AFIT/GCE/ENG/99M-02

Performance Analysis of TCP Enhancements in Satellite Data Networks

THESIS

Ren H. Broyles, Captain, USAF

AFIT/GCE/ENG/99M-02

Approved for Public Release – Distribution Unlimited

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March, 1999	Master's Thesis

4. TITLE AND SUBTITLE
PERFORMANCE ANALYSIS OF TCP ENHANCEMENTS IN SATELLITE DATA NETWORKS

5. FUNDING NUMBERS

6. AUTHOR(S)
Ren H. Broyles, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P. Street
Wright-Patterson AFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/99M-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Lewis Research Center - Satellite Networks and Architectures Branch
Thomas Vondeak
Mail Stop 54-2
21000 Brookpark Rd
Cleveland, OH 44135-3127 Comm: (216) 433-3277

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Richard A. Raines, Major, USAF
DSN: 785-3636 ext. 4715
richard.raines@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research examines two proposed enhancements to the well-known Transport Control Protocol (TCP) in the presence of noisy communication links. The Multiple Pipes protocol is an application-level adaptation of the standard TCP protocol, where several TCP links cooperate to transfer data. The Space Communication Protocol Standard - Transport Protocol (SCPS-TP) modifies TCP to optimize performance in a satellite environment. While SCPS-TP has inherent advantages that allow it to deliver data more rapidly than Multiple Pipes, the protocol, when optimized for operation in a high-error environment, is not compatible with legacy TCP systems, and requires changes to the TCP specification. This investigation determines the level of improvement offered by SCPS-TP's Corruption Mode, which will help determine if migration to the protocol is appropriate in different environments. As the percentage of corrupted packets approaches 5%, Multiple Pipes can take over five times longer than SCPS-TP to deliver data. At high error rates, SCPS-TP's advantage is primarily caused by Multiple Pipes' use of congestion control algorithms. The lack of congestion control, however, limits the systems in which SCPS-TP can be effectively used.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Transmission Control Protocol (TCP), Multiple TCP Connections, Space Communication Protocol Standard -Transport Protocol, Satellite Networks, High Error Rate TCP Connections	140
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

AFIT/GCE/ENG/99M-02

Performance Analysis of TCP Enhancements in Satellite Data Networks

THESIS

Presented to the faculty of the Graduate School of Engineering

Of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Ren H. Broyles, B.S.E.E, M.B.A.

Captain, USAF

March, 1999

Richard A. Raines, Ph.D., Major, USAF
Committee Chairman

Michael A. Temple, Ph.D., Major, USAF
Committee Member

AFIT/GCE/ENG/99M-02

Performance Analysis of TCP Enhancements in Satellite Data Networks

THESIS

Presented to the faculty of the Graduate School of Engineering

Of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Ren H. Broyles, B.S.E.E, M.B.A.

Captain, USAF

March, 1999

Acknowledgements

There are many people I owe a debt of gratitude for their help in completing this thesis. Foremost is my thesis advisor, Major Richard A. Raines, who was very patient and a source of both technical guidance and personal encouragement, as well as the other member of my committee Major Michael A. Temple. I'd like to thank several classmates, particularly Captain Steve Pratt and Captain Doug Loomsdalen who helped me out in many ways. My friends from church are fortunately too numerous to list, and they really helped keep me sane (although some might disagree!). Great thanks are due to my family (Dad, Mom, Cam, Sylvia, and Kaitlyn) – you may have been physically far away but were always close to me. Finally, there is of course my Father in Heaven, with whom anything is possible.

Ren H. Broyles

Table of Contents

ACKNOWLEDGEMENTS ..... **I**

TABLE OF CONTENTS ..... **II**

LIST OF FIGURES ..... **VI**

LIST OF TABLES ..... **VIII**

ABSTRACT ..... **IX**

1 INTRODUCTION **1**

1.1 BACKGROUND 1

1.2 THE PROBLEM 1

1.3 SCOPE/APPROACH 2

1.4 THESIS PRESENTATION 2

2 LITERATURE SEARCH **4**

2.1 INTRODUCTION 4

2.2 SATELLITE NETWORKS 4

2.3 TCP/IP 5

2.3.1 Limitations of TCP/IP *7*

2.3.2 TCP/IP Enhancements *11*

2.4 SUMMARY 18

3 METHODOLOGY **19**

3.1 INTRODUCTION 19

3.2 EXPERIMENT DESIGN 19

3.2.1 Choice of Method *19*

3.2.2 Models *20*

3.3 OPERATIONAL ASSUMPTIONS 24

3.3.1 Packet Size *24*

3.3.2 Traffic Data Rates *24*

3.3.3 SCPS-TP Error Assumption *25*

3.3.4 Window Size *25*

3.3.5 File Size *26*

3.3.6 Delays *26*

3.4 MODELING ERRORS/PACKET LOSS 26

3.4.1 Errors *27*

3.4.2 Alternate Error Approach *31*

3.4.3 Congestion Loss *32*

3.4.4 Timeouts *33*

3.5 DATA GATHERING 35

3.6 MODEL VERIFICATION/VALIDATION 36

3.7 CONCLUSION 37

4 RESULTS AND ANALYSIS 38

4.1 INTRODUCTION 38

4.2 COMPARISON AT G.821 ERROR RATES 38

4.3 COMPARISON AT HIGH ERROR RATES 43

4.4 EFFECTIVE JAMMING TECHNIQUES 49

4.4.1 Error Burst Length *51*

4.4.2 Burst Timing *53*

4.4.3 Possible Counter-Jamming/Performance Improvement Techniques *55*

4.5 SCPS-TP SUMMARY 59

4.6 MPP SUMMARY 61

4.7 CONCLUSION 61

5 CONCLUSIONS 63

5.1 PROTOCOL ADAPTABILITY TO NOISY ENVIRONMENTS 63

5.2 SIMULATION PROBLEMS ENCOUNTERED 63

5.3 RECOMMENDATIONS FOR FUTURE WORK 64

5.4 CONCLUSION 65

APPENDIX A – DETAILED SIMULATION DEFINITION 66

A.1 DESIGNER BDE 66

A.2 MPP 66

A.2.1 Pipes Top Level 66

A.2.2 Pipe Traffic Generator 70

A.2.3 Pipe ACK Generator 72

A.2.4 Pipe ACK Resolution 74

A.2.5 New Segment Transmitted 77

A.2.6 Get Local Sliding Window Variables 78

A.2.8 Error (Timeout) Source 80

A.2.9 Insert Error/Timeout 81

A.2.10 Propagation 83

A.2.11 Uplink/Downlink 84

A.2.12 Thesis Packet – Pipes 84

A.3 SCPS-TP BDE 85

A.3.1 SCPS Top Level 85

APPENDIX B – DESIGNER PRIMITIVES DESCRIPTIONS 90

B.1 DESIGNER PRIMITIVES 90

B.2 MPP PRIMITIVES 90

B.2.1 Assign Packet ID Number Primitive *90*

B.2.2 Assign ACK Number Primitive *91*

B.2.3 Resolve ACK Primitive *93*

B.3 SCPS-TP PRIMITIVES 99

B.3.1 Assign SCPS Packet ID Number Primitive *99*

B.3.2 Generate SNACK Number Primitive *100*

B.3.3 Resolve SNACK Primitive *101*

APPENDIX C – DESIGNER PRIMITIVES CODE 103

BIBLIOGRAPHY 125

VITA 127

List of Figures

Figure 1 - MPP Top Level Model 21

Figure 2 - SCPS-TP Top Level Model 22

Figure 3 - LEO Normalized Performance (.0003, .002 SES) 39

Figure 4 - GEO Normalized Performance (.0003, .002 SES) 41

Figure 5 - LEO Normalized Performance (.01, .025, .05 SES) 45

Figure 6 - GEO Normalized Performance (.01, .025, .05 SES) – Average Case 47

Figure 7 - GEO Normalized Performance (.01, .025, .05 SES) – Worst Case 47

Figure 8 - Effect of Burst Size on Transmission Time 51

Figure 9 - Single Responsible Pipe Retransmission Delay 58

Figure 10 - Pipes Top Level Schematic 67

Figure 11 - Pipe Traffic Generator Schematic 70

Figure 12 - Pipe ACK Generator Schematic 73

Figure 13 - Pipe ACK Resolution Schematic 75

Figure 14 - New Segment Transmitted Schematic 77

Figure 15 - Get Local Sliding Window Variables Schematic 78

Figure 16 - Set Sliding Window Vector Variables Schematic 79

Figure 17 - Error Source Schematic 80

Figure 18 - Insert Error Schematic 82

Figure 19 - Propagation Schematic 83

Figure 20 - Downlink (Left) and Uplink (Right) Schematics 84

Figure 21 - SCPS Top Level Schematic 86

Figure 22 - SCPS Traffic Generator Schematic 88

Figure 23 - SCPS SNACK Generator Schematic 88

Figure 24 - SCPS SNACK Resolution Schematic 89

Figure 25 - Assign Packet ID Number Primitive (MPP) 91

Figure 26 - Assign ACK Number Primitive (MPP) 92

Figure 27 - Resolve ACK Primitive - Periodic Timeout (MPP) 94

Figure 28 - Resolve ACK Primitive - Packet Arrival (MPP) 95

Figure 29 - Slow Start Method 96

Figure 30 - Congestion Avoidance Method 97

Figure 31 - Fast Retransmit Method 98

Figure 32 - Fast Recovery Method 98

Figure 33 - Assign SCPS Packet ID Number Primitive 100

Figure 34 - Generate SNACK Number Primitive 101

Figure 35 - SNACK Resolution Primitive 102

List of Tables

Table 1 - G.821 Error Performance Specification 28

Table 2 - SES Errors 30

Table 3 - Error Burst Sizes 30

Table 4 - Error Count (Severely Degraded Link) 31

Table 5 - LEO Transfer Times (.0003, .002 SES) in seconds 39

Table 6 - GEO Transfer Times (.0003, .002 SES) in seconds 40

Table 7 - SCPS-TP Advantage Due to Initial Sliding Window (Low SES) 42

Table 8 - LEO Transfer Times (.01, .025, .05 SES) in seconds 44

Table 9 - GEO Transfer Times (.01, .025, .05 SES) in seconds 46

Table 10 - SCPS-TP Advantage Due to Initial Sliding Window (High SES) 49

Table 11 - Early vs. Late Error Insertion 54

Table 12 - Pipes Top Level Variables 69

Table 13 - Pipe Traffic Generator Variables 72

Table 14 - Pipe ACK Generator Variables 74

Table 15 - Pipe ACK Resolution Variables 76

Table 16 - Error Source Variables 81

Table 17 - Thesis Packet - Pipes Fields 85

Abstract

Like the rest of society, the armed forces are increasingly reliant on technology to accomplish their missions. Often, however, technology is largely useless without the appropriate information to direct the machines when, where, and how to act. Rapid and reliable information delivery, therefore, is critically important to modern combat and training operations.

This thesis examines two proposed enhancements to the well-known Transport Control Protocol (TCP) in the presence of noisy communication links. The Multiple Pipes Protocol (MPP) is an application-level adaptation of the standard TCP protocol where several TCP links cooperate to transfer data. The Space Communication Protocol Standard – Transport Protocol (SCPS-TP) modifies TCP to optimize performance in a satellite environment. While SCPS-TP has inherent advantages that allow it to deliver data more rapidly than the MPP, the protocol is largely incompatible with legacy TCP systems and requires changes to the TCP specification. This research attempts to determine the level of improvement offered by SCPS-TP to help determine if migration to the protocol is appropriate.

It was determined that as the percent of corrupted packets reaches 5%, the MPP can take over five times longer than SCPS-TP to deliver data. At high error rates (1% corrupted packets and above), SCPS-TP's advantage is primarily caused by the MPP's use of congestion control algorithms. The lack of congestion control, however, limits the systems in which SCPS-TP can be effectively used.

1 Introduction

1.1 Background

In today's military, information superiority is a necessity. Getting the right information to the right place in an expeditious manner is in many ways as important to successful combat (or training) operations as getting the aircraft to the flight line. During the Gulf War, the demand for data transport led to saturation of both military satellites and leased commercial links, forcing critical information to be airlifted into the theater of operations. As the Air Force increasingly finds itself deployed to support operational missions, the importance of wireless communication in both nominal and hostile environments will constantly increase.

The importance of *Transport Control Protocol (TCP)* in modern civilian communication networks continuously drives attempts to improve the protocol's efficiency. As the trend of replacing military-only technical solutions with acceptable commercial products continues, it is necessary for the Air Force to evaluate how well TCP adapts to the military environment.

1.2 The Problem

When used in space communication, TCP experiences unique problems not normally encountered in terrestrial networks. Packet loss due to link outage and noise

corruption either has no terrestrial counterpart or is much more likely to occur in satellite networks. In addition, an enemy will undoubtedly attempt to prevent information flow during combat operations.

This thesis compares the effectiveness of two proposed TCP enhancements (*Multiple Pipes Protocol (MPP)* and *Space Communication Protocol Standard – Transport Protocol (SCPS-TP)*) in noisy environments. While most military communication satellites use *Geosynchronous Earth Orbit (GEO)* systems like Defense Satellite Communication System, commercial *Low Earth Orbit (LEO)* systems like Iridium and Teledesic may be used in the future.

1.3 Scope/Approach

Using an existing network simulation tool (BoNES Designer), simulations of both the MPP and SCPS-TP protocols are developed. Various file sizes are transported between nodes using both GEO satellite and LEO satellite networks. During transport, files are subjected to various corruption levels. The effectiveness and relative usefulness of the two protocols is measured by recording and comparing execution times. To gain confidence in simulation results, several test runs are accomplished for each file size using various error rates.

1.4 Thesis Presentation

This thesis is divided into five chapters. Chapter II presents the challenges facing TCP when operating in a satellite environment and discusses previous efforts to enhance TCP performance. Chapter III discusses the methodology used to develop the MPP and

SCPS-TP simulations. Chapter IV presents, discusses and analyzes simulation results. Finally, Chapter V summarizes the thesis effort and proposes future research.

2 LITERATURE SEARCH

2.1 Introduction

Wireless links are becoming an increasingly important part of the *National Information Infrastructure (NII).* As these links become more common, protocols that traditionally operated in a terrestrial environment are being forced to adapt to the wireless environment. Not surprisingly, the differences between these two environments can significantly impact protocol performance. As new satellite systems like Teledesic are deployed, point-to-point networking exclusively over satellite links will become much more common. This chapter reviews some of the differences between the wired and wireless worlds, examining potential techniques to improve TCP performance.

2.2 Satellite Networks

In the past, most satellite constellations have used vehicles positioned at GEO altitudes (approximately 35,786 km). The primary advantage of this orbit is that worldwide coverage (to approximately ±70° latitude) can be accomplished using only 3 satellites spaced 120° apart. In addition, satellite movement is minimal, so tracking is relatively simple. Finally, satellite handovers occur infrequently due to the large satellite footprint.

Recently, however, new technologies have enhanced the practicality of LEO constellations at altitudes between 500 and 2000 km. At lower altitudes, propagation delay, one of the largest drawbacks of GEO-type constellations, is significantly reduced

(~560ms for GEO to ~4.5ms at Teledesic's 1350 km altitude). Several challenges, however, result from using a LEO constellation. First, satellite footprints are much smaller at lower altitudes, so more satellites are required to provide worldwide coverage. Second, LEO satellites enter and leave a ground station's in-view envelope every 10-20 minutes, so frequent satellite handovers are necessary. Third, although the power required to satisfy radio link margins is reduced, tracking becomes more difficult. Finally, the constantly changing geometry of the constellation produces constant changes in the LEO network topology.

2.3 TCP/IP

Transport Control Protocol, when combined with *Internet Protocol (IP)*, are the most widely used protocols for the Transport (TCP) and Network (IP) layers of the *International Standards Organization (ISO) Open Systems Interconnect (OSI)* model in many network topologies. This is particularly in *Wide Area Networks (WAN)*, where TCP/IP has become the de-facto standard. IP is an unreliable and connectionless protocol used primarily to interconnect a very large number of network nodes worldwide [SaA94]. By definition, IP provides no guarantee that a *datagram* (IP's basic transfer unit) will be delivered. IP does, however, guarantee that if a datagram is not received within a certain "time" period, it will never be delivered. To enforce this guarantee, a Time-to-Live field is included in each datagram's IP header. This field does not represent actual time units – rather it indicates the number of router hops a datagram is allowed to take. As the datagram travels toward its destination, each router decrements the field by one. If the Time-to-Live field reads zero, a router discards the datagram. Since IP is connectionless,

each datagram can be routed along different paths as it travels to the final destination [Com95].

To ensure data is reliably delivered, TCP is implemented on top of IP. TCP is a connection-oriented protocol that ensures that all datagrams are successfully transmitted and properly re-assembled. TCP takes a stream of octets (8 binary digits) from the OSI Session layer, combines groups of octets into a *segment* (TCP's basic transfer unit) and sends the segments to Network layer (IP) for delivery. Usually, each segment is encapsulated into a single IP datagram. Segments can have variable lengths, up to a maximum length which is mutually agreed upon when a TCP connection is established [Com95].

TCP uses a positive acknowledgement system to ensure segments are reliably delivered to a receiving node, utilizing a "*sliding window*" *Automatic Repeat Request* (ARQ) scheme – also known as *Go Back N.* The sliding window operates at the octet level rather than the segment level. When a TCP connection is established, each node chooses a random 32-bit number as an initial sequence number and informs the other node of the choice (each node has a different initial sequence number). Each subsequent octet transmitted by the node is assigned an incrementally increasing sequence number. For each received packet, the node sends back an *Acknowledgment* (ACK) signal containing the highest sequence number with the property that all octets with lower sequence numbers have been correctly received. Three pointers define the sliding window. The "left" pointer relates to the highest octet sequence number for which the node has received an ACK message. The "right" pointer limits to how many octets can

be transmitted without additional ACK signals. The "center" pointer is the octet that is currently being transmitted. When a node receives an ACK signal, it moves the left pointer to the correct sequence number and moves the right pointer to allow transmission of additional octets. ACK signals are cumulative, meaning an ACK's sequence number indicates all previous octets have been successfully received. Consequently, ACK signals are not required for each octet.

TCP has two mechanisms for determining if segments have been lost. The sending node keeps a timer on each segment. If the timer reaches *Retransmission Timeout* (RTO) before receiving an appropriate ACK signal to indicate successful reception, the octet is automatically retransmitted. In addition, TCP uses the reception of duplicate ACK signals as an indication that segments have been lost.

The sliding window size varies with time. To ensure that buffers are not overrun, each segment contains a *window advertisement*; a receiving node's advertisement is used as an upper bound on the sender's sliding window. During transmission, the actual size of the window is varied to maximize throughput.

2.3.1 Limitations of TCP/IP

One of the primary problems arising from using TCP in a satellite environment involves the sliding window scheme. Since TCP stops transmitting when the number of outstanding packets reaches the current window size, performance will be limited whenever TCP is used in systems with a high *Delay-Bandwidth Product (DBWP)* [LiS95]. In these situations, the transmission medium can spend the majority of its time idle waiting for acknowledgements. For example, the minimum time required to send a

packet over a GEO satellite link and receive an acknowledgement of successful receipt is approximately 0.5 sec. When combined with the maximum default TCP window size of Microsoft Windows 95 and Windows NT (64 Kbytes), the maximum throughput for most desktop systems over a single unmodified TCP connection over a GEO satellite link can be found from Equation 1.

$$Throughput\,(bps) = \frac{Buffer\ Size\,(bytes)}{Round\ Trip\ Delay\,(sec)} \approx \frac{64\ KBytes}{0.5\ sec} \approx 128\ KBytes/s \qquad (1)$$

If a user bought a 2 Mbps (a common speed for the Teledesic network) GEO link, then the user's link would be idle over 50% of the time while waiting for ACKs from the receiving node.

TCP contains protocols to keep networks from suffering congestive collapse. During congestive collapse, most data segments and ACKs injected into the system are discarded by intermediate routers due to exceeding the Time-to-Live parameter or router buffer overflow. This causes retransmission of lost segments, which further aggravates the situation, effectively causing a snowball effect. Obviously, little useful communication is occurring in this state. The four algorithms used to manage network congestion are Slow Start [JaK88], Congestion Avoidance [JaK88], Fast Retransmit [Jac90], and Fast Recovery [Jac90] (see RFC2001 for an overview of all four algorithms). The algorithms, however, can negatively impact performance of networks using TCP over satellite links. The negative impacts are caused by modifications each algorithm makes to the sliding window.

The Slow Start and Congestion Avoidance algorithm work together to increase window size (which increases data rate) while trying to avoid overwhelming the

intermediate routers and causing segment loss due to buffer overflow. The Slow Start algorithm uses two variables; the first is called the *Congestion Window* (initially set to 1), and the second is called the *Slow-Start Threshold* – initially set to the number of segments in the receiver's advertised window size[1]. TCP initially sets the sliding window to MIN (congestion window, slow-start threshold). With each arriving ACK, the congestion window size is incremented. Since the effect is cumulative, the size of the congestion window increases exponentially until it either reaches the slow start threshold or segment retransmission occurs due to RTO. When RTO occurs, the slow start threshold is reset to ½ the current congestion window size, and the congestion window size is reset to 1 [All97, Com95]. Slow Start then increases the window size as before until the new slow start threshold is reached or RTO again occurs.

The Congestion Avoidance algorithm works in concert with the Slow Start Algorithm. When the congestion window size, after being restricted due to congestion, becomes greater than (or equal to) the current slow start threshold, the Congestion Avoidance algorithm assumes control over window size increases. Congestion Avoidance allows the window to increase by one only if all segments in the window received ACKs (assuming the new window is not greater than the receiver's advertised window). As a result, the window size increases by a maximum of one for each round-trip time [All97, Com95].

The Fast Retransmit and Fast Recovery algorithms help reduce retransmissions of segments perceived as lost by the sending node. In the TCP standard, segments are

---

[1] Window sizes are actually measured in bytes, but understanding the process is simplified when sizes are expressed in terms of segments

assumed lost and retransmitted when RTO occurs. If the proper ACK is in transit or the segment was successfully received and is waiting in the receiving node's buffer, however, needless retransmissions will occur. Rather than wait for a timeout, the Fast Retransmit algorithm assumes that duplicate ACKs (usually three) indicate that a segment has been lost. The proper segment is immediately retransmitted without waiting for RTO. While duplicate ACKs indicate that congestion is occurring in the network, the fact that the ACKs got through the network indicates that a less drastic approach than Slow Start is appropriate. In this situation, the Fast Recovery algorithm is activated. Fast Recovery, like Congestion Avoidance and Slow Start, modifies the sliding window size. Like Slow Start, Fast Recovery sets the slow start threshold to ½ the current congestion window size. Instead of reducing the congestion window to one, however, the Fast Recovery algorithm initially sets the congestion widow to the new slow start threshold. The congestion window is artificially increased by the number of duplicate ACKs (usually three) on the premise that duplicate ACKs indicate a segment was received and is consequently not in the network. If the congestion window allows, additional segments can be transmitted. Once a non-duplicate ACK is received, the artificial inflation created by the duplicate ACKs is removed until the congestion window reaches the size of the slow start threshold. At this point, the Congestion Avoidance algorithm is implemented.

It is clear that retransmission caused by RTO duplicate ACK arrival have different effects on transmission rate. In the RTO scenario, no inference is made about the network's state. TCP initiates Slow Start, sets the congestion window to one, increases

the congestion window exponentially until it reaches ½ the previous window size, then initiates Congestion Avoidance. In Fast Retransmit, the arrival of duplicate ACKs indicates that data is still moving in the network. The congestion window is set to approximately ½ its previous size and TCP enters congestion avoidance much sooner. The effect on window size (and hence transmission rate) is clearly much less dramatic with Fast Retransmit than RTO [All97, Hay97].

2.3.2 TCP/IP Enhancements

Recognizing that TCP (and other traditional Internet protocols) introduce problems when transitioned to the satellite arena, the *Internet Engineering Task Force (IETF)* formed the *TCP Over Satellite (TCPSAT)* Working Group to look at different approaches to improve TCP performance over satellite links. Some of the techniques proposed by this Working Group (and other Working Groups that examine TCP performance in different situations) have been adopted as part of the TCP standard. Other recommendations are still in the standards process or experimental stages. Three of the most important developments are large windows, SCPS-TP, and MPP.

2.3.2.1 Large Windows

As networks speed increased, TCP performance degrades as the DBWP increases. Under the original TCP standard, a 16-bit field in the TCP header is used to indicate the receive-window size. Consequently, the largest window is 2^{16}, limiting outstanding data to 64 Kbytes. As stated previously, this limits a single unmodified TCP connection over a GEO satellite link to approximately 128 Kbytes/sec. Request for Comments 1323

(RFC1323) proposed using a Window Scale Extension, which allows a TCP window field of 32 bits to be carried in the TCP header's 16-bit window field. The scaling factor is only sent in the *Synchronize (SYN)* segments sent when initializing a TCP connection, and therefore can not be modified without terminating a connection and establishing a new one. To ensure compatibility with versions of TCP not capable of window scaling, both sender and receiver must recognize the Window Scale option – if either side fails to recognize this portion of the SYN segment and respond appropriately, a standard 16-bit window size is used. While RFC1323 has not been officially required by the IETF in TCP implementations, most new versions of TCP, both Unix-based and Microsoft Windows-based, do contain large-window support [Mah97].

The Window Scale option contains a variable called *shift.cnt*; this variable represents how many bits the true size of the receiving window will be right-shifted to fit into the 16-bit TCP window field. The shift amount is limited to 14, making the maximum window size 2^{16+14} or 2^{30} (1 Gbyte). Each site maintains the window variables as 32-bit quantities for local use (i.e. congestion window size).

2.3.2.2 Space Communication Protocol Standard – Transport Protocol

The authors of [DuM96] proposed SCPS-TP, which consists of several extensions to TCP, which improves TCP performance in the space environment. These extensions included changes to both TCP implementation (with no effect on interoperability) and changes to the TCP specification.

Besides the sliding window problem, one of the largest factors limiting TCP performance in the space environment is the protocol's assumption that all packet losses

result from network congestion. When segment loss occurs because of bit-errors due to link noise or intermittent connectivity (both of which are much more common in space networks than traditional terrestrial networks), invocation of the congestion control protocols discussed in Section 2.3.1 result in reduced throughput without providing any benefit. In addition, if connectivity is interrupted, the loss of ACK message flow can result in low throughput and aborted connections [DuM96]. Consequently, optimizing TCP in the space environment requires that the cause of packet loss be established. Once the root cause is determined, the response varies depending on whether the cause is congestion, corruption, or link outage.

While TCP automatically assumes that packet loss was caused by congestion, SCPS-TP allows the default cause of packet loss to be set by the network administrator to any of the three mentioned causes. In addition, a transmitting node can receive information regarding the cause of packet loss by messages from receiving stations (or intermediate network elements) and react appropriately. If congestion is set as the underlying cause of packet loss, SCPS-TP uses the Vegas variant of the Slow Start algorithm to modify the sliding window. In the Vegas variant, exponential increases in the sliding window size are only allowed every *Round Trip Time (RTT).* Between these exponential increases, the algorithm compared expected data throughput with actual data throughput. The switch from Slow Start to Congestion Avoidance occurs when the actual throughput rate falls below the expected throughput rate by a threshold amount. By doing this, Vegas hopes to avoid the loss of segments that normally trigger changes in window size. The change does not effect the standard since a transmitting node is totally

responsible for regulating the data transmission rates. The method of response is unchanged, so a receiving node's protocol is unchanged.

SCPS-TP provides the ability to avoid low throughput and possible aborted connections associated with short-term interruptions in a satellite link. When a node senses a lost link (either through explicit messages or loss of carrier lock) the protocol enters Persistent Mode. In this mode, a node suspends all outstanding segment's RTO clocks and sends out periodic probes until the connection has been re-established. Since timers cannot expire until after the connection is re-established, window size is not unnecessarily restricted. When communication is possible, the node resumes transmission at the same rate as before the interruption.

If packet loss is due to corruption, congestion control is an incorrect response. If the underlying cause of packet loss is assumed to be corruption (hereafter referred to as "Corruption Mode"), SCPS-TP stops using a sliding window altogether. Packets are sent using an open-loop token bucket approach. The transmission rate is adjusted to ensure link capacity is not overrun. This rate is stored in a globally accessible routing structure so any node in a link can make adjustments [DuM96].

SCPS-TP allows hosts to temporarily change the underlying assumption of the cause of packet loss. The most common switch is from either link loss or congestion to corruption. The link layer sends the receiving station information regarding the number of packets that could be de-multiplexed but are corrupted. Once this number's moving average exceeds a certain threshold, the receiver sends an Internet Control Message Protocol (ICMP) message to the transmitting node to inform it of the corruption. The

transmitting node responds to the ICMP message by changing the underlying loss assumption to corruption. The transmitting node stays in this mode for (2*RTT) unless an additional control message arrives. Once the timer expires or a SNACK message without the corruption flag is received, the protocol switches back to the default responses – the congestion window is not modified as a result of this event [DuM96].

SCPS-TP replaces the Fast-Retransmit/Fast Recovery algorithms with a *Selected Negative Acknowledgement (SNACK)* signal. A SNACK signal is sent by a receiving node to initiate retransmission of corrupted packets. SNACK messages have the advantage over TCP's traditional ACK signals in that they can identify multiple holes in the receiver's buffer. ACK signals can identify at most one hole in the buffer, and requires a RTT to signal each additional hole in the out-of-order sequence. By using the SNACK signal to inform the sending node of multiple holes, retransmission time is significantly reduced [DuM96]. The SNACK protocol is a modification of IETF RFC2018 *Selective Acknowledgement (SACK)* TCP option, which is currently deployed in several systems [Mah97], and the Negative Acknowledgement (NACK) proposal [DuM96].

The SCPS-TP approach has the disadvantage of requiring changes in the TCP specification. Consequently, all sites using a satellite network capable of SCPS-TP would have to be running the protocol in order to leverage the full benefit of the protocol. To get around the compatibility problem, the use of non-standard SCPS-TP options is negotiated when a connection is established. If a destination node does not recognize the "SCPS-TP" option embedded in the SYN segment, the protocol will act like standard

TCP and not take full advantage of SCPS-TP's potential performance improvements [DuM96].

2.3.2.3 Multiple TCP Pipes

An alternative approach in [AlK96] is the use of multiple TCP connections to increase throughput. Unlike SCPS-TP, which requires that the each endpoint be using a modified TCP protocol, the multiple connection approach leaves the TCP specification untouched. Applications, as opposed to the TCP protocol, are responsible for establishing and managing multiple independent TCP connections, dividing files between the different TCP channels, and reassembling the files.

In their paper, the FTP protocol was chosen as the application program to test the multiple TCP connection approach. A new version of FTP, called *Extended FTP (XFTP)*, was created to support multiple connections. To ensure backward compatibility, two new messages were added to FTP's message primitives. When establishing a connection, a client site sends one of the new FTP primitive commands to determine if the server is capable of servicing multiple TCP connections. If the server is properly configured, it responds with the other new FTP primitive which indicates the maximum number of concurrent TCP connections the server can support. The client uses this number (as well as its own limit of concurrent TCP connections) and establishes the multiple TCP pipes. If a site does not recognize the new commands, XFTP only opens a single TCP connection. Files are divided into many small (8k) records and arbitrarily assigned to a TCP connection that has available resources. Each TCP link may therefore transfer different amounts of data, and congestion of one connection does not necessarily

affect other connections. In this manner, all pipes are kept as full as possible. Sequence numbers are appended to each segment to ensure proper file recovery.

Surprisingly, throughput does not linearly increase as the number of connections is increased [AlK96]. While the overhead associated with multiple connections was found to be negligible, other aspects of standard TCP combine to limit the benefit of multiple TCP connections. As the number of connections increases, the "effective" window size (window size x number of connections) allows increasing numbers of segments to be injected into the network. Intermediate routers can become saturated, which would cause segments to be discarded. When the server realizes these segments are lost, it immediately executes the congestion control mechanisms described in Section 2.3.1. On the system used in [AlK96], buffer overflow occurred between 8 and 10 TCP connections, and throughput was maximized with 6 to 8 connections.

An effort to control congestion using application-level congestion control algorithms (and thus avoid TCP's congestion control mechanisms) was attempted. The application algorithm opens and closes connections based on bonding limits on RTT.

Using a large number of connections creates a large "effective" window has an additional advantage over using a scaling factor to create a single large window of the same size. As previously noted, a wireless link traditionally has a significantly higher error rate than networks using wired transmission mediums. With a single large window, the likelihood of segment loss (corruption or congestion) is higher than if multiple smaller windows are used. In a single window, the standard congestion control algorithms will reduce the entire window size by ½. If multiple connections are used, not

only is the decrease in window size smaller (since the individual windows are smaller), but the decrease is imposed on only one of the TCP connections. As is readily apparent, the impact of a segment loss is much smaller when there are multiple TCP connections than if there is a single connection.

2.4 Summary

The field of incorporating satellite links into networks has become an area of intense research due to the increasing role satellites play in today's communications. One need only recall the impact to paging systems due to the loss of a single satellite to see that the availability and reliability of satellite links has become critical parts of the NII.

When running protocols designed for terrestrial networks over satellite links, new performance problems occur. Research into enhancing TCP performance in noisy environments has led to the development of methods to overcome the problems associated with TCP's sliding window. Some of these methods are currently in the TCP standard, while others are in the standards process or experimental stages.

3 METHODOLOGY

3.1 Introduction

TCP will be used for many years to come as one of the primary protocols for network communication. Consequently, any modifications to the protocol having the potential to significantly increase TCP's effectiveness require serious evaluation. As wireless communication becomes more common in communication networks, the unique properties of wireless links are violating some of the basic assumptions that formed TCP's foundation. In particular, the portion of packets lost due to noise when compared to the number of packets lost due to congestion is much higher in wireless links than in terrestrial links. Two different methods are proposed to increase TCP throughput: the MPP approach and the SCPS-TP approach. MPP does not change the TCP specification, which yields certain advantages. On the other hand, SCPS-TP, while requiring changes in the TCP specification, is designed around the wireless environment. This thesis compares both approaches and how they perform in noisy environments.

3.2 Experiment Design

3.2.1 Choice of Method

There are three methods for gaining insight into the network performance. The first option is to directly measure performance – this method, of course, requires access to an existing system. The second option is to create a mathematical model of the system and

analytically solve sets of simultaneous equations. The final method is to create a simulation model of the prospective system and run experiments against the simulation.

Simulation is the method of choice for this research. Since access to a satellite channel and the equipment necessary to create and measure performance in various types of TCP channels is not possible, the direct measurement approach is not utilized. While creating a mathematical model is possible, the scope of this thesis makes such an approach intractable. Even making simplifying assumptions, queuing theory states that a network involving N nodes and K packets will require $\left(\frac{N+K-1}{N-1}\right)$ simultaneous equations [Jan96]. Simulation, therefore, is the preferred approach.

Simulations were built using BoNES Designer, published by Cadence Software. This package allows users to build top-down block-oriented network models. Of critical importance is Designer's ability to incorporate user-defined code (called "primitive" modules) into the design. These special modules allow Designer users to create C++ code, which will perform functions not easily implemented using Designer's native constructs.[2]

3.2.2 Models

To keep the different protocols separate, two models are used – one for each approach (MPP and SCPS-TP). While it is not appropriate to review the models in great detail, it is appropriate to present an overview. Details of the model's construction are

[2] Modules and variables created by the author for this thesis are italicized, while Designer modules and variables are quoted

found in Appendix A, and details of the Designer primitives created for this thesis are found in Appendix B.

The first model (Figure 1) simulates the MPP approach to improving TCP performance, while Figure 2 shows the SCPS-TP model.

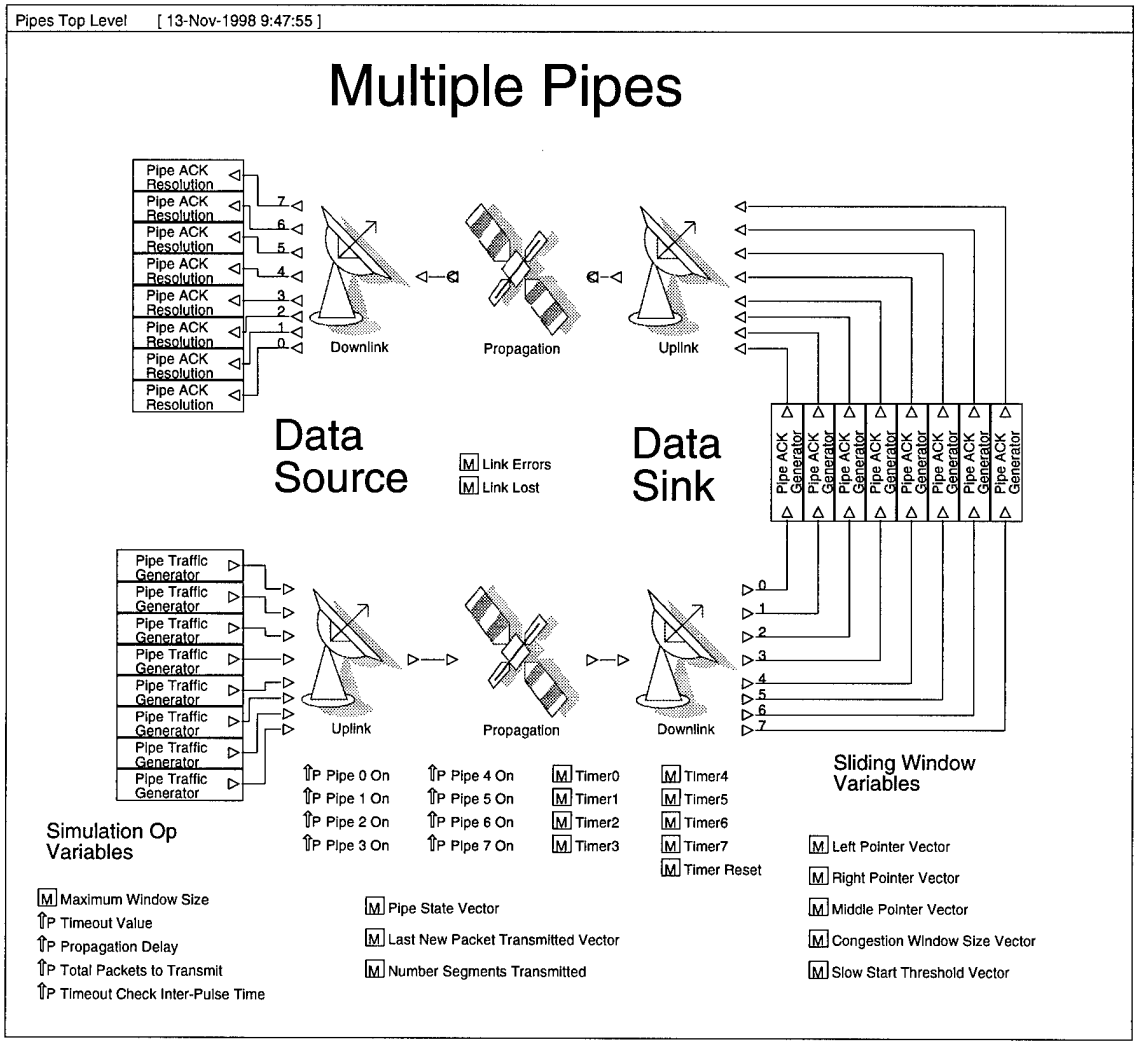

Figure 1 - MPP Top Level Model

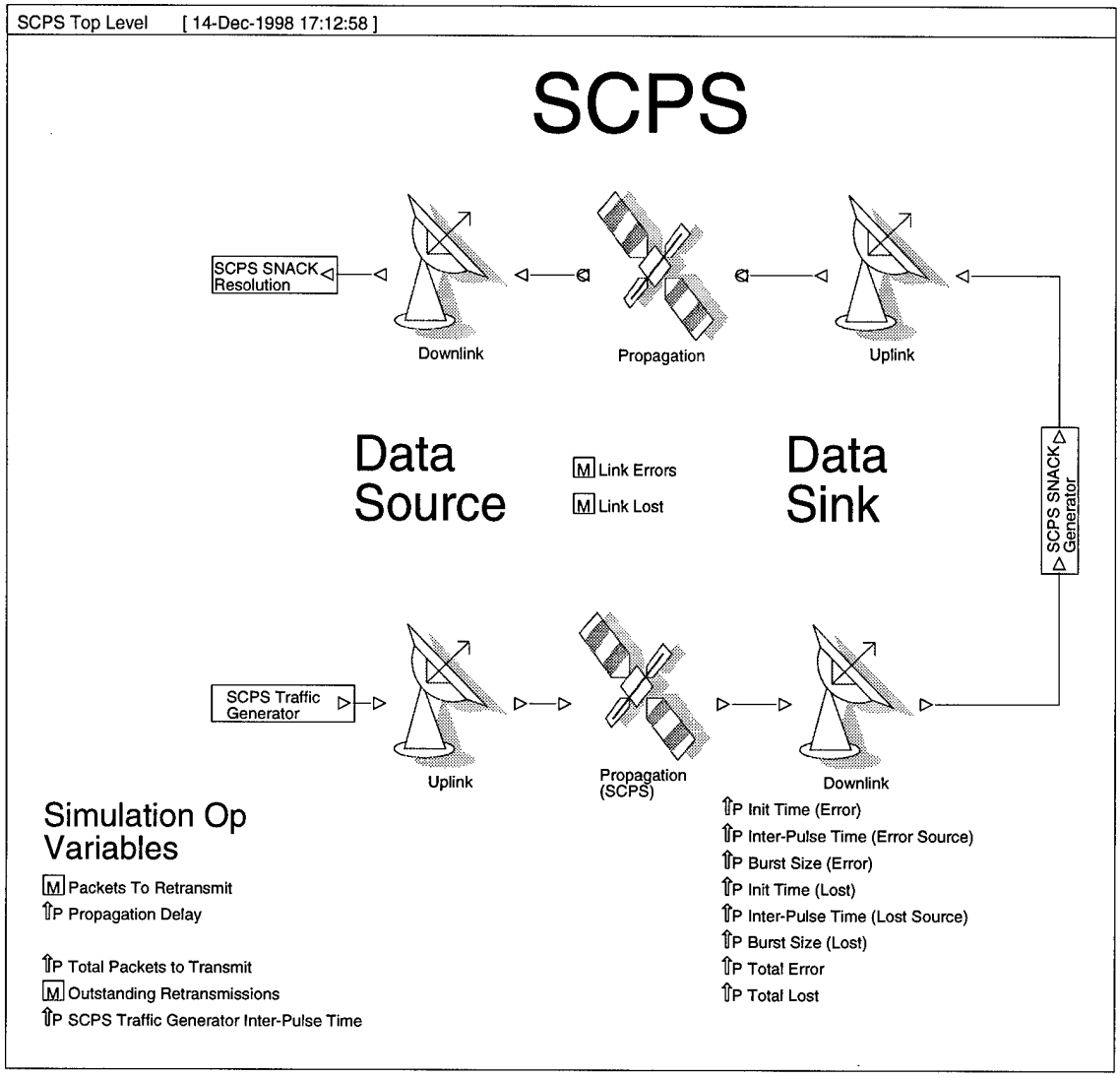

Figure 2 - SCPS-TP Top Level Model

As can been seen from the figures, the models are very similar. The SCPS-TP approach, however, uses a single "pipe" to flow the information from the *Data Source* to the *Data Sink*. The major technical difference between the two models is that different primitives are used to simulate the traditional TCP and SCPS-TP.

The number of pipe connections in the *Pipes Top Level* model deserves comment. As stated in Chapter 2, throughput using the MPP approach is maximized when 6 to 8

pipe connections are used. The authors of [AlK96] found that additional pipes lead to router saturation at the receiving node due to the large "effective" window size. While packet losses occasionally occur when the number of pipes is limited to eight, the level of loss was minimal. For this reason, eight pipes are used as the maximum number of pipe connections in the MPP model.

After Designer initializes a simulation, the *Pipes/SCPS Traffic Generator* modules (depending on the model) begin to generate packets. Primitives within these modules assign *Packet ID* numbers according to either the MPP or SCPS-TP protocol as appropriate.

The *Uplink*, *Propagation*, and *Downlink* modules transmit packets to the *Data Sink* node. Within the *Propagation* module that transmits packets from the *Data Source* to *Data Sink* nodes, errors and packet losses are inserted into the data stream and a propagation delay is imposed on packets. Once packets arrive at the *ACK/SNACK Generator* modules, a primitive generates ACK and SNACK messages.

Finally, *Uplink*, *Propagation*, and *Downlink* modules transmit the ACK/SNACK messages to the *ACK/SNACK Resolution* modules at the *Data Source* node. Within the *ACK/SNACK Resolution* modules, primitives examine the ACK/SNACK messages and adjust the Go Back N ARQ sliding window variables accordingly for MPP or the *Packets to Retransmit* for SCPS-TP. These variables are used by the *Pipe/SCPS Traffic Generator* modules to send the next series of packets, and the process continues until a pre-defined number of packets are sent from the *Data Source* to the *Data Sink.*

3.3 Operational Assumptions

In any simulation, it is important to document assumptions required to make a simulation practical. The following assumptions are incorporated in the Designer models.

3.3.1 Packet Size

Since it is desirable to fit TCP packets "nicely" into IP datagrams (no packet fragmentation), it is necessary to look at IP services to determine the size of the TCP packets to used. The default size of datagrams is 576 bytes [Com95], and when the TCP header (20 bytes) and the IP header (20 bytes) are subtracted, this leaves a segment size of 536 bytes. In this thesis, 512 bytes of user data are put into each packet and standard size TCP and IP headers are assumed. This size allows packets to be encapsulated in a standard IP datagram, and makes all the files of interest even multiples of packets.

3.3.2 Traffic Data Rates

The data rate is 2 Mbps. This rate matches the "Basic Channel" rate of the Teledesic system, which will be the first LEO constellation aimed at providing end-to-end satellite data transfer to the general public. Since packets are 552 bytes (512 bytes of user data plus 40 bytes of TCP and IP header), the number of packets the channel can support is found below:

$$\frac{2\ Mbits}{sec} \bullet \left(\frac{1\ byte}{8\ bits}\right) \bullet \left(\frac{1\ packet}{552\ bytes}\right) \approx \frac{474\ packets}{sec} \approx \frac{1\ packet}{2.11\ m\,sec}$$

Since the pipes in the MPP approach must share the channel, each can send 59.25 packets/sec or 1 packet every 16.9 msec.

3.3.3 SCPS-TP Error Assumption

The underlying cause of packet loss can be adjusted in the SCPS-TP protocol. In this thesis, the underlying cause is assumed to be noise.

3.3.4 Window Size

In Chapter 2 it was discussed how SCPS-TP replaces the sliding window with an open-loop token bucket approach when operating in Corruption Mode. Consequently, the window size does not effect the SCPS-TP simulations. In the MPP model, however, window size is important. RFC1323 [JaB92] allows TCP connections to have windows larger than the 64 Kbytes originally anticipated in the TCP specification. While RFC1323 is currently being supported in many implementations of TCP, it has not yet been included as part of the TCP specification by the IETF. Consequently, the largest window this research uses is 64 Kbytes (2^{16} bytes).

To make the MPP and SCPS-TP simulations comparable, it is important that MPP be capable of 100% link utilization. To accomplish this, the sliding window must be large enough to prevent idle time. In Section 3.3.2, it is shown that each pipe transmits a segment every 16.9 msec. The minimum sliding window is therefore equal to the propagation delay divided by the pipe packet rate; for the LEO simulations the minimum value is 5.917, which rounds to 6 segments, and in the GEO simulations the minimum value is 33.13 or 34 segments. The sliding window should be larger than the delay-

bandwidth product, but should also be small enough to prevent congestion [DuM96]. The MPP simulations will therefore use 48 segments (24 Kbytes of outstanding data) for the maximum sliding window size. Since typical Unix FTP connections use window sizes between 4 Kbytes and 24 Kbytes [AlK96], this figure seems appropriate.

3.3.5 File Size

To test the protocol's adaptability to transfer different size files, file sizes of 100 Kbytes, 1 Mbytes, and 5 Mbytes are used.

3.3.6 Delays

In [AlK96] the additional processing delay associated with multiple TCP connections is deemed trivial. Additional processing delay for this research, therefore, is assumed to be zero. In [Tel97], the latency of the Teledesic system is reported to be as low as 20 msec and always less than 75 msec. Splitting the difference, 50 msec is used as the one-way propagation delay for LEO systems. In [All97] the observed round-trip delay of the NASA ACTS satellite was measured at 560 msec – 280 msec is therefore used to model one-way propagation for GEO systems.

3.4 Modeling Errors/Packet Loss

One of the difficulties in designing the models created for this research was finding ways to model packet loss resulting from system congestion and noise errors. As stated in Chapter 2, timeouts have drastic effects on the sliding window, causing the Slow Start Threshold to be cut by one-half and the Congestion Window Size to be reset to one. On the other hand, if errors cause the transmission of duplicate ACKs, the Fast Retransmit

and Fast Recovery algorithms handle retransmission. While the Fast Retransmit/Fast Recovery algorithms halve the Slow Start Threshold, the sliding window is not reset to one. Clearly, the comparison between the MPP and SCPS models will be greatly effected by the number of transitions between TCP protocols.

3.4.1 Errors

When designing a telecommunications system, it is important to ensure the design conforms to recognized and accepted standards. When a company is designing a system targeted to the world market, it is important that the standards are internationally recognized. This approach minimizes the chance of violating national standards and furthers the goal of creating truly global systems.

3.4.1.1 ITU G.821 Standard

The *International Telecommunication Union (ITU)* is the United Nations Specialized Agency responsible for telecommunications. Within the ITU is the ITU Telecommunication Standardization Sector (ITU-T), which is responsible for "studying technical, operating and tariff questions and issuing recommendations on them with a view of standardizing telecommunications on a worldwide basis" [ITU97]. At the August 1996 *World Telecommunications Standardization Conference (WTSC)*, ITU-T Recommendation G.821 was approved. This recommendation sets error performance parameters for digital networks containing *Integrated Services Digital Networks (ISDN)* links. Since ISDN was designed around digital communication links, and since satellite systems like Teledesic will allow users to continue to use familiar protocols (like ISDN),

using G.821 in this research is appropriate for determining the number of errors to insert in the data stream.

G.821 applies two key parameters when defining error performance. The first parameter is called the *Errored Second Ratio (ESR)*, which measures the ratio of seconds during a connection in which one or more bit-errors occur to the total connection time in seconds. The second parameter is called the *Severely Errored Second (SES)*, which is the ratio of seconds in a connection that have a bit error ratio ≥ 1.10 to the total connection time in seconds. Table 1 shows the ITU specification.

Table 1 - G.821 Error Performance Specification

Performance Classification	Objective
Severely Errored Second Ratio	< 0.002
Errored Second Ratio	< 0.08

The ITU specification is based on time, but the specification calls for the error performance to be monitored for an extended period (usually a month) to establish the performance ratios. The extended sample time clearly indicates that the performance figures are meant to be averages. It seems logical that the ratio of corrupted packets introduced into the data stream to total packets should be the same as the time-based error ratios; this approach seems to meet the intention of G.821.

Knowing that wireless links are much noisier than their wired counterparts, modern satellite systems make extensive use of *Forward Error Correction (FEC)*. Consequently, this research assumes FEC will eliminate errors measured by the Errored Second Ratio, while errors measured in the Severely Errored Second ratio will not be corrected. The

Severely Errored Second Ratio, therefore, is used to determine the number of errors introduced in the model's data streams [ITU97].

The G.821 specification breaks down a single communication link into three different "grades", called Low Grade, Medium Grade, and High Grade. Portions of the Severely Errored Second Ratio are assigned to each grade. The Low Grade defines the link between the endpoint and the local exchange, while the Medium and High Grades represent the long-haul portions of the connection. Exact demarcation between Medium and High Grade vary from location to location. G.821 allocates .0003 of the .002 SES ratio to a satellite hop (which is part of High Grade), while the remaining portion is divided between High Grade, Medium Grade, and Low Grade [ITU97].

When designing the G.821 specification, it seems clear that the authors envisioned satellites providing only part of a communication link. They anticipated connections going through a Local Exchange, Primary Center, Secondary Centers, Tertiary Centers, and an International Switching Center. Depending on the distance between any two of these centers, different types of links could have different portions of the SES budget, depending on where it was placed in a link [ITU97]. This clearly does not consider systems like Teledesic, which will provide end-to-end communications using satellite links exclusively. It is therefore logical to run experiments assigning the entire 0.002 SES to the wireless links in addition to experiments assigning only the .0003 SES portion. Table 2 shows how many errors should be introduced for the file sizes of interest for the two SES levels. For fractional numbers of errors, the number in parentheses is the rounded number of errors.

Table 2 - SES Errors

File Size	Total Packets	Packet Errors (.002 SES)	Packet Errors (.0003 SES)
100 Kbyte	200	0.4 (0)	0.06 (0)
1 Mbyte	2,000	4	0.6 (1)
5 Mbyte	10,000	20	3

Once the number of packet errors is determined, consideration is given to where the errors will be introduced into the streams. Since we are dealing with a satellite system providing end-to-end communication, all errors are introduced in the *Propagation* modules in the two models. Since these modules enforce Time Division Multiplexing in the MPP model, the errors are distributed over the eight pipes. The SCPS-TP model uses only a single pipe, so all errors are introduced into a single data stream. This approach represents real life.

As explained in Appendix A, the *Error Source* modules use a "Uniform Traffic Generator" to increment a counter that corrupts packets. Each time the traffic generator is triggered, a fixed number of errors are injected into the data stream. Since the behavior of the systems may be effected by how many errors are in an error burst (which effects the transition between different TCP protocols), several different burst sizes are used. Table 3 shows the burst sizes that are simulated.

Table 3 - Error Burst Sizes

File Size	Packet Errors (.002 SES)	Burst Sizes	Packet Errors (.0003 SES)	Burst Sizes
100 Kbyte	0	0	0	0
1 Mbyte	4	1, 2, 4	1	1
5 Mbyte	20	1, 2, 4, 10, 20	3	1, 3

3.4.2 Alternate Error Approach

An alternative to using a published standard to determine how many errors to inject into a data stream is to arbitrarily choose a percentage of packets that the user wants to corrupt. Once the percentage is determined, methods similar to those above can be used to determine the number of corrupted packets and how the packets should be distributed.

A system based on standards is traditionally designed to operate in a relatively benign/nominal environment, where the most severe environmental problem encountered is rain fade. The military, on the other hand, sometimes operates in environments where transmissions are intentionally jammed. Any performance evaluation that assumes a standards-compliant link is available, therefore, may not be appropriate in a military environment.

To check to see how these two approaches adapt when links are severely degraded, the number of packet errors introduced into the data stream is increased significantly. Simulations corrupting 1%, 2.5%, and 5% of all packets show how well these systems perform during conditions that may exist in a military environment. These percentages represent a 5-fold to 25-fold increase in the number of errors over the G.821 worst-case specification. Table 4 shows the number of errors that are introduced.

Table 4 - Error Count (Severely Degraded Link)

File Size	1% Error	2.5% Error	5% Error
100 Kbyte	2	5	10
1 Mbyte	20	50	100
5 Mbyte	200	250	500

Like previous simulations, all errors are introduced in the *Propagation* modules of the two models. The burst sizes of the *Error Source* modules are also varied, similar to the variation seen in Table 3.

3.4.3 Congestion Loss

While this research primarily focused on each model's ability to deliver traffic in the presence of errors, packet loss due to congestion must be addressed. Unlike errors, congestion losses are not based on the link between nodes; rather, timeouts are characteristic of the traffic that is carried by the link. As stated in Chapter 2, overrunning the queues of intermediate routers is the primary cause of congestion losses.

The purpose of this research is to compare the MPP and SCPS-TP approaches to improving TCP's ability to transfer a specific amount of data over a wireless link. It does not examine the effects caused by a wide variety of traffic models. Furthermore, there are no on-orbit examples of a large LEO constellation. Although fully deployed, the Iridium system is still undergoing initial testing and performance data is unavailable. This lack of data makes it impossible to realistically model traffic loads. While companies developing these systems hope they will quickly become attractive alternatives to terrestrial communication, there is no basis on which to estimate how heavy the traffic demands will be. Consequently, terrestrial models will likely not be representative of these systems. The premium pricing of these services may mean that few users will transition to these systems, possibly causing router loss to be minimized due to the relatively low amount of data injected into the system. On the other hand, many customers may reason that the truly global nature of these networks is worth the extra

cost. The higher number of users could increase the number of packets that are lost due to congestion.

Rather than estimate the number of packet losses based on projected future use, it is more reasonable to relate the number of losses in the system to the number of errors injected into the system. Toward that end, the number of timeouts that are injected into the system is proportional to the number of errors. Since this research is primarily concerned about comparing the MPP and SCPS-TP approaches in a noisy environment (as opposed to a congested environment), the proportions chosen is $^1/_{10}$ and ¼. Congestion losses are inserted as a single burst to minimize the impact on sliding window variables. In cases where the number of errors times a proportional value is not an integer, the result will be rounded down – again minimizing the impact of lost segments. Again, all congestion losses are introduced in the *Propagation* modules of both models.

3.4.4 Timeouts

Timeouts are caused when an ACK is not received for an outstanding packet within a specific time. Under the first implementation of TCP, the timeout value was fixed. Ideally, the value was set to:

$$\text{Timeout} = \beta * \text{RTT}$$

In the original specification, the value of β was $\beta = 2$. Several researchers soon found, however, that this approach was ineffective when system loads exceeded 30%. In 1989 the TCP specification was changed; the timeout calculation now use both average RTT and estimated RTT variance. The following equations were developed and found in most Unix implementations:

$$DIFF = SAMPLE - Old\_RTT$$

$$Smoothed\_RTT = Old\_RTT + \delta * DIFF$$

$$DEV = Old\_DEV + \rho\,(|DIFF| - Old\_DEV)$$

$$Timeout = Smoothed\_RTT + \eta * DEV$$

BSD 4.4 Unix uses values $\delta = 1/2^3$, $\rho = 1/2^2$, and $\eta = 4$. Timeouts are actually based on the number of 500ms "ticks" that expire between segment transmit and receipt of the appropriate ACK message [Com95, WrS95].

The purpose of segment timers is to adjust the timeout value as conditions in the system change. In the protocols, retransmissions are not used to change RTT; only segments that successfully transmit without retransmission are used to modify RTT. Since RTT of successfully transmitted segments is fixed in both the MPP and SCPS-TP models, the Timeout Value is set to the 500 ms tick that is just larger than the RTT assumed for the GEO and LEO systems. For LEO simulations, a timeout value of 500 msec is used, while GEO simulations use a value of 1.0 seconds.

To check for segment timeouts, a timer is maintained on each segment. When a segment leaves the *Pipes Traffic Generator* modules, a timer corresponding to the *Packet ID* is started. When a packet arrives at the *ACK Resolution* module, all outstanding packets are checked for timeout. If a packet times out, the models enter Slow Start and all packet timers are reset. In addition, timeout checks are triggered periodically whether or not a packet arrives; this prevents the model from becoming deadlocked by packets being corrupted and discarded while the sliding window prevents additional packets from being transmitted.

Since there is no sliding window in the SCPS-TP model to stop segment transmission, timeouts are not necessary in the SCPS-TP simulations. The first uncorrupted segment delivers a SNACK message to the *SNACK Resolutions* modules that indicates all segments requiring retransmission. The lack of a sliding window means there is no difference whether a SNACK message triggers retransmission or a timeout.

3.5 Data Gathering

The ultimate indicator when comparing two systems' ability to perform a specific task is a wall-clock measurement of the time the systems require under identical conditions. Since the goal of this research is to successfully transfer a set amount of data between nodes, simulation execution time is the main metric for comparison. The simulation time of MPP and SCPS-TP is compared to determine each algorithm's effectiveness under the criteria previously developed. The models are designed so that th simulations terminate after enough packets are successfully transferred from *Data Source* to *Data Sink*, simplifying data collection.

Probes are placed in the *ACK/SNACK Resolution* modules to plot *Global Segment Number* vs. "TNow". Data from these probes make it simple to determine how long the simulations take to transfer different amounts of data. In addition, probes are also placed at the output of active *Insert Error* and *Insert Lost* modules to ensure the proper number of errors and congestion losses are inserted into the data stream. Finally, probes are placed on the Sliding Window variables in the module to monitor how the simulation reacts to errors and timeouts, which helps ensure the primitives are functioning properly.

3.6 Model Verification/Validation

Once the models are created in Designer, it is necessary to verify that they are functioning according to plan, and that the plan used to design the models is sound. The Designer tool contains several internal functions to ensure type compatibility between modules, check that all variables are initialized, ensure connectivity, and other checks well suited to mechanical verification.

Once the models are void of obvious errors, it is necessary to ensure the design actually accomplishes the intended tasks. Toward that end, the models went through several layers of peer review with fellow students, faculty experts, and the thesis committee. Oversights were explored and all issues were resolved. Once the design was finalized, individual modules were tested. Several small simulations were constructed to ensure all modules function correctly by comparing simulation results against hand-calculated solutions. In particular, the function of the Designer primitives and their interaction with the sliding window variables was checked. Finally, the entire system was simulated for both models. Designer allows users to step through a simulation and check the system response at each step. Both models were checked using this approach, and found to be functioning correctly.

Unfortunately, no LEO satellite system has been deployed to an extent that permits independent validation of the relative performance of the MPP and SCPS-TP approaches to improving TCP performance. Wherever possible, data used in these simulations has been taken from literature that used actual systems and satellites to perform their analysis. In cases where data was unavailable, it was necessary to extrapolate from related data or

use, in consultation with the thesis advisor, the best judgement of the author. In particular, the number of timeouts and the method of their introduction represent the author's best estimates.

3.7 Conclusion

Through the methods listed above, models of the MPP and SCPS-TP approach to improving TCP performance were created and verified. By examining the performance requirements of an internationally recognized standard, the number of packet errors to introduce into the data stream for different file sizes was developed. To gain information on the time required to transfer various file sizes, multiple simulations were run with various numbers of errors and timeouts being inserted into the data stream of each file size. The goal of these simulations was to determine the relative performance of the two protocols. The results of these simulations are found in Chapter 4.

4 RESULTS AND ANALYSIS

4.1 Introduction

Going into the experiments, it is known that SCPS-TP's lack of a sliding window when operating in Corruption Mode ensures that it requires less time to transfer data under similar conditions than the MPP approach. The question to be addressed is, "How much of a performance advantage does SCPS-TP provide over MPP?" within the constraints discussed in Chapter 3.

4.2 Comparison at G.821 Error Rates

Table 5 shows required execution times to transfer the three file sizes of interest over a LEO network at G.821 error rates. For the 100 KB file at both .002 and .0003 SES and the 1 MB file at .0003 SES, no errors are injected, so transmission times remain constant. For the other test cases, several simulations were run for each file size at each error rate. The "Best Case" and "Worst Case" columns show the execution extremes, while the "Average Case" is the numerical average of all simulation runs for a particular file size/error rate combination.

For the 100 KB file at either .0003 or .002 SES, SCPS-TP shows a 26.1% faster transfer time than MPP. The performance advantage, however, is quickly reduced as the file size increases to 5 MB, where the maximum advantage SCPS-TP holds is 2.7%. Figure 3 graphically shows the performance enhancement offered by SCPS-TP at G.821 error rates. While the ratio of SCPS-TP time to MPP time is 1.35 for the smallest file

(100 KB), it quickly drops, and by the time the file size reaches 5 MB, the ratio is less than 1.011 for .0003 SES and 1.024 for .002 SES.

Table 5 - LEO Transfer Times (.0003, .002 SES) in seconds

		MPP			SCPS-TP		
LEO	SES Rate	Best Case	Worst Case	Average Case	Best Case	Worst Case	Average Case
100KB	0.0003	0.703	0.703	0.703	0.520	0.520	0.520
	0.002	0.703	0.703	0.703	0.520	0.520	0.520
1MB	0.0003	4.520	4.520	4.520	4.320	4.320	4.320
	0.002	4.577	4.596	4.591	4.326	4.328	4.327
5MB	0.0003	21.438	21.438	21.438	21.204	21.206	21.205
	0.002	21.597	21.814	21.748	21.244	21.221	21.233

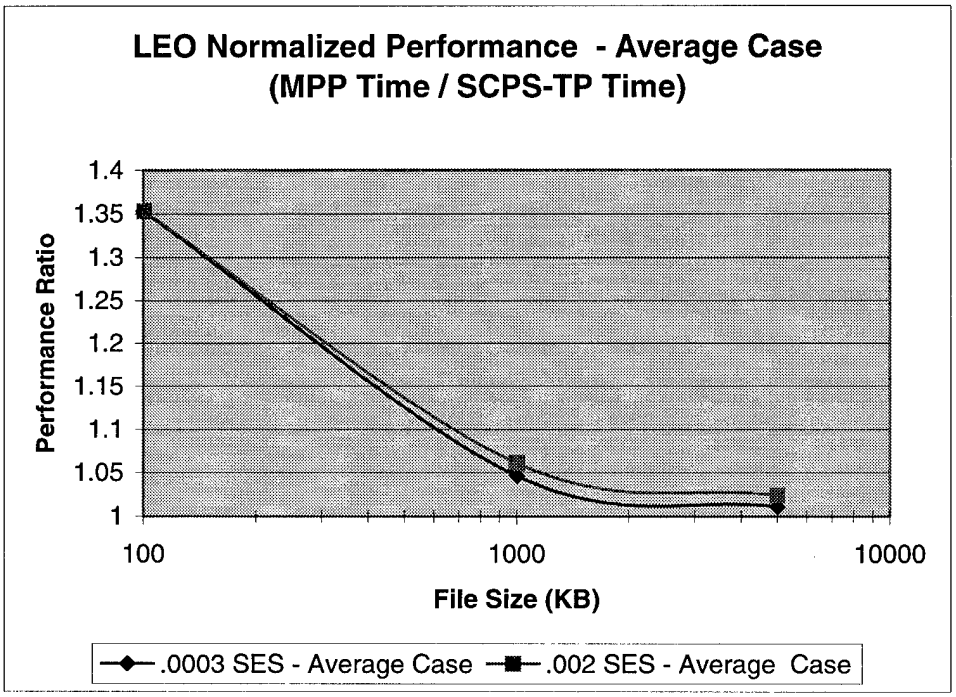

Figure 3 - LEO Normalized Performance (.0003, .002 SES)

While Figure 3 represents the Average Case, the same trend is observed in the Worst Case and Best Case data. In the Worst Case, the ratio is unchanged for .0003 SES and for .002 SES the ratio increased from 1.024 to 1.027. In the Best Case, the ratio is again unchanged for .0003 SES; for .002 SES the ratio decreases from 1.024 to 1.017. At G.821 error rates, the variance and standard deviation of MPP simulations are small. The maximum variance and standard deviation in the LEO simulations is found in the 5 MB file at .002 SES, but even in this case the mean transfer time is 21.748, with standard deviation of 0.090 and variance 0.008. SCPS-TP also has small variance and standard deviation under the same conditions – mean of 21.232 with standard deviation of 0.016 and variance of 0.0002.

When the simulation assumes a geosynchronous satellite is used, the results are predictably different. Table 6 shows the transfer times for GEO simulations.

Table 6 - GEO Transfer Times (.0003, .002 SES) in seconds

		MPP			SCPS-TP		
GEO	Error Rate	Best Case	Worst Case	Average Case	Best Case	Worst Case	Average Case
100KB	0.0003	3.020	3.020	3.020	0.980	0.980	0.980
	0.002	3.020	3.020	3.020	0.980	0.980	0.980
1MB	0.0003	7.323	7.323	7.323	4.780	4.780	4.780
	0.002	7.839	8.255	8.051	4.786	5.326	5.023
5MB	0.0003	24.618	24.707	24.663	21.664	22.223	21.943
	0.002	26.609	31.383	29.647	21.704	22.244	21.974

The pattern observed in the LEO simulations is repeated in the GEO simulations. MPP performance over a GEO satellite, however, is not as close to SCPS-TP performance as in the LEO simulations due to the increased propagation time and the

larger delay-bandwidth product (DBWP). SCPS-TP's transfer time is 67.6% less for the 100 KB file in the GEO simulations, but this advantage is reduced to 29.1% for the 5 MB file at the .002 SES level and 10.1% at the .0003 SES level. Figure 4 shows MPP/SCPS-TP performance ratio for the GEO simulations. Again, that ratio is high for the smallest file (3.08 in this case), but it decreases until at 5 MB the ratio even in the Worst Case is 1.11 and 1.41 for .0003 SES and .002 SES.

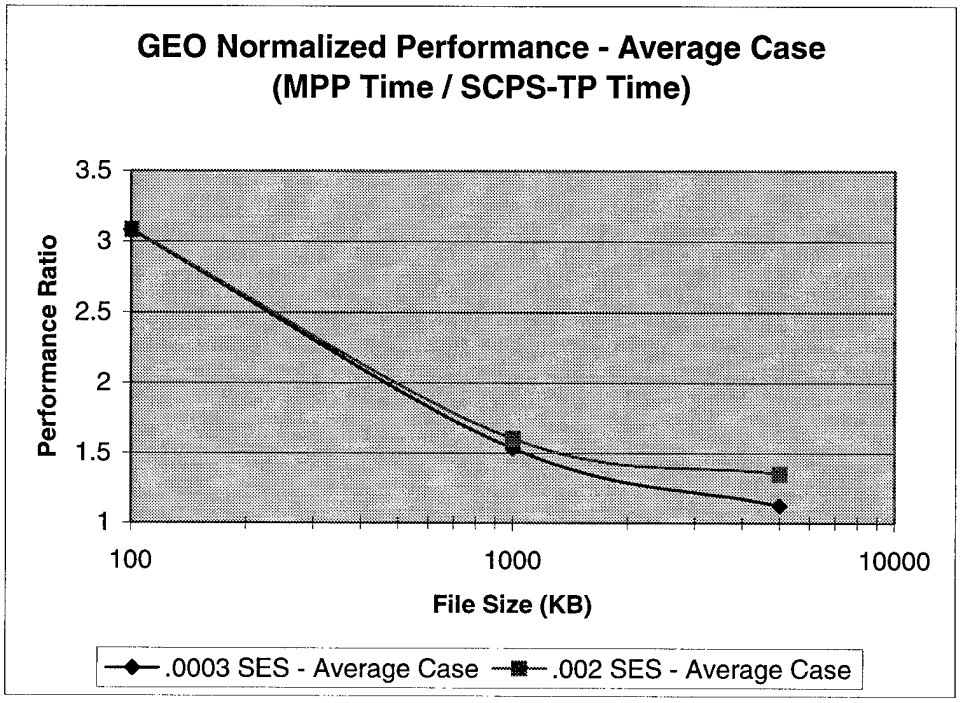

Figure 4 - GEO Normalized Performance (.0003, .002 SES)

While the variance and standard deviation of MPP in the GEO simulations increases from their LEO counterparts, these values are still small. The mean GEO transfer time for 5 MB at .002 SES is 29.647 seconds, with a standard deviation of 2.133 and variance of 4.551. The SCPS-TP variance and standard deviation also increase, but

are still smaller than MPP – mean transfer time of 21.974 with standard deviation 0.382 and variance 0.146.

Discounting SCPS-TP's advantage of not experiencing idle time while the sliding window is initially increased until it "covers" the DBWP, SCPS-TP offers no advantage to the smallest files. The G.821 specification dictates that at .002 SES rate, a file must be at least 500 KB before an error is injected. While one error is inserted in the 1 MB file to approximate the .0003 SES rate, this is because the simulation assumes that fractional errors are always rounded up. In reality, an error is not introduced at .0003 SES until the file size is approximately 1.7MB. Table 7 shows the percent of SCPS-TP's performance advantage (defined as MPP transfer time – SCPS-TP transfer time) that is attributed to the initial sliding window.

Table 7 - SCPS-TP Advantage Due to Initial Sliding Window (Low SES)

Orbit/ File Size	Error Rate	Initial Sliding Window Delay	% SCPS Advantage Best Case	% SCPS Advantage Worst Case	%SCPS Advantage Average Case
LEO					
100KB	0.0003	0.18357	100.0%	100.0%	100.0%
	0.002	0.18357	100.0%	100.0%	100.0%
1MB	0.0003	0.18357	91.6%	91.6%	91.6%
	0.002	0.18357	73.1%	68.5%	69.6%
5MB	0.0003	0.18357	78.4%	79.1%	78.7%
	0.002	0.18357	52.1%	31.0%	35.6%
GEO					
100KB	0.0003	2.04037	100.0%	100.0%	100.0%
	0.002	2.04037	100.0%	100.0%	100.0%
1MB	0.0003	2.04037	80.2%	80.2%	80.2%
	0.002	2.04037	66.8%	69.7%	67.4%
5MB	0.0003	2.04037	69.1%	82.1%	75.0%
	0.002	2.04037	41.6%	22.3%	26.6%

At least 22.3% of SCPS-TP's advantage for all files at both SES rates is attributed to this cause. If the error rate is limited to the portion G.821 specifically allocated to a satellite link (.0003 SES), in the Average Case the maximum percent of total delay attributed to any factor other than the initial sliding window is 25% for a GEO satellite and 21.3% for a LEO satellite.

In addition to initial sliding window delay, a link can experience delays caused by errors reducing the sliding window until it does not cover the DBWP. As stated in Chapter 3, the LEO simulations require six segments to cover the DBWP, while the GEO simulations require 34 segments. In a LEO simulation with a maximum sliding window of 48 segments (24 KB of outstanding data), a minimum of four errors in a single pipe are required to reduce the sliding window to the point that idle time is experienced. In GEO simulations with the same sliding window, a single error is sufficient to cause idle time. The number of times a link experiences idle time from the congestion control algorithms depends on how large the sliding window is each time an error is introduced and into which pipe the error is inserted. For example, the 1 MB file at .002 SES allows four packet errors to be injected. Theoretically, a LEO simulation could experience idle time if all four packet errors are inserted into the same pipe, but since each pipe is equally likely to get an error, the probability of this situation occurring is less than 1 in 500.

4.3 Comparison at High Error Rates

When the error rate reaches levels above the G.821 specification, the performance advantage of SCPS-TP becomes clear. Table 8 shows the transfer times at the high error

rates over a LEO satellite. Unlike SCPS-TP's advantage at low error rates, the performance of the LEO simulations at high error rates is very dependent on whether Best Case, Average Case, or Worst Case data is examined. If Best Case is chosen, the data shows the familiar situation of SCPS-TP's performance advantage diminishing as file size increases. The reason is that in Best Case simulations, all errors are introduced as a small number of large error bursts. As will be explained in Section 4.4, MPP is adept at handling large numbers of errors when they follow the Best Case distribution.

Table 8 - LEO Transfer Times (.01, .025, .05 SES) in seconds

		MPP			SCPS-TP		
LEO	Error Rate	Best Case	Worst Case	Average Case	Best Case	Worst Case	Average Case
100KB	0.01	0.773	0.904	0.839	0.600	0.609	0.604
	0.025	0.798	0.904	0.867	0.530	0.621	0.562
	0.05	0.860	0.919	0.895	0.543	0.642	0.574
1MB	0.01	4.717	4.986	4.883	4.364	4.438	4.382
	0.025	4.837	6.080	5.470	4.434	4.449	4.441
	0.05	5.004	7.333	6.122	4.550	4.607	4.588
5MB	0.01	21.884	23.529	22.886	21.430	21.474	21.452
	0.025	22.335	29.024	25.615	21.779	21.789	21.784
	0.05	23.112	36.000	31.109	21.789	22.316	22.052

Figure 5 shows the pattern seen in both the Average Case and Worst Case data. In the LEO simulations, if the error rate is 2.5% or below, the advantage of SCPS-TP diminishes as file size increases. In the Average Case, the performance ratio is 1.54 and 1.39 for the 100 KB file at .025 and .01 SES respectively (1.54 and 1.23 in the Worst Case). As the file size increases, the SCPS advantage decreases until it is 1.18 and 1.08 in the Average Case (1.03 and 1.02 in the Worst Case) for the 5 MB file.

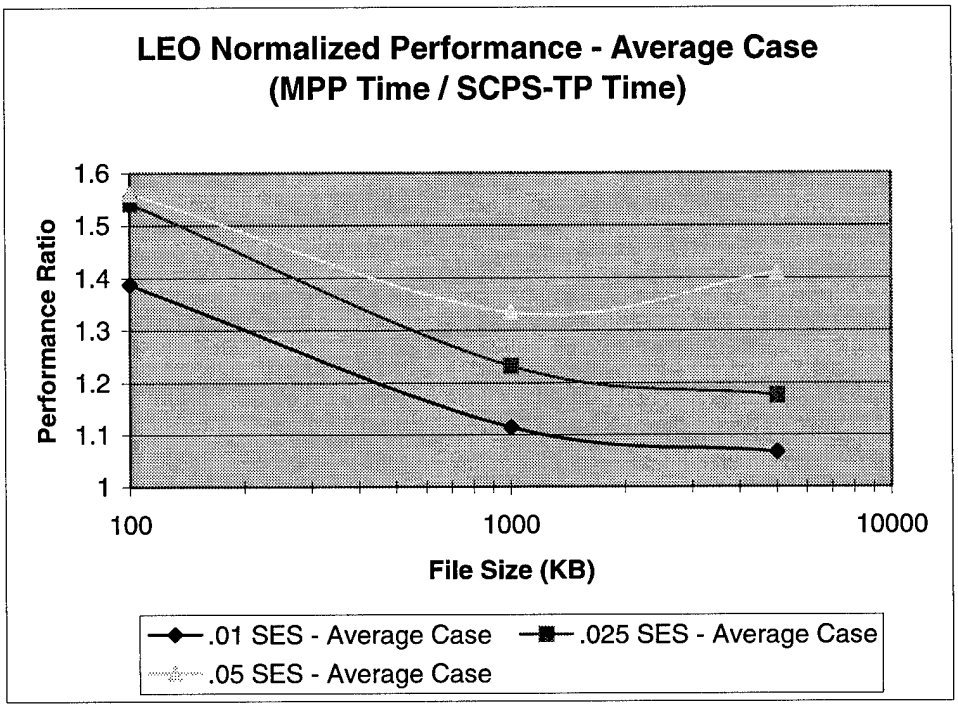

Figure 5 - LEO Normalized Performance (.01, .025, .05 SES)

At .05 SES, the familiar pattern no longer applies. When the 100 KB file is transferred, the initial sliding window causes 73.9% of the total delay (Table 7 – p. 42), which allows SCPS-TP to have 22.4% to 32.7% advantage over MPP. As the file grows, the initial sliding window delay is distributed over the entire transfer time and the delay caused by errors takes over. Once the file size reaches 5 MB, the delay caused by errors causes the performance ratio to increase 5.7% from the minimum observed at 1 MB. As the file size or the error rate increases, the trend of having higher performance ratios will continue, showing SCPS-TP's increasing advantage over MPP. Naturally, the .05 SES MPP simulations have high variance and standard deviation. For the 5 MB file at .05 SES, the average transfer time is 31.109, the standard deviation is 5.019, and the variance is 25.194. On the other hand, the SCPS-TP variance and standard deviation remain low.

Using SCPS-TP, the 5 MB file at .05 SES has an average of 22.973, standard deviation of 0.369 and variance of 0.136.

The pattern observed in LEO simulations is largely repeated in the GEO simulations. Table 9 gives the appropriate transfer times.

Table 9 - GEO Transfer Times (.01, .025, .05 SES) in seconds

		MPP			SCPS-TP		
GEO	Error Rate	Best Case	Worst Case	Average Case	Best Case	Worst Case	Average Case
100KB	0.01	3.067	3.797	3.432	1.229	1.531	1.380
	0.025	3.493	5.421	4.219	1.463	1.543	1.494
	0.05	3.657	5.424	4.254	1.472	1.533	1.501
1MB	0.01	9.570	12.353	10.606	4.740	5.358	5.064
	0.025	10.186	16.322	13.688	4.888	5.419	5.213
	0.05	11.524	16.821	14.467	5.010	5.528	5.269
5MB	0.01	32.062	53.738	46.224	21.890	22.392	22.141
	0.025	27.534	75.959	56.681	22.183	22.709	22.446
	0.05	41.667	119.442	96.404	22.713	23.234	22.973

As in the LEO case, the advantage of SCPS-TP over MPP in the GEO environment is dependent on whether the Best Case, Worst Case, or Average Case data is examined. In the Best Case, the performance advantage again diminishes as file size increases, although the improvement is not as dramatic as in the LEO simulations. The 100 KB file has performance ratios between 2.49 and 2.50 for the three error rates, and these rates are reduced to between 1.24 and 1.83 when the file size is 5 MB.

While the .05 SES GEO simulations show the pattern observed in the .05 SES LEO case, the .025 pattern changes. In the Average Case (Figure 6), the performance ratio of .025 SES decreases from 2.82 to 2.52 as the file size increases. In the Worst Case (Figure 7), however, the .025 SES performance ratio starts at 3.51 for 100 KB, decreases to 3.01 for 1 MB, and then starts increasing again until it is 3.34 for 5 MB.

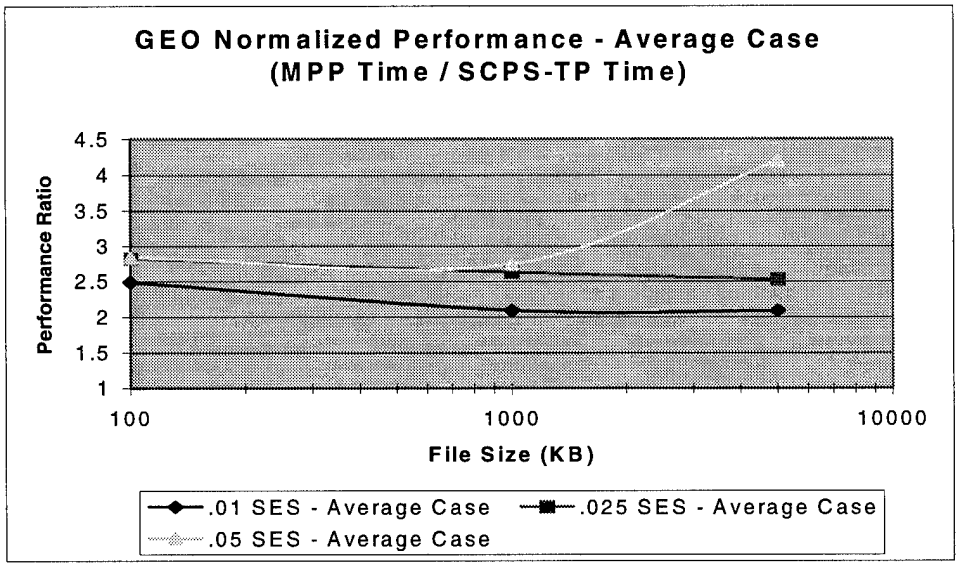

Figure 6 - GEO Normalized Performance (.01, .025, .05 SES) – Average Case

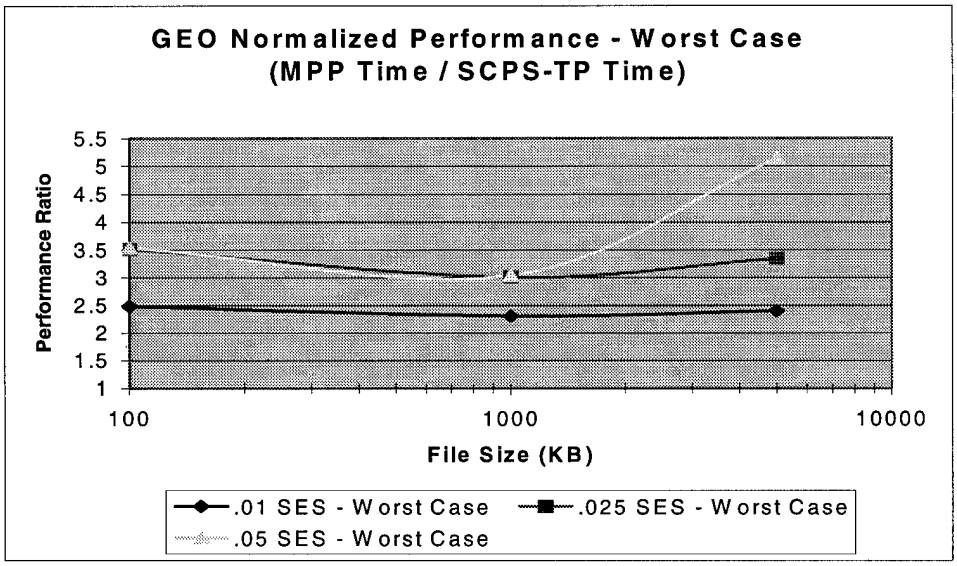

Figure 7 - GEO Normalized Performance (.01, .025, .05 SES) – Worst Case

The .01 SES shows a similar change. In the Average Case, the values are 2.49, 2.09, and 2.08. In the Worst Case, the performance ratio of 100 KB is 2.48, 2.31 at 1 MB, and 2.40 for the 5 MB file.

For the Average Case and Worst Case .05 SES simulations, the advantage of SCPS-TP is very clear. The ratio starts out at 2.83 in the Average Case (3.53 in the Worst Case), dips to 2.74 (3.04) for the 1 MB file, and then climbs to 4.19 (5.14) for the 5 MB file. Again, the decrease at 1 MB is caused by the redistribution of the initial sliding window delay. The ratio increases as the larger files allow the injection of a large number of additional errors. Variance and standard deviation of MPP are again high. For the 5 MB file at .05 SES, the mean is 94.404, the standard deviation is 27.542, and the variance is an incredible 758.569. SCPS-TP, on the other hand, retains small standard deviation and variance figures. At .05 SES for the 5 MB file, SCPS-TP has a mean of 22.973, standard deviation of 0.368, and variance of 0.136.

When analyzing the performance ratio curves, care must be taken when the trends are not pronounced. By placing a 90% confidence interval around certain data sets, it is possible to change the shape of some of the 1% and 2.5% LEO and GEO performance ratio curves. Wherever this situation occurred, however, the change shows SCPS-TP gaining an increasing advantage over MPP at lower error rates. At error ratios covered by G.821, confidence intervals had no effect on curve trends.

The percent of the SCPS-TP advantage directly attributable to the small initial sliding window is much smaller at high SES rates than at the lower rates. Table 10 shows that the entire delay is attributable to the initial sliding window only for the 100 KB file at the smallest error rate in the Best Case scenario. Once the file size reaches 1 MB, delays associated with causes other than the initial sliding window always consist of at least 50.9% of total delay, and on average they are responsible for more than two-thirds

of total delay. When the file size reaches 5 MB, even in the Best Case at .05 SES the initial window is responsible for 13.1% and 9.7% of total delay in the LEO and GEO simulations respectively. In the Worst Case, the initial sliding window causes only 1.3% and 1.9% of total delay.

Table 10 - SCPS-TP Advantage Due to Initial Sliding Window (High SES)

Orbit/ File Size	Error Rate	Initial Sliding Window Delay	% SCPS Advantage Best Case	% SCPS Advantage Worst Case	%SCPS Advantage Average Case
LEO					
100KB	0.01	0.173	100.0%	58.6%	73.9%
	0.025	0.173	64.6%	61.2%	56.8%
	0.05	0.173	54.7%	62.6%	53.8%
1MB	0.01	0.173	49.1%	31.6%	34.6%
	0.025	0.173	42.9%	10.6%	16.8%
	0.05	0.173	38.2%	6.3%	11.3%
5MB	0.01	0.173	38.2%	8.4%	12.1%
	0.025	0.173	31.1%	2.4%	4.5%
	0.05	0.173	13.1%	1.3%	1.9%
GEO					
100KB	0.01	1.838	100.0%	81.1%	89.6%
	0.025	1.838	90.5%	47.4%	67.4%
	0.05	1.838	84.1%	47.2%	66.7%
1MB	0.01	1.838	38.1%	26.3%	33.2%
	0.025	1.838	34.7%	16.9%	21.7%
	0.05	1.838	28.2%	16.3%	20.0%
5MB	0.01	1.838	18.1%	5.9%	7.6%
	0.025	1.838	34.3%	3.5%	5.4%
	0.05	1.838	9.7%	1.9%	2.5%

4.4 Effective Jamming Techniques

When a protocol's effectiveness is studied, the research often inadvertently reveals methods to defeat the protocol. Limitations in resources and counter-jamming techniques (e.g. frequency hopping) often make it impractical to continuously jam a communication channel. To disrupt many different frequencies, jammers are usually forced to limit the

amount of time they spend dwelling at any single frequency, limiting the number of consecutive errors inserted into any single data stream. The goal of "effective" jamming is to maximize link disruption while minimizing the time a jammer spends attacking any particular channel.

In the course of this research, it was discovered that jamming effectiveness against a system using MPP is not based as much on how many segments are disrupted, but rather where in the data stream errors are placed. Some error distributions lead to longer execution times than simulations that have over ten times the number of errors but a different distribution. Execution time is maximized when the sliding window and the Slow Start Threshold are minimized, which increases idle time due to sliding window limitations. In addition, regular error injection forces the system to spend the majority of its time in Congestion Avoidance, which limits window increases to at most one segment per RTT.

The effectiveness of the techniques described below will vary in different MPP systems. A good measure of technique effectiveness, however, can be found by finding the number of segments required to cover the DBWP. The higher the number of segments, the more effective jamming will be at extending file transfer time. Naturally, GEO systems are more susceptible since the DBWP is larger than a comparable LEO system. Since non-piped TCP is simply the single-pipe case of the MPP, the following techniques will disrupt traditional TCP networks. In the future, the effectiveness of these jamming techniques will increase as higher transmission rates increase network's DBWP.

4.4.1 Error Burst Length

As long as the sliding window covers the DBWP, link utilization is 100%. MPP, therefore, is effective at dealing with a small number of large error bursts spread over long intervals; the window recovers from errors and grows to cover the DBWP. Jamming efficiency increases when errors are distributed as a large number of smaller bursts.

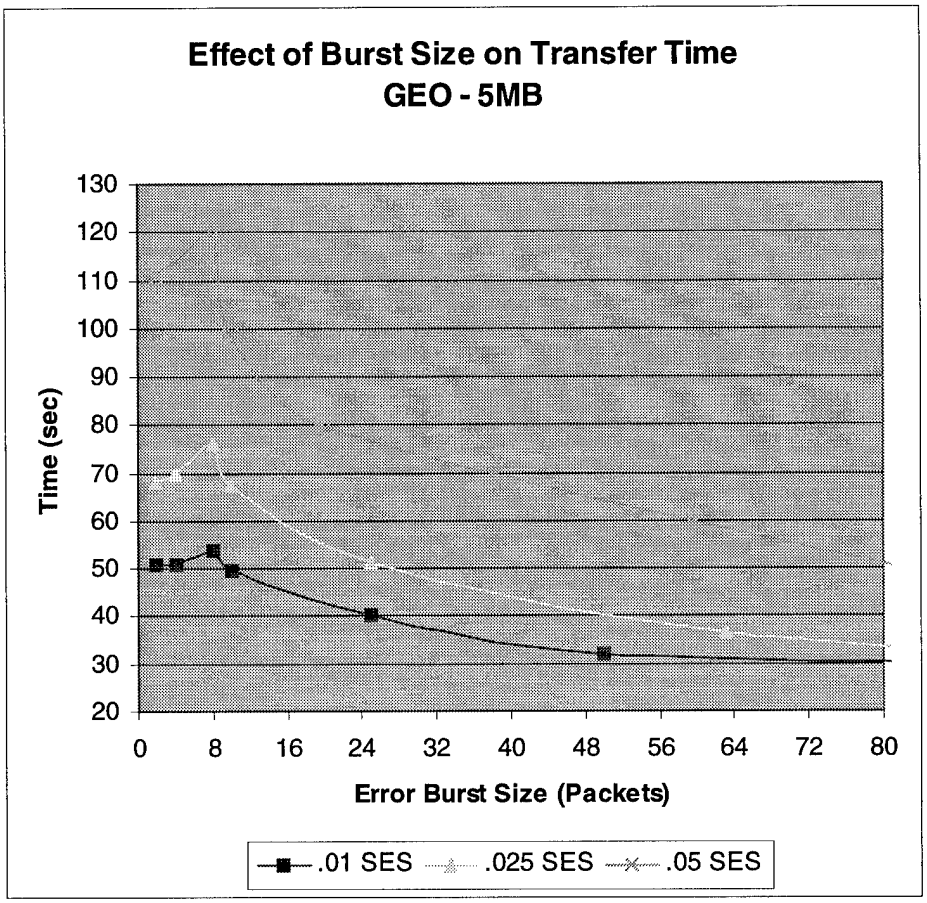

Figure 8 - Effect of Burst Size on Transmission Time

Although the pattern observed in Figure 8 is seen in all simulations, it is most easily recognized in the 5 MB, high SES rate, GEO simulations. Transmission time is

maximized when the number of packets corrupted in each burst equals the number of pipes used to transmit data. For example, the time required to transmit at .01 SES when the burst length is optimized is approximately 7% longer than the .05 SES (which is a five-fold increase in the error rate) that does not optimize the burst length. Burst length should therefore be:

$$\textit{Burst Length} = \textit{Pipe Packet Service Rate} * \textit{Number Pipe Connections} \quad (2)$$

Bursts of this length insert one corrupt packet into each pipe, resulting in the "effective" window being reduced by one-half. This approach negates MPP's goal of confining errors to a single pipe.

The reason for this effect is found in TCP's *Go-Back-N* ARQ strategy. Every packet transmitted between the initial corrupted packet and the time retransmission is triggered must be retransmitted even if the intermediate packets are not corrupted. Corrupting more than one packet in each pipe only delays the transmitting node finding out about the first corrupted segment. The resulting delay is a one-for-one time increase – a jammer's time might better be spent corrupting other links.

While all the pipe's sliding windows are larger than the DBWP, there is no need to synchronize jamming bursts to the transmitter since each pipe is constantly transmitting. Once the sliding windows are smaller than the DBWP, it becomes harder for the jammer to successfully corrupt packets since the time spent transmitting, particularly in a GEO system, is often very small compared to RTT. Jammers must synchronize their error bursts to the actual transmissions. Since RTT varies, it may be necessary to increase the length of each jamming burst to ensure at least one segment in each pipe is corrupted.

4.4.2 Burst Timing

Effective jamming is divided into two phases. In the first phase, the sliding window is reduced to the minimum allowable values ("Initial Reduction"). Once the window is minimized, the jammer works to ensure the sliding window is not allowed to grow significantly ("Sustainment").

Ideally, errors should be introduced before the sliding window has reached the receiving node's advertised window size. If one segment is disrupted before the sliding window reaches four segments, the sliding window will not allow the transmitter to send enough segments to trigger the Fast Recovery/Fast Retransmit algorithms. The resulting timeout reduces the sliding window to one and the Slow Start Threshold to two (the minimum allowed by RFC2001). Once the lost segment is retransmitted, Congestion Avoidance limits sliding window growth.

Table 11 shows the transfer time differential of inserting just one error into each pipe (8 total errors) when the sliding windows are two segments ("Early") versus inserting the error when the sliding windows have reached maximum size ("Late"). As the file sizes increase, the percent over baseline generally decreases as the sliding window recovers from the initial errors and grows until it exceeds the DBWP. Early error insertion, however, requires a jammer to synchronize the jamming bursts to the transmissions. Until the system has sent a number of packets, there is little data to analyze the variations in RTT, which can make synchronization difficult.

Table 11 - Early vs. Late Error Insertion

Orbit/ File Size	0 Errors (Baseline)	Early Error Insertion	% Over Baseline	Later Error Insertion	% Over Baseline
LEO	sec.	sec.		sec.	
100KB	0.703	1.455	106.781	0.855	21.596
1MB	4.501	5.253	16.687	4.653	3.375
5MB	21.381	22.133	3.514	21.533	0.711
GEO					
100KB	3.020	5.651	87.117	4.050	34.092
1MB	7.156	14.513	102.819	8.540	19.343
5MB	24.036	32.710	36.089	25.589	6.461

If a jammer is unable to inject the sliding window before it reaches the maximum size, then several bursts are required to minimize the window. Using the maximum sliding window and packet sizes, it is straightforward to compute the number of bursts required to minimize the sliding window (Equation 3). The floor function is used since the Slow Start Threshold cannot be smaller than two segments.

$$Number\ Bursts = \left\lfloor log_2 \left(\frac{Pipe\ Maximum\ Sliding\ Window}{Packet\ Size} \right) \right\rfloor \quad (3)$$

There must be a delay between the injection of error bursts. Any segments sent between the initial transmission of the first corrupted segment and the time the associated duplicate ACKs (or timeout) triggers retransmission are discarded. In addition to the RTT required for propagation and processing, there is time required to generate duplicate ACKs that trigger Fast Retransmit/Fast Recovery. If the system generates an ACK message with every packet arrival, then the ACK rate is equal to the packet service rate. Some TCP variations, however, do not send an ACK for every packet arrival to avoid a TCP problem called "Silly Window Syndrome" [Com95]. The burst interval is found in Equation 4.

$$Burst\ Interval = RTT + (3 * ACK\ Rate) \tag{4}$$

Once the sliding window and Slow Start Threshold are minimized, a jammer enters the Sustainment phase. Since the Slow Start Threshold is reduced to one-half the current window size when retransmissions are triggered (and cannot be smaller than two), a jammer can wait until the sliding window has grown to four segments before injecting another error. The result of the new error will be a timeout. One RTT later the sliding window will increment to equal two segments, and two more RTT are required under Congestion Avoidance before the sliding window reaches four segments. The interval between error injection, therefore, must be less than (3*RTT).

If it is not possible for a jammer to insert errors less than every (3*RTT) apart, it must insert a series of bursts if it wants to minimize the sliding window. The number of bursts required is found in Equation 5.

$$Number\ Bursts\ To\ Minimize = \left\lfloor \log_2 \left(\frac{Elapsed\ Time}{RTT} \right) \right\rfloor \tag{5}$$

4.4.3 Possible Counter-Jamming/Performance Improvement Techniques

As soon as methods are developed to jam a system, techniques are designed to counter the attempted jamming. These countermeasures are then attacked by counter-countermeasures; this process can be iterated as many times as desired.

As previously stated, introducing errors into the data stream before the sliding window has reached its maximum size has a significant effect on transmission time. In [All97] and [Flo97] it is proposed that the initial sliding window size be initialized to values other than one segment. By initializing the sliding window so it cover the DBWP,

the link it fully utilized from the start and the advantage of inserting the errors early in the data stream is eliminated.

In addition to making the initial sliding window larger, RFC1323 allows a larger maximum window size. Using a larger maximum window reduces the harm that an error inflicts on link utilization. If the sliding window is at least twice the DBWP, then a single error, while reducing the window by one-half, leaves the link capable of 100% utilization. Unfortunately, using a very large sliding window has inherent problems. A properly tuned TCP sliding window, while larger than the DBWP, is supposed to be small enough to prevent network congestion [DuM96]. RFC1323 does not negate this requirement – if the sliding window is too large, then congestion (and associated segment retransmission) is likely to occur, and the network may suffer congestive collapse. Even if congestion is not a factor (i.e. systems dedicated to a small number of high-priority users), a large sliding window is not a complete solution. If traditional Slow Start is used, a file must be larger than the pipe congestion window times the number of pipes to ever reach the maximum window size, and must experience no packet losses during the initial window growth. This problem can, of course, be overcome by initializing the window the maximum allowable window size. A jammer, however, always has an advantage since the congestion control algorithms reduce the sliding window by one-half in response to any packet loss. Even if the maximum window under RFC1323 is used (1GB [JaB92]), a jammer need only introduce $\log_2$(1 GB/512 B) or 21 bursts to minimize the sliding window. In a nominal environment, however, the larger initial window and maximum window size will increase performance.

Another problem of MPP is that once a segment is assigned to a particular pipe, no other pipe can transmit the segment. This leads to the situation where some pipes have transmitted all their segments (and subsequently shut down) while other pipes are still transmitting. Consider HyperText Transfer Protocol (HTTP) 1.0, which is used extensively for World Wide Web access. When a page is accessed, HTTP 1.0 creates several TCP connections – one for each object in the page. Text often arrives before multimedia objects since text usually requires much less data. This situation is very familiar – most people have experienced accessing a web page where the text becomes visible before the pictures are downloaded. In many situations, it may not matter that the transfer is incomplete – text can be read while the pictures are still downloading. A file, on the other hand, is incomplete and generally unusable until the entire file is transferred.

Consider Figure 9. In this figure, a single packet, transmitted at time t=2.296 is corrupted. Duplicate *Global Segment Numbers* trigger a retransmission, but retransmission (*Global Segment Number* = 0) does not begin until time t=3.480; retransmission is not complete until time t=3.733. Back at time t=3.020, however, all the segments in the file, except for the segments that pipe 0 must retransmit, have successfully arrived at the receiving node. This means that 19.1% of total execution time is spent waiting for pipe 0 to retransmit the lost segments.

One approach to improving MPP performance is to allow all pipes to cooperate in retransmitting lost segments. Once all segments have been transmitted at least once (and have therefore been assigned to a specific pipe), pipes that complete transmitting all their assigned segments could help other pipes transmit additional segments. Segments that

require retransmission (as well as outstanding segments that might require retransmission) could be assigned to one (or more) additional pipes depending on data transfer requirements and the probability of subsequent segment loss.

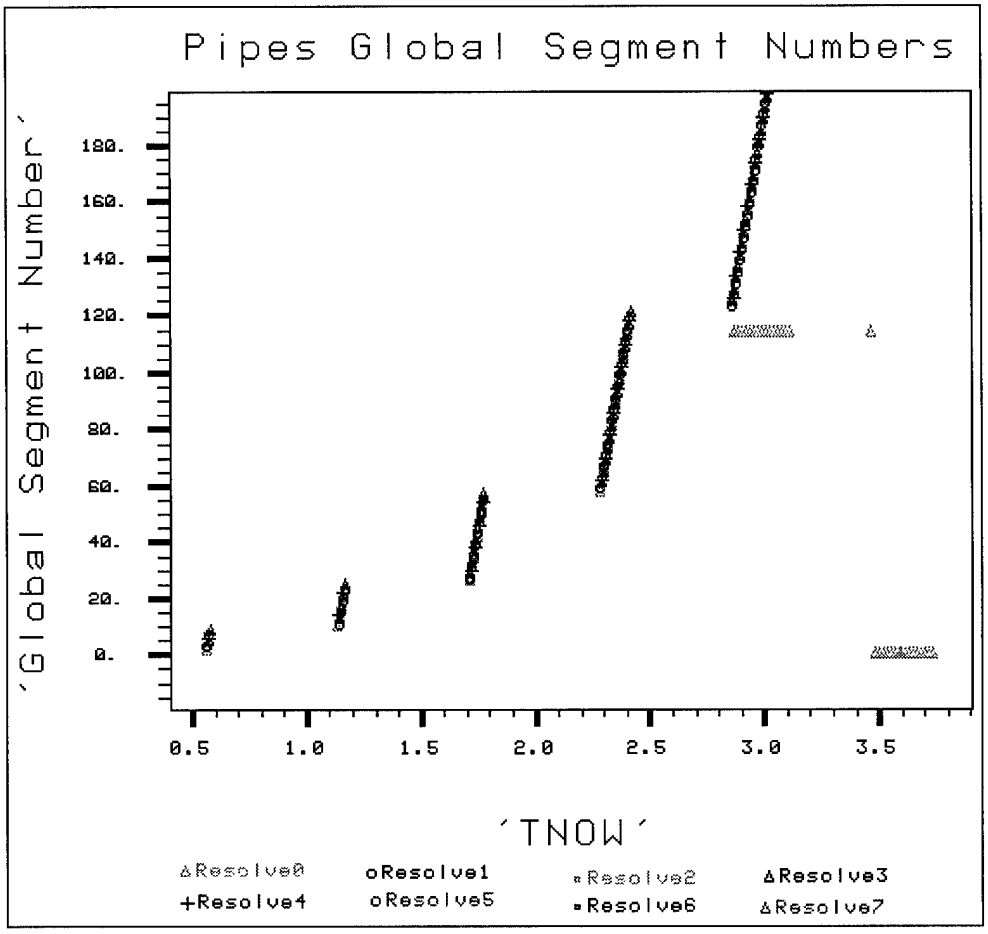

Figure 9 - Single Responsible Pipe Retransmission Delay

This approach could have several advantages. First, segments could be assigned to other pipes if their original pipe's sliding window is causing idle time. In addition, segment duplication means a jammer must corrupt all copies of a segment to prevent successful transmission, increasing the jamming requirement. Pipe cooperation, however, also has drawbacks. Additional functionality and overhead must be added to

the MPP receiving protocol to allow for segments arriving on different (and perhaps multiple) pipes. The transmitting protocol must also be augmented to allow segment reassignment (or duplication). In addition, unnecessary duplicate packets might needlessly take up available bandwidth and other system resources.

4.5 SCPS-TP Summary

One of the strengths of SCPS-TP is that, when operating in Corruption Mode, the time required for data transfer in a jammed environment is almost completely determined by the file size and the total number of segments that are disrupted. Unlike MPP approach, error distribution had little (if any) impact on transfer time. If a jammer can disrupt segments at the very end of a transmission, then idle time is incurred before the transmitting node is made aware of the segment loss. For example, consider the loss of the next-to-last segment. It takes one RTT for the final segment to arrive at the receiving node and a SNACK message to inform the transmitting node of the segment loss. An additional RTT is required to retransmit the segment. To take advantage of this situation, however, jammers must be either lucky or have detailed knowledge of the system and the files being transmitted.

One of the biggest problems of operating SCPS-TP in Corruption Mode in noisy environments, ironically, is the lack of congestion control. When congestion and link outage are assumed as the underlying cause of packet loss, SCPS-TP uses a variation of the standard congestion control mechanisms (TCP Vegas) to avoid congestive collapse. When operating in Corruption Mode, however, there is no means native to the protocol to control network congestion. Any effort to control congestion would have to be

implemented on top of the SCPS-TP protocol, and would force the user to accept the associated overhead. Although this limitation did not effect the simulations run for this thesis because congestion was severely controlled (see Chapter 3), in real system congestion is an important issue to consider.

One possible solution to this problem is to run SCPS-TP in "Congestion Mode". As stated in Chapter 2, the protocol allows hosts to temporarily change from Congestion Mode to Corruption Mode; once the corruption seems to terminate, the protocol switches back to Congestion Mode. While this approach might be very effective if the relatively low noise environment of the G.821 specification and for singular noise bursts, it may not be sufficient in a high-corruption environment caused by an intentional jammer. First, jamming corrupts not only packets being transmitted, but also any ICMP packets sent by the receiver. Once jamming stops, ICMP messages informing the transmitter of corruption losses would be received, but since the jamming would have ceased they would no longer be of any use. As soon as the protocol reenters Congestion Mode, a jammer can again use the methods described in Section 4.4 to reduce the congestion window. Changing the default loss assumption has an additional problem when operating in a jammed environment. As stated in Chapter 2, the TCP-Vegas variant of the Slow Start algorithm increases the sliding window every other RTT as opposed to every RTT as standard TCP does, and modifies when the transmitter enters congestion avoidance. While [BrO94] claim the new algorithm reduces the number of times the sliding window is reduced due to congestion, it provides an added opportunity for a jammer to introduce errors before the sliding window reaches its maximum size.

Once the error rate exceeded 2.5%, SCPS-TP shows a significant advantage over MPP – this advantage increases as the error rate exceeds the 5% limitation of this research. In addition, the small standard deviation allows users of SCPS-TP to develop an accurate transfer time estimation based only on file size and average packet loss rate.

4.6 MPP Summary

When operating in a nominal environment, MPP gives performance that is very comparable to SCPS-TP. The methods described in Section 4.4.3 decrease the advantage held by SCPS-TP. As long as the error percentage (or for that matter the total packets lost due to any cause) remains low, MPP performance is very comparable.

When operating in a jamming environment, the congestion control algorithms of MPP provide an enemy with a great opportunity to disrupt communication. As shown in Section 4.4, once a jammer can inject errors into a channel at will, it is fairly simple to significantly increase transmission times. By corrupting only 5% of the total segments, the transmission requirement can be increased to 514% of the time required by SCPS-TP at the same error rate.

4.7 Conclusion

Both the MPP and SCPS-TP have advantages and disadvantages, and each could be an appropriate choice for improving traditional TCP performance in certain situations. When operating in Corruption Mode, SCPS-TP is capable of transmitting data in very high error environments. The lack of congestion control, however, dictates that it should be used only in system that are likely to suffer corruption and do not suffer from

congestion problems. During normal operations, where the error rates of G.821 apply, MPP is the logical choice for transmitting data. In this case, the advantages provided by running SCPS-TP in Corruption Mode do not overcome the disadvantages of not having native congestion control.

5 CONCLUSIONS

5.1 Protocol Adaptability to Noisy Environments

When operating in a nominal environment, MPP is preferred to SCPS-TP operating in Corruption Mode. Not until the packet loss rate exceeded 2.5% did SCPS-TP show a sustained improvement over MPP. This loss rate is 12.5 times the entire G.821 specification error rate and over 83.3 times the error rate G.821 assigns exclusively to a satellite link.

By making minor adjustments in the MPP model to the transmitting node's initial sliding window in the MPP model, SCPS-TP's performance advantage is weakened. When communication is confronted by enemy jamming and in certain other military environments where packet loss rates may exceed 2.5%, however, the lower transmission times and execution variances may make SCPS-TP the preferred choice.

5.2 Simulation Problems Encountered

While each individual simulation had a relatively short run time, the models did not allow automation of the dozens of simulation runs required for gathering data. It was necessary to manually launch each simulation, examine the simulation results, and adjust simulation variables appropriately for the next simulation.

As long as the number of packet errors being introduced at any one time was smaller than the "effective" sliding window at the moment of injection, the errors were injected very rapidly into the data stream as packets passed through the *Insert Error*

modules. Once the number of errors exceeded the "effective" sliding window, however, the errors could wait a significant amount of time before corrupting a segment. For example, if the "effective" sliding window is 100 and the simulation attempts to inject a burst of 200 errors, only the first 100 errors are quickly assigned to passing segments. All pipes time out, but the sliding window only allows eight packets to be transmitted (and corrupted) at a time. The next 100 errors are inserted eight at a time as the pipes experiences multiple timeouts. This tends to increase the relative impact of large error bursts. To minimize the problem, bursts were never introduced that were larger than the maximum "effective" window size. Since it is known that the MPP can effectively deal with a large number of errors when inserted as a single burst, the impact of not simulating extremely large bursts is minimal.

5.3 Recommendations for Future Work

First, improvements can always be made to simulation models to increase realism. The behavior of a satellite link could be simulated more exactly, with variations in RTT caused by queue size fluctuations, processing delay variations, and other topology changes.

Second, a performance comparison of the two protocols could be performed in terrestrial wireless environment. A smaller DBWP found in ground-based cellular links will cause sliding window delays caused by packet loss to decrease. This may increase the MPP's ability to operate efficiently at higher packet loss rates.

Finally, the RFC2001 version of TCP in the MPP could be replaced with SCPS-TP operating in Congestion Control mode. The MPP would attempt to isolate transmission

problems to a single pipe, while at the same time taking advantage of SCPS-TP (SNACK messages, temporarily adapting to high error or link outage) without sacrificing the ability to adapt to system congestion.

5.4 Conclusion

The ability of SCPS-TP to adapt to short-term changes in the environment (i.e. like link outage and noise bursts) will definitely improve performance. Assuming that all losses are caused by noise, however, limits Corruption Mode's appeal. By sacrificing congestion control, the protocol ensures fast and reliable transmission times but looses the ability to be used in most systems. By retaining congestion control while improving performance by limiting the impact of packet loss, MPP can increase the ability of the armed forces to meet their data delivery requirements.

Appendix A – Detailed Simulation Definition

A.1 Designer BDE

Designer uses a top-down approach to create simulation models. To support this thesis, two models are built – one model simulates the MPP approach to improving TCP performance and the other simulates SCPS-TP. In this appendix, the MPP model is discussed in detail followed by a description of differences between it and SCPS models.

Many variables are required to ensure correct function of the two models. These variables to store data such as sliding window pointer positions and pipe timers. Often, variables are used in several modules in the model. Whenever variables are used in sub-schematics, the same name is used to make understanding the models easier. In this appendix, the function of different variables is explained only in the highest-level schematic where they are first defined.[3]

A.2 MPP

A.2.1 Pipes Top Level

This module is the top-level MPP model (Figure 10). Eight copies of the *Pipe Traffic Generator*, *Pipe ACK Generator*, *and Pipe ACK Resolution* modules are

connected to create eight pipes, each of which can be turned on or off. In later sections, these modules are further explained, but it is important to note that each pipe functions as a separate TCP connection, and is therefore completely independent from all other pipes.

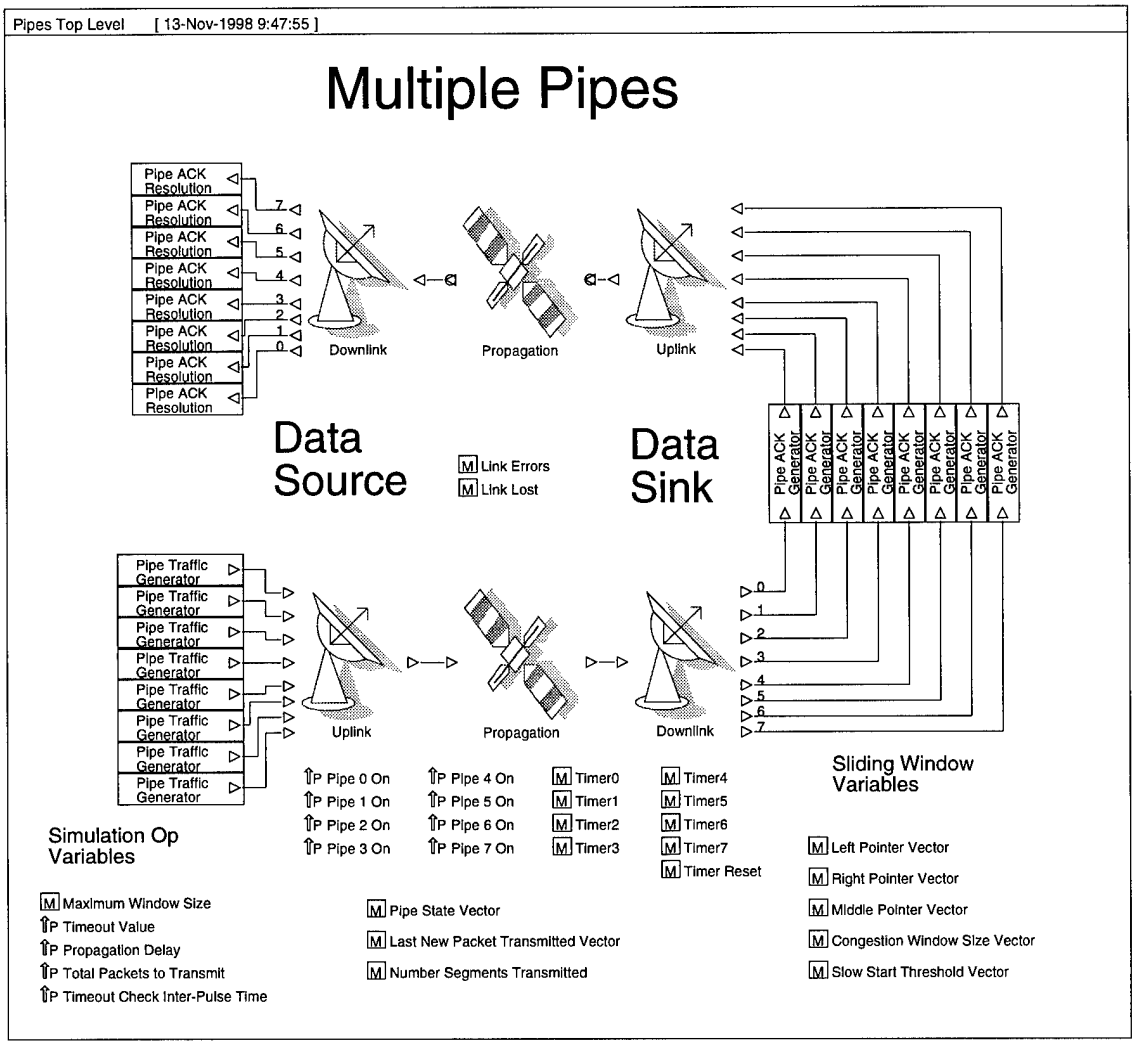

Figure 10 - Pipes Top Level Schematic

---

[3] Modules and variables created by the author for this thesis are italicized, while Designer's native modules and variables are quoted

After model initialization, simulations begin with packets of type *Thesis Packet – Pipe* being generated by each of the eight *Pipe Traffic Generator* modules located at the *Data Source* node. Packets are assigned a *Packet ID* according to the restrictions placed on the pipe by the Go-Back-N Automatic Repeat Request (ARQ) protocol. Packets are then merged into a single data stream by the *Uplink* module, and are transmitted over the communication channel (*Propagation* module). In this channel, they are subject to errors, packet loss (see Chapter 3) and the necessary propagation delay.

The *Downlink* block takes the single data stream, divides packets into their respective pipes, and sends the packets to the eight *Pipe ACK Generator* modules located at the *Data Sink* node. These modules take the *Packet ID* fields of incoming packets and generate the appropriate ACK numbers. ACKs are inserted into outgoing packets, which are sent through *Uplink, Propagation,* and *Downlink* modules back to the *Data Source* node. Upon separation into their individual pipes by the *Downlink* module, the *Pipe ACK Resolution* modules examines the ACK messages and adjusts each pipe's sliding window variables appropriately.

Since all the variables relating to both the sliding window protocol and pipe state are needed in both the *Pipe Traffic Generator* and the *Pipe ACK Resolution* blocks, they are maintained in the top-level schematic. To enable the lower-level modules to operate properly, pointers to these variables are passed down through the schematic hierarchy as required. Variables such as *Propagation Delay* and *Maximum Window Size* are also kept at this level so they can be assigned values when the schematic is simulated. Table 12 gives a summary of the *Pipes Top Level* module's variables.

Table 12 - Pipes Top Level Variables

Variable Name	Type/Range	Initialized Value	Notes
Sliding Window Variables			
Left Pointer Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 0	Contains the left pointer of each pipe's sliding window (Last segment with correct ACK)
Right Pointer Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 1	Contains the right pointer of each pipe's sliding window (Transmit limit without add'l ACKs)
Middle Pointer Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 1	Contains the middle pointer of each pipe's sliding window (Next segment to transmit)
Congestion Window Size Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 1	Contains the current size of the Congestion Windows for each pipe
Slow Start Threshold Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization: 48	Slow Start Threshold for each pipe stage - Initially equal to *Maximum Window Size*
Simulation Op Variables			
Pipe State Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 0	Contains the pipe's state: 0=Slow Start, 1=Congestion Avoidance, 2=Fast Retransmit, 3=Fast Recovery
Last New Packet Transmitted Vector	INT-VECTOR [0..Infinity)	Length:8; Initialization Value: 0	Contains the packet number of the last packet transmitted by each pipe segment
Number Segments Transmitted	INTEGER [0..Infinity)	0	Contains the total segments transmitted so far in simulation - used as a Global Segment Number for first-time transmitted packets
Maximum Window Size	INTEGER [0..Infinity)	Set At Run Time	The maximum size any congestion window can take - set at run time - equal to receiving node's advertisement
Timeout Value	REAL [0..Infinity)	Set At Run Time	Parameter when Timeout is triggered = 0.5 for LEO and 1.0 for GEO
Propagation Delay	REAL [0..Infinity)	Set At Run Time	Propagation Delay between transmitter and receiver - set at run time
Total Packets to Transmit	INTEGER [0..Infinity)	Set At Run Time	Total number of packets to transmit - limits simulation execution
Timeout Check Inter-Pulse Time	REAL [0..Infinity)	0.1	Time interval between checks for packet timeout
Link Loss	INTEGER [0..Infinity)	0	Number of packet losses that will inserted by the *Propagation* modules
Link Errors	INTEGER [0..Infinity)	0	Number of errors that will inserted by the *Propagation* modules
Pipe 0 On to Pipe 7 On	INTEGER [0..1]	Set At Run Time	Activates each pipe segment - 0=Pipe Inactive, 1=Pipe Active
Timer0 to Timer7	REAL-VECTOR [0..Infinity)	Length:3000 Initialization Value: 0	Time packets are transmitted - each pipe has own timer
Timer Reset	REAL-VECTOR [0..Infinity)	Length:3000 Initialization Value: 0	Reset all timers as result of retransmission

A.2.2 Pipe Traffic Generator

The *Pipe Traffic Generator* module (Figure 11) is a sub-component of the *Pipes Top Level* module. The primary purpose of this module is to create and transmit packets that properly follow the TCP protocols discussed in Chapter 2.

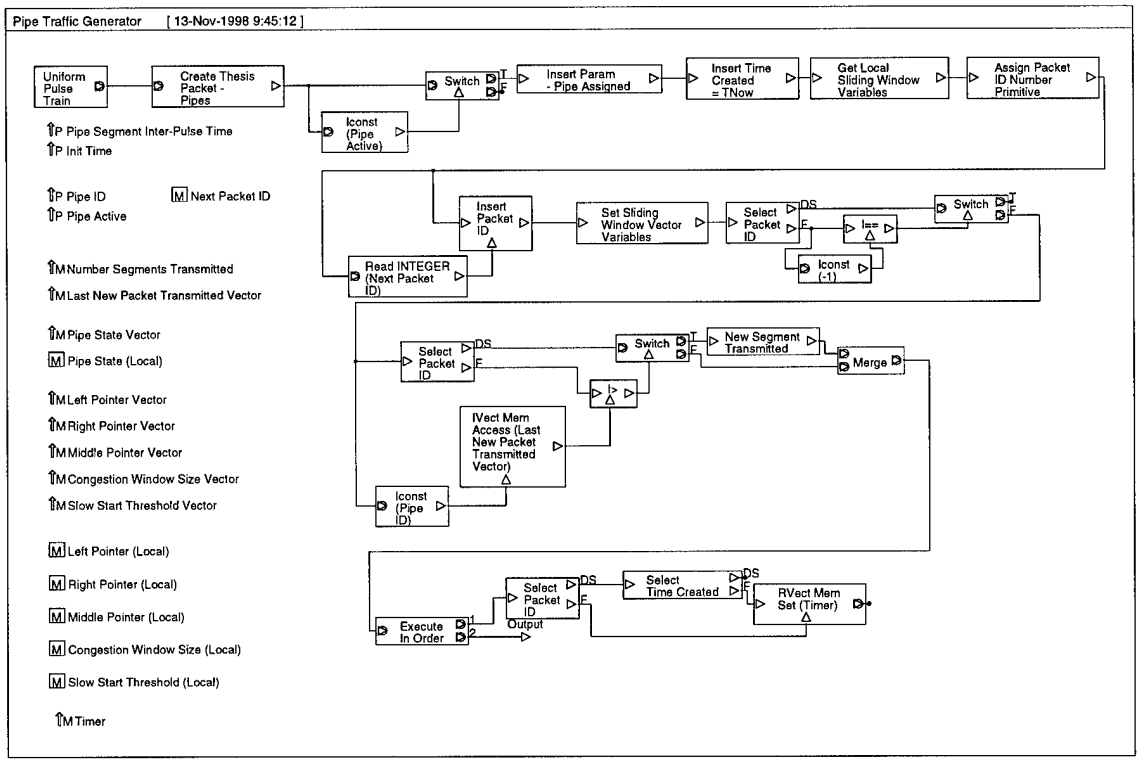

Figure 11 - Pipe Traffic Generator Schematic

The traffic generator creates a series of pulses, which in turn creates a series of packets. The traffic generator shown is a "Uniform Pulse Train" module, which simulates standard size TCP packets common in applications like FTP. To simulate different traffic scenarios, the traffic generator module can easily be replaced. Packets are then routed into a switch and passed along if the *Pipe Active* variable indicates that the particular pipe is turned on – otherwise packets are destroyed.

Once a packet passes through the switch, the *Time Created* and *Pipe Assigned* fields are updated. Packets next enter the *Get Local Sliding Window Variables* module, which updates local copies of the pipe's portion of the top-level sliding window variables. The next module *(Assign Packet ID Number Primitive)* is a Designer primitive created exclusively for this thesis. The primitive generates the appropriate *Packet ID* and inserts the proper value into the *Next Packet ID variable* – additional information about this primitive can be found in Appendix B. Packets then enter the *Set Sliding Window Vector Variables* module, which takes the local copies of the sliding window variables (which may have been altered by the primitive) and updates the top-level sliding window vectors.

Once the *Packet ID* is inserted into the packet data structure and the sliding window variables are updated, the simulation checks to see if the packet should be transmitted. As stated in Chapter 2, a sliding window can only allow a certain number of segments to be outstanding (no ACK received). Upon creation, all *Packet ID* variables are initialized to –1, and this value is used as an error condition indicating that the sliding window cannot support additional packets. If the *Assign Packet ID Number Primitive* module determines that the packet should not be sent due to window limitations, the value is not changed and the packet is destroyed.

The next step determines if the packet is being transmitted for the first time or if it is a retransmission of a previous packet. If the *Packet ID* of the incoming packet is less than the pipe's value in *Last New Packet Transmitted Vector*, then the packet is being retransmitted; a larger *Packet ID* indicates a packet's first transmission. If the packet is a

new transmission, it is routed into the *New Segment Transmitted* block; otherwise the packet skips this block. Finally, the appropriate segment timer (based on *Packet ID*) is set to the packet's *Time Created* field.

Most of the top-level sliding window variables are passed through the *Pipe Traffic Generator* module to the *Assign Packet ID Number Primitive* module. Additional variables used in this module are found in Table 13.

Table 13 - Pipe Traffic Generator Variables

Variable Name	Type/Range	Initialized Value	Notes
Pipe Segment Inter-Pulse Time	REAL [0..1]	Set At Run Time	Control the creation of packets - set to keep pipes full
Init Time	REAL [0..Infinity)	Set At Run Time	Time each pipe begins transmitting segments - each pipe init time staggered by pipe service rate
Pipe ID	INTEGER [0..Infinity)	0 to 7	Gives a ID number for each pipe - used as an index into INT-VECTOR sliding window (and other) variables
Pipe Active	INTEGER [0..Infinity)	Set At Run Time	Determines if a specific pipe is turned on - 0=Pipe Inactive, 1=Pipe Active
Next Packet ID	INTEGER [0..Infinity)	0	Temp variable holding the value that will be inserted into *Packet ID* field
Pipe State (Local)	INTEGER [0..Infinity)	0	Local copy specific pipes value in the *Pipe State Vector*
Left Pointer (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Left Pointer Vector*
Right Pointer (Local)	INTEGER [0..Infinity)	1	Local copy of specific pipe's value in the *Right Pointer Vector*
Middle Pointer (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Middle Pointer Vector*
Congestion Window Size (Local)	INTEGER [0..Infinity)	48	Local copy of specific pipe's value in the *Congestion Window Size Vector*
Slow Start Threshold (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Slow Start Threshold Vector*

A.2.3 Pipe ACK Generator

The *Pipe ACK Generator* module (Figure 12) is a sub-component of the *Pipes Top Level* module. The primary purpose of this module is to generate ACK numbers based on

the *Packet ID* field of incoming packets and send the ACK packets to the *Data Source* node.

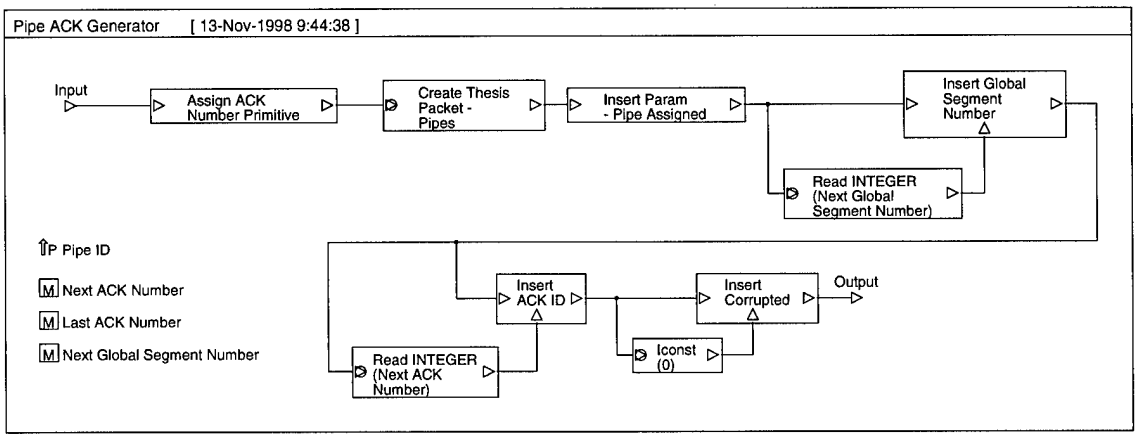

Figure 12 - Pipe ACK Generator Schematic

The TCP specification allows flexibility in when ACKs are generated. Some implementations generate ACK messages with every packet arrival, while others implement delayed ACKs [Com95]. For simplicity, in this thesis ACK messages are generated whenever an undamaged packet arrives at the *Pipe ACK Generator.*

The *Assign ACK Number Primitive* module takes packets arriving from the *Data Source* node, examines the *Packet ID*, and calculates *the Next ACK Number* variable. A detailed examination of the primitive is found in Appendix B.

Since the model is only concerned with traffic flowing from the *Data Source* to *Data Sink* (traffic in the reverse direction only provides a vehicle for delivering ACK messages), many fields in the newly created *Thesis Packet – Pipes* packet are left at their initialized values. First, the proper *Pipe Assigned* is inserted. Next, the last *Global Segment Number* that correctly arrived from the *Data Source* node is inserted; this allows the simulation to be terminated by the *Pipe ACK Resolution* modules after a pre-defined

number of packets have safely arrive at the *Data Sink* node. The value of *Next ACK Number* is then inserted into the *ACK ID* field of the outgoing packet. Table 14 lists the local variables used by each *Pipe ACK Generator* module.

Table 14 - Pipe ACK Generator Variables

Variable Name	Type/Range	Initialized Value	Notes
Pipe ID	INTEGER [0..Infinity)	0 .. 7	Pipe identifier to route packets to the proper pipe
Next ACK Number	INTEGER [0..Infinity)	-1	Temp variable holding the value that will be inserted into *ACK ID* field. -1 used for error condition
Last ACK Number	INTEGER [0..Infinity)	-1	ACK number of the past packet where ACK sent back to *Data Source*. -1 used for error condition
Next Global Segment Number	INTEGER [0..Infinity)	-1	Last Global Segment number that arrive safely. -1 used for error condition

A.2.4 Pipe ACK Resolution

The *Pipe ACK Resolution* module (Figure 13) is a sub-component of the *Pipes Top Level* module. The primary purpose of this module is to adjust the sliding window variables in response to ACK messages and to terminate the simulation.

When packets first enter the module, it is necessary to determine if the incoming packet is the last one that will be received in a particular pipe. If a packet's *Global Segment Number* is greater than or equal to *Total Packets to Transmit*, then this pipe can be shut down – the switch routes the packet to the "Terminate Simulation" module. The "Terminate Simulation" module is set to act in "Cooperate" mode, meaning that the simulation is terminated once the "Terminate Simulation" modules in all the pipes are activated. This approach resolves the problem of a packet containing a *Global Segment*

Number equal to *Total Packets to Transmit* arriving while another pipe is retransmitting lost packets. In this situation, segments being re-transmitted would be lost. By forcing the "Terminate Simulation" module in all eight pipes to activate before the simulation is actually terminated, the model ensures that all packets with a *Global Segment Number* less than *Total Packets to Transmit* are successfully received by the *Data Sink.*

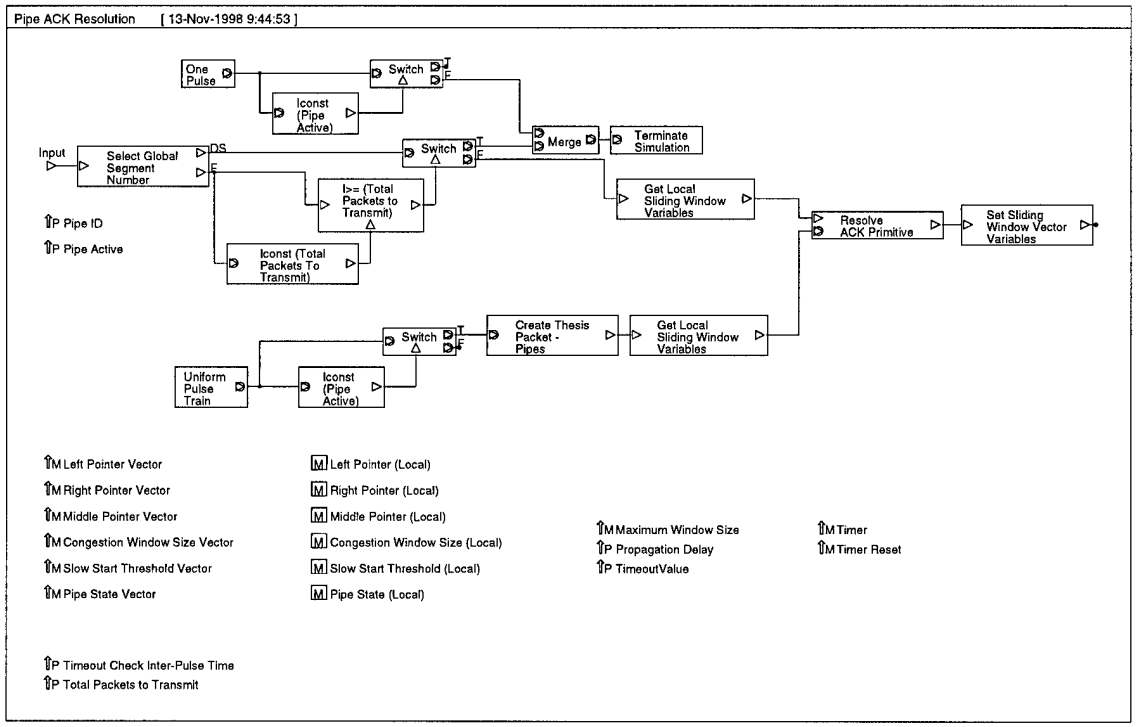

Figure 13 - Pipe ACK Resolution Schematic

If a packet is not the last one in a pipe, then it is routed to logic that adjusts the pipe's sliding window. Packets are sent to the *Resolve ACK Primitive* module (see Appendix B), which adjusts the pipe's sliding window variables appropriately. Like the *Pipe Traffic Generator* module, *Get Local Sliding Window Variables* and *Set Sliding Window Vector Variables* modules are used.

In order to activate the "Terminate Simulation" modules in inactive pipes (which will never receive an incoming packet), a one-shot pulse is activated at the beginning of the simulation to activate the necessary "Terminate Simulation" modules.

The *Resolve ACK Primitive* also checks for timeouts. Whenever a packet arrives, all segments in the current sliding window are checked for a timeout. Since the *Pipe ACK Generator* modules generate ACK messages only in response to undamaged packets, it is possible that all packets in the sliding window could be corrupted and consequently no ACKs generated. To ensure timeouts still happen, a Designer traffic generator is added to trigger periodic timeout checks. Table 15 shows the local variables used by each *Pipe ACK Resolution* module.

Table 15 - Pipe ACK Resolution Variables

Variable Name	Type/Range	Initialized Value	Notes
Pipe ID	INTEGER [0..Infinity)	0 to 7	Gives a ID number for each pipe - used as an index into INT-VECTOR sliding window (and other) variables
Pipe Active	INTEGER [0..Infinity)	Set At Run Time	Determines if a specific pipe is turned on - 0=Pipe Inactive, 1=Pipe Active
Left Pointer (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Left Pointer Vector*
Right Pointer (Local)	INTEGER [0..Infinity)	1	Local copy of specific pipe's value in the *Right Pointer Vector*
Middle Pointer (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Middle Pointer Vector*
Congestion Window Size (Local)	INTEGER [0..Infinity)	48	Local copy of specific pipe's value in the *Congestion Window Size Vector*
Slow Start Threshold (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipe's value in the *Slow Start Threshold Vector*
Pipe State (Local)	INTEGER [0..Infinity)	0	Local copy of specific pipes value in the *Pipe State Vector*

A.2.5 New Segment Transmitted

The *New Segment Transmitted* module (Figure 14) is a sub-component of the *Pipe Traffic Generator* module. The primary purpose of this module is to update the variables controlling the *Global Segment Number* variable (which is used to terminate the simulations).

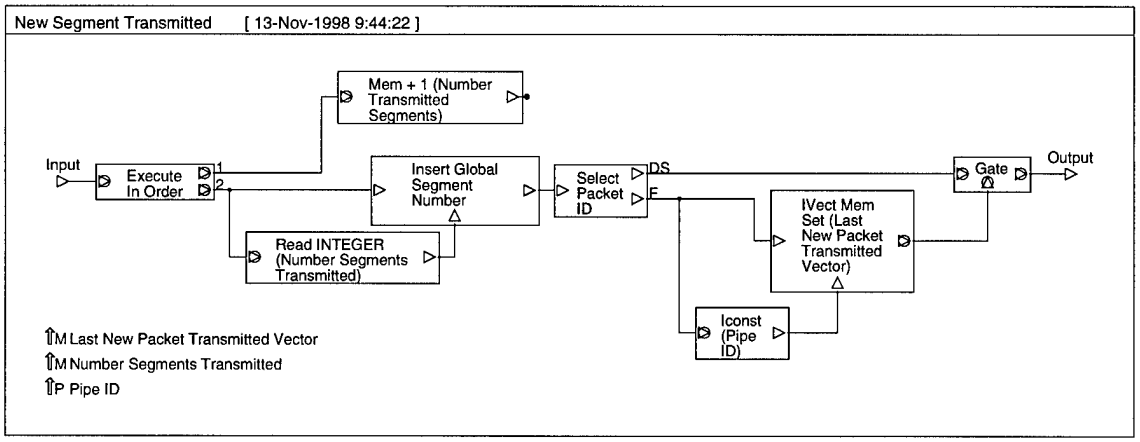

Figure 14 - New Segment Transmitted Schematic

When packets enter this module, the *Number Segments Transmitted* variable is incremented, and is subsequently inserted into the packet's *Global Segment Number*. To ensure that the variable is accessed in the correct order, an "Execute In Order" module is used.

Next, the *Last New Packet Transmitted* variable is updated with the *Packet ID* of the incoming segment. This variable allows the *Pipe Traffic Generator* module to determine if a packet is being transmitted for the first time. Since Designer blocks are used in this module (as opposed to a primitive), it is not necessary to make a local copy of the pipe's value in the *Last New Packet Transmitted Vector*.

A.2.6 Get Local Sliding Window Variables

The *Get Local Sliding Window Variables* module (Figure 15) is a sub-component of the *Pipe Traffic Generator* and *Pipe ACK Resolution* modules. The primary purpose of this module is to create copies of an individual pipe's portion of the sliding window vector variables (i.e. *Right Pointer Vector*, *Congestion Window Size Vector*, etc.).

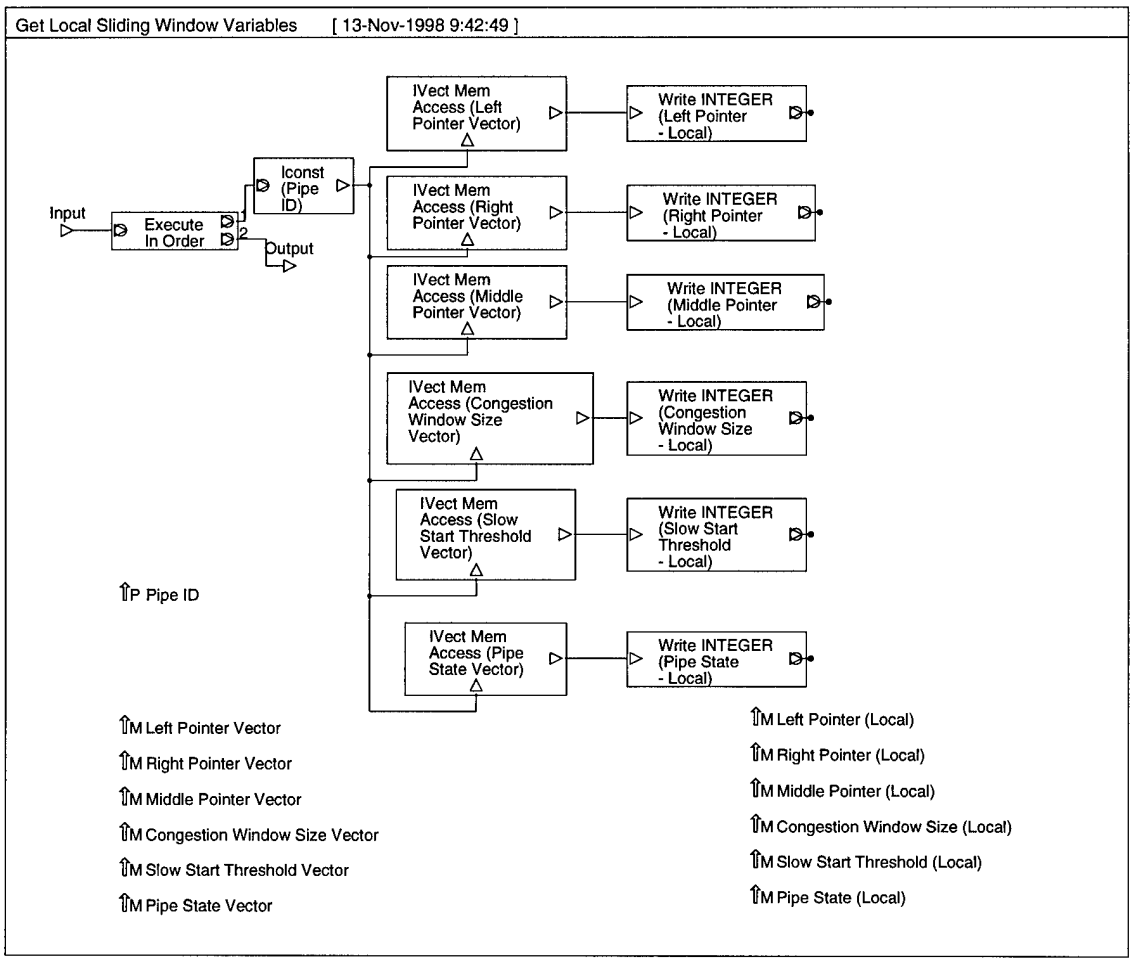

Figure 15 - Get Local Sliding Window Variables Schematic

This module greatly simplifies the primitive's C++ code, since the primitives only have to reference individual " INTEGER" values instead of "INT-VECTOR" variables.

A.2.7 Set Sliding Window Vector Variables

The *Set Sliding Window Vector Variables* module (Figure 16) is a sub-component of the *Pipe Traffic Generator* and *Pipe ACK Resolution* modules. The primary purpose of this module is insert the local copies created by the *Get Local Sliding Window Variables* module back into the top-level sliding window vector variables.

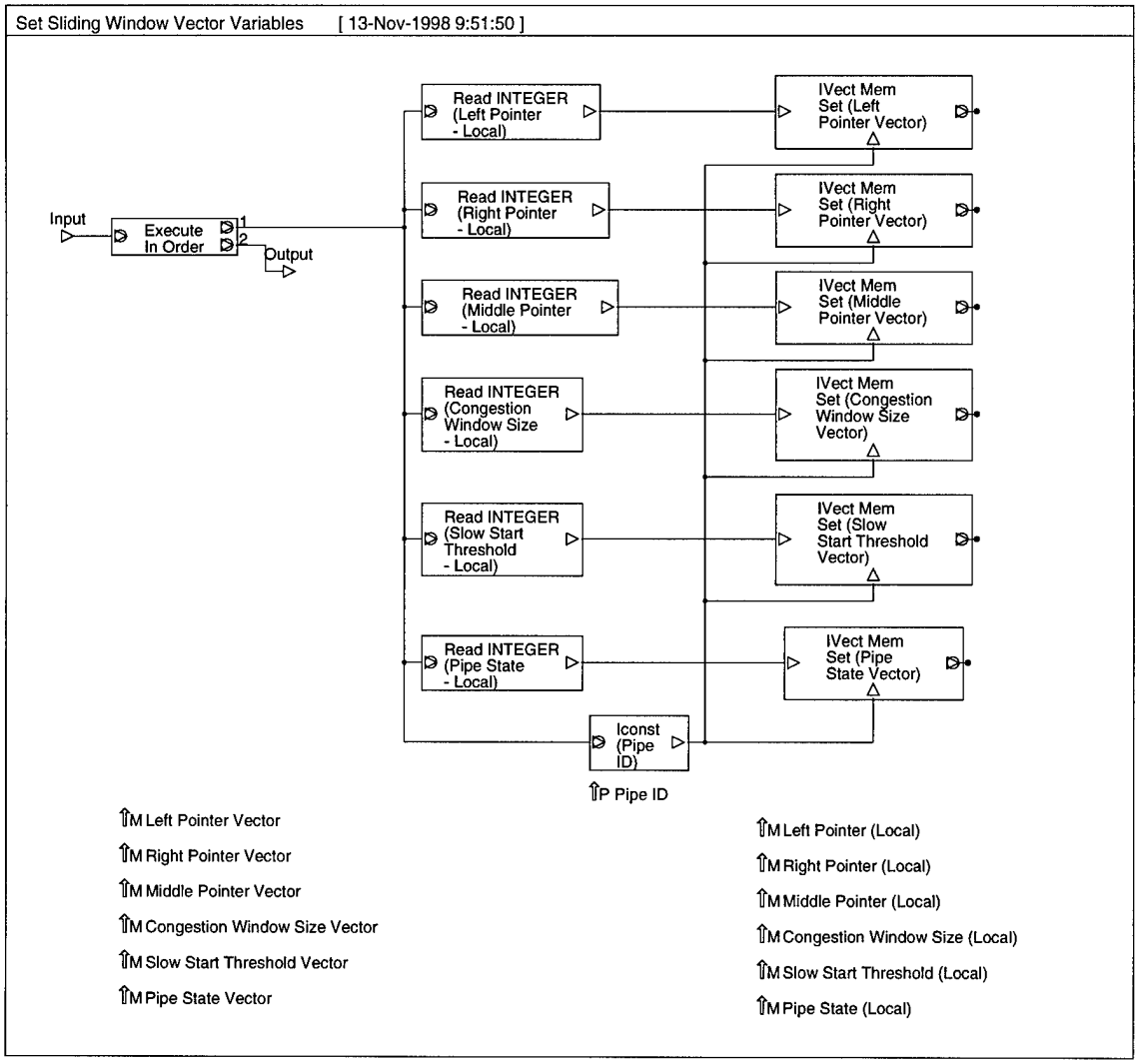

Figure 16 - Set Sliding Window Vector Variables Schematic

This module complements the *Get Local Sliding Window Variables* modules. The local copies of the variables are inserted into the top-level variables so they can be shared between *the Pipe Traffic Generator* and *Pipe ACK Resolution modules.*

A.2.8 Error (Timeout) Source[4]

The *Error (Timeout) Source* (Figure 17) module is a sub-component of the *Propagation* module. The primary purpose of this module is to generate errors (timeouts) that are introduced into packets.

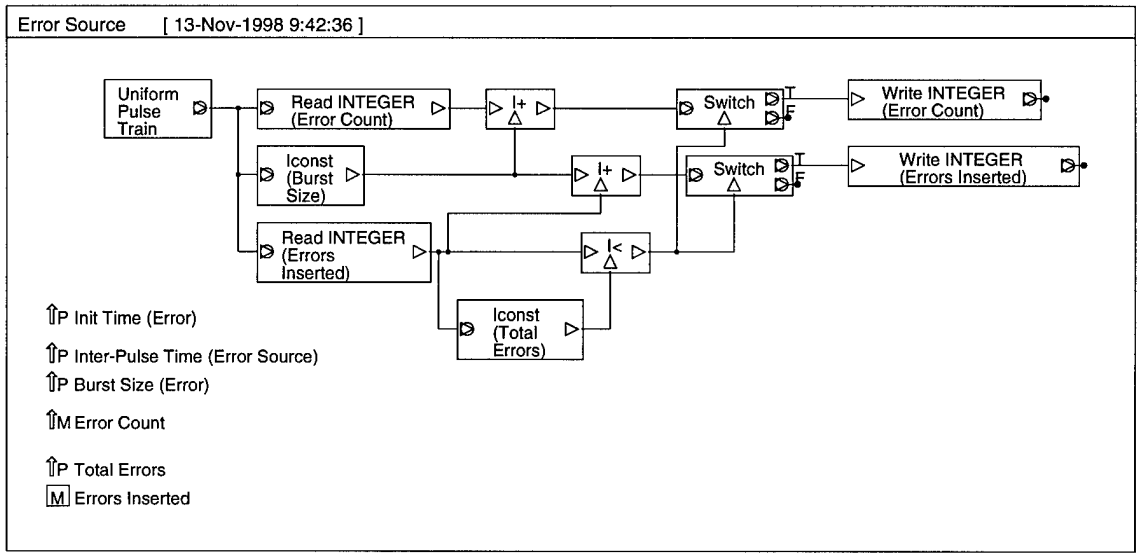

Figure 17 - Error Source Schematic

Originally, a "Bursty Pulse Train" was used to increment the *Error Count* variable. Once the first few simulations were run, however, it was discovered that this traffic generator did not provide enough control to ensure the correct number of errors were

---

[4] The *Error Source* and *Timeout Source* modules are essentially the same structure – the only difference is that *Error Source* accesses the *Error Count* variable, while *Timeout Source* accesses the *Timeout* variable. The discussion of the *Error Source* module therefore also applies to the *Timeout Source* module.

inserted. Consequently, the module was modified. A "Uniform Pulse Train" replaced the "Burst Pulse Train" to give control of the interval between error bursts. If the number of errors inserted up to any point in a simulation is less than the total errors to be inserted, then *Error Count* is incremented by number of errors in the error burst; otherwise the errors are not counted. The variables used by this module is found in Table 16.

Table 16 - Error Source Variables

Variable Name	Type/Range	Initialized Value	Notes
Init Time (Error Source)	REAL [0..Infinity)	Set At Run Time	Time the first burst of errors is inserted
Inter-Pulse Time (Error Source)	REAL [0..Infinity)	Set At Run Time	Interval between error bursts
Burst Size (Error)	INTEGER [0..Infinity)	Set At Run Time	Number of errors inserted each time the traffic generator sends a pulse
Error Count	INTEGER [0..Infinity)	N/A	Used to access top-level error variables
Total Errors	INTEGER [0..Infinity)	Set At Run Time	Total number of errors to insert
Errors Inserted	INTEGER [0..Infinity)	0	Total errors inserted up to current simulation time - local variable

A.2.9 Insert Error/Timeout[5]

The *Insert Error (Timeout)* (Figure 18) module is a sub-component of the *Propagation* module. The primary purpose of this module is to insert errors created by the *Error (Timeout) Source* modules into the data stream.

---

[5] The *Insert Error* and *Insert Timeout* modules are essentially the same structure – the only difference is that *Insert Error* accesses the *Error Count* variable, while *Insert Timeout* accesses the *Timeout* variable. The discussion of the *Insert Error* module therefor also applies to the *Insert Timeout* module.

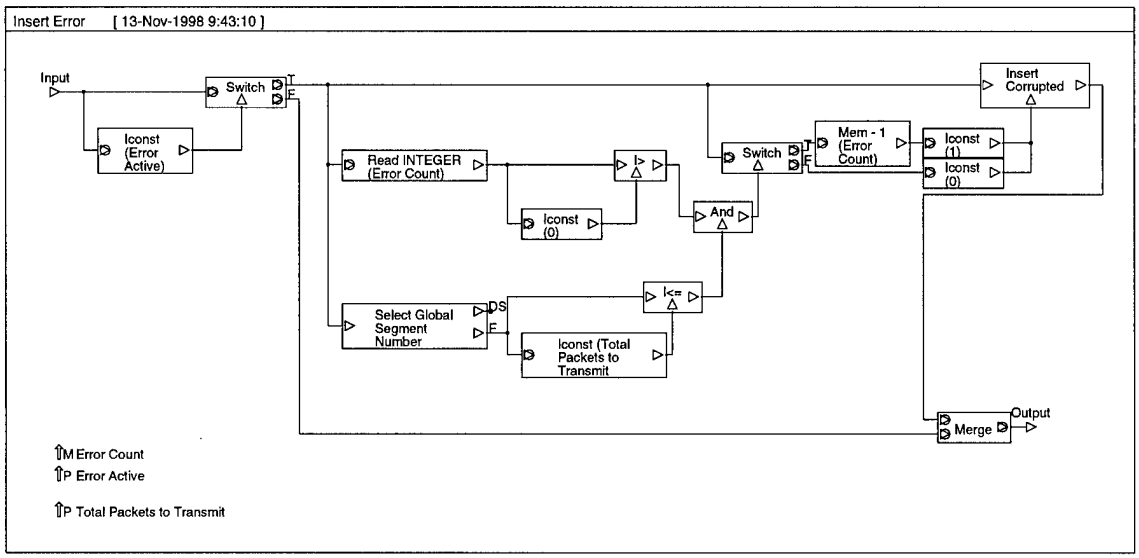

Figure 18 - Insert Error Schematic

As stated in Chapter 3, the need to realistically model errors and timeouts must be balanced against the ability to control where errors are inserted. To gain this control, the *Error Active* variable allows packets to bypass the logic controlling the insertion of errors/timeouts.

When a packet enters the module and is past the switch controlled by the *Error Active* variable, the *Error Count* or *Timeout Count* memory is accessed. If the value is greater than zero, then the packet needs to be corrupted or set to time out. The "I >" module feeds a switch that controls which value will be inserted into the packet's *Corrupted* or *Timeout* field. If logic TRUE (meaning the packet is corrupted or will time out) is inserted into the packet, then the *Error Count* or *Timeout Count* memory is decremented. Logic is added to ensure that errors are not inserted into packets that have a *Global Segment Number* larger than *Total Packets to Transmit*.

A.2.10 Propagation

The *Propagation* module (Figure 19) is a sub-component of the *Pipes Top Level* module. The primary purpose of this module is to simulate the physical transmission of packets between the *Data Source* and the *Data Sink* nodes and insert errors and timeouts.

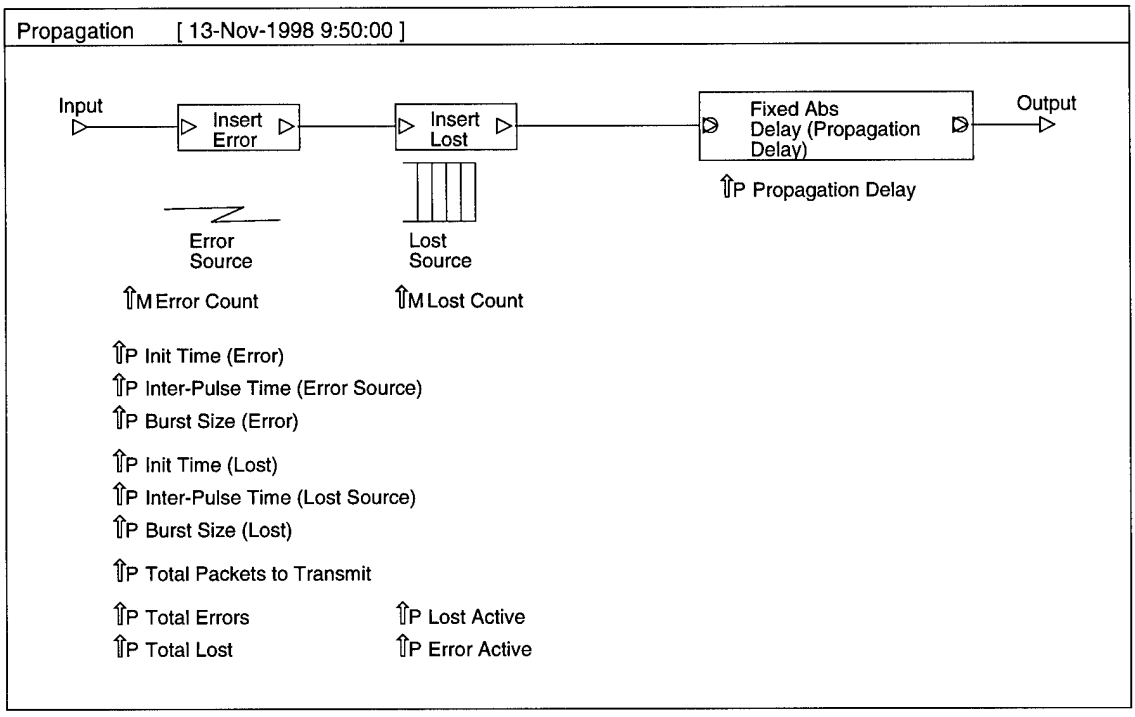

Figure 19 - Propagation Schematic

There are two version of the Propagation module. The module that carries packets from the Pipe ACK Generator to the Pipe ACK Resolution module has the Timeout Source, Insert Timeout, Error Source, and Insert Error modules removed. The Propagation module between Pipe Traffic Generator and Pipe ACK Generator modules is exactly as shown in Figure 19. Again, the reasoning for timeouts and errors is found in Chapter 3.

A.2.11 Uplink/Downlink

The *Uplink* and *Downlink* modules are sub-components of the *Pipes Top Level* module. The primary purpose of these modules is to merge the eight pipe data streams into a single data stream (*Uplink*) or split the single data stream back into the eight pipe data streams (*Downlink*).

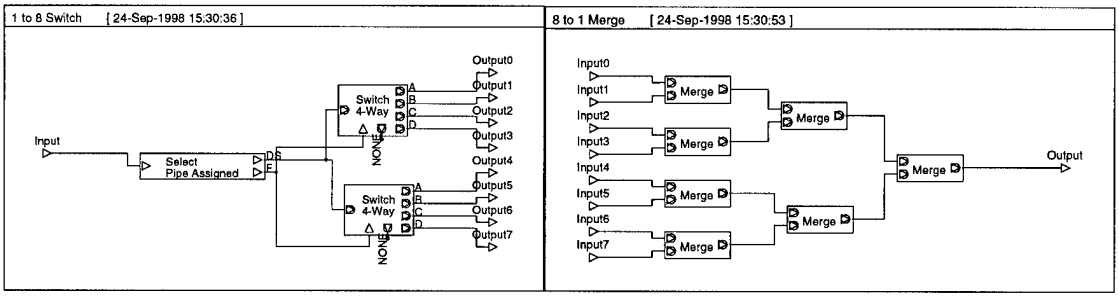

Figure 20 - Downlink (Left) and Uplink (Right) Schematics

The *Uplink* uses a series of "Merge" blocks to create the single data stream, while the *Downlink* block uses the *Pipe Assigned* field of the incoming packet to switch between the eight output streams using two "Switch – 4 Way" modules.

The eight pipe data streams are Time-Division Multiplexed into the single stream travelling between the *Data Source* and *Data Sink*, which is how the data streams would likely be in a real system. This effect is caused by Designer's discrete-event simulation engine.

A.2.12 Thesis Packet – Pipes

To create the simulation, one of the first steps was to create a "Composite" data structure that could hold all the data required in the simulation. The data structure, called

Thesis Packet – Pipes, was refined as the model matured. The final fields of the structure are in Table 17.

Table 17 - Thesis Packet - Pipes Fields

Variable Name	Type/Range	Initialized Value	Notes
Time Created	REAL [0..Infinity)	0.0	Simulation time when a packet is created in the *Pipe Traffic Generator* module
Pipe Assigned	INTEGER [0..7]	0.0	Pipe that a particular packet is assigned to
Packet ID	INTEGER [-1..Infinity)	-1	indicates sliding window can't support additional packets
ACK ID	INTEGER [0..Infinity)	0	ACK number that is sent from the *Data Sink* back to the *Data Source*
Global Segment Number	INTEGER [0..Infinity)	0	Global number that keeps track to where a packet fits in the overall information flow - used to terminate simulation
Corrupted	INTEGER [0..1]	0	Indication whether a packet has been corrupted - 0=Not Corrupted, 1=Corrupted
Timeout	INTEGER [0..1]	0	Indication whether a packet will Timeout - 0= Not Timeout, 1=Timeout
Congestion Window Size	INTEGER [0,,Infinity)	0	Congestion Window size at various time in simulation - used for debugging model

A.3 SCPS-TP BDE

A.3.1 SCPS Top Level

As can be seen in Figure 21, the *SCPS Top Level* model is very similar to the *Pipes Top Level* model, so a detailed examination of all its sub-components is redundant. There are, however, some differences that should be examined. The model in some ways is simpler because of SCPS-TP's use of a leaky-bucket traffic source (see Chapter 2). In addition, experience building the MPP model allowed improvements to be incorporated into the SCPS-TP model. In addition, the primitives used in SCPS-TP perform some of the functionality that in the MPP model was performed using Designer modules. While

this implementation decision complicated the code of the SCPS-TP primitives, it made for a cleaner model.

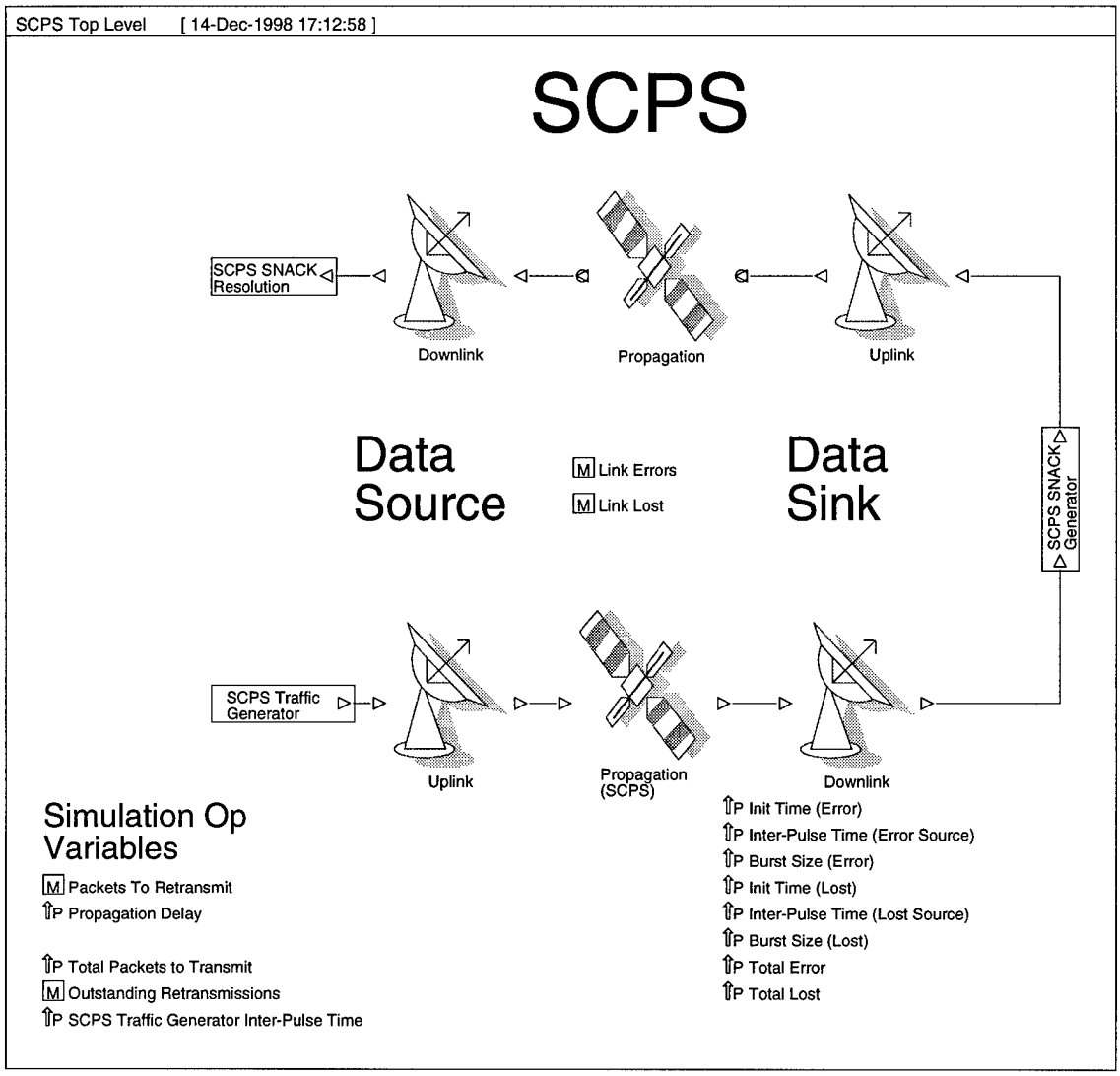

Figure 21 - SCPS Top Level Schematic

There are several changes to the variable types (and names). Since there is only a single "pipe" in the SCPS-TP model, there is no need to use vector-type variables, and all the variables relating to pipes and local copies of sliding window variables are deleted. All the top-level variables except for those relating to "Uniform Pulse Train" modules are

"INTEGER" type, and have [0..Infinity) bounds. Since vectors are not used, there is no need to have an SCPS-TP equivalent to the *Get Local Sliding Window Variables* and *Set Sliding Window Vector Variables* modules.

The largest technical change was in the primitives used by the model. Entirely new primitives are needed in the *SCPS Traffic Generator* module, the *SCPS SNACK Generator* module, and the *SCPS SNACK Resolution* module to implement the Selective Negative Acknowledgement protocol. An additional variable is included in the top-level schematic to convey data between the *SCPS Traffic Generator* module and the *SCPS SNACK Resolution* module. The *Packets to Retransmit* and *Outstanding Retransmissions* variables are used by the primitives in these three modules to control the retransmission of segments that are corrupted or time out.

In the model, exact identification of which packets are corrupted is not necessary. Instead, the model looks at the number of packets that are corrupted or time out. After a series of corrupted packets, the first uncorrupted packet carries the number of missing packets to the *SCPS SNACK Resolution* module. This module accesses the *Outstanding Retransmissions* and *Packets to Retransmit* variables, which the *SCPS-TP Traffic Generator* uses to create the appropriate number of duplicate packets. Additional details of the model's retransmission strategy are found in the discussion of SCPS-TP primitives (Appendix B).

Figure 22 to Figure 24 are the schematics for the rest of the SCPS-TP model that are significantly different from their MPP counterparts, and are presented without additional comment in the interest of completeness.

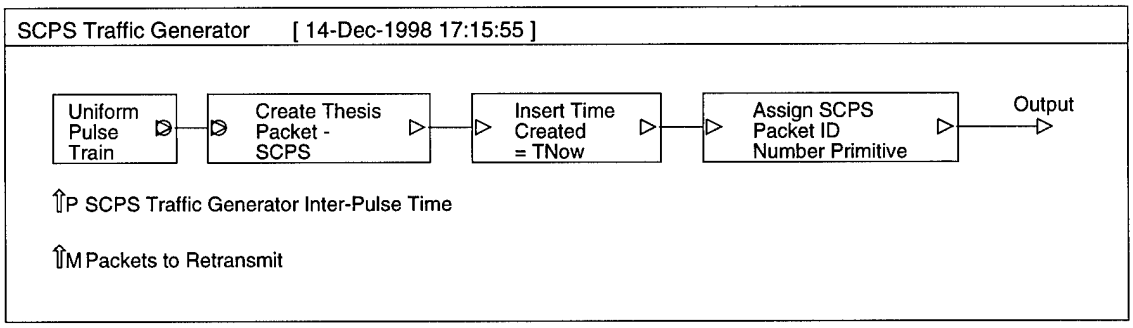

Figure 22 - SCPS Traffic Generator Schematic

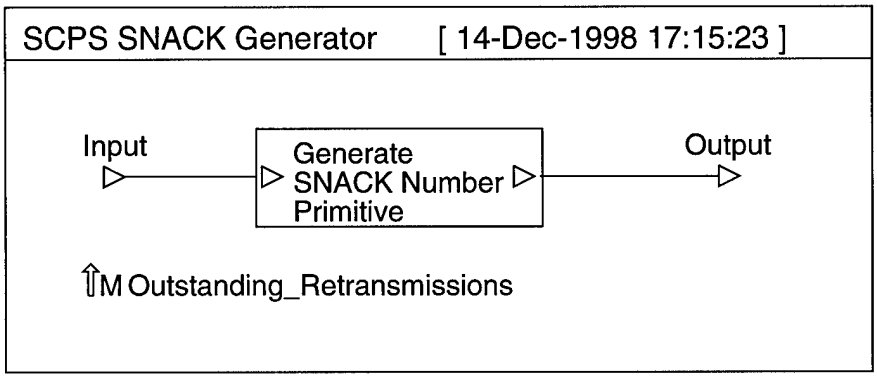

Figure 23 - SCPS SNACK Generator Schematic

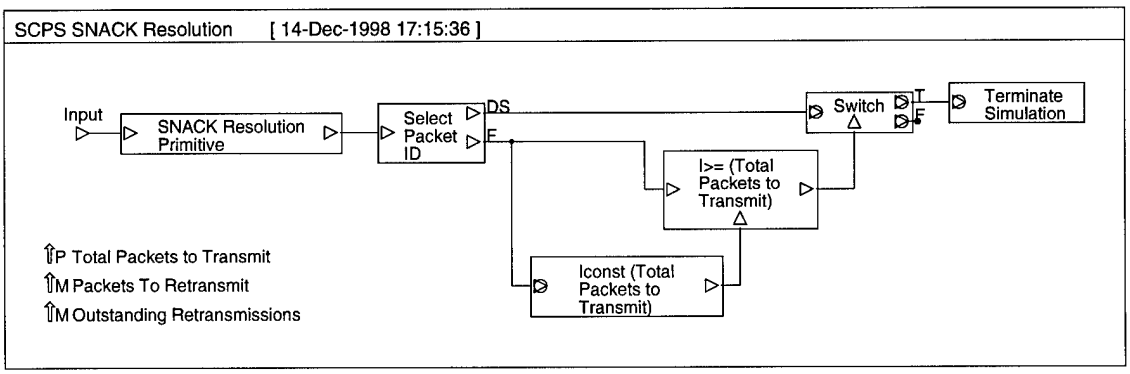

Figure 24 - SCPS SNACK Resolution Schematic

Appendix B – DESIGNER Primitives Descriptions

B.1 Designer Primitives

In Designer, certain functions cannot easily be implemented using the program's native modules. In these situations, Designer allows users to write C++ code that work with memory variables, parameters, and ports. For this thesis, six Designer primitives were required to implement standard TCP and SCPS-TP. In this appendix, flowcharts are used to help explain the function of all the primitives. For completeness, a complete listing of all the primitive's code is included in Appendix C.

B.2 MPP Primitives

B.2.1 Assign Packet ID Number Primitive

The function of this primitive is very straightforward, as seen in Figure 25. When a packet arrives, the primitive examines whether the sliding window can support another outstanding packet. If it can, the *Middle Pointer* becomes the *Packet ID* of the outgoing packet, and the *Middle Pointer* is incremented. If the sliding window cannot support another packet, -1 is used as an error condition by the *Pipes Traffic Generator* to destroy the packet. Since the *Right Pointer* indicates the last packet that can be transmitted without additional ACKs, there are times when the *Middle Pointer* is actually one greater than the *Right Pointer*.

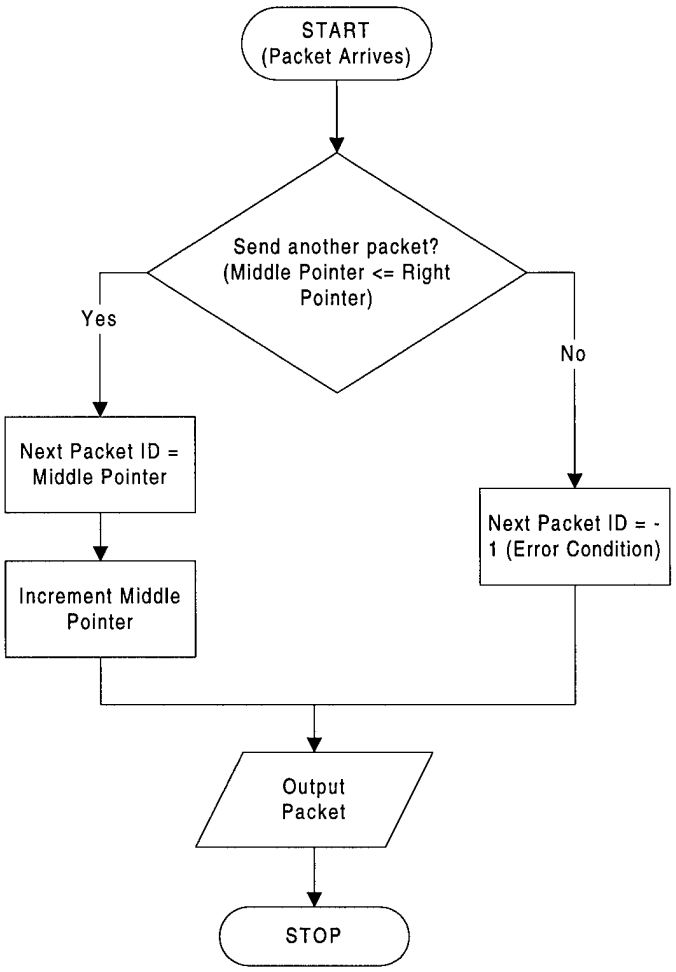

Figure 25 - Assign Packet ID Number Primitive (MPP)

B.2.2 Assign ACK Number Primitive

The *Assign ACK Number Primitive* (Figure 26) takes incoming packets and determines what kind of an ACK message should be sent (if any). If the arriving packet's *Corrupted* or *Lost* fields are set, then no ACK is generated. If a packet is undamaged, an ACK is generated. If the arriving packet's *Packet ID* is one greater than the last uncorrupted packet's value, then the ACK value is set to the new *Packet ID* and *Global*

Segment Number are assigned to the outgoing ACK packet; otherwise a duplicate ACK is sent. As noted in Chapter 3, an ACK is sent for every undamaged packet.

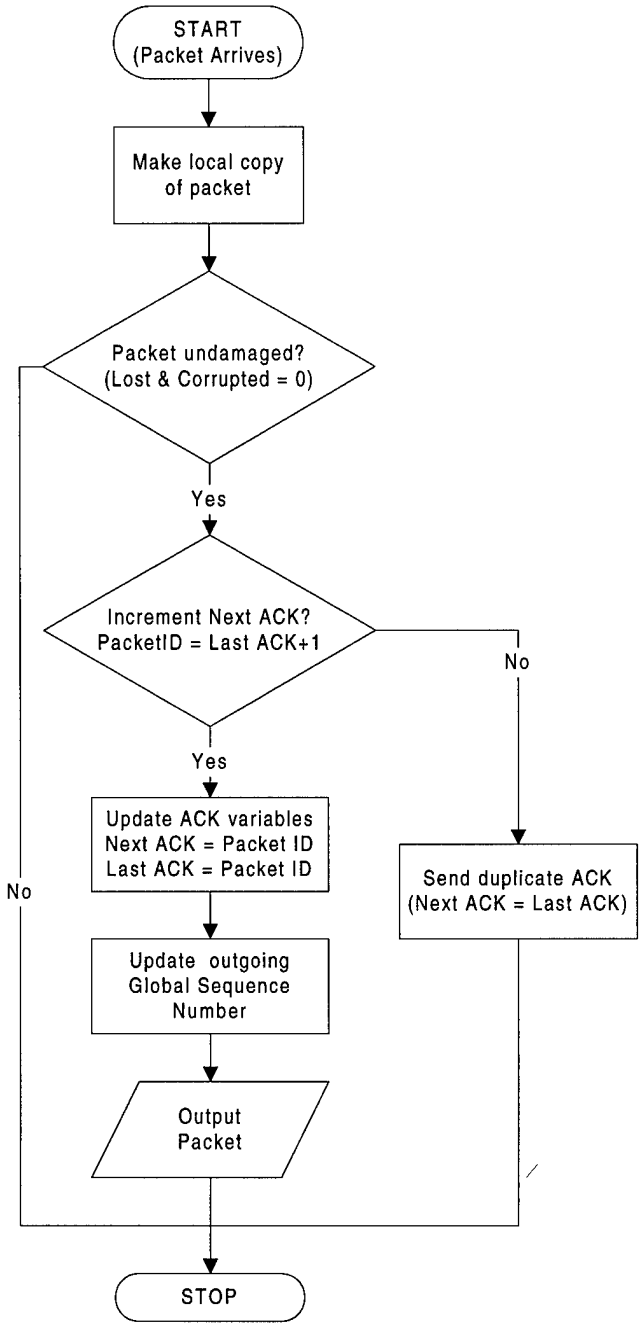

Figure 26 - Assign ACK Number Primitive (MPP)

B.2.3 Resolve ACK Primitive

The majority of implementing TCP in each pipe is accomplished in *the Resolve ACK Primitive* modules. Unlike the other primitives, the *Resolve ACK Primitive* has two triggers. The first, shown in Figure 27, is a periodic check of the timers of outstanding segments to check for a timeout. Every segment currently in the sliding window when this method is triggered is checked to see if the time since the packet was transmitted is larger than the appropriate 500 ms tick. If a timeout occurs, the loop immediately exits after setting the sliding window variables according to RFC2001, and returned a timeout confirmation.

An arriving packet is handled according to the flowchart in Figure 28. As can be seen, if the packet is undamaged and no timeout is indicated, the packet is handled by a method designed to implement the different algorithms. It should be noted that any duplicate ACKs over three are first be sent to Fast Retransmit which forwards the ACK to Fast Recovery.

Figure 29 to Figure 32 show the implementation of the four congestion control algorithms used in the MPP simulation. The decision of which algorithm to use is based on the *ACKID* of the incoming packet and the state the system is in when the packet arrives. Figure 29 shows the implementation of the Slow Start algorithm. First, the method checks to ensure that any duplicate ACKs while the system is in Slow Start do not result in an increase to the sliding window. The sliding window is then incremented as long as the new size is not bigger than the window maximum. Finally, the algorithm checks to see if the pipe should enter Congestion Avoidance.

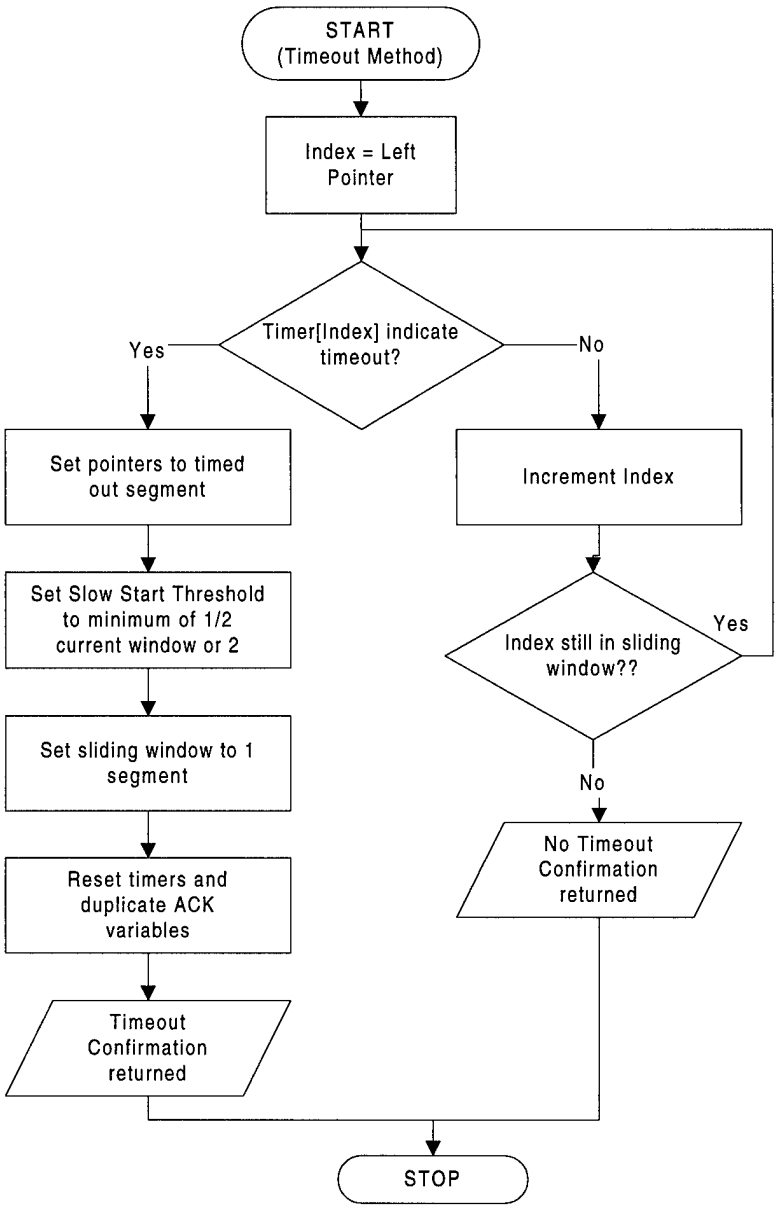

Figure 27 - Resolve ACK Primitive - Periodic Timeout (MPP)

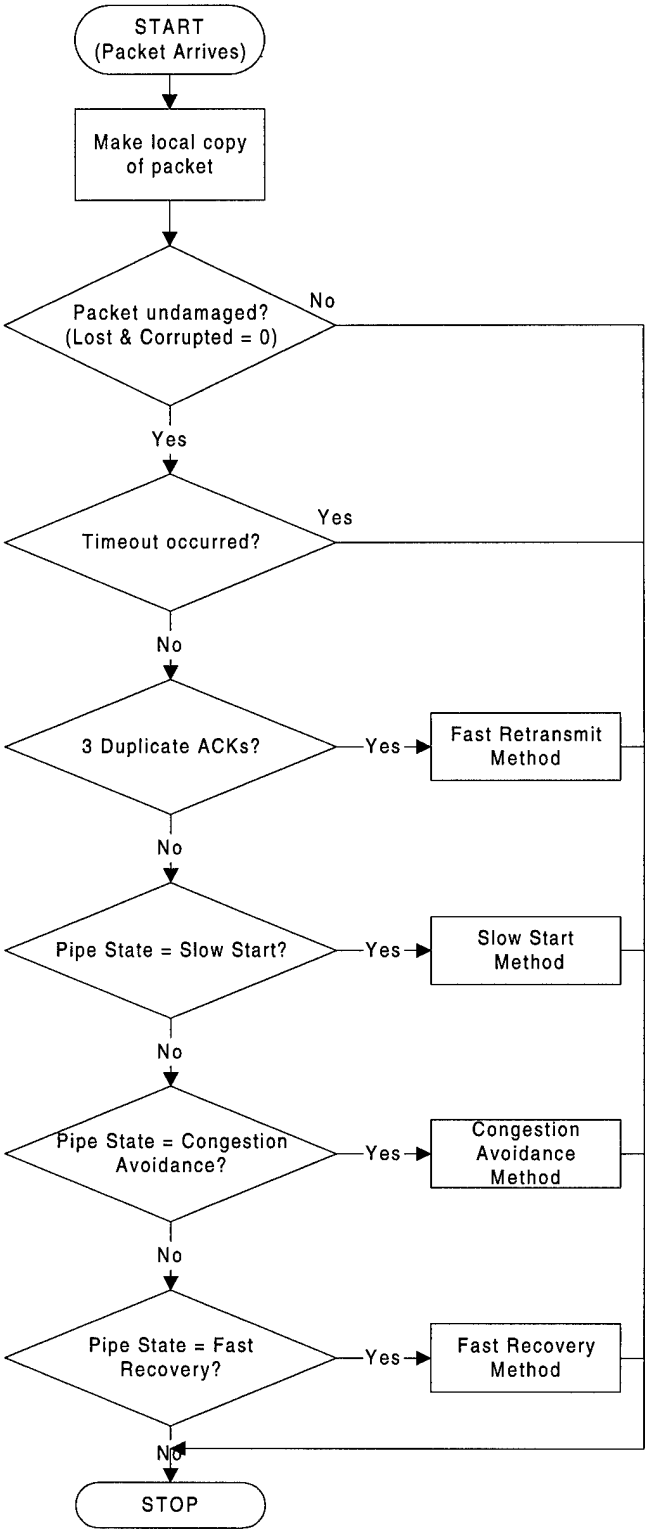

Figure 28 - Resolve ACK Primitive - Packet Arrival (MPP)

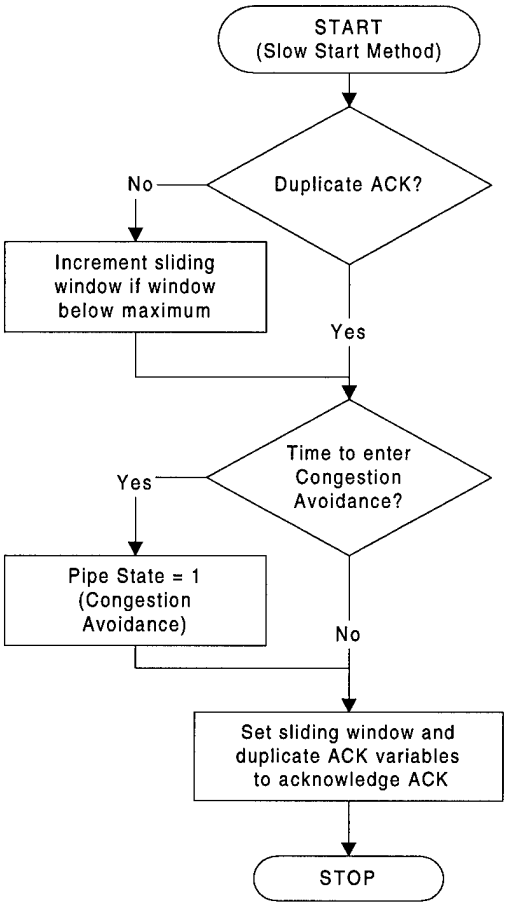

Figure 29 - Slow Start Method

The Congestion Avoidance Method (Figure 30) is primarily concerned with scheduling increases to the sliding window. As stated in Chapter 2, under Congestion Avoidance the sliding window can be increased at most every RTT, and then only when all the segments in the sliding window have been received. When the ACK of the incoming packet indicates the sliding window should increase, the next increase is set the ACK number plus the new size of the sliding window.

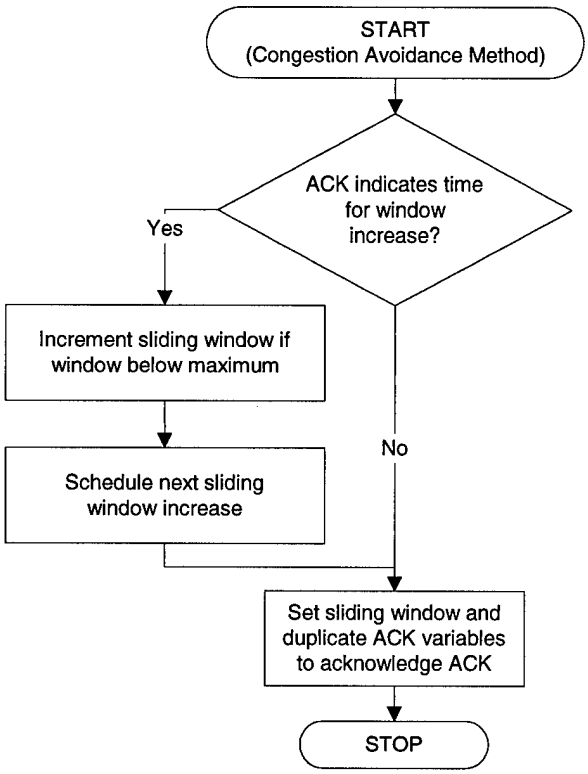

Figure 30 - Congestion Avoidance Method

Fast Retransmit, shown in Figure 31, is fairly self-explanatory. Again, it should be noted that all duplicate ACKs beyond the three needed to trigger the Fast Retransmit algorithm are sent to this method. If the pipe is already in Fast Recovery, the ACK is sent to the Fast Recovery method, shown in Figure 32. The Fast Recovery method will be directly entered from the packet arrival trigger with the deliver of the first non-duplicate ACK after a series of duplicate ACKs. The behavior of the rest of the algorithm is well documented in Chapter 2 and RFC2001, so no additional comment on the algorithms is needed here.

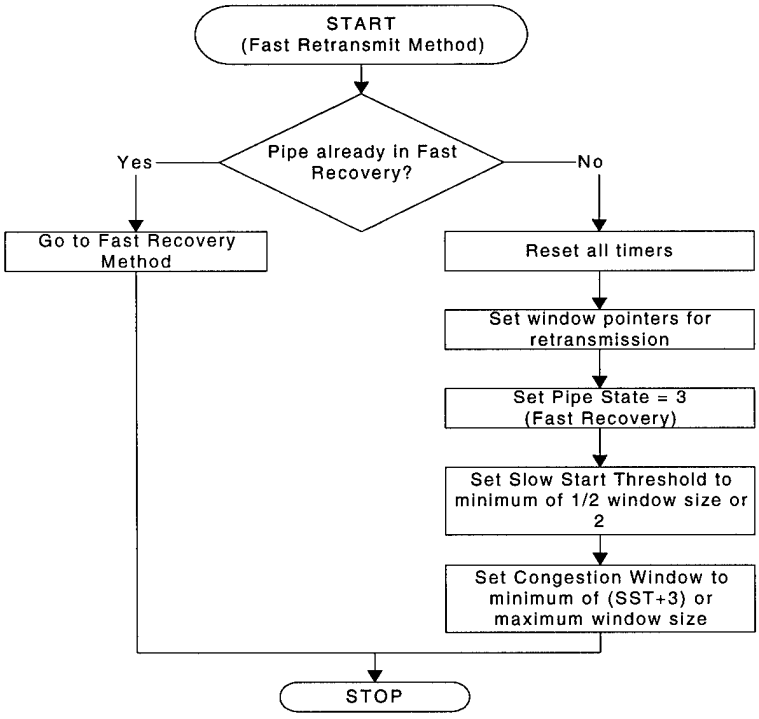

Figure 31 - Fast Retransmit Method

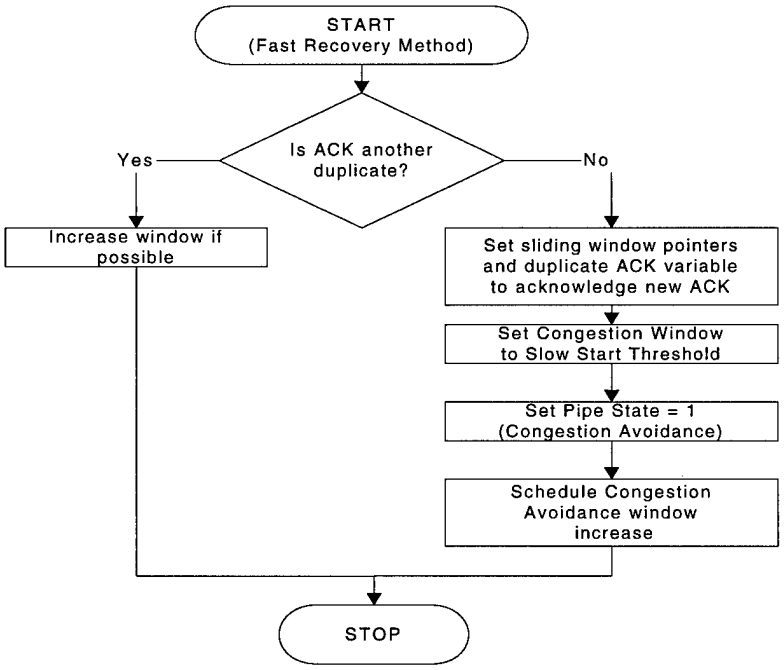

Figure 32 - Fast Recovery Method

B.3 SCPS-TP Primitives

The three SCPS-TP primitives are somewhat easier because of the use of the leaky-bucket technique, although experience with the MPP primitives allows additional functionality to be added to the primitives. The main difference is in the use of SNACK messages. Under the threat of congestion avoidance algorithms, it was important to keep close track of exactly which packets were corrupted or lost so the appropriate times could be reset appropriately. Under SCPS-TP in Congestion Mode, however, there is no sliding window so there is no need to worry about timers. SNACK messages are concerned not with exactly which segments require retransmission but merely with the number of segments that need retransmission. If any segment if lost, then it will be included in a SNACK message. As long as all SNACK messages are serviced, then SNACK retransmissions will cover any segment that would be retransmitted due to timeout. Consequently, SNACK messages are used exclusively to ensure that the proper number of segments is successfully transmitted between the *Data Source* and *Data Sink* nodes.

B.3.1 Assign SCPS Packet ID Number Primitive

The *Assign SCPS Packet ID Number Primitive* (Figure 33) is very similar *to Assign Packet ID Number Primitive*, which is its MPP counterpart. If a segment requires retransmission, the *Packet ID* is 0; otherwise the next sequential packet number is used. By decrementing the number of packets requiring retransmission, the appropriate total number of packets is sent.

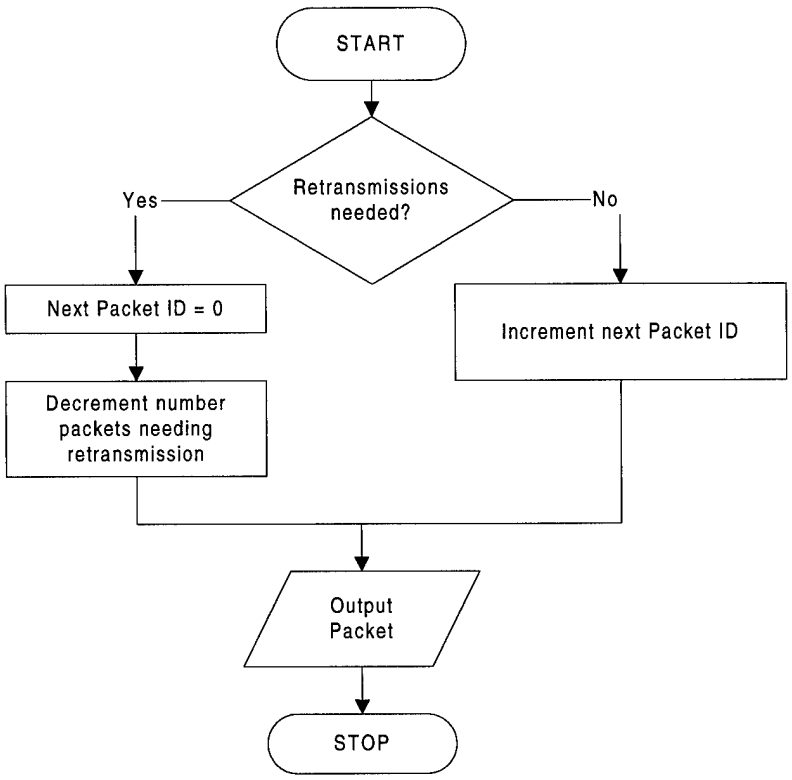

Figure 33 - Assign SCPS Packet ID Number Primitive

B.3.2 Generate SNACK Number Primitive

The *Generate SNACK Number Primitive* (Figure 34) keeps track of the number of disrupted segments. When the first uncorrupted segment arrives, the appropriate number is placed in the outgoing packet's *SNACK ID* field.

While not strictly necessary, the outstanding retransmissions variable helps ensure that no segment retransmission are lost by the simulation terminating too early.

If a segment retransmission is corrupted, it is necessary to decrement the outstanding retransmission variable. This prevents the segment from being counted twice when the next SNACK message arrives at the *Resolve ACK Primitive* module, which is critical to controlling the simulation endgame.

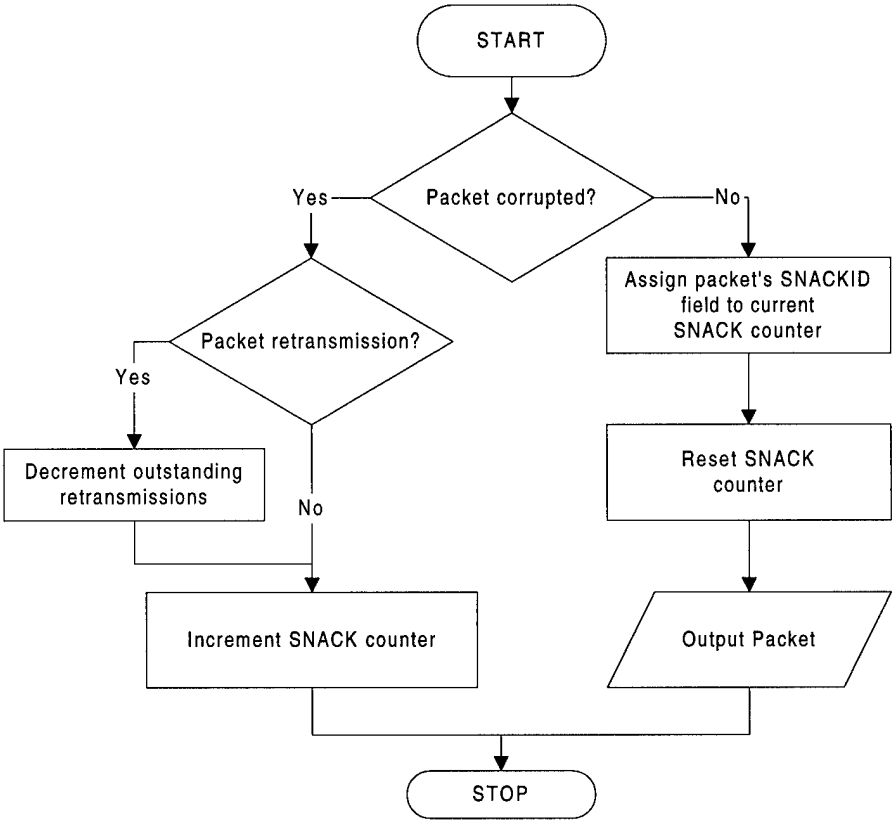

Figure 34 - Generate SNACK Number Primitive

B.3.3 Resolve SNACK Primitive

The final primitive is the *Resolve SNACK Primitive*, which controls the number of segments retransmitted. As seen in Figure 35, if the *SNACK ID* field indicates a certain number of retransmissions are required, the *Packets to Retransmit* and *Outstanding Retransmissions* variables are incremented by *SNACK ID*. If the *Packet ID* indicates that the segment was a retransmission, then the *Outstanding Retransmission* variable is decremented – the Packets to Retransmit variable is decremented by the Assign SCPS Packet ID Primitive. Finally, if there are no outstanding retransmission, the packet is

passed on to the logic that will terminate the simulation when the proper number of packets has successfully been delivered.

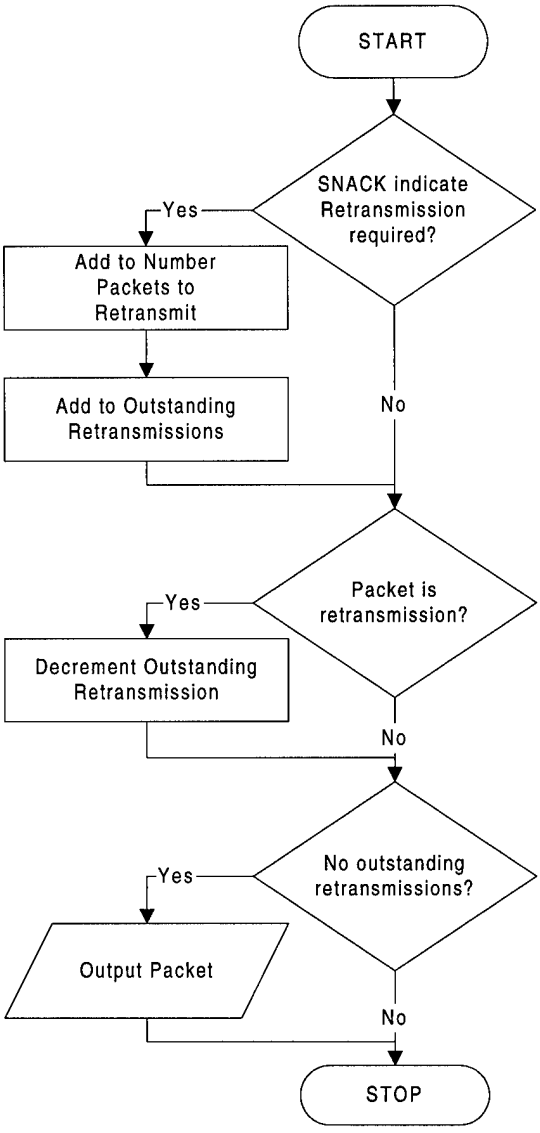

Figure 35 - SNACK Resolution Primitive

Appendix C – DESIGNER Primitives Code

Assign PacketID Number Primitive (MPP)

```
/*
* Module Name :                Assign Packet ID Number Primitive
* Template Created By :        3.0
* Author:                      rbroyles
* Last Modification Date:      25-Nov-1998 13:35:19
* Template Date:               25-Nov-1998 13:35:21
*/
#include <BONeS3.0.hh>
#include <INTEGER.hh>
#include ThesisPacket_Pipes_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class AssignPacketIDNumberPrimitive : public UserPrimitiveInstanceOR
{
public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline AssignPacketIDNumberPrimitive(const Be_ModuleProto& Proto,
      InstanceInit& Inst);
  // Run Function
  void InternalRun(ObjIndex , const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "Middle_ Pointer_Local"*/
  const INTEGER_t& Middle_Pointer_Local;
  void SetMiddle_Pointer_Local(const INTEGER_t& Middle_Pointer_Local);

  /*Definitions for memory: "Right_Pointer_Local"*/
  const INTEGER_t& Right_Pointer_Local;
  void SetRight_Pointer_Local(const INTEGER_t& Right_Pointer_Local);

  /*Definitions for memory: "Next_Packet_ID"*/
  const INTEGER_t& Next_Packet_ID;
  void SetNext_Packet_ID(const INTEGER_t& Next_Packet_ID);

  // Port Definitions:
  // Output Function
  inline void Output(const ThesisPacket_Pipes_t& x);
  // Run Functions:
  inline void Input_Run(const ThesisPacket_Pipes_t& Input);
```

```
  /**** Instance Definitions Below Here ****/
  // void Init();
  ThesisPacket_Pipes_t *TempPacket;
  /**** Instance Definitions Above Here ****/

};

const char* AssignPacketIDNumberPrimitive::SourceRCSId = "3121007719
Assign
      Packet ID Number Primitive $Header: $";

// Constructor
inline
AssignPacketIDNumberPrimitive::AssignPacketIDNumberPrimitive(const
      Be_ModuleProto& Proto, InstanceInit& Inst)
:UserPrimitiveInstanceOR(Proto,
      Inst),
            Middle_Pointer_Local(((INTEGER_t&) Arg(0u))),
            Right_Pointer_Local(((INTEGER_t&) Arg(1u))),
            Next_Packet_ID(((INTEGER_t&) Arg(2u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/
}

#ifndef AssignPacketIDNumberPrimitive_icc
#define AssignPacketIDNumberPrimitive_icc
"AssignPacketIDNumberPrimitive.icc"
#endif
#include AssignPacketIDNumberPrimitive_icc

/**** User Code Below Here ****/
/*
void AssignPacketIDNumberPrimitive::Init()
{
}*/

// Run Functions:
inline void AssignPacketIDNumberPrimitive::Input_Run(const
ThesisPacket_Pipes_t&
      Input)
{
  TempPacket = (ThesisPacket_Pipes_t*) Input.CopyArc();
  if(Middle_Pointer_Local<=Right_Pointer_Local)
  {
    SetNext_Packet_ID(Middle_Pointer_Local);
    SetMiddle_Pointer_Local(Middle_Pointer_Local+1);
  }
  else
```

```
  {
    SetNext_Packet_ID(-1);
  }
  Output(Input);
}
/**** User Code Above Here ****/
```

Assign ACK Number Primitive (MPP)

```
/*
* Module Name :                Assign ACK Number Primitive
* Template Created By :        3.0
* Author:                      rbroyles
* Last Modification Date:      28-Oct-1998 12:33:03
* Template Date:               28-Oct-1998 12:33:04
*/
#include <BONeS3.0.hh>
#include <INTEGER.hh>
#include ThesisPacket_Pipes_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class AssignACKNumberPrimitive : public UserPrimitiveInstanceOR
{
public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline AssignACKNumberPrimitive(const Be_ModuleProto& Proto,
      InstanceInit& Inst);
  // Run Function
  void InternalRun(ObjIndex , const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "NextACK"*/
  const INTEGER_t& NextACK;
  void SetNextACK(const INTEGER_t& NextACK);

  /*Definitions for memory: "LastACK"*/
  const INTEGER_t& LastACK;
  void SetLastACK(const INTEGER_t& LastACK);

  /*Definitions for memory: "NextGlobalSegment"*/
  const INTEGER_t& NextGlobalSegment;
  void SetNextGlobalSegment(const INTEGER_t& NextGlobalSegment);

  // Port Definitions:
  // Output Function
  inline void ACK_Generator_Output(const ThesisPacket_Pipes_t& x);
  // Run Functions:
  inline void ACK_Generator_Input_Run(const ThesisPacket_Pipes_t&
      ACK_Generator_Input);

  /**** Instance Definitions Below Here ****/
  // void Init();
```

```
  ThesisPacket_Pipes_t *TempPacket;
  /**** Instance Definitions Above Here ****/
};

const char* AssignACKNumberPrimitive::SourceRCSId = "3118584783 Assign
      ACK Number Primitive $Header: $";

// Constructor
inline AssignACKNumberPrimitive::AssignACKNumberPrimitive(const
      Be_ModuleProto& Proto, InstanceInit& Inst) :
            UserPrimitiveInstanceOR(Proto, Inst),
            NextACK(((INTEGER_t&) Arg(0u))),
            LastACK(((INTEGER_t&) Arg(1u))),
            NextGlobalSegment(((INTEGER_t&) Arg(2u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/

}

#ifndef AssignACKNumberPrimitive_icc
#define AssignACKNumberPrimitive_icc "AssignACKNumberPrimitive.icc"
#endif
#include AssignACKNumberPrimitive_icc

/**** User Code Below Here ****/
/*
void AssignACKNumberPrimitive::Init()
{
}*/

// Run Functions:
inline void AssignACKNumberPrimitive::ACK_Generator_Input_Run(const
      ThesisPacket_Pipes_t& ACK_Generator_Input)
{
  TempPacket = (ThesisPacket_Pipes_t*) ACK_Generator_Input.CopyArc();
  if (((*TempPacket)->Corrupted==0)&&((*TempPacket)->Lost==0))
  {
    if((*TempPacket)->PacketID==(LastACK+1))
    {
      SetNextACK((*TempPacket)->PacketID);
      SetLastACK((*TempPacket)->PacketID);
      SetNextGlobalSegment((*TempPacket)->GlobalSegmentNumber);
    }
    else
    {
      SetNextACK(LastACK);
    }
    ACK_Generator_Output(ACK_Generator_Input);
  }
```

```
}
/**** User Code Above Here ****/
```

Resolve ACK Primitive (MPP)

```
/*
* Module Name :                Resolve ACK Primitive
* Template Created By :        3.0
* Author:                      rbroyles
* Last Modification Date:      25-Nov-1998 13:38:00
* Template Date:               25-Nov-1998 13:38:03
*/
#include <BONeS3.0.hh>
#include <REAL-VECTOR.hh>
#include <REAL.hh>
#include <INTEGER.hh>
#include <TRIGGER.hh>
#include ThesisPacket_Pipes_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class ResolveACKPrimitive : public UserPrimitiveInstanceOR
{
public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline ResolveACKPrimitive(const Be_ModuleProto& Proto, InstanceInit&
      Inst);
  // Run Function
  void InternalRun(ObjIndex Port, const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "Timer"*/
  const REALVECTOR_t& Timer;
  void SetTimer(const REALVECTOR_t& Timer);

  /*Definitions for memory: "Timer_Reset"*/
  const REALVECTOR_t& Timer_Reset;
  void SetTimer_Reset(const REALVECTOR_t& Timer_Reset);

  /*Definitions for memory: "Left_Pointer_Local"*/
  const INTEGER_t& Left_Pointer_Local;
  void SetLeft_Pointer_Local(const INTEGER_t& Left_Pointer_Local);

  /*Definitions for memory: "Middle_Pointer_Local"*/
  const INTEGER_t& Middle_Pointer_Local;
  void SetMiddle_Pointer_Local(const INTEGER_t& Middle_Pointer_Local);

  /*Definitions for memory: "Right_Pointer_Local"*/
  const INTEGER_t& Right_Pointer_Local;
  void SetRight_Pointer_Local(const INTEGER_t& Right_Pointer_Local);
```

```
/*Definitions for memory: "Congestion_Window_Size_Local"*/
const INTEGER_t& Congestion_Window_Size_Local;
void SetCongestion_Window_Size_Local(const INTEGER_t&
    Congestion_Window_Size_Local);


/*Definitions for memory: "Slow_Start_Threshold_Local"*/
const INTEGER_t& Slow_Start_Threshold_Local;
void SetSlow_Start_Threshold_Local(const INTEGER_t&
    Slow_Start_Threshold_Local);

/*Definitions for parameter: "Maximum_Window_Size"*/
const INTEGER_t& Maximum_Window_Size;

/*Definitions for parameter: "Propagation_Delay"*/
const REAL_t& Propagation_Delay;

/*Definitions for memory: "Pipe_State_Local"*/
const INTEGER_t& Pipe_State_Local;
void SetPipe_State_Local(const INTEGER_t& Pipe_State_Local);

/*Definitions for memory: "LastACK1"*/
const INTEGER_t& LastACK1;
void SetLastACK1(const INTEGER_t& LastACK1);

/*Definitions for memory: "LastACK2"*/
const INTEGER_t& LastACK2;
void SetLastACK2(const INTEGER_t& LastACK2);

/*Definitions for memory: "Next_Congestion_Avoidance_Increase"*/
const INTEGER_t& Next_Congestion_Avoidance_Increase;
void SetNext_Congestion_Avoidance_Increase(const INTEGER_t&
    Next_Congestion_Avoidance_Increase);

/*Definitions for parameter: "TimeoutValue"*/
const REAL_t& TimeoutValue;

/*Definitions for parameter: "Pipe ID"*/
const INTEGER_t& PipeID;

/*Definitions for memory: "Number Timeouts"*/
const INTEGER_t& NumberTimeouts;
void SetNumberTimeouts(const INTEGER_t& NumberTimeouts);

/*Definitions for parameter: "Total Packets To Transmit"*/
const INTEGER_t& TotalPacketsToTransmit;

// Port Definitions:
// Output Function
inline void ACK_Resolution_Output(const ThesisPacket_Pipes_t& x);
// Run Functions:
inline void TimeoutCheck_Input_Run(const TRIGGER_t&
    TimeoutCheck_Input);
inline void ACK_Resolution_Input_Run(const ThesisPacket_Pipes_t&
```

```
      ACK_Resolution_Input);

  /**** Instance Definitions Below Here ****/
  // void Init();
  ThesisPacket_Pipes_t *TempPacket;
  inline void ResolveACKPrimitive::SlowStart();
  inline void ResolveACKPrimitive::CongestionAvoidance();
  inline void ResolveACKPrimitive::FastRetransmit();
  inline void ResolveACKPrimitive::FastRecovery();
  int ResolveACKPrimitive::CheckTimeout();
  /**** Instance Definitions Above Here ****/
};

const char* ResolveACKPrimitive::SourceRCSId = "3121007880 Resolve ACK
      Primitive $Header: $";

// Constructor
inline ResolveACKPrimitive::ResolveACKPrimitive(const Be_ModuleProto&
      Proto, InstanceInit& Inst) : UserPrimitiveInstanceOR(Proto,
      Inst),
            Timer(((REALVECTOR_t&) Arg(0u))),
            Timer_Reset(((REALVECTOR_t&) Arg(1u))),
            Left_Pointer_Local(((INTEGER_t&) Arg(2u))),
            Middle_Pointer_Local(((INTEGER_t&) Arg(3u))),
            Right_Pointer_Local(((INTEGER_t&) Arg(4u))),
            Congestion_Window_Size_Local(((INTEGER_t&) Arg(5u))),
            Slow_Start_Threshold_Local(((INTEGER_t&) Arg(6u))),
            Maximum_Window_Size(((INTEGER_t&) Arg(7u))),
            Propagation_Delay(((REAL_t&) Arg(8u))),
            Pipe_State_Local(((INTEGER_t&) Arg(9u))),
            LastACK1(((INTEGER_t&) Arg(10u))),
            LastACK2(((INTEGER_t&) Arg(11u))),
            Next_Congestion_Avoidance_Increase(((INTEGER_t&)
                  Arg(12u))),
            TimeoutValue(((REAL_t&) Arg(13u))),
            PipeID(((INTEGER_t&) Arg(14u))),
            NumberTimeouts(((INTEGER_t&) Arg(15u))),
            TotalPacketsToTransmit(((INTEGER_t&) Arg(16u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/
}

#ifndef ResolveACKPrimitive_icc
#define ResolveACKPrimitive_icc "ResolveACKPrimitive.icc"
#endif
#include ResolveACKPrimitive_icc

/**** User Code Below Here ****/
/*
```

```
void ResolveACKPrimitive::Init()
{
}*/

// Run Functions:
inline void ResolveACKPrimitive::TimeoutCheck_Input_Run(const
      TRIGGER_t& TimeoutCheck_Input)
{
  // This run function is used to trigger timeout checks. It is a
  // safety valve, since if the timeouts were only checked by the
  // arrival of packets, and since the ACK generator module only sends
  // packets when a packet is received, then no packet would ever be
  // received if errors cause the congestion window to be limited and
  // the Traffic Generator can only send pacekts in the window. The
  // output is basically just a trigger to get the changes to the local
  // sliding window variables to be updated into the top-level vector
  // variables, so they can be used by the Traffic Generator modules.

  ThesisPacket_Pipes_t OutTrigger;
  int i=CheckTimeout();
  if(i==1)
  {
    ACK_Resolution_Output(OutTrigger);
    cout<<"Timeout from Periodic Check\n";
  }
}

inline void ResolveACKPrimitive::ACK_Resolution_Input_Run(const
      ThesisPacket_Pipes_t& ACK_Resolution_Input)
{
  TempPacket = (ThesisPacket_Pipes_t*) ACK_Resolution_Input.CopyArc();

  // First, ensure there are no timeouts, errors, packet loss. or that
  // the packet is one that was sent by the ACK Generator module before
  // the first packet arrived from the Data Source node

  int serviced=0;

  if (((*TempPacket)->GlobalSegmentNumber!=-1)&&(CheckTimeout()==0)&&
      ((*TempPacket)->Corrupted==0)&&((*TempPacket)->Lost==0))
  {
    if((*TempPacket)->GlobalSegmentNumber>=TotalPacketsToTransmit)
    {
      cout<<"GLOBAL SEGMENT NUMBER EQUALS TOTAL PACKETS TO TRANSMIT:
            "<<TNow()<<"\n";
    }

    // Check for Duplicate ACKs
    if((LastACK2==LastACK1)&&((*TempPacket)->ACKID==LastACK1))
    {
      FastRetransmit();
      serviced=1;
    }

    // Branch based on pipe states. The "serviced" variable ensures
```

```
    // only one of these routines services each arriving packet

    if ((Pipe_State_Local==0)&&(serviced==0))
    {
      SlowStart();
      serviced=1;
    }
    else if ((Pipe_State_Local==1)&&(serviced==0))
    {
      CongestionAvoidance();
      serviced=1;
    }
    else if ((Pipe_State_Local==3)&&(serviced==0))
    {
      FastRecovery();
      serviced=1;
    }
  }
  ACK_Resolution_Output(ACK_Resolution_Input);
}


/*********************************************************************/
/**** Here are the function for the different TCP Protocols.**********/
/*********************************************************************/

inline void ResolveACKPrimitive::SlowStart()
{
  // This method is the Slow Start method. Each time there is an ACK
  // message, the size of the congestion window is increased up to the
  // maximum window size. If the new window size is greater than the
  // SST, then go into Congestion Avoidance.  Finally, adjust the
  // sliding window to acknowledge the ACK message

  // If possible, increase the congestion window, making sure you are
  // not "rewarding" a duplicate ACK

  if((Congestion_Window_Size_Local<Maximum_Window_Size)&&
      ((*TempPacket)->ACKID!=LastACK1))
  {
    SetCongestion_Window_Size_Local(Congestion_Window_Size_Local+1);
  }

  // Check to see if enter Congestion Avoidance. If so, change state
  // and make sure to set where the next increase will come once enter
  // Congestion Avoidance

  if(Congestion_Window_Size_Local > Slow_Start_Threshold_Local)
  {
    SetPipe_State_Local(1);
    SetNext_Congestion_Avoidance_Increase((*TempPacket)
      ->ACKID+Congestion_Window_Size_Local);
  }

  // Acknowledge the ACK message
```

```
  SetLeft_Pointer_Local((*TempPacket)->ACKID);
  SetRight_Pointer_Local((*TempPacket)->ACKID+
      Congestion_Window_Size_Local);
  SetLastACK2(LastACK1);
  SetLastACK1((*TempPacket)->ACKID);
}

inline void ResolveACKPrimitive::CongestionAvoidance()
{
  // This method implements the Congestion Avoidance algorithm. It
  // decreases the rate the congestion window increases - the window
  // can increase by 1 approximately every RTT. Finally, move the
  // sliding window vars to acknowledge the ACK message

  // Increase the window size, making sure you do not make it bigger
  // than Maximum_Window_Size, and set where next increase will occur

  if(((*TempPacket)->ACKID==Next_Congestion_Avoidance_Increase)&&
      ((*TempPacket)->ACKID!=LastACK1))
  {
    if(Congestion_Window_Size_Local < Maximum_Window_Size)
    {
      SetCongestion_Window_Size_Local(Congestion_Window_Size_Local+1);
    }
    SetNext_Congestion_Avoidance_Increase((*TempPacket)
      ->ACKID+Congestion_Window_Size_Local);
  }

  // Acknowledge the ACK message
  SetLeft_Pointer_Local((*TempPacket)->ACKID);
  SetRight_Pointer_Local((*TempPacket)->ACKID+
      Congestion_Window_Size_Local);
  SetLastACK2(LastACK1);
  SetLastACK1((*TempPacket)->ACKID);
}

inline void ResolveACKPrimitive::FastRetransmit()

{
  // This method is called each time there are duplicate ACKs. If the
  // system is already in Fast Recovery, then control is passed to the
  // Fast Recovery algorithm, which know how to react to additional
  // (beyond 3) duplicate ACKs. If this is the first occurrence, then
  // the segment is retransmitted and sliding window variable adjusted
  // according to RFC2001

  // First, check to see if already in Fast Recovery. If so, set it
  // to Fast Recovery

  if(Pipe_State_Local==3)
  {
    FastRecovery();
  }
  else
  {
```

```
    // This is done the first time three duplicate ACKs are received
    // First, reset the timers for all outstanding segments.. This way,
    // you don't have segments that are going to have to be re-
    //transmitted timing out.

    SetTimer(Timer_Reset);

    // Get ready to retransmit - next time state is Fast Recovery

    SetLeft_Pointer_Local((*TempPacket)->ACKID-1);
    SetMiddle_Pointer_Local((*TempPacket)->ACKID);
    SetPipe_State_Local(3);

    // Now adjust the rest of the sliding window variables

    SetSlow_Start_Threshold_Local(Congestion_Window_Size_Local / 2);

    // Ensure SST is at least 2 segments

    if (Slow_Start_Threshold_Local<2)
    {
      SetSlow_Start_Threshold_Local(2);
    }
    // Reset the Congestion Window and the Pointers, taking care to
    // ensure they are not set to values greater than the Maximum
    // Window Size

    if (Slow_Start_Threshold_Local+3 <= Maximum_Window_Size)
    {
      SetCongestion_Window_Size_Local(Slow_Start_Threshold_Local+3);
      SetRight_Pointer_Local(Left_Pointer_Local+
            Congestion_Window_Size_Local);
    }
    else
    {
      SetCongestion_Window_Size_Local(Maximum_Window_Size);
      SetRight_Pointer_Local(Left_Pointer_Local+Maximum_Window_Size);
    }
  }
}

inline void ResolveACKPrimitive::FastRecovery()
{
  // This method implements the Fast Recovery algorithm. There are two
  // ways to enter this method. First, if the Packet ID of the incoming
  // packet is the same as the last two received, the program first
  // calls Fast Retransmit. If the Pipe State is already Fast Recovery
  // (i.e. an add'l ACK (beyond 3) arrives), then Fast Recovery is
  // called to deal with the additional duplicate ACK. If a new ACK
  // arrives but the Pipe State is already in Fast Recovery, again Fast
  // Recovery() is called to deal with the new ACK message (see
  // RFC2001).


  // Check to see if this is an additional (beyond 3) duplicate ACK
```

```
  if (((*TempPacket)->ACKID==LastACK1)&&(LastACK1==LastACK2))
  {
    // Increase the window (as long as it is less than Maximum Window
    // Size)

    if(Congestion_Window_Size_Local+1 <= Maximum_Window_Size)
    {
      SetCongestion_Window_Size_Local(Congestion_Window_Size_Local+1);
      SetRight_Pointer_Local((Right_Pointer_Local+1));
    }
  }
  else
  {
    // This is the first ACK after entering Fast Retransmit that
    // acknowledges the reception of new data. Reset window pointers
    // and enter Congestion Avoidance

    SetCongestion_Window_Size_Local(Slow_Start_Threshold_Local);
    SetLeft_Pointer_Local((*TempPacket)->ACKID);
    SetRight_Pointer_Local(Left_Pointer_Local+
      Congestion_Window_Size_Local);
    SetPipe_State_Local(1);
    SetNext_Congestion_Avoidance_Increase(((*TempPacket)
      ->ACKID+Congestion_Window_Size_Local));
    SetLastACK1((*TempPacket)->ACKID);
    SetLastACK2(0);
  }
}

int ResolveACKPrimitive::CheckTimeout()
{
  // This method goes through the Timer variable checking for timers
  // that have expired. It runs throuth the Timer using indexes from
  // the Left_Pointer_Local to Middle_Pointer_Local, which is all the
  // outstanding segments. It checks TNow minus the time the packet was
  // created. If a timeout occurs, it resets all the timers and sliding
  // window variables for Slow Start and returns 1. If no no segment
  // times out, it returns 0. It also resets the LastACK variables to -
  // 1. See Chapter 3 for details on how the timeout value was
  // determined

  int returnvalue=0;
  // cout<<"TimeoutCheck at time: "<<TNow()<<"\n";
  // cout<<"Right:"<<Right_Pointer_Local<<" Middle:"<<
      Middle_Pointer_Local<<"\n";

  for (int i=(Left_Pointer_Local+1);((i<Middle_Pointer_Local)&&
      returnvalue==0));i++)
  {
    // cout<<"Pipe:"<<PipeID<<" i:"<<i<<"\n";
    if ((TNow()-Timer[i]) > TimeoutValue)
    {
      // Segment has timed out, so resend segment and enter Slow Start
```

```
        SetNumberTimeouts(NumberTimeouts+1);
        cout<<"Number Timeouts: "<<NumberTimeouts<<"\n";
        cout<<"Pipe:"<<PipeID<<"Timeout:"<<i<<" Timer i: "<<Timer[i]<<
              "TNow:"<<TNow()<<"\n";
        cout<<"Left:"<<Left_Pointer_Local<<" Middle:"<<
              Middle_Pointer_Local<<"\n";
        SetLeft_Pointer_Local(i-1);
        SetMiddle_Pointer_Local(i);
        SetRight_Pointer_Local(i);
        SetSlow_Start_Threshold_Local((Congestion_Window_Size_Local /
              2));

        // Made sure SST is at least 2

        if (Slow_Start_Threshold_Local<2)
        {
          SetSlow_Start_Threshold_Local(2);
        }
        SetCongestion_Window_Size_Local(1);
        SetPipe_State_Local(0);
        SetTimer(Timer_Reset);
        SetLastACK1(-1);
        SetLastACK2(-1);
        returnvalue = 1;
      }
    }
    return(returnvalue);
}
/**** User Code Above Here ****/
```

Assign PacketID Number (SCPS-TP)

```
/*
* Module Name :                Assign SCPS Packet ID Number Primitive
* Template Created By :        3.0
* Author:                      rbroyles
* Last Modification Date:      18-Nov-1998 15:33:36
* Template Date:               18-Nov-1998 15:33:37
*/
#include <BONeS3.0.hh>
#include <INTEGER.hh>
#include ThesisPacket_SCPS_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class AssignSCPSPacketIDNumberPrimitive :public UserPrimitiveInstanceOR
{
  public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline AssignSCPSPacketIDNumberPrimitive(const Be_ModuleProto& Proto,
      InstanceInit& Inst);
  // Run Function
  void InternalRun(ObjIndex , const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "Packets To Retransmit"*/
  const INTEGER_t& PacketsToRetransmit;
  void SetPacketsToRetransmit(const INTEGER_t& PacketsToRetransmit);

  /*Definitions for memory: "Last New PacketID"*/
  const INTEGER_t& LastNewPacketID;
  void SetLastNewPacketID(const INTEGER_t& LastNewPacketID);

  // Port Definitions:
  // Output Function
  inline void Output(const ThesisPacket_SCPS_t& x);
  // Run Functions:
  inline void Input_Run(const ThesisPacket_SCPS_t& Input);

  /**** Instance Definitions Below Here ****/
  // void Init();
  ThesisPacket_SCPS_t *TempPacket;
  /**** Instance Definitions Above Here ****/

};

const char* AssignSCPSPacketIDNumberPrimitive::SourceRCSId =
      "3120410016 Assign SCPS Packet ID Number Primitive $Header: $";
```

```
// Constructor
inline
AssignSCPSPacketIDNumberPrimitive::AssignSCPSPacketIDNumberPrimitive
      (const Be_ModuleProto& Proto, InstanceInit& Inst)
      : UserPrimitiveInstanceOR(Proto, Inst),
            PacketsToRetransmit(((INTEGER_t&) Arg(0u))),
            LastNewPacketID(((INTEGER_t&) Arg(1u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/
}

#ifndef AssignSCPSPacketIDNumberPrimitive_icc
#define                              AssignSCPSPacketIDNumberPrimitive_icc
"AssignSCPSPacketIDNumberPrimitive.icc"
#endif
#include AssignSCPSPacketIDNumberPrimitive_icc

/**** User Code Below Here ****/
/*
void AssignSCPSPacketIDNumberPrimitive::Init()
{
}*/

// Run Functions:
inline void AssignSCPSPacketIDNumberPrimitive::Input_Run(const
      ThesisPacket_SCPS_t& Input)
{
  // Create the packet that will be output
  TempPacket=(ThesisPacket_SCPS_t*) Input.CopyArc();

  // Check to see if there are packets needing retransmissions
  if(PacketsToRetransmit==0)
  {
    (*TempPacket)->PacketID=LastNewPacketID;
    SetLastNewPacketID(LastNewPacketID+1);
  }

  // There are packets to retransmit
  else
  {
    (*TempPacket)->PacketID=0;
    SetPacketsToRetransmit(PacketsToRetransmit-1);
  }
  // Transmit packet
  Output(*TempPacket);
}
/**** User Code Above Here ****/
```

Assign SNACK Number Primitive (SCPS-TP)

```
/*
* Module Name : Generate SNACK Number Primitive
* Template Created By : 3.0
* Author: rbroyles
* Last Modification Date: 18-Nov-1998 17:26:50
* Template Date: 18-Nov-1998 17:26:52
*/
#include <BONeS3.0.hh>
#include <INTEGER.hh>
#include ThesisPacket_SCPS_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class GenerateSNACKNumberPrimitive : public UserPrimitiveInstanceOR
{
  public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline GenerateSNACKNumberPrimitive(const Be_ModuleProto& Proto,
      InstanceInit& Inst);
  // Run Function
  void InternalRun(ObjIndex , const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "Next_SNACK_Number"*/
  const INTEGER_t& Next_SNACK_Number;
  void SetNext_SNACK_Number(const INTEGER_t& Next_SNACK_Number);

  /*Definitions for memory: "Outstanding_Retransmissions"*/
  const INTEGER_t& Outstanding_Retransmissions;
  void SetOutstanding_Retransmissions(const INTEGER_t&
      Outstanding_Retransmissions);

  // Port Definitions:
  // Output Function
  inline void SNACK_Generator_Out(const ThesisPacket_SCPS_t& x);
  // Run Functions:
  inline void SNACK_Generator_In_Run(const ThesisPacket_SCPS_t&
      SNACK_Generator_In);

  /**** Instance Definitions Below Here ****/
  // void Init();
  ThesisPacket_SCPS_t *TempPacket;
  /**** Instance Definitions Above Here ****/

};
```

```
const char* GenerateSNACKNumberPrimitive::SourceRCSId = "3120416810
      Generate SNACK Number Primitive $Header: $";

// Constructor
inline GenerateSNACKNumberPrimitive::GenerateSNACKNumberPrimitive(const
      Be_ModuleProto& Proto, InstanceInit& Inst)
      : UserPrimitiveInstanceOR(Proto, Inst),
            Next_SNACK_Number(((INTEGER_t&) Arg(0u))),
            Outstanding_Retransmissions(((INTEGER_t&) Arg(1u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/
}

#ifndef GenerateSNACKNumberPrimitive_icc
#define                                    GenerateSNACKNumberPrimitive_icc
"GenerateSNACKNumberPrimitive.icc"
#endif
#include GenerateSNACKNumberPrimitive_icc

/**** User Code Below Here ****/

// Run Functions:
inline void GenerateSNACKNumberPrimitive::SNACK_Generator_In_Run(const
      ThesisPacket_SCPS_t& SNACK_Generator_In)
{
  TempPacket=(ThesisPacket_SCPS_t*) SNACK_Generator_In.CopyArc();
  // If the packet is lost or corrupted, then it must be retransmitted;
  // if not, send the packet through, and insert the number of packets
  // needing retransmission. Since in the model this will never be
  // lost, I need to send this only once. If a retransmission is
  // corrupted, I need to decrement the number of retransmissions the
  // SNACK Resolution is waiting for, or else the packet will be
  // counted twice in the Outstanding Retransmissions variable.
  if(((*TempPacket)->Corrupted==1)||((*TempPacket)->Lost==1))
  {
    SetNext_SNACK_Number(Next_SNACK_Number+1);
    if((*TempPacket)->PacketID==0)
    {
      SetOutstanding_Retransmissions(Outstanding_Retransmissions-1);
    }
  }
  else
  {
    // Insert number of packets needing retransmission
    (*TempPacket)->SNACKID=Next_SNACK_Number;
    // Output the packet
    SNACK_Generator_Out(*TempPacket);
    // Resent the SNACK number
    SetNext_SNACK_Number(0);
```

```
  }
}
/**** User Code Above Here ****/
```

SNACK Resolution Primitive

```
/*
* Module Name :                SNACK Resolution Primitive
* Template Created By :        3.0
* Author:                      rbroyles
* Last Modification Date:      18-Nov-1998 15:33:11
* Template Date:               18-Nov-1998 15:33:13
*/
#include <BONeS3.0.hh>
#include <INTEGER.hh>
#include ThesisPacket_SCPS_hh

/**** Includes and Defines Below Here ****/
/**** Includes and Defines Above Here ****/

class SNACKResolutionPrimitive : public UserPrimitiveInstanceOR
{
public:
  // Internal Functions: Don't Call From Primitive
  // Constructor
  inline SNACKResolutionPrimitive(const Be_ModuleProto& Proto,
      InstanceInit& Inst);
  // Run Function
  void InternalRun(ObjIndex , const BONeSData& NewValue);
  static const char* SourceRCSId;
  static const char* IncludeIdString;
  // End of Internal Functions

  // Arguments:

  /*Definitions for memory: "Packets To Retransmit"*/
  const INTEGER_t& PacketsToRetransmit;
  void SetPacketsToRetransmit(const INTEGER_t& PacketsToRetransmit);

  /*Definitions for memory: "Outstanding Retransmissions"*/
  const INTEGER_t& OutstandingRetransmissions;
  void SetOutstandingRetransmissions(const INTEGER_t&
      OutstandingRetransmissions);

  // Port Definitions:
  // Output Function
  inline void SNACK_Resolution_Output(const ThesisPacket_SCPS_t& x);
  // Run Functions:
  inline void SNACK_Resolution_Input_Run(const ThesisPacket_SCPS_t&
      SNACK_Resolution_Input);

  /**** Instance Definitions Below Here ****/
  // void Init();
  ThesisPacket_SCPS_t *TempPacket;
  /**** Instance Definitions Above Here ****/

};
```

```
const char* SNACKResolutionPrimitive::SourceRCSId = "3120409991 SNACK
Resolution Primitive $Header: $";

// Constructor
inline SNACKResolutionPrimitive::SNACKResolutionPrimitive(const
      Be_ModuleProto&   Proto,        InstanceInit&        Inst)        :
      UserPrimitiveInstanceOR(Proto, Inst),
            PacketsToRetransmit(((INTEGER_t&) Arg(0u))),
            OutstandingRetransmissions(((INTEGER_t&) Arg(1u)))
//Add code here only if you are expert with C++

/**** Initializers for User Defined Instance Objects Below Here ****/
/**** Initializers for User Defined Instance Objects Above Here ****/

{
/**** User Constructor Code Below Here ****/
/**** User Constructor Code Above Here ****/
}

#ifndef SNACKResolutionPrimitive_icc
#define SNACKResolutionPrimitive_icc "SNACKResolutionPrimitive.icc"
#endif
#include SNACKResolutionPrimitive_icc

/**** User Code Below Here ****/

// Run Functions:
inline void SNACKResolutionPrimitive::SNACK_Resolution_Input_Run(const
      ThesisPacket_SCPS_t& SNACK_Resolution_Input)
{
  TempPacket=(ThesisPacket_SCPS_t*) SNACK_Resolution_Input.CopyArc();

  if((*TempPacket)->SNACKID!=0)
  {
    SetPacketsToRetransmit(PacketsToRetransmit+(*TempPacket)->SNACKID);
    SetOutstandingRetransmissions(OutstandingRetransmissions+
      (*TempPacket)->SNACKID);
  }
  // Check to see if this is a retransmissions
  if ((*TempPacket)->PacketID==0)
  {
    SetOutstandingRetransmissions(OutstandingRetransmissions-1);
    cout<<"Retransmission: Outstanding Retransmissions="<<
      OutstandingRetransmissions<<"\n";
  }
  // When the simulation terminates, there must be no packets in the
  // process of retransmission. Since the output of this primitive will
  // be used to determine if the simulation should end, then the output
  // should only be sent if there are no outstanding retransmissions
  if (OutstandingRetransmissions<=0)
  {
    SNACK_Resolution_Output(*TempPacket);
  }
}
/**** User Code Above Here ****/
```

Bibliography

[AlK96] Allman, M., Kruse, H., and Ostermann, S. "An Application-Level Solution to TCP's Satellite Inefficiencies" Proceedings of the First International Conference on Telecommunications Systems via WWW (http://jarok.cs.ohiou.edu/papers), 1996.

[All97] Allman, M. *Improving TCP Performance Over Satellite Channels*, MS thesis, Ohio University, June 1997.

[All98] Allman, M. (Editor) "Ongoing TCP Research Related to Satellites, Internet Engineering Task Force – TCP Over Satellite Working Group (Internet Draft)" WWWeb, http://www.ietf.org/internet-drafts/draft-ietf-tcpsat-res-issues-03.txt, 27 May– 27 Nov 98.

[AlG98] Allman, M. and Glover, D. "Enhancing TCP over satellite channels using Standard Mechanisms, IETF Internet Draft" WWWeb, http://www.ietf.org/internet-drafts/draft-ietf-tcpsat-stand-mech-04.txt, Jun 98.

[BrO94] Brakmo, L., O'Malley, S., and Peterson, L. "TCP Vegas: New Techniques for Congestion Avoidance" Proceedings of ACM SIGCOMM '94 24-35 (October 1994).

[Com95] Comer, D. Internetworking with TCP/IP Prentice Hall, New Jersey, 1995.

[Com97] Comlinks, Inc. "Technical Details of the Teledesic Network" WWWeb, http://www.comlinks.com/sys.teled.htm, 1997.

[DuM96] Durst, R., Miller,G., and Travis, E. "TCP Extensions for Space Communication" Mobile Computing and Networking 15 (1996)

[Flo97] Floyd, S. Note to end2end-interest mailing list, Feb 97

[Hay97] Hayes, C. *Analyzing the Performance of New TCP Extensions Over Satellite Links*, MS thesis, Ohio University, August 1997

[ITU97] International Telecommunications Union "Error Performance of an International Digital Connection Operating at a Bit Rate Below the Primary Service Rate and Forming Part of an Integrated Services Digital Network (G.821)" ITU via WWW (http://www.itu. int/ itudoc/itu-t/rec/g/g800up/g821_28394.html), 1997

[Jac90] Jacobson, V. "Modified TCP Congestion Avoidance Algorithm" Lawrence Berkley Lab via WWW (ftp://ftp.ee.lbl.gov/email/vanj90aapr30.txt), April 90

[Jan96] Janoso, R. *Performance Analysis of Dynamic Routing Protocols in a Low Earth Orbit Satellite Data Network*, MS thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH AFIT/GE/ENG/96D-08, December 1996

[JaB92] Jacobson, V. and R. Braden, R. "TCP Extensions for High Performance (RFC1323)" WWWeb, http://www.cis.ohio-state.edu/htbin/rfc/rfc1323.html, November 1992

[JaK88] Jacobson, V. and Karels, M. "Congestion Avoidance and Control" ACM SIGCOMM Computer Communication Review, 18#4: 314-329 (August 1988),

[Mah97] Mahdavi, J. "Enabling High Performance Data Transfers on Hosts Pittsburgh Supercomputing Center" WWWeb, http://www.psc.edu/networking/perf_tune.html, Nov 1997

[LiS95] Linder, H., Sterling, W., and Hofmann, U. "Performance of XTPX and TCP/ IP in a Satellite Environment" IEEE Conference on Local Computer Networks, 246-253 (1995)

[SaA94] Saadawi, T., Ammar, M., and Hakeem, Ahmed El. Fundamentals of Telecommunication Networks, Wiley, New York, 1994

[Stu97] Sturza, M. "The Teledesic Satellite System" WWWeb, http://www3.csysco. com/budapest", 1997

[Tel97] Teledesic Corporation, "Does Latency Matter?" WWWeb, http://www.teledesic.com/ tech/latency.html, 1997

[Tel98a] Teledesic Corporation, "Teledesic Overview", WWWeb, http://www. teledesic.com/overview.html, 1998

[WrS95] Wright, G. and Stevens, W. TCP Illustrated Volume 2 – The Implementation, Addison-Wesley Publishing Company, Massachusetts, 1995

Vita

Captain Ren Broyles was born on 27 September 1968 in Riverside, California. He graduated from Redlands High School in 1986 and attended Brigham Young University in Provo, Utah. After completing a mission for the Church of Jesus Christ of Latter-Day Saints, he graduated with a Bachelor's degree in Electrical Engineering. He received his United States Air Force commission from the Reserve Officer Training Corps and entered active duty in September 1993. His first assignment was at Los Angeles Air Force Base, where he was assigned to the MILSATCOM Joint Program Office as a System Test Engineer on the Milstar program. He then transferred to the System Acquisition Program office at LAAFB, where he graduated form the Education With Aerospace program. While at LAAFB, he received a M.B.A. from Chapman University. He entered the School of Engineering, Air Force Institute of Technology in September 1997.

Permanent Address: 1032 Lolita
Redlands, CA 92373