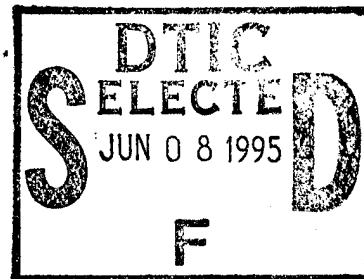# NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

AN IMPLEMENTATION OF THE SNR HIGH SPEED NETWORK COMMUNICATION PROTOCOL (RECEIVER PART)

by

Wen-Jyh Wan

March 1995

Thesis Co-Advisor: G. M. Lundy

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE		Form Approved *OMB No. 0704*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE AN IMPLEMENTATION OF THE SNR HIGH SPEED NETWORK COMMUNICATION PROTOCOL (RECEIVER PART)		5. FUNDING NUMBERS	
6. AUTHOR(S) Wen-Jyh Wan			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	

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

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; Distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis work is to implement the receiver part of the SNR high speed network transport protocol. The approach was to use the Systems of Communicating Machines (SCM) as the formal definition of the protocol. Programs were developed on top of the Unix system using C programming language. The Unix system features that were adopted for this implementation were multitasking, signals, shared memory, semaphores, sockets, timers and process control. The problems encountered, and solved, were signal loss, shared memory conflicts, process synchronization, scheduling, data alignment and errors in the SCM specification itself. The result was a correctly functioning program which implemented the SNR protocol. The system was tested using different connection modes, lost packets, duplicate packets and large data transfers.

The contributions of this thesis are: (1) implementation of the receiver part of the SNR high speed transport protocol; (2) testing and integration with the transmitter part of the SNR transport protocol on an FDDI data link layered network; (3) demonstration of the functions of the SNR transport protocol such as connection management, sequenced delivery, flow control and error recovery using selective repeat methods of retransmission and (4) modifications to the SNR transport protocol specification such as corrections for incorrect predicate conditions, defining of additional packet types formats, solutions for signal lost and processes contention problems etc.

14. SUBJECT TERMS SNR, Transport protocol, Receiver, Implementation, Testing, Tuning, Selective repeat, Implicit timer, Parallel processing, High Speed Network, Systems of Communicating Machines, SCM, Change of Specification,			15. NUMBER OF PAGES 80
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

AN IMPLEMENTATION OF THE SNR HIGH SPEED NETWORK
COMMUNICATION PROTOCOL (RECEIVER PART)

Wen-Jyh Wan
Commander, Taiwan Navy
B.S, Chinese Naval Academy, November 1980

Submitted in partial fulfillment
of the requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March, 1995

Author: _____
Wen-Jyh Wan

Approved By: _____
G. M. Lundy, Co-Advisor

_____
Shridhar B. Shukla, Co-Advisor

_____
Ted Lewis, Chairman
Department of Computer Science

_____
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

iii

iv

ABSTRACT

This thesis work is to implement the receiver part of the SNR high speed network transport protocol. The approach was to use the Systems of Communicating Machines (SCM) as the formal definition of the protocol. Programs were developed on top of the Unix system using C programming language. The Unix system features that were adopted for this implementation were multitasking, signals, shared memory, semaphores, sockets, timers and process control. The problems encountered, and solved, were signal loss, shared memory conflicts, process synchronization, scheduling, data alignment and errors in the SCM specification itself. The result was a correctly functioning program which implemented the SNR protocol. The system was tested using different connection modes, lost packets, duplicate packets and large data transfers.

The contributions of this thesis are: (1) implementation of the receiver part of the SNR high speed transport protocol; (2) testing and integration with the transmitter part of the SNR transport protocol on an FDDI data link layered network; (3) demonstration of the functions of the SNR transport protocol such as connection management, sequenced delivery, flow control and error recovery using selective repeat methods of retransmission and (4) modifications to the SNR transport protocol specification such as corrections for incorrect predicate conditions, defining of additional packet types formats, solutions for signal lost and processes contention problems etc.

Accesion For		
NTIS CRA&I		☒
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

TABLE OF CONTENTS

I. INTRODUCTION

The SNR protocol is a network transport protocol which was designed to efficiently use the high transmission rate and low error rate provided by optical fiber links.

As described in [Ref. 1], 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,

- selective repeat method of retransmission,

- the concept of packet blocking, and

- parallel processing.

The SNR transport protocol is intended to connect two host computers end-to-end across a high-speed network as shown in Figure. 1.1.
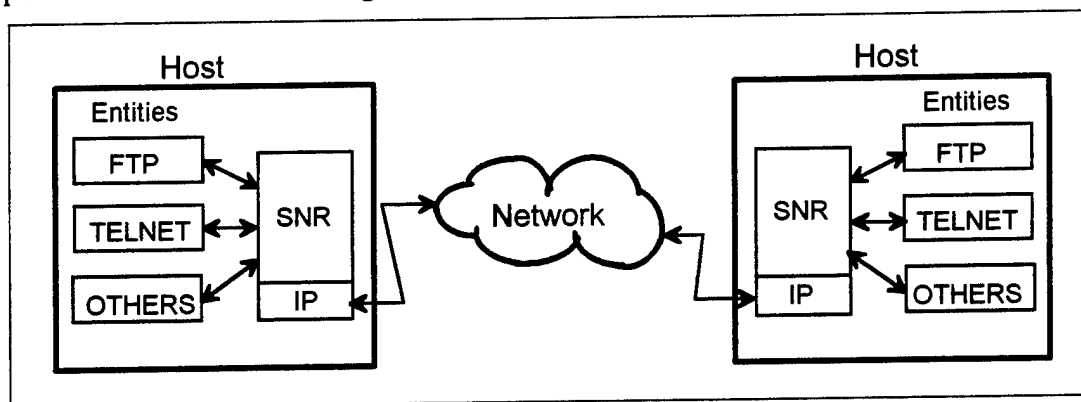
Figure 1.1 - Network, Hosts, Entities and SNR Protocol Process

The 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.

The general organization of the machines is shown in Figure 1.2. Each machine in the protocol performs a specific function in coordination with other machines. The coordination is established by communicating through shared variables.

1

Figure 1.2 - Machine Organization. [From Ref. 1]

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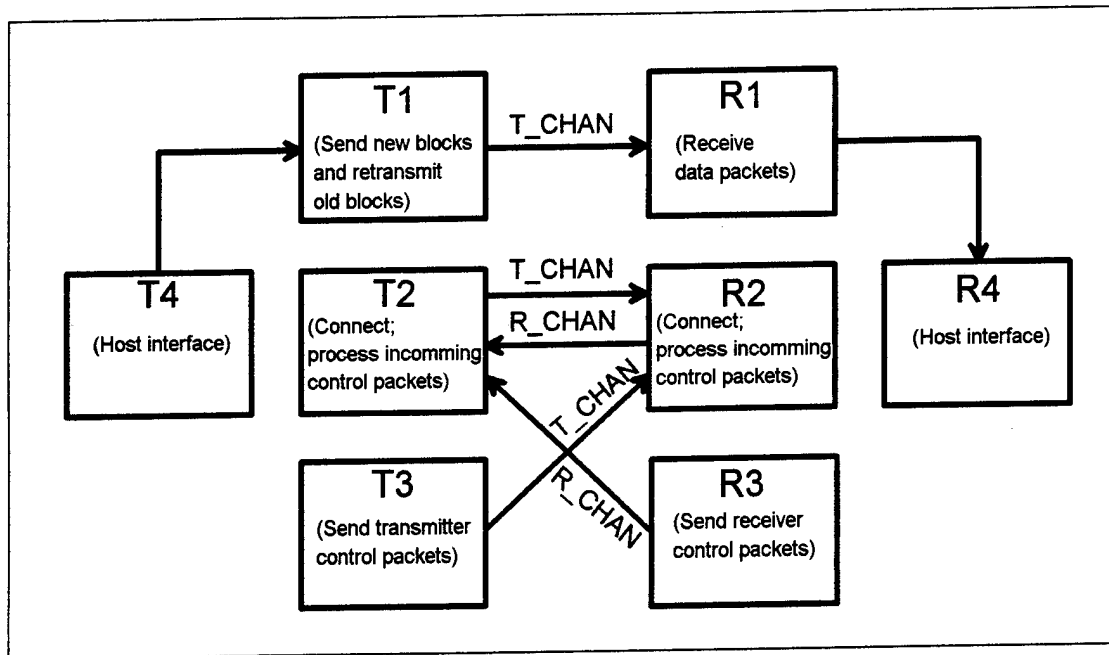s related tables and variables as the blocks are acknowledged. Machine T3 sends transmitter control packets to the receiver periodically. Machine T4 is the host interface of the transmitter. It inserts the incoming data stream into the buffer for transmission by machine T1.

Machine R1 removes the data packets from the transmitter channel and inserts them into the buffer in order according to their sequence numbers. Machines R2 and R3 are receiver counterparts of transmitter machines T2 and T3. Machine R2 receives 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 and passes them to the host.

The services provided by the protocol are as follows:

• multiplexing, demultiplexing,

• connection management,

• sequenced delivery,

2

• flow control, and

• error recovery.

Three modes of operation are specified in this protocol for a more flexible application:

Mode	Error control	Flow control	Application
0	No	No	Virtual circuit NW, quick interaction, short packets etc.
1	No	Yes	real time application, packetized voice, real time monitoring of remote sensor etc.
2	Yes	Yes	Most reliable, used for large file transfers

Table 1.1: Operation Modes in SNR Protocol [After Ref. 1]

Many studies has been made since the design of this protocol. That include a formal specification using Systems of Communicating Machines (SCM), global state analysis, comparison to other existing transport protocols, references and so on. In this paper, an implementation of the receiver part of this protocol based on an SCM specification provided in [Ref. 1] is described. Topics are given by focusing on the implementation related issues:

• the change of specification for the implementation (with some error corrections),

• how the receiver was implemented,

• the problems encountered during the implementation,

• how this implementation was tested, and

• how this implementation can be maintained or be extended in the future.

The final goal (though not accomplished yet) of this implementation is to have the implemented SNR transport protocol to replace the TCP protocol in the TCP/IP network layer.

3

II. SPECIFICATION AND MODIFICATION

A.   GENERAL

The specification of the SNR protocol that this implementation is based on is from [Ref. 1]. It will be referred to as the specification in the following text. The specification gives a formal specification for the SNR protocol by using the formal model Systems of Communicating Machines (SCM). This model was very clear and makes the implementation pretty straight forward. Due to the fact that this specification has been checked by some simulation model, few errors were found during the implementation. There were several changes made to the specification for each machine, some of those were for error corrections and some of those were for getting around problems encountered during the implementation. In this paper, the specifications related to the Receiver part will be covered.

1. Change of Specification

The following changes that are common to the Receiver and the Transmitter were made to the specification.

a. Additional packet type formats. There were three packet types that have formats defined in the specification [Ref. 1]. They were Receiver control packet format (R_state), Transmitter control packet format (T_state) and Data packet format ($Data$). However in the specification, four other packet types were mentioned - $Conn\_req$, $Conn\_ack$, $Conn\_conf$ and $Conn\_disc$. These types of packet format were not defined in the specification. By studying of the function of each packet type and discussion with the author of [Ref. 2], we decided to define a common packet format for the four packet types. It is called connection packet format ($SNRconn\_t$ in Receiver). Different numbers were used for each packet type for identification. The connection packet format is given in Figure 2.1.

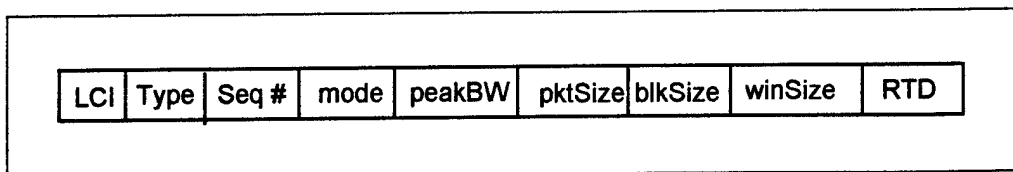| LCI	Type	Seq #	mode	peakBW	pktSize	blkSize	winSize	RTD

Figure 2.1 - Connection Packet Format

The numbers assigned to each connection packet types were 3 for *Conn_req*, 4 for *Conn_ack*, 5 for *Conn_conf* and 6 for *Conn_disc*. Each fields in the connection packet type (except *Conn_disc*) were used for negotiate parameters during the connection establishment.

b. Data length field added to the packet header. In the specification, the Receiver has insufficient information to decide the data length for each data packets. Although the Transmitter would segment the data into equal sized packets, it is always possible that the last packet may contain data that does not use all the space provided by a data packet. Ending data by padding zeros would not work since some data to be transferred could contain bytes of zeros. Using special bit patterns to indicate the end of data would not work for the same reason. Thus an extra field for the data length was added to the packet header. Figure 2.2 depicted this change.
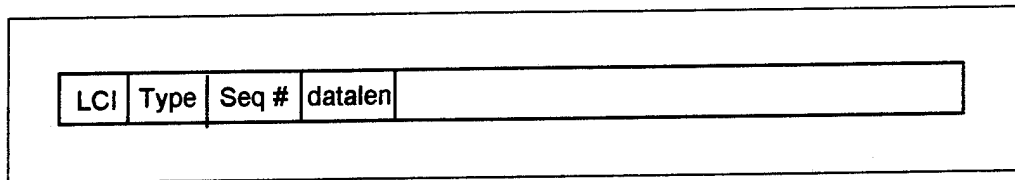
LCI	Type	Seq #	datalen	

Figure 2.2 - General SNR Packet Header With Data Length Field Added

B.   MACHINE R1

Machine R1 removes the data packets from *T_CHAN* and inserts them into their allocated locations in the buffer *INBUF*, discards duplicate packets, and updates the structures used for flow control and error recovery management (*RECEIVE, AREC* and *LOB*). In mode 0, R1 passes the packets to the host directly without buffering and without performing any kind of error or flow control operation [Ref. 1]. The modified state diagram for machine R1 is depicted in Figure 2.3. The modified Predicate-Action Table is given in Table 2.1.

1. Change of Specification

Changes for machine R1 were as following:

a. In the original specification, there was no paragraph about the initialization in each machine. Since the Receiver is going to be running in the background as a daemon process, there should be an initialization for some data structures that would be used repeatedly in each connection. In R1, the initialization was made in *start* transition (i.e. transition from state 0 to state 1). Reference Appendix for more details about the initialization.
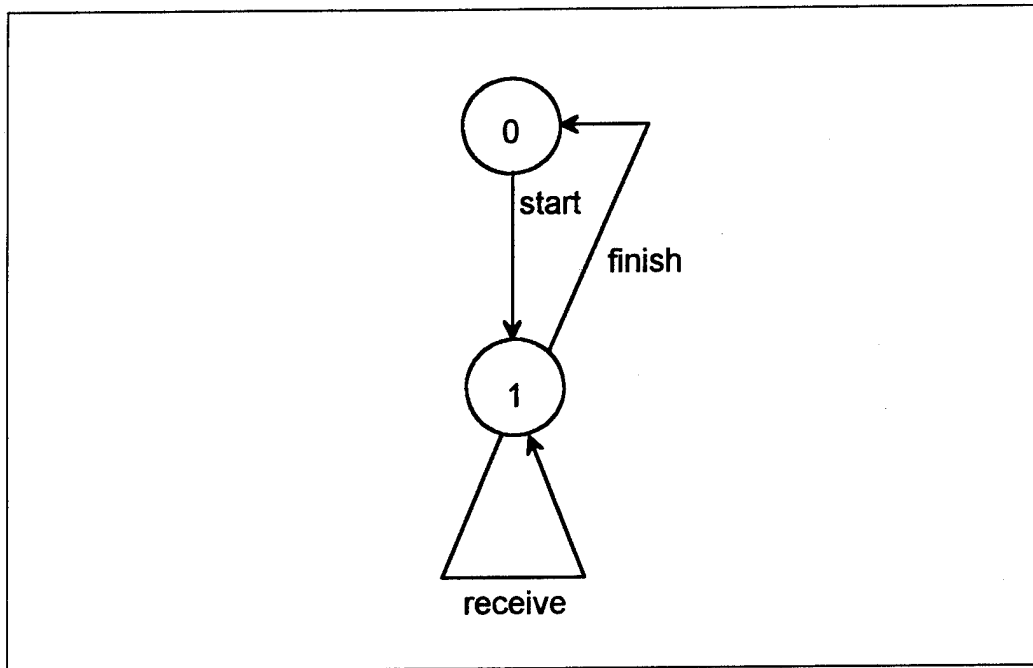
Figure 2.3 - R1 State Diagram

Transition	Predicate	Action
start	R_active = T	Initialization();
finish	R_active = F	if (Disconnect = F) then ProcTchanPkts();
receive	T_CHAN[front].type = Data	ProcTchanPkts();

Table 2.1- R1 Predicate-Action Table

b. In the R1 Predicate-Action Table, when $R\_active = F \wedge Empty(INBUF)$ in state 1 will cause the *finish* transition. This should be corrected as $R\_active = F \wedge Empty(T\_CHAN)$ since R1 is responsible to retrieve data packets from T_CHAN and inserts them into $INBUF$. Additionally, the predicate condition $Empty(T\_CHAN)$ would be checked inside the function *ProcTchanPkts* in this implementation thus this predicate condition is not shown in the modified Predicate-Action Table.

c. The signal provided by the UNIX system was not very reliable. Signals could be lost during processing (to be described in Chapter III section A.3). To overcome this problem, whenever R1 receives a signal that informs R1 to retrieve data packets from T_CHAN, R1 will try to retrieve all the data packets in T_CHAN. This modification was made inside function *ProcTchanPkts*.

7

d. There would be an unexpected timeout disconnection in mode 0 when transferring a large file. This was because in mode 0, R1 does not set *received* to *True* whenever a data packet is received. This would cause R3 to expand its time interval to send *R_state* control packet. Thus on the Transmitter side, no matter how long can it wait to initiate a timeout disconnection, there would be always possible to get a file large enough to cause the timeout. This may not be a problem if mode 0 was not designed to handle large files by the original SNR protocol designers of [Ref. 3], but however, in this implementation, a change was made to have R1 sets *received* to *True* in mode 0 when data packet received to make the protocol more flexible. This change is made inside the function *ProcTchanPkts*.

e. A race condition problem between R1 and R4 caused by parallel processing (see Chapter III section A.5.a for more details.) was solved by having SNR_TC (the process that handles the packets received from the Transmitter and put them into *T_CHAN*) notifies R1 about the arrival of *Conn_disc* packet and let R1 notify R2 when all *Data* packets in *T_CHAN* has been processed. The modification for R1 was made inside function *ProcTchanPkts*.

f. In machine R1 State Diagram and Predicate-Action Table, state 2 is an internal state that R1 will temporarily stay in. When in state 2, R1 could always transit to state 1 by checking the mode for this connection to perform *buffer* or *no_buf* transition. In this implementation, no state transition would be made for an internal state. The program branches to perform appropriate processing depending on the related predicate conditions for the internal state. In this paper, an internal state is drawn as a dash-lined circle in a State Transition Diagram. In addition, the checking of modes for different processing (*no_buf* and *buffer*) was made inside the function *ProcTchanPkts*. This further removes the internal state 2 from the modified State Transition Diagram.

g. There would be no point for R1 to process *Data* packets in the *T_CHAN* when *Disconnect* = *T*. Under this concern, a checking of *Disconnect* status is made in the action of the *finish* transition to decide whether to perform *ProcTchanPkts* or not.

C.   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.

In the 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 *scount* 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 [Ref. 1]. The modified State Diagram is depicted in Figure 2.4. The modified Predicate-Action Table is in Table 2.2.
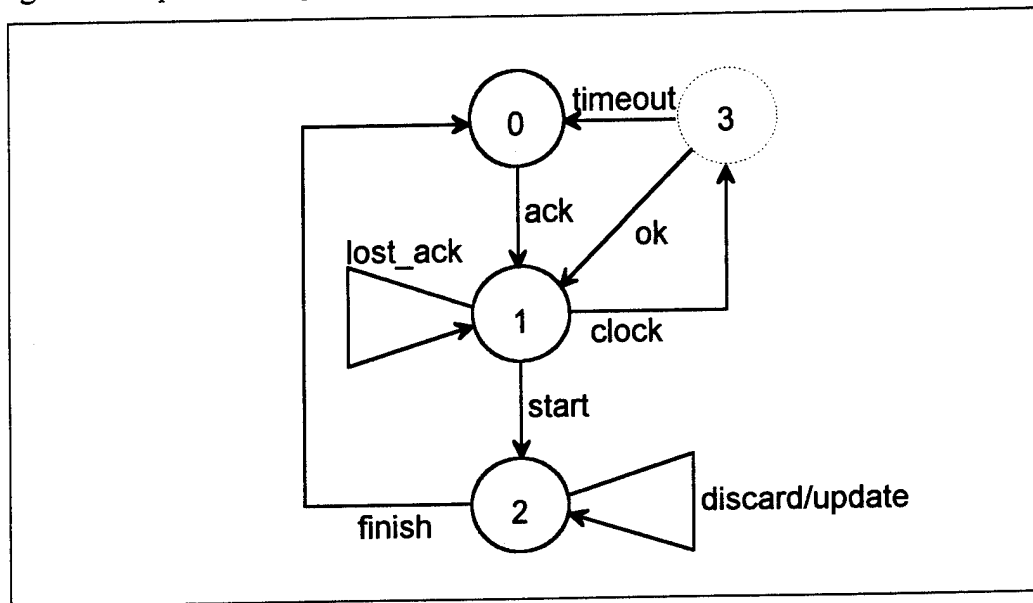
Figure 2.4 - R2 State Diagram

1. Change of Specification

The following changes were made on the machine R2.

a. State 3 is an internal state, R2 could transit to other state by checking the predicate condition *delay < reset* or *delay = reset*.

b. In R2, the initialization would be made in *ack* transition (i.e. transition from state 0 to state 1).

c. *Enqueue(R_CHAN, Conn_ack)* was implemented by directly sending the packet over the *R_CHAN* rather than queuing it to the *R_CHAN* before sending.

9

Transition	Predicate	Action
ack	T_CHAN[front].type = Conn_Req	Conn_Req = Dequeue(T_CHAN); Evaluate(Conn_Req); SendPkt(Conn_Ack) over R_CHAN; ReqestTimerSrv(IPT); Initialization();
clock	clock_tick ^ Empty(T_CHAN)	inc(delay);
ok	delay < reset	SendPkt(Conn_Ack) over R_CHAN
timeout	delay = reset	CancelTimerSrv();
start	T_CHAN[front].type = Conn_Conf v T_stateFlag = T v T_CHAN[front].type = Data	if (packet type = Conn_Conf) then Conn_Conf = Dequeue(T_CHAN); Retrieve negotiated parameters; CancelTimerSrv(); R_active = T;
lost_ack	T_CHAN[front].type = Conn_Req	Dequeue(T_CHAN); SendPkt(Conn_Ack) over R_CHAN;
finish	Disconnect = T v Conn_discFlag = T	R_active = F
update	T_stateFlag = T ^ T_state.seq > high	high = T_state.seq
discard	T_CHAN[front].type = Conn_Conf v T_CHAN[front].type = Conn_Req	Dequeue(T_CHAN);

Table 2.2 - R2 Predicate-Action Table

d. A periodic timer signal for state 1 was requested in the *ack* transition.

e. In *start* transition, the retrieval of negotiated parameters is performed if the packet type is *Conn_conf*.

f. T_state packets were implemented as out-of-band packets (i.e. they will not be queued in *T_CHAN*). The checking of predicate condition *T_CHAN[front]* = *T_state* was changed to *T_stateFlag* = *T*. Similar discussions applied for the change of predicate condition *T_CHAN[front]* = *Conn_disc* to *Conn_discFlag* = *T*. See Chapter III section A.5.c for more details about out-of-band packets.

g. In *start* transition, the periodic timer signal service would be canceled before transit to state 2.

D. MACHINE R3

R3 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 Predicate-Action Table of T3 is the use of the variables *R_active* and *received* instead of *T*_active and *sent* for the same purpose [Ref. 1]. The modified State Diagram is depicted in Figure 2.5. The modified Predicate-Action Table is in Table 2.3.

Figure 2.5 - R3 State Diagram

1. Change of Specification

The following changes were made on the machine R3.

a. In *start* transition, the initialization and request for timer service were made.

b. States 2, 3 and 4 were internal states.

c. The *timeout* transition predicate *count* = $k \wedge scount$ = *Lim* should be corrected as *count* = $k \vee scount$ = *Lim* since the *timeout* should occur when either there is no data packet been received for *count* implied time period or there is no *T_state* control packet been received for *scount* implied time period .

11

Transition	Predicate	Action
start	R_active = T	Initialization(); ReqTimerSrv(Tin);
clock	clock_tick ^ R_active = T	inc(scount);
no_data	received = F	inc(count);
delay	count < k ^ scount < Lim	null
timeout	count = k v scount = Lim	SendPkt(R_state) over R_CHAN; k = min(2*k, klim);
data	received = T	SendPkt(R_state) over R_CHAN; k = 1;
no_disc	scount < Lim	received = F; count = 0;
disc	scount = Lim	Disconnect = T; CancelTimerSrv();
confirm	R_active = F;	null
finish	R_active = F	CancelTimerSrv();

Table 2.3 - R3 Predicate-Action Table

d. The timer service would be canceled in both *disc* and *finish* transition.

E.   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 occur during the reception of the data packets [Ref. 1]. The modified State Diagram is depicted in Figure 2.6. The modified Predicate-Action Table is in Table 2.4.

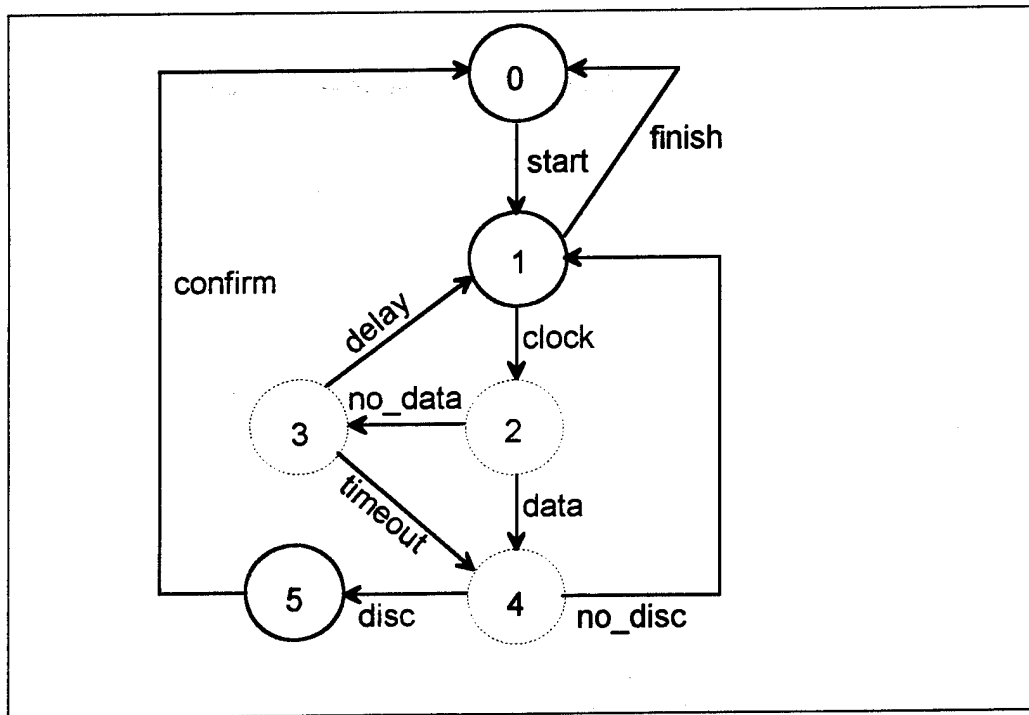### 1. Change of Specification

The following changes were made to the specification of machine R4.

a. The *start* transition predicate was changed to $R\_active = T$ ^ *(mode = 1 v mode = 2)* since R4 only interested in these two modes.

b. An initialization action was made in the *start* transition.

c. R4 will inform host about the connection during *start* transition.

d. R4 will inform host about the completion during *finish* transition.

e. The checking for *mode = 1 v mode = 2* predicate condition has been removed from *accept* transition predicate since the check has been made in *start* transition as mentioned in a.

f. States 2 and 3 were internal states.

12
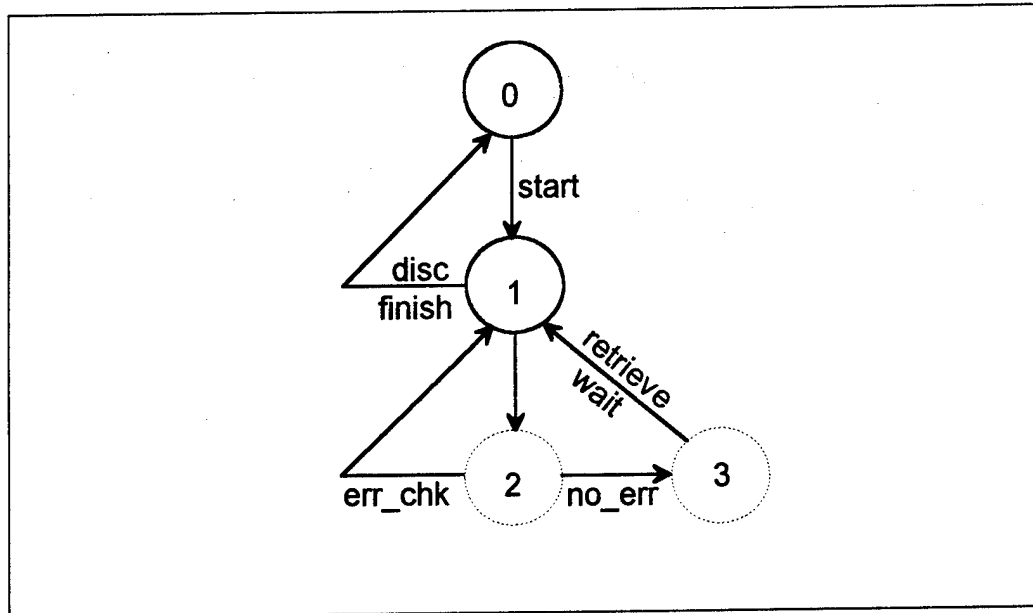
Figure 2.6 - R4 State Diagram

Transition	Predicate	Action
start	R_active = T ^ (mode = 1 v mode = 2)	Initialization(); NotifyHostOfConnection(mode);
finish	R_active = F ^ Empty(INBUF) ^ Disconnect = F	NotifyHostOfCompletion();
disc	Disconnect = T	NotifyHostOfDisconnect();
accept	Disconnect = F ^ not (Empty(INBUF)) ^ signal from host	null
no_err	mode = 1	null
wait	WaitBulk(INBUF, RECEIVE) = T	null
retrieve	WaitBulk(INBUF, RECEIVE) = F	Retrieve_mode1();
err_chk	mode = 2	Retrieve_mode2();

Table 2.4 - R4 Predicate-Action Table

13

III. IMPLEMENTATION OF THE SNR RECEIVER

A.   GENERAL

The implementation of the SNR protocol was divided into two parts to be able to fit into a thesis work. By the nature of the SNR protocol, it was divided into the transmitter part and the receiver part (referred to as Transmitter and Receiver in the following text). The Transmitter was implemented by Farah Mezhoud under the advisory of Professor Lundy. The paper of the implementation of the Transmitter would be referred to as [Ref. 2] in this paper.

The Receiver was implemented to work as a daemon running in the background on the UNIX system. It was implemented using C programming language. Each machine mentioned in the specification was implemented as a process. A shared memory space was allocated for the purposes of interchanging and sharing information among processes. To avoid racing of the same data structure in the shared memory, semaphores were used for access control. Each process would pause (put into a wait state by the operating system) for an event to occur. This avoided busy-waiting for some variables to change. When an event occurred in a process, it would notify the processes waiting for this event by using signal system calls. The general schematic diagram for the Receiver is shown in Figure 3.1.

In the schematic diagram, the circle stands for a process, the sink shaped line stands for a variable pool, in this case, the shared memory.   The arrow headed lines stand for the accessibility of the processes to the pool. The arrow pointed to the pool stands for the write accessibility, and the arrow pointed from the pool means read accessibility. In this schematic, SNR_R has the write accessibility to the shared memory and the SNR_R1 has both read and write accessibility to the shared memory.

1. Processes

There are six processes in the SNR receiver. The Receiver root (SNR_R) forks all the working processes in the Receiver. Each process once forked will perform an initialization, then goes to state 0. The SNR_TC receives a packet from the socket, enqueue the packet to the *T_CHAN*, then notifies the relevant process according to the packet type.
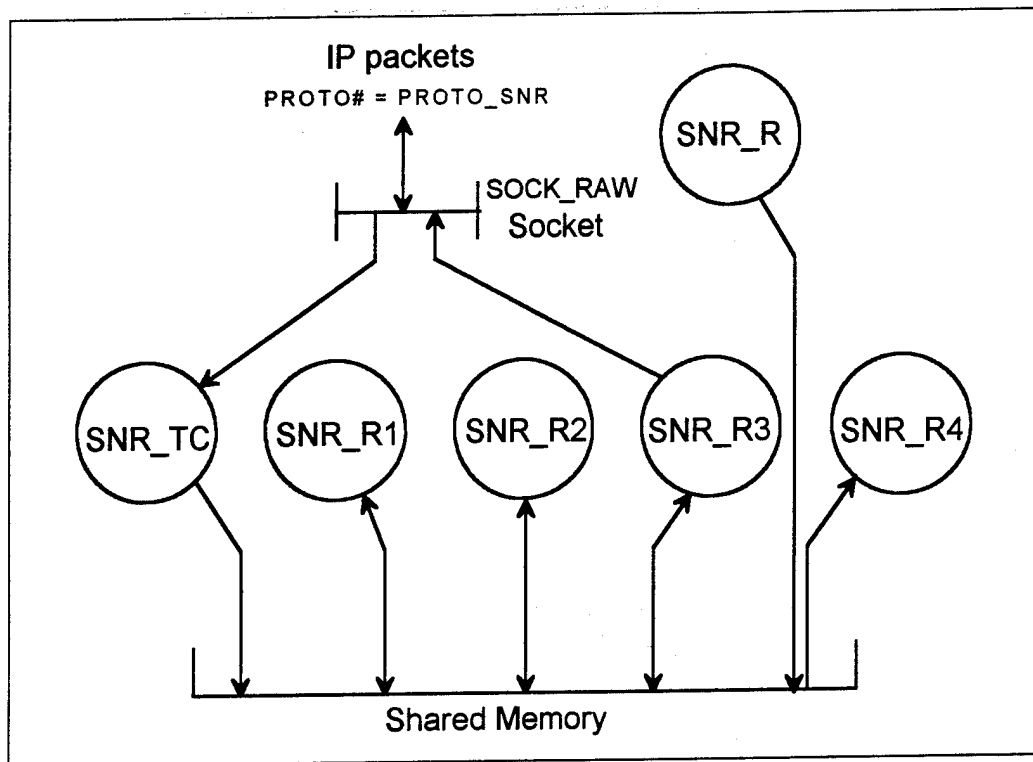
15

Figure 3.1 - SNR Transport Protocol Receiver System Schematic Diagram.

2. Sockets

To avoid using the TCP transport layer that has been provided by the UNIX system, the *SOCK_RAW* option should be used when creating a socket to establish an IP layer communication. This option requires the process that creates the *SOCK_RAW* socket has a root privilege. This was done by creating and assigning the executable file snr_r* with root privilege[1] by the system administrator. The socket descriptor of the created socket would be stored in the shared memory and can be used by other child processes[2] without the need for root privilege.

The eight bit protocol number field in IP header used to identify the SNR transport protocol should be specified when creating a *SOCK_RAW* socket. This number must be different from numbers that already in use by other protocols (e.g. TCP, UDP, ICMP etc.). The protocol number being used for SNR in this implementation was 191. Choice was made by the author of [Ref. 2].

---

[1]   It should be noted that the executable files should only be overwritten to the privileged files by the compiler in order to retain the root privilege vice a normal copying or renaming UNIX command.

[2]   The socket descriptor returned by a socket system call is similar to a file descriptor in the UNIX system [Ref. 4 pp. 269]. This socket descriptor could only be used by the process that created the socket or its child processes that were forked after the creation of the socket [After Ref. 4 pp. 56].

3. Signals

There are two user signals, SIGUSR1 and SIGUSR2, available in the UNIX system. In the Receiver implementation, SIGUSR1 was chosen to commit all the signaling from processes to processes. A process issues *sighold* system call to hold the signal from interrupting before start processing the signal it received and then issues *sigrelse* to release the interruption of the signal. The *sigpause* system call incorporates the release of interruption and pause for the signal in one call. When timer signal was requested for some processes (i.e. machine R2 and R3), the signal SIGALRM was sent by the system clock to the process that requested the service. For each signal that could be received by the process, a signal handler should be provided. If no signal handler was provided, the process would be terminated when receiving the signal.

The signal system calls in the UNIX systems that we used for the implementation were not very reliable. Signals could be lost when a process has issued a *sighold* system call and has not release the hold while more than one signal arrived. The system does not maintain a signal queue for each process. The following functions were defined to try to save all signals from being lost.

• Signal handler - increments a file scope variable EventCnt whenever a signal is received by the process.

```
void SigHdlr() {
    EventCnt++;
}
```

• Function WaitForEvent - waits only when the EventCnt is less than or equal to zero. [After Ref. 4 pp. 53]

```
void WaitForEvent(){
    sighold(SIGUSR1);        /* inhibits the interrupt from SIGUSR1 */
    if (EventCnt <= 0)
        sigpause(SIGUSR1);   /* enable the interrupt then pause for SIGUSR1 */
    else
        sigrelse(SIGUSR1);   /* enable the interrupt from SIGUSR1 */
    EventCnt--;
}
```

• System Call sigset - sets up the signal handler for the signal interested

```
sigset(SIGUSR1, (void (*)())SigHdlr);
```

These functions improved the stability of the program by reducing the signal hold time to the minimum (only one check for the EventCnt is performed within the inhibited area). But more

17

than one signal could still possibly arrive at the process when it inhibits the interruption of the signal (especially when the process is being scheduled to a wait state by the operating system). Some modifications of the original specification was made for the signal lost problem (see Chapter II section B.1 for the change of specification for R1).

4. Shared Memory / Semaphores

The shared memory was used in this implementation for the following reasons:

• Save time for conveying information from machine to machine. e.g. *T_CHAN*, *INBUF* etc.

• Sharing common variables among machines. e.g. *mode*, *LWr*, *UWr* etc.

• Control variable that starts and terminates of all machines. e.g. *SNR_ON*.

In order to prevent the race condition from occurring among processes that share the same data structure in the shared memory, semaphores were used. Table 3.1 illustrates the semaphores and the processes that access to data structures in the shared memory that were controlled by semaphores. In this implementation, all semaphores were created to have a count as 1. That means each resource controlled by a semaphore could only be accessed by one process at a time.

Semaphore	Process	R/W	Controlled shared memory data structures
INBUFsem	R1, R4	W	INBUF, head, RECEIVE, AREC, LOB, LWr, UWr
	R4	R	
SCOUNTsem	R2, R3	W	scount
	R3	R	
T_CHANsem	T_CHAN	W	T_CHAN, T_statePkt, T_stateFlag, Conn_discFlag
	R1, R2	R	
RCVDsem	R1, R3	W	received
	R3	R	

Table 3.1 - Semaphore / Process Access Control Table

The decision of whether to use a semaphore or not on certain shared memory variables were made according to the following rules:

a. The variable could be written by more than one process; or

b. The variable is a compound data structure (e.g. arrays, records etc.) that may cause inconsistency in the reading process if the process could not exclude other processes from writing while in the middle of a read.

18

In this implementation, a rule has been observed that one process should wait on one semaphore (i.e. lock one resource) at a time to prevent deadlocks. A survey through each process in the Receiver showed that in some processes (e.g. R1, R2) there were needs for locking two (and at most two) shared resources at a time. Fortunately, there were always *T_CHAN* involved. So the second rule been used that when a process needs to lock two resources, one of the semaphores should be *T_CHANsem*, and the process should always wait on *T_CHANsem* first before it waits on the other semaphore. Thus the two resources would virtually be controlled by one *T_CHANsem*. An example of two processes each wait on two semaphores at a time is given in Table 3.2.

SNR_R1	SNR_R2
sem_wait(*T_CHANsem*);	sem_wait(*T_CHANsem*);
·	·
·	·
·	·
sem_wait(*INBUFsem*);	sem_wait(*SCOUNTsem*);
·	·
·	·
·	·
sem_signal(*INBUFsem*);	sem_signal(*SCOUNTsem*);
·	·
·	·
·	·
sem_signal(*T_CHANsem*);	sem_signal(*T_CHANsem*);

Table 3.2 - Example of Processes Each Wait On Two Semaphores At a Time

In our UNIX system, the maximum shared memory size allowed was 1 mega bytes. This was defined in system header file <sys/shm.h> by macro definition

#define SHMSIZE   1024 /* maximum shared memory segment size (in Kbytes) */

The shared memory size might need to be increased for a further implementation of the SNR protocol that could handle more than one connection at a time.

The base value for shared memory used when creating the shared memory for the Receiver was chosen to be 8890 [After Ref. 4 pp. 157]. This value should be unique among all applications that require shared memory within the same workstation. If two or more processes on the same machine created their own shared memory using the same base value, the result would be unpredictable. Inexplicable phenomena occurred during the test of shared memory, but was eliminated after different base values were chosen for the Transmitter and Receiver.

The uniqueness of shared memory base value between the Receiver and the Transmitter is guaranteed in the implementations.

5. Scheduling / Processes synchronization

The parallel processing feature of the SNR protocol was implemented by employing the UNIX system multitasking ability. Some problems that were caused by the nature of parallel processing were encountered during the implementation.

a. Race condition between R1 and R4. In the original specification, a race condition exists between R1 and R4 when $R\_active = F$. Normally, when R1 detected $R\_active = F$, if the T_CHAN is empty, it would transit to state 0 otherwise it would process the data packets in T_CHAN and put them in $INBUF$ until T_CHAN is empty then transit to state 0. On the other hand, when R4 detected $R\_active = F$, if $INBUF$ is empty it would transit to state 0 otherwise it would process the $INBUF$ until the $INBUF$ is empty then transit to state 0. In mode 2, the race condition would not occur since the Transmitter should not send $Conn\_disc$ until all blocks are acknowledged. However, in mode 1 the Transmitter does not need to wait for blocks to be acknowledged before sending $Conn\_disc$. So when R2 receives $Conn\_disc$ packet and set R_active to False there might be some data packets still have not processed in T_CHAN by R1. If R4 detected that the $R\_active = F$ first before R1 (by the scheduling of a parallel processing system) and checks the $INBUF$ that it was empty then R4 would consider that the connection is completed and transit to state 0. This causes loss of the data packets in T_CHAN. In order to make sure R1 completes its processing of T_CHAN data packets and set up the $INBUF$ before R4 considered that the $INBUF$ is empty, the SNR_TC when received $Conn\_disc$ packet would notify R1, and R1 will notify R2 when all data packets in the T_CHAN has been processed[3]. A timing diagram illustrates the occurrence of the race condition in Figure 3.2.

b. Conn_disc missed by R2. In the specification, R2 should check $T\_CHAN[front]$ to see if the packet type $Conn\_disc$ is received and do the appropriate processing. In the testing of the early implementation, when R2 receives the notification from SNR_TC about the arrival of $Conn\_disc$, the front of T_CHAN is occupied by $Data$ packet. The problem could also be solved by having SNR_TC notify R1 about the arrival of $Conn\_disc$ and let R1 notify R2 when R1 has processed all data packets in the T_CHAN.

---

[3]   This modification was not intended to solve the late arriving data packets that come after $Conn\_disc$ due to the network routing. This kind of loss is not guaranteed by mode 1 in the SNR protocol specification.
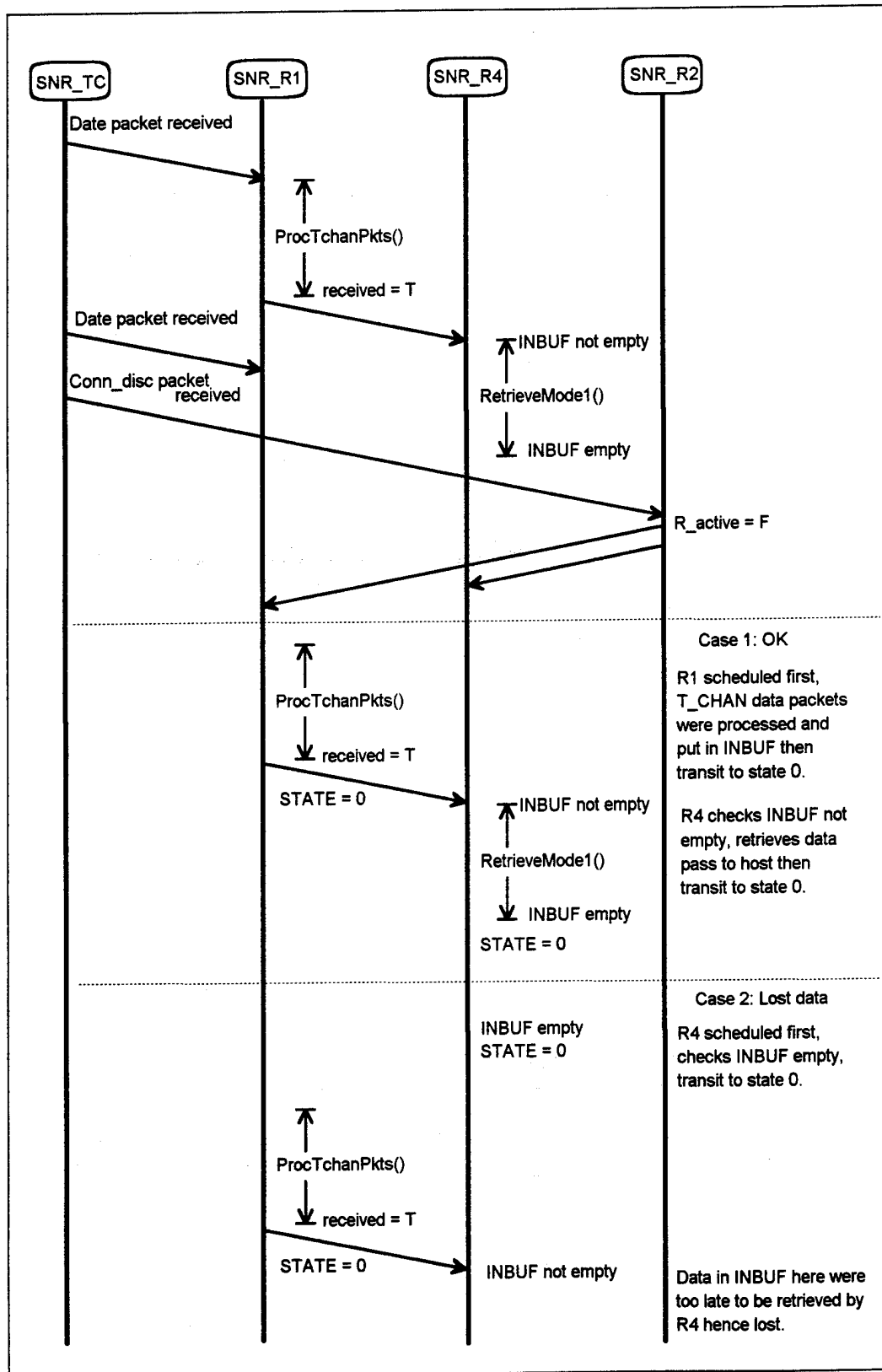
Figure 3.2 - Timing Diagram of Race Condition Between R1 and R4

21

c. T_state missed by R2. A similar problem exists for *T_state* packets that would be interleaved with data packets in the *T_CHAN*. The remedy of this problem was to treat the *T_state* type packets as out-of-band packets. In the shared memory, one space was allocated for *T_state* packet. The SNR_TC would refresh this packet whenever a *T_state* packet is received from the Transmitter. A *T_stateFlag* in the shared memory would be set by SNR_TC to indicate that the *T_state* packet has been refreshed[4].

d. Child processes are scheduled less frequently than parent process. Originally, in order to categorize functions into separate modules, a timer process was implemented to handle the timer service. Whenever a process needs periodic signals from a timer, it forks a timer process to do the job. It was found that the timer process has a very low priority that it rarely be scheduled to perform its task. That made the timer signal used by R3 for state information exchange not very smoothly and hence the Transmitter has to be tuned loosely to avoid frequent timeout disconnection. Once noticed of this, a design decision were made to transplant codes in the timer process into the client processes and the problem was eliminated.

6. Data Alignment

There were problems related to the C compiler. When we define some large data structures (e.g. array, struct etc.), if a structure is not started at an address of a multiple of four (possibly related to the size of a machine word), the compiler would padded memory spaces before the data structure with zero to make it so. This problem caused miss alignment of field accessing in the structure by the machines on the other side. So far, the data structures that encountered this problem in our implementation were *SNRhdr_t* and *SNRrstate_t*.

In the *SNRhdr_t*, originally we added the *datalen* field as an unsigned short integer (two bytes in size), that made the total length of the header six bytes. Thus when the data part (defined as array of unsigned characters) of the packet were appended to the header to form a complete packet, two bytes of zero were padded to the end of the header by the compiler. This meant the Receiver could not retrieve data correctly. To overcome this problem, the data type for *datalen* field was changed to integer (four bytes in size). That made the total length of the header eight bytes and the starting address of the data following the header was a multiple of four and no dummy bytes were padded.

---

[4] A check on the sequence number should still be made by R2 to see if the refreshed *T_state* packet is newer than the previous *T_state* in accordance with the specification.

Similarly, the *SNRrstate_t* has an *LOB* followed by an integer *errChk* (the error checking word). Originally, the *LOB* has a size of ten. There were two dummy bytes added to the end of the *LOB* by the compiler to make the *errChk* starts at an address of multiple of four. The miss alignment problem happened again during test, with the experience of the data packet, we decided to change the *LOB* size to eight. That made the *errChk* start at an address which was a multiple of four and solved the problem[5].

B. RECEIVER

The SNR protocol Receiver was implemented by using six processes. They are SNR_R, SNR_R1, SNR_R2, SNR_R3, SNR_R4 and SNR_TC. The program structures are described in the following section.

1. Receiver Root (SNR_R)

The SNR_R is the Receiver root process of all the other machines. The SNR_R is also the only process that requires the root privilege to create the *SOCK_RAW* socket. When executed by the system administrator, the SNR_R creates shared memories, semaphores, sockets and initializes shared memory data structures (e.g. *T_CHAN*), then it sets the shared memory variable *SNR_ON* to *True* and activates all the receiver machines (SNR_R1 through SNR_R4) and the transmitter channel (SNR_TC). Once all these been done, the individual Receiver machines are started to work on their own. The SNR_R then waits for the command from system administrator to terminate the Receiver. The SNR_R, when terminated by the system administrator, sets the *SNR_ON* to *False* then informs each machine and waits for them to terminate. After all machines are terminated, SNR_R removes the shared memory and closes all semaphores created in the process.

2. Receiver Machines (SNR_R1, SNR_R2, SNR_R3 and SNR_R4)

All four Receiver machines were designed using the same program structure. Take SNR_R1 as an example. The SNR_R1 when activated, first gets the shared memory id using the same key (i.e. *SHMKEY*) that SNR_R used to create the shared memory, attaches to the shared memory by a pointer *Shm*, then opens the semaphores *INBUFsem*, *RCVDsem* and *T_CHANsem* that was created by SNR_R and going to be used in this process. The SNR_R1 then sets up the

---

[5]   Making the LOB size multiple of four is not really a good way of solving the problem since there is a restriction on LOB size. Another way of solving this problem could be changing the *errChk* type from integer to four consecutive unsigned characters and cast them to integer when being used. This way, the restriction on *LOB* size would be relaxed.

signal handler for the signal *SIGUSR1* that was designated as the signal for the Receiver process communication. When these are done, the SNR_R1 falls inside a *while* loop that is controlled by the shared memory variable *SNR_ON*. Whenever the *SNR_ON = T*, the SNR_R1 keeps on waiting for event (in this implementation, pause for the *SIGUSR1* signal) to occur and process the event according to the state that SNR_R1 is in. When *SNR_ON = F* (i.e. the Receiver is terminated by the system administrator), SNR_R1 exits the *while* loop, detaches the shared memory, closes all the semaphores and then exits.

3. Transmitter Channel (SNR_TC)

The SNR_TC has some differences in the program structure from the four Receiver machines. The SNR_TC, when activated, would get and attach the shared memory, open semaphore and set up the signal handler in the same way as the four Receiver machines do. Then the SNR_TC falls inside the *while* loop that is controlled by *SNR_ON*, however, instead of waiting for events to occur, the SNR_TC waits on *T_CHAN* for the IP packets that were packed and sent by the Transmitter to arrive. When an IP packet is received in the SNR_TC, it peels off the IP header and extracts the SNR packet from the IP packet. Check sum is also performed during the extraction of the packet. Then, depends on the SNR packet type, the SNR_TC enqueues this packet and notifies the relevant machine to process the packet. The termination of this process is the same as all the other four Receiver machines.

IV. TESTING AND TUNING

The SNR protocol Receiver implementation was incorporated with the Transmitter and tested using three UNIX workstations. These three workstations were connected next to each other using FDDI. The testing environment is shown in Figure 4.1.
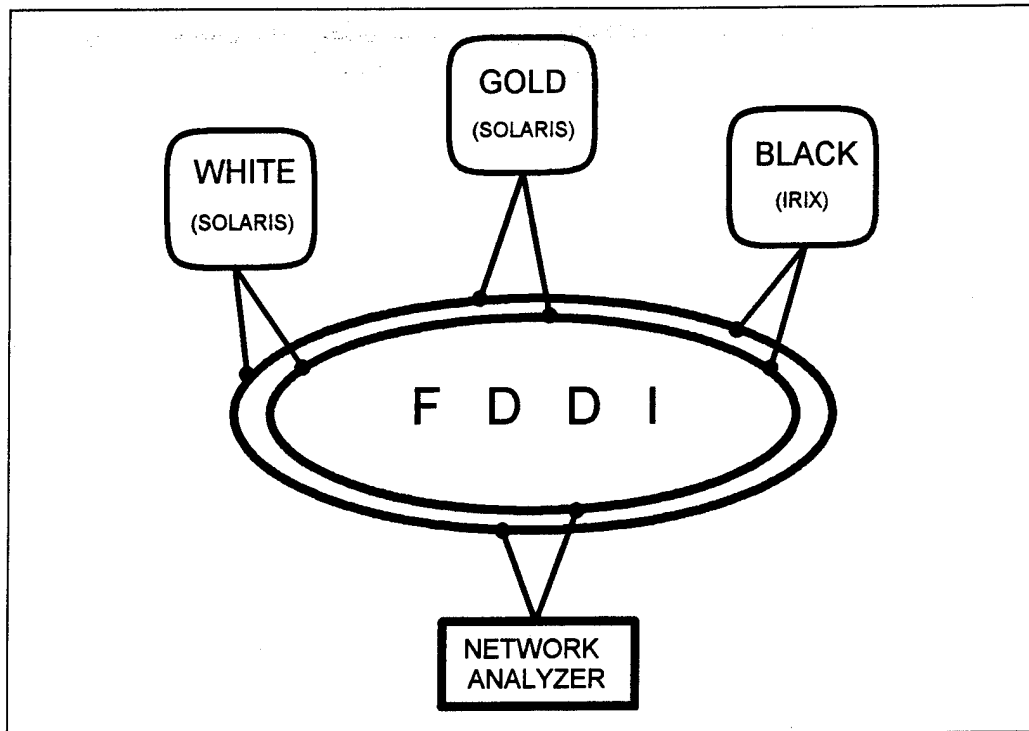
Figure 4.1 - Testing Environment for SNR Protocol Implementation

Three workstations, namely WHITE, GOLD and BLACK, run UNIX operating system in two different versions. The WHITE and GOLD run SOLARIS while BLACK runs IRIX. Not too many implementation differences were found between these two systems[6]. A network analyzer was connected to the network for debugging and testing. The analyzer shows information of the IP packets that were put onto the network for communication by any machine in the network. From the analyzer, we could read the information of each field in the SNR packets by counting the byte position in the IP packets. In addition, the protocol number, source and destination of packet could be specified to filter out uninterested packets for the analyzer display. It was very helpful to have an analyzer for the testing of this implementation.

---

[6]   See snr_env.h file in Appendix for more information.

A.   GENERAL

When testing, the Receiver should be invoked first from one workstation. Once the Receiver is started, it will receive and process any IP packet that has its protocol number set to 191 (macro defined as *PROTO_SNR* in snr_tr.h header file). Then from another workstation, the Transmitter is invoked, it immediately starts the establishing of connection and transmission of packets to the specified host[7]. A group accessible file[8] called *README* was stored in a directory that can be accessed by the Transmitter for testing. In order to keep track of the current status of each process during the test, extra codes were added to the program to print out some important messages to the screen.

B.   USING DIFFERENT MODES FOR CONNECTIONS

Three modes as mentioned in the specification were tested. In mode 0, there is no flow control and error checking (i.e. retransmission). In mode 1, only flow control is adopted. In mode 2, there are both flow control and error checking. In the testing, the *LWr* and *LOB* information would be printed out on the screen when the *R_state* control packets were received on the Transmitter side. If a retransmission occurs, the retransmission information would be printed out to the screen as well. When the connection is completed, the Receiver would print out all the data that has been received for verification. It appeared consistently in the test that mode 0 was the fastest mode among all modes. The modes 0 and 1 have very low error rates. It was estimated that over ninety percent these two modes could get all data correctly transferred. Mode 2 showed its power by a hundred percent correct rate but with a disadvantage of lower performance.

C.   LOST PACKETS

To simulate the packet lost in a natural way, in the Receiver SNR_TC process, a piece of code were added to deliberately skip one packet with a chosen sequence number for once.

In mode 0, this test could be used to check whether the loss really occurred. In mode 1, this test shows how the flow control could be affected by the lost packet. During the test in

---

[7]   In the specification, no port number could be specified for a connection thus connection could only be established from host to host. Also note that the port number could be added to the SNR packet header for entity (applications such as FTP, e-mail, rlogin etc.) to entity connection in further extension of this implementation.

[8]   The file that is transferred by this protocol would only need to be accessed by the Transmitter side. However, it would be convenient during testing to have the designer of the Transmitter as well as the designer of the Receiver (in the same user group) to be able to change the file in order to test different sized files individually.

mode 1, the *LWr* (lower window in Receiver) would be stuck by the lost packet and later be released when the number of packets received after the lost packet exceeded the *wait_lim*. In mode 2, this test caused retransmission of one whole block of packets.

D.   DUPLICATE PACKETS

Codes were added to the SNR_R1 to print out messages when a duplicate packet was received by the Receiver. By tight adjustment on some retransmission parameters (that could cause some unnecessary retransmissions) on the Transmitter side, duplicate packets were generated. This test is mainly conducted in mode 2 since no other mode does retransmission. It appeared that in the test, all duplicate packets were detected and discarded by the Receiver. No duplicate packet test was conducted for mode 1 since it uses the same predicate condition as mode 2 does for duplicate packet checking.

E.   LARGE DATA TRANSFER

This test is mainly testing the buffer wrapping around ability. In this implementation, the Receiver *INBUF* and the related data structures (i.e. RECEIVE, AREC) could accommodate 1024 packets. We tested this by sending a file big enough to have these data structures be exhausted and wrapped around for reuse. Due to the fact that in this implementation, the block number field was implemented using unsigned character (1 byte), thus the maximum block number could be 255. In other words, the maximum number of packets that could be transferred in one connection would be 255 * maximum block size. The maximum block size is limited by the packet number field that was implemented using unsigned character as well. However, the choosing of block size is not that flexible since it would be related to the number of packets to be transmitted in a retransmission. In our test, a block size of 8 was the most often been used since it requires less packets be transmitted in a retransmission. This made the maximum number of packets that can be transmitted in one connection 2040 packets. However, the block size and packet size could be determined at runtime during connection establishment by negotiation of parameters. Thus, if an application needs to transmit a large file, it should use larger block size and larger packet size for the connection. In this implementation, the largest file that can be transferred in one connection[9] is

---

[9]   The limitation only applies to mode 1 and mode 2 since these two modes call *OrderInsert* that requires block number and packet number information. In mode 0, the block number and packet number are irrelevant, thus the restriction of these two factors are not applied. Hence, theoretically, mode 0 could keep on transferring

maximum block number * maximum packet number * maximum packet data length[10]

$= 255 * 255 * (128 - 8 - 4) = 7,542,900$ bytes.

F.   TRANSFERRING DATA FROM SAME WORKSTATION

This test checks if the implementation could transfer data in one machine from one shell to another shell by running the Receiver on one shell and invoking the Transmitter on another shell and specify the same host name as the transferring destination.   With a little lower performance due to the increased load of the system, the test was conducted successfully.   By the use of the network analyzer, we observed that the IP protocol was intelligent enough to recognize a packet being sent to a process in the same machine, and thus to simply pass it directly to the process and not to put it on the network.

G.   TUNINGS

The goals of tuning for this implementation were as follows:

1. To Ensure the Connection Could be Established As Long As the Receiver is Activated And the Network is Working

During the connection establishment phase, the Receiver requests a timer then waits for the Transmitter to send *Conn_conf* packet. If the timeout occurs before the *Conn_conf* packet is received, the connection would not be established.   To ensure the time out would not occur if the network is working, we need a longer period for the timeout to occur in the Receiver.

The parameters in the Receiver side related to this issue is *InitIPT* and *TOtick.*

TOtick is the number of times that *Tin* time interval expired.   This value incorporated with *InitIPT* decided the Receiver time out during a connection establishment.   The larger these values are, the more likely the connection could be established.   However, the trade off is the amount of time the Receiver should wait to initiate a timeout disconnection when there is a problem on the network or the Transmitter.   In our test, with a *TOtick* of 32 and *InitIPT* of 100 ms, the Receiver never timed out during the connection establishment.

---

packets in one connection as long as the timeout in R3 does not occur. (see Chapter II for change of specification for R1 about the discussion to extend the timeout in R3 in mode 0)

[10]   In this implementation, the maximum SNR packet length is 128, packet header length is 8 and ErrChk word size is 4. Thus the maximum SNR packet data length is 116.

2. To Minimize the Unnecessary Retransmissions

A retransmission would occur when the Transmitter does not receive *R_state* control packets that carry information for update in time. There were two possibilities that may cause an unnecessary retransmission. Firstly, the *R_state* packets were sent not frequently enough that the Transmitter could get the update information in time. Secondly, the Transmitter sets a stringent condition for retransmission.

The parameters in the Receiver side related to this issue are *Tin*, *kou* and *IPT*. The *Tin* was decided by the following formula at runtime.

$$Tin = \max(RTD/kou, IPT)$$

Since the *RTD* should be decided at runtime, the other factors that could affect the result of *Tin* would be *kou* and the *IPT*. To minimize the unnecessary retransmissions, we need to have the *Tin* as small as possible. So we need a large *kou* and a small *IPT*. However, the trade off for a small *Tin* is that more control packets were sent to the Transmitter.

In our test, normally the retransmission seldom occurs if there is no packet lost. However it occurred more often when there were many processes running on the testing workstations (e.g. other users rlogin, more shells were created etc.) since the machines for the protocol were less often being scheduled whereas the timers were not affected too much by the processes load. This brought up an issue that the protocol might need to consider about the load of the host by monitoring the load of the system and putting a *load* parameter in the parameter negotiation or in the control packets and use the *load* parameter to affect the condition for retransmission.

3. *To Have R_state* Control Packets Being Sent In A Sufficient Manner

More control packets mean more load on the machines and the networks. So we need to tune the sending of control packets so that the Transmitter could detect the loss of a packet and retransmit it early enough before the Transmitter be blocked by the flow control (due to the lack of refreshing *R_state* information) from transmitting new data packets.

The parameters in the Receiver side related to this issue are *Tin*, *kou*, *IPT* and *Klim*. For *Tin*, same discussion was provided in section 2. The *Klim* determines the limit of the *k* value that would be used to decide number of *Tin*'s when a data has not been received in the Receiver

before transmitting an *R_state* packet. The larger the *Klim* is, the less frequent the *R_state control* packet would be transmitted when there is no data packet been received by the Receiver.

In our test, the *R_state* packets were sent with the *LWr* jumps approximately every four blocks (i.e. each *Tin* time interval, four blocks were received by the Receiver). A retransmission would occur when the fourth *R_state* control packet was received by the Transmitter after a packet lost. The *LWr* jumps approximately sixteen blocks after the Receiver receives the retransmission data packets.

4. To Have Timeout Disconnection Initiated Only When Problem Really Occurred On the Connection

The Receiver would timeout disconnect when R3 does not receive *T_state* for *Lim* implied time period. Thus the adjustment of *Lim* incorporated with *Tin* would decide the disconnection timeout. In our test, the timeout disconnection never occurs when no error occurs in a connection.

The final tuning parameter values are provided in Table 4.1. All the test results mentioned in this Chapter are based on these parameters. These parameters were obtained by testing under the packet data length of 8, block size of 8, window size of 64 and an estimated average RTD of 100 ms.

Due to the fact that the protocol was implemented and tested only under the simple environment as mentioned above and not sufficient samples were gathered, the tuning parameters provided here are immature. This Chapter gives an idea that how a test and tuning could be conducted for this implementation.

Parameter name	Value	Remarks
kou	2	power of 2 constant, used by R2 to decide *IPT*.
TOtick	32	time out ticks for *reset*, used by R2.
InitIPT	100,000	initial *IPT*, in microsecond, used by R3.
AcceptableRatio	3/4	acceptable ratio, used in R4 mode 1 to determine *wait_lim*
Lim	32	*scount* limit, used by R3 for disconnect timeout.
Klim	32	*count* limit, used by R3 for send R_state timeout.

Table 4.1 - Receiver Tuning Parameters

V. EVALUATION

This implementation complies the SNR protocol specification [Ref. 1]. Some differences exist due to various reasons described in Chapter II. Extra efforts in almost every way were put on program documentation and maintainability for the possible extension of this implementation in the future.

A. MAINTAINABILITY

1. Relationship Between Variables

In snr_tr.h file, all related constants were defined using formulae to depict their relationships. For example, to allocate a larger *INBUF* for the implementation, only one constant *MaxBufSize* need to be changed. By changing the constant *MaxBufSize*, the constants that related to it (e.g. *RCVsize*, *ARECsize* etc.) would be changed automatically by the relationship formulae. This saves the time for a programmer to understand and compute these values manually and hence reduces the possibility of making mistakes.

2. Implementation Limitations

All the limitations that were adopted were collected in snr_tr.h file.

a. *MaxLci* = 1. The *MaxLci* is the maximum allowed logical connection identifier (LCI). In this implementation, only one logical connection is allowed. The usage of the constant *MaxLci* is not strictly enforced in this implementation.[11] However, the inclusion of this constant in this implementation points out a direction toward the future multiple LCI implementation.

b. *MaxPktSize* = 128. This constant specifies the maximum packet size that could be used for the SNR packet. This number also implies the maximum data length that could be used in each packet by the formula *SNRdataLen = MaxPktSize - SNRhdrLen - ErrChkSize*. Since this constant affects the storage space of the major data structures (e.g. *SNRpkt_t, INBUF, T_CHAN* etc.) in this implementation, the consideration of the value for this constant would be the limitation of the shared memory size that can be allocated for the Receiver.

---

[11]   One possible multiple LCI implementation could be that all the data structures that would be used for a connection be declared as an array of size *MaxLci* and indexed by the LCI for that logical connection (e.g. *STATE[MaxLci], LOB[MaxLci][LOBsize]* etc.). These kind of declarations or accessing of variables do not appear in this implementation.

c. *MaxWinSize* = 64. This constant specifies the maximum window size allowed for the communication protocol. It is used for determining the *LOBsize* that would be used in *R_state* and *T_state* control packets to be put onto the network periodically during a connection. However, the size of *LOB* should not be a main factor for the consideration of *MaxWinSize* constant since the *LOB* size is affected in bit (i.e. one block in the window would be indicated by one bit in the *LOB*) by the constant. More specifically, this constant should be considered just as its role in the flow control and trying to gain efficient use of the network resource (e.g. the high speed communication media etc.) vice storage space saving or negligible load on the network. Note that since the declaration of *LOB* in this implementation is array of unsigned characters (bytes), the value chosen for this constant should be a multiple of 8.

d. *MaxBufSize* = 1024. This constant is the size of *INBUF* and *T_CHAN* thus the consideration for this constant should be a storage space issue. Note that this constant would also be used for determining the size of *RECEIVE* that is used in bit and declared as array of unsigned characters so the value for this constant should also be a multiple of 8.

3. Tuning Parameters

All constants that related to the tuning of this protocol were gathered in snr_r.h file for convenience. These parameters include *kou*, *TOtick*, *Klim*, *Lim*, *InitIPT* and *AcceptableRatio*. Refer to Chapter IV for the usage of these parameters.

4. Functions for Application Program Interface (API)

Due to the fact that the API for the SNR protocol has not been designed, all the operations related to the communication with the host in this implementation were implemented as functions with meaningful function names. In these functions, messages were printed out to simulate the operation. The possible ways of using the data structures of this implementation in an application interface were instanced by some of these functions (e.g. *PutOutBuf, ShowOutBuf* etc.) as well. The files that have functions or operations in the implementation that related to the API were described as follows:

a. snr_r.c - the Receiver Root Process

In this implementation, for the system administrator to control the termination of this protocol, input on the console terminal through keyboard was checked in the Receiver root

process. This could be changed in the design of an interface between the system administrator (or a system initialization program) and this protocol.

The implementation limitation constants *MaxPktSize*, *MaxBufSize*, *MaxLci and MaxWinSize* could be determined by the system administrator with the concern of the size of shared memory and the load of the system when invoking the Receiver (rather than hard coded in the program requiring compilation after each change) in the interface design.

b. snr_r1.c - the Machine R1

The following functions are the API related functions for machine R1.

• *PassDataToHost* - Passes the data obtained from a data packet to the host.

• *PutOutBuf* - Puts data in the *OUTBUF*. This function gives an example for API.

• *ShowOutBuf* - Displays the *OUTBUF* to the screen. This function gives an example for API.

c. snr_r4.c - the Machine R4

The following functions are the API related functions for machine R4.

• *NotifyHostAboutConn* - Notifies the host about the connection is established using a specified mode.

• *NotifyHostAboutDisc* - Notifies the host about the connection is disconnected.

• *NotifyHostAboutCompletion* - Notifies the host about the connection is completed.

• *PassDataToHost*, *PutOutBuf* and *ShowOutBuf* - Refer to section A.4.b.

5. Portability

In this implementation, all the differences related to different versions of UNIX system (so far, SOLARIS and IRIX) were gathered in the header file snr_env.h and were covered up by macro definitions. It is possible that when trying to port this implementation to other systems more differences will be found and put into this file. By putting all these differences in the header file, one avoids the conditional compilation directives in the programs, makes the programs easier to read and system independent.

B.   DAEMON PROCESS

Efforts were spent to implement the Receiver as a daemon process. The first step being made is to have the Receiver serve for different connections repeatedly without the intervention of a person. The Receiver could continuously handle different modes and connection parameters (e.g. packet size, block size etc.) that were specified on the Transmitter side when invoked for different connections

Although most of the characteristics of a typical system daemons[12] could be found in the Receiver implementation, it is still far from a real daemon process because it is not disassociated with the control terminal (thus it will tie up the terminal while it is executing), and it cannot be started and terminated by the system **init** process etc. as described in [Ref. 4].

C.   EVENT DRIVEN

In this implementation, processes are basically idle (pausing for signal) when no event occurs. This saves CPU time that could be used for other processes running on the same machine. Events that are generated by the processes of this implementation are referred to as internal events in this paper. Events that are generated by the Transmitter, system administrator or other processes which are not part of the Receiver implementation are called external events. The external events are considered more natural than the internal events since the generation of the external events are not that "artificial."

In this implementation, an external event occurs when a packet is received by the SNR_TC process through T_CHAN. This causes an internal event and signal to be sent to the

---

[12]   In [Ref. 4] pp. 73, the characteristics of a typical system daemon were specified as following:
   • They are started once, when the system is initialized.
   • Their "lifetime" is the entire time that the system is operation; usually they do not die and get restarted later.
   • They spend most of their time waiting for some event to occur at which time they perform their service.
   • They frequently spawn other processes to handle service requests.

relevant process that is responsible for this packet type by the SNR_TC process. In this implementation, internal events never occur spontaneously. It is always the byproduct of an external event. Table 5.1 summarizes the internal events in this implementation.

Internal Events	Generated in Process	Relevant Processes				
		SNR_R1	SNR_R2	SNR_R3	SNR_R4	SNR_TC
Conn_req received	SNR_TC		•			
Conn_conf received	SNR_TC		•			
Data received	SNR_TC	•	•¹			
T_state received	SNR_TC		•			
Conn_disc received	SNR_TC	•²				
R_active = T	SNR_R2	•		•	•	
R_active = F	SNR_R2	•		•	•	
Disconnect = T	SNR_R3		•		•	
received = T	SNR_R1				•	
time tick	SYSTEM³		•	•		
SNR_ON = F	SNR_R	•	•	•	•	•

[1]   If the *Data* packet received before *R_active = T*, then SNR_R2 would be notified, otherwise the SNR_R1 would be notified.

[2]   Instead of notifying SNR_R2 about the receiving of *Conn_disc* packet, the SNR_R1 is notified on concern of the race condition between R1 and R4. (see Chapter III section 5.a)

[3]   The signal for time tick generated by the system timer is considered as an internal event since it is requested by and for the SNR_R2 and SNR_R3 processes in different phase of a connection.

Table 5.1 - Summarizing of Internal Events for the Receiver Implementation

D.   NEGOTIATION OF PARAMETERS

In this implementation, all final decisions of the negotiation parameters to be used for each connection are made on the Transmitter side. The Receiver responds to the *Conn_req* packet sent by the Transmitter with the *Conn_ack* packet by filling in the maximum capabilities that the Receiver has. Then the Receiver accepts whatever the parameters that were confirmed by the Transmitter in the *Conn_conf* packet. Rules should be followed by the Transmitter to choose the parameters that would not exceed the Receiver's capability. This way of design was based on the

philosophy that the Receiver is a passive entity that it should always try to cooperate with the Transmitter (that carries the request from the client - the application) without launching any unnecessary interference.

E.   GRACEFUL TERMINATION

By using the shared memory variable *SNR_ON*, the Receiver machines (child processes) could be terminated gracefully by the SNR_R (Receiver root). Each machine, when receiving a signal, checks if *SNR_ON* is *True* before it proceeds. If *SNR_ON* is *False*, the machine exits the *while* loop and does some housekeeping to terminate itself gracefully. The SNR_R does the housekeeping for its termination after all the child processes are terminated. By "graceful", it is meant that no timer is requested by SNR_R for the waiting of the child processes to terminate, and no kill signal (i.e. SIGKILL) is sent to abruptly terminate a child process.[13]

F.   FURTHER EXTENSIONS

There was still some work needed to be done in order to reach the goal that the SNR protocol can be used to replace the current transport protocols. The works are briefly described as follows:

1. Multiple Logical Connections

As mentioned in section A.2.a, this implementation could allow for only one logical connection at a time. To improve the number of logical connections that can be handled simultaneously in the Receiver, three schemes are suggested as follows:

a. One root, forks one set of Receiver machines with multiple (i.e. MaxLci) sets of data structures one for each connection. In this scheme, all the data structures that would be used for a connection should be declared as an array of size *MaxLci* (e.g. *STATE[MaxLci]*, *LOB[MaxLci][LOBsize]* etc.) and indexed by the LCI for that logical connection. It would be inefficient to use the logical connection identifier as an array index since by the nature of the

---

[13]   If the Receiver is not terminated properly (e.g. by typing a contol-C on the control terminal etc.), there would be shared memory and semaphores left over in the host system and need to be cleaned by the user that invoked the Receiver or by a system administrator with root privilege.

LCI it could be not be consecutive in the Receiver side and becomes very big. This problem could be solved by a table look up that converts an LCI to an array index. The structure of the table is shown on Table 5.2.

Index	LCI	Validity
0	123	1
1	246	1
2	xxx	0
3	xxx	0
.	.	.
.	.	.
.	.	.
MaxLci - 1	xxx	0

Table 5.2 - Logical Connection Identifier and Array Index Conversion Table.

The advantage of this scheme is:

• Fast response for connection establishment (no forking processes needed for each connection).

• Only one set of processes exist through out the whole service period that saves the operating system overhead for process management. This also speed up the processing after the connection has been established.

Some problems or disadvantages that come with this scheme are:

• Memory allocated for multiple connections may not be used efficiently. Although there might be ways for dynamically allocate shared memory that could be shared among all machines during runtime for each connection but this is considered as optional savings for this scheme.

• Need to identify signals that are sent for a specific logical connection. In the current UNIX systems, there is no way to identify a signal in a receiving process. Thus, when an internal event[14] occurred in a process, before notify the relevant process about this event, some provisions would be needed to make sure the processes being notified know which LCI this signal related to in order to perform the proper processing for that logical connection.

b. One root, forks one set of machines for each connection when the connection is requested. In this scheme, there is only one root process exist when no connection is established.

---

[14] Only the internal events would require the identifying of signal since the external event would only occur when receiving of packet or termination of this protocol. For the former case, the LCI information is provided in the packet that received, for the later case, the LCI is irrelevant.

37

This root process would fork one set of processes for each connection. This is what a normal daemon process would do. The advantages of this scheme are:

- Simplifies the design of using multiple connection data structures.
- Efficiently use of the system resources (processes, shared memory etc.).
- Eliminates or reduces signal identification problem.

However, some problems and disadvantages exist in this scheme:

- The root process should get involved in the connection establishment phase (at least the root process should be notified when a *Conn_req* packet is received in the *T_CHAN*). Some functions related to connection establishment up to a certain level (e.g. check the front of *T_CHAN*, get the packet type etc.) would be necessary to transplant to the root process. This could complicate the design and affects the role of machine R2 in the connection establishment phase. From the above discussion, it might lead to a solution by using machine R2 as the root process. But then all the relationships between R2 and other Receiver machines are affected. All the works that were currently performed by the root process should be moved to R2. For each connection, R2 should create the shared memory and semaphores and remove them when the connection is completed. State 0 would no longer be needed for each machine except for machine R2 since the other machines terminate when a connection for which they were activated is completed. In addition, signal identification problem as mentioned in section F.1.a would again occur in R2 since it has to handle multiple logical connections.

- The response time for a connection request on the Receiver side would be slow due to the forking of processes.

- The system load for managing processes would be heavy when many logical connections being established.

c. One root, forks maximum sets of machines before connection is requested. In this scheme, the root process forks multiple (i.e. *MaxLci*) number of processes and have them waiting for connection request to occur. Since this scheme wastes system resources greatly, it is considered as the last scheme that would be adopted for the multiple logical connection implementation. No further discussion is provided for this scheme.

One common issue that might be overlooked for the implementation of multiple logical connection is the use of semaphores. A semaphore is identified by the semaphore id which is

obtained when creating the semaphore. There should be a unique set of semaphores (that could be indexed by the LCI in the same way as mentioned in section F.1.a) provided for each connection.

Among all the three schemes, the first scheme is recommended since it keeps the nature of each Receiver machine as per specification; it has a good response time both in connection establishment phase and after the connection is established; the signal identification problem would be nasty but still can be solved (busy-waiting could be the last resort); the memory expenditure would become less and less significant as the evolution of computer technology. Most of all, this scheme is the only scheme that could work on a separate hardware protocol processor described in the original design of this protocol in [Ref. 3] since that only contains one set of protocol machines.

2. Entity-to-entity Communication

In this implementation, the connection could be established for the host-to-host (i.e. workstation to workstation) communication level. In order to improve the communication to an entity-to-entity level, the source and destination port numbers[15] should be specified for each connection. This could be implemented by adding two fields - *SrcPortNr* and *DestPortNr* in the connection type packets (i.e. *SNRconn_t* in this implementation). After the connection been established, the port number should be stored in shared memory array data structure declared as *SrcPortNr[MaxLci]* and *DestPortNr[MaxLci]* that can be indexed indirectly[16] by the logical connection identifier. However, the entity-to-entity communication issue would better be considered when designing the API.

3. Application Program Interface (API) Design

The Application Program Interface is the interface that is supported by the transport protocol and used by the higher layer applications. In designing the API for the SNR transport protocol, the functions that were described in section A.4 in this Chapter should be considered. The port numbers and all the negotiation parameters (e.g. connection mode, window size etc.) should be specified through the API by the application that initiates the connection. Two

---

[15]   In TCP or UDP, some applications were assigned with well known port numbers for identification (e.g. the File Transport Protocol has a port number of 12 in TCP, the Telnet server is on TCP port 23 etc.) [Ref. 5].
[16]   A table lookup would be necessary for converting a logical connection identifier to an array index since the logical connection identifier could be inefficient to be used as an array index directly.

popular API for applications using the TCP/IP protocols are called sockets and TLI (Transport Layer Interface), the former was originally developed by Berkeley Software Distribution (BSD), and the later was originally developed by AT&T. Detailed description on both the socket and TLI are provided in [Ref. 4]. It is recommended that if possible an API design for the SNR protocol could provide all [17] of the interfaces that these two API have adopted to make the existing applications that might be needed to alter to the SNR protocol much easier.

4. Port Onto Other Operating Systems

Currently, this implementation could only work on two versions of UNIX system (i.e. the SOLARIS and IRIX). However, many other operating systems exist today. The least requirements for an operating system that this implementation could be ported onto is that it should provide multitasking, signal, timer, semaphore operations, shared memory and IP layer communication. However, the IP layer communication may not be necessary for a local area network but then a lower layer communication should be available and the socket operations in this implementation would all need to be changed.

G.   UNCLEARED ISSUES

The following issues related to the SNR protocol were not clearly specified in any publications. Some possible solutions are given below:

1. LCI Conflict Resolving

It was assumed that a unique logical connection identifier would be available whenever a transmitter needs to establish a connection. This is true if there is only one host that could transfer data to the Receiver. Since then, the transmitter side could maintain a cyclic sequence of LCI to be used for next connection and no LCI conflict would occur. However this is not always true. When more than two hosts could possibly establish connections for data transfer to a specific Receiver, if the selections of LCI are not well coordinated among these protocol machines on different hosts, the conflict would occur and that would mess up the connections. A possible solution is provided here under the premise that an LCI conflict could only occur

---

[17]   To provide an exactly complete set of interfaces that the socket or TLI have provided is very important. Thus the existing applications would need the least modification for the alternation to the SNR protocol. For some sockets or TLI interfaces that exercises some TCP specific features that are not provided by the SNR protocol, empty function body with the same function signature could be used.

during connection establishment phase when the Receiver receives a *Conn_req* packet that carries an LCI that has already been used on the Receiver side (i.e. the LCI appears in the LCI table that was described in F.1. Table 5.2). Under this situation, the Receiver should be able to detect the conflict by the packet type and the free slot allocating operation in the LCI table. The Receiver then tries to resolve the conflict by allocating an LCI that is larger by at least *MaxLci* than the Transmitter requested LCI and not appear in the LCI table[18]. This LCI is stored in the LCI table and filled in a field in the *Conn_ack* packet for this provision. On the Transmitter side, if it observes a rule that it always chooses the LCI increasingly by 1 for each connection, the Receiver acknowledged LCI should not cause a second conflict with the LCI that is in use in the Transmitter side and thus the resolved LCI could be adopted and be used for that connection started from the *Conn_conf* packet and so on. A timing diagram illustrates the two way LCI conflict resolving protocol is provided in Figure 5.1
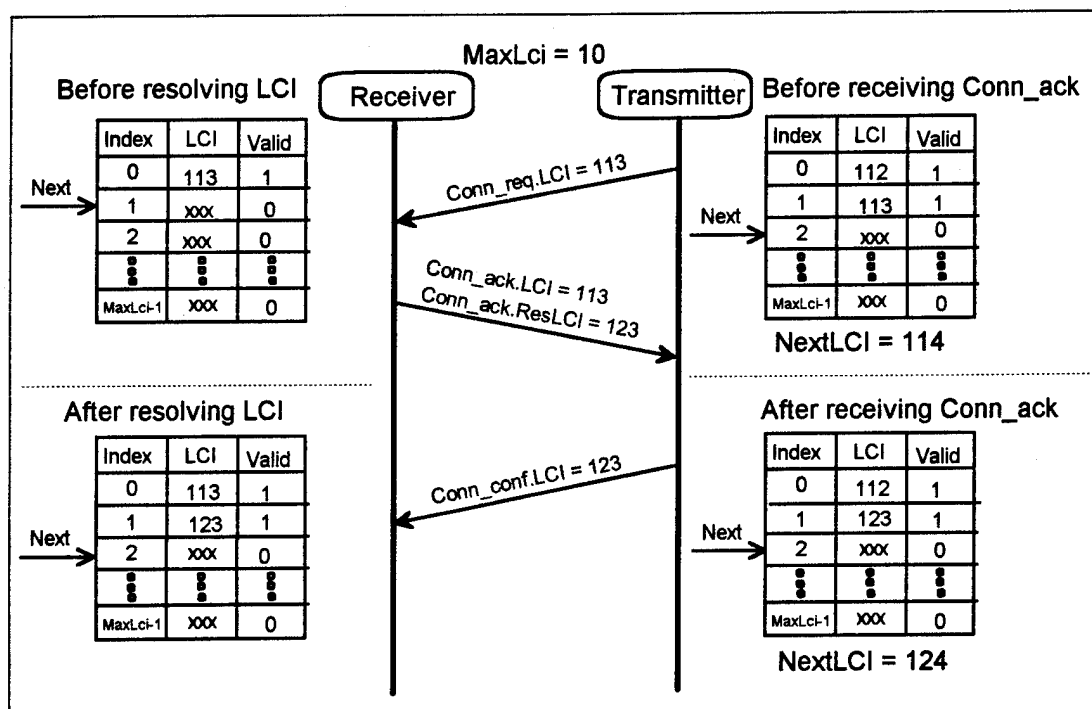
Figure 5.1 - Conflict LCI Resolving Protocol

---

[18]   The word 'larger' is in a cyclic manner. i.e. if the maximum LCI is reached, 0 or 1 may be used. With an unsigned character field for the LCI in this implementation, 255 LCI's could be used repeatedly. This is sufficient for either the Transmitter or the Receiver to avoid second LCI conflict using the rules provided. It should note that the resolve LCI chose by the Receiver need not and better not be larger than all the LCI's that appear in the Receiver LCI table. As long as the LCI is larger than the LCI in the *Conn_req* packet by *MaxLci* and the LCI is not appeared in the LCI table, then this is a qualified resolve LCI.

One might ask "If the Receiver receives two hosts sending *Conn_req* with the same LCI at the same time? Wouldn't they both be resolved by using the same LCI on the Receiver side?" It should note that in a non multiprocessing system, no matter how close the two requests are, there is always one request can be processed before the other. Thus for this situation, the first conflict LCI would be resolved as the requested LCI + *MaxLci*, the second LCI would cause a conflict and by the rule that an LCI should be larger than the requested LCI by **at least** *MaxLci*, the requested LCI + *MaxLci* + 1 (if there was no LCI in the Receiver LCI table take on this value) would be chose for the second connection. These LCI's are all unique in the Receiver LCI table. Figure 5.2 illustrates the resolving for this case that would be referred to as "contention on the Receiver side."
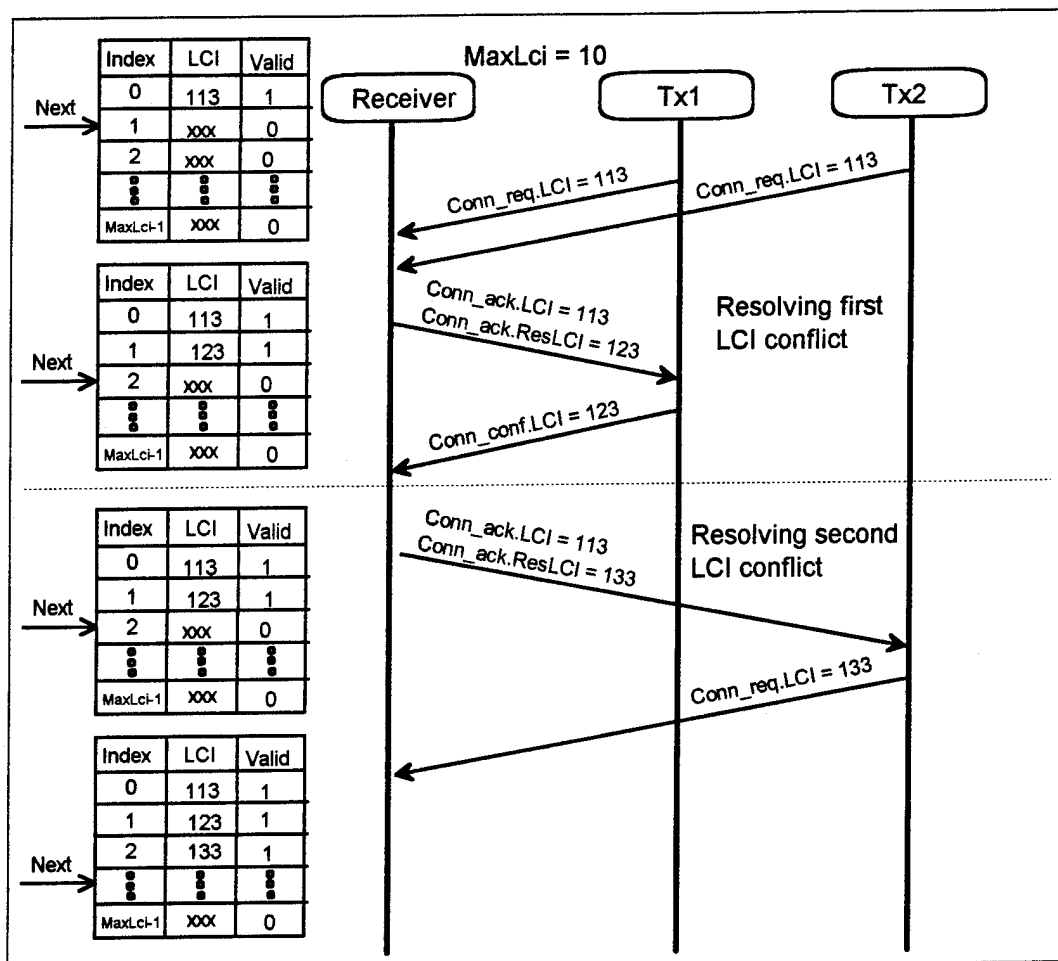
Figure 5.2 - Conflict LCI Resolving Protocol (Contention On the Receiver Side)

Others might ask "What if the Transmitter after sending the *Conn_req* for a connection that caused an LCI conflict, before the *Conn_ack* that carries the resolved LCI is received, another *Conn_req* with an incremented LCI is sent for another connection by the same Transmitter. Wouldn't the resolved LCI be conflict with the LCI that is used for the second connection on the Transmitter side?" The answer is that the Receiver chose an LCI that is larger than the requested LCI by at least *MaxLci*. The Transmitter should not get any LCI conflict if it increased the LCI for the second connection by **1**. The timing diagram that illustrates the resolving of this case is given in Figure 5.3 and would be referred to as "contention on the Transmitter side."
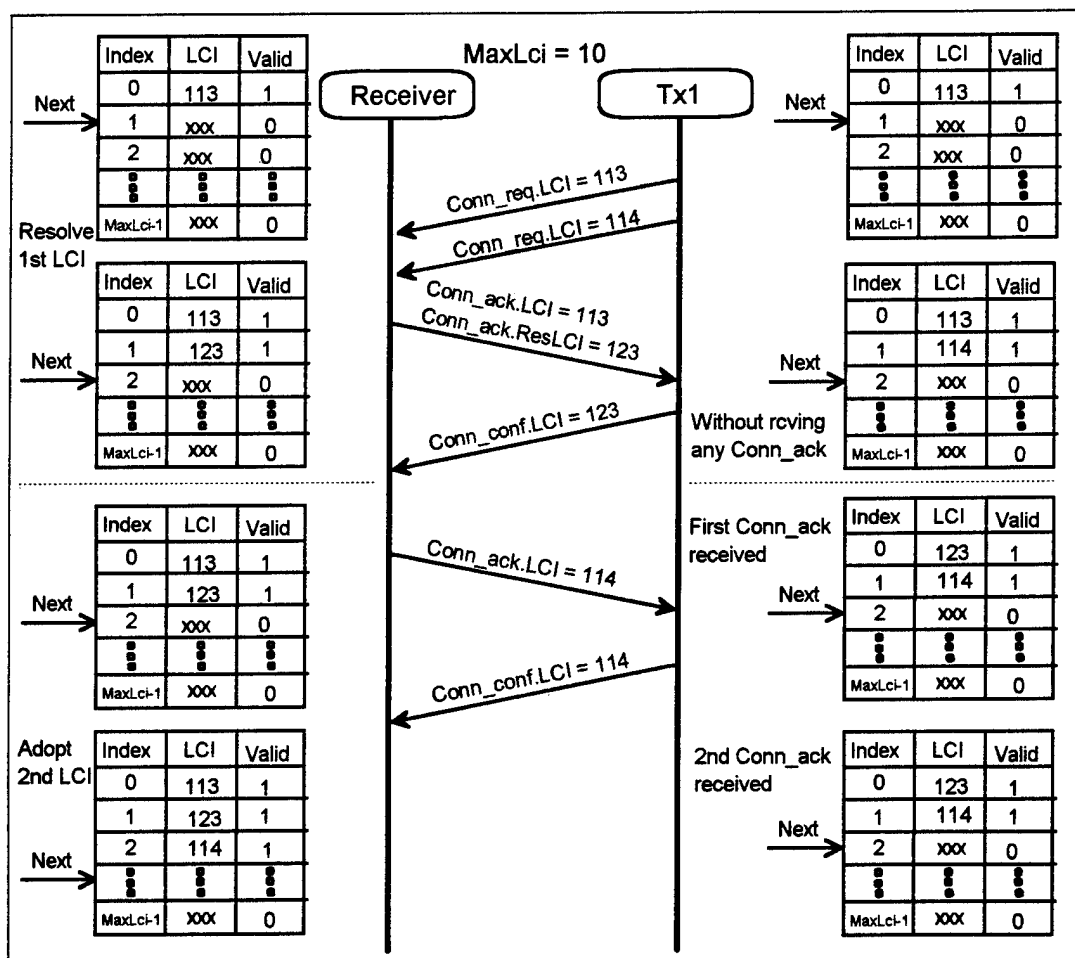
Figure 5.3 - Conflict LCI Resolving Protocol (Contention On the Transmitter Side)

43

2. Unsettled Negotiation Parameters

In the specification, connection can be established when machine R2, after reply the Transmitter with *Conn_ack*, receives *Conn_conf*, *T_state* or *Data* packet. If the connection is established by the receiving of *Conn_conf*, the negotiation parameters could be retrieved from this packet and be adopted for this connection, so nothing would go wrong. But if the connection is established by the receiving of *T_state* packet or *Data* packet, there would be no way for the Receiver to get the negotiation parameters for the connection. This would cause problem since the Receiver side and the Transmitter side may use different parameters for communication. One solution could be using the parameters provided in *Conn_req*, but what if the requested parameters are out of the Receiver's capability? Other solution could be change the specification to have the connection could only be established by the receiving of *Conn_conf* packet. The later is more conservative thus more secure but what should be done if some *Data* packets arrives before the Receiver receives the *Conn_conf* packet due to the network routing? This is an issue that requires further study.

VI. SUMMARY

In this thesis the implementation of the SNR high speed network communication protocol has been described. This thesis is concerned primarily with the receiver part of the protocol; the transmitter part was implemented and described in another thesis [Ref. 2].

Much work has been done since the initial design of the SNR high speed network communication protocol [Ref. 3]. The formalizing of the specification [Ref. 1], analyzing and simulation of the protocol were accomplished prior to this thesis. There is no doubt about the careful consideration of the specification for this protocol in the previous work. However, an implementation is a very detailed work that usually leads to some unexpected problems.

The problems encountered during the implementation include:

- the signal loss problems;

- the shared memory key base conflict problems;

- the process synchronization problems;

- the scheduling problems;

- the C compiler data alignment problems;

and some problems that were caused by unfamiliarity of the UNIX system calls and the UNIX system environments. Fortunately, these problems could be solved in a way that does not affect the features of this protocol.

Some changes to the specification were made to correct errors that were found during the implementation or to make the specification more practical for implementation. These changes include:

- additional field added to the SNR packet header;

- additional packet types formats defined for some insufficiently specified packets;

- internal states used to better describe some states;

- corrections for incorrect predicate conditions;

- initialization of data structures specified in the corresponding state;

- timer service request and cancellation specified in the corresponding states;

- the solutions for signal lost and processes contention problems; and

45

• out-of-band packets used for practical implementation.

A system schematic diagram was drawn to give a high level description of this implementation. Categorized program structures of protocol machines were described as well.

The testing of this implementation incorporated with the Transmitter implementation was described. These tests include:

• using different modes for connections;

• lost packets;

• duplicate packets;

• large data transfer; etc.

Though this is still not a fully mature implementation and the testing network environment was not as complicated as a production environment, tuning was still performed on this implementation to gain some experiences of how the implementation would respond to the change of these tuning parameters. Goals of tuning were set up as follows:

• to ensure the connection could be established as long as the Receiver is activated and the network is working;

• to minimize the unnecessary retransmissions;

• to have *R_state* control packets being sent in a sufficient manner; and

• to have timeout disconnection initiated only when problem really occurred on the connection.

In order to know which work has been done and which work has not been sufficiently done in this implementation, evaluation of this implementation was given. The issues include:

• maintainability;

• portability;

• daemon process;

• event driven;

• negotiation of parameters;

• graceful termination; and

• further extensions.

For further extensions of this implementation, the following issues were addressed:

- multiple logical connections;

- entity-to-entity communication (also known as multiplexing);

- application program interface (API) design; and

- port onto other operating systems.

Possible solutions were provided to the uncleared issues that were not included in the specification and have appeared in the related publications. The issues include:

- logical connection identifier conflict resolving; and

- unsettled negotiation parameters.

In this thesis, some issues that may sound trivial or wordy but were still included for the completeness and to retain the many experiences that were gained in the process of developing this implementation as possible.

By the implementation and testing result of the core of this protocol, the functionality of this protocol has been proven as it was specified. There is still much work that needs to be done to really make this protocol an alternative to the existing transport protocols. However, if the studies go on, the goal will eventually be reached.

APPENDIX

The main programs and functions mentioned in this paper are summarized here to avoid interrupting the flow of the body text. There are totally eight programs that have been designed for this implementation. They are snr_r.c, snr_r1.c, snr_r2.c, snr_r3.c, snr_r4.c, snr_tc.c, snr_que.c and snr_util.c. Five header files were defined for the implementation as well. They are snr_env.h, snr_tr.h, snr_r.h, snr_que.h and snr_util.h. One program sem.c for semaphore operations was referred to [Ref. 4]. Not all details are shown here. Some functions may have only their prototye shown and few lines of description or even completely omitted if the processing is plain.

A.   snr_r.c

The Receiver root process.

1. Main program

```
void
main() {
    Create shared memory;
    Attach to shared memory by the pointer Shm;
    Create semaphores for INBUF, scount, received and T_CHAN;
    Initialize shared memory data structures;

    /* Create raw (IP layer) socket for SNR protocol */
    Shm->Tsock = socket(AF_INET, SOCK_RAW, PROTO_SNR);

    /* Turn the Receiver main power on */
    Shm->SNR_ON = True;

    /* Activates each processes */
    Shm->Pid[1] = Activate(SNR_R1);
    Shm->Pid[2] = Activate(SNR_R2);
    Shm->Pid[3] = Activate(SNR_R3);
    Shm->Pid[4] = Activate(SNR_R4);
    Shm->Pid[5] = Activate(SNR_TC);

    Wait for system administrator to terminate the protocol;

    /* turn the Receiver main power off */
    Shm->SNR_ON = False;
```

49

```
/* Inform each processes about the termination and wait for them to terminate */
for (i = R1; i <= TC; i++) { /* inform all machines about termination */
    notify(i);
    wait for the machine to terminate;
}

Detach shared memory;
Remove shared memory;
Close all semohpores;
} /* SNR_R */
```

B.   snr_r1.c

The machine R1 of the Receiver.

1. Main program

```
void
main() { /* SNR_R1 */
    int   STATE = 0;
    flag TxCompleted = False;
    Attach to shared memory by the pointer Shm;
    Open Semaphores for INBUF,  received and T_CHAN;
    while (Shm->SNR_ON) {
        WaitForEvent();
        switch (STATE) {
        case 0:
            if (Shm->R_active) /* start */
              Initialization();
              STATE = 1;
            break;
        case 1:
            ProcTchanPkts();
            if (!Shm->R_active) { /* finish */
              if (!Shm->Disconnect) { /* Tx completed, retrieve all data packets */
                ProcTchanPkts();
                TxCompleted = True;
              }
              STATE = 0;
            } /* if finish */
            if ((!Shm->Disconnect) && (Shm->recieved || TxCompleted))
              Notify(R4); /* inform R4 data received or Tx Completed */
            break;
        } /* switch */
```

```
        } /* while */
    Detach shared memory;
    Close all semohpores;
} /* SNR_R1 */


2. Functions

    a. Initialization
        void
        Initialization() {
            /* Initialize RECEIVE, AREC, LOB, INBUF */
            for (i = 0; i < RCVsize; i++)
                Shm->RECEIVE[i] = 0;
            for (i = 0; i < ARECsize; i++)
                Shm->AREC[i] = 0;
            for (i = 0; i < LOBsize; i++)
                Shm->LOB[i] = 0;
            for (i = 0; i < INBUFsize; i++)
                Shm->INBUF[i].datalen = 0;
            Shm->INBUFHead = InitSeqNr;
            Shm->LWr = InitSeqNr;
            Shm->UWr = LWr + NegoWinSize - 1;
            Mode = Shm->NegoMode;
            BlkSize = Shm->NegoBlkSize;
            BufSizeInBlk = INBUFsize / BlkSize;
            TxCompleted = False;
        } /* Initialization */


    b. UpdLWrUWr
        void
        UpdLWrUWr() {
            pos = NDX(Shm->LWr, BufSizeInBlk);
            while (BitIsSet(Shm->AREC, pos) && (Shm->LWr < Shm->UWr)) {
                ClrBit(Shm->AREC, pos);
                pos = NDX(++Shm->LWr, BufSizeInBlk);
            }
            Shm->UWr = Shm->LWr + Shm->NegoWinSize - 1;
        } /* UpdLWrUWr */


    c. OrderInsert
        flag
        OrderInsert(SNRpkt_t *pkt) {
            BlkNr = pkt.seq.BlkNo;
            PktNr = pkt.seq.PktNo;
```

```
SeqNr = (BlkNr - InitSeqNr) * BlkSize + PktNr;
INBUFndx = NDX(SeqNr, INBUFsize);
if (BlkNr < Shm->LWr || (BlkNr > Shm->UWr)
                        || BitIsSet(Shm->RECEIVE, INBUFndx) {
    /* duplicated or out of flow control packet */
    return True;
}else { /* process packet */
    Shm->INBUF[INBUFndx].datalen = pkt->hdr.datalen;
    bcopy(pkt->info.Dat.data, Shm->INBUF[INBUFndx].data, pkt->hdr.datalen);
    SetBit(Shm->RECEIVE, INBUFndx);
    Shm->UWr = max(Shm->UWr, BlkNr);
      Shm->BufAvail--;
    if (BlkAllSet(SeqNr)) {
        SetBit(Shm->AREC, NDX(BlkNr, BlksPerBuf)); /* Update AREC */
        if (BlkNr = Shm->LWr) /* Update LWr, UWr */
          UpdLWrUWr();
        BitsCopy(Shm->AREC, Shm->LOB, Shm->LWr,
                    Shm->NegoWinSize, Shm->L, BlksPerBuf); /* Update LOB */
    } /* if BlkAllSet */
} /* else process packet */
return False;
} /* OrderInsert */


d. ProcTchanPkts
   void
   ProcTchanPkts() {
     PktType = PeekQue(T_chan);
     while (PktType != INVALID) { /* Not Empty */
         Pkt = DeQue(&T_CHAN);
         if (PktType == Data) { /* receive */
           Duplicate = False;
           if (Mode = 0) /* no buffer */
             PassDataToHost(Pkt);
           else {
             if (Mode == 1 || Mode == 2) { /* buffer */
                 duplicate = OrderInsert(&Pkt);
             } /* if buffer */
           } /* else */
           if (!duplicate) {
               Shm->received = True;
           } /* if duplicate */
         } else
           if (PktType == Conn_disc) /* remember to notify R2 */
             Shm->Conn_discFlag = True;
```

```
                    PktType = PeekQue(&T_CHAN);
            } /* while */
            if (Shm->Conn_discFlag)
                Notify(R2);
        } /* ProcTchanPkts */
```

e. *Interfacing functions*

The following functions are defined for proofing the implementation, they all need to be redefined when developing the interface between the SNR protocol and the applications.

- void PassDataToHost();
  Stores the retrieved packet data into an out buffer for display.
- void ShowOutBuf();
  Displays the out buffer for testing purpose.

C.   **snr_r2.c**

The machine R2 of the Receiver.

1. Main program

```
void
main() { /* SNR_R2 */
    int  STATE = 0;
    Attach to shared memory by the pointer Shm;
    Open Semaphores for scount and T_CHAN;
    while (Shm->SNR_ON) {
        WaitForEvent();
        switch (STATE) {
        case 0:
            while ((PktType = PeekQue(&T_CHAN) != Conn_req ) &&
                    (!Empty(&T_CHAN))
                DeQue(&T_CHAN);  /* skip to the first Conn_req pkt */
            if (PktType == Conn_req ) { /* ack for conn_req */
                conn_req = DeQue(&T_CHAN);
                Evaluate(&conn_req); /* evaluates the parameters and construct conn_ack */
                SendPkt(&conn_ack); /* send the acknoledge packet over R_CHAN */
                TimerIsOn = ReqTimerSrv(IPT); /* request timer service for state 1 */
                Initialization();
                STATE = 1;
            }
            break;
        case 1:
            PktType = GetPktType();
            if (PktType == INVALID) { /* Empty(T_CHAN) */
                if (TimerIsOn) {
                    delay++;
                    if (delay < reset) /* ok */
```

53

```
                SendPkt(&conn_ack);
            else { /* time out */
              TimerIsOn = CancelTimerSrv();
              STATE = 0;
            }
          } /* if timer is on */
        } else { /* T_CHAN is not empty */
          if ((PktType == Conn_conf) ||
              (PktType == T_state) ||
              (PktType == Data)) { /* start */
            if (PktType == Conn_conf) { /* retrieve negotiated parameters */
              conn_conf = DeQue(&T_CHAN);
              RetrieveNegoPara(&conn_conf); /* added */
            }
            TimerIsOn = CancelTimerSrv();
            STATE = 2;
            Shm->R_active = True;
            /* inform R1, R3 and R4 R_active = True */
            Notify(R1);
            Notify(R3);
            Notify(R4);
          } else /* not start */
            if (PktType == Conn_req) { /* lost ack */
              DeQue(&T_CHAN);
              SendPkt(Conn_ack);
            } /* if lost ack */
        } /* else T_CHAN is not empty */
        break;
      case 2:
        PktType = GetPktType();
        if (PktType == T_state ) {
          SeqNr = Shm->T_statePkt.SeqNr;
          if (SeqNr > high) { /* update */
            high = SeqNr;
          } else { /* no update */
            if ((PktType == Conn_conf) || (PktType == Conn_req)) { /* discard */
              DeQue(&T_CHAN);
            else
              if ((Shm->Disconnect) || (PktType == Conn_disc)) { /* finish */
                Shm->R_active = False;
                STATE = 0;
                /* inform R1, R3 and R4 R_active = False */
                Notify(R1);;
                Notify(R3);
```

54

```
                    Notify(R4);
                } /* if finish */
            } /* else no update */
            break;
        } /* switch */
    } /* while */
    Detach shared memory;
    Close all semohpores;
} /* SNR_R2 */
```

2. Functions

a. *Initialization*

```
void
Initialization() {
    high = 0;
    delay = 0;
    Shm->T_stateFlag = False;
    Shm->Conn_discFlag = False;
} /* Initialization */
```

b. *GetPktType*

```
int
GetPktType() {
    if (Shm->Conn_discFlag) {
        Shm->Conn_discFlag = False;
        return(Conn_disc);
    } else
        if (Shm->T_stateFlag) {
            Shm->T_stateFlag = False;
            return(T_state);
        } else
            return(PeekQue(&T_CHAN));
} /* GetPktType */
```

c. *Evaluate*

Prototype: void Evaluate(SNRpkt_t *conn_req);
This function evaluates the parameters retrieved from the packet Conn_conf sent by Transmitter to determine the paremeters to be used in Conn_ack that will be sent back to the Transmitter.

d. *SendPkt*

Prototye: void SendPkt(SNRpkt_t *pktP);
This functin sends the packet over the R_CHAN.

D.   snr_r3.c

The machine R3 of the Receiver.

55

1. Main program

```
void
main() {/*  SNR_R3 */
    int   STATE = 0;
    Attach to shared memory by the pointer Shm;
    Open Semaphores for scount and received;
    while (Shm->SNR_ON) {
        WaitForEvent();
        switch (STATE) {
        case 0:
            if (Shm->R_active) /* start */
              TimerIsOn = ReqTimerSrv(Tin); /* request timer service for state 1 */
              Initialization();
              STATE = 1;
           break;
        case 1:
            if (!Shm->R_active) { /* finish */
              TimerIsOn = CancelTimerSrv();
              STATE = 0;
           }else
             if (TimerIsOn) { /* clock */
               Shm->scount++;
               State4 = True; /* need to enter internal state 4 later */
               if (!Shm->received) { /* no_data */
                 count++;
                 if (count >= k) || (Shm->scount >= Lim) { /* timeout */
                   BuildRstate();
                   SendPkt(Rstate);
                   k = min(2 * k, Klim); /* expand Tin */
                 } else /* delay */
                   State4 = False; /* doesn't need to enter internal state 4 later */
               } else { /* data */
                 BuildRstate();
                 SendPkt(Rstate); /* send R_state to ack the packet received */
               } /* else data */
               if (State4) { /* internal STATE 4 */
                 if (Shm->scount < Klim) { /* no disc */
                   Shm->received = False;
                   count = 0;
                 } else { /* disc */
                   TimerIsOn = CancelTimerSrv();
                   Shm->Disconnect = True;
                   State = 5;
```

56

```
                    /* inform R2 and R4 Disconnect = True */
                    Notify(R2);
                    Notify(R4);
                } /* else disc */
              } /* if STATE 4 */
            } /* if clock */
          break;
        case 5:
          if (!Shm->R_active) /* confirm */
            STATE = 0;
          break;
        } /* switch */
      } /* while */
    Detach shared memory;
    Close all semohpores;
} /* SNR_R3 */
```

2. Functions

a. *Initialization*

```
void
Initialization() {
    count = 0;
    CurrBlkNr = InitSeqNr;
    CurrPktNr = InitSeqNr;
    Shm->scount = 0;
    Shm->k = InitK;
} /* Initialization */
```

b. *BuildR_state*

Prototype: void BuildR_state();

This function constructs the R_state packet using the informations available inside the Receiver.

c. *SendPkt*

See SNR_R2 function SendPkt.

E.   snr_r4.c

The machine R4 of the Receiver.

1. Main program

```
void
main() {/* SNR_R4*/
```

```
int STATE = 0;
Attach to shared memory by the pointer Shm;
Open Semaphore for INBUF;
while (Shm->SNR_ON) {
    WaitForEvent();
    switch (STATE) {
    case 0:
        if ((Shm->R_active) && (mode == 1 || mode == 2)) { /* start */
          NotifyHostAboutConnection(mode);
          Initialization();
          STATE = 1;
        break;
    case 1:
        if (Shm->Disconnect) { /* disc */
          NotifyHostAboutDisconnection();
          STATE = 0;
        } else { /* not disc */
           if (!Empty(INBUF) && SignalFromHost) { /* accept */
             /* State2 */
             if (mode == 2) { /* err chk */
               RetrieveMode2();
             } else /* no err */
               /* State3 */
               if (WaitBulk(wait_lim)) /* retrieve */
                 RetrieveMode1();
               } /* if retrieve */
               /* State1 */
             } /* else no err */
           } /* if accept */
           if (!Shm->R_active) { /* finish */
             NotifyHostAboutCompletion();
             STATE = 0;
           } /* if finish */
        } /* else not disc */
        break;
    } /* switch */
  } /* while */
  Detach shared memory;
  Close all semohpores;
} /* SNR_R4 */
```

2. Functions

a. Initialization

58

```
Initialization() {
    for (i = 0; i < OUTBUFsize; i++)
        *(OutBuf + i) = 0;
    TotalDataLen = 0;
    BlkSize = Shm->NegoBlkSize;
    BufSizeINBlk = INBUFsize / BlkSize;
    WinSizeInPkt = Shm->NegoWinSize * BlkSize;
    wait_lim = BlkSize * AcceptableRatio;
} /* Initialization */
```

b. *WaitBulk*

```
WaitBulk(int BulkCnt) {
    return(!BitIsSet(Shm->RECEIVE, Head) &&
            (NrBitSet(Shm->RECEIVE, NDX(*Head + 1, INBUFsize),
                    BSTART(*Head, BlkSize, INBUFsize)) < BulkCnt));
} /* WaitBulk */
```

c. *RetrieveMode1*

```
RetrieveMode1() {
    BlkNr = BLKNDX(Head, BlkSize, BufSizeInBlk);
    Skip = False;
    if (!BitIsSet(Shm->RECEIVE, Head) { /* skip this packet */
        SetBit(Shm->RECEIVE, Head);
        Skip = True;
    }
    while (BitIsSet(Shm->RECEIVE, Head)) {
        if (!Skip)
            PassDataToHost();
        else {
            Skip = False;
            if (BlkAllSet(Heae)) { /* promote the stucked LWr */
                SetBit(Shm->AREC, NDX(BlkNr, BlksPerBuf));
                if (BlkNr = Shm->LWr)
                    UpdLWrUWr();
                /* update LOB */
                            BitsCopy(Shm->AREC,  Shm->LOB,  Shm->LWr,
Shm->NegoWinSize,
                    Shm->L, BlksPerBuf);
                Shm->Buffer_avail--;
            } /* if BlkAllSet */
        } /* else */
        if (BlkCompleted()) {
            Shm->Buffer_avail++;
            ClrBlk(Shm->RECEIVE, BlkNr, BufSizeInBlk);
        } /* if BlkCompleted */
        INC(Head, INBUFsize);
```

```
            BlkNr = BLKNDX(Head, BlkSize, BufSizeInBlk);
        } /* while */
    } /* RetrieveMode1 */
```

d. *RetrieveMode2*

```
    RetrieveMode2() {
        BlkNr = BLKNDX(Head, BlkSize, BufSizeInBlk);
        while (BitIsSet(Shm->RECEIVE, Head)) {
            PassDataToHost();
            if (BlkCompleted()) {
                Shm->Buffer_avail++;
                ClrBlk(Shm->RECEIVE, BlkNr, BufSizeInBlk);
            } /* if BlkCompleted */
            INC(Head, INBUFsize);
            BlkNr = BLKNDX(Head, BlkSize, BufSizeInBlk);
        } /* while */
    } /* RetrieveMode2 */
```

e. *UpdLWrUWr*

See SNR_R1 function UpdLWrUWr.

F.   snr_tc.c

The Transmitter channel.

1. Main program

```
main() {/*  SNR_TC/
    Attach to shared memory by the pointer Shm;
    Open semaphore for  T_CHAN;
    while (Shm->SNR_ON) {
        len = recvfrom(Shm->Tsock, IPpkt); /* get IP packet from T_CHAN */
        PktType = ExtractPkt(IPpkt, len, &SNRpkt); /* extract SNR pkt from IP pkt */
        switch (PktType) { /* notify relevant machines */
        case T_state:
            if (CheckSumOk) {
                Shm->TstatePkt = SNRpkt; /* latest T_state packet */
                Shm->TstateFlag = True;
            }
            Notify(R2);
            break;
        case Data:
            if (CheckSumOk) {
                EnQue(&T_CHAN, &SNRpkt);
                if (!Shm->R_active)
                    Notify(R2);
                else /* inform R1 */
                    notify(R1);
```

60

```
            }
            break;
         case Conn_req:
            EnQue(&T_CHAN, &SNRpkt);
            notify(R2);
            break;
         case Conn_conf:
            EnQue(&T_CHAN, &SNRpkt);
            Notify(R2);
            break;
         case Conn_disc:
            EnQue(&T_CHAN, &SNRpkt);
            Notify(R1); /* R1 will notify R2 about Conn_disc */
            break;
         case R_state:     /* discard self send packets */
            break;
         case Conn_ack: /* discard self send packets */
            break;
      } /* switch */
   } /* while */

   Detach shared memory;
   Close semohpore;
}/*  SNR_TC */
```

2. Function

a. ExtractPkt

```
u_char
ExtractPkt(u_char *IPpkt, int pktlen, SNRpkt_t *pktP) {
    pktlen = pktlen - IPhdrLenInBytes; /* get SNR packet length */
    snrP = (SNRpkt_t *) (IPpkt + IPhdrLenInBytes); /* points to SNR packet */
    pktP->hdr = snrP->hdr; /* extract header information */
    InfoLen = pktlen - sizeof(SNRhdr_t);
    bcopy(snrP->info.Dat.data, pktP->info.Dat.data, InfoLen);
    ErrChkP = (int *) ((u_char *) snrP->info.Dat.data + InfoLen - ErrChkSize);
    ChkSumOk = (ChkSum(u_short *) snrP, pktlen - ErrChkSize) == *ErrChkP);
    return(pktP->hdr.type);
} /* ExtractPkt */
```

G.   snr_que.c
This program is a collection of packet queue operation functions.
1. Data structure
```
typedef struct {
    SNRpkt_t          que[PktsPerBuf];
```

```
        int                 front,
                            tail;
    flag                    full;
} PktQ_t;
```
By defining the PktQ_t type, the T_CHAN can be declared in the program as

```
PktQ_t              T_CHAN;
```

and the address of T_CHAN (i.e. &T_CHAN) could be passed as a parameter for every queue operation function since all the information that required to operate the packet queue has been included inside the PktQ_t data structure already. This means we need only one queue operation program that could operate on many different queues of the PktQ_t type. It would not be very helpful to make this kind of design if there is only one queue in the whole implementation. But note that it might become necessary to have many T_CHANs one for each connection when improving this implementation in the future. Another benifit of using this design is to make the program looks more like the specification (e.g. EnQue(&T_CHAN) etc.) and this is an effort that author has being put through out the whole implementation.

2. Functions

a. EnQue

void EnQue(SNRpkt *pktP, PktQ_t *Pq);

This function put the SNR packet in the packet queue.

e.g. EnQue(&T_CHAN, conn_ack);

b. DeQue

SNRpkt_t DeQue(PktQ_t *Pq);

This function gets a SNR packet from the front of the packet queue.

e.g. conn_req = DeQue(&T_CHAN);

c. PeekQue

u_char PeekQue(PktQ_t *Pq);

This function checks if the queue is empty, if not, it returns the packet type of the packet at the front of the queue. If the queue is empty, it returns INVALID.

e.g. PktType = PeekQue(&T_CHAN);

d. InitQue

void InitQue(PktQ_t *Pq);

This function initializes the queue.

e.g. InitQue(&T_CHAN);

e. Empty

flag Empty(PktQ_t *Pq);

This function returns True if the queue is empty.

e.g. if (Empty(&T_CHAN)) ...

H.   snr_util.c

This program is a collection of general utility functions. These functions include bit operations, timer and check sum.

1. Bit operation functions

Due to the fact that ths SNR protocol starts the sequence number from one, the bit operation functions that need to access to some arrays (e.g. INBUF, AREC etc.) using the

sequence number has to subtract one before indexing to the array in order to access the array from the first element (e.g. INBUF[0]). These offset problem has been taken care of by all the bit operation functions in this program so the array index arguments need not to be subtracted before the call.

    a. SetBit

        void SetBit(u_char *BITS, int BitPos);

        This function sets the bit in BITS at BitPos to 1. (Note that the BITS is an array of unsigned characters that has finite length, this function doesn't do any boundary checking but leave this to the user's responsibility. Also note that the minimum BitPos is 1 rather than 0).

        e.g. SetBit(AREC, BlkNr);

    b. ClrBit

        void ClrBit(u_char *BITS, int BitPos);

        This function clears the bit in BITS at BitPos to 0.

        e.g. ClrBit(RECIEVE, Head);

    c. BitIsSet

        flag BitIsSet(u_char *BITS, int BitPos);

        This function checks if the bit in BITS at BitPos is 1.

        e.g. if (BitIsSet(RECEIVE, SeqNr)) ...

    d. NrBitsSet

        int NrBitsSet(u_char *BITS, int start, int end);

        This function counts how many bits in BITS from start to end is 1. (Note that this function would wrap arround when the bit position exceeds the array boundary. This is done for circular buffers in the SNR protocol. e.g. RECEIVE)

        e.g. EmptyINBUF = !NrBitsSet(RECEIVE, Head, INBUFsize, BlkStart);

    e. ClrBlk

        void ClrBlk(u_char *BITS, int BlkNo, int BlkSize);

        This function clears BlkSize number of bits at the positions specified by BlkNo in BITS to 0.

        e.g. ClrBlk(RECEIVE, BlkNr, BlkSize);

    f. BitsCopy

        void BitsCopy(u_char *SRC, u_char *DST, int start,
                    int CpySize, int DSTsize, int SRCsize);

        This function copies CpySize of bits in SRC started from the start position to DST started from the first element. If the CpySize is smaller than the DSTsize, the rest bits of DST would be set to 0. The wrap arround of SRC is taken care of in this function by using SRCsize.

        e.g. BitsCopy(AREC, LOB, LWr, NegoWinSize, L, ARECsize);

2. Timer functions

    a. ReqTimerSrv

        flag ReqTimerSrv(int tick);

        This function requested a periodic timer that will send signal SIGALRM to the process every tick microseconds. It always returns True.

        e.g. TimerIsOn = ReqTimerSrv(Tin);

63

b. *ReqTimerSrv*
       flag CancelTimerSrv(int tick);
       This functin cancels the timer service. It always returns False.
       e.g. TimerIsOn = CancelTimerSrv();

3. Check sum

This function does the check sum computation. (Refers to [Ref. 4] pp. 455).

   a. *ChkSum*
      int ChkSum(u_short *ptr, int nbytes);
      This function computes the check sum of the data structure pointed by ptr for nbytes. In this implementation, the check sum is computed for the received SNR packets from header through all the data field exclude the error check field. Note that since the error check field should always immediately following the last data byte in the packet and the data length of a data packet varies from connection to connection, the error check field has to be determined at runtime by using a pointer.
      e.g. ChkSumOk = ChkSum((u_short *) snrP, pktlen - ErrChkSize) == *ErrChkP);

I.   snr_env.h

This header file defines macros that cover up the differences exist between two target UNIX system (i.e. SOLARIS and IRIX) environments. By using this file, no macro needs to appear in the C programs for different environments thus increases the programs readability and maintainability. Some macros were defined for the preferences of the author as well.

J.   snr_tr.h

This program collects all the definitions that should be common to both the Transmitter and the Receiver. The definitions are categorized as following:

1. IP Definitions

   a. *PROTO_SNR - The IP protocol number for SNR protocol.*

   b. *IPhdrLen - IP header length that used to computes the maximum SNR data length.*

2. SNR Implementation Limits

   a. *MaxLci - the maximum logical connection identifier allowed.*

   b. *MaxPktSize - the SNR maximum packet length.*

   c. *MaxWinSiez - the maximum window size (in blocks).*

3. Definition for SNR packet types

   a. *#define   R_state   0*

   b. *#define   T_state   1*

c. #define   Data   2

d. #define   Conn_Req   3

e. #define   Conn_Ack   4

f. #define   Conn_Conf 5

g. #define   Conn_Disc 6

4. Definitions for Shared Memory Key Base Values

a. #define   TxShmKBase 7890

b. #define   RxShmKBase 8890

5. Definitions for SNR packets

The following SNR packet type structures were defined.

a. SNRhdr_t - the SNR header type.

b. SNRdata_t - the SNR data packet type.

c. SNRrstate_t - the SNR R_state packet type.

d. SNRtstate_t - the SNR T_state packet type.

e. SNRconn_t - the SNR connection packet type. Which includes Conn_req, Conn_ack, Conn_conf and Conn_disc packet types.

f. SNRinfo_t - the union of all SNR packet types (i.e. SNRdata_t, SNRrstate_t, SNRtstate_t and SNRconn_t) . This makes the receiving buffer could be uniformly retrieved as a general SNR packet type and later be used as different packet types without using casting.

g. SNRpkt_t - the general SNR packet type. Which is constructed by SNRhdr_t and SNRinfo_t.

6. Definition for INBUF data structure

a. BUFdata_t - The element structure of INBUF array. Consist of data length and the data.

K.   **snr_r.h**

This file defines macros or data structures common to all the Receiver machines. There is actually one data structure defined in this file - SNRshmRec, for shared memory.

1. **Receiver machine definitions**

   a. #define   R1   1

   b. #define   R2   2

   c. #define   R3   3

   d. #define   R4   4

   e. #define   TC   5

   f. #define   Nprocesses   6

2. **Tuning parameters for Receiver**

   These parameters are collected to make the tuning easier.

   a. *kou - the constant for compute Tin, used by R2.*

   b. *TOtick - timeout counts for reset, used by R3.*

   c. *Klim - the limit for count, used by R3.*

   d. *Lim - the limit for scount, used by R3.*

   e. *InitIPT - the IPT value in micro seconds, used by R2.*

   f. *AcceptableRatio - the ratio that used to determine the wait_lim, used by R1 in mode 1.*

3. **Unspecified initial values**

   a. *InitMode - best mode that can be provided by the Rx, used by Receiver Root.*

   b. *InitK - the initial k value, used by R3.*

4. **Shared memory key definition**

   a. *SHMKEY - the shared memory key base value*

5. **Semaphore key definition**

   a. *SEMKEY1 - the semaphoe key value, used by INBUF semaphore*

   b. *SEMKEY2 - the semaphoe key value, used by SCOUNT semaphore*

   c. *SEMKEY3 - the semaphoe key value, used by RECEIVED semaphore*

   d. *SEMKEY4 - the semaphoe key value, used by T_CHAN semaphore*

   e. *PERMS - shared memory creation option*

6. Shared memory record - SNRshmRec

All the shared memory variables or data structures are included in this record.

a. Transmitter channel - shared by T_CHAN, R1, R2.

    PktQ_t   T_CHAN;

b. Input buffer - shared by R1 and R4.

    BUFdata_t   INBUF[INBUFsize + InitSeqNr];

    int   INBUFHead;

c. Output buffer - shared by R1, R4 and supposingly the host.

    u_char   OUTBUF[OUTBUFsize];

d. Out-of-band packets - shared by T_CHAN and R2.

    SNRpkt_t   TstatePkt;

    flag   T_statePkt;

    flag   Conn_discFlag;

e. Transmitter channel socket id - shared by T_CHAN, R1, R2 and R3.

    int   Tsock;

    struct sockaddr_in Xmtr;

f. Logical connection identifier - not implemented in this implementation.

Supposingly these variables should be shared by all processes when multiplexing is implemented.

    int   LCI[MaxLci];

    int   CurrLciNdx;

g. Receiving states - shared by R1, R2 and R3.

    u_char   RECEIVE[RCVsize];

    u_char   AREC[ARECsize];

    u_char   LOB[LOBsize];

h. Process ID - shared by T_CHAN, R1, R2, R3 and R4 for sending signals.

    int   Pid[Nprocesses];

i. Inter-process control flags - shared by R1, R2, R3 and R4.

    flag   R_active;

```
flag            received;

flag            Disconnect;
```

j. Mescellaneous variables

```
int             mode;

int             peakBW;

int             LWr;

int             UWr;

int             Buffer_avail;

int             scount;

int             L;

int             k;

int             Tin;

int             IPT;
```

k. Negotiated parameters - shared by R1, R2, R3 and R4

```
int             NegoMode;

int             NegoWinSize;

int             NegoPeakBW;

int             NegoRTD;

int             NegoPktSize;

int             NegoBlkSize;
```

l. Receiver main power - shared by T_CHAN, R1, R2, R3 and R4.

```
flag            SNR_ON;
```

L.   Other header files

1. snr_que.h

The header file for snr_que.c.

2. snr_util.h

The header file for snr_util.c.

LIST OF REFERENCES

1. Gilbert M. Lundy and H. Alphan Tipici, "Specification and Analysis of the SNR High-Speed Transport Protocol," *IEEE/ACM Transactions on Networking*, Vol. 2, No 5, October 1994.

2. Farah Mezhoud, "An Implementation of the SNR Protocol (Transmitter Part)," Master's Thesis, Naval Postgrduate School, Monterey, CA, March 1995.

3. Arun N. Netravali, W. D. Roome and K. Sabnani, "Design and Implementation of A High-Speed Transport Protocol," *IEEE Transactions on Communications*, Vol. 38, No. 11, November 1991.

4. W. Richard Stevens, *UNIX Network Programming*, Prentice Hall, Atlanta, GA, 1990.

5. W. Richard Stevens, *TCP/IP Illustrated vol. 1*, Addison-Wesley, Chicago, IL, 1994.

6. Gilbert. M. Lundy and R. C. McArthur, *Formal Model of a High Speed Transport Protocol in Protocol Specification, Testing & Verification XII*, North-Holland, Amsterdam, Holland, 1992.

7. William Stallings, *Data and Computer Communications,* fourth edition, Macmillan, Philadelphia, PA, 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria VA 22304-6145

2. Library Code 52   2
   Naval Postgraduate School
   Monterey CA 93943-5101

3. Chairman Code EC   1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey CA 93943-5121

4. Chairman Code CS   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey CA 93943-5118

5. Prof. G. M. Lundy, Code CS/Ln   2
   Department of Computer Science
   Naval Postgraduate School
   Monterey CA 93943-5118

6. Prof. Shridhar B. Shukla, Code EC/Sh   1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey CA 93943-5121

7. Library   1
   Chinese Naval Academy
   P.O. Box 90175, Tso-Ying,
   Kao-Hsiung, Taiwan, R.O.C.

8. Cdr. Wan, Wen-Jyh   2
   #57 Aly 48 Ln 29 Chung-Hwa-Yi Rd.
   Kao-Hsiung, Taiwan, R.O.C.