

2
1993

NAVAL POSTGRADUATE SCHOOL Monterey, California

AD-A268 941




DTIC
ELECTE
SEP 08 1993
S E D

THESIS

**SPECIFICATION AND ANALYSIS
OF A HIGH SPEED TRANSPORT PROTOCOL**

by
H. Alphan TIPICI, LTJG

June 1993

Thesis Advisor: **Prof. G.M. Lundy**

Approved for public release; distribution is unlimited.

93-20694


93 9 08 0 7

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) SPECIFICATION AND ANALYSIS OF A HIGH SPEED TRANSPORT PROTOCOL(Unclassified)			
12. PERSONAL AUTHOR(S) Tipici, Huseyin Alphan, LTJG.			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 09/92 TO 06/93	14. DATE OF REPORT (Year, Month, Day) June 1993	15. PAGE COUNT 95
16. SUPPLEMENTARY NOTATION 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 United States Government.			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	Formal Specification, Transport Protocol, System State Analysis, Global Analysis, SCM, CFSM	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) While networks have been getting faster, perceived throughput at the application has not always increased accordingly and the bottleneck has moved to the communications processing part of the system. The issues that cause the performance bottlenecks in the current transport protocols are discussed in this thesis, and a further study on a high speed transport protocol which tries to overcome these difficulties with some unique features is presented. By using the Systems of Communicating Machines (SCM) model as a framework, a refined and improved version of the formal protocol specification is built over the previous work, and it is analyzed to verify that the protocol is free from logical errors such as deadlock, unspecified reception, unexecuted transitions and blocking loops. The analysis is conducted in two phases which consists of the application of the associated system state analysis and the simulation of the protocol using the programming language ADA. The thesis also presents the difficulties encountered during the course of the analysis, and suggests possible solutions to some of the problems.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof G.M. Lundy		22b. TELEPHONE (Include Area Code) (408) 656-2094/2440	22c. OFFICE SYMBOL CS/Ln

Approved for public release; distribution is unlimited

**SPECIFICATION AND ANALYSIS
OF A HIGH SPEED TRANSPORT PROTOCOL**

by
H. Alphan Tipici
Lieutenant Junior Grade, Turkish Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

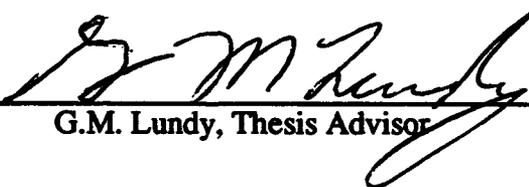
June 1993

Author:

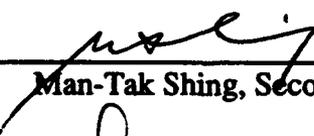


H. Alphan Tipici

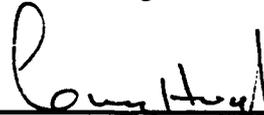
Approved By:



G.M. Lundy, Thesis Advisor



Man-Tak Shing, Second Reader



Gary Hughes, Chairman,
Department of Computer Science

ABSTRACT

While networks have been getting faster, perceived throughput at the application has not always increased accordingly and the bottleneck has moved to the communications processing part of the system. The issues that cause the performance bottlenecks in the current transport protocols are discussed in this thesis, and a further study on a high speed transport protocol which tries to overcome these difficulties with some unique features is presented. By using the Systems of Communicating Machines (SCM) model as a framework, a refined and improved version of the formal protocol specification is built over the previous work, and it is analyzed to verify that the protocol is free from logical errors such as deadlock, unspecified reception, unexecuted transitions and blocking loops. The analysis is conducted in two phases which consists of the application of the associated system state analysis and the simulation of the protocol using the programming language ADA. The thesis also presents the difficulties encountered during the course of the analysis, and suggests possible solutions to some of the problems.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	OBJECTIVES	3
C.	SCOPE AND ORGANIZATION OF THE THESIS.....	4
II.	PROBLEMS WITH EXISTING TRANSPORT PROTOCOLS.....	5
A.	OPERATING SYSTEM OVERHEAD	6
B.	TIMERS AND ROUND TRIP DELAY ESTIMATION	7
C.	NON-STANDARD PACKET FORMATS	8
D.	GO-BACK-N METHOD OF ERROR RECOVERY	8
E.	FLOW CONTROL TIED TO ERROR DETECTION AND RECOVERY ...	10
III.	SNR TRANSPORT PROTOCOL	12
A.	DESIGN PHILOSOPHY	12
B.	MODES OF OPERATION.....	14
C.	MACHINE ORGANIZATION AND GENERAL OVERVIEW	15
D.	SERVICES PROVIDED.....	17
1.	Quality of service.....	17
2.	Multiplexing, demultiplexing	17
3.	Connection management.....	17
4.	Sequenced delivery	19
5.	Flow control.....	19
6.	Error recovery	20
E.	PACKET FORMATS AND THEIR TRANSMISSIONS	20
1.	Transmitter and Receiver Control Packets	21
a.	Control Packet Transmission Periods.....	21

b.	Control Packet Formats	23
2.	Data Packet Formats	25
F.	COMMUNICATION STRUCTURES	25
1.	Buffers	25
2.	RECEIVE and AREC	26
3.	LUP Table.....	26
IV.	THE SPECIFICATION MODEL:	
	SYSTEMS OF COMMUNICATING MACHINES (SCM)	29
A.	DEFINITION OF THE SCM MODEL	29
B.	SYSTEM STATE ANALYSIS.....	31
1.	Definitions	31
2.	Analysis Algorithm.....	32
3.	Comparison with the Global Analysis.....	33
V.	FORMAL SPECIFICATION	35
A.	COMMUNICATION STRUCTURES FOR THE SPECIFICATION	35
1.	Communication Channels.....	35
a.	T_CHAN	36
b.	R_CHAN	37
2.	Buffers	37
a.	OUTBUF	37
b.	INBUF	39
3.	RECEIVE.....	40
4.	AREC.....	40
B.	FINITE STATE MACHINE DESCRIPTIONS.....	41
1.	Machine T1	41
2.	Machine T2	44
3.	Machine T3	46
4.	Machine T4.....	48

5. Machine R1	50
6. Machine R2.....	51
7. Machine R3.....	53
8. Machine R4.....	55
C. SUBROUTINES	56
1. Subroutines used by the transmitter.....	56
2. Subroutines used by the receiver	60
VI. ANALYSIS.....	66
A. SYSTEM STATE ANALYSIS OF THE SNR PROTOCOL.....	66
1. Software Tool For the System State Analysis of the SNR Protocol.....	67
2. Results of the System State Analysis.....	69
a. Connection Establishment Phase Analysis.....	69
b. Data Transfer Phase Analysis.....	73
c. An Improved Method	74
B. SOFTWARE SIMULATION OF THE SNR PROTOCOL	77
1. General Description	77
2. Simulation Results.....	78
VII. CONCLUSION.....	80
A. SUMMARY OF THE RESEARCH	80
B. CONTRIBUTIONS OF THIS THESIS	81
C. FURTHER RESEARCH OPPORTUNITIES.....	81
LIST OF REFERENCES.....	83
INITIAL DISTRIBUTION LIST	85

LIST OF FIGURES

Figure 1: Network, Hosts, Entities and Protocol Processors.	15
Figure 2: Machine Organization	16
Figure 3: Receiver control packet format	23
Figure 4: Transmitter control packet format	24
Figure 5: Data packet format	25
Figure 6: Machine Organization Including the Shared Variables	36
Figure 7: OUTBUF	38
Figure 8: INBUF and RECEIVE	39
Figure 9: T1 State Diagram	42
Figure 10: T2 State Diagram	44
Figure 11: T3 State Diagram	47
Figure 12: T4 State Diagram	49
Figure 13: R1 State Diagram	50
Figure 14: R2 State Diagram	52
Figure 15: R3 State Diagram	54
Figure 16: R4 State Diagram	55
Figure 17: Transmitter Subroutines	58
Figure 18: Receiver Subroutines	62
Figure 19: Algorithm of procedure ANALYZE	68
Figure 20: Connection Establishment System State Analysis	70
Figure 21: Part of Mode-0 Analysis	73
Figure 22: Sample Analysis Using Indexed Transitions	76
Figure 23: General Task Structure	77

I. INTRODUCTION

A. BACKGROUND

The invention of the fiber optic cable has the same significance in the world of telecommunications as the invention of transistor in electronics. Now that we can transmit gigabits of information per second, we can realize the dreams of the past. This technology is still under development and it will continue to achieve yet higher data rates. Ultimately all information will be digitized, and the networks will move bits representing voice, TV, high definition TV, computer data, etc. thousands of times faster than the current networks.

However, the current implementations of communications protocols cannot fully utilize this potential. The throughput at the application has not increased in proportion to the network speed. So, instead of being able to achieve gigabits per second, the user can at best achieve the maximum throughput and end-to-end delay available from his communications protocol processor which is usually only a small fraction of the transmission bandwidth [HEAT89]. The communications processing part of the system has become the bottleneck today. Consequently, a tremendous amount of research has been devoted to the development of current standards or their implementations to match the data rates of fiber optic networks.

The transport protocol layer of a communications system is the first layer which provides an end-to-end connection through the network, and it is the keystone of the whole architecture. This layer, especially in connectionless protocols has considerable functionality, and is typically executed in software by the host processor at the end points of the network. It is thus a likely source of processing overhead and may be responsible for the low throughput of the whole system [CLAR89].

There are two approaches to improve the speed of the transport protocols [STAL91]:

- Improve the performance of the existing protocols,

- Design new protocols with the networking environment clearly in mind.

The defenders of the first approach claim that the protocols are not in fact the source of the overhead often observed in packet processing, and that the current protocols can support very high speeds if they are properly implemented. These researchers try to improve methods of optimization (like header prediction), or interfacing the protocol with the host operating system and the rest of the environment in order to gain better performance from the implementations of the standard protocols.

Most of the research work done in this area is usually concentrated on two popular protocols: classes of protocols which implement the Open Systems Interconnection Model (OSI)¹ and the Transmission Control Protocol (TCP/IP)². In [HEAT89], the authors try to identify the factors that affect performance in implementations of the layers of the OSI protocols. They note that all transport level protocols offering the same services as TP4 have certain implementation problems in common such as timer management, buffer management, connection state management, transfer of data from the user, division of the protocol processing into processes, interprocess communication, scheduling, and that the choices made in solving these problems in a particular implementation have a dramatic effect on performance. In [CLAR89], the authors present the results of an analysis made on the processing overhead of TCP, and conclude that the reasons for the slowness of the protocol lie in the implementation and the environmental factors. They feel that the experience gained with the current protocols can be effectively used to improve them and that casting the protocols in silicon may yield inflexible protocols which cannot be made to work better.

¹ The OSI reference model, which is developed by the International Organization for Standardization (ISO), is a framework for defining standards to link heterogeneous computers. For more information see [STAL91].

² TCP is a transport protocol from the Internet protocol suite. It is always used on top of a network level protocol called Internet Protocol, and commonly known as TCP/IP. For more information see [STAL91].

Other researchers defending the second approach try to design new protocols from scratch which make best use of the high speed networks. These protocols are called lightweight transport protocols. Examples of experimental lightweight protocols include NETBLT (Network Bulk Transfer), VMTP (Versatile Message Transaction Protocol), XTP (Express Transfer Protocol) and SNR (Sabnani, Netravali and Roome AT&T Bell Labs Protocol, also called high speed transport protocol) protocols.

The major goal of all of these protocols is high throughput. For this purpose, NETBLT uses a rate control scheme (packets per second) which is based on the network congestion, groups packets into blocks and uses selective retransmission error recovery. VMTP also uses packet grouping and selective retransmission, however, instead of using rate control, it chooses to transmit large groups of packets in a burst as fast as the network allows, which is found to be more efficient in processing cost. XTP is designed for hardware implementation, and it combines the transport layer with the network layer. Flow control in XTP is achieved through the use of parameters which provide visibility of the receiver's buffer to the transmitter. In addition to that, it also uses rate control and selective repeat method of error recovery.

B. OBJECTIVES

This thesis is on one of those lightweight protocol mentioned above, the SNR protocol. The SNR transport protocol is an attempt to overcome the difficulties experienced by the current transport protocols with some unique features which are different than the features of the other lightweight protocols. It was first introduced in [NETR90] by using the Communicating Finite State Machines (CFSM) model, and in [MCAR92] a formal specification was given by using the Systems of Communicating Machines (SCM) model.

This thesis will present the results of a further study on this protocol which consists of

- (i) Refining and improving the SCM specification given in [MCAR92],
- (ii) Applying of the associated *system state analysis* to the protocol,
- (iii) Simulating the protocol by using a high level programming language (ADA).

One of the two major goals of this thesis is to improve the specification by completing the missing points. The second goal is to verify the specification after the improvements. For this purpose, first the *system state analysis* will be applied to the protocol, and then the protocol will be simulated.

C. SCOPE AND ORGANIZATION OF THE THESIS

It is intended to avoid getting into the implementation details while making the specification and simulating the protocol. The original protocol is improved and some details are made clearer in order to apply the system state analysis and the simulation. However, the goal was always to stay close to the original specification and to add only what was needed. These modifications will be pointed out as necessary, however the original specification presented in [NETR90] and the SCM specification presented in [MCAR92] will not be repeated here.

The rest of the thesis is organized into six chapters. Chapter II discusses the reasons why the existing transport protocols cannot reach the high speeds required by the fiber optic media. Chapter III introduces the SNR transport protocol and its solutions to the problems encountered by the current protocols. Chapter IV defines the SCM model which is used to formally specify the protocol and the associated analysis method, system state analysis. Chapter V includes the formal specification with SCM model, and Chapter VI summarizes the results of the system state analysis study and simulation of the protocol. Finally, Chapter VII provides the conclusion of the thesis.

II. PROBLEMS WITH EXISTING TRANSPORT PROTOCOLS

One important consideration in the design of the conventional transport protocols was not to saturate the transmission media with too high data rates. The protocols were faster than the underlying media and the throughput was limited by the bandwidth. Therefore, methods were developed to decrease the number of bits transmitted at the expense of increased processing overhead. An example of this is the variable length packets. In spite of this, the processing speeds were still higher than the bit rates provided by the media.

To give an idea about the data rates of the traditional networks, characteristics of some of the constituent networks of DARPA Internet¹ are shown in TABLE 1 (taken from [STAL91]). Another example is TYMNET, which was developed to provide connection of terminals to central time sharing computers. Typical data rates of TYMNET are 9600 bps for land links and 56-kbps for satellite links.

TABLE 1 : DARPA INTERNET NETWORK CHARACTERISTICS

Network Type	Message Size (Octets)	Speed ^a	Delay ^b	Guaranteed Delivery	Notes
ARPANET	1008	Medium	Medium	Yes	WAN
SATNET	256	Low	High	No	Satellite network
Pronet	2048	High	Low	Yes	LAN
Ethernet	1500	High	Low	Yes	LAN
Telenet	128	Low	Medium	Yes	WAN
Packet radio	254	Medium	Medium	No	Varying topology
Wideband	2000	High	High	No	Satellite network

^aLow speed is < 100 kbps; medium speed is 100 kbps to 1 Mbps; high speed is > 1 Mbps.

^bLow delay is < 50ms; medium delay is 50 to 500 ms; high delay is > 500 ms.

¹ DARPA Internet is an internet project supported by DOD which consists of over 150 interconnected networks. For more information see [STAL91] and [DOD83].

These examples show that none of the conventional networks even come close to the gigabit per second data rate level of a fiber-optic network. With the improvement of the fiber optic technology, the networks became faster. However, the processing speeds did not increase at the same rate and the bottleneck has moved to the communications processing part of the system.

In this chapter, several problems which are hindering utilization of the full potential offered by the fiber optic technology will be discussed.

A. OPERATING SYSTEM OVERHEAD

An analysis has been done on one of the most commonly used transport protocols, TCP, and the results are summarized in [CLAR89]. The authors have found the operating system to be the most pronounced overhead. The following paragraph is an excerpt taken from that paper:

The first overhead is the operating system, since packet processing requires considerable support from the operating system. It is necessary to take an interrupt, allocate a packet buffer, free a packet buffer, restart the I/O device, wake up a process (or two or three), and reset a timer. In a particular implementation there may be other costs that we did not identify in this study.

In a typical operating system, these functions may turn out to be very expensive. Unless they were designed for this function, they may not match the performance requirements at all.

Even if the future protocols are implemented in hardware, the operating system will continue to be a source of overhead since the protocol must be interfaced with the host operating system. However, measures can be taken to get the best support from the operating system, some of which are listed below:

- Parallel processing of independent functions of the protocol,
- Avoiding the movement of data in the memory, since this is the most costly operation in packet processing,
- Making minimal use of operating system timer package.

As the operating systems become increasingly faster, the processing times of the protocols decreases. Nevertheless, the operating system will continue to be a challenge to the future protocol designer.

B. TIMERS AND ROUND TRIP DELAY ESTIMATION

Timer mechanisms are the backbones of current transport protocols. The only way of recovering from channel losses and performing error recovery is through the use of timers. There must be a timer associated with every data packet, if a positive acknowledgment scheme is to be used. Each time a packet is transmitted or received, a timer must be set, monitored, cleared and reset. The use of timers is a great burden and has significant contribution to the processing overhead.

Besides being difficult to manage, another important problem associated with the timers is the calculation of the reset values. A retransmission timer should be set to a value slightly longer than the round trip delay (RTD). If the value is too small, there will be many unnecessary transmissions, wasting network capacity and delaying transmissions of new packets. If the value is too large, the protocol will be slow to respond to data packet losses. Worse, the round trip delay is variable even under constant load and statistics of the delay will vary with changing network conditions. Those problems become duplicated with high speed networks, since estimating the timer value a fraction of a second off might mean wasting thousands of packets.

Many solutions have been proposed to solve this problem, each of which have its own drawbacks. The SNR protocol suggests a different approach by using counter variables instead of explicit timers. The details of this method will be explained in the next chapter. This method does not require a timer to be maintained associated with each data packet and variations of the round trip delay are automatically reflected onto the retransmission timeout values. Once an average estimate of the RTD is obtained, the protocol naturally adjusts the retransmission frequency in a very simple way.

C. NON-STANDARD PACKET FORMATS

Since the current protocols are virtually faster than the underlying conventional networks, the major concern in current protocols is not to overflow the transmission channel. Therefore these protocols use variable size packets that are just large enough to fulfill the need. Moreover, redundant transmission of packets are prohibited. This is done in expense of increased overhead, since the variable length packets increase the amount of processing time at the receiver due to decoding operations. With slower networks, this increase in the processing time was not a problem, but as the networks got faster, the problem became more and more noticeable. In fact, the researchers are looking for ways of reducing this overhead like header prediction methods even with the current networks [CLAR89].

With fiber optic networks, the situation is just the reverse. The bandwidth is so large that trying to minimize the packet length is a wasted effort. Also, the overhead increase caused by the non-standard packet formats can be very costly for these high speed networks. Increasing the processing time only for a fraction of a second might mean wasting useful time during which thousands of packets could be transmitted.

The obvious solution is using standard packet formats. The advantages of standardized packets that help improvement of the processing times can be summarized as follows:

- Several components of the packets can be processed in parallel and routed to their appropriate places within the receiver's architecture.
- Decoding operations at the receiver are not necessary,
- Easy hardware implementation.

D. GO-BACK-N METHOD OF ERROR RECOVERY

The shortcomings of the current protocols mentioned so far were only increasing the packet processing times. A more serious defect of current protocols is the utilization of go-back-N method of error recovery. In this method, when the receiver detects the loss of a packet, it sends a negative acknowledgment message (NAK) to the transmitter, requesting

retransmission of all the packets after the last correctly received packet. If the data rates are high or the transmission channel is long, this method may require many good packets to be retransmitted, which may be quite costly. Consider the following example:

Assume that data is being transmitted over a 2000 km transmission channel with 1000 bit packets, and that go back-N method is being used. Assume further that the very first packet gets lost. When the receiver starts receiving packet number 2, it detects the loss and sends a NAK-1 message. This message arrives at the transmitter one *RTD* period later after it starts transmitting packet 2, which is 20 ms in this case (ignoring the packet processing time at the receiver). By using the formula $RTD \times (\text{data rate}) / (\text{packet size})$ to calculate the number of outstanding packets at one time, the following results can be obtained:

(i) If the data rate is 50,000 bps (which is the case of a traditional network), then there can be at most one outstanding data packet when the transmitter receives the NAK message. In fact, the transmitter will have just finished transmitting packet number 2.

(ii) If a fiber optic network with 1 Gb/s data rate is being used, the transmitter may have transmitted up to 40,000 data packets when the NAK arrives.

In either case, according to the go-back-N method, all the packets must be retransmitted. This means wasting 1 packet in a traditional slow network versus 40,000 packets in a fast network, which is clearly unacceptable.

Therefore, the go-back-N method of error recovery can cause significant loss of throughput. To overcome this problem, the new experimental lightweight protocols are oriented for selective repeat method of retransmission in which only the lost packet is resent. One difficulty with the selective repeat retransmission is that large tables must be maintained and rather complicated error recovery algorithms must be utilized. To avoid these difficulties, most of the high speed protocols use the concept of blocking, which is also adapted to the SNR protocol. Detailed explanation of using this method in the protocol is left to the following chapters.

E. FLOW CONTROL TIED TO ERROR DETECTION AND RECOVERY

A conservative flow control scheme which uses the sliding window technique may limit the throughput of the protocol in long-delay situations [STAL91]. This is because the sliding window technique normally does not decouple acknowledgments from flow control. To clarify this point, consider the following situation:

Assume that the receiver has several window sizes of buffer space which is shared between different logical connections. When the transmitter uses up all its credit by transmitting a whole window of packets, it has to stop and wait for the window size to be increased by acknowledgments. If the receiver acknowledges the first n packets, then the transmitter increases its window size by n , allowing n more packets to be transmitted. However, if the first packet of a window gets lost, then the receiver cannot acknowledge any of the rest of the packets and has to wait for the lost packet to be recovered before it increases the window size. On the other side of the network, the transmitter also has to wait, since its credit has expired, and it cannot transmit any more packets. Therefore, both the transmitter and the receiver wait idly for the retransmission timer to expire, doing no useful work. If the network is a high speed network, this wait might mean wasting time during which thousands of packets could be transmitted.

This example shows how error control halts the flow of packets even if the receiver has enough buffer space. A better solution could be using a credit scheme which would allow advancing the lower edge of the window, thus letting the transmitter to transmit more packets without acknowledging any of the previously transmitted packets [STAL91].

These two functions have to be separated in order not to stop the flow because of the lost packets. The transport protocols of the future will use rate control to slow down the transmitter before the buffer limit has been reached rather than stopping the flow when the buffer is full or a loss is detected. Another solution proposed in [MCAR92], which is convenient to use in the SNR protocol, is to use prediction method. In this method, the transmitter assumes that a packet is lost if a certain number of packets following the unacknowledged packet has been acknowledged by the receiver, and retransmits the packet

before its timer expires. The specification given in this thesis does not include this feature, however it is considered to be a potentially useful method.

III. SNR TRANSPORT PROTOCOL

The reasons for the insufficient processing speeds of the existing transport protocols to match the high data transmission speeds of optical telecommunication technology were explained in the previous chapter. In this chapter, the basic ideas and the building blocks of the SNR Transport Protocol will be presented, together with a description of how it provides the necessary services expected from a transport protocol. Then, in the following chapter, the formal specification of the protocol will be given using the SCM model.

The definitions of the abstract communication structures given in this chapter will be parallel to the definitions given in [NETR90] to keep the originality. The extensions and modifications made to the protocol for analysis and simulation purposes will be presented in Chapter V.

A. DESIGN PHILOSOPHY

The purpose of the SNR protocol is to overcome the processing bottlenecks and insufficiencies experienced by the existing transport protocols. The key idea in the design of the SNR protocol is to provide a high processing speed by simplification of the protocol, reduction of the processing overhead and utilization of parallel processing. In order to achieve these goals, the following design principles are observed:

- Periodic exchange of complete state information and eliminating explicit timers,
- Using selective repeat method of retransmission,
- Using the concept of blocking,
- Parallel processing.

Periodic exchange of complete state information. In most of the current protocols, changes in state are exchanged only when certain events occur, such as detected loss of a packet, buffer overflow, etc. This requires using a number of nonstandard and variable size packet formats and elaborate error recovery procedures which complicates the protocol

processing significantly. Therefore, to avoid these complications and keep the protocol as simple as possible, a different approach is taken in the SNR protocol by allowing exchange of complete state information between the receiver and the transmitter on a frequent and periodic basis. This technique increases the number of packets transmitted, but this increase in the number of packets is negligible compared to the very high bandwidth of fiber and the speedup it provides in processing. In addition to simplifying the protocol, this technique has two more important advantages: First, it allows parallelizing the protocol processing and therefore leads to higher performance. Secondly, with this method, the loss of a control packet is not a problem since the information in the next control packet will supersede all the previous information, thus, the recovery procedures required to overcome the lost control packets are not needed. For the purpose of state information exchange, transmitter and receiver control packets are used. These packets have standard sizes and formats for faster processing.

Another very important consequence of this scheme is that it helps elimination of the explicit timers used for error recovery purposes. This point will be left abstract for now, but when the error recovery structure is explained later in this chapter, it will be clarified

Using selective repeat method of retransmission and blocking. As it was pointed out before, go-back-N error recovery method potentially wastes network resources by causing thousands of good packets to be retransmitted. Therefore, like other protocols designed for high speed networks, this protocol also uses selective repeat method of retransmission, and the concept of blocking. Blocking reduces the overhead of maintaining large tables and complex procedures that are required for selective repeat procedures. A group of packets (typically 8) is called a block. All of the packets constituting a block are transmitted and handled separately by the network. Upon successful reception of all of the packets in a block, the receiver acknowledges the block, rather than the individual packets. If a packet in a block is faulty or missing then the whole block of packets are retransmitted.

This method is slightly different than a pure selective repeat algorithm in which only the faulty packets are retransmitted, as it causes retransmission of other good packets in the

block. But it must be remembered that the fiber-optic media supporting the transport protocol has very low error rates and retransmissions do not occur as frequently as in other types of media, which the current transport protocols were designed for. On the other hand, it is expected that the simplicity and high processing speeds gained this way will compensate for this additional load.

Parallel processing. As mentioned in Chapter II, one important source of overhead for the transport protocols is the underlying operating system. Therefore, efficient use of operating system resources is essential. This idea immediately calls for the concurrent execution of several independent functions of the protocol, namely parallel processing. As it will be explained shortly, the SNR protocol is composed of eight machines (four in the transmitter and four in the receiver), each of which perform a specific function. These machines operate almost independently with a small amount of interaction between them. Consequently, the protocol automatically lends itself to parallel implementation. Since the function of each machine is simpler than the protocol as a whole, they can also be implemented on hardware with ease. This contributes significantly to the improvement of the throughput performance.

B. MODES OF OPERATION.

In order to give some flexibility to the protocol, the following three modes of operation are specified:

Mode 0 has no error control or flow control. It is suited for virtual circuit networks and for the cases where quick interaction between the communicating entities is desired (e.g. terminals connected to a host) and short packets are used.

Mode 1 has no error control but provides flow control. This mode is suitable for real time applications such as packetized voice or real-time monitoring of a remote sensor where error control is not needed and packet sizes are small. This mode is also convenient if the underlying network is reliable (like type-A network defined by ISO¹).

Mode 2 has both error control and flow control. This is the most reliable mode and it is useful for large file transfers in all types of network services.

C. MACHINE ORGANIZATION AND GENERAL OVERVIEW

The protocol can be envisioned as connecting two host computers end-to-end across a high speed network as shown in Figure 1:

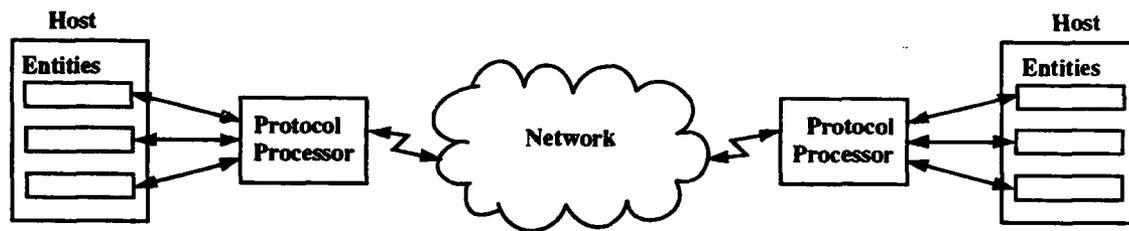


Figure 1: Network, Hosts, Entities and Protocol Processors.

This protocol requires a full duplex link between two host systems. Each host system in the network consists of eight finite state machines (FSM), four for executing the transmitter functions, and four for executing the receiver functions. Since an entity in a host (such as application programs, file transfer packages, electronic mail facilities and terminals) can either transmit or receive data, it can only use either the transmitter machines or the receiver machines of the protocol. Therefore the protocol only specifies the operations of the transmitter machines in the transmitting host and the operations of the receiver machines in the receiving host. In this thesis, multiplexing of the entities will be considered as an implementation detail and will be omitted.

The general organization of the machines is shown in Figure 2. Each machine in the protocol performs a specific function in coordination with other machines. The

¹ Three types of network service are defined by ISO: Reliable network service (Type-A), failure-prone network service (Type-B) and unreliable network service (Type-C). For more information see [STAL91].

coordination is established by communicating through some *shared variables* which will be explained later in the thesis.

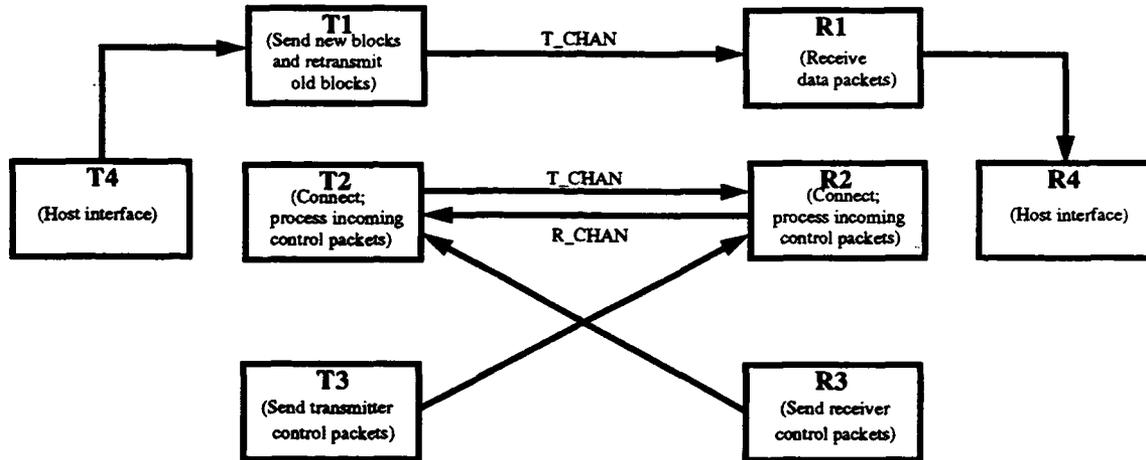


Figure 2: Machine Organization

Machine T1 is responsible for the transmission of new data packets and retransmission of old packets as necessary through the transmitter channel. Machine T2 first establishes the connection with the receiver and thereafter processes the incoming receiver control packets and updates related tables and variables as the blocks are acknowledged. Machine T3 sends transmitter control packets to the receiver periodically through the transmitter channel. Machine T4 is the host interface of the transmitter. It inserts the incoming data stream into the buffer for transmission by machine T1.

Machine R1 removes the data packets from the transmitter channel and inserts them into the buffer in order according to their sequence numbers and updates the related variables and tables in the receiver. Machines R2 and R3 are receiver counterparts of transmitter machines T2 and T3. Machine R2 replies the connection request messages sent by machine T2. After the connection establishment, it receives the transmitter control packets. Machine R3 sends the receiver control packets at periodic intervals through the

receiver channel. Machine R4 is the host interface of the receiver. It retrieves the data packets from the buffer if they are in sequential order and passes them to the host.

D. SERVICES PROVIDED

The protocol provides for the following general services:

- Quality of service,
- Multiplexing, demultiplexing,
- Connection management,
- Sequenced delivery,
- Flow control,
- Error recovery.

1. Quality of service.

As explained previously, the protocol provides for three modes of operation. The number of services provided and quality of transmission depend on the selected mode of operation: Mode 0 does not have error recovery, sequenced delivery and flow control, and mode 1 does not have error recovery. Mode 2 is the most reliable mode which has both error control and flow control functions. The users of the protocol can choose the operation mode depending on their needs. Although the protocol does not specify dynamic change of operation mode, this feature can also be added to the protocol with minor changes.

2. Multiplexing, demultiplexing

Multiplexing and demultiplexing is done in all three modes. The front-end processor implementing the protocol is thought of as connecting several hosts and many logical connections within each host to the network (see Figure 1).

3. Connection management

The transmitting host is responsible for establishing and terminating connections. The initial connection establishment phase is based on the standard three-way handshake in which the following parameters are negotiated: mode of communication, peak

bandwidth per connection, packet size, block size, buffer required at the receiver in units of blocks. In addition, it is expected that an estimate of the round-trip delay (RTD) would be available during the connection set-up.

Connection establishment: The machines involved in connection establishment phase are transmitter machine T2 and receiver machine R2. Upon reception of transmission signal from the host, the transmitter sends a connection request (*Conn_req*) message through machine T2, which includes the parameter values that the transmitting entity wants to use. Machine R2 of the receiving host evaluates the requested parameters and responds by sending a connection acknowledgment (*Conn_ack*) message that includes the modified values for the parameters under which the receiver can operate. If machine T2 cannot get a *Conn_ack* in a certain period of time, it retransmits the *Conn_req* again. After a number of unsuccessful attempts, the transmitter quits and notifies the host of unsuccessful connection.

When machine T2 receives the *Conn_ack*, it evaluates the parameters and if the parameter values are acceptable by the transmitter, then it transmits a connection confirmation (*Conn_conf*) message and the data transfer phase starts, otherwise, the transmitter rejects the connection and notifies the host of the failed connection attempt. The receiver goes into data transfer phase with the reception of the confirmation message, the first transmitter control packet or the first data packet.

Connection termination: Connection is normally terminated by a disconnect (*Disc*) message sent by machine T2 after transmission of all the data packets. Either side can also abnormally terminate the connection if they cannot receive any control packets for a long time. In this case, the transmitter sends a *Disc* message and leaves the network, and the receiver just aborts the connection after it timeouts. Thereafter, the host interface machines (T4 and R4) notify their hosts of the abnormal disconnection.

4. Sequenced delivery

Especially in connectionless data transfers there is a risk that the data packets will not arrive in the order in which they were sent, because they may traverse different paths through the network. This protocol provides for sequenced delivery in modes 1 and 2. It is the receiver's responsibility to reorder the data packets according to their sequence numbers. The basic structure used in reordering is the buffer at the receiver.

This service also includes the detection of duplicate packets. Packets may be duplicated either by the network or because of retransmissions. To detect duplicate packets, the sequence numbers of the packets are used. Each time a packet is received (data packet or control packet), a check is made to see if the packet has already been received. If the packet is a duplicate, it is discarded.

5. Flow control

Flow control is done only in modes 1 and 2. For this purpose, the receiver writes the available buffer space it has in units of blocks into the *buffer_available* field of the receiver control packets. The following variables are used in the transmitter for flow control purposes:

L is the maximum window size in units of blocks. It is chosen to be slightly larger than $\left[\frac{RTD \times \text{maximum bandwidth}}{\text{(number of bits in a block)}} \right]$.

NOU is the number of outstanding blocks which have been transmitted but not acknowledged yet. Every time the transmitter completes the transmission of a block of packets, it increments NOU , and every time it receives a receiver control packet, it decrements NOU by the number of acknowledged blocks. The transmitter starts transmission of another new block only if NOU is less than L and *buffer_available* is greater than NOU . For retransmissions, this check is not necessary since the retransmitted block already has a reserved buffer space in the receiver (NOU is decremented when the block is first transmitted).

6. Error recovery

The protocol provides for this service only in mode 2 operation. In this mode, if a block has not been acknowledged for a predetermined amount of time, then all of the packets constituting this block are retransmitted. The transmitter maintains necessary structures for error recovery which will be explained later.

Since there is no error recovery in mode 1, a problem is encountered when the packets get lost during the data transfer in this mode: for how long should the receiver wait for the lost packets and what action should be taken? This problem has not been addressed in the original protocol and later in [MCAR92]. In this thesis, a solution will be suggested which uses the following approach: if a packet which is to be retrieved is missing, then wait until two packets with a higher sequence number is received and then skip the missing packet and set the corresponding *RECEIVE* array bit to 1 as if it were received correctly. The reason for waiting for the reception of two packets is that it is the smallest number of packets that the receiver should wait after a missing packet to perform its reordering functions (second received packet could be the missing one). A higher number can be selected if the loss rate is low but packets are considerably disordered.

E. PACKET FORMATS AND THEIR TRANSMISSIONS

The packets exchanged between the transmitter and the receiver can be classified into four categories: connection control packets, transmitter control packets, receiver control packets and data packets.

Connection messages include connection request (*Conn_Req*), connection acknowledgment (*Conn_Ack*), connection confirmation (*Conn_Conf*), and disconnect (*Disc*) messages, and they are used for connection management purposes (connection establishment and termination). These packets do not carry any formation related to data transfer so they can have appropriate formats much like the formats used in current protocols. In what follows, calculation of the transmission periods and formats of the control packets, and data packet formats are explained.

1. Transmitter and Receiver Control Packets

Control packets have two purposes: First, they are used for periodic transmission of complete state information of the communicating entities, secondly, the communicating entities get informed of each other's existence through the reception of control packets even when the data transfer ceases for a while.

a. Control Packet Transmission Periods

In both the transmitter and the receiver, control packet transmission periods, T_{in} , are calculated by the formula $T_{in} = \max(RTD/kou, IPT)$, where RTD is the estimated round trip delay for the logical connection, the constant kou is typically a power of 2, such as 32, and IPT is the average time between two data packet transmissions. T_{in} is initially set to a value calculated by this formula but if no data packets are transmitted (in the transmitter) or received (in the receiver) since the transmission (or reception) of the last control packet, then it is doubled up to the limit $\max(RTD/m, IPT)$ where m is another constant (e.g. 8). Upon the transmission (or reception) of the next data packet, the inter-transmission time of control packets are decreased back to T_{in} again.

For example, if RTD is 60 ms, including the propagation delay and processing time in the receiver, and kou is chosen to be 32, then a control packet will be sent every $T_{in} = 1.875$ ms, provided that this value is larger than IPT . If the connection remains inactive for one T_{in} period, then this interval is successively increased to 3.75 ms, 7.5 ms and so on, up to the limit given by $\max(RTD/m, IPT)$. If the data packet transmission (or reception) interval IPT suddenly jumps to, say 9 ms, transmitting a control packet sooner than every 9 ms becomes redundant, since more control packets than data packets will be transmitted, and the control packets will mostly carry the same information. Therefore, in this case T_{in} is set equal to IPT , keeping the control packet transmission in proportion with data packet transmission (or reception) rate. As soon as the connection

becomes active again, T_{in} is decreased back to 1.875 ms. As it will be explained later, all this is done through the use of variables. The actual clock period is not changed.

In this formula, the constant kou is the maximum number of receiver control packets that the transmitter should expect to receive before the acknowledgment of a block. This can be explained as follows: According to the formula, if the average time between two data packet transmissions (IPT) is less than RTD/kou (meaning that the data packet transfer rate is high), then a control packet will be transmitted every RTD/kou seconds, otherwise control packet transmission interval will be equal to data packet transmission interval. Therefore, if T_{in} is equal to RTD/kou then the maximum number of receiver control packets that the transmitter should expect to receive before a block is acknowledged after its transmission is kou . On the other hand, the average time between two data packet transmissions (IPT) increases as data packet transmission rate decreases. If it finally becomes greater than RTD/kou , then T_{in} becomes equal to IPT and the control packet transmission interval increases. Control packet transmission interval also increases when T_{in} is doubled. Hence, the number of receiver control packets that the transmitter can receive before the acknowledgment of a block becomes less than kou . In a later section, it will be shown how this idea is used in the protocol for retransmission purposes.

Having explained the calculation of T_{in} , some comments are in order:

- **The event *clock-tick*:** The value of T_{in} calculated with this formula also determines the period of an event called "*clock-tick*" which is a periodic event occurring at T_{in} second intervals. This is the timing mechanism of the protocol. Its occurrence initiates an evaluation of the internal state variables to determine whether some action should be taken such as transmission of a control packet or retransmission of a connection message.

- The protocol does not specify the details of how T_{in} is changed dynamically. This, in fact, is an implementation detail and beyond the scope of this thesis. Here, it will suffice to note that the protocol allows T_{in} to be changed dynamically. To do this, (i) data

packet transmission intervals, and (ii) round trip delay times should be monitored and new values of T_{in} should be calculated by using the formula $T_{in} = \max (RTD / kou, IPT)$. This operation provides better RTD values than the first estimation to adjust the retransmission times according to the changes in RTD. However it is an optional operation and if it is not done, the retransmission timeout values will not be terribly off, since the protocol has the ability to adjust itself automatically with respect to the changes in RTD.

b. Control Packet Formats

Receiver control packets (R_state) contain the state information of the receiver. Figure 3 shows the format of the receiver control packets²:

LCI	Type = 0	Seq #	k	LW_r	<i>buffer_available</i>	LOB	Error check
-----	----------	-------	-----	--------	-------------------------	-----	-------------

Figure 3: Receiver control packet format

LCI is the logical connection identifier, which is a unique sequence number across all the logical links which the host computer is engaged in. The type field identifies the type of the packet and contains 0, 1 or 2 for receiver control packets, transmitter control packets and data packets respectively. Seq # field contains the sequence number of the packet which differs from the sequence number of the transmitter control packets or the data packets. The next four fields contain the values of the variables k , LW_r , *buffer_available* and *LOB* in the receiver just prior to the transmission of the packet. The variable k is the interval between two control packet transmissions of receiver in units of

² As shown in [NETR90] and in [MCAR92], the control and data packets formats do not contain an address field. The same packet formats will also be used in the explanations in this thesis, and it will be assumed that a convenient address field can be added to the packet formats in any implementation of the protocol.

T_{in} , LW_r is a block sequence number such that all the blocks with sequence numbers less than this have been correctly received and acknowledged, $buffer_available$ is buffer space available at the receiver in units of blocks, and LOB is a bit map representing the outstanding blocks between LW_r and $(LW_r + L - 1)$. The first bit of LOB corresponds to LW_r and is always set to 0. The other bits are set to 1 if the corresponding blocks have been received correctly, otherwise they are set to 0. LW_r and LOB fields are used together to acknowledge the blocks received correctly. The last field contains an error detection code such as a 16-bit standard CRC (cyclic redundancy code).

Transmitter control packets (T_state) contain the state information of the transmitter. The packet format is shown in Figure 4:

LCI	Type = 1	Seq #	k	UW_t	No. of blocks queued	Error check
-----	----------	-------	-----	--------	----------------------	-------------

Figure 4: Transmitter control packet format

- The first field is LCI. The second field, the type field, contains 1 to indicate that this packet contains the transmitter's state. Seq # is the sequence number of the packet. The next two fields contain the values of the variables k and UW_t of the transmitter just prior to the transmission of the packet. Similar to receiver control packets, k is the interval between two control packet transmissions of the transmitter in units of T_{in} , UW_t is the maximum sequence number of the block below which every block has been transmitted (but not necessarily acknowledged). As explained in [NETR90], the queue length can be used for a variety of purposes, such as congestion control within the network, to decide whether the receiver should accept another connection, etc. It can also be used when the transmitter does not have enough packets to complete a block. In such a case the transmitter sends a partial block and the receiver does not classify this as an outstanding block. The last field is the error check field.

2. Data Packet Formats

The format of a data packet is shown in Figure 5. The purpose of the first three fields is the same as the transmitter and receiver control packets.

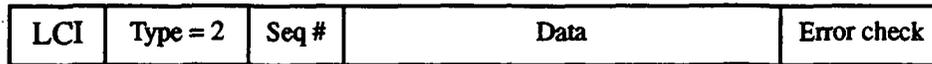


Figure 5: Data packet format

Data packet sequence numbers extend across the lifetime of the connection. The sequence number of a packet contains $p + q$ number of bits in the Seq # field. The first p bits give the sequence number of the block which contains the packet and the next q bits give the packet number in the block. Therefore, a block consists of 2^q packets and the message can have 2^p blocks.

The length of the data packets are constant throughout the connection, which is determined during the connection establishment phase. If a message does not fit into an integral number of packets then, depending on the implementation, the space in the last packet can be padded with null characters. This is intended to simplify the packet processing in the receiver.

F. COMMUNICATION STRUCTURES

1. Buffers

The original protocol described in [NETR90] requires each logical connection to have a pre-negotiated buffer at the receiver. The buffering scheme in the transmitter was not defined to provide the abstraction. In [MCAR92], the buffers in both the transmitter and the receiver were explicitly defined and included in the SCM specification. To keep the abstract protocol definition given in this chapter close to the original protocol, the specification details and modifications required for analysis and simulation, which include

the buffer structures, will not be presented here. Explanation of these modifications will be left to Chapter V.

The buffer in the receiver is used as a place to temporarily hold the data packets until they are retrieved by the receiving host. Another function of this buffer is to reorder the data packets that arrive out of sequence. Since it is guaranteed that the transmitter will never send more than one window size of data packets, this buffer can be just big enough to hold all the packets in the greatest window or larger. In other words, the size of the buffer at the receiver should be at least $(RTD \times \text{negotiated peak bandwidth})$ bits.

2. *RECEIVE* and *AREC*

RECEIVE and *AREC* are tables maintained at the receiver and are updated as new packets arrive from the transmitter. *RECEIVE* maintains information about received packets. *RECEIVE*(*i*) is set equal to 1 (0) if the *i*th data packet is received correctly (incorrectly). *AREC* maintains information about received blocks. *AREC*(*j*) is set equal to 1 (0) if the if all packets constituting the *i*th block have (have not) been correctly received.

The use of these tables provides detection of duplicate packets whose block sequence numbers are greater than LW_r , and also determining whether or not a whole block has been received for acknowledgment purposes.

3. *LUP* Table

LUP table is the structure through which error recovery is performed. It is maintained in the transmitter, and used only in mode 2. The size of *LUP* table is equal to the number of blocks in the largest window size. Each element of this table has three fields: *SEQ*, *COUNT* and *ACK*. After the transmission of a whole block of packets is completed, an entry is made into the table element whose index is calculated from the block sequence number. The block sequence number is copied into *SEQ* field, *ACK* bit is set to zero, and

the *COUNT* field is set to $(RTD / T_{in}) + cons$ where *cons* is a constant (e.g. 2). The reason for adding *cons* is to make the time-out period slightly larger than the round-trip delay therefore the block will not have to be retransmitted if the acknowledgment arrives a little late.

When a block is acknowledged, its *ACK* bit is set to 1. In effect, this action removes the block entry from the table and the table entry need not be cleared. After every reception of a receiver control packet, the *COUNT* fields of unacknowledged blocks are decremented by *k*, which is the interval between two control packet transmissions of the receiver, expressed in units of T_{in} . This number is obtained from the *k* field of the receiver control packet. A block is scheduled for retransmission only if it is not acknowledged and its *COUNT* field reaches zero. This error recovery scheme has the following useful properties:

- As it was mentioned before, this error recovery scheme eliminates the need for explicit retransmission timers through the use of counters and periodic transmission of the control packets.

- As the load on the receiver increases, or if the network is heavily loaded, the effective round trip delay through the system increases, increasing the time interval between the reception of successive receiver control packets at the transmitter. As a consequence of this, the transmitter decrements the *count* fields of the *LUP* table entries less frequently and the retransmission timeout increases automatically. This is an important property in that it eliminates the need for recalculation of RTD in order to adjust the retransmission timeout values.

- For low-activity connections, control packet transmission intervals are increased by doubling the value of *k*. However, since the counter of the *LUP* table is decreased by *k*, the effective timeout period remains the same. This property prevents unnecessary control packet transmissions.

- If the receiver is extremely busy and the number of packets waiting to be processed increases, in order to prevent “retransmission avalanche” a further enhancement can be made by increasing the receiver control packet transmission intervals while keeping the k variable constant [NETR90] (The specification presented in this thesis does not have this property.)

IV. THE SPECIFICATION MODEL: SYSTEMS OF COMMUNICATING MACHINES (SCM)

In the previous chapters, the motivation issues that led to the development of the SNR protocol are discussed and the basic structures and functions of the SNR protocol are presented. Before continuing on with the formal specification of the protocol, the particular model used to formally specify and analyze the protocol will be described in this chapter. The specification model is called *systems of communicating machines (SCM)* and the analysis method associated with it is called *system state analysis*. A more detailed description appears in [LUND88] and [LUND91].

There are various methods for protocol specification and verification, each of which has its own advantages and disadvantages. The following references present discussions on other different types of protocol specification methods: CFSM [VUON83], CSP (Communicating Sequential Processes) [HOAR78], LOTOS [BRIN85], Ada [CAST85], Estelle [BUDK87], [DIAZ89], [LINN85].

A. DEFINITION OF THE SCM MODEL

The SCM model is a formal model used for the specification of the communication protocols. It is derived from the *communicating finite state machines (CFSM)* model and attempts to reduce its disadvantages. To reduce the number of states in each machine, local variables are added, and instead of the implicit queues, shared variables are used for communication between processes. A channel may be modeled as a process explicitly, whenever appropriate. This model tries to stay close to the CFSM model, keeping as much of its simplicity as possible, however, it is also inclined toward the programming language models as can be seen from the following definition of the model:

A *system of communicating machines* is an ordered pair $C=(M,V)$, where

$$M=\{m_1,m_2,\dots,m_n\}$$

is a finite set of *machines*, and

$$V = \{v_1, v_2, \dots, v_k\}$$

is a finite set of *shared variables*, with two designated subsets R_i and W_i specified for each machine m_i . The subset R_i of V is called the set of read access variables for machine m_i , and the subset W_i the set of write access variables for m_i .

Each machine $m_i \in M$ is defined by a tuple $(S_i, s_0, L_i, N_i, \tau_i)$, where

(1) S_i is a finite set of states;

(2) $s_0 \in S_i$ is a designated state called the *initial state* of m_i ;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p, a) ,

where p is a *predicate* on the variables of $L_i \cup R_i$ and a is an *action* on the variables of $L_i \cup R_i \cup W_i$. Specifically, an action is a partial function $a: L_i \times R_i \rightarrow L_i \times W_i$ from the values contained in the local variables and read access variables to the values of the local variables and write access variables.

(5) $\tau_i: S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i .

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, R_i and W_i are the subsets of V to which m_i has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with that name is executed. The action changes the values of local and/or shared variables, thus allowing other predicates to become true.

Let $\tau(s_1, n) = s_2$ be a transition which is defined on machine m_i . Transition τ is enabled if the enabling predicate p , associated with name n , is true. Transition τ may be executed whenever m_i is in state s_1 and the predicate p is true (enabled). The *execution* of τ is an

atomic action, in which both the state change and the action a associated with n occur simultaneously.

The set L_i of local variables specifies a name and a range for each. The range must be a finite or countable set of values.

It is convenient to produce a table called *predicate-action table (PAT)* which lists each transition name and the predicate and action associated with that transition. This table, together with the FSM diagrams and the variables, make up the formal specification.

B. SYSTEM STATE ANALYSIS

The analysis method associated with the SCM model is called the *system state analysis*. This method is analogous to the *reachability analysis* of the CFSM model. However, it tries to address the two well known drawbacks of the reachability analysis: the undecidability of the finiteness of the reachability graph due to unbounded queues, and the state explosion problem for nontrivial protocols which make the analysis impractical. These problems are not trivial problems and a considerable amount of research is done to cope with them and to develop improved methods.

1. Definitions

Before getting into how the *system state analysis* is done, it is necessary to make some further definitions:

A *system state tuple* is a tuple of all machine states. That is, if (M, V) is a system of n communicating machines, and s_i for $1 < i < n$, is a state of machine m_i , then the n -tuple (s_1, s_2, \dots, s_n) is the system state tuple of (M, V) .

A *system state* is a system state tuple, plus the outgoing transitions which are enabled. That is, two system states are *equivalent* if the corresponding machines are in the same states with the same outgoing transitions enabled.

The *initial system state* is the system state such that every machine is in its initial state, and the outgoing transitions are the same as in the initial global state.

The *global state* of a system contains the system state, plus the values of all variables, both local and shared. It may be written as a larger tuple, combining the system state with the values of the variables. The *initial global state* is the initial system state, with the additional requirement that all variables have their initial values. A global state *corresponds* to a system state if the corresponding variables have the same value and the corresponding machines have the same state with the same outgoing transitions enabled. That is, a global state consists of a tuple of machine states, plus the values of all variables. A system state with the same tuple of machine states and the same enabled outgoing transitions is the corresponding system state.

2. Analysis Algorithm

The *system state analysis* consists of generating all the system states reachable from the initial system state. This is done by constructing a graph whose nodes are the reachable system states, and whose arcs indicate the transitions leading from one system state to another. Given the protocol specification, which includes the FSMs and the predicate action table, the graph is constructed as follows:

1. Generate the starting node. This node is just the initial system state where all the machines are in their initial states, and all the variables have their initial values.

2. Select an unexplored node (parent¹). From the current system state tuple and the variable values, determine all the enabled outgoing transitions according to the PAT. For each of these transitions, determine the system state which results from its execution (children²).

3. For each child, examine the rest of the graph whether an *equivalent* system state has already been generated (two system states are equivalent if every machine is in the same state, and the same outgoing transitions are enabled.) If there is an equivalent system

¹ A node which is being explored is called a *parent*.

² A new node which is generated from the explored node is called a *child*.

state, then draw an arc from the current state to it, labeling the arc with the transition name. Otherwise add the new system state to the graph, draw an arc from the current system state to it, and label the arc with the name of the transition.

4. Repeat step 2 and 3 until no more new states are generated.

3. Comparison with the Global Analysis

It is also possible to make a global analysis of the SCM specifications. A global analysis is done exactly the same way as a reachability analysis: all the reachable global states are generated and two global states are considered to be equivalent only if all the machines are in the same state and all the variable values are the same. For the global analysis, it is not necessary to compare the outgoing transitions of the equivalent states, since if the machines are in the same states and the variable values are the same, then the outgoing transitions will be the same.

The third step of the algorithm described above is the reason why the *system state analysis* generates less states than a global analysis. The reason is that if the equivalent system state of a new child is found, then the child is deleted, and hence, none of the dependent system states, which would have been generated if the child had been explored, are generated. This is also the property of the *system state analysis* which makes it eventually terminate even if the number of global states are infinite.

On the other hand, note that if the values of all variables are restricted to some finite range, then the model can theoretically be reduced to a simple finite state machine. Otherwise, an infinite number of global states are possible. However, even if the number of global states is infinite, the number of system states is finite, because of the finiteness of each machine. This may allow a reachability analysis on the system states, when a reachability analysis on the global states is infinite. Even when the values of all variables are of a finite range, the number of global states in the equivalent FSM system may be so large as to be intractable. This model reduces these difficulties for some specific protocols.

The SCM model and the *system state analysis* was applied to a number of common protocols including the token ring protocol, CSMA/CD protocol, token bus protocol, FDDI protocol and a general data transfer protocol with variable window size (e.g., HDLC). In Chapter V of this thesis, the specification of the SNR protocol with the SCM model will be reproduced and refined. In Chapter VI, the results of applying the *system state analysis* to the protocol will be presented.

V. FORMAL SPECIFICATION

The basic ideas and building blocks of the SNR transport protocol were presented in Chapter III. In this chapter, a formal specification of the SNR transport protocol will be made by using the SCM model, which was introduced in Chapter IV. This model consists of Finite State Machine (FSM) specifications and a Predicate-Action Table (PAT) containing the enabling predicate and action for each transition. In order to apply the system state analysis and to simulate the protocol, the basic concepts introduced in Chapter III had to be improved and the degree of abstraction had to be reduced, i.e. the data structures and the operations on these structures had to be described, too.

The first section of this chapter defines these improved communication structures which are more detailed versions of the communication structures given in [MCAR92]. Next, the FSMs are described in detail and the predicate-action tables (PAT) of each machine in the protocol are given. Then, in the subsequent section, the subroutines which are used in the predicate-action table are described.

A. COMMUNICATION STRUCTURES FOR THE SPECIFICATION

To illustrate of discussions in this chapter, the machine organization diagram presented in Chapter III is extended to include some of the communication structures as well, and is depicted in Figure 6. This figure also illustrates the global shared variables in the transmitter and the receiver (local variables are not shown for clarity).

1. Communication Channels

As it was stated in Chapter III, this protocol requires a full duplex link between two communicating entities. The logical links connecting the two entities are modeled as queues which are called "communication channels" in the specification of the protocol.

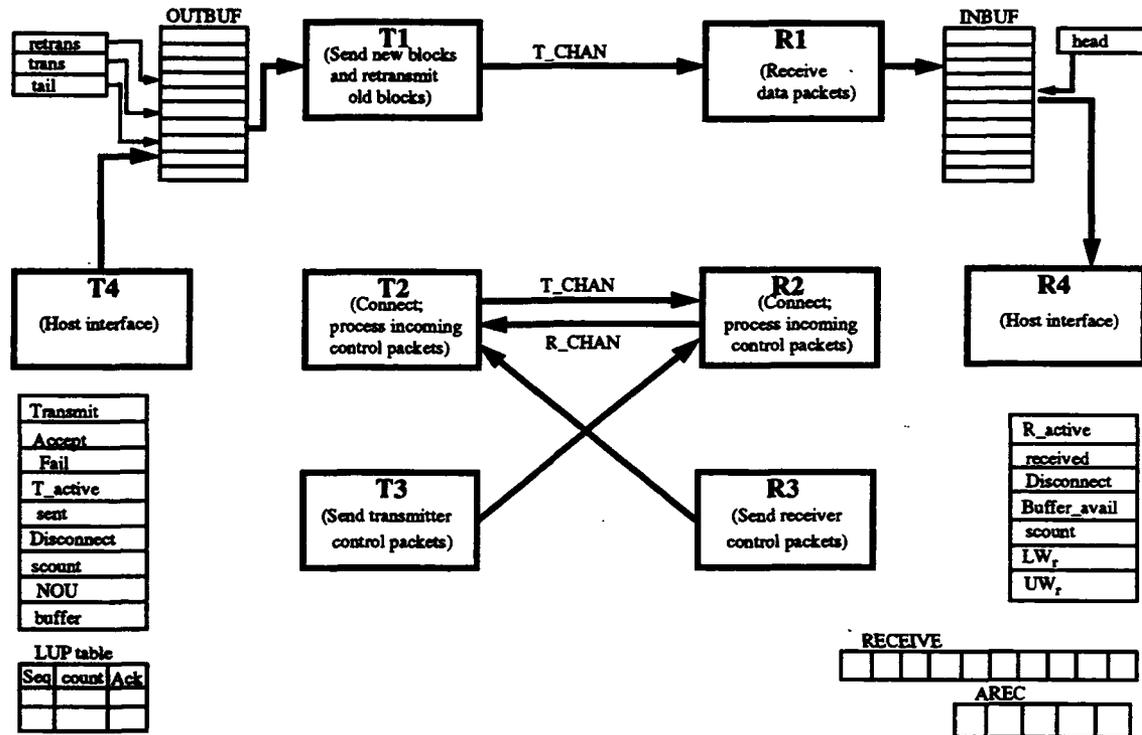


Figure 6: Machine Organization Including the Shared Variables

This model of the links differs from the actual link in that an actual link carries the packets as a stream of bits from the transmitter to the receiver, whereas this model assumes that the packets are transmitted as units by being inserted at the end of a queue. Application of this model into real world situations is an implementation detail and will not be considered in this thesis. The communicating entities see themselves attached to these communication channels where message deletion and reordering is allowed, as it would be expected from a real network. The required full duplex connection consists of two communication channels explained below, which also appear in [MCAR92].

a. T_CHAN

T_CHAN is the channel from the transmitter to the receiver. This is the channel in which the transmitter dumps connection requests, connection confirmation messages, data packets, transmitter control packets and disconnect messages.

The machines connected to *T_CHAN* can be seen in Figure 6. On the transmitter side, machine T2 enqueues all the messages related to connection management operations, machine T1 enqueues data packets, and machine T3 enqueues transmitter control packets into *T_CHAN*. On the receiver side, machine R2 dequeues connection management messages and transmitter control packets, and machine R1 dequeues data packets.

b. R_CHAN

R_CHAN is the channel from the receiver to the transmitter. This channel carries the connection acknowledgment messages and receiver control packets sent by the receiver. On the receiver side, machine R2 enqueues the connection acknowledgment messages and machine R3 enqueues the receiver control packets into this channel. On the transmitter side, both kinds of messages are dequeued by machine T2.

2. Buffers

Figure 6 shows the buffers used in the transmitter and the receiver. Here, the buffers and the pointers used with the buffers will be explained. It will be assumed that the data stream is already divided into packets by the host, and that data can be moved around in the form of packets rather than a stream of bits (or characters). Furthermore, it will be assumed that each buffer location holds a data packet (without the header parts).

The use of the buffers were left more abstract in the protocol descriptions given in [NETR90] and [MCAR92]. However, in order to apply the system state analysis and to simulate the protocol, the structure and the use of the buffers had to be described, too.

a. OUTBUF

OUTBUF is a buffer space into which machine T4 deposits the data packets it gets from the host for transmission. Machine T1 extracts the packets from here, adds the header parts and transmits them to the receiver.

A schematic illustration of *OUTBUF* is shown in Figure 7. As it can be seen in the figure, this buffer has two parts: Retransmission buffer and transmission buffer. The retransmission buffer is located before the transmission buffer and it holds the packets that have been transmitted by machine T1 but not acknowledged yet. The transmission buffer holds the packets which have been buffered by machine T4 and waiting to be transmitted.

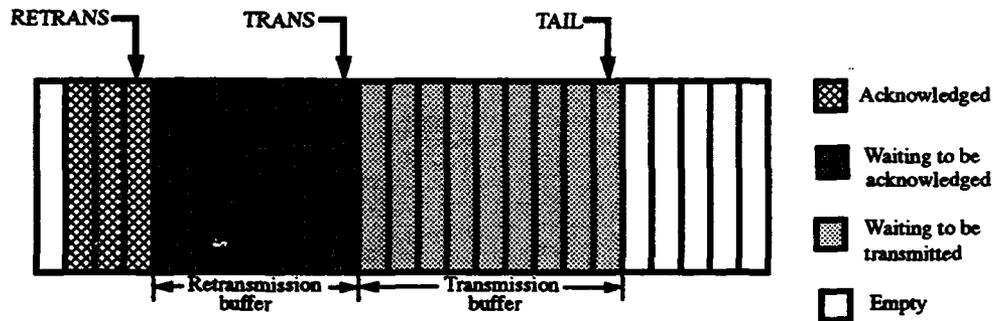


Figure 7: *OUTBUF*

These buffers are separated by three pointers called *RETRANS*, *TRANS* and *TAIL* as explained below:

The pointer *RETRANS* holds the index of the buffer location just before the first unacknowledged packet in the buffer and it indicates the beginning of the retransmission buffer. *RETRANS* is incremented when the first packet in the retransmission buffer is acknowledged.

TRANS holds the index of the buffer location just before the first packet in the transmission buffer. It shows the beginning of the transmission buffer. *TRANS* is incremented when the first packet in the transmission buffer is transmitted. Since the retransmission buffer area extends from *RETRANS* to *TRANS*, incrementing *TRANS* puts the packet in the retransmission buffer.

TAIL holds the index of the location just before the beginning of the empty buffer spaces. It is incremented when a packet is put in the buffer to be transmitted.

Initially, when there is no data in either buffer, $TAIL = TRANS = RETRANS$. The transmission buffer is empty when $TRANS = TAIL$, and the buffer is full when the packets in both buffers fill all the available buffer spaces.

The purpose of dividing the *OUTBUF* into two parts is to avoid the movement of data packets in the buffer, which is a costly operation. With the buffer scheme explained here, the data packets which are enqueued at the end of the transmission buffer remain in their places until they are acknowledged by the receiver. This scheme is also suitable for circular buffer implementations.

b. INBUF

Each logical connection has its own pre-negotiated buffer in the receiver called *INBUF*. This is the buffer place in the receiver to temporarily hold the data packets until they are retrieved by the receiving host. Another function of this buffer is to reorder the data packets that arrive out of sequence. Received data packets are inserted into buffer locations whose indexes are calculated from the sequence numbers. As it was pointed out in Chapter III, the size of *INBUF* should be at least $(RTD \times \text{negotiated peak bandwidth})$ bits in order to prevent it from overflowing. Figure 8 depicts the schematic diagram of *INBUF*:

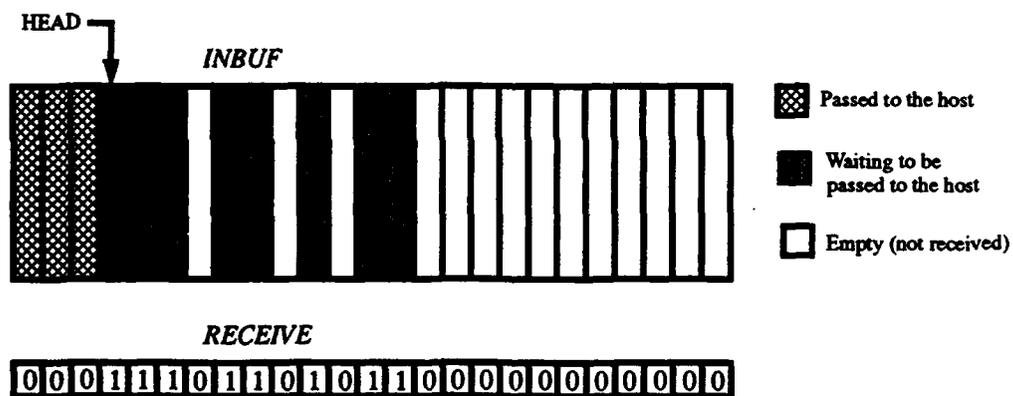


Figure 8: *INBUF* and *RECEIVE*

As it can be seen in this figure, associated with *INBUF* is an index variable *HEAD*, which is used for retrieval of packets in sequential order. This variable holds the index of the buffer location which contains the first data packet which is not retrieved yet. The *RECEIVE* array indicates if the packets are received and put in the buffer by machine R1. Then the packets are removed from the buffer at the host's convenience by machine R4.

3. *RECEIVE*

This is an array of bits where each bit maps to a location of *INBUF*, therefore, the size of *RECEIVE* is equal to the maximum number of data packets that *INBUF* can hold. The purpose of this bit array is to indicate if any location of *INBUF* contains a data packet or not as shown in Figure 8. A *RECEIVE* bit set to 1 means that there is data in the corresponding *INBUF* location, hence *RECEIVE*(*i*) is set to 1 upon insertion of a data packet into the i^{th} location of *INBUF*. This scheme has three uses: First, it helps detection of duplicate packets whose block sequence numbers are greater than LW_r , secondly, it is used in determining whether or not a whole block has been received for acknowledgment purposes and finally it indicates to machine R4 whether or not there is a data packet in the buffer ready to be retrieved.

After machine R4 passes a block of packets to the host, it sets the corresponding *RECEIVE* bits to 0 without clearing the buffer itself. With this scheme, the buffer allocation and deallocation for the packets is done in a very simple way by the protocol and no operating system support is needed, except for the allocation of a buffer space for *INBUF* in the memory.

4. *AREC*

AREC is another array of bits, whose size is equal to the number of blocks that can be hold in *INBUF*. Each bit in this array corresponds to a block size of packets in *INBUF* starting from the first location, so that bit 1 of *AREC* corresponds to the first block of packets, bit 2 corresponds to the second block of packets and so on. When all the packets

in a block have been received, the *AREC* bit for this block is set to 1. This array is used to acknowledge the blocks together with *LW_r* and *LOB* array.

These structures are used as follows in order to acknowledge the data packets received correctly: Upon reception of a data packet by the receiver, a check is done for duplicate detection. The packet is a duplicate if the block number that contains the packet is less than *LW_r*, or if the *RECEIVE* bit corresponding to the packet sequence number is 1. In this case, the packet is discarded. Otherwise, it is inserted into the corresponding *INBUF* location and the *RECEIVE* bit for this location is set to 1. If this packet completes the reception of a whole block of packets, then either *LW_r* is increased until it is equal to the sequence number of the first incomplete block (if the completed block is *LW_r*), or a bit corresponding to that block in *AREC* is set to 1 (if the completed block is different than *LW_r*). Thereafter, *AREC* is copied into *LOB* array for transmission to the transmitter in a receiver control packet to acknowledge the successfully received blocks.

B. FINITE STATE MACHINE DESCRIPTIONS

The machine organization and a general overview of the protocol is given in Chapter III. Here, the operation of the machines will be explained in detail and it will be shown how the coordination is achieved through the use of shared variables.

1. Machine T1

Machine T1 is responsible for transmission of new data packets and retransmission of unacknowledged packets whenever required. Figure 9 shows the state diagram of machine T1 and the relevant part of the Predicate Action Table is in TABLE 2.

Machine T1 starts its operation when the global variable *T_active* is set to TRUE by machine T2 upon successful connection establishment.

In mode 0, T1 transmits data packets as long as the variable *T_active* remains TRUE and there is data in the *OUTBUF* to transmit, without being concerned with flow

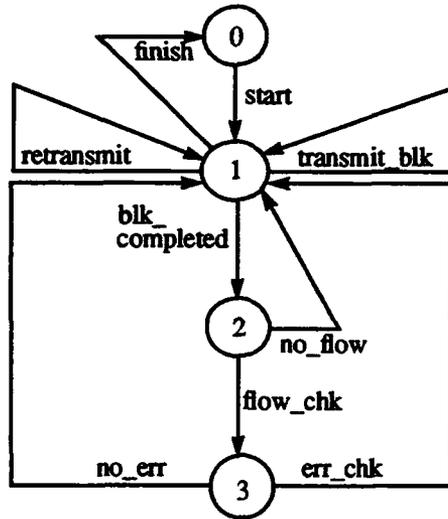


Figure 9: T1 State Diagram

TABLE 2 : PREDICATE ACTION TABLE FOR MACHINE T1

start	$T_active=T$	null
finish	$T_active=F$	null
retransmit	$T_active=T \wedge$ $mode = 2 \wedge expired(LUP) \neq 0$	Packet.seq:=(Expired(LUP)-1)*block_size+retrans_count; Packet.data:=OUTBUF(Packet.seq mod OUTBUF'length); Enqueue(Packet,T_CHAN); sent := T; inc (retrans_count); if retrans_count > block_size then retrans_count:=1; LUP((Expired(LUP)-1) mod L+1).count:=initial value; end if;
transmit_blk	$T_active=T \wedge$ not (Empty(OUTBUF)) ^ trans_count <= block_size ^ (mode=0 v ((NOU < L ^ buffer - NOU > 0) ^ (mode=1 v Expired(LUP)=0)))	retrans_count := 1; Dequeue(Packet.data,OUTBUF); Packet.seq:=UW _t * block_size + trans_count; Enqueue(Packet,T_CHAN); sent:=T; inc (trans_count);
blk_completed	trans_count > block_size	trans_count := 1; inc (UW _t);
no_flow	mode = 0	null;
flow_chk	mode=1 v mode = 2	inc (NOU);
no_err	mode = 1	null;
err_chk	mode = 2	Insert (UW _t , LUP);

control or error recovery. Every time the transmission of a block is completed, it increments the variable UW_t .

In mode 1, before transmitting a new block, the current value of NOU is compared with the maximum window size L and the last reported value of $buffer_available$ from the receiver control packet to make sure that the maximum window size will not be exceeded and that the receiver has enough buffer space to receive the block. The block is transmitted if $NOU < L$ and $buffer_available - NOU > 0$. Upon transmission of a complete block, NOU is incremented. This variable is decremented by machine T2 by the number of acknowledged blocks when new state information becomes available from the receiver, thus generating availability for the new blocks to be transmitted.

In mode 2, retransmissions are done prior to transmission of any new packets. If the *count* field of any block in the *LUP* table reaches 0, then T1 stops transmitting new packets and retransmits all the packets in the expired block. After it retransmits the whole block, it reinitializes the *count* field of the block in the *LUP* table and continues transmitting new packets, flow control permitting. Every time the transmission of a new block is completed, an entry is made into the *LUP* table, in addition to incrementing UW_t and NOU . Flow control mechanism in mode 2 is exactly the same mechanism explained in the previous paragraph for mode 1.

Upon either transmission or retransmission of a data packet, the global variable *sent* is set to TRUE. This indicates to machine T3 that data has been transmitted and that control packet transmission frequency may need to be readjusted. Each time machine T3 sends a control packet, it resets the value of *sent* to FALSE. This toggling mechanism adjusts the control packet transmission frequency based on the activity of the current logical connection.

2. Machine T2

Machine T2 has two responsibilities: (i) connection establishment and termination, (ii) reception and processing of receiver control packets. The state diagram is presented in Figure 10 and the predicate action table is given in TABLE 3.

To start the connection establishment process, T2 waits for the variable *Transmit* to be set to TRUE by machine T4. Connection establishment process follows the standard three-way handshake procedure, which is outlined previously (see Connection management on page 17).

The event *clock_tick* is used as a timing mechanism to wait for the arrival of *Conn_ack* message from the receiver and to send successive requests. After transmission of the first *Conn_req* message, the local variable *delay* is incremented every time a *clock_tick* occurs, until it becomes equal to *reset*, and then another *Conn_req* is transmitted. Therefore, $reset \times T_{in}$ gives the waiting time for the connection acknowledgment message

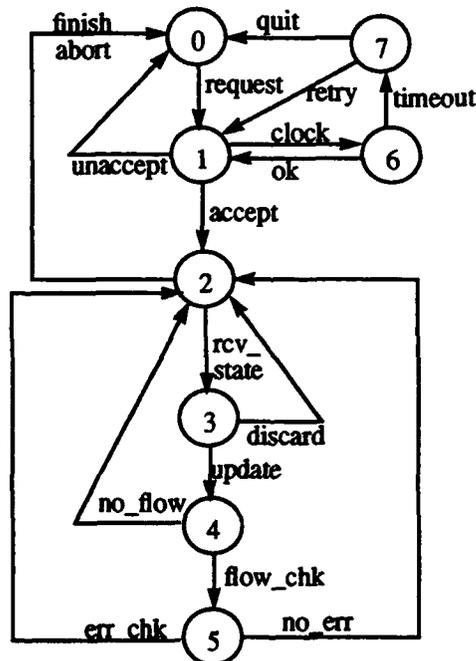


Figure 10: T2 State Diagram

TABLE 3 : PREDICATE ACTION TABLE FOR MACHINE T2

Event	Predicate	Action
request	Transmit=T ^ Accept=T ^ Fail=F	Enqueue (<i>Conn_Req</i> , T_CHAN);
accept	R_CHAN(front) = <i>Conn_Ack</i> ^ Acceptable (R_CHAN(front))	T_active := T; Enqueue (<i>Conn_Conf</i> , T_CHAN); Dequeue (R_CHAN);
unaccept	R_CHAN(front) = <i>Conn_Ack</i> ^ not (Acceptable (R_CHAN(front)))	Accept:=F; Dequeue (R_CHAN);
clock	Empty (R_CHAN) ^ <i>clock_tick</i>	inc (delay);
ok	delay < reset	null
timeout	delay = reset	inc (attempts); delay:=0;
retry	attempts < max_attempts	Enqueue (<i>Conn_Req</i> , T_CHAN)
quit	attempts = max_attempts	Fail := T
finish	Transmit = F ^ Empty (OUTBUF) ^ Disconnect = F ^ ((mode = 2 ^ Empty (LUP)) v mode = 1 v mode = 0)	T_active:=F; Enqueue (<i>Disc</i> , T_CHAN);
abort	Disconnect = T	T_active:=F; Transmit=F;
rcv_state	not (Empty (R_CHAN)) ^ Disconnect=F	null
discard	R_CHAN(front).seq <= high v R_CHAN(front)= <i>Conn_Ack</i>	Dequeue(R_CHAN);
update	R_CHAN(front).seq > high	scount:=0; high:=R_CHAN(front).seq;
no_flow	mode = 0	Dequeue(R_CHAN);
flow_chk	mode = 1 v mode = 2	Balance (R_CHAN(front).LOB, HOLD, R_CHAN(front).LW _r , LW _t , NOU); HOLD := R_CHAN(front).LOB; LW _t := R_CHAN(front).LW _r ; buffer := R_CHAN(front).buffer_available; Update_outbuf (OUTBUF, LW _t);
no_err	mode = 1	Dequeue (R_CHAN);
err_chk	mode = 2	Update_LUP (LUP, HOLD, LW _t , R_CHAN(front).k); Dequeue (R_CHAN);

to be received. Every time a connection request is sent, the variable *attempts* is incremented. If this variable becomes equal to *max_attempts*, which is the maximum number of trials to establish a connection, the transmitter aborts connection establishment process and reports to the host that the attempts have failed.

Upon successful connection establishment, machine T2 sets the variable *T_active* TRUE and all other transmitter machines start their execution in parallel. When all the data have been transmitted, machine T4 sets *Transmit* to FALSE, signalling machine T2 to

terminate the connection. Machine T2 waits until the *LUP* table is cleared (in mode 2 only), transmits a *Disc* message and sets *T_active* to FALSE, upon which all the machines make their final transitions to reach their initial states.

The main job of machine T2 during the data transfer is reception and processing of receiver control packets that are in increasing order. Out of sequence control packets as well as any duplicate *Conn_ack* packets which may still remain in the *R_CHAN* from connection establishment phase are discarded. The information in the receiver control packet is used to update the variables depending on the mode of operation as follows:

- The variable *scout* is set to 0 (for the use of this variable, see the explanation for machine T3).

- Procedure **Balance** is called to increment *NOU* by the number of acknowledged blocks in the receiver control packet (in modes 1 and 2).

- LW_r is copied into LW_t , *LOB* bit field is copied into *HOLD*, and buffer availability information is copied into *buffer_available* (in modes 1 and 2).

- Data packets in the acknowledged blocks are removed from the *OUTBUF* by the procedure **Update_OUTBUF** (in modes 1 and 2).

- The procedure **Update_LUP** is executed to remove the entries for the acknowledged blocks from the *LUP* table and decrement the *count* fields of all the unacknowledged blocks by *k*, which is obtained from the last receiver control packet. If machines T1 and T2 are trying to use *LUP* table, T2 is given priority in order to prevent unnecessary retransmissions.

3. Machine T3

Machine T3 has two main responsibilities in the protocol: periodic transmission of transmitter control packets and initialization of abnormal connection termination if no receiver control packets are received for a predetermined amount of time. The state diagram is presented in Figure 11 and the predicate action table is in TABLE 4.

Machine T3 starts its execution when T_active is set to true by machine T2 and executes its function each time a $clock_tick$ occurs. When $clock_tick$ occurs, T3 checks the value of the variable $sent$, which is set to TRUE by machine T1 after every data packet transmission. If $sent$ is TRUE, then T3 transmits a control packet and changes the value of

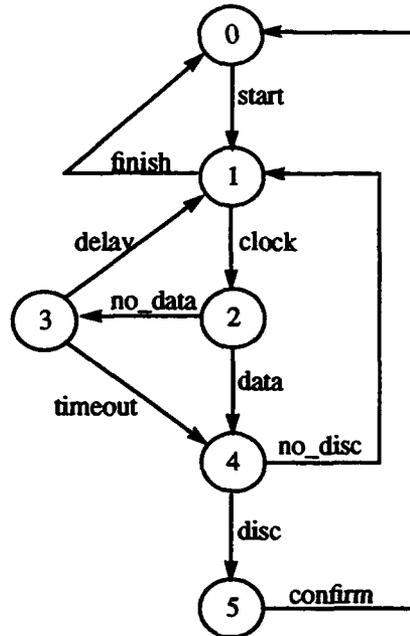


Figure 11: T3 State Diagram

TABLE 4 : PREDICATE ACTION TABLE FOR MACHINE T3

start	$T_active = T$	null
clock	$clock_tick \wedge T_active = T$	inc (scount)
no_data	$sent = F$	inc (count)
delay	$count < k \wedge scount < Lim$	null
timeout	$count = k \wedge scount = Lim$	Enqueue (T_state, T_CHAN); $k := \min(2*k, klim)$
data	$sent = T$	Enqueue (T_state, T_CHAN); $k := 1$
no_disc	$scount < Lim$	$sent := F$; $count := 0$
disc	$scount = Lim$	Disconnect := T
confirm	$T_active = F$	null
finish	$T_active = F$	null

sent to FALSE. Conversely, if the value of *sent* is FALSE, which indicates that machine T1 has not transmitted any data packets since the transmission of the last control packet, then the variable *k* is recalculated using the formula $k = \max(2 \times k, kLim)$ and control packet transmission is delayed for $k \times T_{in}$ seconds. To accomplish the delay, a local variable *count* is incremented at every *clock_tick* (which occurs every T_{in} seconds) until it becomes equal to *k*. If machine T1 transmits data during the meantime, then machine T3 stops waiting, transmits a control packet immediately and resets the variables *k* to 1 and *count* to 0. This mechanism keeps the control packet transmission rate in proportion with data packet transmission rate.

If the receiver remains silent for a for a predetermined amount of time, which is most likely to happen when a network failure occurs, machine T3 initiates the connection termination. For this purpose, another counter variable *scout* is used. T3 increments *scout* every time it transmits a control packet, and T2 sets this variable to 0 every time it receives a receiver control packet. If *scout* ever reaches the predetermined value *Lim*, then machine T3 sends a *Disc* message and sets the variable *Disconnect* to TRUE. Then, machine T2, which monitors this variable continuously, sets the variables *T_active* and *Transmit* to FALSE, causing all other machines to terminate immediately. In this case, machine T4 notifies host of the abnormal termination.

4. Machine T4

Being the host interface, Machine T4 performs the necessary communication between the transmitting host and the other machines. The state diagram is depicted in Figure 12 and the predicate action table is given in TABLE 5.

Upon receiving a transmission signal from the host, T4 sets the variable *Transmit* to TRUE indicating to machine T2 that a connection should be established. At this point, T4 starts monitoring the variables *T_active*, *Fail* and *Accept*. A TRUE state of *Fail* shows that the attempts of machine T2 to establish a connection have failed and it could not get a

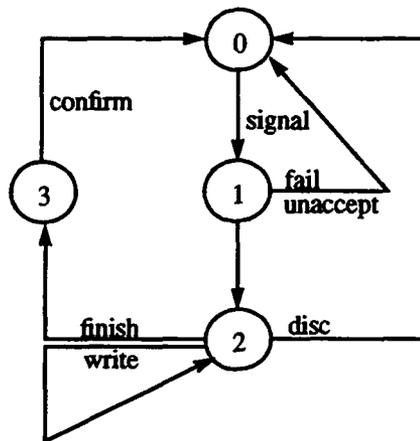


Figure 12: T4 State Diagram

TABLE 5 : PREDICATE ACTION TABLE FOR MACHINE T4

Event	Predicate	Action
signal	<i>transmission signal from the host</i>	Transmit := T
fail	Fail = T	Transmit := F; <i>notify host of failure to connect;</i>
unaccept	Accept =	<i>notify host of unacceptable connection</i>
start	T_active = T	null
write	not (Full (OUTBUF)) ^ not(eot) ^ T_active=T	Enqueue (<i>data stream from the host</i> , OUTBUF)
finish	eot ^ T_active = T	Transmit := F
confirm	T_active = F	<i>notify host of completion</i>
disc	T_active = F	<i>notify host of disconnect</i>

Conn_ack from the receiver, and a FALSE state of *Accept* shows that machine T2 has received an acknowledgment message whose parameters were unacceptable. In both cases, T4 sets the variable *Transmit* to FALSE and gives an appropriate message to the host. If, none of these happen and machine T2 establishes the connection successfully, it sets the variable *T_active* to TRUE. With this signal, machine T4 starts depositing the data to be transmitted into the buffer *OUTBUF*.

As long as the connection is active, T4 writes the data into the buffer and T1 transmits them. When the end of transmission signal is received from the host, T4 sets

Transmit to FALSE and waits for *T_active* to turn FALSE. Connection is not broken until all the data in the buffer is transmitted and acknowledged by the receiver, whereupon T2 sets *T_active* to FALSE and T4 notifies host of the completion. If the connection gets lost during the data transfer phase, machine T2 sets *T_active* to FALSE terminating all other machines. Then, machine T4 notifies host of the disconnect and it terminates also.

5. Machine R1

Machine R1 removes the received the data packets from *T_CHAN* and either inserts them into the buffer *INBUF* or passes them to the host directly, depending on the mode of operation. The state diagram for machine R1 is depicted in Figure 13 and the predicate action table is given in TABLE 6.

Machine R1 starts its operation when the global variable *R_active* is set to TRUE by machine R2 subsequent to successful connection establishment. In mode 0, R1 passes the packets to the host directly without buffering and without performing any kind of error or flow control operation.

In modes 1 and 2, each time machine R1 receives a data packet, it calls the procedure *Order_insert*, which inserts the packet into its allocated location in the buffer unless the packet is a duplicate. If the packet is duplicate, then it is discarded. Otherwise,

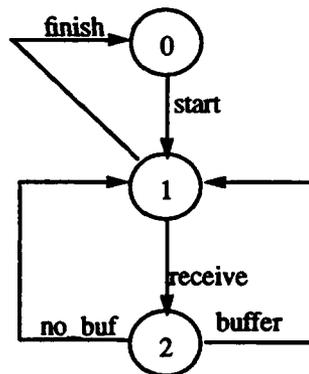


Figure 13: R1 State Diagram

TABLE 6 : PREDICATE ACTION TABLE FOR MACHINE R1

	Predicate	Action
start	R_active = T	null
finish	R_active = F ^ Empty (INBUF)	null
receive	T_CHAN (front) = DATA	null;
no_buf	mode = 0	Pass T_CHAN(front) to the host; Dequeue (T_CHAN)
buffer	mode = 1 v mode = 2	Order_insert(T_CHAN(front), INBUF, RECEIVE, LW _r duplicate); if not duplicate then received := T; Process_packet (T_CHAN(front).seq, RECEIVE, AREC,buffer_avail,LW _r UW _r LOB); end if; Dequeue (T_CHAN);

the variable *Received* is set to TRUE and the procedure **Process_packet** is called to update various variables. This procedure sets a bit in the *RECEIVE* array telling machine R4 that the corresponding buffer location holds a data packet. If the packet completes a block, then **Process_packet** decrements the variable *buffer_avail*, updates *UW_r*, *LW_r*, *AREC* and *LOB* for acknowledgment of the block.

6. Machine R2

Machine R2 is the receiver counterpart of transmitter machine T2. First, it establishes the connection with the transmitter and thereafter receives and processes the transmitter control packets. The state diagram for machine R2 is depicted in Figure 13 and the predicate action table is given in TABLE 7.

Machine R2 is activated upon reception of a *Conn_req* message. It evaluates the requested connection parameters and responds with a *Conn_ack* message containing the parameter values under which the receiver can operate. Upon reception of this message, the transmitter does its own evaluation and sends a *Conn_conf* message, if it accepts the connection with these parameters.

After R2 transmits the *Conn_ack*, three things can go wrong: (i) *Conn_ack* can get lost, (ii) *Conn_conf* can get lost, or (iii) the transmitter may not accept the connection. If

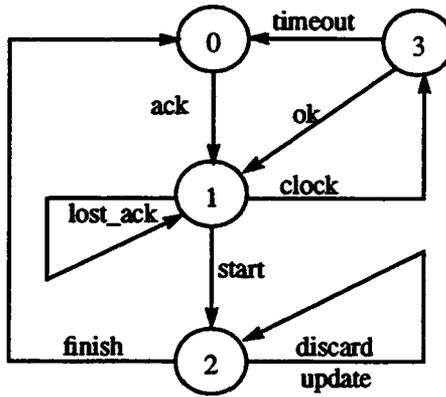


Figure 14: R2 State Diagram

TABLE 7 : PREDICATE ACTION TABLE FOR MACHINE R2

ack	$T_CHAN(front) = Conn_req$	Evaluate (<i>Conn_req</i>); Dequeue (<i>T_CHAN</i>); Enqueue (<i>Conn_ack</i> , <i>R_CHAN</i>);
clock	$clock_tick \wedge Empty(T_CHAN)$	inc (delay)
ok	$delay < reset$	Enqueue (<i>Conn_ack</i> , <i>R_CHAN</i>);
timeout	$delay = reset$	null
start	$T_CHAN(front) = Conn_conf \vee$ $T_CHAN(front) = T_state \vee$ $T_CHAN(front) = Data$	$R_active := T$; if $T_CHAN(front) = Conn_conf$ then Dequeue(<i>T_CHAN</i>); end if;
finish	$Disconnect = T \vee T_CHAN(front) = Disc$	$R_active := F$;
update	$T_CHAN(front) = T_state \wedge$ $T_CHAN(front).seq > high$	$scount := 0$; $high := T_CHAN(front).seq$; Dequeue (<i>T_CHAN</i>);
discard	$(T_CHAN(front) = Conn_conf \vee$ $T_CHAN(front) = Conn_req) \vee$ $(T_CHAN(front) = T_state \wedge$ $T_CHAN(front).seq \leq high)$	Dequeue (<i>T_CHAN</i>);
lost_ack	$T_CHAN(front) = Conn_req$	Dequeue <i>T_CHAN</i>); Enqueue (<i>Conn_ack</i> , <i>R_CHAN</i>)

Conn_ack gets lost transmitter sends another *Conn_req* after it times out, and R2 retransmits the *Conn_ack* message. Loss of *Conn_conf* is not a problem, since the transmitter immediately starts transmitting the control packets and the data packets after sending *Conn_conf*. Therefore, the connection automatically becomes implicit with the

reception of any of these messages. On the other hand, if the transmitter does not accept the parameters it aborts the connection and leaves the network. In this case, R2 waits for a while and terminates when it times out. To accomplish the timeout, the variable *delay* is incremented every time a *clock_tick* occurs until it becomes equal to *reset*. Therefore, the timeout value is $reset \times T_{in}$ seconds. Also, R2 retransmits the *Conn_ack* message at every *clock_tick*.

If none of the events above happen and the connection can be established successfully, R2 sets *R_active* to true to indicate the beginning of data transfer phase to the other machines. In data transfer phase, R2 receives the control packets from the transmitter and processes them. It only accepts the packets with monotonically increasing sequence numbers, discarding all the others. Every time R2 receives a control packet it sets the variable *scout* to 0, as an indication to machine R3 that the control packets are being received and the connection is still alive. This is exactly the same mechanism that the transmitter uses.

Machine R2 terminates if the variable *Disconnect* is set to TRUE by machine R3 (abnormal termination) or a *Disc* message is received from the transmitter. In either case, R2 sets *R_active* to FALSE to indicate the end of data transfer phase, and terminates.

7. Machine R3

The state diagram for machine R3 is shown in Figure 15 and the predicate action table is given in TABLE 8.

Machine R3 has exactly the same structure and function as the transmitter machine T3: it transmits the receiver control packets periodically to the transmitter through *R_CHAN*, and initiates an abnormal connection termination if no transmitter control packets are received for a predetermined amount of time. The only difference from the PAT of T3 is the use of the variables *R_active* and *received* instead of *T_active* and *sent* for the same purpose.

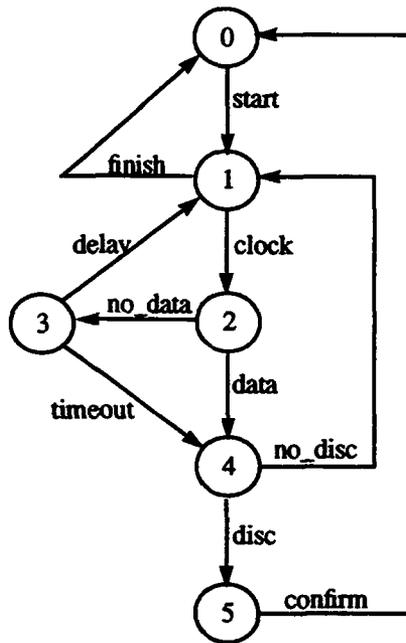


Figure 15: R3 State Diagram

TABLE 8 : PREDICATE ACTION TABLE FOR MACHINE R3

start	$R_active=T$	null
clock	$clock_tick \wedge R_active=T$	$inc(scount)$
no_data	$received=F$	$inc(count)$
delay	$count < k \wedge scount < Lim$	null
timeout	$count = k \wedge scount = Lim$	$enqueue(R_state, R_CHAN);$ $k := \min(2 * k, klim)$
data	$received=T$	$enqueue(R_state, R_CHAN);$ $k := 1$
no_disc	$scount < Lim$	$received := F; count := 0$
disc	$scount = Lim$	$Disconnect := T$
confirm	$R_active=F$	null
finish	$R_active=F$	null

8. Machine R4

Machine R4 provides the interface to the receiving host by passing the data in *INBUF* to the host and notifying the host of any errors which may occur during the reception of the data packets. The state diagram of machine R4 is depicted in Figure 16 and the PAT is given in TABLE 9.

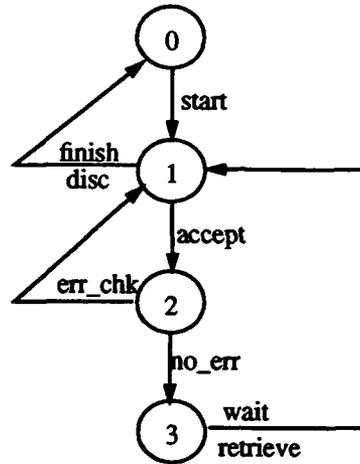


Figure 16: R4 State Diagram

TABLE 9 : PREDICATE ACTION TABLE FOR MACHINE R4

Event	Predicates	Actions
start	$R_active = T$	null;
finish	$R_active = F \wedge \text{Empty}(INBUF) \wedge$ $\text{Disconnect} = F$	null;
disc	$\text{Disconnect} = T$	<i>notify host of disconnect</i>
accept	$\text{Disconnect} = F \wedge$ $\text{not}(\text{Empty}(INBUF)) \wedge$ $\text{mode}=1 \vee \text{mode}=2 \wedge$ <i>signal from host</i>	null;
no_err	$\text{mode} = 1$	null;
wait	$\text{Wait}(INBUF, RECEIVE) = T$	null;
retrieve	$\text{Wait}(INBUF, RECEIVE) = F$	$\text{Retrieve_mode1}(INBUF, RECEIVE, AREC,$ $\text{buffer_avail}, LW_r, UW_r, LOB);$
err_chk	$\text{mode} = 2$	$\text{Retreive_mode2}(INBUF, RECEIVE, \text{buffer_avail});$

R4 starts its operation when *R_active* becomes TRUE upon successful connection establishment. The functions performed by this machine show little difference in different operation modes.

In mode 0, machine R4 only performs the notification duties and does not pass data to the host because in this mode, machine R1 does this function.

In mode 1 and 2, in addition to performing the notification duties, R4 retrieves data from *INBUF* and passes to the host in sequential order as long as there is data in the buffer. This is done by the procedures **Retrieve_mode1** and **Retrieve_mode2** depending on the operation mode. Both procedures check the *RECEIVE* array to determine if there is data in the buffer location from which the next data packet is to be retrieved.

If the buffer location is empty, then machine R4 waits until the data becomes available through the error recovery procedure. However, since error recovery is not done in mode 1, in this mode R4 waits until certain number of packets appear in the buffer after the missing one (e.g. 2), and then skips the missing packet. Procedure **Retrieve_mode1** also updates the structures, if the reception of a packet completes an entire block. Both procedures increment *buffer_avail* as the buffer space is created for new blocks.

C. SUBROUTINES

The subroutines are described in the form of algorithms using ADA's syntax to avoid getting too much into implementation details. It will be assumed that ring buffers are utilized in the protocol whose structures are as described earlier (Figure 7 and Figure 8). In the following sections, lists of the transmitter and receiver subroutines will be given. The algorithms for the subroutines are shown in Figure 17 and Figure 18.

1. Subroutines used by the transmitter

Enqueue (DATA, OUTBUF): Inserts the data packets at the end of the transmission buffer of *OUTBUF* for transmission to the receiver.

Dequeue (DATA, OUTBUF): Returns the data packet in the buffer *OUTBUF* following the *TRANS* pointer and advances the *TRANS* pointer to the next location.

Empty (OUTBUF): Returns true if $TRANS = TAIL$, indicating that there are no data packets in the transmission buffer of *OUTBUF*.

Full (OUTBUF): Returns true if there are no empty buffer locations in *OUTBUF* to write data packets

Update_OUTBUF (OUTBUF, LW_r): Advances the *RETRANS* pointer of *OUTBUF* so that it points to the buffer location just before the first packet of block number LW_r , thus leaving the acknowledged packets out of the retransmission buffer area.

Expired (LUP): This function returns the sequence number of the expired block, or if none of the blocks has expired, it returns zero.

Insert (UW_t , LUP): After the transmission of a whole block of packets has been completed (block number UW_t), this procedure makes an entry into the LUP table for the block and initializes the retransmission counter. The initial value of the retransmission counter is calculated by $RTD / T_{in} + cons$ as explained previously.

Update_LUP (LUP, LOB, LW_r , k): Every time a control packet is received from the receiver, the transmitter updates the LUP table by using this procedure. To update the table, the ACK bits of the acknowledged blocks are set to 1, thereby allowing new entries to be made into the table, and the retransmission counters of the unacknowledged blocks are decremented by k , which is read from the receiver control packet.

Empty (LUP): This function returns true if the ACK bits of all the blocks in the LUP table are 1.

Balance (LOB, HOLD, LW_r , LW_t , NOU): Every time a control packet is received from the receiver, this procedure decrements *NOU* by the number of newly acknowledged blocks. To accomplish this, the bit-map of the previous control packet, which is stored in the variable *HOLD* is compared with the *LOB* field of the currently received control packet.

Acceptable (Conn_ack): This function evaluates the connection parameters in the *Conn_ack* packet received from the receiver. If the parameters are acceptable, it returns the boolean result TRUE to the transmitter.

```
Enqueue (DATA, OUTBUF)
begin
  OUTBUF((TAIL + 1) mod OUTBUF'LENGTH) := DATA;
  TAIL := (TAIL + 1) mod OUTBUF'LENGTH;
end

Dequeue (DATA, OUTBUF)
begin
  DATA:=OUTBUF((TRANS + 1) mod OUTBUF'LENGTH);
  TRANS := (TRANS + 1) mod OUTBUF'LENGTH;
end

Empty (OUTBUF) return BOOLEAN
begin
  return TRANS = TAIL;
end

Full (OUTBUF) return BOOLEAN
begin
  return (TAIL + 1) mod OUTBUF'LENGTH = RETRANS;
end;

Update_OUTBUF (OUTBUF, LWr)
begin
  RETRANS := ((LWr - 1) * block_size) mod OUTBUF'LENGTH;
end;

Expired (LUP) return NATURAL
begin
  for I in LUP'RANGE loop
    if LUP(I).ACK = 0 and then LUP(I).COUNT = 0 then
      return LUP(I).SEQ;
    else
      return 0;
    end if;
  end loop;
end;
```

Figure 17: Transmitter Subroutines

```

Insert (UWt, LUP)
begin
  LUP ((UWt - 1) mod LUP'LENGTH + 1) := (SEQ => UWt,
                                          COUNT => RTD / Tin + cons
                                          ACK => 0);
end;

Update_LUP (LUP, LOB, LWr, k)
begin
  --Set the ACK bits of acknowledged blocks to 1
  for I in LUP'RANGE loop
    if LUP(I).SEQ < LWr then
      LUP(I).ACK := 1;
    end if;
  end loop;

  for I in LOB'RANGE loop
    if LOB(I) = 1 then
      LUP ((LWr + I - 2) mod LUP'LENGTH + 1).ACK := 1;
    end if;
  end loop;

  --Decrement the counters of unacknowledged blocks
  for I in LUP'RANGE loop
    if LUP(I).ACK = 0 then
      if LUP(I).COUNT <= k then
        LUP(I).COUNT := 0;
      else
        LUP(I).COUNT := LUP(I).COUNT - k;
      end if;
    end if;
  end loop;
end;

Empty (LUP) return BOOLEAN
begin
  for I in LUP'RANGE loop
    if LUP(I).ACK = 0 then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end;

Balance (LOB, HOLD, LWr, LWt, NOU)
begin
  for I in LWt .. LWr - 1 loop
    if HOLD(I - LWt + 1) = 0 then

```

Figure 17: Transmitter Subroutines (Cont)

```

        NOU := NOU - 1;
    end if;
end loop;

for I in LWr..LWt + L - 1 loop
    if (LOB(I - LWr + 1) = 1) and (HOLD(I - LWt + 1) = 0) then
        NOU := NOU - 1;
    end if;
end loop;

for I in (LWt + L - LWr + 1) .. L loop
    if LOB(I) = 1 then
        NOU := NOU - 1;
    end if;
end loop;
end;

Acceptable (Conn_ack) return BOOLEAN
begin
    for all negotiated parameters in Conn_ack loop
        if parameteri is not acceptable then
            return FALSE;
        end if;
    end loop;
    return TRUE;
end;

```

Figure 17: Transmitter Subroutines (Cont)

2. Subroutines used by the receiver

The following subroutines are used in the receiver. Detailed algorithms for the subroutines are shown in Figure 18.

Wait (INBUF, RECEIVE, wait_lim): This function is used only in mode 1. It returns true if the head packet of *INBUF* has not been received and there are not *wait_lim* number of packets in *INBUF* after the head (not necessarily in the subsequent locations). If *Wait* is true, then the receiver waits until the next calls of this function return false, which happens if the missing packet is received or if the number of packets in the buffer becomes equal to *wait_lim*, and then it skips the head location.

Order_insert (DATA, INBUF, RECEIVE, LW_r, duplicate): This procedure has three important tasks: (i) detects duplicate packets, (ii) if the packets are not duplicate,

inserts them into their places in the buffer, calculated from the packet sequence numbers, (iii) sets the receive bit to indicate that the corresponding *INBUF* location holds a data packet. A packet is a duplicate if the packet sequence number is less than LW_n , or if the corresponding *RECEIVE* bit is set to 1. In this case the packet is discarded.

Retrieve_mode2 (INBUF, RECEIVE, buffer_avail): This procedure is used in mode 2 to retrieve the data packets from *INBUF* and pass them to the host (a different retrieval procedure is used in mode 1, since the algorithm used in mode 1 is different). Packets are retrieved from the buffer only if they are in sequential order. The pointer *HEAD* holds the index of the *INBUF* location that contains the next data packet to be retrieved. The *RECEIVE* bit corresponding to this pointer is checked to determine if the packet has been received. If the *RECEIVE* bit is 1, the packet is retrieved from the buffer, passed to the host and the head pointer is advanced. If this packet completes retrieval of an entire block of packets, then all of the *RECEIVE* bits for the block are reset to 0 and *buffer_avail* is increased.

Retrieve_mode1 (INBUF, RECEIVE, AREC, buffer_avail, LW_n , UW_n , LOB): This is the procedure used in mode 1 to retrieve the data packets from *INBUF* and pass to the host. This procedure is called only when a data packet can be retrieved from the buffer. After checking the *RECEIVE* array, if there is a data packet in the head location of *INBUF*, then the packet is retrieved from the buffer, passed to the host and the head pointer is advanced.

If the head location of *INBUF* does not contain a data packet, then this location is skipped. The *RECEIVE* bit for the skipped packet is set to 1 and a check is done to determine whether a whole block has been received, in which case the variables are updated to provide the acknowledgment of the completed block.

Empty (INBUF, RECEIVE): This function returns true if the *INBUF* locations after *HEAD* are empty (in a ring buffer, all the buffer locations up to the beginning of the block which contains the *HEAD* should be checked).

Process_packet (Data_seq, RECEIVE, AREC, Buffer_avail, LW_r, UW_r, LOB): This procedure checks if the reception of a data packet completes an entire block in order to update some parameters. The completion of a block is determined by checking the *RECEIVE* bits. If a block is completed, then *buffer_avail* is decreased, *AREC* bit for the block is set to 1, *LW_r*, and *UW_r*, are updated and *AREC* is copied into *LOB*. Some of these variables are copied later into the receiver control packets to acknowledge the completed blocks.

```

Wait (INBUF, RECEIVE, WAIT_LIM) return BOOLEAN
begin
  NUM_PACKETS := 0;
  INDEX := HEAD mod INBUF'LENGTH + 1;
  BLOCK_START := ((HEAD - 1) / block_size) * block_size + 1;

  if RECEIVE (HEAD) = 1 then
    return FALSE;
  else
    --Check if WAIT_LIM number of packets have been received
    while INDEX /= BLOCK_START and then NUM_PACKETS < WAIT_LIM loop
      if RECEIVE(INDEX) = 0 then
        INDEX := INDEX mod INBUF'LENGTH + 1;
      else
        NUM_PACKETS := NUM_PACKETS + 1;
        INDEX := INDEX mod INBUF'LENGTH + 1;
      end if;
    end loop;
    if NUM_PACKETS = WAIT_LIM then
      return FALSE;
    else
      return TRUE;
    end if;
  end if;
end;

Order_insert (Data_packet, INBUF, RECEIVE, LWr, DUPLICATE)
begin
  --Check if the data packet is a duplicate
  if (((Data_packet.SEQ - 1) / block_size + 1) < LWr) or else
    (RECEIVE((data_packet.SEQ - 1) mod RECEIVE'LENGTH + 1) = 1) then
    DUPLICATE := TRUE; --Discard the packet
  end if;
end;

```

Figure 18: Receiver Subroutines

```

else
  DUPLICATE := FALSE;
  --Insert into the buffer
  INBUF ((data_packet.SEQ - 1) mod INBUF'LENGTH + 1) := data_packet.DATA;
  --Update RECEIVE
  RECEIVE((data_packet.SEQ - 1) mod RECEIVE'LENGTH + 1) := 1;
end if;
end;

Retrieve_mode2 (INBUF, RECEIVE, buffer_avail)
begin
  --Check if there is a packet in the buffer element
  if RECEIVE(HEAD) = 1 then
    Pass the data to the host
    --Check if a whole block has been received
    if HEAD mod block_size = 0 then
      --Reset receive bits for the block
      for I in HEAD - block_size + 1 .. HEAD loop
        RECEIVE(I) := 0;
      end loop;
      INC (buffer_avail);
    end if;
    --Increment head pointer
    HEAD :=HEAD mod INBUF'LENGTH + 1;
  end if;
end;

Retrieve_mode1 (INBUF, RECEIVE, AREC, buffer_avail, LW, UW, LOB)
begin
  BLOCK_COMPLETED := TRUE;
  INDEX := 0;
  BLOCK_START := ((HEAD - 1) / block_size) * block_size + 1;
  --Check if there is a packet in the buffer element
  if RECEIVE(HEAD) = 1 then
    Pass the data to the host
  else
    --Skip over the unreceived packet
    RECEIVE (HEAD) := 1;
    --If all of the packets in the block have been passed then ack the block
    while BLOCK_COMPLETED and then INDEX < block_size loop
      if RECEIVE(BLOCK_START + INDEX) = 0 then
        BLOCK_COMPLETED := FALSE;
      else
        INC (INDEX);
      end if;
    end loop;

    if BLOCK_COMPLETED then
      --Decrement buffer_avaialable

```

Figure 18: Receiver Subroutines (Cont)

```

buffer_avail := buffer_avail - 1;
--Update LWr and update AREC
INC (LWr);
while AREC((LWr - 1) mod AREC'LENGTH + 1) = 1 loop
    AREC((LWr - 1) mod AREC'LENGTH + 1) := 0;
    INC (LWr);
end loop;
--Update UWr
if LWr > UWr then
    UWr := LWr;
end if;
--Update LOB
for I in LOB'RANGE loop
    LOB(I) := AREC((LWr + I - 2) mod AREC'LENGTH + 1);
end loop;
end if;
end if;
--Check if a whole block has been passed
if HEAD mod block_size = 0 then
    --Reset receive bits for the block
    for I in HEAD - block_size + 1 .. HEAD loop
        RECEIVE(I) := 0;
    end loop;
    INC (buffer_avail);
end if;
--Increment head pointer
HEAD := HEAD mod INBUF'LENGTH + 1;
end;

Empty (INBUF, RECEIVE) return BOOLEAN
begin
    INDEX := HEAD mod INBUF'LENGTH + 1;
    BLOCK_START := ((HEAD - 1) / block_size) * block_size + 1;
    --Check HEAD
    if RECEIVE(HEAD) = 1 then
        return FALSE;
    else
        --Check if there is a packet in the buffer after HEAD
        while INDEX /= block_start loop
            if RECEIVE(INDEX) = 1 then
                return FALSE;
            else
                INDEX := INDEX mod INBUF'LENGTH + 1;
            end if;
        end loop;
    end if;
    return TRUE;
end;
end;

```

Figure 18: Receiver Subroutines (Cont)

```

Process_packet (Data_seq, RECEIVE, AREC, buffer_avail, LWr, UWr, LOB)
begin
  BLOCK_NUM := (Data_seq - 1) / block_size + 1;
  BLOCK_COMPLETED := TRUE;
  INDEX := 0;
  BLOCK_START := (((Data_seq - 1) mod RECEIVE'LENGTH) / block_size) * block_size + 1;
  --Check if an entire block has been received
  while BLOCK_COMPLETED and then INDEX < BLOCK_SIZE loop
    if RECEIVE(BLOCK_START + INDEX) = 0 then
      BLOCK_COMPLETED := FALSE;
    else
      INC (INDEX);
    end if;
  end loop;

  if BLOCK_COMPLETED then
    --Decrement buffer_avaialable
    BUFFER_AVAIL := BUFFER_AVAIL - 1;
    --Update LWr and/or update AREC
    if BLOCK_NUM = LWr then
      INC (LWr);
      while AREC((LWr - 1) mod AREC'LENGTH + 1) = 1 loop
        AREC((LWr - 1) mod AREC'LENGTH + 1) := 0;
        INC (LWr);
      end loop;
    else
      AREC((block_num - 1) mod AREC'LENGTH + 1) := 1;
    end if;
    --Update UWr
    if BLOCK_NUM > UWr then
      UWr := BLOCK_NUM;
    end if;
    --Update LOB
    for I in LOB'RANGE loop
      LOB(I) := AREC((LWr + I - 2) mod AREC'LENGTH + 1);
    end loop;
  end if;
end;

```

Figure 18: Receiver Subroutines (Cont)

VI. ANALYSIS

The formal SCM specification of the SNR protocol is given in the previous chapter. As with any other protocol specification, the next step is to analyze the specification to verify that the protocol is free from logical errors like deadlock, unspecified reception, unexecuted transitions and blocking loops. This chapter presents the work done on the SNR protocol during this verification phase.

Two different methods are applied to the protocol for analysis: system state analysis and software simulation with a programming language (ADA). Since the protocol was specified with the SCM model, the first intent was to apply the system state analysis. However, it was found out that applying the system state analysis alone, just as it was defined in Chapter IV of this thesis was not sufficient to make a complete analysis. The reasons for this difficulty will be explained later in this chapter. Due to the time limitations, instead of scrutinizing the insufficiencies and trying to introduce different methods to help the system state analysis, a completely separate method was applied by simulation to determine whether the specification was logically correct.

Neither of these efforts gave a complete analysis, but most aspects of the protocol were analyzed, and when taken together, a high degree of confidence in the correctness of the protocol was gained. Also, numerous deficiencies and mistakes in the specification were discovered and corrected. In the following sections, these analyses and their results will be discussed.

A. SYSTEM STATE ANALYSIS OF THE SNR PROTOCOL

For simple protocols, the *system state analysis* generates a reasonable number of system states so that the analysis can be conducted manually without difficulty. However, the specification of a practical protocol can be so complex, containing many states, transitions and variables, that it may not be feasible to apply the analysis manually. This

problem brings in the idea of automation of the analysis, that is writing a program which produces all of the reachable system states beginning from the initial system state. This has been the subject matter of other studies. In [ROTH92] implementation of such a program is presented. The program executes the analysis procedure against any two-machine protocol specified using the model. Another study presented in [BULB93] extends this program for arbitrary number of machines (up to 8).

Since manual application of the system state analysis to the eight-machine SNR protocol was not practical because of the existence of too many variables, a similar but separate program special for the SNR protocol was written using the programming language ADA to automate the analysis. The mechanical nature of the analysis method conveniently lent itself to such an automation. The output generated with this program is later manually converted into an analysis graph. In this section, the main algorithm of this program, its output for the connection management phase, and the difficulties encountered during the direct application of the system state analysis will be presented.

1. Software Tool For the System State Analysis of the SNR Protocol

The analysis program constructs the *system state analysis graph* in the computer memory using access type variables (pointers) of ADA. The main algorithm is basically the 4-step algorithm given in Chapter IV. In addition, a hashing algorithm is used to make the searches faster. Each node of the graph is represented as a record variable which contains the following: a state tuple of all the machine states, all of the variables used in the protocol, including the buffers and the transmission channels, a pointer to the outgoing transition list and a pointer to the next node in the graph. The predicate action table is programmed into subroutines and a package is formed consisting of the procedures each of which represent a machine in the PAT. These procedures are used to determine the possible transitions that can be taken from a system state and the new variable values after a transition is taken.

The analysis starts by generating an initial system state node, where all the machines are in their initial states and the variables have their initial values. This is the first

step of the algorithm. Thereafter, for each unexplored node, the procedure *ANALYZE* is invoked. This is the main analysis procedure and executes step 2 and 3 of the analysis algorithm. Figure 19 shows the algorithm of this procedure. It creates the children of the current parent, and for each child, executes a search in the graph to determine if there is an equivalent system state which has already been generated. If an equivalent system state is found, then the child is deleted and its transition pointer is changed to the found system state. Otherwise, the child is added to the graph and the analysis continues with the next unexplored system state.

```

procedure ANALYZE (CURRENT)
begin
  create all the children nodes of the current node according to the tuple and
  the variable values
  for each child loop
    find the transitions that can be taken from this child
    search the graph for a node with the same tuple as the child
    for each node found loop
      if the node has already been explored then
        compare its outgoing transitions with the outgoing transitions of the child
      else
        find the outgoing transitions of the node
        compare its outgoing transitions with the outgoing transitions of the child
      end if;
      if an equivalent node is found then
        exit the loop
      end if.
    end loop;

    if an equivalent node is found then
      change the pointer to the child to the equivalent node
      delete the child node
    else
      insert the child to the graph
    end if;
  end loop;
end

```

Figure 19: Algorithm of procedure *ANALYZE*

The program allows the analysis to be conducted up to a certain break point, which is specified by the user. After the specified portion of the analysis has been

completed, the user has the options of looking at the variable values at any system state, obtaining a printout of the analysis in the computer memory or continuing the analysis by specifying a new break point.

2. Results of the System State Analysis

a. Connection Establishment Phase Analysis

The system state analysis is first applied to the connection establishment phase. The output of the program was manually converted into a graph which is shown in Figure 20. In this graph, a system state tuple is represented with eight integers enclosed in parenthesis and a subscript. First four integers correspond to the states of the transmitter machines T1, T2, T3 and T4 respectively, and the last four integers correspond to the states of the receiver machines R1, R2, R3 and R4 respectively. If there is more than one system state with the same tuple, then a different subscript is used to indicate the difference.

The analysis begins with all machines in state 0, which is the system state (0000-0000)₀. The process is initiated when the transmitting host gives a *signal* to the host interface T4. Then, the machines do their respective jobs following the connection establishment procedure defined in Chapter III. The analysis graph shows that if the connection establishment is successful, the process will lead to system states (0201-0000) or (0201-0100), and the data transfer phase will be entered. Unsuccessful attempts will lead back to the initial state.

The connection analysis graph shows a total of 33 system states, of which 17 are unique and 16 are duplicate. This graph analyzes every possible event that can happen during the connection establishment phase and also gives a better understanding of the sequence of events that follow one another. Furthermore, it carries some other information which leads to the conclusion that the system state analysis alone is not sufficient to analyze this protocol as explained below:

For example, consider the section of the graph in the Figure 20 marked with dashed lines. There are three important points to be noticed: (i) *clockT* and *okT* transitions

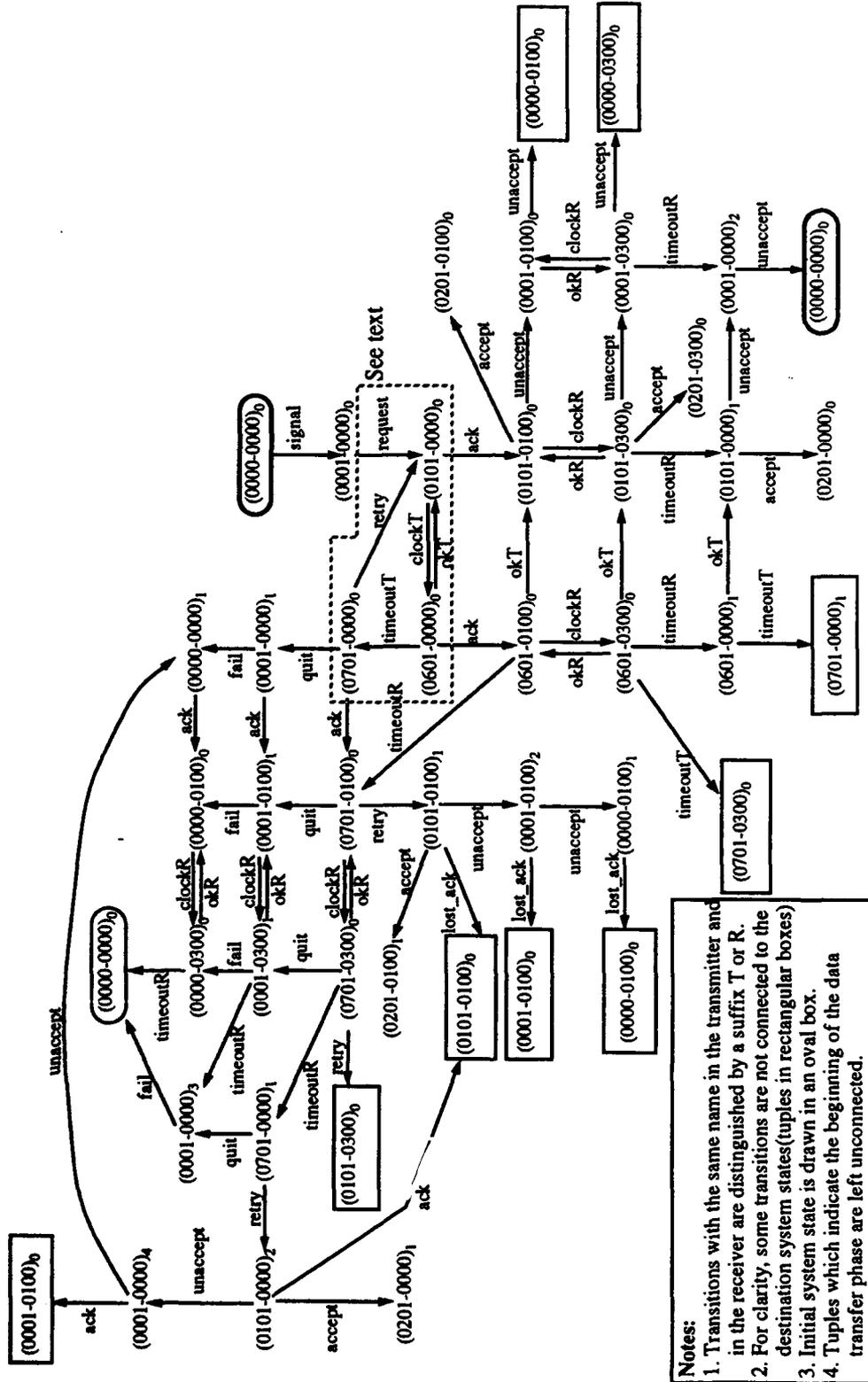


Figure 20: Connection Establishment System State Analysis

make a cycle between two system states $(0101-0000)_0$ and $(0601-0000)_0$, (ii) *clockT*, *timeoutT* and *retry* transitions make another cycle, (iii) machine T2 is allowed to make two transitions, *timeoutT* and *okT* from the same system state $(0601-0000)_0$.

As it can be seen in the graph, cycles exist in the graph at various other places, as well. The existence of cycles in an analysis normally indicate some logical errors in the protocol. However, the cycles in this case are not due to protocol errors and there are no such cycles in a global analysis. Causing cycles is a natural consequence of the system state analysis, when it is applied to this protocol straightly because of the counter variables. In this example, during the *clockT* transition between the two system states $(0101-0000)_0$ and $(0601-0000)_0$ the only variable whose value changes is the counter variable *delay* (see Machine T2 PAT on page 45). It is incremented going from state $(0101-0000)_0$ to state $(0601-0000)_0$. During the transition *okT*, none of the variables are changed and a system state, say $(0101-0000)_x$ is reached (only machine T2 goes from state 6 to state 1) for all integral value *x*. This state has exactly the same outgoing transitions as $(0101-0000)_0$ (because having the variable *delay* at a value which is one larger than it was before does not affect the outgoing transitions) so, according to the system state analysis, it is exactly *equivalent* to the state $(0101-0000)_0$. Therefore the state $(0101-0000)_x$ is deleted and the transition is connected to its *equivalent* state $(0101-0000)_0$, causing a loop to be formed.

The same reasoning can be applied to the second loop: after taking the *retry* transition from state $(0701-0000)_0$ to state $(0101-0000)_y$ for all integral value *y*, the only variable whose value changes is the counter *attempts*, but increasing the value of *attempts* does not cause a different set of outgoing transition to be taken from the state $(0101-0000)_y$ and the transition is connected to the equivalent state $(0101-0000)_0$, causing the loop.

From this example the effect of the counter variables on the system state analysis becomes clear: counter variables may assume different values as the transitions are taken, however, the system state analysis is not interested in the values of the variables but

the system state tuples and the outgoing transitions, which do not necessarily change every time the value of a counter variable changes.

The other point mentioned above, allowing machine T2 to take multiple transitions from state $(0601-0000)_0$, is related with the “counter variables problem” but it is an artificial case. It should be interpreted that T2 can either take the transition *timeoutT* or the transition *okT*, depending on the correct values of the variables (obviously, it cannot be in two different states at the same time). Without this assumption, it is impossible to continue the analysis, since the transition *timeoutT* can never be taken from the system state $(0601-0000)_0$ because of the cycling of the transitions *clockT* and *okT*. This change makes the graph in Figure 18 more intuitive than an actual system state analysis graph: we *assume* that the machines get into the loop, take *clock* and *ok* transitions several times successively, therefore causing the transport entity to “wait” for the acknowledgment or some other message from the other entity, and get out of the loop when the variable values force it. This assumption works here, because the cycle consists of only two transitions and the transition *ack* can be taken either from $(0601-0000)_0$ or $(0101-0000)_0$, but it does not necessarily hold in general.

The “counter variables problem” outlined above appears whenever some counters are incremented and, depending on their values, some decisions are made whether to take a transition or not. The problem becomes more serious in the data transfer phase causing unexecuted transitions. However, this does not necessarily mean that the system state analysis cannot be applied to this protocol, since it gives useful results at other places of the analysis. As it can be seen from the graph, the connection establishment will be completed without deadlock (if the physical link is up), or the system will return to the initial system state without deadlock. In addition to this, during the process of carrying out the analysis, a greater understanding of the protocol and its behavior in different situations was gained as well as having detected some errors. All of the data structures and the

communication between the machines have been inspected and many deficiencies have been discovered, some of which were pointed out in the previous chapters.

In order to apply the system state analysis to this protocol without generating loops, it must be supported with some other methods where it fails. An example of such a method will be presented below. This can also be a subject of further research.

b. Data Transfer Phase Analysis

The system state analysis algorithm outlined in Chapter IV is applied to the mode-0 operation of the protocol with the help of the analysis program. A small portion of the computer output is manually converted into a graph as shown in Figure 21.

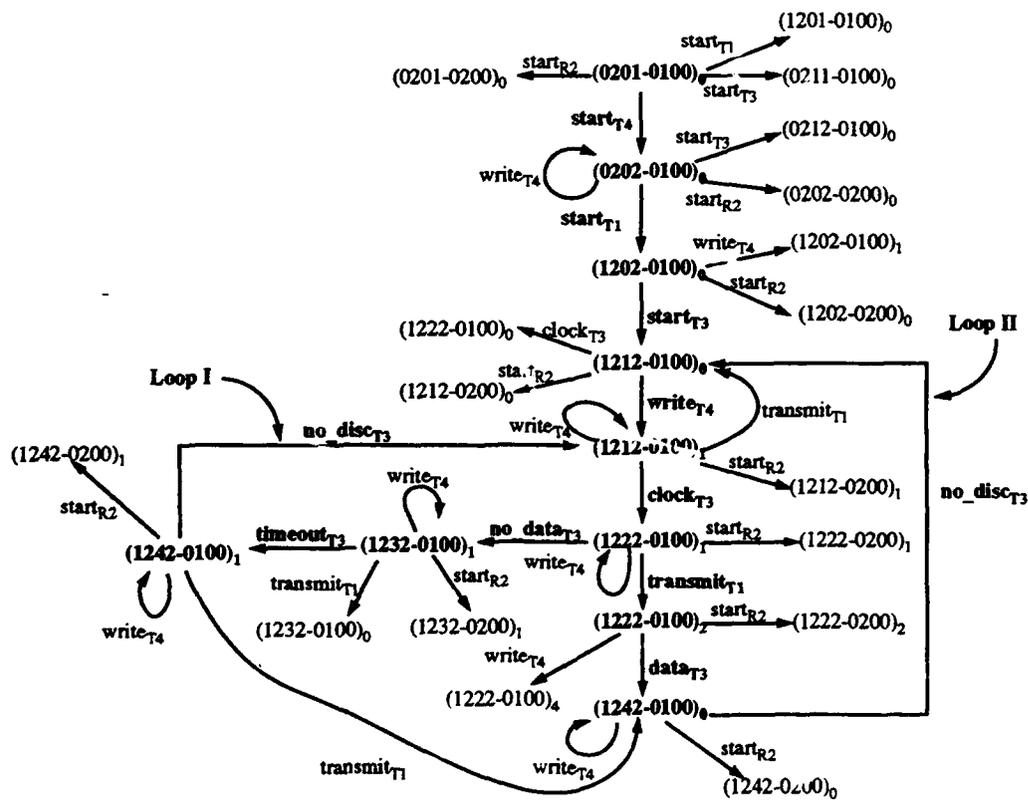


Figure 21: Part of Mode-0 Analysis

The effect of the loops can be seen clearly in this figure: once the analysis gets into a loop, it is not known if and when the analysis will get out of the loop. In fact, because of the existence of such loops, some transitions are never executed in the analysis. For example, consider the loop in Figure 21 marked "Loop I" which starts at the system state $(1212-0100)_1$: at this point, machine T3 is in state 1 and is waiting for the event *clock_tick* to occur (See "Machine T3" on page 46.) If the variable *sent* is FALSE after the *clock_tick*, then machine T3 compares the variable *count* with *k* to decide whether a control packet should be send or not. If *count* is equal to *k* (which it is at the start-up), it takes the transition *timeout* whereby it transmits a control packet. Then, depending on the value of the variable *scount*, it takes the transition *no_disc* ($scount < Lim$) or *disc* ($scount = Lim$). Since at the start-up the first condition is true, it takes the *no_disc* transition and completes the "Loop I." A similar argument can be made for the other loop, "Loop II." As a result, machine T3 can never take the transitions *delay* and *disc*. There are other transitions which are actually taken in a global analysis but cannot be shown in the system state analysis graph because of the same reason.

c. An Improved Method

To solve the looping problem caused by the counter variables, a method which "unwraps" the loops without changing the specification and the system state analysis algorithm is used. In effect, this method consists of applying a global system state analysis to the blocked parts of the analysis.

The idea of this method originates from the fact that two system states are equivalent if and only if the tuples are the same and the outgoing transitions are the same. Therefore, if a numeric index is appended at the end of the transition names, which takes on values equal to the updated counters (or any variables or a combination of them), then two system states can be equivalent if and only if (i) they have the same tuple (ii) the same set of outgoing transitions are enabled and (iii) values of the critical variables (counters in

this case) are the same. This ensures that a system state will not be considered as an *equivalent system state* unless it has the same counter variable values.

In order to illustrate how this method “unwraps” the loops, consider a previous example from the connection establishment phase: the *clock-ok* cycle. The transition names *clock*, *ok*, *timeout* and *retry* are changed as shown in TABLE 10 by appending indexes which assume the values of the two counter variables *delay* and *attempts*:

TABLE 10: INDEXED TRANSITIONS FOR MACHINE T2

Transition Name	Indexed Transition Name	Possible Transitions
clock	clock _{delay,attempts}	clock _{0,0} , clock _{1,0} , clock _{0,1} , clock _{1,1}
ok	ok _{delay,attempts}	ok _{1,0} , ok _{1,1}
timeout	timeout _{attempts}	timeout ₀ , timeout ₁
retry	retry _{attempts}	retry ₁

In this case, it is assumed that both *delay* and *attempts* can take on values from the set (0, 1, 2). Possible values of these variables are determined according to the predicate action table of machine T2 which is shown on page 45. Note that if the variable *delay* gets the value 2 when machine T2 is in state 6, then the transition *timeout* is taken, and if the variable *attempts* gets the value 2 when T2 is in state 7, then the transition *quit* is taken. Using these indexed transitions, the graph shown in Figure 22 can be obtained for the previously discussed part of the analysis. This graph looks like a global analysis graph, however, in contrast to the global analysis, not all the variables are considered and the whole analysis generates less system states than the global analysis.

As it can be seen in this example, this method does not require the specification to be changed in order to eliminate the cycles, and uses all the advantages of the system state analysis. Also, it is suitable for automation. This method was applied to the

mode-0 data transfer phase analysis of the SNR protocol and it successfully “unwrapped” the loops. On the other hand, besides these advantages, it also brings an old problem with it: the state explosion problem. By using the modified version of the analysis program, over 100,000 system states were generated for this simplest mode, and the analysis stopped only because of storage error.

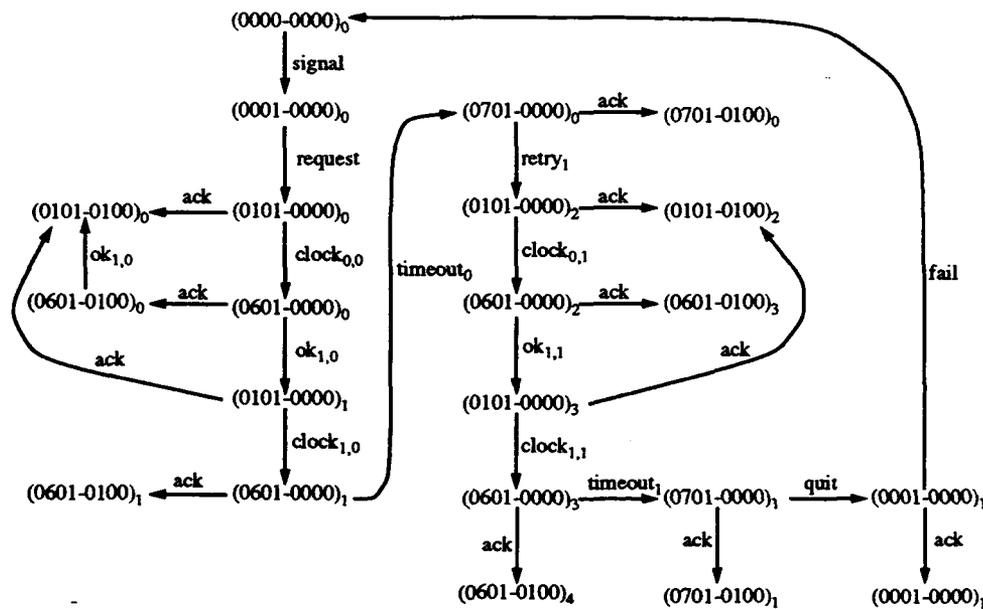


Figure 22: Sample Analysis Using Indexed Transitions

At this point, some comments are necessary: the insufficiencies pointed out in the previous paragraphs are not indications of inapplicability of the system state analysis to this protocol. The analysis is capable of producing useful results, however the nature of the insufficiencies need to be studied and at those places the method need to be enhanced by some other method, such as the one outlined above. Also, the some limitations could be introduced to the use of counter variables in the specifications. Analyzing the problem further is beyond the scope of this thesis, and it should be emphasized that this work alone can not prove the inadequacy of the analysis method.

B. SOFTWARE SIMULATION OF THE SNR PROTOCOL

Simulation is not as reliable as analysis to verify that a protocol is free from logical errors, for it cannot test every possibility. The results may depend on the implementation of the runtime environment. Nevertheless, it is a valid verification method and is used to further test the SNR protocol.

1. General Description

ADA's tasks are used to simulate each machine and the communication channels. The machine tasks are written so that their structures reflect the FSM diagram and the PAT specification of the machines as shown in Figure 23: machine states are represented by *case statements* and the predicate action table is represented by *if clauses*.

```
task body MACHINE_Ti is
  STATE: INTEGER := 0;
  Other local variable declarations
begin
  delay 10.0; --Wait for the initialization of the shared variables
  while STATE = 0 loop
    Wait for the "start transition" to be activated
  end loop;
  while STATE /= 0 loop
    case STATE is
      when 1 =>
        if predicate then
          action
          STATE :=...
        elsif predicate then
          action
          STATE :=...
        end if;
      when 2 =>
        if predicate then
          action
          STATE :=...
        elsif predicate then
          action
          STATE :=...
        end if;
    end case;
  end loop;
end MACHINE_T1;
```

Figure 23: General Task Structure

The data structures used in the simulation are chosen to be exactly the same as in the specification. All of the shared variables in each communicating entity are kept together in a "locked record" in order to prevent the tasks attempting to update the variables at the same time. The inter-process communication is achieved through the shared variables of the protocol, and ADA's rendezvous mechanism is used to coordinate the access of the tasks to these variables. Therefore, the simulation is far from a real implementation, and it somewhat retains the abstractness of the specification. The simulation is similar to applying the system state analysis without generating a complete graph.

The simulated transmitter machines read a text file whose name is given by the user into character strings representing the data packets. The data packets are "transmitted" to the receiver machines through the simulated channels. The channel tasks store the data packets in local linked lists and reorder or delete some of the packets at random, simulating the packet losses and reordering. For the simulation purposes, the data packet and the transmitter control packet formats are assumed to be fixed. Receiver control packets have the same format as in the specification. Upon reception of the data packets, the receiver tasks process the packets, store them in the buffer, acknowledge the completed blocks, and print the received information on the monitor.

Some variables are entered by the user to simulate the negotiated variables and the channel loss rates. These are data packet size (in number of characters), block size, operation mode, maximum window size, buffer size in the transmitter, retransmission counter initialization value, values of the variables *reset*, *max_attempts*, *kLim*, *Lim*, T_{in} and channel loss rate.

2. Simulation Results

As it was noted before, the purpose of the simulation was to check the correctness of the protocol specification, that is whether the protocol, as it was specified in Chapter V of this thesis, could transfer data packets from the transmitter to the receiver without being deadlocked. Therefore, rather than applying any performance tests, the simulated protocol

is tested for logical correctness in all the three modes of operation under different conditions with different values of the negotiated variables and channel loss rates.

While the simulation program was being developed and the tests were being conducted, the values of the variables were checked by using the debugger of the ADA compiler to see if the tasks simulating the FSMs could communicate as required. Also, some output statements were included in the code to trace the variables faster. In this way, several errors in the specification have been detected and corrected. By making the channels arbitrarily "loose" and "reorder" the data packets, the behavior of the protocol in such realistic conditions has been tested. It was seen that the resultant simulation program, which represented the protocol, could transfer the data packets successfully to the receiver tasks to be printed on the monitor without deadlock, unspecified reception, blocking loops or any other kind of logical errors.

This result indicates the correctness of the protocol specification with a relatively high degree of confidence. However, it does not strictly prove that the specification is free from errors, and the reason is that it cannot cover all the possible situations, some of which may end up causing a logical error. On the other hand, through this simulation process, most of the structures of the protocol are reviewed and a more detailed specification is obtained. It is the author's belief that the SNR protocol is now ready to be implemented to produce a prototype protocol. Then, it will be possible to make some performance measurements to compare it with the other transport protocols.

VII. CONCLUSION

A. SUMMARY OF THE RESEARCH

The objectives of this thesis has been to present the design, specification and analysis of the SNR protocol which is designed for providing high throughput consistent with the evolving high speed physical networks based on fiber optic transmission lines. The SNR protocol tries to overcome the difficulties encountered by the current transport protocols which are hindering utilization of the full potential offered by the fiber optic technology. It has some unique features which provide a high processing speed by simplification of the protocol, reduction of the processing overhead and utilization of parallel processing.

The SCM specification of the SNR protocol given in [MCAR92] has been improved, and some of the abstract structures of the protocol have been redefined to accomplish an analysis. Two different methods were applied to the protocol for analysis: system state analysis and software simulation with a programming language (ADA).

Since the protocol was specified with the SCM model, the first intent was to apply the system state analysis. For this purpose, a program was written which implemented the analysis specifically for this protocol. However, it was found out that a straightforward application of the system state analysis algorithm was not sufficient to make a complete analysis due to the effect of the counter variables which caused cycles in the analysis graph. To solve the looping problem caused by the counter variables, a method which "unwraps" the loops without changing the specification and the system state analysis algorithm was suggested. With this method, a kind of global system state analysis was done on the blocked parts of the analysis. One major drawback of this method was the state explosion problem.

Finally, the protocol was simulated using concurrent programming with ADA tasks in which each machine and the communication channels were represented with a task. The machine tasks were written so that their structures reflected the FSM diagram and the PAT

specification of the machines. Therefore, the simulation was different than a real implementation, and it somewhat retained the abstractness of the specification. This simulation was similar to applying the system state analysis without generating a complete graph. It was seen that the resultant simulation program, which represented the protocol, could transfer the data packets successfully to the receiver tasks without deadlock, unspecified reception, blocking loops or any other kind of logical errors.

B. CONTRIBUTIONS OF THIS THESIS

This thesis has the following contributions:

1. A complete connection establishment analysis of the protocol has been done.
2. The system state analysis has been applied to the data transfer phase and a partial analysis has been accomplished. The analysis has not been completed because these analyses have revealed a problem related with the looping effect of the counter variables in the system state analysis. A possible solution to this problem has also been suggested.
3. To do a further analysis, the protocol has been simulated and its ability to handle errors has been tested by allowing the communication channels to "lose" or "reorder" the messages at-random.

Also, during these analyses the deficiencies found in the previous specification have been corrected and the specification has been improved. These analyses have provided a higher level of confidence in the correctness of the protocol as well as a better understanding.

C. FURTHER RESEARCH OPPORTUNITIES

This thesis can form the starting point of two types of research: (i) further research on the SNR protocol and (ii) improvement of the system state analysis.

An important question concerned with the SNR protocol is whether the protocol is efficient enough to provide the high throughput which is expected from the lightweight transport protocols. To answer this question, the protocol needs to be implemented in

software and realistic performance tests need to be performed. The simulation program written for the analysis of the protocol can form the basis for this kind of research.

Another research can be concentrated on the system state analysis itself. The difficulties and insufficiencies related with the application of the system state analysis need to be studied further, and the conditions under which the system state analysis may be applied in place of global analysis need to be determined. In doing this, the problem areas can be located and the balance between the states and the variables in these areas of the protocol specification can be examined to solve the looping problem.

To overcome the state explosion problem, an attempt can be made to split up the protocol into smaller pieces and apply the system state analysis to each piece. Then it can be shown that when all those pieces are combined together, a complete analysis can be obtained. This may require using some trial and error and therefore using the analysis program.

LIST OF REFERENCES

- [BRIN85] Brinksma, E., "A tutorial on LOTOS," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.
- [BUDK87] Budkowsky, S., Dembinsky, P., "The Formal Specification Technique Estelle," *Comp. Networks ISDN Syst* 14, 1987.
- [BULB93] Bulbul, Z., B., *A Protocol Validator for the SCM and CFMS Models*, Master's Thesis, Naval Postgraduate School, Monterey California, June 1993.
- [CAST85] Castenet, R., Dupuex, A., Guitton, P., "Ada, a Well-suited Language for the Specification and Implementation of Protocols," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.
- [CLAR89] Clark, D., Jacobson, V., Romkey, J. and Salwen, H. "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, June 1989.
- [DIAZ89] Diaz, M., Ansart, J.P., Courtiat, J., Azema, P., Chari, V., *The Formal Description Technique Estelle*, North-Holland Elsevier, 1989.
- [DOD83] Department of Defence. *Military Standard Internet Protocol*, MIL-STD-1777, August 12, 1983.
- [HEAT89] Heatley, S., Stokesberry, D., "Analysis of Transport Measurements Over a Local Area Network," *IEEE Communications Magazine*, June 1989.
- [HOAR78] Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, Vol. 21, August 1978.
- [LINN85] Linn, R.J., "The Features and Facilities of Estelle: a Formal Description Technique Based upon an Extended Finite State Machine Model," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.
- [LUND88] Lundy, G.M., *Systems of Communicating Machines: A Model for Communication Protocols*, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 1988.

- [LUND91] Lundy, G.M., Miller, R.E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing*, 1991.
- [MCAR92] McArthur, R.C., *Design and Specification of a High Speed Transport Protocol*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.
- [NETR90] Netravali, A., Roome, W., and Sabnani, K., "Design and Implementation of a High Speed Transport Protocol," *IEEE Transactions in Communications*, vol.38, #11, Nov 1990.
- [ROTH92] Rothlisberger, M., J., *Automated Tools for Validating Network Protocols*, Master's Thesis, Naval Postgraduate School, Monterey California, September 1992.
- [STAL91] Stallings, W. *Data and Computer Communications*, 3rd ed., Macmillan Publishing Co., 1991.
- [VUON83] Vuong, S.T., Cowan, D.D., "Reachability Analysis of Protocols with FIFO Channels," *Communication Architectures and Protocols*, ACM SICCOMM, University of Texas at Austin, March 8-9 1983.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code 37 CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. G.M. Lundy, Code CS/Ln Assistant Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
Dr. Man-Tak Shing, Code CS/Sh Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
Dr. Raymond E. Miller Department of Computer Science A.V. Willams Bldg. University of Maryland Collage Park, MD 20742	1
Dr. Krishan Sabnani AT&T Bell Labs Room 2C-218 Murray Hill, NJ 07974	1
Deniz Kuvvetleri Komutanligi Personel Daire Baskanligi Bakanliklar, Ankara / TURKEY	1

Golcuk Tersanesi Komutanligi 1
Golcuk, Kocaeli / TURKEY

Deniz Harp Okulu Komutanligi 1
81704 Tuzla, Istanbul / TURKEY

Taskizak Tersanesi Komutanligi 1
Kasimpasa, Istanbul / TURKEY

LTJG. H. Alphan TIPICI 1
Istiklal Mah. Burc Cad. Kartal Apt. B2 / D12
81240 Umraniye, Istanbul / TURKEY