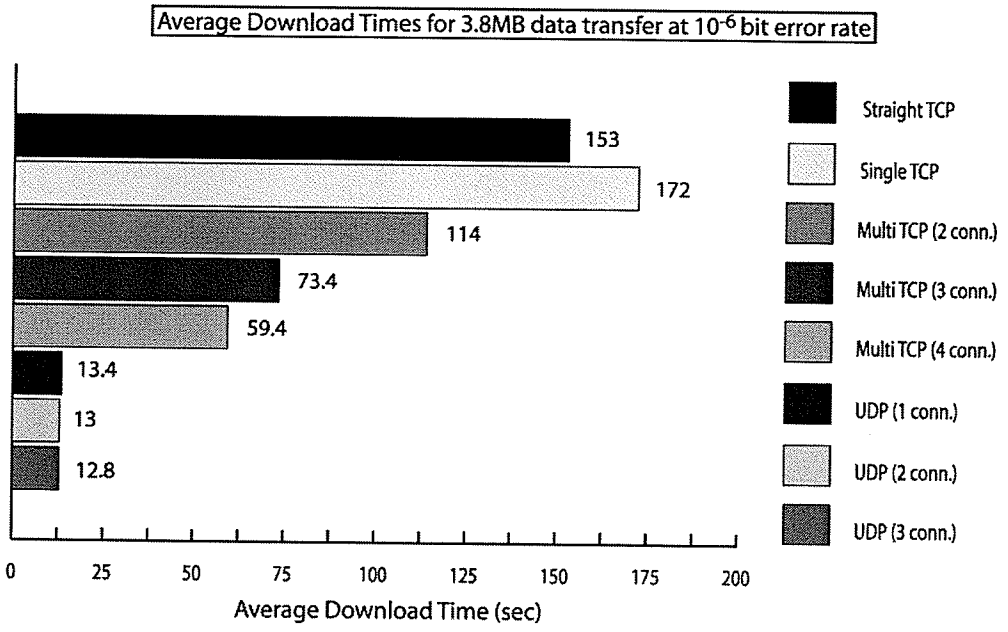

Fig. 4.9. Average transfer times for 3.8 MB file at 10^{-6} bit error rate.

the average download times and the performance improvement achieved by the various proxy architectures, with Table 4.2 also comparing with the results for utilizing the 10^{-7} bit error rate.

By observing the results in Table 4.2, it is noticed that for the new increased bit error rate there is almost a tripling of the download time for straight TCP. Undoubtedly a result of the more frequent retransmissions required and particularly the increased use of congestion mitigation resulting in more frequent "shrinking" of the TCP sliding window. This increase in download times is also observed for both TCP proxy schemes with the reasons for this being the same as for the straight TCP case. What is of interest is that the single-connection TCP proxy scheme now performs worse than straight TCP going from an average 6.9% performance improvement to an 11% decrease.

Another important fact regarding the multiple-connection TCP proxy scheme is that now there is a significant increase in performance by increasing the number of connections utilized as opposed to the marginal increases obtained when using the lower error rate. Also these

10^{-7} / 10^{-6} Connections	Straight TCP	Single connection TCP	Throughput increase (%)	Multi-connection TCP	Throughput increase (%)	UDP	Throughput increase (%)
1	54.3 sec / 153 sec	50.8 sec / 172 sec	6.9 % / -11%			12.4 sec / 13.4 sec	338 % / 1041 %
2	(54.3 sec) / (153 sec)			33.5 sec / 114 sec	63 % / 34.2 %	13.1 sec / 13 sec	315 % / 1077 %
3	(54.3 sec) / (153 sec)			31.4 sec / 73.4 sec	73 % / 108 %	13.3 sec / 12.8 sec	308 % / 1095 %
4	(54.3 sec) / (153 sec)			59.4 sec	158 %		

Table 4.2 Average transfer times and the percent improvement in throughput for 10^{-6} bit error rate.

performance improvements are of greater magnitude for an increased number of connections. This is explained by the fact that the increased bit error rate also has a greater impact on TCP performance (as analyzed above) thus providing more room for improvement which is realized by utilizing a greater number of inter-proxy connections. For the UDP proxy scheme on the other hand, it is observed that the download times have remained largely unchanged indicating that the increased bit error rate had no effect. This is apparently due to the fact, as noted in the previous round of experiments, that the UDP proxy scheme does not utilize a window and the inter-proxy transmission rate is largely unaffected by the bit error rate. As a result the UDP proxy scheme exhibits rather exceptional performance improvements of an order of magnitude.

The UDP proxy scheme was also tested for varying redundancy ratios in order to observe the effects of increasing the erasure code redundancy on performance. Fig. 4.10 and 4.11 depict the results obtained by varying the erasure code redundancy from a (3, 2) erasure code, to (4, 2) and (5, 2), for both bit error rates of 10^{-6} and 10^{-7}. From the figures it can be seen that the variation of the redundancy ratio for either bit error rate, and for utilizing a different number of connections, has small effect on performance. This is very likely due to the fact that the aggregate throughput is not high enough to stress the UDP proxies and any modest increase

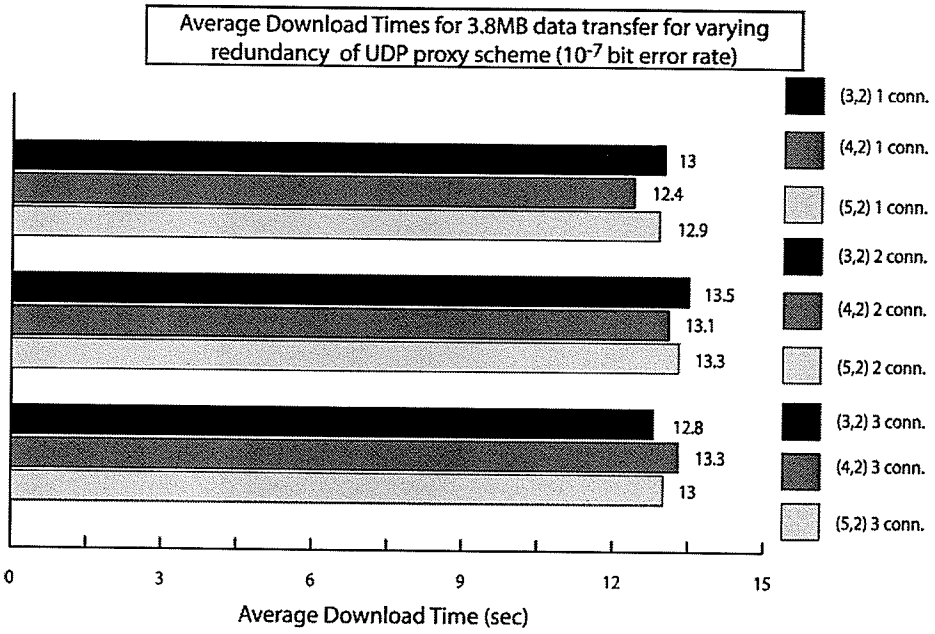

Fig. 4.10. UDP proxy schemes for varying redundancies at 10^{-7} bit error rate.

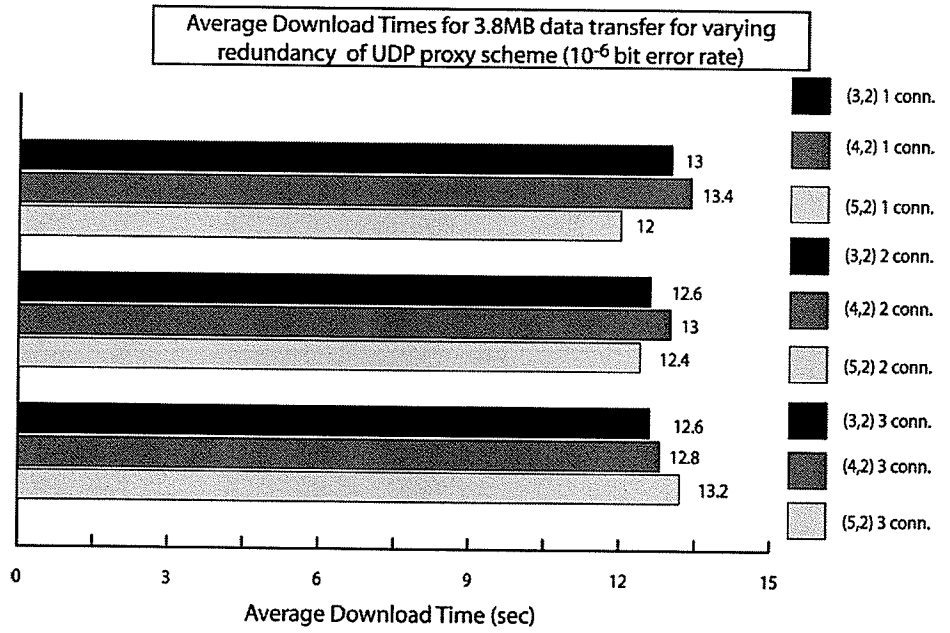

Fig. 4.11. UDP proxy schemes for varying redundancies at 10^{-6} bit error rate.

of redundancy does not affect a significant change.

Two important facts must be noted regarding the UDP connection-splitting architecture. Firstly, a bug in the erasure code open source code prevents it from reconstructing the data properly when running on the Solaris operating system, and this I was not able to correct in time. The erasure code coding and decoding still run normally with the correct number of bytes being transferred thus leaving the performance benchmarks unaffected and valid, with the simple difference that the received at the end host data while correct in amount is not the same byte-for-byte as the data transmitted. A second note is that during the testing of the UDP proxy scheme, in some instances (when using a single inter-proxy connection and very rarely when using miltiple connections) the data transfer would fail due to relatively rare burst losses that resulted in not enough packets being available to reconstruct the source data packets from a given packet block. This, and the fact the in practice the presence of rain fade can result in similar burst or extended losses dictate that for a real-world implementation the UDP proxy scheme must be complemented with a retransmission mechanism to recover from such losses. While using multiple connections and/or increased redundancy can mitigate the effect of extended losses, a retransmission mechanism is still deemed necessary. One was not implemented due to time constraints, but it is proposed that an explicit loss notification (or "negative" acknowledgment) mechanism be implemented rather than a "positive" acknowledgment mechanism. I.e. there would not be an acknowledgment for every packet or group of packets received, but rather only when there is an insufficient number of packets to reconstruct the source data packets would a retransmission request be sent for the packets of that given block. The use of explicit loss notification rather than a "positive" acknowledgment mechanism would result in far less traffic in the reverse channel (important when bandwidth asymmetry exists), be less complex and pose a smaller burden on the proxies. This is one of the recommendations for future work as noted in the next chapter.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

The work described in this thesis was motivated by the need for improving the performance of TCP over satellite channels which suffers to a large degree. In this thesis three connection-splitting architectures based on transparent proxies were implemented and tested. The connection-splitting architectures implemented and the performance improvements they yielded are as follows:

5.1 Single-TCP-Connection Connection-Splitting Performance Enhancing Proxy Architecture

This connection-splitting architecture utilizes two transparent proxies that use a single TCP connection for inter-proxy data exchange. The performance enhancements obtained at the typical GEO satellite bit error rate of 10^{-7} where marginal yielding only a 6.9% improvement in throughput on average compared to standard straight TCP. At an increased bit error rate of 10^{-6} the performance of the architecture is worse actually leading to a decrease in throughput performance (of 11% on average) compared to straight TCP. While the use of proxies does isolate the end-hosts from the satellite link attributes that adversely effect TCP performance, the adverse effects of the satellite link on inter-proxy data exchange is the major pitfall of this scheme. Thus, this architecture proved to be ineffective.

5.2 Mutli-TCP-Connection Connection-Splitting Performance Enhancing Proxy Architecture

This proxy architecture is similar to the previous one with the difference that multiple TCP connections, as opposed to just one, are used for inter-proxy communication. The goal is to be more aggressive in inter-proxy data transfer by using multiple connections. This proved to

be quite effective yielding average throughput performance improvements of the order of greater than 60% for using 2 and 3 connections at the typical satellite bit error rate of 10^{-7}. While yielding even greater throughput improvements at the higher error rate of 10^{-6} that exceeded 150% on average when using 4 inter-proxy connections. Thus, this connection-splitting proxy architecture obviously proved to be very effective.

5.3 UDP Connection-Splitting Performance Enhancing Proxy Architecture

This was the third and last connection-splitting architecture implemented and proved to be the most effective of all of them, yielding exceptional throughput performance increases. Under the typical bit error rate (10^{-7}), the UDP proxy architecture provided a throughput increase of more than 4-fold even for utilizing just a single inter-proxy UDP connection. At a higher bit error rate (10^{-6}), the throughput increase was even greater leading to an order of magnitude increase in throughput.

5.4 Future Work and Recommendations

i) As noted in chapter 4 for the UDP connection-splitting architecture, in some instances extended losses can cause data transfer to fail due to occasional burst losses or rain fade that can result in not enough packets being available to reconstruct the source data packets from a given packet block. This dictates that for a real-world implementation, the UDP proxy scheme must be complemented with a retransmission mechanism in order to be able to recover from such losses. While using multiple connections and/or increased redundancy can mitigate the effect of extended losses, a retransmission mechanism is necessary for the (infrequent but occurring) case that packet losses are too great for the redundancy mechanism to recover. It is proposed that an explicit loss notification mechanism be implemented rather than a "positive" acknowledgment mechanism, meaning there would not be an acknowledgment for every packet or group of packets received. Rather, only when there is an insufficient number of packets to reconstruct the original source data packets, would a retransmission request be sent for the

packets of that given block. The use of explicit loss notification rather than a "positive" acknowledgment mechanism would result in far less traffic in the reverse channel (important for when bandwidth asymmetry exists), be less complex and pose a smaller burden on the proxies.

ii) Another possible enhancement to the UDP connection-splitting architecture is the runtime variation of the erasure code redundancy based on the observed amount of packet loss. In other words, if the packet loss due to corruption is less than expected then the erasure code redundancy could be reduced to allow for more efficient operation and vice versa if the packet loss rate due to corruption is larger than expected. This would also require a mechanism that would be able to identify if packets losses are due to corruption or congestion, which can prove to be very difficult to implement.

iii) An important potential enhancement for the UDP connection-splitting architecture is the use of a more efficient erasure code. An ideal erasure code would be the relatively new Tornado erasure codes, which are according to benchmarks very efficient, requiring very small encoding and decoding times.

iv) One potential enhancement for all types of connection-splitting proxy architectures is the implementation of a more sophisticated end-to-end flow control mechanism to complement the "back-pressure" algorithm in order to reduce performance dependency on proxy buffer sizes. Some such potential modifications have been proposed in [BhBB99], such as for example deliberately causing packets losses in some instances in order to reduce the end host transmission rate.

v) A final recommendation regarding the actual implementation of the connection-splitting proxy architectures, is that for better performance a lower layer (e.g. kernel level) or even more suitably a hardware implementation (as with IP routers for example) would be more suitable for a real-world implementation.

Of course there are a plethora of modifications that can be made to the connection-

splitting proxy architectures that have been implemented and studied in this thesis. For example, by using modified or radically different inter-proxy protocols, as there have been some commercial and non-commercial implementations of connection-splitting proxy architectures, that go beyond the scope of this thesis to cover.

REFERENCES

[Allm97] M. Allman, "Improving TCP performance over satellite channels", Master's thesis, Ohio University, June 1997.

[Allm98] M. Allman, "On the generation and use of TCP acknowledgments", ACM Computer Communication Review, 28(5), October 1998.

[BaPK97] H. Balakrishnan, V. N. Padmanabhan and R. Katz, "The effects of asymmetry on TCP performance", Proceedings of the ACM/IEEE Mobicom, Budapest, Hungary, ACM, September 1997.

[BaRS99] H. Balakrishnan, H. Rahul and S. Seshan, "An integrated congestion management architecture for Internet hosts", ACM SIGCOMM, September 1999.

[BaSa98] K. Bajaj and H. Saran, "A link layer protocol for improving TCP performance over satellite channels", Dept. of Computer Science & Eng., Indian Institute of Technology, New Delhi-110 016, India, 1998.

[Bhar99] V. G. Bharadwaj, "Improving TCP performance over high-bandwidth geostationary satellite links", Master of Science thesis, Department of Electrical and Computer Engineering, University of Maryland, 1999.

[BhBB99] V. Bharadwaj, J. S. Baras and N. P. Butts, "An architecture for Internet Service via Broadband satellite networks", Centre for Satellite and Hybrid Communication Networks, Technical Research Report, CSHCN T.R. 99-12 (ISP T.R. 99-22), 1999.

[BSAK95] H. Balakrishnan, S. Seshan, E. Amir and R. Katz, "Improving TCP/IP performance over wireless networks", Proceedings of 1st ACM International Conference on

Mobile Computing and Networking (Mobicom), 1995.

[CaSA98] N. Cardwell, S. Savage and T. Anderson, "Modelling the performance of short TCP connections", Department of Computer Science and Engineering, University of Washington, October 1998.

[DENP97] M. Degermark, M. Engan, B. Nordgren and S. Pink, "Low-loss TCP/IP header compression for wireless networks", ACM/Baltzer Journal on Wireless Networks, vol. 3, no. 5, p. 375-87.

[FaFl96] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, V. 26, N. 3, p. 5-21, July 1996.

[Floy94] S. Floyd, "TCP and explicit congestion notification", ACM Computer Communication Review, V. 24, N. 5, October 1994.

[Frie95] D. E. Friedman, "Error control for satellite and hybrid communication networks", Master's thesis, CSHCN M.S. 95-1 (ISR M.S. 95-10), Center for Satellite and Hybrid Communication Networks - University of Maryland, 1995.

[GhDi99] N. Ghani and S. Dixit, "TCP/IP enhancements for satellite networks", IEEE Communications Magazine, p. 64-72, July 1999.

[JaKa88] R. Van Jacobson and M. J. Karels, "Congestion avoidance and control", ACM SIGCOMM. 1988.

[JHoe96] J. Hoe, "Improving the startup behavior of a congestion control scheme for TCP", ACM SIGCOMM, August 1996.

[KaVR98] L. Kalampoukas, A. Varma and K. K. Ramakrishnan, "Improving TCP throughput

over two-way asymmetric links: Analysis and solutions", Measurement and Modeling of Computer Systems, p. 78-89, 1998.

[Krus95] H. Kruse, "Performance of common data communications protocols over long delay links: An experimental examination", 3rd International Conference on Telecommunication Systems Modeling and Design, 1995.

[Metz00] C. Metz, "IP-over-satellite: Internet connectivity blasts off", IEEE Internet Computing, p. 84-89, July/August 2000.

[NIST00] NIST Net emulation package 2.0.10, National Institute of Standards and Technology, http://www.antd.nist.gov/itg/nistnet.

[PaSh97] C. Partridge and T. J. Shepard, "TCP/IP performance over satellite links", IEEE Network, September/October, p. 44-49, 1997.

[Phil00] S. Philopoulos, "Internet Mirror Site Parallel Downloading Schemes", Department of Electrical and Computer Engineering, University of Manitoba, Winnipeg, MB, Canada, 2000.

[RFC1144] V. Jacobson. "RFC 1144: Compressing TCP/IP headers", February 1990.

[RFC1323] V. Jacobson, R. Braden and D. Borman, "RFC 1323: TCP extensions for high performance networks", May 1992.

[RFC1379] R. Braden, "RFC 1379: Transaction TCP - Concepts", September 1992.

[RFC1644] R. Braden, "RFC 1644: T/TCP - TCP extensions for transactions: Functional specification", July 1994.

[RFC2018] M. Mathias, J. Mahdavi, S. Floyd and A. Romanow, "RFC 2018: TCP selective acknowledgment options", October 1996.

[RFC2140] J. Touch, "RFC 2140: TCP control block interdependence", April 1997.

[RFC2414] M. Allman, S. Floyd and C. Partridge, "RFC 2414: Increasing TCP's initial window", September 1998.

[RFC2481] K. Ramakrishnan and S. Floyd, "RFC 2481: A proposal to add Explicit Congestion Notification (ECN) to IP", January 1999.

[RFC2488] M. Allman et al., "RFC 2488: Enhancing TCP over satellite channels using standard mechanisms", January 1999.

[RFC2507] M. Degermark, B. Nordgren and S. Pink, "RFC 2507: IP header compression", February 1999.

[RFC2760] M. Allman et al., "RFC 2760: Ongoing TCP research related to satellites", February 2000.

[Rizz97] L. Rizzo, "Effective erasure codes for reliable computer communication protocols", Dip. di Ingegneria dell'Informazione, Universita di Pisa, 1997.

[SLSC98] B. Suter, T. Lakshman, D. Stiliadis and A. Choudhury, "Design considerations for supporting TCP with per-flow queuing", Proceedings of IEEE Infocom '98 Conference, 1998.

[XyPo99] G. Xylomenos and G. C. Polyzos, "Internet protocol performance over networks with wireless links", IEEE Network, p. 55-63, July/August 1999.

APPENDIX A

C LANGUAGE LISTING FOR SINGLE-CONNECTION-TCP CONNECTION-SPLITTING PROXIES

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>


//Choose whether serv-side proxy Name or IP read from file
//#define SS_PRX_USE_NAME
#define SS_PRX_USE_IP


#define  C_PORTNUMBER 6005       //Server-side proxy will listen on different port
(get from file)
#define  FILENAME   "smpl_cs_proxy_file.txt"   //File that contains address & port
# of serv-side proxy
#define  SOCK_BUFFER_SIZE   1024
#define  CS_TO_SS_PROXYBUFFER_SIZE  100  //# of nodes in buffer
#define  SS_TO_CS_PROXYBUFFER_SIZE  5000  //# of nodes in buffer


#define  DELAY_IN_MSEC 0
//#define USE_DELAY
```

```
/*
* Function prototypes
*/
void *proxyside_send_ThreadFunc(void *);
void *proxyside_recv_ThreadFunc(void *);
void *clientside_send_ThreadFunc(void *);
void *clientside_recv_ThreadFunc(void *);



struct BufferNode {
        int  EntryFull;  //Flag (serves as semaphore). If 1 then value stored, else is
empty and can be written into
        long int   DataSize;  //Size of data stored in entry
        char  Data[SOCK_BUFFER_SIZE];
};




int   sockfd, newsockfd, servlen, cli_len, childpid;
struct sockaddr_in   cli_addr, serv_addr;
int   niaa;
struct BufferNode  cs_to_ss_ProxyBuffer[CS_TO_SS_PROXYBUFFER_SIZE];
//Buffer used for client->server_side_proxy traffic
struct BufferNode  ss_to_cs_ProxyBuffer[SS_TO_CS_PROXYBUFFER_SIZE];
//Buffer used for server_side_proxy->client traffic
int   ssp_sock;   //socket used to connect to server-side proxy



int main(void) {

        //Create the socket
        if ( (sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
                perror("ERROR: socket creation \n");
                exit(1);
        }


        //Create (client-side) proxy address
```

```
memset(&serv_addr, 0, sizeof(struct sockaddr_in));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(C_PORTNUMBER);
servlen = sizeof(struct sockaddr_in);

niaa = INADDR_ANY;
memcpy(&serv_addr.sin_addr, &niaa, sizeof(long));


//Bind socket to server address
if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
        perror("ERROR: bind \n");
        exit(1);
}


//Listen for client connections
if ( listen(sockfd, 5) < 0 ) {
        perror("ERROR: listen \n");
        exit(1);
}


for (;;) {

        cli_len = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);

        if (newsockfd < 0) {
                perror("ERROR: Can't create new socket \n");
                exit(1);
        }


        //Create child process
        if ( (childpid = fork()) < 0 ) {
                perror("ERROR: Fork \n");
                exit(1);
        }
        else if (childpid == 0) {   /* ****** child process ****** */
```

```
                /* Child process vars */
                struct hostent   *hp;   //used in connecting to server-side
proxy
                struct sockaddr_in   name;   //used in connecting to server-
side proxy
                int   len;   //used in connecting to server-side proxy
                int   thr_concurrency_counter;
                thread_t   ThreadID[4];   //Array of thread IDs for
thr_create calls
                void   *dummyPTR[4];   //Array of dummy pointers used in
thr_create calls
                FILE   *fd;
                int   S_Port_Number;   //Port on which server-side proxy
listens
                char   S_Proxy_Name[64];   //IP USED NOW!! DNS name of
server-side proxy
                int   i; //counter var
                size_t  TaskStatus[4]; //Used in thr_join() calls
                int mm;
                char  S_Proxy_IP[55]; //IP address of proxy now read
                ulong_t addr;  //Used for getting serv.side proxy info from its
IP address


                close(sockfd);        //close original socket


                //-------------------code-------------------

                printf("CHECK: CHILD process created \n\n");



                /*
                * Read file to obtain address (name & port #) of server-side
proxy
                * Format: "port# ss-proxy-name" ("'s not included)
                */
```

```
//sem_lock not needed

if ( (fd = fopen(FILENAME, "r")) == NULL ) {
        perror("ERROR: Couldn't open file \n");
        exit(1);
}

#ifdef SS_PRX_USE_NAME
 fscanf(fd, "%d %s", &S_Port_Number, S_Proxy_Name);
#endif
#ifdef SS_PRX_USE_IP
 fscanf(fd, "%d %s", &S_Port_Number, S_Proxy_IP);
#endif



//Check values read from file
#ifdef SS_PRX_USE_NAME
 printf("CHECK: S_Proxy_Name: %s\n", S_Proxy_Name);
#endif

#ifdef SS_PRX_USE_IP
 printf("CHECK: S_Proxy_IP: %s\n", S_Proxy_IP);
#endif

printf("CHECK: S_Port_Number: %d \n", S_Port_Number);


fclose(fd);

//sem_unlock not needed




/*
* Initialize proxy buffers (i.e. set all flags to 0)
*/
for (i=0; i<CS_TO_SS_PROXYBUFFER_SIZE; i++) {
        cs_to_ss_ProxyBuffer[i].EntryFull = 0;
```

```
                    }
                    for (i=0; i<SS_TO_CS_PROXYBUFFER_SIZE; i++) {
                            ss_to_cs_ProxyBuffer[i].EntryFull = 0;
                    }



                    /*
                    * Open connection to server-side proxy
                    */
                    //-----------
                    //Get server-side proxy's address
                    #ifdef SS_PRX_USE_NAME
                     if ( (hp = gethostbyname(S_Proxy_Name)) == NULL ) {
                            printf("ERROR: Can't get server-side proxy address
(%s)\n\n", S_Proxy_Name);
                            exit(1);
                     }
                    #endif

                    #ifdef SS_PRX_USE_IP
                    //Get server-side proxy's address
                    if ((int)(addr = inet_addr(S_Proxy_IP)) == -1) {
                            printf("IP-address must be of the form a.b.c.d\n");
                            exit (2);
                    }
                     hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
                    if (hp == NULL) {
                            printf("host information for %s not found\n",
S_Proxy_IP);
                            exit (3);
                    }
                    #endif



                    //Create socket
                    if ( (ssp_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                            printf("ERROR: Can't create socket for server-side proxy
\n\n");
                            exit(1);
```

```
}


//Create server address
memset(&name, 0, sizeof(struct sockaddr_in));

name.sin_family = AF_INET;
name.sin_port = htons(S_Port_Number);
memcpy(&name.sin_addr, hp->h_addr_list[0], hp->h_length);
len = sizeof(struct sockaddr_in);


//Connect to server
if ( connect(ssp_sock, (struct sockaddr *) &name, len) < 0 ) {
        printf("ERROR: Can't connect to server-side proxy\n\n");
        exit(1);
}


//----------


/*
* Spawn the sending/receiving threads
*/
thr_create(NULL, 0, clientside_recv_ThreadFunc, (void *)
dummyPTR[0], 0, &ThreadID[0]);
thr_create(NULL, 0, clientside_send_ThreadFunc, (void *)
dummyPTR[1], 0, &ThreadID[1]);
thr_create(NULL, 0, proxyside_recv_ThreadFunc, (void *)
dummyPTR[2], 0, &ThreadID[2]);
thr_create(NULL, 0, proxyside_send_ThreadFunc, (void *)
dummyPTR[3], 0, &ThreadID[3]);



/*
*Set the thread concurrency. Could improve thread scheduling.
*/
thr_concurrency_counter = 4+1;  //Include main()
thr_setconcurrency(thr_concurrency_counter);
```

```
                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    thr_join(ThreadID[0], 0, (void *) &TaskStatus[0]);
                    thr_join(ThreadID[1], 0, (void *) &TaskStatus[1]);
                    thr_join(ThreadID[2], 0, (void *) &TaskStatus[2]);
                    thr_join(ThreadID[3], 0, (void *) &TaskStatus[3]);



            } //endelseif                /* ****** child process ****** */


            close(newsockfd);


        } //endfor


} /*main*/


/*
**********************************************************************
********************** */


void *clientside_recv_ThreadFunc(void *dummyPTR)  //Receives data from client &
stores in buffer(cs_to_ss_ProxyBuffer)
{

        int   n;
        int   i;  //counter var
        char   buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;
```

```
        printf("CHECK: clientside_recv_Thread spawned\n\n");


        i=-1;  //Init counter
        while ((n = recv(newsockfd, buf, sizeof(buf), 0)) > 0) {


                //Increment counter
                if (i < (CS_TO_SS_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                /*
                * Copy received data to cs_to_ss_ProxyBuffer;
                */
                while (cs_to_ss_ProxyBuffer[i].EntryFull != 0) {
                        printf("STATUS/CHECK(clientside_recv): Waiting on
cs_to_ss_ProxyBuffer != 0\n\n");
                        thr_yield();
                }
                memcpy(cs_to_ss_ProxyBuffer[i].Data, buf, n);
                cs_to_ss_ProxyBuffer[i].DataSize = n;
                cs_to_ss_ProxyBuffer[i].EntryFull = 1;


        }  //while
        if (n<=0) {
                printf("CHILD PROC. EXITING: n<=0 while recv. from client(client-side
proxy)\n");
                printf("IF AT END OF CLIENT-SERVER COMMUN. THEN OK \n\n");
                close(newsockfd);
                exit(1);
                thr_exit(thr_exit_status);
        }


}  //clientside_recv_ThreadFunc
```

```
/*
*****************************************************************************
********************* */


void *clientside_send_ThreadFunc(void *dummyPTR)  //Gets data from
buffer(ss_to_cs_ProxyBuffer) & sends to client
{

        int   n;
        int   i;  //Counter var
        char  buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;



        printf("CHECK: clientside_send_Thread spawned\n\n");


        i = -1;  //Init counter
        while (1) {

                /*
                * Get data from ss_to_cs_ProxyBuffer;
                */

                //Increment counter
                if (i < (SS_TO_CS_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                while (ss_to_cs_ProxyBuffer[i].EntryFull != 1) {
                        thr_yield();
                }
                memcpy(buf, ss_to_cs_ProxyBuffer[i].Data,
ss_to_cs_ProxyBuffer[i].DataSize);
                n = ss_to_cs_ProxyBuffer[i].DataSize;
                ss_to_cs_ProxyBuffer[i].EntryFull = 0;
```

```
                /*
                * Send data to client
                */
                if ( send(newsockfd, buf, n, 0) < 0) {
                        printf("ERROR: (smpl_cs_proxy) While sending data to client
<0 \n\n");
                        exit(1);
                        thr_exit(thr_exit_status);
                }


        } //while


}  //clientside_send_ThreadFunc


/*
*****************************************************************************
********************** */


void *proxyside_recv_ThreadFunc(void *dummyPTR)  //Gets data from
buffer(ss_to_cs_ProxyBuffer) & sends to client
{


        int   n;
        int   i; //Counter var
        char  buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;


        printf("CHECK: proxyside_recv_Thread spawned\n\n");
        i = -1;
        while ((n = recv(ssp_sock, buf, sizeof(buf), 0)) > 0) {

                /*
                * Copy received data to ss_to_cs_ProxyBuffer;
```

```
                */

                //Increment counter
                if (i < (SS_TO_CS_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                /*
                * Copy received data to ss_to_cs_ProxyBuffer;
                */
                while (ss_to_cs_ProxyBuffer[i].EntryFull != 0) {
                        printf("STATUS/CHECK(proxyside_recv): Waiting on
ss_to_cs_ProxyBuffer != 0\n\n");
                        thr_yield();
                }
                memcpy(ss_to_cs_ProxyBuffer[i].Data, buf, n);
                ss_to_cs_ProxyBuffer[i].DataSize = n;
                ss_to_cs_ProxyBuffer[i].EntryFull = 1;


        } //while
        if (n<=0) {
                printf("FATAL ERROR: n<=0 while recv. from server-side proxy (client-
side proxy) \n\n");
                close(ssp_sock);
                exit(1);
                thr_exit(thr_exit_status);
        }


} //proxyside_recv_ThreadFunc


/*
**********************************************************************************
********************** */


void *proxyside_send_ThreadFunc(void *dummyPTR)  //Gets data from
```

```
buffer(cs_to_ss_ProxyBuffer) & sends to server-side proxy
{

        int   n;
        int  i;  //Counter var
        char   buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;

        struct timeval  *start_time;
        struct timeval  *curr_time;
        long  start_time_in_msec, curr_time_in_msec;



        printf("CHECK: proxyside_send_Thread spawned\n\n");



        /*
        * Allocate memory to time vars
        */
        start_time = (struct timeval *) malloc(sizeof(struct timeval));
        curr_time = (struct timeval *) malloc(sizeof(struct timeval));


        i=-1;
        while (1) {

                /*
                * Get data from cs_to_ss_ProxyBuffer;
                */

                //Increment counter
                if (i < (CS_TO_SS_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                while (cs_to_ss_ProxyBuffer[i].EntryFull != 1) {
                        thr_yield();
```

```
                }
                memcpy(buf, cs_to_ss_ProxyBuffer[i].Data,
cs_to_ss_ProxyBuffer[i].DataSize);
                n = cs_to_ss_ProxyBuffer[i].DataSize;
                cs_to_ss_ProxyBuffer[i].EntryFull = 0;


        /*
        * Send data to server-side proxy
        */

     if (n>0) {


        #ifdef USE_DELAY

        // ----------------

        if ( gettimeofday(start_time, NULL) <0 ) {
                printf("ERROR(proxy_send): gettimeofday error\n\n");
                exit(1);
        }

        start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;



        while (1) {

                if ( gettimeofday(curr_time, NULL) <0 ) {
                        printf("ERROR(proxy_send): gettimeofday error\n\n");
                        exit(1);
                }

                curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;

                if ( (curr_time_in_msec - start_time_in_msec) >= DELAY_IN_MSEC)
                        break;
```

```
                //Since haven't reached timeout value yet have thread sleep/yield
                thr_yield();

        } //while


        // ----------------

        #endif


        if ( send(ssp_sock, buf, n , 0) < 0) {
                printf("ERROR: (smpl_cs_proxy) While sending data to server-side
proxy <0 \n\n");
                exit(1);
                thr_exit(thr_exit_status);
        }


    } //if n>0


  } //while


} //proxyside_send_ThreadFunc
```

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>



//#define SERV_USE_NAME
#define SERV_USE_IP



#define S_PORTNUMBER 5006       //Server-side proxy listens on this port
#define FILENAME   "smpl_ss_proxy_file.txt"  //File that contains address & port
# of server
#define SOCK_BUFFER_SIZE   1024
#define  CS_TO_S_PROXYBUFFER_SIZE  100  //# of nodes in buffer
#define  S_TO_CS_PROXYBUFFER_SIZE  5000  //# of nodes in buffer

#define  DELAY_IN_MSEC 0
//#define USE_DELAY
```

```
/*
* Function prototypes
*/
void *serverside_send_ThreadFunc(void *);
void *serverside_recv_ThreadFunc(void *);
void *cproxyside_send_ThreadFunc(void *);
void *cproxyside_recv_ThreadFunc(void *);



struct BufferNode{
        int EntryFull; //Flag (serves as semaphore). If 1 then value stored, else
is empty and can be written into
        long int DataSize; //Size of data stored in entry
        char Data[SOCK_BUFFER_SIZE];
};



int sockfd, newsockfd, servlen, cli_len, childpid;
struct sockaddr_in cli_addr, serv_addr;
struct BufferNode
csproxy_to_server_ProxyBuffer[CS_TO_S_PROXYBUFFER_SIZE]; //Buffer used
for client->server_side_proxy traffic
struct BufferNode
server_to_csproxy_ProxyBuffer[S_TO_CS_PROXYBUFFER_SIZE]; //Buffer used
for server_side_proxy->client traffic
int server_sock; //Socket used to connect to server
int niaa;



int main(void) {

        //Create the socket
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            perror("ERROR: socket creation \n");
```

```
    exit(1);
}


//Create (server-side) proxy address
memset(&serv_addr, 0, sizeof(struct sockaddr_in));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(S_PORTNUMBER);
servlen = sizeof(struct sockaddr_in);

niaa = INADDR_ANY;
memcpy(&serv_addr.sin_addr, &niaa, sizeof(long));


//Bind socket to server address
if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
    perror("ERROR: bind \n");
    exit(1);
}


//Listen for client connections
if ( listen(sockfd, 5) < 0 ) {
    perror("ERROR: listen \n");
    exit(1);
}


for (;;) {

    cli_len = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);

    if (newsockfd < 0) {
            perror("ERROR: Can't create new socket \n");
            exit(1);
    }


    //Create child process
```

```
if ( (childpid = fork()) < 0 ) {
        perror("ERROR: Fork \n");
        exit(1);
}
else if (childpid == 0) {   /* ****** child process ****** */


        /* Child process vars */
        struct hostent   *hp;   //used in connecting to server
        struct sockaddr_in   name;   //used in connecting to server
        int   len;   //used in connecting to server
        int   thr_concurrency_counter;
        thread_t   ThreadID[4];   //Array of thread IDs for
thr_create calls
        void   *dummyPTR[4];   //Array of dummy pointers used in
thr_create calls
        FILE   *fd;
        int   Server_PortNumber;   //Port on which server listens
        char   Server_Name[64];   //DNS name of server
        char  Server_IP[64];  //IP of server
        ulong_t  addr;
        int   i;  //counter var
        size_t  TaskStatus[4];  //Used in thr_join() calls



        close(sockfd);        //close original socket


        //-------------------code-------------------

        printf("CHECK: CHILD process created \n\n");


        /*
        * Initialize proxy buffers (i.e. set all flags to 0)
        */
        for (i=0; i<CS_TO_S_PROXYBUFFER_SIZE; i++) {
                csproxy_to_server_ProxyBuffer[i].EntryFull = 0;
        }
        for (i=0; i<S_TO_CS_PROXYBUFFER_SIZE; i++) {
```

```
                server_to_csproxy_ProxyBuffer[i].EntryFull = 0;
        }


        /*
        * Read file to obtain address (name & port #) of server
        * Format: "port# server_name" ("'s not included )
        */
        //sem_lock not needed

        if ( (fd = fopen(FILENAME, "r")) == NULL) {
                perror("ERROR: Couldn't open file \n");
                exit(1);
        }

        #ifdef SERV_USE_NAME
        fscanf(fd, "%d %s", &Server_PortNumber, Server_Name);
        #endif
        #ifdef SERV_USE_IP
        fscanf(fd, "%d %s", &Server_PortNumber, Server_IP);
        #endif


        //Check values read from file
        #ifdef SERV_USE_NAME
        printf("CHECK: Server_Name: %s\n", Server_Name);
        #endif
        #ifdef SERV_USE_IP
        printf("CHECK: Server_IP: %s\n", Server_IP);
        #endif

        printf("CHECK: Server_PortNumber: %d \n",
Server_PortNumber);


        fclose(fd);

        //sem_unlock not needed
```

```
                /*
                * Open connection to server
                */
                //-----------
                //Get server's address
                #ifdef SERV_USE_NAME
                if ( (hp = gethostbyname(Server_Name)) == NULL ) {
                        printf("ERROR: Can't get server address(%s)\n\n",
Server_Name);
                        exit(1);
                }
                #endif
                #ifdef SERV_USE_IP
                if ((int)(addr = inet_addr(Server_IP)) == -1) {
                        printf("IP-address must be of the form a.b.c.d\n");
                        exit (2);
                }
                hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
                if (hp == NULL) {
                        printf("host information for %s not found\n",
Server_IP);
                        exit (3);
                }
                #endif


                //Create socket
                if ( (server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                        printf("ERROR: Can't create socket for server \n\n");
                        exit(1);
                }


                //Create server address
                memset(&name, 0, sizeof(struct sockaddr_in));

                name.sin_family = AF_INET;
                name.sin_port = htons(Server_PortNumber);
                memcpy(&name.sin_addr, hp->h_addr_list[0], hp->h_length);
                len = sizeof(struct sockaddr_in);
```

```
                    //Connect to server
                    if ( connect(server_sock, (struct sockaddr *) &name, len) < 0
) {
                            printf("ERROR: Can't connect to server \n\n");
                            exit(1);
                    }


                    //----------


                    /*
                    * Spawn the sending/receiving threads
                    */
                    thr_create(NULL, 0, cproxyside_recv_ThreadFunc, (void *)
dummyPTR[0], 0, &ThreadID[0]);
                    thr_create(NULL, 0, cproxyside_send_ThreadFunc, (void *)
dummyPTR[1], 0, &ThreadID[1]);
                    thr_create(NULL, 0, serverside_recv_ThreadFunc, (void *)
dummyPTR[2], 0, &ThreadID[2]);
                    thr_create(NULL, 0, serverside_send_ThreadFunc, (void *)
dummyPTR[3], 0, &ThreadID[3]);



                    /*
                    *Set the thread concurrency. Could improve thread scheduling.
                    */
                    thr_concurrency_counter = 4+1;  //Include main()
                    thr_setconcurrency(thr_concurrency_counter);



                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    thr_join(ThreadID[0], 0, (void *) &TaskStatus[0]);
                    thr_join(ThreadID[1], 0, (void *) &TaskStatus[1]);
                    thr_join(ThreadID[2], 0, (void *) &TaskStatus[2]);
```

```
                thr_join(ThreadID[3], 0, (void *) &TaskStatus[3]);
            } //endelseif                    /* ****** child process ****** */

            close(newsockfd);

        } //endfor


} /*main*/


/*
**************************************************************************
********************* */


void *cproxyside_recv_ThreadFunc(void  *dummyPTR)  //Receives data from
client-side proxy & stores in csproxy_to_server_ProxyBuffer
{

        int   n;
        int   i;  //counter var
        char  buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;



        i=-1;  //Init counter
        while ((n = recv(newsockfd, buf, sizeof(buf), 0)) > 0) {

            /*
            * Copy received data to csproxy_to_server_ProxyBuffer;
            */


            //Increment counter
            if (i < (CS_TO_S_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;
```

```
                while (csproxy_to_server_ProxyBuffer[i].EntryFull != 0) {
                        printf("STATUS/CHECK(cproxyside_recv): Waiting on
csproxy_to_server_ProxyBuffer != 0\n\n");

                        thr_yield();
                }
                memcpy(csproxy_to_server_ProxyBuffer[i].Data, buf, n);
                csproxy_to_server_ProxyBuffer[i].DataSize = n;
                csproxy_to_server_ProxyBuffer[i].EntryFull = 1;

        } //while
        if (n<=0) {
            printf("CHILD PROC.EXITING:n<=0 while recv. from client-side
proxy(server-side proxy)\n");
            printf("IF END OF CLIENT-SERVER COMMUN. THEN IS OK\n\n");
            close(newsockfd);
            exit(1);
            thr_exit(thr_exit_status);
        }


} //cproxyside_recv_ThreadFunc


/*
**********************************************************************************
********************** */


void *cproxyside_send_ThreadFunc(void  *dummyPTR)  //Gets data from
buffer(server_to_csproxy_ProxyBuffer) & sends to client-side proxy
{

        int   n;
        int   i;
        char   buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;

        struct timeval  *start_time;
```

```
struct timeval  *curr_time;
long  start_time_in_msec, curr_time_in_msec;



/*
* Allocate memory to time vars
*/
start_time = (struct timeval *) malloc(sizeof(struct timeval));
curr_time = (struct timeval *) malloc(sizeof(struct timeval));


i=-1;
while (1) {

    /*
    * Get data from server_to_csproxy_ProxyBuffer;
    */

    //Increment counter
    if (i < (S_TO_CS_PROXYBUFFER_SIZE-1))
            i++;
    else
            i = 0;


    while (server_to_csproxy_ProxyBuffer[i].EntryFull != 1) {
            thr_yield();
    }
    memcpy(buf, server_to_csproxy_ProxyBuffer[i].Data,
server_to_csproxy_ProxyBuffer[i].DataSize);
    n = server_to_csproxy_ProxyBuffer[i].DataSize;
    server_to_csproxy_ProxyBuffer[i].EntryFull = 0;



    /*
    * Send data to client-side proxy
    */
    if (n>0) {
```

```
                    #ifdef USE_DELAY

                    // ----------------

                     if ( gettimeofday(start_time, NULL) <0 ) {
                           printf("ERROR(proxy_send): gettimeofday error\n\n");
                           exit(1);
                     }

                    start_time_in_msec = (start_time->tv_sec)*1000 +
(start_time->tv_usec)/1000;



                    while (1) {
                           if ( gettimeofday(curr_time, NULL) <0 ) {
                                 printf("ERROR(proxy_send): gettimeofday
error\n\n");
                                 exit(1);
                           }

                           curr_time_in_msec = (curr_time->tv_sec)*1000 +
(curr_time->tv_usec)/1000;

                           if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                 break;


                           //Since haven't reached timeout value yet have thread
sleep/yield
                           thr_yield();

                    } //while


                    // ----------------

                    #endif
```

```
                    if ( send(newsockfd, buf, n, 0) < 0) {
                            printf("ERROR: (smpl_ss_proxy) While sending data to
client-side proxy <0 \n\n");
                            exit(1);
                            thr_exit(thr_exit_status);
                    }

            } //if n>0

  } //while


} //cproxyside_send_ThreadFunc


/*
**********************************************************************************
********************* */


void *serverside_recv_ThreadFunc(void *dummyPTR) //Receives data from
server & stores in buffer (server_to_csproxy_ProxyBuffer)
{

        int  n;
        int  i;
        char  buf[SOCK_BUFFER_SIZE];
        void  *thr_exit_status;


        i=-1;
        while ((n = recv(server_sock, buf, sizeof(buf), 0)) > 0) {

           /*
           * Copy received data to server_to_csproxy_ProxyBuffer;
           */

           //Increment counter
           if (i < (S_TO_CS_PROXYBUFFER_SIZE-1))
                   i++;
           else
```

```
                    i = 0;
                while (server_to_csproxy_ProxyBuffer[i].EntryFull != 0) {
                    printf("STATUS/CHECK(serverside_recv): Waiting on
server_to_csproxy_ProxyBuffer != 0\n\n");
                    thr_yield();
                }
                memcpy(server_to_csproxy_ProxyBuffer[i].Data, buf, n);
                server_to_csproxy_ProxyBuffer[i].DataSize = n;
                server_to_csproxy_ProxyBuffer[i].EntryFull = 1;


            } //while
            if (n<=0) {
                printf("FATAL ERROR: n<=0 while recv. from server (server-side
proxy) \n\n");
                close(server_sock);
                exit(1);
                thr_exit(thr_exit_status);
            }


} //serverside_recv_ThreadFunc


/*
**********************************************************************************
********************** */


void *serverside_send_ThreadFunc(void *dummyPTR) //Retrieves data from
buffer (csproxy_to_server_ProxyBuffer) & sends to server
{

            int  n;
            int  i; //counter var
            char  buf[SOCK_BUFFER_SIZE];
            void  *thr_exit_status;


            i=-1;
            while (1) {
```

```
                /*
                * Get data from csproxy_to_server_ProxyBuffer;
                */
                //Increment counter
                if (i < (CS_TO_S_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                while (csproxy_to_server_ProxyBuffer[i].EntryFull != 1) {
                        thr_yield();
                }
                memcpy(buf, csproxy_to_server_ProxyBuffer[i].Data,
csproxy_to_server_ProxyBuffer[i].DataSize);
                n = csproxy_to_server_ProxyBuffer[i].DataSize;
                csproxy_to_server_ProxyBuffer[i].EntryFull = 0;



                /*
                * Send data to server
                */
                if ( send(server_sock, buf, n, 0) < 0) {
                        printf("ERROR: (smpl_ss_proxy) While sending data to server
<0 \n\n");
                        exit(1);
                        thr_exit(thr_exit_status);
                }


  } //while


} //serverside_send_ThreadFunc
```

APPENDIX B

C LANGUAGE LISTING FOR MULTI-CONNECTION-TCP CONNECTION-SPLITTING PROXIES

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>



//Debug "defines"
//#define new2_PRINT_DEBUG

//Choose whether serv-side proxy Name or IP read from file
//#define SS_PRX_USE_NAME
#define SS_PRX_USE_IP




#define C_PORTNUMBER 6005       //Client-side proxy listens on this port
#define  FILENAME   "p_cs_proxy_file.txt"  //File that contains address & port #
of server-side proxy
#define  SOCK_BUFFER_SIZE   1024
#define  PS_TO_C_PROXYBUFFER_SIZE  4000  //# of nodes in buffer
#define  C_TO_PS_PROXYBUFFER_SIZE  200  //# of nodes in buffer
//#define  PROXY_SIDE_BUFFER_SIZE   1000  //# of nodes in buffer
#define  S_PROXY_SIDE_BUFFER_SIZE   1  //NOT USED -50- # of nodes in buffer
```

```
#define  R_PROXY_SIDE_BUFFER_SIZE   10000  //5000 # of nodes in buffer

#define LENGTH_HEADER_SIZE 2  //Length of Length header that is appended in
front of packets

#define MAX_CONN_NUMB_PER_LOGICAL_CONN  5  //Max # of connections per
logical connection

//#define USE_DELAY
#define  DELAY_IN_MSEC 0



/*
* Function prototypes
*/
void *proxyside_recv_ThreadFunc(void *);
void *clientside_send_ThreadFunc(void *);
void *clientside_recv_ThreadFunc(void *);
void *client_to_proxy_Control_ThreadFunc(void *);
void *proxy_to_client_Control_ThreadFunc(void *);



struct BufferNode {
        int  EntryFull;  //Flag (serves as semaphore). If 1 then value stored, else
is empty and can be written into
        long int   DataSize;  //Size of data stored in entry
        char  Data[SOCK_BUFFER_SIZE];
};



int   sockfd, servlen, cli_len, childpid;
int   client_sock;   //Socket created from cl.side-proxy listening
struct sockaddr_in   cli_addr, serv_addr;
int   niaa;
int   servsideproxy_sockfd[MAX_CONN_NUMB_PER_LOGICAL_CONN]; //sockets
```

```
used to connect to serv-side proxy
struct BufferNode  client_to_ssproxy_ProxyBuffer[C_TO_PS_PROXYBUFFER_SIZE];
//Buffer used for client->server_side_proxy traffic
struct BufferNode  ssproxy_to_client_ProxyBuffer[PS_TO_C_PROXYBUFFER_SIZE];
//Buffer used for server_side_proxy->client traffic
struct BufferNode
proxy_side_sendBuffer[S_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that will be

//sent out to cl.side proxy
struct BufferNode
proxy_side_recvBuffer[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that is

//received fromt to cl.side proxy
int Connection_Number;  //Number of connections to be opened for a given proxy-
to-proxy 'logical' connection
```

```
int main(void) {

        //Create the socket
        if ( (sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            perror("ERROR: socket creation \n");
            exit(1);
        }


        //Create (client-side) proxy address
        memset(&serv_addr, 0, sizeof(struct sockaddr_in));

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(C_PORTNUMBER);
        servlen = sizeof(struct sockaddr_in);

        niaa = INADDR_ANY;
```

```
        memcpy(&serv_addr.sin_addr, &niaa, sizeof(long));


        //Bind socket to address
        if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
            perror("ERROR: bind \n");
            exit(1);
        }


        //Listen for client connections
        if ( listen(sockfd, 7) < 0 ) {
            perror("ERROR: listen \n");
            close(sockfd);
            exit(1);
        }




        for (;;) {

            cli_len = sizeof(cli_addr);
            client_sock = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);

            if (client_sock < 0) {
                    perror("ERROR: Can't create new socket (from cl.side proxy
listening socket) \n");
                    exit(1);
            }



            //Create child process
            if ( (childpid = fork()) < 0 ) {
                    perror("ERROR: Fork \n");
                    exit(1);
            }
            else if (childpid == 0) {   /* ****** child process ****** */
```

```
                    /* Child process vars */
                    struct hostent   *hp;   //used in connecting to server
                    struct sockaddr_in   name;   //used in connecting to server
                    int   len;   //used in connecting to server
                    int   n;
                    int   thr_concurrency_counter;
                    thread_t
ThreadID[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4]; //Array of thread IDs
for thr_create calls
                    int
thr_ID_PTR[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4];  //Array of thr_ID
pointers used in thr_create calls
                    FILE   *fd;
                    int   ServSideProxy_PortNumber;   //Port on which server
listens
                    char   ServSideProxy_Name[64];   //DNS name of server
                    int   i,j;  //counter var
                    size_t
TaskStatus[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4];  //Used in thr_join()
call
                    char buf[SOCK_BUFFER_SIZE];  //Buffer used by parent to
send to other proxy # of connections to be opened
                    char Connection_Number_str[31];  //String containing
Connection_Number(as to be sent to other proxy)

                    char ServSideProxy_IP[55];  //Used for reading from file serv-
side proxy IP address
                    ulong_t  addr;



                    close(sockfd);        //close original socket


                    //-------------------code-------------------

                    printf("CHECK(p_cs_proxy): CHILD process created \n\n");
```

```
                    /*
                    * Read file to obtain address (name & port #) of server-side
proxy
                    * Format: "port# proxy_name" ("'s not included )
                    */
                    //sem_lock not needed

                    if ( (fd = fopen(FILENAME, "r")) == NULL) {
                            perror("ERROR(p_cs_proxy): Couldn't open file \n");
                            exit(1);
                    }

                    #ifdef SS_PRX_USE_NAME
                     fscanf(fd, "%d %s", &ServSideProxy_PortNumber,
ServSideProxy_Name);
                    #endif

                    #ifdef SS_PRX_USE_IP
                     fscanf(fd, "%d %s", &ServSideProxy_PortNumber,
ServSideProxy_IP);
                    #endif


                    //Check values read from file
                    #ifdef SS_PRX_USE_NAME
                     printf("CHECK(p_cs_proxy): ServSideProxy_Name: %s\n",
ServSideProxy_Name);
                    #endif

                    #ifdef SS_PRX_USE_IP
                     printf("CHECK(p_cs_proxy): ServSideProxy_IP: %s\n",
ServSideProxy_IP);
                    #endif


                    printf("CHECK(p_cs_proxy): ServSideProxy_PortNumber: %d
\n", ServSideProxy_PortNumber);

                    fclose(fd);
```

```
                    //sem_unlock not needed
                    /*
                    * Open connection to server-side proxy
                    */

                    //-----------
                    //Get server-side proxy's address

                    #ifdef SS_PRX_USE_NAME
                    if ( (hp = gethostbyname(ServSideProxy_Name)) == NULL ) {
                            printf("ERROR(p_cs_proxy): Can't get server-side proxy
address(%s)\n\n", ServSideProxy_Name);
                            exit(1);
                    }
                    #endif

                    #ifdef SS_PRX_USE_IP
                    if ((int)(addr = inet_addr(ServSideProxy_IP)) == -1) {
                             printf("IP-address must be of the form a.b.c.d\n");
                            exit (2);
                    }
                    hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
                     if (hp == NULL) {
                            printf("host information for %s not found\n",
ServSideProxy_IP);
                            exit (3);
                    }
                    #endif



                    //Create server-side proxy address
                    memset(&name, 0, sizeof(struct sockaddr_in));

                    name.sin_family = AF_INET;
                    name.sin_port = htons(ServSideProxy_PortNumber);
                    memcpy(&name.sin_addr, hp->h_addr_list[0], hp->h_length);
                    len = sizeof(struct sockaddr_in);
```

```
                    //Create socket
                    if ( (servsideproxy_sockfd[0] = socket(AF_INET,
SOCK_STREAM, 0)) < 0) {
                            printf("ERROR(p_cs_proxy): Can't create socket for
server-side proxy \n\n");
                            exit(1);
                    }



                    //Connect to server-side proxy
                    if ( connect(servsideproxy_sockfd[0], (struct sockaddr *)
&name, len) < 0 ) {
                            printf("ERROR(p_cs_proxy): Can't connect to server-
side proxy \n\n");
                            exit(1);
                    }



                    /*
                    * Read # of TOTAL connections to be opened (for each
'logical' connection to serv-side proxy)
                    * & send it to server-side proxy.
                    * Format: string containing # followed by '\0'
                    */
                    printf("Enter # of phys. connections per logical connection (%d
max):", MAX_CONN_NUMB_PER_LOGICAL_CONN);
                    scanf("%d", &Connection_Number);

                    sprintf(Connection_Number_str, "%d", Connection_Number);

                    //Copy connection # in buf to send out to serv-side proxy &
set buf size
                    strcpy(buf, Connection_Number_str);
                    n = strlen(buf) + 1;  //strlen()+1 to account for the '\0' char
that must be included
```

```
                    if ( send(servsideproxy_sockfd[0], buf, n , 0) < 0) {
                        printf("ERROR(p_cs_proxy): While sending connection #
to serv-side proxy <0 \n\n");
                        exit(1);
                    } //if


                    printf("CHECK(p_cs_proxy): %d connections to be opened
\n\n",Connection_Number);



                    /*
                    * Open (Connection_Number-1) connections .
NOTE:Connection_Number is TOTAL connections to be opened
                    */
                    for (i=1; i<Connection_Number; i++) {

                        //Create socket
                        if ( (servsideproxy_sockfd[i] = socket(AF_INET,
SOCK_STREAM, 0)) < 0) {
                            printf("ERROR(p_cs_proxy): Can't create socket
for server-side proxy for i:%d\n\n", i);
                            exit(1);
                        }


                        //Connect to server-side proxy
                        if ( connect(servsideproxy_sockfd[i], (struct sockaddr
*) &name, len) < 0 ) {
                            printf("ERROR(p_cs_proxy): Can't connect to
server-side proxy for i:%d\n\n", i);
                            exit(1);
                        }


                    } //for
```

```
/*
* Initialize proxy buffers (i.e. set all flags to 0)
*/
for (i=0; i<S_PROXY_SIDE_BUFFER_SIZE; i++) {
        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)    {
                proxy_side_sendBuffer[i][j].EntryFull = 0;
        }
}
for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {
        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)    {
                proxy_side_recvBuffer[i][j].EntryFull = 0;
        }
}


for (i=0; i<PS_TO_C_PROXYBUFFER_SIZE; i++) {
        ssproxy_to_client_ProxyBuffer[i].EntryFull = 0;
}
for (i=0; i<C_TO_PS_PROXYBUFFER_SIZE; i++) {
        client_to_ssproxy_ProxyBuffer[i].EntryFull = 0;
}



//----------


/*
* Init/fill-in thr_ID array. Thread IDs start from ZERO !!!
*/
for (i=0; i<(MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4);
i++)
        thr_ID_PTR[i] = i;
```

```
                    /*
                    *
                    * NOTE: proxy-side sending threads have thread IDs from (0
to Connection_Number-1) and
                    * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
                    *
                    */



                    /*
                    * Spawn the server-side proxy sending/receiving threads. Pair
for each connection w/ serv-side proxy
                    */


                    for (i=Connection_Number; i<(2*Connection_Number); i++) {
                          thr_create(NULL, 0, proxyside_recv_ThreadFunc, (void
*) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    } //for



                    if (i != (2*Connection_Number)) {
                          printf("ERROR/CHECK(p_cs_proxy): i counter <>
2*Connection_Number \n");
                          exit(1);
                    }


                    /*
                    * Spawn the client-side sending/receiving threads. Just one pair
for the single connection to client
                    */
                    thr_create(NULL, 0, clientside_send_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);
```

```
                    i++;
                    thr_create(NULL, 0, clientside_recv_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);


                    /*
                    * Spawn Control threads
                    */
                    i++;
                    thr_create(NULL, 0, client_to_proxy_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, proxy_to_client_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);


                    /*
                    *Set the thread concurrency. Could improve thread scheduling.
                    */
                    thr_concurrency_counter = Connection_Number*1+4 + 1;
                    thr_setconcurrency(thr_concurrency_counter);




                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    for (i= Connection_Number; i<(2*Connection_Number+4); i++)
{
                            thr_join(ThreadID[i], 0, (void *) &TaskStatus[i]);
                    }


          } //endelseif              /* ****** child process ****** */
```

```
            /*
            * Close in parent the "passed off to child" socket
            */
            close(client_sock);


        } //endfor

} /*main*/



/*
**********************************************************************************
********************* */

void *proxyside_recv_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n,m;
        int   i,k;  //counter vars
        char  *buf;  //socket buffer
        char  *bufStart;  //Used to store temporarily the socket buffer starting
address
        int buf_size;  //Size of buf. Used because sizeof(buf) doesn't work for
char *
        unsigned int  Packet_Length;  //Holds packet length derived from arrived
packet's Length header


        //Init buffer
        buf_size = (SOCK_BUFFER_SIZE+2);  // new2 - buf_size was
```

```
SOCK_BUFFER_SIZE
            buf = (char *) malloc( buf_size*sizeof(char) );

            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */


            i=0; //Init counter

            while (1) {  //new2


               /*
               * Get exaclty LENGTH_HEADER_SIZE bytes of new packet i.e. the
Length header
               */

               m = 0;
               while (m<2) {

                        if ((n=recv(servsideproxy_sockfd[thr_ID-Connection_Number],
buf+m, LENGTH_HEADER_SIZE-m, 0)) > 0) {
                                 m = m+n;
                        }  //if
                        else{
                                 printf("FATAL ERROR(prx_side_recv_Thread): n<=0
while recv. Length header\n\n");
                                 close(servsideproxy_sockfd[thr_ID-
Connection_Number]);
                                 exit(1);
                        }

               }  //while
```

```
#ifdef new2_PRINT_DEBUG
printf("CHECK new2: right after recv() #1 \n\n");
#endif



/*
* Get length from Length field
*/
Packet_Length = buf[0];
Packet_Length = (Packet_Length<<8) | (buf[1]);



/*
* Receive entire packet now
*/
m = 0;

while (m<Packet_Length) {

        if ((n=recv(servsideproxy_sockfd[thr_ID-Connection_Number],
buf+m, Packet_Length-m, 0)) > 0) {

                m = m+n;

        } //if
        else {
                printf("FATAL ERROR(prx_side_recv_Thread): n<=0
while recv. remaining packet \n\n");
                close(servsideproxy_sockfd[thr_ID-
Connection_Number]);
                exit(1);
        } //else


} //while
```

```
                //Error check
                if (m != Packet_Length) {
                        printf("ERROR(recv[%d]): m != Packet_Length (is >) \n\n",
thr_ID);
                        close(servsideproxy_sockfd[thr_ID-Connection_Number]);
                        exit(1);
                } //if



                /*
                * Now that got entrire packet, write it (w/o Length header) to
proxy_side_recvBuffer
                */

                while (proxy_side_recvBuffer[i][thr_ID-Connection_Number].EntryFull
!= 0) {
                        printf("CHECK(prx_recv[%d]): Waiting for
proxy_side_recvBuffer[][]!=0\n\n", thr_ID);
                        thr_yield();
                } //while

                memcpy(proxy_side_recvBuffer[i][thr_ID-Connection_Number].Data,
buf, Packet_Length);
                proxy_side_recvBuffer[i][thr_ID-Connection_Number].DataSize =
Packet_Length;
                proxy_side_recvBuffer[i][thr_ID-Connection_Number].EntryFull = 1;


                //Increment counter
                if (i < (R_PROXY_SIDE_BUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


        } //while(1)


} //proxyside_recv_ThreadFunc
```

```
/*
**************************************************************************
********************* */


void *clientside_send_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i; //counter var
        char   buf[SOCK_BUFFER_SIZE];



        i=-1;
        while (1) {

            /*
            * Get data from ssproxy_to_client_ProxyBuffer;
            */
            //Increment counter
            if (i < (PS_TO_C_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (ssproxy_to_client_ProxyBuffer[i].EntryFull != 1) {
                    thr_yield();
            }
            memcpy(buf, ssproxy_to_client_ProxyBuffer[i].Data,
ssproxy_to_client_ProxyBuffer[i].DataSize);
            n = ssproxy_to_client_ProxyBuffer[i].DataSize;
```

```
                ssproxy_to_client_ProxyBuffer[i].EntryFull = 0;



            /*
            * Send data to client
            */
            if (n>0) {
                if ( send(client_sock, buf, n, 0) < 0) {
                                printf("ERROR(p_cs_proxy server_send_thread): While
sending data to client <0 \n\n");
                                exit(1);
                } //if
            } //if


    } //while


} //clientside_send_ThreadFunc


/*
**********************************************************************************
********************** */


void *clientside_recv_ThreadFunc(void *thr_ID_PTR) {


            /*
            * Get thread ID
            */
            int  thr_ID = *((int *) thr_ID_PTR);

            int   n;
            int   i; //counter var
            char  buf[SOCK_BUFFER_SIZE];
```

```
        i=-1;

        while ((n = recv(client_sock, buf, sizeof(buf), 0)) > 0) {

            /*
            * Copy received data to client_to_ssproxy_ProxyBuffer;
            */

            //Increment counter
            if (i < (C_TO_PS_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (client_to_ssproxy_ProxyBuffer[i].EntryFull != 0) {
                    printf("CHECK(cl_recv): waiting for
client_to_ssproxy_ProxyBuffer[i] != 0\n\n");

                    thr_yield();
            }
            memcpy(client_to_ssproxy_ProxyBuffer[i].Data, buf, n);
            client_to_ssproxy_ProxyBuffer[i].DataSize = n;
            client_to_ssproxy_ProxyBuffer[i].EntryFull = 1;



        } //while
        if (n<=0) {
            printf("FATAL ERROR(p_cs clientside_recv): n<=0 while recv. from
client \n");
            printf("(Non-fatal if end of client-server communication)\n\n");
            close(client_sock);
            exit(1);
        }


} //clientside_recv_ThreadFunc
```

```
/*
**************************************************************************
********************* */

void *proxy_to_client_Control_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   i,j,k; //counter vars

        int  First_Time = 0;
        time_t  curr_time, start_time;
        int  NO_RECV_TIMEOUT = 10;


        /*
        * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
        * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
        * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
        * receiving threads
        */


        k=-1;  //Init counter var

  while (1) {


        for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {


          /*
          * Get data from proxy_side_recvBuffer and copy to
client_to_ssproxy_ProxyBuffer
```

```
*/

for (j=0; j<Connection_Number; j++) {

    //Increment k counter (used as client_to_ssproxy_ProxyBuffer
index)
    if (k < (PS_TO_C_PROXYBUFFER_SIZE-1))
        k++;
    else
        k = 0;

    while (proxy_side_recvBuffer[i][j].EntryFull != 1) {

        // ---------
        if (First_Time==0) {
            First_Time=1;
            start_time = time(NULL);
        }

        curr_time = time(NULL);

        if ( (float)difftime(curr_time, start_time) >
NO_RECV_TIMEOUT ) {
            printf("EXIT(recv_Control): %d sec w/ no new
data received\n\n",NO_RECV_TIMEOUT);
            exit(1);
        }

        // -------

        thr_yield();

    } //while

    First_Time = 0;  //Reset flag
```

```
                    while (ssproxy_to_client_ProxyBuffer[k].EntryFull != 0) {
                        printf("CHECK(recvCntrl): waiting for
ssproxy_to_client_ProxyBuffer[] != 0\n\n");
                        thr_yield();
                    }


                    /*
                    * Copy to ssproxy_to_client_ProxyBuffer
                    */
                    memcpy(ssproxy_to_client_ProxyBuffer[k].Data,
proxy_side_recvBuffer[i][j].Data, proxy_side_recvBuffer[i][j].DataSize);
                    ssproxy_to_client_ProxyBuffer[k].DataSize =
proxy_side_recvBuffer[i][j].DataSize;
                    proxy_side_recvBuffer[i][j].EntryFull = 0;
                    ssproxy_to_client_ProxyBuffer[k].EntryFull = 1;


            } //for - j


            //Wrap counter var
            if (i == R_PROXY_SIDE_BUFFER_SIZE-1)
                break;
            //      i=-1;



        }  //for - i


    }  //while (1)


}  //proxy_to_client_Control_ThreadFunc


/*
```

```
**************************************************************************
********************* */

void *client_to_proxy_Control_ThreadFunc(void *thr_ID_PTR) {


        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   i,j,k; //counter vars
        int  n;
        unsigned char  buf[SOCK_BUFFER_SIZE+2];
        int  send_sock_index = -1;
        struct timeval  *start_time;
        struct timeval  *curr_time;
        long  start_time_in_msec, curr_time_in_msec;


        /*
        * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
        * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
        * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
        * receiving threads
        */


        /*
        * Allocate memory to time vars
        */
        start_time = (struct timeval *) malloc(sizeof(struct timeval));
        curr_time = (struct timeval *) malloc(sizeof(struct timeval));



        /*
```

```
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */


            k=-1;  //Init counter var

   while (1) {


                /*
                * Get data from client_to_ssproxy_ProxyBuffer
                */
                //Increment k counter (used as client_to_ssproxy_ProxyBuffer index)
                if (k < (C_TO_PS_PROXYBUFFER_SIZE-1))
                        k++;
                else
                        k = 0;


                while (client_to_ssproxy_ProxyBuffer[k].EntryFull != 1) {
                        thr_yield();
                }



                /*
                * Copy to buf
                */
                memcpy(buf+LENGTH_HEADER_SIZE,
client_to_ssproxy_ProxyBuffer[k].Data,
client_to_ssproxy_ProxyBuffer[k].DataSize);
                n = client_to_ssproxy_ProxyBuffer[k].DataSize;
                client_to_ssproxy_ProxyBuffer[k].EntryFull = 0;
```

```
                /*
                * Insert Length header
                */
                buf[0] = (unsigned char)(n>>8);
                buf[1] = (unsigned char)n;



                #ifdef USE_DELAY

                // ----------------

                 if ( gettimeofday(start_time, NULL) <0 ) {
                        printf("ERROR(proxy_send[%d]): gettimeofday error\n\n",
thr_ID);
                        exit(1);
                 }

                 start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;


                while (1) {
                        if ( gettimeofday(curr_time, NULL) <0 ) {
                                printf("ERROR(proxy_send[%d]): gettimeofday
error\n\n", thr_ID);
                                exit(1);
                        }

                        curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;


                        if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                break;
```

```
                    //Since haven't reached timeout value yet have thread
sleep/yield
                    thr_yield();

                } //while

                // ----------------

                #endif



                if ((n+LENGTH_HEADER_SIZE)>0) {

                    // ---------
                    //Increment socket index
                    if (send_sock_index < (Connection_Number-1))
                        send_sock_index++;
                    else
                        send_sock_index = 0;
                    // ---------
                    if ( send(servsideproxy_sockfd[send_sock_index], buf,
n+LENGTH_HEADER_SIZE, 0) < 0) {
                                printf("ERROR(send/Control): While sending data
to serv-side proxy <0 \n\n");
                                exit(1);
                            } //if

                } //if n+LENGTH_HEADER_SIZE>0



  } //while (1)


} //client_to_proxy_Control_ThreadFunc


/*
*******************************************************************************
```

```
********** */
```

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>



// Debuging "defines" (enable debug printouts)
//#define new2_PRINT_DEBUG


//#define SERV_USE_NAME
#define SERV_USE_IP




#define S_PORTNUMBER 6006       //Server-side proxy listens on this port
#define  FILENAME   "p_ss_proxy_file.txt"  //File that contains address & port #
of server
#define  SOCK_BUFFER_SIZE   1024
#define  CS_TO_S_PROXYBUFFER_SIZE  200  //# of nodes in buffer.
#define  S_TO_CS_PROXYBUFFER_SIZE  5000  //4000 # of nodes in buffer.
//#define  PROXY_SIDE_BUFFER_SIZE   1000  //# of nodes in buffer.
#define  S_PROXY_SIDE_BUFFER_SIZE   2  //NOT USED 2000- # of nodes in buffer.
```

```
#define  R_PROXY_SIDE_BUFFER_SIZE   50  //# of nodes in buffer.

#define MAX_CONN_NUMB_PER_LOGICAL_CONN  5  //Max # of connections per
logical connection

#define LENGTH_HEADER_SIZE 2  //Length of Length header that is appended in
front of packets

//#define USE_DELAY
#define  DELAY_IN_MSEC 0




/*
* Function prototypes
*/
void *proxyside_recv_ThreadFunc(void *);
void *serverside_send_ThreadFunc(void *);
void *serverside_recv_ThreadFunc(void *);
void *proxy_to_server_Control_ThreadFunc(void *);
void *server_to_proxy_Control_ThreadFunc(void *);




struct BufferNode {
        int  EntryFull;  //Flag (serves as semaphore). If 1 then value stored, else
is empty and can be written into
        long int   DataSize;  //Size of data stored in entry
        char  Data[SOCK_BUFFER_SIZE];
};




int   sockfd, servlen, cli_len, childpid;
int   newsockfd[MAX_CONN_NUMB_PER_LOGICAL_CONN];  //Array of sockets.
Used to accept
struct sockaddr_in   cli_addr, serv_addr;
```

```
int   niaa;
char   buf_par[SOCK_BUFFER_SIZE];  //Buffer used by parent process to receive
# of connections opened
int   m_par, c_par;  //Counter vars
int   n_par;
int   Connection_Number;  //Number of connections to be opened for a given
proxy-to-proxy 'logical' connection
char   Connection_Number_str[31];  //String containing Connection_Number as
read from other proxy
int   server_sock;   //socket used to connect to server
struct BufferNode
csproxy_to_server_ProxyBuffer[CS_TO_S_PROXYBUFFER_SIZE];  //Buffer used
for client->server_side_proxy traffic
struct BufferNode
server_to_csproxy_ProxyBuffer[S_TO_CS_PROXYBUFFER_SIZE];  //Buffer used
for server_side_proxy->client traffic
struct BufferNode
proxy_side_sendBuffer[S_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that will

// be sent out to cl.side proxy
struct BufferNode
proxy_side_recvBuffer[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that

//is received fromt to cl.side proxy




int main(void) {

        //Create the socket
        if ( (sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            perror("ERROR: socket creation \n");
            exit(1);
        }
```

```
            //Create (server-side) proxy address
            memset(&serv_addr, 0, sizeof(struct sockaddr_in));

            serv_addr.sin_family = AF_INET;
            serv_addr.sin_port = htons(S_PORTNUMBER);
            servlen = sizeof(struct sockaddr_in);

            niaa = INADDR_ANY;
            memcpy(&serv_addr.sin_addr, &niaa, sizeof(long));


            //Bind socket to (server-side) proxy address
            if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
                perror("ERROR: bind \n");
                exit(1);
            }


            //Listen for client connections
            if ( listen(sockfd, 7) < 0 ) {
                perror("ERROR: listen \n");
                close(sockfd);
                exit(1);
            }


            for (;;) {

                cli_len = sizeof(cli_addr);
                newsockfd[0] = accept(sockfd, (struct sockaddr *) &cli_addr,
&cli_len);

                if (newsockfd[0] < 0) {
                        perror("ERROR: Can't create new socket (1st proxy listening
socket)\n");
                        exit(1);
                }


                /*
```

```
                * Read number of TOTAL connections to be opened (for this 'logical'
connection).
                * Format: string containing # followed by '\0'
                */

                strcpy(Connection_Number_str,"\0");   //Init

                m_par=0;   //Init counter
                while ((n_par = recv(newsockfd[0], buf_par, sizeof(buf_par), 0)) >
0) {
                        m_par = m_par+n_par;
                        strncat(Connection_Number_str, buf_par, n_par);
                        if ( Connection_Number_str[m_par-1] == '\0' )
                                break; //if '\0' in string then done so exit recv/while loop

                }
                if (n_par<=0) {
                        printf("FATAL ERROR: n_par<=0 while recv. connection #
(parent process)\n\n");
                        close(newsockfd[0]);
                        exit(1);
                }


                //Convert # of connections to int from string
                sscanf(Connection_Number_str,"%d", &Connection_Number);

                printf("CHECK(p_ss_proxy): %d connections to be opened (%d
max)\n\n",Connection_Number,MAX_CONN_NUMB_PER_LOGICAL_CONN);



                /*
                * Wait for Connection_Number connections to be opened.
NOTE:Connection_Number is total connections to be opened
                */
                for (c_par=1; c_par<Connection_Number; c_par++) {
                        newsockfd[c_par] = accept(sockfd, (struct sockaddr *)
&cli_addr, &cli_len);

                        if (newsockfd[c_par] < 0) {
```

```
                perror("ERROR: Can't create new socket \n");
                exit(1);
            }
        }



        //Create child process
        if ( (childpid = fork()) < 0 ) {
            perror("ERROR: Fork \n");
            exit(1);
        }
        else if (childpid == 0) {   /* ****** child process ****** */



            /* Child process vars */
            struct hostent   *hp;   //used in connecting to server
            struct sockaddr_in   name;   //used in connecting to server
            int   len;   //used in connecting to server
            int   thr_concurrency_counter;
            thread_t
ThreadID[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4]; //Array of thread IDs
for thr_create calls
            int
thr_ID_PTR[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4];  //Array of thr_ID
pointers used in thr_create calls
            FILE   *fd;
            int   Server_PortNumber;   //Port on which server listens
            char   Server_Name[64];   //DNS name of server
            char Server_IP[64];  //IP of server
            ulong_t  addr;

            int   i,j;  //counter var
            size_t
TaskStatus[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4];  //Used in thr_join()
calls



            close(sockfd);        //close original socket
```

```
//------------------code------------------

printf("CHECK(p_ss_proxy): CHILD process created \n\n");


/*
* Initialize proxy buffers (i.e. set all flags to 0)
*/
for (i=0; i<S_PROXY_SIDE_BUFFER_SIZE; i++) {
        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)    {
                proxy_side_sendBuffer[i][j].EntryFull = 0;
        }
}
for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {
        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)    {
                proxy_side_recvBuffer[i][j].EntryFull = 0;
        }
}



for (i=0; i<CS_TO_S_PROXYBUFFER_SIZE; i++) {
        csproxy_to_server_ProxyBuffer[i].EntryFull = 0;
}
for (i=0; i<S_TO_CS_PROXYBUFFER_SIZE; i++) {
        server_to_csproxy_ProxyBuffer[i].EntryFull = 0;
}



/*
* Read file to obtain address (name & port #) of server
* Format: "port# server_name" ("'s not included )
*/
//sem_lock not needed

if ( (fd = fopen(FILENAME, "r")) == NULL) {
        perror("ERROR(p_ss_proxy): Couldn't open file \n");
```

```
                exit(1);
        }

        #ifdef SERV_USE_NAME
        fscanf(fd, "%d %s", &Server_PortNumber, Server_Name);
        #endif
        #ifdef SERV_USE_IP
        fscanf(fd, "%d %s", &Server_PortNumber, Server_IP);
        #endif


        //Check values read from file
        #ifdef SERV_USE_NAME
        printf("CHECK(p_ss_proxy): Server_Name: %s\n",
Server_Name);
        #endif
        #ifdef SERV_USE_IP
        printf("CHECK(p_ss_proxy): Server_IP: %s\n", Server_IP);
        #endif

        printf("CHECK(p_ss_proxy): Server_PortNumber: %d \n",
Server_PortNumber);


        fclose(fd);

        //sem_unlock not needed



        /*
        * Open connection to server
        */
        //-----------
        //Get server's address
        #ifdef SERV_USE_NAME
        if ( (hp = gethostbyname(Server_Name)) == NULL ) {
                printf("ERROR(p_ss_proxy): Can't get server
address(%s)\n\n", Server_Name);
                exit(1);
        }
```

```
#endif

#ifdef SERV_USE_IP
if ((int)(addr = inet_addr(Server_IP)) == -1) {
        printf("IP-address must be of the form a.b.c.d\n");
        exit (2);
}

hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
if (hp == NULL) {
        printf("host information for %s not found\n",
Server_IP);
         exit (3);
}
#endif



//Create socket
if ( (server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("ERROR(p_ss_proxy): Can't create socket for
server\n\n");
        exit(1);
}


//Create server address
memset(&name, 0, sizeof(struct sockaddr_in));

name.sin_family = AF_INET;
name.sin_port = htons(Server_PortNumber);
memcpy(&name.sin_addr, hp->h_addr_list[0], hp->h_length);
len = sizeof(struct sockaddr_in);


//Connect to server
if ( connect(server_sock, (struct sockaddr *) &name, len) < 0
) {
        printf("ERROR: Can't connect to server \n\n");
        exit(1);
}
```

```
                    //----------


                    /*
                    * Init/fill-in thr_ID array. Thread IDs start from ZERO !!!
                    */
                    for (i=0; i<(MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4);
i++)
                            thr_ID_PTR[i] = i;



                    /*
                    *
                    * NOTE: proxy-side sending threads have thread IDs from (0
to Connection_Number-1) and
                    * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
                    *
                    */



                    /*
                    * Spawn the proxy-side sending/receiving threads. Pair for
each connection w/ client-side proxy
                    */

                    for (i=Connection_Number; i<(2*Connection_Number); i++) {
                            thr_create(NULL, 0, proxyside_recv_ThreadFunc, (void
*) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    } //for



                    if (i != (2*Connection_Number))
                            printf("ERROR/CHECK(p_ss_proxy): i counter <>
2*Connection_Number \n");
```

```
                    /*
                    * Spawn the server-side sending/receiving threads. Just one
pair for the single connection to server
                    */
                    thr_create(NULL, 0, serverside_send_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, serverside_recv_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);


                    /*
                    * Spawn Control threads
                    */
                    i++;
                    thr_create(NULL, 0, server_to_proxy_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, proxy_to_server_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);



                    /*
                    *Set the thread concurrency. Could improve thread scheduling.
                    */
                    thr_concurrency_counter = Connection_Number*1+4 + 1;
                    thr_setconcurrency(thr_concurrency_counter);



                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    for (i=Connection_Number; i<(2*Connection_Number+4); i++) {
                            thr_join(ThreadID[i], 0, (void *) &TaskStatus[i]);
                    }
```

```
            } //endelseif                    /* ****** child process ****** */
            /*
            * Close in parent the "passed off to child" sockets
            */
            for (c_par=0; c_par<Connection_Number; c_par++) {
                  close(newsockfd[c_par]);
            }


         } //endfor

} /*main*/



/*
*************************************************************************
********************* */



void *proxyside_recv_ThreadFunc(void *thr_ID_PTR) {

         /*
         * Get thread ID
         */
         int  thr_ID = *((int *) thr_ID_PTR);

         int   n,m;
         int   i,k;  //counter vars
         char  *buf;  //Buffer into which received data is copied into & data
pasted to
         char  *bufStart;  //Used to store temporarily the socket buffer starting
address
         int  buf_size;  //Used because sizeof(buf) doesn't work for char *
         unsigned int  Packet_Length;  //Holds packet length derived from arrived
```

```
packet's Length header


        //Init buffer
        buf_size = (SOCK_BUFFER_SIZE+2);  //buf_size was
SOCK_BUFFER_SIZE
        buf = (char *) malloc( buf_size*sizeof(char) );



        /*
        * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
        * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
        * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
        * receiving threads
        */



        i=0; //Init counter

        while (1) {


          /*
          * Get exaclty LENGTH_HEADER_SIZE bytes of new packet i.e. the
Length header
          */

          m = 0;
          while (m<2) {

                if ((n=recv(newsockfd[thr_ID-Connection_Number], buf+m,
LENGTH_HEADER_SIZE-m, 0)) > 0) {
                      m = m+n;
                } //if
                else{
                      printf("FATAL ERROR(prx_side_recv_Thread): n<=0
```

```
while recv. Length header\n\n");
                        close(newsockfd[thr_ID-Connection_Number]);
                        exit(1);
                }

        } //while


        #ifdef new2_PRINT_DEBUG
        printf("CHECK: right after recv() #1 \n\n");
        #endif



        /*
        * Get length from Length field
        */
        Packet_Length = buf[0];
        Packet_Length = (Packet_Length<<8) | (buf[1]);
        printf("CHECK: Packet_Length: %d\n\n", Packet_Length);



        /*
        * Receive entire packet now
        */
        m = 0;

        while (m<Packet_Length) {

                if ( (n=recv(newsockfd[thr_ID-Connection_Number], buf+m,
Packet_Length-m, 0)) > 0) {

                        m = m+n;

                } //if
                else {
                        printf("FATAL ERROR(prx_side_recv_Thread): n<=0
while recv. remaining packet \n\n");
                        close(newsockfd[thr_ID-Connection_Number]);
```

```
                exit(1);
            } //else


        } //while



        //Error check
        if (m != Packet_Length) {
            printf("ERROR(recv[%d]): m != Packet_Length (is >) \n\n",
thr_ID);
            close(newsockfd[thr_ID-Connection_Number]);
            exit(1);
        } //if




        /*
        * Now that got entrire packet, write it (w/o Length header) to
proxy_side_recvBuffer
        */

        while (proxy_side_recvBuffer[i][thr_ID-Connection_Number].EntryFull
!= 0) {
            printf("CHECK(prx_recv[%d]): Waiting for
proxy_side_recvBuffer[][]!=0\n\n", thr_ID);
            thr_yield();
        } //while

        memcpy(proxy_side_recvBuffer[i][thr_ID-Connection_Number].Data,
buf, Packet_Length);
        proxy_side_recvBuffer[i][thr_ID-Connection_Number].DataSize =
Packet_Length;
        proxy_side_recvBuffer[i][thr_ID-Connection_Number].EntryFull = 1;


        //Increment counter
        if (i < (R_PROXY_SIDE_BUFFER_SIZE-1))
            i++;
```

```
            else
                    i = 0;

        } //while(1)

} //proxyside_recv_ThreadFunc


/*
**********************************************************************************
********************* */


void *serverside_send_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i; //counter var
        char   buf[SOCK_BUFFER_SIZE];



        i=-1;
        while (1) {

            /*
            * Get data from csproxy_to_server_ProxyBuffer;
            */
            //Increment counter
            if (i < (CS_TO_S_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (csproxy_to_server_ProxyBuffer[i].EntryFull != 1) {
                    thr_yield();
```

```
                }
                memcpy(buf, csproxy_to_server_ProxyBuffer[i].Data,
csproxy_to_server_ProxyBuffer[i].DataSize);
                n = csproxy_to_server_ProxyBuffer[i].DataSize;
                csproxy_to_server_ProxyBuffer[i].EntryFull = 0;



                /*
                * Send data to server
                */
                if (n>0) {
                        if ( send(server_sock, buf, n, 0) < 0) {
                                printf("ERROR(p_ss_proxy/server_send_thread): While
sending data to server <0 \n\n");
                                exit(1);
                        }
                }


        } //while


} //serverside_send_ThreadFunc


/*
**********************************************************************************
********************** */


void *serverside_recv_ThreadFunc(void *thr_ID_PTR) {


        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i; //counter var
```

```
        char   buf[SOCK_BUFFER_SIZE];



        i=-1;
        while ((n = recv(server_sock, buf, sizeof(buf), 0)) > 0) {

            /*
            * Copy received data to server_to_csproxy_ProxyBuffer;
            */

            //Increment counter
            if (i < (S_TO_CS_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (server_to_csproxy_ProxyBuffer[i].EntryFull != 0) {
                    printf("CHECK(serv_recv): waiting for
server_to_csproxy_ProxyBuffer[i] != 0 \n\n");

                    thr_yield();
            }
            memcpy(server_to_csproxy_ProxyBuffer[i].Data, buf, n);
            server_to_csproxy_ProxyBuffer[i].DataSize = n;
            server_to_csproxy_ProxyBuffer[i].EntryFull = 1;



        } //while
        if (n<=0) {
            printf("FATAL ERROR: n<=0 while recv. from server (server-side
proxy) \n\n");
            close(server_sock);
            exit(1);
        }


} //serverside_recv_ThreadFunc
```

```
/*
**********************************************************************************
********************** */
void *proxy_to_server_Control_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   i,j,k; //counter vars


        /*
        * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
        * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
        * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
        * receiving threads
        */



        k=-1;  //Init counter var

   while (1) {

        for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {


            /*
            * Get data from proxy_side_recvBuffer and copy to
csproxy_to_server_ProxyBuffer
            */

            for (j=0; j<Connection_Number; j++) {
```

```
                    //Increment k counter (used as
csproxy_to_server_ProxyBuffer index)
                    if (k < (CS_TO_S_PROXYBUFFER_SIZE-1))
                            k++;
                    else
                            k = 0;


                    while (proxy_side_recvBuffer[i][j].EntryFull != 1) {
                            thr_yield();
                    }
                    while (csproxy_to_server_ProxyBuffer[k].EntryFull != 0) {
                            printf("CHECK(recvControl): waiting for
csproxy_to_server_ProxyBuffer[i] != 0 \n\n");

                            thr_yield();
                    }


                    /*
                    * Copy to csproxy_to_server_ProxyBuffer
                    */
                    memcpy(csproxy_to_server_ProxyBuffer[k].Data,
proxy_side_recvBuffer[i][j].Data, proxy_side_recvBuffer[i][j].DataSize);
                    csproxy_to_server_ProxyBuffer[k].DataSize =
proxy_side_recvBuffer[i][j].DataSize;
                    proxy_side_recvBuffer[i][j].EntryFull = 0;
                    csproxy_to_server_ProxyBuffer[k].EntryFull = 1;


            } //for - j


            //Wrap counter var
            if (i == R_PROXY_SIDE_BUFFER_SIZE-1)
                    break;
            //      i=-1;


        } //for - i
```

```
    } //while (1)


} //proxy_to_server_Control_ThreadFunc

/*
**********************************************************************************
********************* */

void *server_to_proxy_Control_ThreadFunc(void *thr_ID_PTR) {


            /*
            * Get thread ID
            */
            int  thr_ID = *((int *) thr_ID_PTR);

            int   i,j,k; //counter vars
            int  n;
            char  buf[SOCK_BUFFER_SIZE+2];
            int  send_sock_index = -1;
            struct timeval  *start_time;
            struct timeval  *curr_time;
            long  start_time_in_msec, curr_time_in_msec;



            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */
```

```
            /*
            * Allocate memory to time vars
            */
            start_time = (struct timeval *) malloc(sizeof(struct timeval));
            curr_time = (struct timeval *) malloc(sizeof(struct timeval));


            k=-1;  //Init counter var

            while (1) {

                /*
                * Get data from server_to_csproxy_ProxyBuffer
                */

                //Increment k counter (used as server_to_csproxy_ProxyBuffer
index)
                if (k < (S_TO_CS_PROXYBUFFER_SIZE-1))
                        k++;
                else
                        k = 0;


                while (server_to_csproxy_ProxyBuffer[k].EntryFull != 1) {
                        thr_yield();
                }



                /*
                * Copy to proxy_side_sendBuffer
                */
                memcpy(buf+LENGTH_HEADER_SIZE,
server_to_csproxy_ProxyBuffer[k].Data,
server_to_csproxy_ProxyBuffer[k].DataSize);
                n = server_to_csproxy_ProxyBuffer[k].DataSize;
                server_to_csproxy_ProxyBuffer[k].EntryFull = 0;


                /*
```

```
                * Insert Length header
                */
                buf[0] = (unsigned char)(n>>8);
                buf[1] = (unsigned char)n;
                #ifdef USE_DELAY

                // ----------------

                 if ( gettimeofday(start_time, NULL) <0 ) {
                        printf("ERROR(proxy_send[%d]): gettimeofday error\n\n",
thr_ID);
                        exit(1);
                 }

                 start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;


                round_count++;
                while (1) {
                        if ( gettimeofday(curr_time, NULL) <0 ) {
                                printf("ERROR(proxy_send[%d]): gettimeofday
error\n\n", thr_ID);
                                exit(1);
                        }

                        curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;


                        if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                break;


                        //Since haven't reached timeout value yet have thread
sleep/yield
                        thr_yield();

                } //while
```

```
                // ----------------

                #endif
                if (n+LENGTH_HEADER_SIZE>0) {

                        // ---------
                        //Increment socket index
                        if (send_sock_index < (Connection_Number-1))
                                send_sock_index++;
                        else
                                send_sock_index = 0;
                        // ---------
                        if ( send(newsockfd[send_sock_index], buf,
n+LENGTH_HEADER_SIZE, 0) < 0) {
                                printf("ERROR(send/Control):While sending data to \n");
                                printf("client-side proxy <0 \n\n");
                                exit(1);
                        }  //if

                } //if n+LENGTH_HEADER_SIZE>0



        } //while (1)


} //server_to_proxy_Control_ThreadFunc


/*
**********************************************************************
********************** */
```

APPENDIX C

C LANGUAGE LISTING FOR UDP CONNECTION-SPLITTING PROXIES

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include "fec.c"



// Debuging "defines" (enable debug printouts)
//#define ENC_DEBUG_ON
//#define new2_PRINT_DEBUG
#define FLOW_CNTRL_DEBUG_PRINT
//#define RECV_CONTROL_DEBUG


//Choose whether serv-side proxy Name or IP read from file
//#define SS_PRX_USE_NAME
#define SS_PRX_USE_IP
//#define EXTRA_RECV_PACKET_CHECKS



#define SSP_UDP_BASE_PORTNUMBER 8000  //Server-side proxy UDP base
portnumber i.e. (SSP_UDP_BASE_PORTNUMBER+thread ID)
            //(or (SSP_UDP_BASE_PORTNUMBER+thread ID-Connection_Number)) is
```

```
port number to be used by each thread
#define CSP_UDP_BASE_PORTNUMBER 7000  //Client-side proxy UDP base
portnumber
#define C_PORTNUMBER 6005        //Client-side proxy listens on this port
#define  FILENAME   "p_cs_proxy_file.txt"  //File that contains address & port #
of server-side proxy
#define  SOCK_BUFFER_SIZE   2048
#define  PS_TO_C_PROXYBUFFER_SIZE  4000  //# of nodes in buffer
#define  C_TO_PS_PROXYBUFFER_SIZE  200  //# of nodes in buffer
//#define  PROXY_SIDE_BUFFER_SIZE   10  //# of nodes in buffer
#define  R_PROXY_SIDE_BUFFER_SIZE   6000  //# of nodes in buffer
#define  S_PROXY_SIDE_BUFFER_SIZE   50  //# of nodes in buffer

#define MAX_CONN_NUMB_PER_LOGICAL_CONN  5  //Max # of connections per
logical connection
#define EXIT_TIME_DELAY  3  //Number of sec of time delay before child proc.
terminated after client has closed
#define RNR_PROXY_RECV_BUF_SIZE_THRESH 0.8  //Percentage full treshhold of
proxy recv.buffer which if exceeded
                                                        //results in RNR
message being sent to serv-side proxy
#define RR_PROXY_RECV_BUF_SIZE_THRESH 0.6  //Percentage full treshhold of
proxy recv.buffer which if exceeded
                                                        //results in RNR
message being sent to serv-side proxy

#define K_DESIRED 2  //# of client_to_ssproxy_ProxyBuffer we WANT to
read(actual # read may differ)
#define THR_YIELD_MAX 1  //Max number of times will thread yield to read entry
from client_to_ssproxy_ProxyBuffer
                                        //Set it to 0 if don't want to wait at all

#define N_FACTOR 2  //N_FACTOR*k packets will be produced by encoding and
sent out(k = #data packets)
#define ENC_DEBUG_FILENAME  "er_cs_enc_dbg_file.txt" //File descriptor of a file
used for debug output in client-to-proxy thread
#define HEADER_SIZE 8  //Size (in bytes) of header added to encoded packets
(0x00,k, n, seq.#,length(2-bytes),packet in block,0x00)
#define MAX_SEQ_NUMB_VALUE 255
#define HEADER_SRT_END_BYTES_CODE_VALUE (unsigned char)0x01
#define DEC_ARRAY_SIZE 10  //Number of array entries for decoding array
```

```
(array of gf*)
#define DEC_ARR_BUF_SIZE SOCK_BUFFER_SIZE  //Size of each decoding array
buffer
#define ENC_PACKET_BUF_SIZE  SOCK_BUFFER_SIZE  //Size of buffer containing
encoded packet(what you get from enc.func.)
#define ENC_SOURCE_BUF_BUFFER_SIZE  SOCK_BUFFER_SIZE  //Size of buffers
containing source data headed for encoding
#define CL_RECV_SOCK_BUFFER_SIZE  1024  //Socket buffer size for client
receiving
#define TEMP_CUMUL_BUF_SIZE_TRESH   CL_RECV_SOCK_BUFFER_SIZE  //If total
# of bytes to be encoded(&sent-out) each time
                                                        //is smaller than this value
then amalgamate into one big packet(for efficiency)

#define  DELAY_IN_MSEC 0
//#define USE_DELAY



/*
* Function prototypes
*/
void *proxyside_recv_ThreadFunc(void *);
void *clientside_send_ThreadFunc(void *);
void *clientside_recv_ThreadFunc(void *);
void *Cong_Msg_Recv_ThreadFunc(void *);
void *client_to_proxy_Control_ThreadFunc(void *);
void *proxy_to_client_Control_ThreadFunc(void *);




struct BufferNode {
        int  EntryFull;  //Flag (serves as semaphore). If 1 then value stored, else
is empty and can be written into
        long int   DataSize;  //Size of data stored in entry
        unsigned char  Data[SOCK_BUFFER_SIZE + HEADER_SIZE];
        int  Skip;  //Used only for proxy_side_recvBuffer. Recv. thread sets flag
```

```
to 1 to tell control to skip cell
            int  Control_Skip;  //Flag set by recv_Control "stating" that it has
skeeped the entry (because doesn't need
                                        //more data for current block)
};



int   sockfd, servlen, cli_len, childpid;
int   client_sock;   //Socket created from cl.side-proxy listening
struct sockaddr_in   cli_addr, serv_addr;
int   niaa;
int   servsideproxy_sockfd[MAX_CONN_NUMB_PER_LOGICAL_CONN]; //sockets
used to connect to serv-side proxy
int tcp_sock;  //Socket for TCP connection used to tranmsit to server-side proxy
the # of UDP connections to be used
struct BufferNode  client_to_ssproxy_ProxyBuffer[C_TO_PS_PROXYBUFFER_SIZE];
//Buffer used for client->server_side_proxy traffic
struct BufferNode  ssproxy_to_client_ProxyBuffer[PS_TO_C_PROXYBUFFER_SIZE];
//Buffer used for server_side_proxy->client traffic
struct BufferNode
proxy_side_sendBuffer[S_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that will be

//sent out to server-side proxy
struct BufferNode
proxy_side_recvBuffer[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that is

//received fromt to cl.side proxy
int Connection_Number;  //Number of connections to be opened for a given proxy-
to-proxy 'logical' connection
struct hostent   *hp;   //used in connecting to server

int  RecvBufSizeCounter = 0;  //Used for congestion control(# of full cells of proxy
recv. buffer)
int  RecvBufSizeCounter_copy = 0;  //Copy of above counter. Used to unlock
mutex quickly
int  HoldBack = 0;  //Flag used for congestion control i.e. when 1 then send
```

```
threads pause
mutex_t  RecvBufSizeCounter_Lock;  //RecvBufSizeCounter semaphore
mutex_t  HoldBack_Lock;  //HoldBack semaphore
int  PROXY_RECV_BUF_CAPACITY;  //Capacity of proxy-side recv buffer. Is
R_PROXY_SIDE_BUFFER_SIZE*(#UDP connections)
mutex_t
Control_Skip_Lock[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LOGIC
AL_CONN];  //Mutex for setting/reseting Control_Skip flag. One mutex for each
cell of proxy_side_recvBuffer
int  mi,mj; //Counter vars




int main(void) {


            /*
            * Init HoldBack mutex var - Used for congestion control i.e. when 1 then
send threads pause
            */
   mutex_init(&HoldBack_Lock, USYNC_THREAD, (void *) NULL);


            /*
            * Init RecvBufSizeCounter mutex var - Used for congestion control(# of
full cells of proxy recv. buffer)
            */
   mutex_init(&RecvBufSizeCounter_Lock, USYNC_THREAD, (void *) NULL);




            /*
            * Init Control_Skip_Lock mutex vars (one mutex for each cell of
proxy_side_recvBuffer)
            */
            for (mi=0; mi<R_PROXY_SIDE_BUFFER_SIZE; mi++)
                for (mj=0; mj<MAX_CONN_NUMB_PER_LOGICAL_CONN; mj++)
```

```
                    mutex_init(&Control_Skip_Lock[mi][mj], USYNC_THREAD,
(void *) NULL);



    //Create the socket
    if ( (sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        perror("ERROR: socket creation \n");
        exit(1);
    }


    //Create (client-side) proxy address
    memset(&serv_addr, 0, sizeof(struct sockaddr_in));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(C_PORTNUMBER);
    servlen = sizeof(struct sockaddr_in);

    niaa = INADDR_ANY;
    memcpy(&serv_addr.sin_addr, &niaa, sizeof(long));


    //Bind socket to address
    if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
        perror("ERROR: bind \n");
        exit(1);
    }


    //Listen for client connections
    if ( listen(sockfd, 7) < 0 ) {
        perror("ERROR: listen \n");
        close(sockfd);
        exit(1);
    }
```

```
        for (;;) {

            cli_len = sizeof(cli_addr);
            client_sock = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);

            if (client_sock < 0) {
                    perror("ERROR: Can't create new socket (from cl.side proxy
listening socket) \n");
                    exit(1);
            }



            //Create child process
            if ( (childpid = fork()) < 0 ) {
                    perror("ERROR: Fork \n");
                    exit(1);
            }
            else if (childpid == 0) {   /* ****** child process ****** */



                    /* Child process vars */

                    struct sockaddr_in   name;   //used in connecting to server
                    struct sockaddr_in   cli_name;   //used for server-side proxy
UDP sockets
                    int   len;   //used in connecting to server
                    int   n;
                    int   thr_concurrency_counter;
                    thread_t
ThreadID[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1]; //Array of thread IDs
for thr_create calls
                    int
thr_ID_PTR[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1];  //Array of thr_ID
pointers used in thr_create calls
                    FILE   *fd;
                    int   ServSideProxy_PortNumber;   //Port on which server
listens
                    unsigned char   ServSideProxy_Name[64];   //DNS name of
server
```

```
                int   i,j;  //counter var
                size_t
TaskStatus[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1];  //Used in
thr_join() call
                unsigned char buf[SOCK_BUFFER_SIZE];  //Buffer used by
parent to send to other proxy # of connections to be opened
                unsigned char Connection_Number_str[31];  //String
containing Connection_Number(as to be sent to other proxy)
                int  nn;

                char  ServSideProxy_IP[50];  //Used for reading serv-side
proxy IP address from file
                ulong_t  addr;


                close(sockfd);        //close original socket


                //-------------------code-------------------



                /*
                * Init the erasure code encoder/decoder
                */
                init_fec();




                /*
                * Read file to obtain address (name & port #) of server-side
proxy
                * Format: "port# proxy_name" ("'s not included )
                */
                //sem_lock not needed

                if ( (fd = fopen(FILENAME, "r")) == NULL) {
                        perror("ERROR(p_cs_proxy): Couldn't open file \n");
                        exit(1);
                }
```

```
                    #ifdef SS_PRX_USE_NAME
                     fscanf(fd, "%d %s", &ServSideProxy_PortNumber,
ServSideProxy_Name);
                    #endif

                    #ifdef SS_PRX_USE_IP
                     fscanf(fd, "%d %s", &ServSideProxy_PortNumber,
ServSideProxy_IP);
                    #endif


                    //Check values read from file
                    #ifdef SS_PRX_USE_NAME
                    printf("CHECK(p_cs_proxy): ServSideProxy_Name: %s\n",
ServSideProxy_Name);
                    #endif

                    #ifdef SS_PRX_USE_IP
                    printf("CHECK(p_cs_proxy): ServSideProxy_IP: %s\n",
ServSideProxy_IP);
                    #endif

                    printf("CHECK(p_cs_proxy): TCP ServSideProxy_PortNumber:
%d \n", ServSideProxy_PortNumber);

                    fclose(fd);

                    //sem_unlock not needed



                    /*
                    * Open connection to server-side proxy
                    */

                    //----------
                    //Get server-side proxy's address

                    #ifdef SS_PRX_USE_NAME
                    if ( (hp = gethostbyname(ServSideProxy_Name)) == NULL ) {
                          printf("ERROR(p_cs_proxy): Can't get server-side proxy
```

```
address(%s)\n\n", ServSideProxy_Name);
                        exit(1);
                }
                #endif

                #ifdef SS_PRX_USE_IP
                if ((int)(addr = inet_addr(ServSideProxy_IP)) == -1) {
                        printf("IP-address must be of the form a.b.c.d\n");
                        exit (2);
                }
                hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
                if (hp == NULL) {
                        printf("host information for %s not found\n",
ServSideProxy_IP);
                        exit (3);
                }
                #endif



                //Create server-side proxy address
                memset(&name, 0, sizeof(struct sockaddr_in));

                name.sin_family = AF_INET;
                name.sin_port = htons(ServSideProxy_PortNumber);
                memcpy(&name.sin_addr, hp->h_addr_list[0], hp->h_length);
                len = sizeof(struct sockaddr_in);



                /*
                *
                * tcp_sock is TCP socket used to communicate to server-side
proxy the # of connections
                * to be used. After connection # is sent the TCP connection is
closed with UDP being used
                * from that point on to tranmsit/receive data
                *
                * servsideproxy_sockfd[i] are UDP sockets to be used
                *
                */
```

```
                    //Create TCP connection socket
                    if ( (tcp_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                            printf("ERROR(p_cs_proxy): Can't create socket for
server-side proxy \n\n");
                            exit(1);
                    }



                    //Establish TCP connection to server-side proxy
                    if ( connect(tcp_sock, (struct sockaddr *) &name, len) < 0 ) {
                            printf("ERROR(p_cs_proxy): Can't establish TCP
connection to server-side proxy \n\n");
                            exit(1);
                    }

                    printf("CHECK(p_cs_proxy): TCP connection to serv-side proxy
opened \n\n");




                    /*
                    * Read # of TOTAL connections to be opened (for each
'logical' connection to serv-side proxy)
                    * & send it to server-side proxy over the TCP connection.
                    * Format: string containing # followed by '\0'
                    */
                     printf("Enter # of phys. connections per logical connection (%d
max): ", MAX_CONN_NUMB_PER_LOGICAL_CONN);
                     scanf("%d", &Connection_Number);
                    //Connection_Number = 1;


                    sprintf(Connection_Number_str, "%d", Connection_Number);

                    //Copy connection # in buf to send out to serv-side proxy &
set buf size
                    strcpy(buf, Connection_Number_str);
                    n = strlen(buf) + 1;  //strlen()+1 to account for the '\0' char
that must be included
```

```
                    if ( send(tcp_sock, buf, n , 0) < 0) {
                            printf("ERROR(p_cs_proxy): While sending connection #
to serv-side proxy <0 \n\n");
                            exit(1);
                    }  //if



                    /*
                    * Now that we got the # of UDP connections to be used
calculate the proxy recv. buffer capacity
                    */
                    PROXY_RECV_BUF_CAPACITY =
R_PROXY_SIDE_BUFFER_SIZE*Connection_Number;


                    /*
                    * - Now that UDP connection # has been sent to server-side
proxy close the TCP connection
                    */
                    //close(tcp_sock); - need TCP socket to send/receive RNR/RR
flow control messages



                    /*
                    * Open Connection_Number UDP sockets (numbered from 0 to
Connection_Number-1).
                    */
                    for (i=0; i<Connection_Number; i++) {

                            //Create socket
                            if ( (servsideproxy_sockfd[i] = socket(AF_INET,
SOCK_DGRAM, 0)) < 0) {
                                    printf("ERROR(er_cs_proxy): Can't create UDP
socket for server-side proxy for i:%d\n\n", i);
                                    exit(1);
                            }
```

```
                        //Create the address to bind the UDP socket. Port #:
(CSP_UDP_BASE_PORTNUMBER + i)
                        memset(&cli_name, 0, sizeof(struct sockaddr_in));
                        cli_name.sin_family = AF_INET;
                        cli_name.sin_port =
htons(CSP_UDP_BASE_PORTNUMBER + i);
                        nn = INADDR_ANY;
                        memcpy(&cli_name.sin_addr, &nn, sizeof(long));


                        //Bind UDP socket to above address
                        if ( bind(servsideproxy_sockfd[i], (struct sockaddr *)
&cli_name, sizeof(struct sockaddr_in)) <0 ) {
                                printf("ERROR: Can't bind UDP socket to port
%d\n\n",CSP_UDP_BASE_PORTNUMBER+i);
                                exit(1);
                        }


                } //for



                /*
                * Initialize proxy buffers (i.e. set all flags to 0)
                */
                for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {
                        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)            {
                                proxy_side_recvBuffer[i][j].EntryFull = 0;
                                proxy_side_recvBuffer[i][j].Skip = 0;
                                proxy_side_recvBuffer[i][j].Control_Skip = 0;
                        }
                }
                for (i=0; i<S_PROXY_SIDE_BUFFER_SIZE; i++) {
                        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)            {
                                proxy_side_sendBuffer[i][j].EntryFull = 0;
                        }
                }
```

```
                for (i=0; i<PS_TO_C_PROXYBUFFER_SIZE; i++) {
                        ssproxy_to_client_ProxyBuffer[i].EntryFull = 0;
                }
                for (i=0; i<C_TO_PS_PROXYBUFFER_SIZE; i++) {
                        client_to_ssproxy_ProxyBuffer[i].EntryFull = 0;
                }


                //----------


                /*
                * Init/fill-in thr_ID array. Thread IDs start from ZERO !!!
                */
                for (i=0; i<(MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1);
i++)
                        thr_ID_PTR[i] = i;



                /*
                *
                * NOTE: proxy-side sending threads have thread IDs from (0
to Connection_Number-1) and
                * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
                *
                */



                /*
                * Spawn the server-side proxy sending/receiving threads and
receive buffering threads.
                * Pair for each connection w/ serv-side proxy.
                */

                for (i=Connection_Number; i<(2*Connection_Number); i++) {
                        thr_create(NULL, 0, proxyside_recv_ThreadFunc, (void
*) &thr_ID_PTR[i], 0, &ThreadID[i]);
                } //for
```

```
                    if (i != (2*Connection_Number)) {
                            printf("ERROR/CHECK(p_cs_proxy): i counter <>
2*Connection_Number \n");
                            exit(1);
                    }


                    /*
                    * Spawn the client-side sending/receiving threads. Just one pair
for the single connection to client
                    */
                    thr_create(NULL, 0, clientside_send_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, clientside_recv_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);



                    /*
                    * Spawn Control threads
                    */
                    i++;
                    thr_create(NULL, 0, client_to_proxy_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, proxy_to_client_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);



                    /*
                    * Spawn RNR/RR message receiving thread that will receive
congestion messages from serv-side proxy
                    */
                    i++;
                    thr_create(NULL, 0, Cong_Msg_Recv_ThreadFunc, (void *)
NULL, 0, &ThreadID[i]);


                    /*
```

```
                    *Set the thread concurrency. Could improve thread scheduling.
                    */
                    thr_concurrency_counter = Connection_Number*1+4+1;
                    thr_setconcurrency(thr_concurrency_counter);



                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    for (i=Connection_Number; i<(Connection_Number*2+4+1);
i++) {
                            thr_join(ThreadID[i], 0, (void *) &TaskStatus[i]);
                    }


            } //endelseif                    /* ****** child process ****** */



            /*
            * Close in parent the "passed off to child" socket
            */
            close(client_sock);


        } //endfor

} /*main*/



/*
**********************************************************************************
********************* */
```

```
void *proxyside_recv_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i,k,m;  //counter vars
        unsigned char  *buf;  //socket buffer
        unsigned char  *bufStart;  //Used to store temporarily the socket buffer
starting address
        unsigned char  *tempbuf;  //Temp. buffer used to copy segments read
from buf & copy into proxy recv buffer at end
        int  tempbuf_Size;  //Current capacity of tempbuf i.e. # of bytes
currently stored
        unsigned char  *tempbuf_Start;  //Used to store temporarily tempbuf
starting address
        int  tempbuf_Capacity; //Capacity of tempbuf. Value set later on &
must remain constant
        int buf_size;  //Size of buf. Used because sizeof(buf) doesn't work for
char *
        unsigned char  *c_location;  //Contains result of memccpy operation
        int  Total_buf_BytesRead;//Number of bytes already read from socket
buffer(buf). Used in writing received data to proxy_side_recvBuffer
        int  ZeroByteAtEndLastRound=0; //Used as a flag. True(i.e. 1) when last
byte in buf(in 1st or nth round) was a 0x00 byte
        int  ConcatRemaining_buf_contents_to_buf_for_next_round = 0;
//Flag
        int  concated_to_buf_start_bytes = 0;
        unsigned char *new_received_data_start_addr_in_buf; //In case of
concating data at start of buf, this is the pointer to
                                        //the first byte in buf of the newly
received(from the network) data
        unsigned char  *dummy_buf;  //Buffer into to which socket data is
copoed into initially
        int  dummy_buf_size;  //Capacity of dummy_buf buffer
        int  len;  //Used in recvfrom() call
        struct sockaddr_in  from;  //Address of sender of data i.e. server-side
proxy. Used in recvfrom() call
        int  Header_Derived_Packet_Length;  //Packet length as derived from
```

```
header
            unsigned char  hb_size, lb_size;  //Used in getting packet length from
header
            unsigned int  Next_Expected_Seq_Number,
Next_Expected_Seq_Number_copy, Previous_Seq_Number;
            unsigned int  Current_Packet_Derived_Seq_Number;
            int  Cells_To_Skip;
            int  Wrap_Around_Counter;  //Used in seq.# validation and skip count
calculation




            //Init buffer
            buf_size = 2*(SOCK_BUFFER_SIZE*2+2+HEADER_SIZE);  //buf_size was
SOCK_BUFFER_SIZE
            buf = (unsigned char *) malloc( buf_size*sizeof(unsigned char) );


            dummy_buf_size = (SOCK_BUFFER_SIZE*2+2+HEADER_SIZE);
//buf_size was SOCK_BUFFER_SIZE




            //Init Previous_Seq_Number
            Previous_Seq_Number = (thr_ID-Connection_Number+1) -
Connection_Number;



            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */
```

```
//Copy starting address of socket buffer
bufStart=buf;

i=0; //Init counter

while (1) {

        len = sizeof(struct sockaddr_in);
        if ((n=recvfrom(servsideproxy_sockfd[thr_ID-
Connection_Number], buf,dummy_buf_size, 0, (struct sockaddr *) &from,&len)) >
0) {



                // ---------- UDP packet handling ----------

                /*
                *Check that first bytes are header - DEBUG
                */


                if ( !((buf[0]==(unsigned char)0x00) &&
(buf[1]!=(unsigned char)0x00) && (buf[1]!=(unsigned char)0xff) &&
(buf[HEADER_SIZE-1]==(unsigned char)0x00)) ) {
                        printf("ERROR(proxyside_recv): Header does not
verify\n\n");
                        exit(1);
                } //if - header incorrect



                /*
                *Verify length etc. -DEBUG
                */
                //Get length
                if ( ((unsigned char)(0x40) & buf[4]) != (unsigned
char)0x00) {
```

```
                        hb_size = ((unsigned char)(0x3f) & buf[4]);
                        lb_size = ((unsigned char)(0x7f)) & buf[5];
                }
                else {
                        hb_size = ((unsigned char)(0x7f)) & buf[4];
                        lb_size = buf[5];
                }

                Header_Derived_Packet_Length = hb_size;
                Header_Derived_Packet_Length =
(Header_Derived_Packet_Length<<8);
                Header_Derived_Packet_Length =
Header_Derived_Packet_Length | lb_size;



                //Verify length
                if (Header_Derived_Packet_Length != n) {
                        printf("ERROR(proxyside_recv): Header derived
length != recv.packet length\n\n");
                        exit(1);
                }



                #ifdef EXTRA_RECV_PACKET_CHECKS


                /*
                *Check header bytes <>0x00 (except start/end bytes) -
DEBUG
                */
                for (m=1; m<HEADER_SIZE-1; m++) {
                        if (buf[m] == 0) {
                                printf("ERROR: %d th byte of header is
0x00\n\n", m+1);
                                exit(1);
                        } //if
                }
```

```
                        #endif
                        /*
                        * Check the sequence number. Must be
Next_Expected_Seq_Number OR a multiple of
                        * Connection_Number + Next_Expected_Seq_Number
OR may have wrapped around.
                        */


                        Next_Expected_Seq_Number = Previous_Seq_Number
+ Connection_Number;
                        //If exceeded seq.# max value wrap it around
                        if (Next_Expected_Seq_Number >
MAX_SEQ_NUMB_VALUE)
                                Next_Expected_Seq_Number =
Next_Expected_Seq_Number-MAX_SEQ_NUMB_VALUE;


                        Current_Packet_Derived_Seq_Number = buf[3];



                        /*
                        * NOTE: Packet seq.# must be ==
Next_Expected_Seq_Number OR be Next_Expected_Seq_Number
                        *               + a multiple of Connection_Number OR
Packet seq.# may have wrapped around skipping cells
                        */
                        if (Current_Packet_Derived_Seq_Number >
Next_Expected_Seq_Number) {
                                if  ( ((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number) % Connection_Number) != 0) {
                                        printf("ERROR: Seq.# INCORRECT(#1).
Expecting %d and got
%d\n\n",Next_Expected_Seq_Number,Current_Packet_Derived_Seq_Number);
                                        exit(1);
                                }
                        } //if - Current_Packet_Derived_Seq_Number >
Next_Expected_Seq_Number
                        else if (Current_Packet_Derived_Seq_Number <
```

```
Next_Expected_Seq_Number) { //Wrap-around case
                                Next_Expected_Seq_Number_copy =
Next_Expected_Seq_Number;

                                Wrap_Around_Counter = 0; //Reset
                                while (Next_Expected_Seq_Number_copy <
MAX_SEQ_NUMB_VALUE) {
                                        Next_Expected_Seq_Number_copy =
Next_Expected_Seq_Number_copy + Connection_Number;
                                        Wrap_Around_Counter++;
                                }

                                //Now that Next_Expected_Seq_Number_copy
> MAX_SEQ_NUMB_VALUE wrap it
                                Next_Expected_Seq_Number_copy =
(Next_Expected_Seq_Number_copy - MAX_SEQ_NUMB_VALUE);

                                if ( ((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number_copy) % Connection_Number) != 0) {
                                        printf("ERROR: Seq.# INCORRECT(#2).
Expecting %d and got
%d\n\n",Next_Expected_Seq_Number,Current_Packet_Derived_Seq_Number);
                                        exit(1);
                                }

                        } //else



                        Previous_Seq_Number =
Current_Packet_Derived_Seq_Number;



                        //If sequence != Next_Expected_Seq_Number then skip
cells & clear/reset buf

                        if (Current_Packet_Derived_Seq_Number !=
Next_Expected_Seq_Number) {
```

```
                                //Calculate # of cells to skip
                                if (Current_Packet_Derived_Seq_Number >
Next_Expected_Seq_Number) {

Cells_To_Skip=(Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number)/Connection_Number;
                                }
                                else {  //Current_Packet_Derived_Seq_Number <
Next_Expected_Seq_Number
                                        Cells_To_Skip = Wrap_Around_Counter +
((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number_copy)/Connection_Number);
                                }



                               for (m=1; m<=Cells_To_Skip; m++) {

                                    //Wait for cell to be read
                                    while ( (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull!=0) || (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Skip!=0) ) {
                                            printf("CHECK(prx_recv[%d]): #1
waiting for proxy_side_recvBuffer[][]!=0\n\n", thr_ID);
                                            thr_yield();
                                    }  //while


                                    // *********
                                    mutex_lock(&Control_Skip_Lock[i][thr_ID-
Connection_Number]);

                                     if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip == 0) {


                                            //Set Skip bit for cell
                                            proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Skip = 1;
```

```
                                        //Since data just written into proxy-
side recv. buffer increment RecvBufSizeCounter counter

mutex_lock(&RecvBufSizeCounter_Lock);
                                         RecvBufSizeCounter++;
                                        mutex_unlock(&RecvBufSizeCounter_Lock);

                                         } //if
                                         else if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip==1) { //Control_Skip is set
                                              proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip = 0;
                                         } //else if - Control_Skip is set
                                         else {
                                              printf("ERROR(proxy_side_recv):
Control_Skip != 0,1\n\n");
                                              exit(1);
                                         } //else - error case


mutex_unlock(&Control_Skip_Lock[i][thr_ID-Connection_Number]);
                                        // *********


                                        //Increment counter
                                        if (i < (R_PROXY_SIDE_BUFFER_SIZE-1))
                                              i++;
                                        else
                                              i = 0;


                                   } //for - skip cells



                              } //if - seq.# != next expected seq.#
```

```
                        // ------------------ UDP packet handling ------------------


                    } //if
                    else{
                            printf("FATAL ERROR(prx_side_recv_Thread): n<=0
while recv. \n\n");
                            close(servsideproxy_sockfd[thr_ID-
Connection_Number]);
                            exit(1);
                    } //else - n<=0 from recvfrom()




                    /*
                    * Copy data from temp. buffer (buf) into
proxy_side_recvBuffer & set size and flag
                    */
                    while ( (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull!=0) || (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Skip!=0) ) {
                            printf("CHECK(prx_recv): #3 waiting for
proxy_side_recvBuffer[][]!=0\n\n");

                            thr_yield();
                    } //while



                    // *********
                    mutex_lock(&Control_Skip_Lock[i][thr_ID-
Connection_Number]);

                    if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip == 0) {
```

```
                        memcpy(proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Data, buf, n);
                        proxy_side_recvBuffer[i][thr_ID-
Connection_Number].DataSize = n;
                        proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull = 1;


                        //Since data just written into proxy-side recv. buffer
increment RecvBufSizeCounter counter
                        mutex_lock(&RecvBufSizeCounter_Lock);
                        RecvBufSizeCounter++;
                        mutex_unlock(&RecvBufSizeCounter_Lock);

                } //if
                else if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip==1) { //Control_Skip is set
                        proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip = 0;
                } //else if - Control_Skip is set
                else {
                        printf("ERROR(proxy_side_recv): Control_Skip !=
0,1\n\n");
                        exit(1);
                } //else - error case

                mutex_unlock(&Control_Skip_Lock[i][thr_ID-
Connection_Number]);
                // *********


                //Increment counter
                if (i < (R_PROXY_SIDE_BUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


        } //while - 1
```

```
} //proxyside_recv_ThreadFunc
/*
**********************************************************************************
********************* */


void *clientside_send_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i; //counter var
        unsigned char   buf[SOCK_BUFFER_SIZE];



        i=-1;
        while (1) {

            /*
            * Get data from ssproxy_to_client_ProxyBuffer;
            */
            //Increment counter
            if (i < (PS_TO_C_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (ssproxy_to_client_ProxyBuffer[i].EntryFull != 1) {
                    thr_yield();
            }
            memcpy(buf, ssproxy_to_client_ProxyBuffer[i].Data,
ssproxy_to_client_ProxyBuffer[i].DataSize);
            n = ssproxy_to_client_ProxyBuffer[i].DataSize;
            ssproxy_to_client_ProxyBuffer[i].EntryFull = 0;
```

```
            /*
            * Send data to client
            */
            if (n>0) {
                if ( send(client_sock, buf, n, 0) < 0) {
                                printf("ERROR(p_cs_proxy server_send_thread): While
sending data to client <0 \n\n");
                                exit(1);
                } //if
            } //if


  } //while


} //clientside_send_ThreadFunc


/*
**********************************************************************************
********************** */


void *clientside_recv_ThreadFunc(void *thr_ID_PTR) {


            /*
            * Get thread ID
            */
            int  thr_ID = *((int *) thr_ID_PTR);

            int   n;
            int   i,j; //counter vars
            unsigned char   buf[CL_RECV_SOCK_BUFFER_SIZE];
            time_t  client_sock_close_time;  //timestamp, used to implement time
delay before child terminated
```

```
        i=-1;
        while ((n = recv(client_sock, buf, sizeof(buf), 0)) > 0) {

            /*
            * Copy received data to client_to_ssproxy_ProxyBuffer;
            */

            //Increment counter
            if (i < (C_TO_PS_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;


            while (client_to_ssproxy_ProxyBuffer[i].EntryFull != 0) {
                    thr_yield();
            }
            memcpy(client_to_ssproxy_ProxyBuffer[i].Data, buf, n);
            client_to_ssproxy_ProxyBuffer[i].DataSize = n;
            client_to_ssproxy_ProxyBuffer[i].EntryFull = 1;



        } //while
        if (n<=0) {
            printf("FATAL ERROR(er_cs/clientside_recv): n<=0 while recv. from
client \n");
            printf("(Non-fatal if end of client-server communication)\n\n");
            /*
            * Client has closed so won't tranmsit anything else. Implement a wait
using thr_yield() ro
            * allow for any data remaining to be sent (& received by) to client
            */

            client_sock_close_time = time(NULL);

            close(client_sock);

            while ( ((time_t) difftime(time(NULL), client_sock_close_time)) <=
EXIT_TIME_DELAY)
```

```
                thr_yield();
            exit(1);

        }

} //clientside_recv_ThreadFunc


/*
**********************************************************************
********************* */


void *proxy_to_client_Control_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   i,j,k; //counter vars
        int  RNR_flag = 0; //True if an RNR message has been sent to serv-side
proxy.
        unsigned char  cong_msg_buf[1024];  //byte buffer used to send RR &
RNR messages to serv-side proxy
        int cong_Control_Check_Counter;  //Used to control when cong.cntrl
checks occur
        unsigned int  Next_Expected_Seq_Number=0;  //Between 1 and
MAX_SEQ_NUMB_VALUE (not 0 to MAX_SEQ_NUMB_VALUE)!!!
        unsigned int  Block_First_Seq_Num;  //Seq.# of first packet of a block
        int  Encoded_Packets_Number;  //Number of encoded packets for
current block(i.e. what is known as "n" in header)
        int  Encoded_Packets_Read;  //Counter var. Used in reading packets
from every block
        int  Source_Packets_Number;  //Number of source packets for current
block(i.e. what is known as "k" in header)
        int  Source_Packets_Read;  //Number of source/data packets already
read from current block
        int  Curr_Block_Packet_Size;  //Size if packets(not including header) of
```

```
a given block
            int  RR_counter=0;  //Bookeeping var that just keep track of RR
messages sent out
            int  RNR_counter=0;  //Bookeeping var that just keep track of RNR
messages sent out
            gf *dec_src[DEC_ARRAY_SIZE];  //Used as argument in decoding
function
            gf *dec_dst[DEC_ARRAY_SIZE];  //Used as argument in decoding
function
            int dec_index[DEC_ARRAY_SIZE];  //Used as argument in decoding
function. Indexes of packets handed to dec. function
            int  ic;  //Counter var
            int  EPR_Start=1;
            int  Block_First_Seq_Num_Set = 0;  //Flag
            int  Number_of_Packets_Skipped;
            int  CONG_CNTRL_CHECK_PERIOD;  //Period of congestion control check.
CAUTION: If this value is too large then
                                                //might fall into deadlock (i.e.
not sending a RR at all after a RNR)!!!
            time_t  start_time, curr_time;
            int  First_Time=0;
            int  NO_RECV_TIMEOUT = 10;
            int  last_index_value;  //Used in index calculation (for decoding)

            int Cumul_Source_Packets_Read=0;  //Test var
            int TestThresh = 760;

            /*
            * Set congestion control check period
            */
            //CONG_CNTRL_CHECK_PERIOD = Connection_Number;
            CONG_CNTRL_CHECK_PERIOD = 0;

            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
```

```
*/
/*
* Allocate memory to dec_src and dec_dst
*/
for (ic=0; ic<DEC_ARRAY_SIZE; ic++) {
    dec_src[ic]= (gf *) malloc(DEC_ARR_BUF_SIZE*sizeof(gf));
    dec_dst[ic]= (gf *) malloc(DEC_ARR_BUF_SIZE*sizeof(gf));
} //for



k = 0;  //Init counter var
i = 0;      //Init counter var
j = 0;      //Init counter var

cong_Control_Check_Counter = CONG_CNTRL_CHECK_PERIOD;  //Init



while (1) {  //Handle current block of packets


    if (Cumul_Source_Packets_Read > TestThresh)
            printf("CHECK**: RIGHT BEFORE T/O WHILE
#1(Pack#:%d)\n",Cumul_Source_Packets_Read);

    First_Time = 0;
    while ( (proxy_side_recvBuffer[i][j].EntryFull != 1) &&
(proxy_side_recvBuffer[i][j].Skip != 1) ) {



    // ********
            #ifdef RECV_CONTROL_DEBUG
            if (Cumul_Source_Packets_Read > TestThresh)
                    if (First_Time==0)
                            printf("CHECK**: RIGHT INSIDE T/O WHILE
#1(Pack#:%d)\n",Cumul_Source_Packets_Read);
            #endif
```

```
                    if (First_Time==0) {
                            First_Time=1;
                            start_time = time(NULL);
                    }

                    curr_time = time(NULL);

                    if ( (float)difftime(curr_time, start_time) >
NO_RECV_TIMEOUT ) {
                            printf("EXIT(recv_Control): %d sec w/ no new data
received\n\n",NO_RECV_TIMEOUT);
                            exit(1);
                    }




                    thr_yield();
            } //while

            First_Time = 0;

            #ifdef RECV_CONTROL_DEBUG
            if (Cumul_Source_Packets_Read > TestThresh)
                    printf("CHECK**: RIGHT AFTER T/O WHILE #1
(Pack#:%d)\n",Cumul_Source_Packets_Read);
            #endif



            if (proxy_side_recvBuffer[i][j].Skip == 1) {


                    //Caclulate next expected sequence number. Inited to 0. Starts
from 1 (1->MAX_SEQ_NUMB_VALUE)
                    if (Next_Expected_Seq_Number < MAX_SEQ_NUMB_VALUE)
                            Next_Expected_Seq_Number++;
                    else
                            Next_Expected_Seq_Number = 1;
```

```
                //Reset Skip bit
                proxy_side_recvBuffer[i][j].Skip = 0;



                /*
                * Get sequence # of first packet of block (if not done already)
                */
                if (Block_First_Seq_Num_Set != 1) {
                        Block_First_Seq_Num = Next_Expected_Seq_Number;
                        Block_First_Seq_Num_Set = 1;
                }  //if



                //Skip current cell. Have for loop below start after skipped
cells
                EPR_Start = EPR_Start++;


                // i,j incrementation
                if (j<(Connection_Number-1))
                        j++;
                else {
                        j = 0;

                        if (i<(R_PROXY_SIDE_BUFFER_SIZE-1))
                                i++;
                        else
                                i = 0;
                }
                // i,j incrementation


                //Proceed to next cell
                continue;


        }  //if - must Skip current cell
```

```
		/*
		* Now that we got out of skipping first packet(s) check if we skipped
entire block
		*/
		Number_of_Packets_Skipped = EPR_Start - 1;
		if (Number_of_Packets_Skipped >=
proxy_side_recvBuffer[i][j].Data[6]) {
			printf("ERROR/STATUS: Entire block skipped.
Terminating\n\n");
			exit(1);
		} //if - check if entire block skipped



		/*
		* Get sequence # of first packet of block (if not already done)
		*/
		if (Block_First_Seq_Num_Set != 1) {
			Block_First_Seq_Num = proxy_side_recvBuffer[i][j].Data[3];
		} //if
		else
			Block_First_Seq_Num_Set = 0;



		/*
		* NOTE: Header format: (0x00, k, n, seq.#, length(2-bytes), packet#
in block,0x00)
		*/


		/*
		* Get number of packets for current block i.e. "n"
		*/
		Encoded_Packets_Number = proxy_side_recvBuffer[i][j].Data[2];


		/*
		* Get number of source/data packets for this block i.e. "k"
```

```
		*/
		Source_Packets_Number = proxy_side_recvBuffer[i][j].Data[1];

		/*
		* Get packet size (NOT including header) for current block
		*/
		Curr_Block_Packet_Size = proxy_side_recvBuffer[i][j].DataSize -
HEADER_SIZE;



		/*
		* Reset vars
		*/
		Source_Packets_Read = 0;



		/*
		* Go through Encoded_Packets_Number entries of proxy-side receive
buffer i.e. one block of "n" packets
		*/
		for (Encoded_Packets_Read=EPR_Start;
Encoded_Packets_Read<=Encoded_Packets_Number; Encoded_Packets_Read++)
{


			//Caclulate next expected sequence number. Inited to 0. Starts
from 1 (1->MAX_SEQ_NUMB_VALUE)
			if (Next_Expected_Seq_Number < MAX_SEQ_NUMB_VALUE)
				Next_Expected_Seq_Number++;
			else
				Next_Expected_Seq_Number = 1;


		//while and if used to be here before Control_Skip


		if (Source_Packets_Read < Source_Packets_Number) {  //i.e. not
enough packets read yet for decoding
```

```
                    // *********

                    #ifdef RECV_CONTROL_DEBUG
                    if (Cumul_Source_Packets_Read > TestThresh)
                                printf("CHECK**: RIGHT BEFORE T/O WHILE
#2(Pack#:%d w/
k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                          #endif


                    //Wait until entry full or recv. thread "says" to skip current cell
                    First_Time = 0;
                    while ( (proxy_side_recvBuffer[i][j].EntryFull != 1) &&
(proxy_side_recvBuffer[i][j].Skip != 1) ) {


                          // ******
                          #ifdef RECV_CONTROL_DEBUG
                          if (Cumul_Source_Packets_Read > TestThresh)
                                if (First_Time==0)
                                            printf("CHECK**: INSIDE T/O WHILE
#2(Pack#:%d w/
k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                          #endif

                          if (First_Time==0) {
                                      First_Time=1;
                                      start_time = time(NULL);
                                }

                                curr_time = time(NULL);

                                if ( ((float)difftime(curr_time, start_time)) >
NO_RECV_TIMEOUT ) {
                                      printf("EXIT(recv_Control): %d sec w/ no
new data received\n\n",NO_RECV_TIMEOUT);
                                      exit(1);
                                }
```

```
                    thr_yield();

                } //while
                // ********BELOW
                First_Time = 0;
                #ifdef RECV_CONTROL_DEBUG
                if (Cumul_Source_Packets_Read > TestThresh)
                    printf("CHECK**: RIGHT AFTER T/O WHILE #2
(Pack#:%d w/
k=%d)\n\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                #endif


                if (proxy_side_recvBuffer[i][j].EntryFull == 1) {

                    //Check if seq.number valid (stored in 4th byte of
header)
                    if (proxy_side_recvBuffer[i][j].Data[3] !=
Next_Expected_Seq_Number) {
                        printf("ERROR(proxy_to_client_Control):
Sequence # is WRONG. %d expected but got %d
\n\n",Next_Expected_Seq_Number,proxy_side_recvBuffer[i][j].Data[3]);
                        exit(1);
                    } //if - seq.number is NOT next expected seq.number



                    //Error check: check if k and n of current packet is
thart of current block
                    if (proxy_side_recvBuffer[i][j].Data[1] !=
Source_Packets_Number) {

printf("ERROR(er_cs/proxy_to_client_Control): Source_Packets_Number(k) is not
that expected \n\n");
                        exit(1);
                    } //if
                    if (proxy_side_recvBuffer[i][j].Data[2] !=
Encoded_Packets_Number) {

printf("ERROR(er_cs/proxy_to_client_Control): Encoded_Packets_Number(n) is
```

```
not that expected \n\n");
                                        exit(1);
                                } //if


                                        //Error check: compare size with
Curr_Block_Packet_Size
                                if ( (proxy_side_recvBuffer[i][j].DataSize-HEADER_SIZE)
!= Curr_Block_Packet_Size) {
                                        printf("ERROR(er_cs/proxy_to_client_Control):
Size of read packet<>Curr_Block_Packet_Size\n\n");
                                        exit(1);
                                } //if - error check for size


                        } //if - if data in cell perfrom some checks


                        // *********



                        //Read cell data if cell full, else if Skip specified just reset Skip
flag
                        if (proxy_side_recvBuffer[i][j].EntryFull == 1) {

                                memcpy(dec_src[Source_Packets_Read],
proxy_side_recvBuffer[i][j].Data+HEADER_SIZE,
proxy_side_recvBuffer[i][j].DataSize-HEADER_SIZE);



                                //Calculate dec_index[] for fec decoding
                                if (proxy_side_recvBuffer[i][j].Data[3] >=
Block_First_Seq_Num) {
                                        dec_index[Source_Packets_Read] =
proxy_side_recvBuffer[i][j].Data[3] - Block_First_Seq_Num;
                                        last_index_value =
dec_index[Source_Packets_Read];
                                }
                                else if (proxy_side_recvBuffer[i][j].Data[3] <
```

```
Block_First_Seq_Num)
                                dec_index[Source_Packets_Read] =
last_index_value + proxy_side_recvBuffer[i][j].Data[3];


                        proxy_side_recvBuffer[i][j].EntryFull = 0;


                                //Increment counter of # of packets read from
current block
                        Source_Packets_Read++;



                } //if
                else if (proxy_side_recvBuffer[i][j].Skip == 1) {
                        proxy_side_recvBuffer[i][j].Skip = 0;
                } //else if
                else {
                        printf("ERROR(er_cs/proxy_to_client_Control): Neither
entry full nor Skip set\n\n");
                        exit(1);
                } //else


        } //if - # of packets read until now < Source_Packets_Number i.e.
not enough packets yet
        else { //Source_Packets_Read packets have been read i.e. enough
data packets obtained for decoding


                #ifdef RECV_CONTROL_DEBUG

                if (Cumul_Source_Packets_Read > TestThresh)
                                printf("CHECK**: IN 'ENOUGH PACKETS' START #3
(Pack#:%d w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                #endif


                // *********
                        mutex_lock(&Control_Skip_Lock[i][j]);
```

```
                        //Error check
                        if ( (proxy_side_recvBuffer[i][j].EntryFull == 1) &&
(proxy_side_recvBuffer[i][j].Skip == 1) ) {
                                printf("ERROR(recv_Control): EntryFull && Skip
==1\n\n");
                                exit(1);
                        }

                        if (proxy_side_recvBuffer[i][j].EntryFull == 1) {
                        proxy_side_recvBuffer[i][j].EntryFull = 0;
                } //if
                else if (proxy_side_recvBuffer[i][j].Skip == 1) {
                        proxy_side_recvBuffer[i][j].Skip = 0;
                } //else if
                else {
                        proxy_side_recvBuffer[i][j].Control_Skip = 1;
                        //Since data just written into proxy-side recv. buffer
increment RecvBufSizeCounter counter

                        mutex_lock(&RecvBufSizeCounter_Lock);
                          RecvBufSizeCounter++;  //Increment to offset default
decrement below
                        mutex_unlock(&RecvBufSizeCounter_Lock);
                } //else


                        mutex_unlock(&Control_Skip_Lock[i][j]);
                        // *********


                #ifdef RECV_CONTROL_DEBUG

                if (Cumul_Source_Packets_Read > TestThresh)
                                printf("CHECK**: IN 'ENOUGH PACKETS' END #3
(Pack#:%d w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                #endif



        } //else - Source_Packets_Read packets have been read i.e. enough
```

data packets obtained for decoding

```
#ifdef RECV_CONTROL_DEBUG
Cumul_Source_Packets_Read++;
#endif
        // i,j incrementation
        if (j<(Connection_Number-1))
                j++;
        else {
                j = 0;

                if (i<(R_PROXY_SIDE_BUFFER_SIZE-1))
                        i++;
                else
                        i = 0;
        }
        // i,j incrementation



// ------------------CONGESTION CONTROL-------------------------

/*
* Congestion control check perfromed every
CONG_CNTRL_CHECK_PERIOD+1 rounds. Is VAR(not constant)!!!
*/

        //Regardless of if will perform congestion control check,
packet reception/reading must be
        // logged i.e. RecvBufSizeCounter decremented
        mutex_lock(&RecvBufSizeCounter_Lock);
          RecvBufSizeCounter--;
          RecvBufSizeCounter_copy = RecvBufSizeCounter;
        mutex_unlock(&RecvBufSizeCounter_Lock);


      if (cong_Control_Check_Counter == 0) {

        //Reset cong_Control_Check_Counter
        cong_Control_Check_Counter = CONG_CNTRL_CHECK_PERIOD;
```

```
                    if (RecvBufSizeCounter_copy<0) {
                            printf("ERROR: RecvBufSizeCounter < 0\n\n");
                            exit(1);
                    }
                            if (RNR_flag == 0) {  //Send RNR message IF congested

                            if (
(RecvBufSizeCounter_copy/PROXY_RECV_BUF_CAPACITY) >=
RNR_PROXY_RECV_BUF_SIZE_THRESH ) {
                                    RNR_flag = 1;  //Set flag

                                    //Send ReceiveNotReady-RNR message("RNR"
NULL terminated string) to server-side proxy
                                    strcpy(cong_msg_buf, "RNR");
                                    if ( send(tcp_sock, cong_msg_buf,
strlen(cong_msg_buf)+1, 0) <0) {
                                            printf("ERROR: send()<0 for RNR  \n\n");
                                            close(tcp_sock);
                                            exit(1);
                                    }

                                    RNR_counter++;  //Bookeeping var
                                    printf("STATUS: %d RNR messages sent out until
now\n\n",RNR_counter);

                            }  //if
                            }  //if - RNR_flag == 0
                            else if (RNR_flag == 1)  {  //RNR_flag == 1 & thus send
RR message if not congested anymore
                            if (
(RecvBufSizeCounter_copy/PROXY_RECV_BUF_CAPACITY) <
RR_PROXY_RECV_BUF_SIZE_THRESH ) {

                                    RNR_flag = 0;  //Set flag

                                    //Send ReceiveReady-RR message("RR" NULL
terminated string) to server-side proxy
                                    strcpy(cong_msg_buf, "RR");
                                    if ( send(tcp_sock, cong_msg_buf,
strlen(cong_msg_buf)+1, 0) <0) {
```

```
                                        printf("ERROR: send()<0 for RR  \n\n");
                                        close(tcp_sock);
                                        exit(1);
                                }

                                RR_counter++;  //Bookeeping var
                                printf("STATUS: %d RR messages sent out until
now\n\n",RR_counter);

                        }  //if

                        } //elseif - RNR_flag == 1
                else  //error case
                        printf("ERROR: RNR_flag neither 0 nor 1 \n\n");

        } //if - cong_Control_Check_Counter == 0
        else { //else - cong_Control_Check_Counter > 0
                cong_Control_Check_Counter--;
        } //else - cong_Control_Check_Counter > 0


        // ------------------CONGESTION CONTROL-------------------------


        #ifdef RECV_CONTROL_DEBUG

        if (Cumul_Source_Packets_Read > TestThresh)
                        printf("CHECK**: AT END OF 'BLOCK' FOR LOOP #4
(Pack#:%d w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
        #endif


    } //for - Go through Encoded_Packets_Number entries of proxy-side
receive buffer


        #ifdef RECV_CONTROL_DEBUG

        if (Cumul_Source_Packets_Read > TestThresh)
                        printf("CHECK**: AFTER 'BLOCK' FOR LOOP #5
(Pack#:%d w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
```

```
            #endif



            /*
            * Having read entire current block do error check if enough packets read
needed for decoding
            */
            if (Source_Packets_Read != Source_Packets_Number) {
                printf("ERROR(er_cs/proxy_to_client_Control): Not enough packets
for decoding(or too many)\n\n");
                exit(1);
            } //if


            /*
            * Having read current block and having sufficient # of packets for
decoding, proceed w/ decoding
            * and writing to ssproxy_to_client_ProxyBuffer
            */

                #ifdef RECV_CONTROL_DEBUG

                if (Cumul_Source_Packets_Read > TestThresh) {
                                printf("CHECK**: RIGHT BEFORE DECODING fec #6
(Pack#:%d w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
                        printf("dec_index's: %d %d - Source_Packets_Number:%d -
Curr_Block_Packet_Size:%d\n",dec_index[0],dec_index[1],Source_Packets_Numb
er,Curr_Block_Packet_Size);

                }
                #endif


           //Decode - decode_fec(gf *src[], gf *dst[], int index[], int k, int sz);
           decode_fec(dec_src, dec_dst, dec_index, Source_Packets_Number,
Curr_Block_Packet_Size);


                #ifdef RECV_CONTROL_DEBUG
```

```
            if (Cumul_Source_Packets_Read > TestThresh)
                        printf("CHECK**: AFTER DECODING fec #7 (Pack#:%d
w/ k=%d)\n",Cumul_Source_Packets_Read,Source_Packets_Number);
            #endif
          //Write decoded packets to ssproxy_to_client_ProxyBuffer
          for (ic=0; ic<Source_Packets_Number; ic++) {

            while (ssproxy_to_client_ProxyBuffer[k].EntryFull != 0) {
                printf("CHECK(recvControl): waiting for
ssproxy_to_client_ProxyBuffer[]!=0\n\n");
                thr_yield();
            }

            memcpy(ssproxy_to_client_ProxyBuffer[k].Data, dec_dst[ic],
Curr_Block_Packet_Size);
            ssproxy_to_client_ProxyBuffer[k].DataSize =
Curr_Block_Packet_Size;
            ssproxy_to_client_ProxyBuffer[k].EntryFull = 1;


            if (k < (PS_TO_C_PROXYBUFFER_SIZE-1))
                        k++;
                else
                        k = 0;

          } //for - write decoded packets into ssproxy_to_client_ProxyBuffer



          //Reset
            EPR_Start = 1;


                #ifdef RECV_CONTROL_DEBUG

            if (Cumul_Source_Packets_Read > TestThresh)
                        printf("CHECK**: AT END 'WHILE' LOOP #8- (Pack#:%d
w/ k=%d)\n\n",Cumul_Source_Packets_Read,Source_Packets_Number);
            #endif
```

```
  } //while (1) - Handle current block of packets


} //proxy_to_client_Control_ThreadFunc


/*
**********************************************************************************
********************** */



void *client_to_proxy_Control_ThreadFunc(void *thr_ID_PTR) {


            /*
            * Get thread ID
            */
            int  thr_ID = *((int *) thr_ID_PTR);

            int   i,j,k; //counter vars
            int  K_actual=0;  //# of entries actually read(& to be encoded) from
client_to_ssproxy_ProxyBuffer
            int  m, thryield_counter, p;  //Counter vars
            unsigned char *cumul_buf;  //Cumulative buffer into which packets read
from client_to_ssproxy_ProxyBuffer are copied into
            int  cumul_buf_Size=0;  //Current size(i.e. # entries full) of cumul_buf
            int  cumul_buf_Size_temp=0;  //A temp var
            unsigned char *cumul_buf_Start;  //Used to store original starting
address of cumul_buf
            int  packet_Size=0;  //Size of packets to be encoded and writeen to
send-out buffer
            int  packet_Number=0;  //Number of packets to be encoded. Equal to
K_actual in general
            int  encode_index;  //COunter used as index when calling encoding
function
            gf *encoded_Packet;  //Used in encoding function call. Contains encoded
packet
            gf *source_data_Packets[K_DESIRED];  //Array of buffers containing
```

```
source(data) packets to be used in encoding
        int  Num_Encoded_Packets=0;  //Total number of packets to be
produced from encoding
        FILE *enc_debug_fd;  //File descriptor of a file used for debug output
ENC_DEBUG_FILENAME
        unsigned char
header_added_Packet[SOCK_BUFFER_SIZE+HEADER_SIZE];  //Packet copied into
send buffer. Includes header plus the encoded packet
        unsigned int  Sequence_Number=1;  //Sequence number used in header.
Inited to 1!!!
        unsigned char *Start_Adrr_of_Remaining_Bytes;

        struct timeval  *start_time;
        struct timeval  *curr_time;
        long  start_time_in_msec, curr_time_in_msec;

        struct sockaddr_in
ssp_name[MAX_CONN_NUMB_PER_LOGICAL_CONN];  //Server-side proxy
address
        int  send_socket_index = -1;  //Init




  //
**************************************************************************
***************

        /*
        * Allocate memory to time vars
        */
        start_time = (struct timeval *) malloc(sizeof(struct timeval));
        curr_time = (struct timeval *) malloc(sizeof(struct timeval));



        /*
        * Setup address on server-side proxy with which this thread
```

```
communicates
            */
            for (i=0; i<Connection_Number; i++) {
                memset(&ssp_name[i], 0, sizeof(struct sockaddr_in));
                ssp_name[i].sin_family = AF_INET;
                ssp_name[i].sin_port = htons(SSP_UDP_BASE_PORTNUMBER + i);
                memcpy(&ssp_name[i].sin_addr, hp->h_addr_list[0], hp->h_length);
//hp has been set from above in child process
            } //for



            /*
            * Open encoding debug file descriptor
            */
            #ifdef ENC_DEBUG_ON
              enc_debug_fd = fopen(ENC_DEBUG_FILENAME, "w");
            #endif




            /*
            * Allocate memory to encoded_Packet and source_data_Packets[]
            */
            encoded_Packet = (gf *) malloc(ENC_PACKET_BUF_SIZE*sizeof(gf));

            for (m=0; m<K_DESIRED; m++) {
                source_data_Packets[m]= (gf *)
malloc(ENC_SOURCE_BUF_BUFFER_SIZE*sizeof(gf));
            } //for



            /*
            * Allocate memory to cumul_buf
            */
            cumul_buf = (unsigned char *)
malloc((K_DESIRED+1)*SOCK_BUFFER_SIZE*sizeof(unsigned char));
```

```
            /*
            * Store starting address of cumul_buf
            */
            cumul_buf_Start = cumul_buf;


            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */


 k = 0;  //Init client_to_ssproxy_ProxyBuffer counter
 i = 0;  //Init
 j = 0;  //Init




 while (1) {

            K_actual = 0;  //Reset



            /*
            * Reset cumul_buf_Size and cumul_buf address
            */
            cumul_buf_Size = 0;
            cumul_buf = cumul_buf_Start;



            /*
            * Try to get/read K_DESIRED entries from
```

```
client_to_ssproxy_ProxyBuffer.
            * K_actual is number of entries that were actually read.
            */

            // ---------------------------------------------------------------------------

            for (m=1; m<=K_DESIRED; m++) {
               if (m==1) {

                        //Wait until entry/packet written into next cell to be read
                        while (client_to_ssproxy_ProxyBuffer[k].EntryFull != 1) {
                                    thr_yield();
                        } //while

                        K_actual = 1;

               } //if - m==1
               else { //else - m>1

                        for (thryield_counter=1; thryield_counter<=THR_YIELD_MAX;
thryield_counter++) {

                                 if (client_to_ssproxy_ProxyBuffer[k].EntryFull != 1) {
                                        #ifdef ENC_DEBUG_ON
                                          fprintf(enc_debug_fd, "Thread slept to get %d
th packet\n", m);
                                        #endif
                                        thr_yield();
                                 } //if
                                 else
                                        break;

                        } //for

                        if (client_to_ssproxy_ProxyBuffer[k].EntryFull != 1)
                                 break;


                        //Increment # of packets/entries read from
client_to_ssproxy_ProxyBuffer
                        K_actual = K_actual +1;
```

```
            } //else - m>1


        //Write packet into cumulative buffer & increment cumulative buffer
size(# bytes in c.buffer) etc.
            memcpy(cumul_buf, client_to_ssproxy_ProxyBuffer[k].Data,
client_to_ssproxy_ProxyBuffer[k].DataSize);
            cumul_buf=cumul_buf + client_to_ssproxy_ProxyBuffer[k].DataSize;
//Move cumul_buf start so as to copy in next packet next round (if done)
            cumul_buf_Size = cumul_buf_Size +
client_to_ssproxy_ProxyBuffer[k].DataSize;
            client_to_ssproxy_ProxyBuffer[k].EntryFull = 0;



            //Increment client_to_ssproxy_ProxyBuffer index/counter k (at END
of for loop)
            if (k < (C_TO_PS_PROXYBUFFER_SIZE-1))
                  k++;
            else
                  k = 0;

        } //for

        // ---------------------------------------------------------------------


        /*
        * Write K_actual/K_DESIRED to enc. debug file
        */
        #ifdef ENC_DEBUG_ON
          fprintf(enc_debug_fd, "K_actual/K_DESIRED = %d/%d \n", K_actual,
K_DESIRED);
        #endif


        /*
```

```
            * Make an error check for K_actual
            */
            if (K_actual > K_DESIRED) {
                printf("ERROR(client_to_proxy_Control_Thread): K_actual >
K_DESIRED\n\n");
                exit(1);
            }
            /*
            * Having read K_actual entries from client_to_ssproxy_ProxyBuffer,
now encode these entries/packets
            * & write them to proxy_side_sendBuffer
            */
            //
==========================================================================
==============

            //Reset cumul_buf to its starting address
            cumul_buf = cumul_buf_Start;


            //Copy cumul_buf_Size into a temp var to use
            cumul_buf_Size_temp = cumul_buf_Size;




            //Set size of packets to be encoded and written to send-out buffer
(K_actual is # of packets)
            if (cumul_buf_Size_temp > TEMP_CUMUL_BUF_SIZE_TRESH ) {


                while ( (cumul_buf_Size_temp % K_actual) != 0 )
                        cumul_buf_Size_temp = cumul_buf_Size_temp - 1;


                #ifdef ENC_DEBUG_ON
                        fprintf(enc_debug_fd,"cumul_buf_Size=%d and
cumul_buf_Size_temp=%d\n",cumul_buf_Size,cumul_buf_Size_temp);
                #endif
```

```
                //Error check. cumul_buf_Size_temp must have reduced in size no
more than (K_actual-1)
                if ( (cumul_buf_Size - cumul_buf_Size_temp) > (K_actual-1) ) {
                        printf("ERROR(client_to_proxy_Control_Thread): More than
(K_actual-1) bytes 'removed' from cumul_buf\n\n");
                        exit(1);
                } //if


                packet_Size = (cumul_buf_Size_temp/K_actual);
                packet_Number = K_actual;

            } //if
            else { //cumul_buf_Size_temp<=TEMP_CUMUL_BUF_SIZE_TRESH so
encode/send out as one packet(for efficiency)


                packet_Size = cumul_buf_Size_temp;
                packet_Number = 1;
                #ifdef ENC_DEBUG_ON
                  if (K_actual>1)
                        fprintf(enc_debug_fd, "Amalgamation occurred(%d bytes)
\n", cumul_buf_Size_temp);
                #endif
            } //else - cumul_buf_Size_temp<=TEMP_CUMUL_BUF_SIZE_TRESH so
encode/send out as one packet(for efficiency)


            //Print blank line in enc. debub file
            #ifdef ENC_DEBUG_ON
                fprintf(enc_debug_fd,"\n");
            #endif


            //Error check
            if (packet_Size > CL_RECV_SOCK_BUFFER_SIZE) {
                printf("ERROR(client_to_proxy_Control): packet_Size >
CL_RECV_SOCK_BUFFER_SIZE \n\n");
                exit(1);
```

```
        } //if


        //Copy source(data) packets into source_data_Packets[] array of
buffers
        for (p=0; p<packet_Number; p++) {
            source_data_Packets[p] = cumul_buf+(p*packet_Size);
        }


        //Record addres of byte after end of last packet read (needed below
for sending out remaining bytes)
        Start_Adrr_of_Remaining_Bytes = cumul_buf+(p*packet_Size);


        //Set # of packets to be produced from encoding
        Num_Encoded_Packets = N_FACTOR*packet_Number;



        /*
        * Encode the packet_Number packets of size packet_Size and copy
them to proxy_side_sendBuffer[][]
        */
        for (encode_index=0 ; encode_index<Num_Encoded_Packets;
encode_index++) {


            //Get one encoded packet (out of Num_Encoded_Packets)
            build_fec(source_data_Packets, packet_Number, packet_Size,
header_added_Packet+HEADER_SIZE, encode_index);


            /*
            * Copy encoded packet to proxy_side_sendBuffer
            */
            //Add header (k, n, Seq.#, length[2-bytes], packet# in block)
            header_added_Packet[0] = (unsigned char) 0x00;
            header_added_Packet[1] = (unsigned char)packet_Number;  //Set k
i.e. # of data/source packets
```

```
                header_added_Packet[2] = (unsigned char)Num_Encoded_Packets;
//Set n i.e. # of enc.packets
                header_added_Packet[3] = (unsigned char)Sequence_Number;
//Set sequence #. Keep it here!!!
                header_added_Packet[5] = (unsigned
char)(packet_Size+HEADER_SIZE);  //Set length (2nd byte [low])
                header_added_Packet[4] = (unsigned
char)((packet_Size+HEADER_SIZE)>>8);  //Set length (1st byte [high])
                header_added_Packet[4] = (unsigned char)( ((unsigned char)(0x80))
| header_added_Packet[4]);  //Code so that <>0

                if (header_added_Packet[5] == (unsigned char)0x00) {
                        header_added_Packet[5] = (unsigned char)( ((unsigned
char)(0x80)) | header_added_Packet[5] );
                        header_added_Packet[4] = (unsigned char)( ((unsigned
char)(0x40)) | header_added_Packet[4] );
                } //if - code length fields so that <>0


                header_added_Packet[6] = encode_index+1;
                header_added_Packet[7] = (unsigned char) 0x00;


                //Increment sequence number after using it
                if (Sequence_Number < MAX_SEQ_NUMB_VALUE)
                        Sequence_Number++;
                else
                        Sequence_Number=1;



        // ++++++++++++++++++++++++++++++++++++++++++++++

        /*
            * Send data to server-side proxy
        */


            //Check congestion control flag HoldBack before sending data and
wait if set
            mutex_lock(&HoldBack_Lock);
```

```
            while (HoldBack != 0) {
                    mutex_unlock(&HoldBack_Lock);
                    thr_yield();
                    mutex_lock(&HoldBack_Lock);
            }
            mutex_unlock(&HoldBack_Lock);


            #ifdef USE_DELAY

            // ----------------

             if ( gettimeofday(start_time, NULL) <0 ) {
                    printf("ERROR(send_Control): gettimeofday error\n\n");
                    exit(1);
             }

             start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;



            while (1) {
                    if ( gettimeofday(curr_time, NULL) <0 ) {
                            printf("ERROR(send_Control): gettimeofday error\n\n");
                            exit(1);
                    }

                    curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;


                    if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                            break;


                    //Since haven't reached timeout value yet have thread
sleep/yield
                    thr_yield();
```

```
                } //while


                // ----------------

                #endif


                //---------
                //Increment send socket identifier/index (to change sending socket
used)
                if ( send_socket_index < (Connection_Number-1) )
                        send_socket_index++;
                else
                        send_socket_index = 0;
                //---------


                //TCP-if ( send(servsideproxy_sockfd[send_socket_index], buf, n , 0)
< 0) {
                        if (sendto(servsideproxy_sockfd[send_socket_index],
header_added_Packet, packet_Size+HEADER_SIZE, 0, (struct sockaddr *)
&ssp_name[send_socket_index],sizeof(struct sockaddr_in)) < 0) {
                                printf("ERROR(send/Control): While sending data to
serv-side proxy <0 \n\n");
                                exit(1);
                        } //if



        // ++++++++++++++++++++++++++++++++++++++++++++++


                //Update j & i values
                if (j < (Connection_Number-1))
                        j++;
                else { //j wraps-around so update i also
                        j = 0;

                        if (i < (S_PROXY_SIDE_BUFFER_SIZE-1))
```

```
                    i++;
                else
                    i=0;
            }

        } //for - encode packets & copy them to proxy_side_sendBuffer[][]
        /*
        * If bytes left over send them out - NO PASTING else can fall into
deadlock !!!
        */
        if (cumul_buf_Size_temp < cumul_buf_Size ) {

            // ********


            packet_Number = 1;
            packet_Size = cumul_buf_Size - cumul_buf_Size_temp;

            //Set # of packets to be produced from encoding
            Num_Encoded_Packets = N_FACTOR*packet_Number;


            //Copy source(data) packets into source_data_Packets[] array of
buffers
            source_data_Packets[0] = Start_Adrr_of_Remaining_Bytes;



          for (encode_index=0 ; encode_index<Num_Encoded_Packets;
encode_index++) {

            //Get one encoded packet (out of Num_Encoded_Packets)
            build_fec(source_data_Packets, packet_Number, packet_Size,
header_added_Packet+HEADER_SIZE, encode_index);


            /*
            * Copy encoded packet to proxy_side_sendBuffer
            */
            //Add header (k, n, Seq.#, length[2-bytes], packet# in block)
```

```
                header_added_Packet[0] = (unsigned char) 0x00;
                header_added_Packet[1] = (unsigned char)packet_Number;  //Set k
i.e. # of data/source packets
                header_added_Packet[2] = (unsigned char)Num_Encoded_Packets;
//Set n i.e. # of enc.packets
                header_added_Packet[3] = (unsigned char)Sequence_Number;
//Set sequence #. Keep it here!!!
                header_added_Packet[5] = (unsigned
char)(packet_Size+HEADER_SIZE);  //Set length (2nd byte [low])
                header_added_Packet[4] = (unsigned
char)((packet_Size+HEADER_SIZE)>>8);  //Set length (1st byte [high])
                header_added_Packet[4] = (unsigned char)( ((unsigned char)(0x80))
| header_added_Packet[4]);  //Code so that <>0

                if (header_added_Packet[5] == (unsigned char)0x00) {
                        header_added_Packet[5] = (unsigned char)( ((unsigned
char)(0x80)) | header_added_Packet[5] );
                        header_added_Packet[4] = (unsigned char)( ((unsigned
char)(0x40)) | header_added_Packet[4] );
                } //if - code length fields so that <>0

                header_added_Packet[6] = encode_index+1;
                header_added_Packet[7] = (unsigned char) 0x00;


                //Increment sequence number after using it
                if (Sequence_Number < MAX_SEQ_NUMB_VALUE)
                        Sequence_Number++;
                else
                        Sequence_Number=1;



                // ++++++++++++++++++++++++++++++++++++++++++++

            /*
            * Send data to server-side proxy
            */


                //Check congestion control flag HoldBack before sending data and
```

```
wait if set
                mutex_lock(&HoldBack_Lock);
                while (HoldBack != 0) {
                        mutex_unlock(&HoldBack_Lock);
                        thr_yield();
                        mutex_lock(&HoldBack_Lock);
                }
                mutex_unlock(&HoldBack_Lock);


                #ifdef USE_DELAY

                // ----------------

                 if ( gettimeofday(start_time, NULL) <0 ) {
                        printf("ERROR(send_Control): gettimeofday error\n\n");
                        exit(1);
                 }

                 start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;



                while (1) {
                        if ( gettimeofday(curr_time, NULL) <0 ) {
                                printf("ERROR(send_Control): gettimeofday error\n\n");
                                exit(1);
                        }

                        curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;


                        if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                break;


                        //Since haven't reached timeout value yet have thread
sleep/yield
```

```
                thr_yield();

            } //while

            // ----------------

            #endif
            //---------
            //Increment send socket identifier/index (to change sending socket
used)
            if ( send_socket_index < (Connection_Number-1) )
                send_socket_index++;
            else
                send_socket_index = 0;
            //---------

            // printf("CHECK(sendControl)Sending via socket
%d\n\n",send_socket_index);


            //TCP-if ( send(servsideproxy_sockfd[send_socket_index], buf, n , 0)
< 0) {
                if (sendto(servsideproxy_sockfd[send_socket_index],
header_added_Packet, packet_Size+HEADER_SIZE, 0, (struct sockaddr *)
&ssp_name[send_socket_index],sizeof(struct sockaddr_in)) < 0) {
                    printf("ERROR(send/Control): While sending data to
serv-side proxy <0 \n\n");
                    exit(1);
                } //if



        // ++++++++++++++++++++++++++++++++++++++++++++


            //Update j & i values
            if (j < (Connection_Number-1))
                j++;
            else { //j wraps-around so update i also
                j = 0;
```

```
                    if (i < (S_PROXY_SIDE_BUFFER_SIZE-1))
                        i++;
                    else
                        i=0;
                }


            } //for - encode and write-out packets to send buffer



                // ********

            } //if
            else if (cumul_buf_Size_temp > cumul_buf_Size) { //error case
                printf("ERROR(server_to_proxy_Control): cumul_buf_Size_temp >
cumul_buf_Size\n\n");
                exit(1);
            } //else - error case



            //
================================================================
==============


  } //while(1)

  //
*****************************************************************************
***************

} //client_to_proxy_Control_ThreadFunc


/*
```

```
**************************************************************************
********************* */




/*
* Thread (one instance) used to receive congestion control messages from server-
side proxy(RNR/RR)
*/
void *Cong_Msg_Recv_ThreadFunc(void *JunkPTR) {

        int  n,m;
        unsigned char  cong_msg_buf[20];  //Buffer used to receive congestion
control messages
        unsigned char  CongMessage[20];  //Congestion control message
eventually written into here
        int  Last_Message_was_RNR=2;  //Flag. Init to a neutral value


        while (1) {


          /*
          * Read-in congestion control message from server-side proxy
          */
          strcpy(CongMessage,"");  //Init string
          m=0;
          while ( (n = recv(tcp_sock, cong_msg_buf, sizeof(cong_msg_buf),
0)) >0) {

                m=m+n;
                strncat(CongMessage, cong_msg_buf, n);
                if ( CongMessage[m-1] == '\0' )
                        break; //if '\0' in string then done so exit recv/while loop

          } //while
          if (n<=0) {
                printf("ERROR(Cong_Msg_Recv_Thread): n<=0 recv() \n\n");
                close(tcp_sock);
```

```
                exit(1);
        }


        /*
        * Read cong. control message received and act appropriately
        */
        if (strcmp(CongMessage, "RNR") == 0) {
                if (Last_Message_was_RNR == 1)
                        printf("ERROR/WARNING(Cong_Msg_Recv_Thread):
Multiple RNR messages received in a row\n\n");

                //Print out that "RNR" message received
                #ifdef FLOW_CNTRL_DEBUG_PRINT
                 printf("STATUS(Cong_Msg_Recv_Thread): RNR message
received\n\n");
                #endif

                Last_Message_was_RNR = 1;  //Set flag

                mutex_lock(&HoldBack_Lock);
                  HoldBack = 1;
                mutex_unlock(&HoldBack_Lock);

        }
        else if (strcmp(CongMessage, "RR") == 0) {
                if (Last_Message_was_RNR == 0)
                        printf("ERROR/WARNING(Cong_Msg_Recv_Thread):
Multiple RR messages received in a row\n\n");

                //Print out that "RR" message received
                #ifdef FLOW_CNTRL_DEBUG_PRINT
                 printf("STATUS(Cong_Msg_Recv_Thread): RR message
received\n\n");
                #endif

                Last_Message_was_RNR = 0;  //Set flag

                mutex_lock(&HoldBack_Lock);
                  HoldBack = 0;
                mutex_unlock(&HoldBack_Lock);
```

```
                }
                else { //Error case
                        printf("ERROR(Cong_Msg_Recv_Thread): CongMessage neither
RNR nor RR \n\n");
                        close(tcp_sock);
                        exit(1);
                } //else


        } //while(1)

} //Cong_Msg_Recv_ThreadFunc
```

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <sys/stat.h>
#include <thread.h>
#include <synch.h>
#include <sched.h>
#include <sys/uio.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <memory.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include "fec.c"



// Debuging "defines" (enable debug printouts)
//#define ENC_DEBUG_ON
//#define new2_PRINT_DEBUG
#define FLOW_CNTRL_DEBUG_PRINT


//#define CS_PRX_USE_NAME
#define CS_PRX_USE_IP
//#define SERV_USE_NAME
#define SERV_USE_IP
//#define EXTRA_RECV_PACKET_CHECKS
```

```
#define SSP_UDP_BASE_PORTNUMBER 8000  //Server-side proxy UDP base
portnumber i.e. (SSP_UDP_BASE_PORTNUMBER+thread ID)
            //(or (SSP_UDP_BASE_PORTNUMBER+thread ID-Connection_Number)) is
port number to be used by each thread
#define CSP_UDP_BASE_PORTNUMBER 7000  //Client-side proxy UDP base
portnumber
#define S_PORTNUMBER 6006       //Server-side proxy listens on this port
#define  FILENAME  "p_ss_proxy_file.txt"  //File that contains address & port #
of server
#define  CSP_FILENAME  "erss_cs_proxy_file.txt"  //File that contains client-side
proxy name(DNS)
#define  SOCK_BUFFER_SIZE  2048
#define  CS_TO_S_PROXYBUFFER_SIZE  200  //# of nodes in buffer.
#define  S_TO_CS_PROXYBUFFER_SIZE  4000  //# of nodes in buffer.
//#define  PROXY_SIDE_BUFFER_SIZE  10  //# of nodes in buffer.
#define  R_PROXY_SIDE_BUFFER_SIZE  50  //# of nodes in buffer.
#define  S_PROXY_SIDE_BUFFER_SIZE  2000  //# of nodes in buffer.
#define MAX_CONN_NUMB_PER_LOGICAL_CONN  5  //Max # of connections per
logical connection
#define SERV_SLEEP_COUNTER 1000  //NOT USED. Amount of times to call
thr_yield to implement a wait after server has closed
#define EXIT_TIME_DELAY  3  //NOT USED(??m is). Number of sec of time delay
before child proc. terminated after server  has closed
#define RNR_PROXY_RECV_BUF_SIZE_THRESH 0.8  //Percentage full treshhold of
proxy recv.buffer which if exceeded
                                                        //results in RNR
message being sent to serv-side proxy
#define RR_PROXY_RECV_BUF_SIZE_THRESH 0.6  //Percentage full treshhold of
proxy recv.buffer which if exceeded
                                                        //results in RNR
message being sent to serv-side proxy

#define K_DESIRED 2  //# of client_to_ssproxy_ProxyBuffer we WANT to
read(actual # read may differ)
#define THR_YIELD_MAX 1  //Max number of times will thread yield to read entry
from client_to_ssproxy_ProxyBuffer
                                        //Set it to 0 if don't want to wait at all

#define N_FACTOR 2  //N_FACTOR*k packets will be produced by encoding and
sent out(k = #data packets)
#define ENC_DEBUG_FILENAME  "er_ss_enc_dbg_file.txt" //File descriptor of a file
```

```
used for debug output in client-to-proxy thread
#define HEADER_SIZE 8  //Size (in bytes) of header added to encoded packets
(0x00,k, n, seq.#,length(2-bytes),packet# in block,0x00)
#define MAX_SEQ_NUMB_VALUE 255
#define HEADER_SRT_END_BYTES_CODE_VALUE (unsigned char)0x01
#define DEC_ARRAY_SIZE 10  //Number of array entries for decoding array
(array of gf*)
#define DEC_ARR_BUF_SIZE SOCK_BUFFER_SIZE  //Size of each decoding array
buffer
#define ENC_PACKET_BUF_SIZE  SOCK_BUFFER_SIZE  //Size of buffer containing
encoded packet(what you get from enc.func.)
#define ENC_SOURCE_BUF_BUFFER_SIZE  SOCK_BUFFER_SIZE  //Size of buffers
containing source data headed for encoding
#define SERV_RECV_SOCK_BUFFER_SIZE  1024  //Socket buffer size for server
receiving
#define TEMP_CUMUL_BUF_SIZE_TRESH  SERV_RECV_SOCK_BUFFER_SIZE  //If
total # of bytes to be encoded(&sent-out) each time
                                                        //is smaller than this value
then amalgamate into one big packet(for efficiency)
//#define  NETW_RECV_BUFFER_SIZE  40  // # of cells for each network receive
buffer (used in
                                                    //
proxyside_recvBuffering_ThreadFunc threads to buffer network UDP packets)

#define  DELAY_IN_MSEC 0
//#define USE_DELAY




/*
* Function prototypes
*/
void *proxyside_recv_ThreadFunc(void *);
void *serverside_send_ThreadFunc(void *);
void *serverside_recv_ThreadFunc(void *);
void *Cong_Msg_Recv_ThreadFunc(void *);
void *proxy_to_server_Control_ThreadFunc(void *);
void *server_to_proxy_Control_ThreadFunc(void *);
```

```
struct BufferNode {
        int  EntryFull;  //Flag (serves as semaphore). If 1 then value stored, else
is empty and can be written into
        long int   DataSize;  //Size of data stored in entry
        unsigned char  Data[SOCK_BUFFER_SIZE + HEADER_SIZE];
        int  Skip;  //Used only for proxy_side_recvBuffer. Recv. thread sets flag
to 1 to tell control to skip cell
        int  Control_Skip;  //Flag set by recv_Control "stating" that it has
skeeped the entry (because doesn't need
                                //more data for current block)

};



int   sockfd, servlen, cli_len, childpid;
int   newsockfd[MAX_CONN_NUMB_PER_LOGICAL_CONN];  //Array of UDP
sockets
int  tcp_sock;  //TCP socket used to receive the # of UDP connections to be used
struct sockaddr_in  ssp_name;  //Address of server-side proxy
int  nn;  //Used in setting up UDP socket addresses
struct sockaddr_in   cli_addr, serv_addr;
int   niaa;
unsigned char   buf_par[SOCK_BUFFER_SIZE];  //Buffer used by parent process to
receive # of connections opened
int   m_par, c_par;  //Counter vars
int   n_par;
int   Connection_Number;  //Number of connections to be opened for a given
proxy-to-proxy 'logical' connection
unsigned char   Connection_Number_str[32];  //String containing
Connection_Number as read from other proxy
int   server_sock;   //socket used to connect to server
struct BufferNode
csproxy_to_server_ProxyBuffer[CS_TO_S_PROXYBUFFER_SIZE];  //Buffer used
for client->server_side_proxy traffic
struct BufferNode
server_to_csproxy_ProxyBuffer[S_TO_CS_PROXYBUFFER_SIZE];  //Buffer used
for server_side_proxy->client traffic
struct BufferNode
proxy_side_sendBuffer[S_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that will
```

```
// be sent out to cl.side proxy
struct BufferNode
proxy_side_recvBuffer[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LO
GICAL_CONN]; //Used to store temporarily data that

//is received fromt to cl.side proxy
struct hostent *chp;  //Used to get client-side proxy address

int  RecvBufSizeCounter = 0;  //Used for congestion control(# of full cells of proxy
recv. buffer)
int  RecvBufSizeCounter_copy = 0;  //Copy of above counter. Used to unlock
mutex quickly
int  HoldBack = 0;  //Flag used for congestion control i.e. when 1 then send
threads pause
mutex_t  RecvBufSizeCounter_Lock;  //RecvBufSizeCounter semaphore
mutex_t  HoldBack_Lock;  //HoldBack semaphore
int  PROXY_RECV_BUF_CAPACITY;  //Capacity of proxy-side recv buffer. Is
R_PROXY_SIDE_BUFFER_SIZE*(#UDP connections)
mutex_t
Control_Skip_Lock[R_PROXY_SIDE_BUFFER_SIZE][MAX_CONN_NUMB_PER_LOGIC
AL_CONN];  //Mutex for setting/reseting Control_Skip flag. One mutex for each
cell of proxy_side_recvBuffer
int  mi,mj;  //Counter vars
int PFTPblock_size=10001;



int main(void) {

        /*
        * Init HoldBack mutex var - Used for congestion control i.e. when 1 then
send threads pause
        */
        mutex_init(&HoldBack_Lock, USYNC_THREAD, (void *) NULL);


        /*
        * Init RecvBufSizeCounter mutex var - Used for congestion control(# of
full cells of proxy recv. buffer)
        */
```

```
            mutex_init(&RecvBufSizeCounter_Lock, USYNC_THREAD, (void *) NULL);
            /*
            * Init Control_Skip_Lock mutex vars (one mutex for each cell of
proxy_side_recvBuffer)
            */
            for (mi=0; mi<R_PROXY_SIDE_BUFFER_SIZE; mi++)
                for (mj=0; mj<MAX_CONN_NUMB_PER_LOGICAL_CONN; mj++)
                        mutex_init(&Control_Skip_Lock[mi][mj], USYNC_THREAD,
(void *) NULL);


            //Create the socket
            if ( (sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
                perror("ERROR: socket creation \n");
                exit(1);
            }


            //Create (server-side) proxy address
            memset(&serv_addr, 0, sizeof(struct sockaddr_in));

            serv_addr.sin_family = AF_INET;
            serv_addr.sin_port = htons(S_PORTNUMBER);
            servlen = sizeof(struct sockaddr_in);

            niaa = INADDR_ANY;
            memmove(&serv_addr.sin_addr, &niaa, sizeof(long));


            //Bind socket to (server-side) proxy address
            if ( bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
                perror("ERROR: bind \n");
                exit(1);
            }


            //Listen for client connections
            if ( listen(sockfd, 5) < 0 ) {
                perror("ERROR: listen() \n");
                close(sockfd);
                exit(1);
```

```
        }
        for (;;) {

            cli_len = sizeof(cli_addr);
            tcp_sock = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);

            if (tcp_sock < 0) {
                    perror("ERROR: Can't create new socket (1st proxy listening
socket)\n");
                    exit(1);
            }


            /*
            * Read number of TOTAL connections to be opened (for this 'logical'
connection).
            * Format: string containing # followed by '\0'
            */

            strcpy(Connection_Number_str,"\0");   //Init

            m_par=0;   //Init counter
            while ((n_par = recv(tcp_sock, buf_par, sizeof(buf_par), 0)) > 0) {
                    m_par = m_par+n_par;
                    strncat(Connection_Number_str, buf_par, n_par);
                    if ( Connection_Number_str[m_par-1] == '\0' )
                            break; //if '\0' in string then done so exit recv/while loop

            }
            if (n_par<=0) {
                    printf("FATAL ERROR: n_par<=0 while recv. UDP connection #
(parent process)\n\n");
                    close(tcp_sock);
                    exit(1);
            }



            /*
            * - xNow that connection # has been received close the TCP socketx -
```

```
will use for cong.control messages
                */
                //close(tcp_sock);



                //Convert # of connections to int from string
                sscanf(Connection_Number_str,"%d", &Connection_Number);

                printf("CHECK(er_ss_proxy): %d UDP connections to be opened\n\n",
Connection_Number);



                /*
                * Now that we got the # of UDP connections to be used calculate the
proxy recv. buffer capacity
                */
                PROXY_RECV_BUF_CAPACITY =
R_PROXY_SIDE_BUFFER_SIZE*Connection_Number;



                /*
                * Wait for Connection_Number connections to be opened.
NOTE:Connection_Number is total connections to be opened
                */
                for (c_par=0; c_par<Connection_Number; c_par++) {

                        //Create socket
                        if ( (newsockfd[c_par] = socket(AF_INET, SOCK_DGRAM, 0))
< 0) {
                                printf("ERROR(er_ss_proxy): Can't create UDP socket
for i:%d\n\n", c_par);
                                exit(1);
                        }


                        //Create the address to bind the UDP socket. Port #:
```

```
(SSP_UDP_BASE_PORTNUMBER + c_par)
                    memset(&ssp_name, 0, sizeof(struct sockaddr_in));
                    ssp_name.sin_family = AF_INET;
                    ssp_name.sin_port = htons(SSP_UDP_BASE_PORTNUMBER +
c_par);
                    nn = INADDR_ANY;
                    memmove(&ssp_name.sin_addr, &nn, sizeof(long));


                    //Bind UDP socket to above address
                    if ( bind(newsockfd[c_par], (struct sockaddr *) &ssp_name,
sizeof(struct sockaddr_in)) <0 ) {
                              printf("ERROR: Can't bind UDP socket to port
%d\n\n",SSP_UDP_BASE_PORTNUMBER+c_par);
                              exit(1);
                    }


          } //for



          //Create child process
          if ( (childpid = fork()) < 0 ) {
                    perror("ERROR: Fork \n");
                    exit(1);
          }
          else if (childpid == 0) {   /* ****** child process ****** */



                    /* Child process vars */
                    struct hostent   *hp;   //used in connecting to server
                    struct sockaddr_in   name;   //used in connecting to server
                    int   len;   //used in connecting to server
                    int   thr_concurrency_counter;
                    thread_t
ThreadID[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1]; //Array of thread IDs
for thr_create calls
                    int
thr_ID_PTR[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1];  //Array of thr_ID
```

```
pointers used in thr_create calls
                    FILE   *fd;
                    int   Server_PortNumber;   //Port on which server listens
                    unsigned char   Server_Name[64];   //DNS name of server
                    unsigned char   Server_IP[64];   //IP of server
                    ulong_t  addr;
                    int   i,j;  //counter var
                    size_t
TaskStatus[MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1];  //Used in
thr_join() calls
                    unsigned char  client_side_proxy_name[55];  //String
containing clien-side proxy as read from file
                    unsigned char  client_side_proxy_IP[50];



                    close(sockfd);        //close original socket



                    //------------------code------------------

                    printf("CHECK(er_ss_proxy): CHILD process created \n\n");


                    /*
                    * Init the erasure code encoder/decoder
                    */
                    init_fec();


                    /*
                    * Initialize proxy buffers (i.e. set all flags to 0)
                    */
                    for (i=0; i<R_PROXY_SIDE_BUFFER_SIZE; i++) {
                            for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)                        {
                                    proxy_side_recvBuffer[i][j].EntryFull = 0;
                                    proxy_side_recvBuffer[i][j].Skip = 0;
                                    proxy_side_recvBuffer[i][j].Control_Skip = 0;
                            }
```

```
                }
                for (i=0; i<S_PROXY_SIDE_BUFFER_SIZE; i++) {
                        for (j=0; j<MAX_CONN_NUMB_PER_LOGICAL_CONN;
j++)            {
                                proxy_side_sendBuffer[i][j].EntryFull = 0;
                        }
                }


                for (i=0; i<CS_TO_S_PROXYBUFFER_SIZE; i++) {
                        csproxy_to_server_ProxyBuffer[i].EntryFull = 0;
                }
                for (i=0; i<S_TO_CS_PROXYBUFFER_SIZE; i++) {
                        server_to_csproxy_ProxyBuffer[i].EntryFull = 0;
                }



                /*
                * Read file to obtain address (name & port #) of server
                * Format: "port# server_name" ("'s not included )
                */
                //sem_lock not needed

                if ( (fd = fopen(FILENAME, "r")) == NULL) {
                        perror("ERROR(p_ss_proxy): Couldn't open file \n");
                        exit(1);
                }

                #ifdef SERV_USE_NAME
                fscanf(fd, "%d %s", &Server_PortNumber, Server_Name);
                #endif
                #ifdef SERV_USE_IP
                fscanf(fd, "%d %s", &Server_PortNumber, Server_IP);
                #endif


                //Check values read from file
                #ifdef SERV_USE_NAME
                printf("CHECK(p_ss_proxy): Server_Name: %s\n",
Server_Name);
```

```
                #endif
                #ifdef SERV_USE_IP
                printf("CHECK(p_ss_proxy): Server_IP: %s\n", Server_IP);
                #endif

                printf("CHECK(p_ss_proxy): Server_PortNumber: %d \n",
Server_PortNumber);


                fclose(fd);

                //sem_unlock not needed



                /*
                * Get server address & open connection to server
                */
                //-----------
                //Get server's address
                #ifdef SERV_USE_NAME
                if ( (hp = gethostbyname(Server_Name)) == NULL ) {
                        printf("ERROR(er_ss_proxy): Can't get server
address(%s)\n\n", Server_Name);
                        exit(1);
                }
                #endif
                #ifdef SERV_USE_IP
                        if ((int)(addr = inet_addr(Server_IP)) == -1) {
                                printf("IP-address must be of the form
a.b.c.d\n");
                                exit (2);
                        }
                        hp = gethostbyaddr((char *)&addr, sizeof(addr),
AF_INET);
                        if (hp == NULL) {
                                printf("host information for %s not found\n",
Server_IP);
                                exit (3);
                        }
                #endif
```

```
                //Create socket
                if ( (server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                        printf("ERROR(er_ss_proxy): Can't create socket for
server\n\n");
                        exit(1);
                }


                //Create server address
                memset(&name, 0, sizeof(struct sockaddr_in));

                name.sin_family = AF_INET;
                name.sin_port = htons(Server_PortNumber);
                memmove(&name.sin_addr, hp->h_addr_list[0], hp-
>h_length);
                len = sizeof(struct sockaddr_in);


                //Connect to server
                if ( connect(server_sock, (struct sockaddr *) &name, len) < 0
) {
                        printf("ERROR: Can't connect to server \n\n");
                        exit(1);
                }




                //-----------------------------------------------------------
                /*
                * Get the client-side proxy's name(DNS)
                */


                /*
                * Read file to obtain address (DNS) of server
                * Format: "client_side_proxy_name" ("'s not included )
                */

                if ( (fd = fopen(CSP_FILENAME, "r")) == NULL) {
```

```
                        perror("ERROR(er_ss_proxy): Couldn't open file to get
csprx name(or IP)\n");
                        exit(1);
                }

                #ifdef CS_PRX_USE_NAME
                 fscanf(fd, "%s", client_side_proxy_name);
                #endif

                #ifdef CS_PRX_USE_IP
                 fscanf(fd, "%s", client_side_proxy_IP);
                #endif


                //Check values read from file
                #ifdef CS_PRX_USE_NAME
                printf("CHECK(er_ss_proxy):client_side_proxy_name: %s\n",
client_side_proxy_name);
                #endif
                #ifdef CS_PRX_USE_IP
                printf("CHECK(er_ss_proxy):client_side_proxy_IP: %s\n",
client_side_proxy_IP);
                #endif



                fclose(fd);


                #ifdef CS_PRX_USE_NAME
                if ((chp = gethostbyname(client_side_proxy_name)) == NULL)
{
                        printf("ERROR: Can't get client-side proxy address
(gethostbyname)\n\n");
                        exit(1);
                }
                #endif
                #ifdef CS_PRX_USE_IP
                 if ((int)(addr = inet_addr(client_side_proxy_IP)) == -1) {
                        printf("IP-address must be of the form a.b.c.d\n");
                        exit (2);
```

```
                    }
                    chp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
                    if (chp == NULL) {
                            printf("host information for %s not found\n",
client_side_proxy_IP);
                            exit (3);
                    }
                    #endif


                    //---------------------------------------------------------------


                    //-----------


                    /*
                    * Init/fill-in thr_ID array. Thread IDs start from ZERO !!!
                    */
                    for (i=0; i<(MAX_CONN_NUMB_PER_LOGICAL_CONN*2+4+1);
i++)
                            thr_ID_PTR[i] = i;



                    /*
                    *
                    * NOTE: proxy-side sending threads have thread IDs from (0
to Connection_Number-1) and
                    * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
                    *
                    */


                    /*
                    * Spawn the proxy-side sending/receiving threads. Pair for
each connection w/ client-side proxy
                    */

                    for (i=Connection_Number; i<(2*Connection_Number); i++) {
```

```
                    thr_create(NULL, 0, proxyside_recv_ThreadFunc, (void
*) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    } //for
                    if (i != (2*Connection_Number))
                        printf("ERROR/CHECK(er_ss_proxy): i counter <>
2*Connection_Number \n");


                    /*
                    * Spawn the server-side sending/receiving threads. Just one
pair for the single connection to server
                    */
                    thr_create(NULL, 0, serverside_send_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, serverside_recv_ThreadFunc, (void *)
&thr_ID_PTR[i], 0, &ThreadID[i]);



                    /*
                    * Spawn Control threads
                    */
                    i++;
                    thr_create(NULL, 0, server_to_proxy_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);
                    i++;
                    thr_create(NULL, 0, proxy_to_server_Control_ThreadFunc,
(void *) &thr_ID_PTR[i], 0, &ThreadID[i]);



                    /*
                    * Spawn RNR/RR message receiving thread that will receive
congestion messages from serv-side proxy
                    */
                    i++;
                    thr_create(NULL, 0, Cong_Msg_Recv_ThreadFunc, (void *)
NULL, 0, &ThreadID[i]);
```

```
                    /*
                    *Set the thread concurrency. Could improve thread scheduling.
                    */
                    thr_concurrency_counter = 1*Connection_Number+4+1;
                    thr_setconcurrency(thr_concurrency_counter);


                    /*
                    * Make thr_join calls so that program doesn't terminate right
after creating threads
                    */
                    for (i=Connection_Number; i<(2*Connection_Number+4+1);
i++) {
                            thr_join(ThreadID[i], 0, (void *) &TaskStatus[i]);
                    }



            } //endelseif                    /* ****** child process ****** */



            /*
            * Close in parent the "passed off to child" UDP sockets & the TCP
socket tcp_sock(used for cong.control)
            */
            close(tcp_sock);
            for (c_par=0; c_par<Connection_Number; c_par++) {
                    close(newsockfd[c_par]);
            }


        } //endfor


} /*main*/
```

```
/*
**************************************************************************
********************* */


void *proxyside_recv_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i,k, m;  //counter vars
        unsigned char  *buf;  //Buffer into which received data is copied into &
data pasted to
        unsigned char  *bufStart;  //Used to store temporarily the socket buffer
starting address
        int  buf_size;  //Used because sizeof(buf) doesn't work for char *
        unsigned char  *tempbuf;  //Temp. buffer used to copy segments read
from buf & copy into proxy recv buffer at end
        int  tempbuf_Size;  //Current capacity of tempbuf i.e. # of bytes
currently stored
        unsigned char  *tempbuf_Start;  //Used to store temporarily tempbuf
starting address
        int  tempbuf_Capacity; //Capacity of tempbuf. Value set later on &
must remain constant
        unsigned char  *c_location;  //Contains result of memccpy operation
        int Total_buf_BytesRead;  //Number of bytes already read from socket
buffer(buf). Used in writing received data to proxy_side_recvBuffer
        int  ZeroByteAtEndLastRound=0; //Used as a flag. True(i.e. 1) when last
byte in buf(in 1st or nth round) was a 0x00 byte
        int  ConcatRemaining_buf_contents_to_buf_for_next_round = 0;
//Flag
        int  concated_to_buf_start_bytes = 0;
        unsigned char *new_received_data_start_addr_in_buf; //In case of
concating data at start of buf, this is the pointer to
                                        //the first byte in buf of the newly
```

```
received(from the network) data
        unsigned char  *dummy_buf;  //Buffer into to which socket data is
copoed into initially
        int  dummy_buf_size;
        struct sockaddr_in from;  //Used in recvfrom() call. Address of sender
is written into this var after every recvfrom()
        int len; //Used for recvfrom()
        int  Header_Derived_Packet_Length;  //Packet length as derived from
header
        unsigned char  hb_size, lb_size;  //Used in getting packet length from
header
        unsigned int  Next_Expected_Seq_Number,
Next_Expected_Seq_Number_copy, Previous_Seq_Number;
        unsigned int  Current_Packet_Derived_Seq_Number;
        int  Cells_To_Skip;
        int  Wrap_Around_Counter;  //Used in seq.# validation and skip count
calculation



        //Init buffer
        buf_size = 2*(SOCK_BUFFER_SIZE*2+2+HEADER_SIZE);  //buf_size was
SOCK_BUFFER_SIZE
        buf = (unsigned char *) malloc( buf_size*sizeof(unsigned char) );


        dummy_buf_size = (SOCK_BUFFER_SIZE*2+2+HEADER_SIZE);
//buf_size was SOCK_BUFFER_SIZE



        //Init Previous_Seq_Number
        Previous_Seq_Number = (thr_ID-Connection_Number+1) -
Connection_Number;


        /*
        * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
        * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
```

```
        * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
        * receiving threads
        */
        //Copy starting address of socket buffer
        bufStart=buf;


        i=0; //Init counter


        while (1) {


           len = sizeof(struct sockaddr_in);
           if ((n = recvfrom(newsockfd[thr_ID-Connection_Number], buf,
dummy_buf_size, 0, (struct sockaddr *) &from, &len)) > 0) {



            // ------------------ UDP packet handling ------------------



                /*
                *Check that first bytes are header - DEBUG
                */
                if ( !((buf[0]==(unsigned char)0x00) && (buf[1]!=(unsigned
char)0x00) && (buf[1]!=(unsigned char)0xff) && (buf[HEADER_SIZE-1]==(unsigned
char)0x00)) ) {
                                printf("ERROR(proxyside_recv): Header
does not verify\n\n");
                                exit(1);
                } //if - header incorrect



                /*
                *Verify length etc. -DEBUG
                */
                //Get length
```

```
                    if ( ((unsigned char)(0x40) & buf[4]) != (unsigned char)0x00) {
                            hb_size = ((unsigned char)(0x3f) & buf[4]);
                            lb_size = ((unsigned char)(0x7f)) & buf[5];
                    }
                            else {
                                    hb_size = ((unsigned char)(0x7f)) & buf[4];
                                    lb_size = buf[5];
                            }

                            Header_Derived_Packet_Length = hb_size;
                            Header_Derived_Packet_Length =
(Header_Derived_Packet_Length<<8);
                            Header_Derived_Packet_Length =
Header_Derived_Packet_Length | lb_size;



                            //Verify length
                            if (Header_Derived_Packet_Length != n) {
                                    printf("ERROR(proxyside_recv): Header derived
length != recv.packet length\n\n");
                                    exit(1);
                            }



                            #ifdef EXTRA_RECV_PACKET_CHECKS


                            /*
                            *Check header bytes <>0x00 (except start/end bytes) -
DEBUG
                            */
                            for (m=1; m<HEADER_SIZE-1; m++) {
                                    if (buf[m] == 0) {
                                            printf("ERROR: %d th byte of header is
0x00\n\n", m+1);

                                            exit(1);
                                    } //if
                            }
```

```
                    #endif

                    /*
                    * Check the sequence number. Must be
Next_Expected_Seq_Number OR a multiple of
                    * Connection_Number + Next_Expected_Seq_Number
                    */


                    Next_Expected_Seq_Number = Previous_Seq_Number
+ Connection_Number;
                    //If exceeded seq.# max value wrap it around
                    if (Next_Expected_Seq_Number >
MAX_SEQ_NUMB_VALUE)
                    Next_Expected_Seq_Number =
(Next_Expected_Seq_Number-MAX_SEQ_NUMB_VALUE);


                    Current_Packet_Derived_Seq_Number = buf[3];



                    /*
                    * NOTE: Packet seq.# must be ==
Next_Expected_Seq_Number OR be Next_Expected_Seq_Number
                    *           + a multiple of Connection_Number OR
Packet seq.# may have wrapped around skipping cells
                    */
                    if (Current_Packet_Derived_Seq_Number >
Next_Expected_Seq_Number) {
                        if  ( ((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number) % Connection_Number) != 0) {
                            printf("ERROR: Seq.# INCORRECT(#1).
Expecting %d and got
%d\n\n",Next_Expected_Seq_Number,Current_Packet_Derived_Seq_Number);
                            exit(1);
                            }
                    } //if - Current_Packet_Derived_Seq_Number >
```

```
Next_Expected_Seq_Number
                        else if (Current_Packet_Derived_Seq_Number <
Next_Expected_Seq_Number) { //Wrap-around case
                                Next_Expected_Seq_Number_copy =
Next_Expected_Seq_Number;

                                Wrap_Around_Counter = 0; //Reset
                                while (Next_Expected_Seq_Number_copy <
MAX_SEQ_NUMB_VALUE) {
                                Next_Expected_Seq_Number_copy =
Next_Expected_Seq_Number_copy + Connection_Number;
                                        Wrap_Around_Counter++;
                                }

                                //Now that Next_Expected_Seq_Numbe_copy >
MAX_SEQ_NUMB_VALUE wrap it
                                Next_Expected_Seq_Number_copy =
(Next_Expected_Seq_Number_copy - MAX_SEQ_NUMB_VALUE);

                                if ( ((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number_copy) % Connection_Number) != 0) {
                                        printf("ERROR: Seq.# INCORRECT(#2).
Expecting %d and got
%d\n\n",Next_Expected_Seq_Number,Current_Packet_Derived_Seq_Number);
                                        exit(1);
                                }

                        } //else



                        Previous_Seq_Number =
Current_Packet_Derived_Seq_Number;



                        //If sequence # != Next_Expected_Seq_Number then
skip cells & clear/reset buf

                        if (Current_Packet_Derived_Seq_Number !=
```

```
Next_Expected_Seq_Number) {

                                printf("CHECK(recv[%d]): Expecting %d and got
%d\n\n",
thr_ID,Next_Expected_Seq_Number,Current_Packet_Derived_Seq_Number);

                                //Calculate # of cells to skip
                                if (Current_Packet_Derived_Seq_Number >
Next_Expected_Seq_Number) {

Cells_To_Skip=(Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number)/Connection_Number;
                                        printf("CHECK: in 1st Cell to Skip
case\n\n");
                                }
                                else {  //Current_Packet_Derived_Seq_Number <
Next_Expected_Seq_Number
                                        Cells_To_Skip = Wrap_Around_Counter +
((Current_Packet_Derived_Seq_Number-
Next_Expected_Seq_Number_copy)/Connection_Number);
                                        printf("CHECK: in 2nd Cell to Skip
case\n\n");
                                }


                                printf("STATUS(proxyside_recv[%d]): Going to
skip %d cells w/ i=%d\n\n",thr_ID,Cells_To_Skip,i);


                                for (m=1; m<=Cells_To_Skip; m++) {

                                        //Wait for cell to be read
                                        while ( (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull!=0) || (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Skip!=0) ) {
                                                thr_yield();
                                        }  //while


                        mutex_lock(&Control_Skip_Lock[i][thr_ID-
```

```
Connection_Number]);

                        if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip == 0) {
                                                //Set Skip bit for cell

proxy_side_recvBuffer[i][thr_ID-Connection_Number].Skip = 1;

                                        printf("CHECK(recv)[%d]: Skip bit set in
[%d][%d]\n\n",thr_ID,i,thr_ID-Connection_Number);


mutex_lock(&RecvBufSizeCounter_Lock);
                                        RecvBufSizeCounter++;  //Increment

mutex_unlock(&RecvBufSizeCounter_Lock);

                        }  //if
                        else if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip==1) {  //Control_Skip is set

proxy_side_recvBuffer[i][thr_ID-Connection_Number].Control_Skip = 0;
                                        }  //else if - Control_Skip is set
                                        else {

printf("ERROR(proxy_side_recv): Control_Skip != 0,1\n\n");
                                                exit(1);
                                        }  //else - error case


mutex_unlock(&Control_Skip_Lock[i][thr_ID-Connection_Number]);
                                        // *********



                                        //Increment counter
                                        if (i <
(R_PROXY_SIDE_BUFFER_SIZE-1))
                                                i++;
                                        else
                                                i = 0;
```

```
                                        } //for - skip cells


                                        printf("STATUS(proxyside_recv[%d]):
Skipped %d cells w/ i=%d now\n\n",thr_ID,Cells_To_Skip,i);


                                    } //if - seq.# != next expected seq.#



                                    // ------------------ UDP packet handling ----------------
--


                 } //if
                else {
                 printf("FATAL ERROR(prx_side_recv_Thread): n<=0 while recv.\n");
                                printf("(Non-fatal if end of client-server
communication)\n\n");
                                close(newsockfd[thr_ID-Connection_Number]);
                                exit(1);
                 }



                                /*
                                * Copy data from temp. buffer (buf) into
proxy_side_recvBuffer & set size and flag
                                */
                                while ( (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull!=0) || (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Skip!=0) ) {
                                        printf("CHECK(prx_recv[%d]): #2 waiting for
proxy_side_recvBuffer!=0\n\n",thr_ID);
```

```
                    thr_yield();
                } //while


                mutex_lock(&Control_Skip_Lock[i][thr_ID-
Connection_Number]);

                 if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip == 0) {


memmove(proxy_side_recvBuffer[i][thr_ID-Connection_Number].Data, buf, n);
                    proxy_side_recvBuffer[i][thr_ID-
Connection_Number].DataSize = n;
                    proxy_side_recvBuffer[i][thr_ID-
Connection_Number].EntryFull = 1;

                    //Since data just written into proxy-side
recv. buffer increment RecvBufSizeCounter counter
                    mutex_lock(&RecvBufSizeCounter_Lock);
                 RecvBufSizeCounter++;  //Increment to offset
default decrement below
                mutex_unlock(&RecvBufSizeCounter_Lock);

                } //if
                else if (proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip==1) {  //Control_Skip is set
                    proxy_side_recvBuffer[i][thr_ID-
Connection_Number].Control_Skip = 0;
                    printf("STATUS/CHECK(proxy_recv):
**Control said to skip cell(refers to BELOW)** \n");
                } //else if - Control_Skip is set
                else {
                    printf("ERROR(proxy_side_recv):
Control_Skip != 0,1\n\n");
                    exit(1);
                } //else - error case

            mutex_unlock(&Control_Skip_Lock[i][thr_ID-
```

```
Connection_Number]);


                        //Increment counter
                        if (i < (R_PROXY_SIDE_BUFFER_SIZE-1))
                                        i++;
                        else
                                i = 0;


        } //while - 1


} //proxyside_recv_ThreadFunc


/*
**************************************************************************
********************** */


void *serverside_send_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i; //counter var
        unsigned char   buf[SOCK_BUFFER_SIZE];



        i=-1;
        while (1) {

           /*
           * Get data from csproxy_to_server_ProxyBuffer;
           */
```

```
                //Increment counter
                if (i < (CS_TO_S_PROXYBUFFER_SIZE-1))
                        i++;
                else
                        i = 0;


                while (csproxy_to_server_ProxyBuffer[i].EntryFull != 1) {
                        thr_yield();
                }
                memmove(buf, csproxy_to_server_ProxyBuffer[i].Data,
csproxy_to_server_ProxyBuffer[i].DataSize);
                n = csproxy_to_server_ProxyBuffer[i].DataSize;
                csproxy_to_server_ProxyBuffer[i].EntryFull = 0;



            /*
            * Send data to server
            */
            if (n>0) {
                if ( send(server_sock, buf, n, 0) < 0) {
                        printf("ERROR(p_ss_proxy/server_send_thread): While
sending data to server <0 \n\n");
                        exit(1);
                }
            }


      } //while


} //serverside_send_ThreadFunc


/*
**********************************************************************************
********************** */

void *serverside_recv_ThreadFunc(void *thr_ID_PTR) {
```

```
        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   n;
        int   i,j; //counter vars
        unsigned char   buf[SERV_RECV_SOCK_BUFFER_SIZE];
        time_t  serv_sock_close_time;  //timestamp use in implementing delay
vbefore terminating child process

        int  recv_counter;
        int  recv_data;


        i=-1;
        while ((n = recv(server_sock, buf, sizeof(buf), 0)) > 0) {


            /*
            * Copy received data to server_to_csproxy_ProxyBuffer;
            */

            //Increment counter
            if (i < (S_TO_CS_PROXYBUFFER_SIZE-1))
                    i++;
            else
                    i = 0;



            while (server_to_csproxy_ProxyBuffer[i].EntryFull != 0) {
                    printf("CHECK(serv_recv): waiting for
server_to_csproxy_ProxyBuffer !=0\n\n",thr_ID);
                    thr_yield();
            }
            memmove(server_to_csproxy_ProxyBuffer[i].Data, buf, n);
            server_to_csproxy_ProxyBuffer[i].DataSize = n;
            server_to_csproxy_ProxyBuffer[i].EntryFull = 1;
```

```
        } //while
        if (n<=0) {
            printf("FATAL ERROR: n<=0 while recv. from server (server-side
proxy) \n");
            printf("(Not fatal if client-server communication over)\n\n");
            /*
            * Client has closed so won't tranmsit anything else. Implement a wait
using thr_yield() ro
            * allow for any data remaining to be sent (& received by) to client
            */

            serv_sock_close_time = time(NULL);

            close(server_sock);

            while ( ((time_t) difftime(time(NULL), serv_sock_close_time)) <=
EXIT_TIME_DELAY)
                thr_yield();
                exit(1);
            }

        }

} //serverside_recv_ThreadFunc



/*
**********************************************************************************
********************** */


void *proxy_to_server_Control_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);
```

```
        int   i,j,k; //counter vars
        int  RNR_flag = 0; //True if an RNR message has been sent to serv-side
proxy.
        unsigned char  cong_msg_buf[1024]; //byte buffer used to send RR &
RNR messages to serv-side proxy
        int cong_Control_Check_Counter;  //Used to control when cong.cntrl
checks occur
        unsigned int  Next_Expected_Seq_Number=0;  //Between 1 and
MAX_SEQ_NUMB_VALUE (not 0 to MAX_SEQ_NUMB_VALUE)!!!
        unsigned int  Block_First_Seq_Num;  //Seq.# of first packet of a block
        int  Encoded_Packets_Number;  //Number of encoded packets for
current block(i.e. what is known as "n" in header)
        int  Encoded_Packets_Read;  //Counter var. Used in reading packets
from every block
        int  Source_Packets_Number;  //Number of source packets for current
block(i.e. what is known as "k" in header)
        int  Source_Packets_Read;  //Number of source/data packets already
read from current block
        int  Curr_Block_Packet_Size;  //Size if packets(not including header) of
a given block
        int  RR_counter=0;  //Bookeeping var that just keep track of RR
messages sent out
        int  RNR_counter=0;  //Bookeeping var that just keep track of RNR
messages sent out
        int  size_temp1, size_temp2;  //Utility ints
        gf *dec_src[DEC_ARRAY_SIZE];  //Used as argument in decoding
function
        gf *dec_dst[DEC_ARRAY_SIZE];  //Used as argument in decoding
function
        int dec_index[DEC_ARRAY_SIZE];  //Used as argument in decoding
function. Indexes of packets handed to dec. function
        int  ic;  //Counter var
        int  EPR_Start=1;
        int  Block_First_Seq_Num_Set = 0;  //Flag
        int  Number_of_Packets_Skipped;
        int  CONG_CNTRL_CHECK_PERIOD;  //Period of congestion control check.
CAUTION: If this value is too large then
                                                //might fall into deadlock (i.e.
not sending a RR at all after a RNR)!!!
        int  last_index_value;  //Used in index calculation for decoding
```

```
            /*
            * Set congestion control check period
            */
            CONG_CNTRL_CHECK_PERIOD = 0;



            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
            * receiving threads
            */


            /*
            * Allocate memory to dec_src and dec_dst
            */
            for (ic=0; ic<DEC_ARRAY_SIZE; ic++) {
                dec_src[ic]= (gf *) malloc(DEC_ARR_BUF_SIZE*sizeof(gf));
                dec_dst[ic]= (gf *) malloc(DEC_ARR_BUF_SIZE*sizeof(gf));
            }  //for



            k = 0;  //Init counter var
            i = 0;      //Init counter var
            j = 0;      //Init counter var

            cong_Control_Check_Counter = CONG_CNTRL_CHECK_PERIOD;  //Init


  while (1) {  //Handle current block of packets
```

```
        while ( (proxy_side_recvBuffer[i][j].EntryFull != 1) &&
(proxy_side_recvBuffer[i][j].Skip != 1) )
                thr_yield();


        if (proxy_side_recvBuffer[i][j].Skip == 1) {


                //Caclulate next expected sequence number. Inited to 0. Starts
from 1 (1->MAX_SEQ_NUMB_VALUE)
                if (Next_Expected_Seq_Number < MAX_SEQ_NUMB_VALUE)
                        Next_Expected_Seq_Number++;
                else
                        Next_Expected_Seq_Number = 1;



                //Reset Skip bit
                proxy_side_recvBuffer[i][j].Skip = 0;



                /*
                * Get sequence # of first packet of block (if not done already)
                */
                if (Block_First_Seq_Num_Set != 1) {
                        Block_First_Seq_Num = Next_Expected_Seq_Number;
                        Block_First_Seq_Num_Set = 1;
                }  //if



                //Skip current cell. Have for loop below start after skipped
cells
                EPR_Start = EPR_Start++;


                // i,j incrementation
```

```
                if (j<(Connection_Number-1))
                        j++;
                else {
                        j = 0;

                        if (i<(R_PROXY_SIDE_BUFFER_SIZE-1))
                                i++;
                        else
                                i = 0;
                }
                // i,j incrementation


                //Proceed to next cell
                continue;


        } //if - must Skip current cell



        /*
        * Now that we got out of skipping first packet(s) check if we skipped
entire block
        */
        Number_of_Packets_Skipped = EPR_Start - 1;
        if (Number_of_Packets_Skipped >=
proxy_side_recvBuffer[i][j].Data[6]) {
                printf("ERROR/STATUS: Entire block skipped.
Terminating\n\n");
                exit(1);
        } //if - check if entire block skipped



        /*
        * Get sequence # of first packet of block (if not already done)
        */
        if (Block_First_Seq_Num_Set != 1) {
                Block_First_Seq_Num = proxy_side_recvBuffer[i][j].Data[3];
        } //if
```

```
                else
                        Block_First_Seq_Num_Set = 0;



                /*
                * NOTE: Header format: (0x00, k, n, seq.#,length(2-bytes), packet#
in block0x00)
                */


                /*
                * Get number of packets for current block i.e. "n"
                */
                Encoded_Packets_Number = proxy_side_recvBuffer[i][j].Data[2];


                /*
                * Get number of source/data packets for this block i.e. "k"
                */
                Source_Packets_Number = proxy_side_recvBuffer[i][j].Data[1];


                /*
                * Get packet size(NOT including header) for current block
                */
                Curr_Block_Packet_Size = proxy_side_recvBuffer[i][j].DataSize -
HEADER_SIZE;



                /*
                * Reset vars
                */
                Source_Packets_Read = 0;



                /*
                * Go through Encoded_Packets_Number entries of proxy-side receive
```

```
buffer i.e. one block of "n" packets
                */
                for (Encoded_Packets_Read=1;
Encoded_Packets_Read<=Encoded_Packets_Number; Encoded_Packets_Read++)
{


                        //Caclulate next expected sequence number. Inited to 0. Starts
from 1 (1->MAX_SEQ_NUMB_VALUE)
                        if (Next_Expected_Seq_Number < MAX_SEQ_NUMB_VALUE)
                                Next_Expected_Seq_Number++;
                        else
                                Next_Expected_Seq_Number = 1;


                        //while check and if used to be here before Control_Skip


                if (Source_Packets_Read < Source_Packets_Number) {  //i.e. not
enough packets read yet for decoding



                        // ********

                        //Wait until entry full or recv. thread "says" to skip current cell
                        while ( (proxy_side_recvBuffer[i][j].EntryFull != 1) &&
(proxy_side_recvBuffer[i][j].Skip != 1) )
                                thr_yield();



                        if (proxy_side_recvBuffer[i][j].EntryFull == 1) {

                                //Check if seq.number valid (stored in 4th byte of
header)
                                if (proxy_side_recvBuffer[i][j].Data[3] !=
Next_Expected_Seq_Number) {
                                        printf("ERROR(proxy_to_server_Control):
Sequence # is WRONG. %d expected but got %d
\n\n",Next_Expected_Seq_Number,proxy_side_recvBuffer[i][j].Data[3]);
```

```
                        exit(1);
                    } //if - seq.number is NOT next expected seq.number



                    //Error check: check if k and n of current packet is
thart of current block
                    if (proxy_side_recvBuffer[i][j].Data[1] !=
Source_Packets_Number) {

                        printf("ERROR(er_ss/proxy_to_server_Control):
Source_Packets_Number(k) is not that expected \n\n");
                        exit(1);
                    } //if
                    if (proxy_side_recvBuffer[i][j].Data[2] !=
Encoded_Packets_Number) {

                        printf("ERROR(er_ss/proxy_to_server_Control):
Encoded_Packets_Number(n) is not that expected \n\n");
                            exit(1);
                        } //if


                        //Error check: compare size with
Curr_Block_Packet_Size
                    if ( (proxy_side_recvBuffer[i][j].DataSize-HEADER_SIZE)
!= Curr_Block_Packet_Size) {
                        printf("ERROR(er_ss/proxy_to_server_Control):
Size of read packet<>Curr_Block_Packet_Size\n\n");
                        exit(1);
                    } //if - error check for size


                } //if - if data in cell perfrom some checks

                // ********



                //Read cell data if cell full, else if Skip specified just reset Skip
flag
```

```
                    if (proxy_side_recvBuffer[i][j].EntryFull == 1) {


                            memmove(dec_src[Source_Packets_Read],
proxy_side_recvBuffer[i][j].Data+HEADER_SIZE,
proxy_side_recvBuffer[i][j].DataSize-HEADER_SIZE);

                            if (proxy_side_recvBuffer[i][j].Data[3] >=
Block_First_Seq_Num) {
                                    dec_index[Source_Packets_Read] =
proxy_side_recvBuffer[i][j].Data[3] - Block_First_Seq_Num;
                                    last_index_value =
dec_index[Source_Packets_Read];
                            }
                            else if (proxy_side_recvBuffer[i][j].Data[3] <
Block_First_Seq_Num)
                                    dec_index[Source_Packets_Read] =
last_index_value + proxy_side_recvBuffer[i][j].Data[3];

                            proxy_side_recvBuffer[i][j].EntryFull = 0;


                                    //Increment counter of # of packets read from
current block
                            Source_Packets_Read++;


                    }  //if
                    else if (proxy_side_recvBuffer[i][j].Skip == 1) {
                            proxy_side_recvBuffer[i][j].Skip = 0;
                    }  //else if
                    else {
                            printf("ERROR(er_cs/proxy_to_server_Control): Neither
entry full nor Skip set\n\n");
                            exit(1);
                    }  //else


            }  //if - # of packets read until now < Source_Packets_Number i.e.
not enough packets yet
```

```
            else {  //Source_Packets_Read packets have been read i.e. enough
data packets obtained for decoding

                        // *********
                                mutex_lock(&Control_Skip_Lock[i][j]);


                                //Error check
                                if ( (proxy_side_recvBuffer[i][j].EntryFull == 1) &&
(proxy_side_recvBuffer[i][j].Skip == 1) ) {
                                        printf("ERROR(recv_Control): EntryFull && Skip
==1\n\n");
                                        exit(1);
                                }

                                if (proxy_side_recvBuffer[i][j].EntryFull == 1) {
                                proxy_side_recvBuffer[i][j].EntryFull = 0;
                        }  //if
                        else if (proxy_side_recvBuffer[i][j].Skip == 1) {
                                proxy_side_recvBuffer[i][j].Skip = 0;
                        }  //else if
                        else {
                                proxy_side_recvBuffer[i][j].Control_Skip = 1;
                                mutex_lock(&RecvBufSizeCounter_Lock);
                                 RecvBufSizeCounter++;          //Increment to offset
decrement below
                                mutex_unlock(&RecvBufSizeCounter_Lock);
                        }  //else


                                mutex_unlock(&Control_Skip_Lock[i][j]);
                                // *********

            }  //else - Source_Packets_Read packets have been read i.e. enough
data packets obtained for decoding



                        // i,j incrementation
                        if (j<(Connection_Number-1))
```

```
                j++;
        else {
                j = 0;

                if (i<(R_PROXY_SIDE_BUFFER_SIZE-1))
                        i++;
                else
                        i = 0;
        }
        // i,j incrementation



        // ------------------CONGESTION CONTROL-------------------------

        /*
        * Congestion control check perfromed every
CONG_CNTRL_CHECK_PERIOD+1 rounds. Is VAR(not constant)!!!
        */


    //Regardless of if will perform congestion control check, packet
reception/reading must be
    // logged i.e. RecvBufSizeCounter decremented
    mutex_lock(&RecvBufSizeCounter_Lock);
     RecvBufSizeCounter--;
     RecvBufSizeCounter_copy = RecvBufSizeCounter;
    mutex_unlock(&RecvBufSizeCounter_Lock);


    if (cong_Control_Check_Counter == 0) {

            //Reset cong_Control_Check_Counter
            cong_Control_Check_Counter = CONG_CNTRL_CHECK_PERIOD;



       if (RecvBufSizeCounter_copy<0) {
            printf("ERROR: RecvBufSizeCounter < 0\n\n");
            exit(1);
       }
```

```
            if (RNR_flag == 0) {  //Send RNR message IF congested

                    if ( (RecvBufSizeCounter_copy/PROXY_RECV_BUF_CAPACITY)
>= RNR_PROXY_RECV_BUF_SIZE_THRESH ) {
                            RNR_flag = 1;  //Set flag

                            //Send ReceiveNotReady-RNR message("RNR" NULL
terminated string) to server-side proxy
                            strcpy(cong_msg_buf, "RNR");
                            if ( send(tcp_sock, cong_msg_buf,
strlen(cong_msg_buf)+1, 0) <0) {
                                    printf("ERROR: send()<0 for RNR  \n\n");
                                    close(tcp_sock);
                                    exit(1);
                            }

                            RNR_counter++;  //Bookeeping var
                            printf("STATUS: %d RNR messages sent out until
now\n\n",RNR_counter);

                    }  //if
            }  //if - RNR_flag == 0
            else if (RNR_flag == 1)  {  //RNR_flag == 1 & thus send RR message
if not congested anymore
                    if ( (RecvBufSizeCounter_copy/PROXY_RECV_BUF_CAPACITY)
< RR_PROXY_RECV_BUF_SIZE_THRESH ) {

                            RNR_flag = 0;  //Set flag

                            //Send ReceiveReady-RR message("RR" NULL terminated
string) to server-side proxy
                            strcpy(cong_msg_buf, "RR");
                            if ( send(tcp_sock, cong_msg_buf,
strlen(cong_msg_buf)+1, 0) <0) {
                                    printf("ERROR: send()<0 for RR  \n\n");
                                    close(tcp_sock);
                                    exit(1);
                            }
```

```
                        RR_counter++;  //Bookeeping var
                        printf("STATUS: %d RR messages sent out until
now\n\n",RR_counter);

                    }  //if

                } //elseif - RNR_flag == 1
                else  //error case
                    printf("ERROR: RNR_flag neither 0 nor 1 \n\n");


                } //if - cong_Control_Check_Counter == 0
            else {  //else - cong_Control_Check_Counter > 0
                    cong_Control_Check_Counter--;
            } //else - cong_Control_Check_Counter > 0


                // ------------------CONGESTION CONTROL-----------------------


        } //for - Go through Encoded_Packets_Number entries of proxy-side
receive buffer



        /*
        * Having read entire current block do error check if enough packets read
needed for decoding
        */
        if (Source_Packets_Read != Source_Packets_Number) {
            printf("ERROR(er_cs/proxy_to_server_Control): Not enough packets
for decoding(or too many)\n\n");
            exit(1);
        }  //if


        /*
        * Having read current block and having sufficient # of packets for
decoding, proceed w/ decoding
        * and writing to csproxy_to_server_ProxyBuffer
        */
```

```
            //Decode - decode_fec(gf *src[], gf *dst[], int index[], int k, int sz);
            decode_fec(dec_src, dec_dst, dec_index, Source_Packets_Number,
Curr_Block_Packet_Size);


            //Write decoded packets to csproxy_to_server_ProxyBuffer
            for (ic=0; ic<Source_Packets_Number; ic++) {

                while (csproxy_to_server_ProxyBuffer[k].EntryFull != 0)
                        thr_yield();

                memmove(csproxy_to_server_ProxyBuffer[k].Data, dec_dst[ic],
Curr_Block_Packet_Size);
                csproxy_to_server_ProxyBuffer[k].DataSize =
Curr_Block_Packet_Size;
                csproxy_to_server_ProxyBuffer[k].EntryFull = 1;


                if (k < (CS_TO_S_PROXYBUFFER_SIZE-1))
                                k++;
                        else
                                k = 0;

            } //for - write decoded packets into csproxy_to_server_ProxyBuffer



                //Reset
                EPR_Start = 1;


  } //while (1) - //Handle current block of packets


} //proxy_to_server_Control_ThreadFunc


/*
**********************************************************************
********************** */
```

```
void *server_to_proxy_Control_ThreadFunc(void *thr_ID_PTR) {

        /*
        * Get thread ID
        */
        int  thr_ID = *((int *) thr_ID_PTR);

        int   i,j,k; //counter vars
        int  K_actual=0;  //# of entries actually read(& to be encoded) from
client_to_ssproxy_ProxyBuffer
        int  m, thryield_counter, p;  //Counter vars
        unsigned char *cumul_buf;  //Cumulative buffer into which packets read
from client_to_ssproxy_ProxyBuffer are copied into
        int  cumul_buf_Size=0;  //Current size(i.e. # entries full) of cumul_buf
        int  cumul_buf_Size_temp=0;  //A temp var
        unsigned char *cumul_buf_Start;  //Used to store original starting
address of cumul_buf
        int  packet_Size=0;  //Size of packets to be encoded and writeen to
send-out buffer
        int  packet_Number=0;  //Number of packets to be encoded. Equal to
K_actual in general
        int  encode_index;  //COunter used as index when calling encoding
function
        gf *encoded_Packet;  //Used in encoding function call. Contains encoded
packet
        gf *source_data_Packets[K_DESIRED];  //Array of buffers containing
source(data) packets to be used in encoding
        int  Num_Encoded_Packets=0;  //Total number of packets to be
produced from encoding
        FILE *enc_debug_fd;  //File descriptor of a file used for debug output
ENC_DEBUG_FILENAME
        unsigned char
header_added_Packet[SOCK_BUFFER_SIZE+HEADER_SIZE];  //Packet copied into
send buffer. Includes header plus the encoded packet
        unsigned int  Sequence_Number=1;  //Sequence number used in header.
Inited to 1!!!
        unsigned char *Start_Adrr_of_Remaining_Bytes;

        struct timeval  *start_time;
```

```
        struct timeval  *curr_time;
        long  start_time_in_msec, curr_time_in_msec;

        struct sockaddr_in
csp_name[MAX_CONN_NUMB_PER_LOGICAL_CONN];  //Client-side proxy address
        int  send_socket_index = -1;


        /*
        * NOTE: sending thread will place 0x00 bytes on left & right of the
header. Thus the first 0x00 byte
        * will be followed by k(# of source/data packets) which is <>0x00 and
<>0xFF so header is uniquely
        * identified at receiver
        */


  //
**********************************************************************************
***************

        /*
        * Allocate memory to time vars
        */
        start_time = (struct timeval *) malloc(sizeof(struct timeval));
        curr_time = (struct timeval *) malloc(sizeof(struct timeval));


        /*
        * Setup addresses on client-side proxy with which this thread
communicates (multiple of them !!)
        */
        for (i=0; i<Connection_Number; i++) {

            memset(&csp_name[i], 0, sizeof(struct sockaddr_in));
            csp_name[i].sin_family = AF_INET;
            csp_name[i].sin_port = htons(CSP_UDP_BASE_PORTNUMBER + i);
            memcpy(&csp_name[i].sin_addr, chp->h_addr_list[0], chp-
>h_length);  //chp has been set above in child

        }  //for
```

```
            /*
            * Open encoding debug file descriptor
            */
            #ifdef ENC_DEBUG_ON
              enc_debug_fd = fopen(ENC_DEBUG_FILENAME, "w");
            #endif
            /*
            * Allocate memory to encoded_Packet and source_data_Packets[]
            */
            encoded_Packet = (gf *) malloc(ENC_PACKET_BUF_SIZE*sizeof(gf));

            for (m=0; m<K_DESIRED; m++) {
                source_data_Packets[m]= (gf *)
malloc(ENC_SOURCE_BUF_BUFFER_SIZE*sizeof(gf));
            }  //for



            /*
            * Allocate memory to cumul_buf
            */
            cumul_buf = (unsigned char *)
malloc((K_DESIRED+1)*SOCK_BUFFER_SIZE*sizeof(unsigned char));


            /*
            * Store starting address of cumul_buf
            */
            cumul_buf_Start = cumul_buf;



            /*
            * NOTE: proxy-side sending threads have thread IDs from (0 to
Connection_Number-1) and
            * proxy-side receiving threads have thread IDs from
(Connection_Number to 2*Connection_Number-1)
            * Therefore (thr_ID-Connection_Number) is the connection # that should
be used by the proxy-side
```

```
        * receiving threads
        */


k = 0;  //Init server_to_csproxy_ProxyBuffer counter
i = 0;  //Init
j = 0;  //Init



while (1) {

        K_actual = 0;  //Reset



        /*
        * Reset cumul_buf_Size and cumul_buf address
        */
        cumul_buf_Size = 0;
        cumul_buf = cumul_buf_Start;



        /*
        * Try to get/read K_DESIRED entries from
server_to_csproxy_ProxyBuffer.
        * K_actual is number of entries that were actually read.
        */

        // ------------------------------------------------------------------------

        for (m=1; m<=K_DESIRED; m++) {


           if (m==1) {

                //Wait until entry/packet written into next cell to be read
                while (server_to_csproxy_ProxyBuffer[k].EntryFull != 1) {
                        thr_yield();
```

```
                } //while

                K_actual = 1;

        } //if - m==1
        else { //else - m>1

                for (thryield_counter=1; thryield_counter<=THR_YIELD_MAX;
thryield_counter++) {

                        if (server_to_csproxy_ProxyBuffer[k].EntryFull != 1) {
                                #ifdef ENC_DEBUG_ON
                                  fprintf(enc_debug_fd, "Thread slept to get %d
th packet\n", m);
                                #endif
                                thr_yield();
                        } //if
                        else
                                break;

                } //for

                if (server_to_csproxy_ProxyBuffer[k].EntryFull != 1)
                        break;


                //Increment # of packets/entries read from
server_to_csproxy_ProxyBuffer
                K_actual = K_actual +1;


        } //else - m>1



        //Write packet into cumulative buffer & increment cumulative buffer
size(# bytes in c.buffer) etc.
        memcpy(cumul_buf, server_to_csproxy_ProxyBuffer[k].Data,
server_to_csproxy_ProxyBuffer[k].DataSize);
        cumul_buf=cumul_buf + server_to_csproxy_ProxyBuffer[k].DataSize;
//Move cumul_buf start so as to copy in next packet next round (if done)
```

```
                cumul_buf_Size = cumul_buf_Size +
server_to_csproxy_ProxyBuffer[k].DataSize;
                server_to_csproxy_ProxyBuffer[k].EntryFull = 0;



                //Increment server_to_csproxy_ProxyBuffer index/counter k (at END
of for loop)
                if (k < (S_TO_CS_PROXYBUFFER_SIZE-1))
                        k++;
                else
                        k = 0;

            } //for

            // ---------------------------------------------------------------------------


            /*
            * Write K_actual/K_DESIRED to enc. debug file
            */
            #ifdef ENC_DEBUG_ON
              fprintf(enc_debug_fd, "K_actual/K_DESIRED = %d/%d \n", K_actual,
K_DESIRED);
            #endif


            /*
            * Make an error check for K_actual
            */
            if (K_actual > K_DESIRED) {
                printf("ERROR(serv_to_proxy_Control_Thread): K_actual >
K_DESIRED\n\n");
                exit(1);
            }



            /*
            * Having read K_actual entries from server_to_csproxy_ProxyBuffer,
now encode these entries/packets
```

```
            * & write them to proxy_side_sendBuffer
            */
            //
================================================================
==============

            //Reset cumul_buf to its starting address
            cumul_buf = cumul_buf_Start;
            //Copy cumul_buf_Size into a temp var to use
            cumul_buf_Size_temp = cumul_buf_Size;



            //Set size of packets to be encoded and written to send-out buffer
(K_actual is # of packets)
            if (cumul_buf_Size_temp > TEMP_CUMUL_BUF_SIZE_TRESH ) {


                while ( (cumul_buf_Size_temp % K_actual) != 0 )
                        cumul_buf_Size_temp = cumul_buf_Size_temp - 1;


                #ifdef ENC_DEBUG_ON
                        fprintf(enc_debug_fd,"cumul_buf_Size=%d and
cumul_buf_Size_temp=%d\n",cumul_buf_Size,cumul_buf_Size_temp);
                #endif


                //Error check. cumul_buf_Size_temp must have reduced in size no
more than (K_actual-1)
                if ( (cumul_buf_Size - cumul_buf_Size_temp) > (K_actual-1) ) {
                        printf("ERROR(serv_to_proxy_Control_Thread): More than
(K_actual-1) bytes 'removed' from cumul_buf\n\n");
                        exit(1);
                } //if


                packet_Size = (cumul_buf_Size_temp/K_actual);
                packet_Number = K_actual;

            } //if
```

```
            else {  //cumul_buf_Size_temp<=TEMP_CUMUL_BUF_SIZE_TRESH so
encode/send out as one packet(for efficiency)


                packet_Size = cumul_buf_Size_temp;
                packet_Number = 1;
                #ifdef ENC_DEBUG_ON
                  if (K_actual>1)
                        fprintf(enc_debug_fd, "Amalgamation occurred(%d bytes)
\n", cumul_buf_Size_temp);
                #endif
            } //else - cumul_buf_Size_temp<=TEMP_CUMUL_BUF_SIZE_TRESH so
encode/send out as one packet(for efficiency)


            //Print blank line in enc. debub file
            #ifdef ENC_DEBUG_ON
                fprintf(enc_debug_fd,"\n");
            #endif


            //Error check
            if (packet_Size > SERV_RECV_SOCK_BUFFER_SIZE) {
                printf("ERROR(serv_to_proxy_Control): packet_Size >
SERV_RECV_SOCK_BUFFER_SIZE \n\n");
                exit(1);
            } //if



            //Copy source(data) packets into source_data_Packets[] array of
buffers
            for (p=0; p<packet_Number; p++) {
                source_data_Packets[p] = cumul_buf+(p*packet_Size);
            }


            //Record addres of byte after end of last packet read (needed below
for sending out remaining bytes)
            Start_Adrr_of_Remaining_Bytes = cumul_buf+(p*packet_Size);
```

```
            //Set # of packets to be produced from encoding
            Num_Encoded_Packets = N_FACTOR*packet_Number;

            /*
            * Encode the packet_Number packets of size packet_Size and copy
them to proxy_side_sendBuffer[][]
            */
            for (encode_index=0 ; encode_index<Num_Encoded_Packets;
encode_index++) {


                //Get one encoded packet (out of Num_Encoded_Packets)
                build_fec(source_data_Packets, packet_Number, packet_Size,
header_added_Packet+HEADER_SIZE, encode_index);


                /*
                * Copy encoded packet to proxy_side_sendBuffer
                */
                //Add header (k, n, Seq.#, length[2-bytes], packet# in block)
                header_added_Packet[0] = (unsigned char) 0x00;
                header_added_Packet[1] = (unsigned char)packet_Number;  //Set k
i.e. # of data/source packets
                header_added_Packet[2] = (unsigned char)Num_Encoded_Packets;
//Set n i.e. # of enc.packets
                header_added_Packet[3] = (unsigned char)Sequence_Number;
//Set sequence #. Keep it here!!!
                header_added_Packet[5] = (unsigned
char)(packet_Size+HEADER_SIZE);  //Set length (2nd byte [low])
                header_added_Packet[4] = (unsigned
char)((packet_Size+HEADER_SIZE)>>8);  //Set length (1st byte [high])
                header_added_Packet[4] = (unsigned char)( ((unsigned char)(0x80))
| header_added_Packet[4]);  //Code so that <>0

                if (header_added_Packet[5] == (unsigned char)0x00) {
                        header_added_Packet[5] = (unsigned char)( ((unsigned
char)(0x80)) | header_added_Packet[5] );
                        header_added_Packet[4] = (unsigned char)( ((unsigned
char)(0x40)) | header_added_Packet[4] );
                } //if - code length fields so that <>0
```

```
            header_added_Packet[6] = encode_index+1;
            header_added_Packet[7] = (unsigned char) 0x00;


            //Increment sequence number after using it
            if (Sequence_Number < MAX_SEQ_NUMB_VALUE)
                  Sequence_Number++;
            else
                  Sequence_Number=1;



         /*
         * Send data to server-side proxy
         */


            //Check congestion control flag HoldBack before sending data and
wait if set
            mutex_lock(&HoldBack_Lock);
            while (HoldBack != 0) {
                  mutex_unlock(&HoldBack_Lock);
                  thr_yield();
                  mutex_lock(&HoldBack_Lock);
            }
            mutex_unlock(&HoldBack_Lock);


            #ifdef USE_DELAY

            // ----------------

             if ( gettimeofday(start_time, NULL) <0 ) {
                  printf("ERROR(send_Control): gettimeofday error\n\n");
                  exit(1);
             }

             start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;
```

```
                while (1) {
                        if ( gettimeofday(curr_time, NULL) <0 ) {
                                printf("ERROR(send_Control): gettimeofday error\n\n");
                                exit(1);
                        }

                        curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;

                        if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                break;


                        //Since haven't reached timeout value yet have thread
sleep/yield
                        thr_yield();

                } //while


                // ----------------

                #endif



                //---------
                //Increment send socket identifier/index (to change sending socket
used)
                if ( send_socket_index < (Connection_Number-1) )
                        send_socket_index++;
                else
                        send_socket_index = 0;
                //---------

                // printf("CHECK(sendControl)Sending via socket
%d\n\n",send_socket_index);
```

```
            //TCP-if ( send(newsockfd[send_socket_index], buf, n , 0) < 0) {
                    if (sendto(newsockfd[send_socket_index],
header_added_Packet, packet_Size+HEADER_SIZE, 0, (struct sockaddr *)
&csp_name[send_socket_index],sizeof(struct sockaddr_in)) < 0) {
                            printf("ERROR(send/Control): While sending data to
serv-side proxy <0 \n\n");
                            exit(1);
                    } //if



            //Update j & i values
            if (j < (Connection_Number-1))
                    j++;
            else { //j wraps-around so update i also
                    j = 0;

                    if (i < (S_PROXY_SIDE_BUFFER_SIZE-1))
                            i++;
                    else
                            i=0;
            }


        } //for - encode packets & copy them to proxy_side_sendBuffer[][]



        /*
        * If bytes left over send them out - NO PASTING else can fall into
deadlock !!!
        */
        if (cumul_buf_Size_temp < cumul_buf_Size ) {

            // ********


            packet_Number = 1;
            packet_Size = cumul_buf_Size - cumul_buf_Size_temp;
```

```
                //Set # of packets to be produced from encoding
                Num_Encoded_Packets = N_FACTOR*packet_Number;


                //Copy source(data) packets into source_data_Packets[] array of
buffer
                source_data_Packets[0] = Start_Adrr_of_Remaining_Bytes;

            for (encode_index=0 ; encode_index<Num_Encoded_Packets;
encode_index++) {

                //Get one encoded packet (out of Num_Encoded_Packets)
                build_fec(source_data_Packets, packet_Number, packet_Size,
header_added_Packet+HEADER_SIZE, encode_index);


                /*
                * Copy encoded packet to proxy_side_sendBuffer
                */
                //Add header (k, n, Seq.#, length[2-bytes], packet# in block)
                header_added_Packet[0] = (unsigned char) 0x00;
                header_added_Packet[1] = (unsigned char)packet_Number;  //Set k
i.e. # of data/source packets
                header_added_Packet[2] = (unsigned char)Num_Encoded_Packets;
//Set n i.e. # of enc.packets
                header_added_Packet[3] = (unsigned char)Sequence_Number;
//Set sequence #. Keep it here!!!
                header_added_Packet[5] = (unsigned
char)(packet_Size+HEADER_SIZE);  //Set length (2nd byte [low])
                header_added_Packet[4] = (unsigned
char)((packet_Size+HEADER_SIZE)>>8);  //Set length (1st byte [high])
                header_added_Packet[4] = (unsigned char)( ((unsigned char)(0x80))
| header_added_Packet[4]);  //Code so that <>0

                if (header_added_Packet[5] == (unsigned char)0x00) {
                        header_added_Packet[5] = (unsigned char)( ((unsigned
char)(0x80)) | header_added_Packet[5] );
                        header_added_Packet[4] = (unsigned char)( ((unsigned
char)(0x40)) | header_added_Packet[4] );
                } //if - code length fields so that <>0
```

```
            header_added_Packet[6] = encode_index+1;
            header_added_Packet[7] = (unsigned char) 0x00;


            //Increment sequence number after using it
            if (Sequence_Number < MAX_SEQ_NUMB_VALUE)
                    Sequence_Number++;
            else
                    Sequence_Number=1;


          /*
          * Send data to server-side proxy
          */


            //Check congestion control flag HoldBack before sending data and
wait if set
            mutex_lock(&HoldBack_Lock);
            while (HoldBack != 0) {
                    mutex_unlock(&HoldBack_Lock);
                    thr_yield();
                    mutex_lock(&HoldBack_Lock);
            }
            mutex_unlock(&HoldBack_Lock);


            #ifdef USE_DELAY

            // ----------------

             if ( gettimeofday(start_time, NULL) <0 ) {
                    printf("ERROR(send_Control): gettimeofday error\n\n");
                    exit(1);
             }

             start_time_in_msec = (start_time->tv_sec)*1000 + (start_time-
>tv_usec)/1000;
```

```
                while (1) {
                        if ( gettimeofday(curr_time, NULL) <0 ) {
                                printf("ERROR(send_Control): gettimeofday error\n\n");
                                exit(1);
                        }

                        curr_time_in_msec = (curr_time->tv_sec)*1000 + (curr_time-
>tv_usec)/1000;

                        if ( (curr_time_in_msec - start_time_in_msec) >=
DELAY_IN_MSEC)
                                break;


                        //Since haven't reached timeout value yet have thread
sleep/yield
                        thr_yield();

                } //while


                // ----------------

                #endif


                //---------
                //Increment send socket identifier/index (to change sending socket
used)
                if ( send_socket_index < (Connection_Number-1) )
                        send_socket_index++;
                else
                        send_socket_index = 0;
                //---------



                //TCP-if ( send(newsockfd[send_socket_index], buf, n , 0) < 0) {
                        if (sendto(newsockfd[send_socket_index],
header_added_Packet, packet_Size+HEADER_SIZE, 0, (struct sockaddr *)
&csp_name[send_socket_index],sizeof(struct sockaddr_in)) < 0) {
```

```
                        printf("ERROR(send/Control): While sending data to
serv-side proxy <0 \n\n");
                        exit(1);
                    } //if

                //Update j & i values
                if (j < (Connection_Number-1))
                        j++;
                else { //j wraps-around so update i also
                        j = 0;

                        if (i < (S_PROXY_SIDE_BUFFER_SIZE-1))
                                i++;
                        else
                                i=0;
                }


            } //for - encode and write-out packets to send buffer




                // ********

            } //if
            else if (cumul_buf_Size_temp > cumul_buf_Size) { //error case
                printf("ERROR(server_to_proxy_Control): cumul_buf_Size_temp >
cumul_buf_Size\n\n");
                exit(1);
            } //else - error case



            //
==========================================================================
==============


  } //while(1)
```

```
  //
*****************************************************************************
***************

} //server_to_proxy_Control_ThreadFunc


/*
*****************************************************************************
********************* */


/*
* Thread (one instance) used to receive congestion control messages from client-
side proxy(RNR/RR)
*/
void *Cong_Msg_Recv_ThreadFunc(void *JunkPTR) {

        int  n,m;
        unsigned char  cong_msg_buf[20];  //Buffer used to receive congestion
control messages
        unsigned char  CongMessage[20];  //Congestion control message
eventually written into here
        int  Last_Message_was_RNR=2;  //Flag. Init to a neutral value


        while (1) {


           /*
           * Read-in congestion control message from client-side proxy
           */
           strcpy(CongMessage,"");  //Init string
           m=0;
           while ( (n = recv(tcp_sock, cong_msg_buf, sizeof(cong_msg_buf),
0)) >0) {

                m=m+n;
                strncat(CongMessage, cong_msg_buf, n);
                if ( CongMessage[m-1] == '\0' )
```

```
                    break; //if '\0' in string then done so exit recv/while loop

            } //while
            if (n<=0) {
                    printf("ERROR(Cong_Msg_Recv_Thread): n<=0 recv() \n\n");
                    close(tcp_sock);
                    exit(1);
            }
            /*
            * Read cong. control message received and act appropriately
            */
            if (strcmp(CongMessage, "RNR") == 0) {
                    if (Last_Message_was_RNR == 1)
                            printf("ERROR/WARNING(Cong_Msg_Recv_Thread):
Multiple RNR messages received in a row\n\n");

                    Last_Message_was_RNR = 1;  //Set flag


                    //Print out that "RNR" message received
                    #ifdef FLOW_CNTRL_DEBUG_PRINT
                     printf("STATUS(Cong_Msg_Recv_Thread): RNR message
received\n\n");
                    #endif


                    mutex_lock(&HoldBack_Lock);
                      HoldBack = 1;
                    mutex_unlock(&HoldBack_Lock);

            }
            else if (strcmp(CongMessage, "RR") == 0) {
                    if (Last_Message_was_RNR == 0)
                            printf("ERROR/WARNING(Cong_Msg_Recv_Thread):
Multiple RR messages received in a row\n\n");

                    Last_Message_was_RNR = 0;  //Set flag


                    //Print out that "RR" message received
                    #ifdef FLOW_CNTRL_DEBUG_PRINT
```

```
                        printf("STATUS(Cong_Msg_Recv_Thread): RR message
received\n\n");
                    #endif

                    mutex_lock(&HoldBack_Lock);
                      HoldBack = 0;
                    mutex_unlock(&HoldBack_Lock);
                }
                else { //Error case
                    printf("ERROR(Cong_Msg_Recv_Thread): CongMessage neither
RNR nor RR \n\n");
                    close(tcp_sock);
                    exit(1);
                } //else


            } //while(1)

} //Cong_Msg_Recv_ThreadFunc
```