

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**FILE TRANSFER WITH SNR
HIGH-SPEED TRANSPORT PROTOCOL**

by

Veliddin Eran Sezgin

December 1995

Thesis Advisor:

Gilbert Lundy

Approved for public release; distribution is unlimited.

19960408 107

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, 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 December 1995		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE File Transfer With SNR High-Speed Transport Protocol			5. FUNDING NUMBERS	
6. AUTHOR(S) Sezgin, Veliddin Eran				
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 United States Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) To validate SNR as a high speed transport protocol, efficient means of transferring large data files are required. The problem is that no file transfer program is currently implemented for SNR. The SNR protocol was described in IEEE Transactions on Communications 91 Vol. 38 #11. The approach taken was to modify the Trivial File Transfer Protocol (TFTP) and use it with the SNR Receiver and Transmitter implementations in both the FDDI and Ethernet LANs. The program was developed on top of the IP layer in the UNIX operating system using the C programming language. The UNIX system features that were adopted for this implementation were multitasking, shared memory, raw sockets and process control. This required overcoming the problems as signal loss, shared memory size, conflicts among the raw sockets and network interface configuration in an IP host. The results were a fully functioning TFTP code for a modified SNR Transmitter and Receiver code and a new scheme in transferring files with SNR. An artifact of this thesis was that both client and server were single CPU running eleven processes each for file transfers. Due to this constraint, a large amount of latency in file transfer times, compared to Internet Protocol FTP, was observed.				
14. SUBJECT TERMS File Transfer Protocol, SNR, Implementation, High Speed Network, Raw Socket, Maximum Transmission Unit,			15. NUMBER OF PAGES 46	
			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	

Approved for public release; distribution is unlimited

FILE TRANSFER WITH SNR HIGH-SPEED TRANSPORT PROTOCOL

Veliddin Eran Sezgin
LTJG, Turkish Navy
B.S., Naval Academy, Turkey, 1988

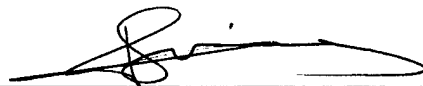
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

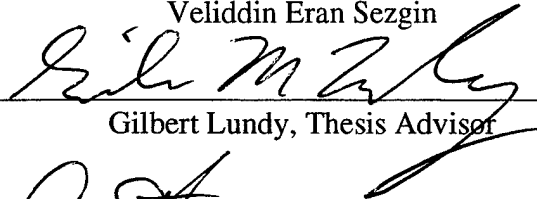
**NAVAL POSTGRADUATE SCHOOL
December 1995**

Author: _____

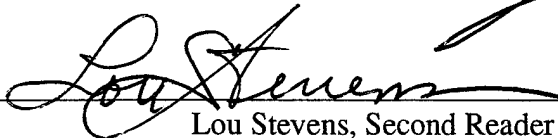


Veliddin Eran Sezgin

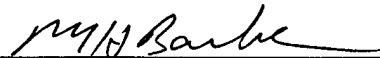
Approved by: _____



Gilbert Lundy, Thesis Advisor



Lou Stevens, Second Reader



Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

To validate SNR as a high speed transport protocol, efficient means of transferring large data files are required. The problem is that no file transfer program is currently implemented for SNR. The SNR protocol was described in IEEE Transactions on Communications 91 Vol. 38 #11.

The approach taken was to modify the Trivial File Transfer Protocol (TFTP) and use it with the SNR Receiver and Transmitter implementations in both the FDDI and Ethernet LANs. The program was developed on top of the IP layer in the UNIX operating system using the C programming language. The UNIX system features that were adopted for this implementation were multitasking, shared memory, raw sockets and process control. This required overcoming the problems as signal loss, shared memory size, conflicts among the raw sockets and network interface configuration in an IP host.

The results were a fully functioning TFTP code for a modified SNR Transmitter and Receiver code and a new scheme in transferring files with SNR. An artifact of this thesis was that both client and server were single CPU running eleven processes each for file transfers. Due to this constraint, a large amount of latency in file transfer times, compared to Internet Protocol FTP, was observed.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	MOTIVATION AND BACKGROUND FOR THIS WORK	4
B.	TRIVIAL FILE TRANSFER PROTOCOL (TFTP)	4
C.	IMPLEMENTATION ENVIRONMENT	5
D.	ORGANIZATION OF THIS THESIS	6
II.	RAW SOCKET APPLICATION PROGRAM INTERFACE (API)	7
A.	INTRODUCTION	7
B.	GENERAL	7
C.	PROTOCOL CONTROL BLOCKS (PCBs)	9
D.	DELIVERY OF A RECEIVED PACKET IN RAW IP SOCKETS	9
E.	RAW IP PACKET SIZE	11
III.	FTP IMPLEMENTATION WITH SNR	13
A.	TERMINOLOGY AND CONCEPTS IN CLIENT SERVER	13
B.	OVERVIEW OF THE TFTP	14
C.	DATA FORMATS IN TFTP	15
D.	RELATION TO IMPLEMENTATION OF SNR FTP PROTOCOL	16
E.	MAXIMUM TRANSMISSION UNIT (MTU)	20
F.	SHARED MEMORY SIZE	23
G.	ORGANIZATION OF THE PROCESSES IN SNR FTP	23
H.	AN EXAMPLE	26
IV.	LIMITATIONS OF SNR/IP APPLICATIONS	29
A.	IMPLEMENTATION LIMITATIONS	29

B.	THEORETICAL SNR PERFORMANCE	29
V.	EVALUATION	33
A.	GENERAL	33
B.	FUTURE IMPROVEMENTS	33
	LIST OF REFERENCES	35
	INITIAL DISTRIBUTION LIST	37

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. It was first introduced in (Netravali, et al.,1991), then in (Lundy, Tipici, 1994) a formal specification was given using the System of Communicating Machines (SCM) model. More details about the protocol can be found in (Lundy, Tipici, 1994).

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.

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.

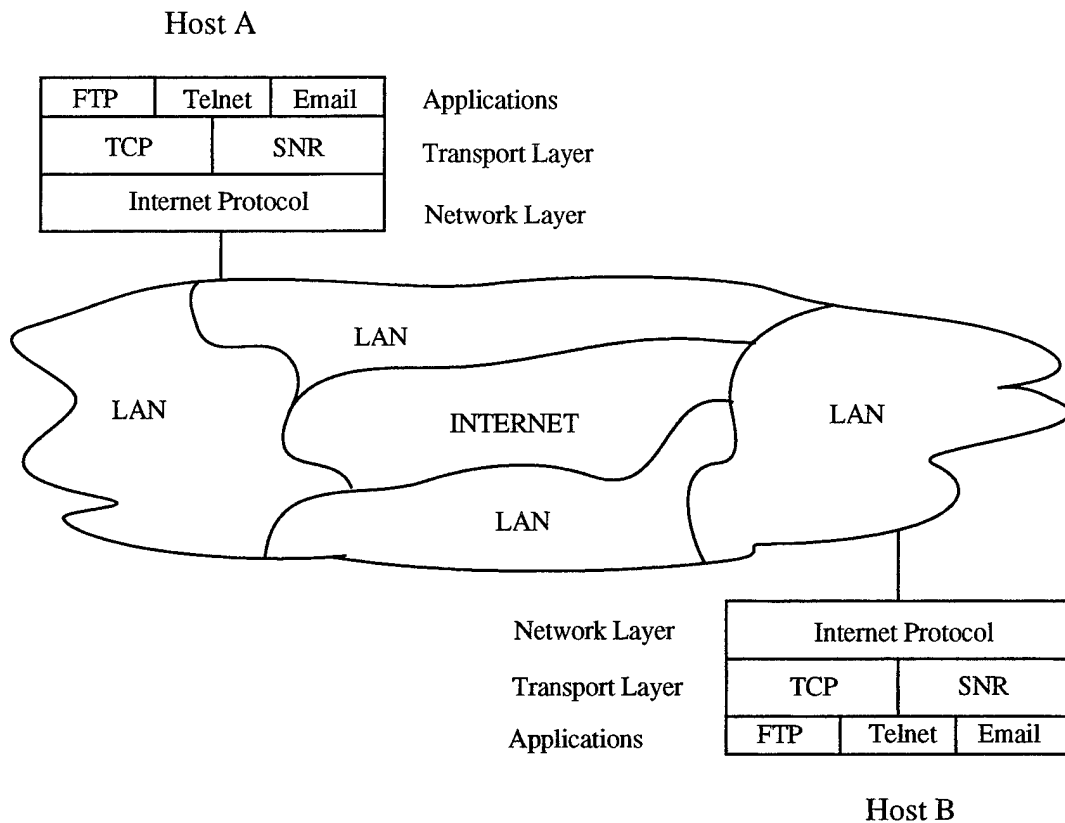


Figure 1. Network Hosts, Entities and SNR Protocol Process

We have used two UNIX workstations in our FTP implementation. They are connected to each other with both FDDI and Ethernet. Two workstations, namely WHITE and GOLD, run System V, version IV of UNIX operating system.

The general organization of the machines in two hosts running SNR is shown in Figure 2. Each machine in the protocol perform a specific function in coordination with other machines. The coordination is established by communicating through shared variables.

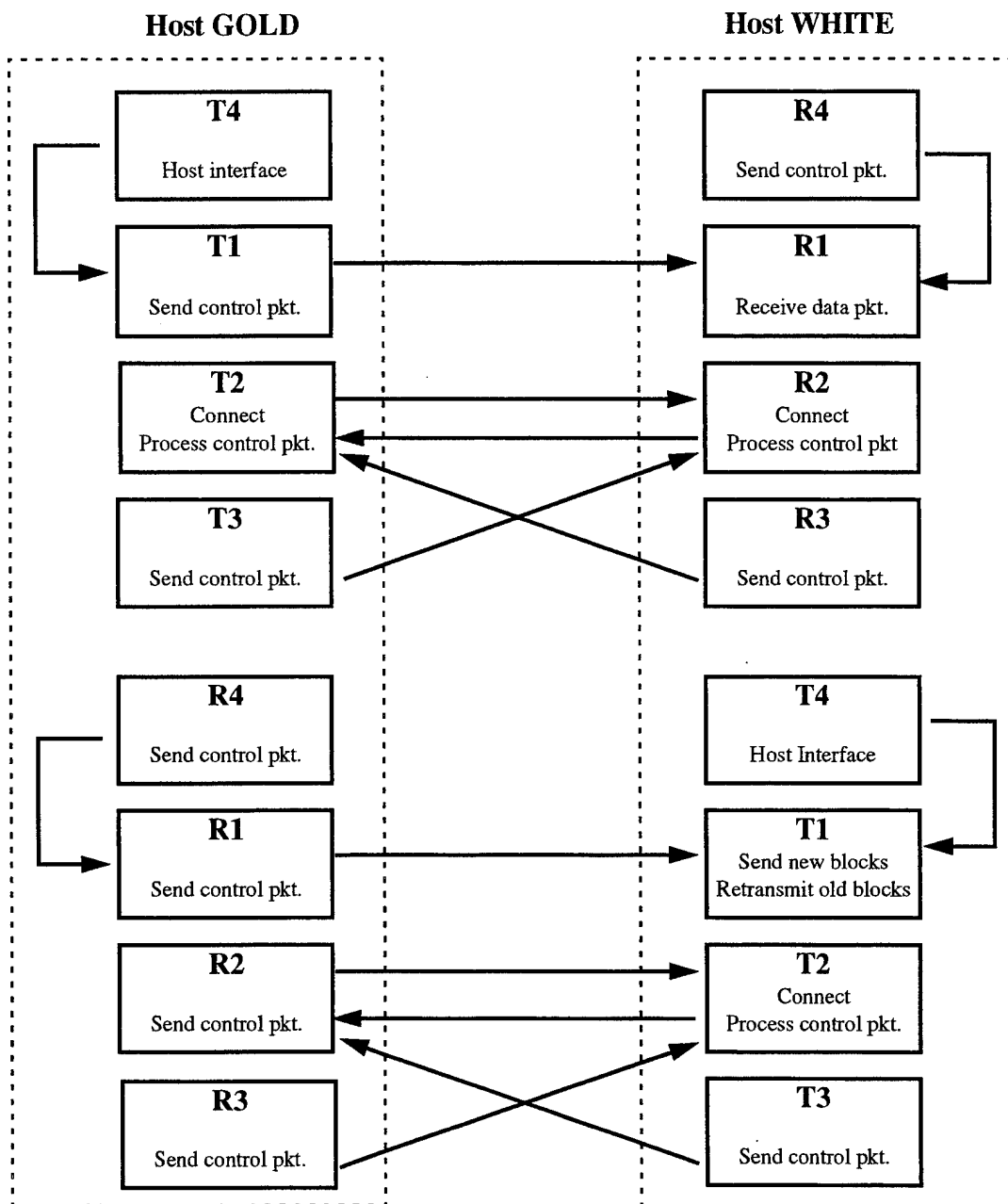


Figure 2. Machine Organizations in SNR Layer in Two Hosts. After Lundy, Tipici, 1994.

A. MOTIVATION AND BACKGROUND FOR THIS WORK

File transfer is an important part of any network. In this thesis we develop a client and server implementation of the Internet Trivial File Transfer Protocol (TFTP) (Sollins, 1981) by using SNR transport protocol designed for high speed networks. This file transfer protocol is specified by Request for Comments (RFC) 783. Although its specification calls for it to be implemented using UDP, a cooperating client and server pair can be implemented using almost any desired transport protocol.

We have used the SNR receiver (Wan, 1995) implemented by W. J. Wan, LTJG Taiwanese Navy and SNR transmitter (Mezhoud, 1995) implemented by Farah Mezhoud, Major Tunisian Army, as the underlying transport protocol.

B. TRIVIAL FILE TRANSFER PROTOCOL (TFTP)

Trivial File Transfer Protocol (TFTP) is a very simple protocol used to transfer files between two systems. It was designed to be small and easy to implement. It is much smaller than the Internet File Transfer Protocol (FTP) and does not provide many of the features that FTP provides (directory listings, user authentication, etc.). The only service provided by TFTP is the ability to send and receive files between a client process and a server process. It is implemented on top of the Internet User Datagram protocol (UDP or Datagram) to transfer files between machines on different networks.

Although FTP is the most general file transfer protocol in the TCP/IP suite, it is also the most complex and difficult to program (Comer, 1991, Vol. 1). Many applications do not need the full functionality FTP offers, nor can they afford the complexity. The only thing TFTP can do is read and write files (or mail) from or to a remote server. It can not list directories and currently has no provisions for user authentication. In common with other Internet protocols, it is character oriented and uses packets of up to 65535 bytes.

Three modes of transfer are currently supported in TFTP: netascii¹; octet, eight bit bytes; mail, netascii characters sent to a user rather than a file. Additional modes can be defined by pairs of cooperating hosts. Each nonterminal packet is acknowledged separately.

C. IMPLEMENTATION ENVIRONMENT

The experimental LAN environment used in SNR FTP is depicted in Figure 3. Each computer has two network interfaces, Ethernet and FDDI with separate IP addresses, as they are shown in Figure 3. The SNR Receiver (Wan, 1995) and Transmitter (Mezhoud, 1995) implementations are done on the same LAN. The machine Gold and White has SunOS Release 5.3, whereas, the machine Black has IRIX Release 4.0.5.F as the operating system. Both operating systems are different versions of UNIX.

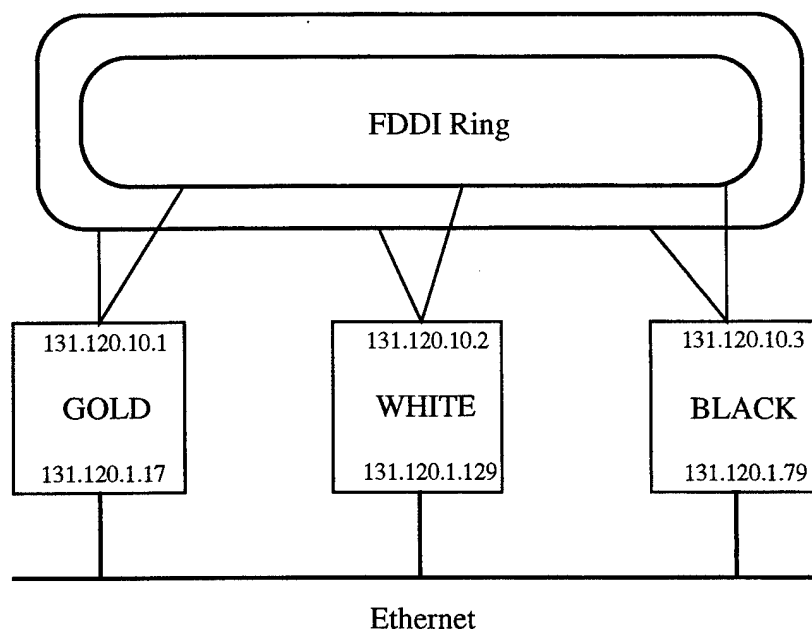


Figure 3. Experimental LAN

1. This is ASCII as defined in "USA Standard Code for Information Interchange (USASI, 1968)" with the modifications specified in "Telnet Protocol Specification (RFC 764, 1980)". It is eight bit ASCII and the term "netascii" will be used throughout the thesis to mean this particular version of ASCII.

D. ORGANIZATION OF THIS THESIS

This thesis is organized in the following manner: Chapter I is an introduction and briefly outlines the SNR transport protocol and the work done prior to this thesis. The chapters are organized to relay background information to the reader about the practical constraints imposed by multiprocessing with one-CPU hosts on SNR FTP implementation.

Chapter II gives information about raw socket application program interface, which is used in the UNIX environment to develop and experiment with the protocols running on Internet Protocol. Based on the discussion in this chapter, we have improved the SNR Receiver (Wan, 1995) and Transmitter (Mezhoud, 1995) implementations.

Chapter III gives an insight into the client-server paradigm and the practical limitations we have experienced. We show how much we have adopted the generally accepted client-server rules in our implementation. We conclude the chapter with an example explaining what happens just before the actual file transport starts between two hosts, regarding the Address Resolution Protocol (ARP).

Chapter IV gives information about theoretical SNR performance, and a comparison is made with the theoretical performance of TCP transport protocol, based on the discussion in (Stevens, 1994).

Chapter V is about the evaluation of the implementation with the recommended future improvements that can be done in this implementation.

Because of its capacity, the source code is not included in this thesis. The line counts of source code for the transmitter, receiver, server, and client are 5000, 4450, 3000, and 4800 respectively. The SNR Receiver (Wan, 1995) source code is not modified however, the Transmitter (Mezhoud, 1995) source code is modified significantly. The SNR FTP server and client source codes are based on the publicly available TFTP source code in 4.3 BSD UNIX. For a detailed discussion about the TFTP source code, refer to (Stevens, 1990).

II. RAW SOCKET APPLICATION PROGRAM INTERFACE (API)

A. INTRODUCTION

This chapter provides a brief discussion of the significant aspects of the *raw socket* in UNIX System V, taken from (Stevens, 1995, Vol. 2). The IP, TCP and UDP software is a part of today's UNIX operating systems. Any transport protocol other than TCP or UDP can be implemented outside the operating system by using *raw socket* API. Once the protocol has stabilized, it can be made part of the operating system to improve the performance. Since it is used in the SNR Receiver (Wan, 1995) and Transmitter (Mezhoud, 1995) implementations, the manner in which it handles the IP packets affects the performance of SNR FTP implementation.

B. GENERAL

The *raw socket* requires superuser privilege to create. The superuser is allowed unrestricted access to files and additional permissions over other processes. This is to prevent random users from writing their own IP datagrams to the network. *Raw socket* provides three capabilities for an IP host:

- They are used to send and receive ICMP and IGMP messages.
- They allow a process to build its own IP headers.
- They allow additional IP-based protocols to be supported in a user process.

The socket API fills in a few fields in the IP header, but it also allows a process to supply its own IP header. This allows diagnostic programs to create any type of IP datagram. The socket API input provides three types of filtering for incoming IP datagrams. The process chooses to receive datagrams based on:

- The protocol number field (set by *socket* system call),
- The source IP address (set by *connect* system call), and
- The destination IP address (set by *bind* system call).

The process chooses which combination of these three filters to apply. Figure 4 shows the related fields of a received IP packet that is filled in by sender by using *socket*, *bind* and *connect* system calls.

These three tests imply that a process can create a raw socket with a protocol of 0 and not *bind* a local address, not *connect* to a foreign address, and the process receives all datagrams processed by the operating system.

In raw socket, an IP header can be filled in by the operating system or provided by the application. An application can provide an IP header by enabling the socket, using the system call *setsockopt* with the argument *IP_HDRINCL*. The *Type of Service (TOS)* and the *Time to Live (TTL)* fields of *raw socket* IP packet is set to 0 and 255, respectively, when the operating system fills in the header.

One drawback in using raw IP sockets for an application is that the application process can not be traced with a debugger in user space, since they require super user privilege to run.

0	15	16	31	
Version	Header Length	Type of Service	Total Length of the Packet	
Unique ID Number of the Packet			Flag	Fragment Offset
Time to Live		Protocol No	Header Checksum	
Source (Sender's) IP Address of the Packet				
Destination (Receiver's) IP Address of the Packet				

Figure 4. IP Datagram Header

C. PROTOCOL CONTROL BLOCKS (PCBs)

In the UNIX, when a process executes the system call *socket*, the process table structure is accessed by the operating system. One of the fields in that structure keeps track of the file descriptors that is used by this process via a chain of structures connected to each other with pointers. The descriptor that is returned by the *socket* system call is basically a derivative of conventional file descriptor. The part we are concerned with is the doubly linked list of Protocol Control Block structures pointed by the socket structure.

PCB structure fields that concern us are depicted in Figure 5. The *foreign address* and *local address* fields of a PCB is filled in by *connect* and *bind* system calls respectively. The protocol no field is provided by the third argument of *socket* system call when creating a socket. For a detailed discussion, refer to (Stevens, 1994, Vol. 2).

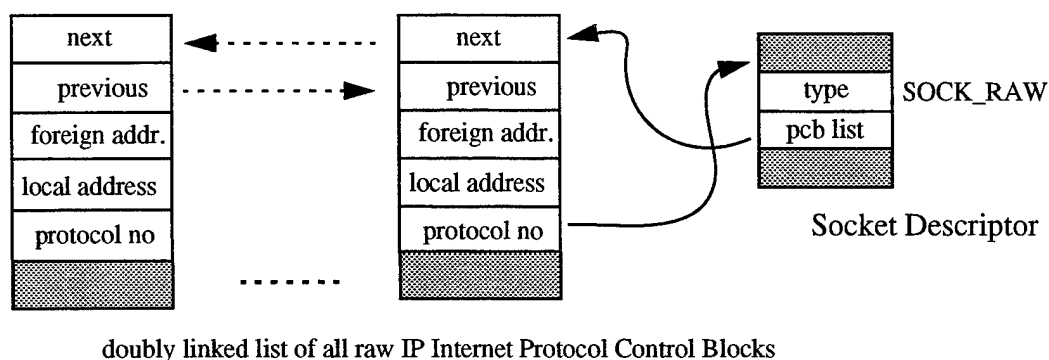


Figure 5. Protocol Control Blocks of a Socket Descriptor. After Stevens, 1994, Vol. 2.

D. DELIVERY OF A RECEIVED PACKET IN RAW IP SOCKETS

Incoming raw IP packets are handled by the kernel according to the following steps:

1. Save Source IP Address

The source address from the received IP datagram is put into a global variable in the system level, which is used in finding a matching PCB. Unlike UDP or TCP, there is

no concept of a port number with raw IP, so the *sin_port* field in the *sockaddr_in* structure is always 0.

2. Search All IP PCBs for One or More Matching Entries

The raw socket API handles its list of PCBs differently from UDP and TCP. UDP and TCP protocols maintain a pointer to the PCB for the most recently received datagram (a one-behind cache). They then call the generic function *in_pcblookup* to search for a single best match, when the received datagram does not equal the cache entry. The raw socket API has totally different criteria for a matching PCB, so it searches the PCB list itself. A raw IP datagram can be delivered to multiple sockets, so every PCB on the PCB list must be scanned. This is similar to UDP's handling of a received datagram destined for a broadcast or multicast address.

3. Compare Protocols

If the protocol field in the PCB is nonzero, and if it doesn't match the protocol field in the IP header, the PCB is ignored. This implies that a raw socket with a protocol value of 0 (the third argument to *socket* system call) can match any received raw IP datagram.

4. Compare Local and Foreign IP Addresses

If the local address in the PCB is nonzero, and if it doesn't match the destination IP address in the IP header, the PCB is ignored. If the foreign address in the PCB is nonzero, and if it doesn't match the source IP address in the IP header, the PCB is ignored.

5. Pass Copy of Received Datagram to Processes

Operating system passes a copy of the received datagram to the process. If more than one process receives a copy of the datagram, operating system makes the copies and passes to the processes. However, if only one process receives the datagram, there is no need to make a copy.

6. Undeliverable Datagram

If no matching sockets are found for the datagram from the PCB list, some system level counters are incremented or decremented. For a detailed discussion, refer to (Stevens, 1994, Vol. 2).

Handling of the incoming raw IP packets led us to have two separate raw IP sockets in our implementation; one for receiver and one for transmitter on each host. This prevented client's transmitter (receiver) from processing server's transmitter (receiver) control packets and vice versa. The modification saved a considerable amount of time in file transfer.

E. RAW IP PACKET SIZE

The maximum size of an IP datagram is 65535 bytes, imposed by the 16-bit *total length of the packet* field in the IP header (Figure 4). With an IP header of 20 bytes and a SNR header of eight bytes, this leaves a maximum of 65507 bytes of user data in a SNR/IP datagram. Most IP implementations, however, provide less than this maximum packet size.

There are two limits we can encounter. First, the application program may be limited by its programming interface. The sockets application program interface (API) provides a function, the application can call to set the size of the receive and the send buffer. For a *raw socket*, this size is directly related to the maximum size IP datagram the application can read or write. Most systems today provide a default of just over 8192 bytes for the maximum size of a *raw socket* IP datagram that can be read or written. The next limitation comes from the kernel's implementation of IP. There may be implementation features (or bugs) that limit the size of an IP datagram to less than 65535 bytes.

III. FTP IMPLEMENTATION WITH SNR

A. TERMINOLOGY AND CONCEPTS IN CLIENT SERVER

The client server paradigm divides communicating applications into two broad categories, depending on whether the application waits for communication or initiates it. This section provides a concise, comprehensive definition of the two categories, and explains many of the subtleties. (Comer, 1991)

1. Clients and Servers

The client-server paradigm uses the direction of initiation to categorize whether a program is a client or server. In general, an application that initiates peer-to-peer communication is called a client. Each time a client application executes, it contacts a server, sends a request, and awaits a response. When a response arrives, the client continues to process. Clients are often easier to build than servers and usually require no special system privileges to operate. A server is any program that waits for incoming communication requests from a client. The server receives a client's request, performs the necessary computation, and returns the result to the client.

2. Connectionless Vs. Connection Oriented Servers

Connectionless transport implies unreliable delivery, whereas, connection oriented transport means reliable delivery. Connectionless transport should be used, if the application using it handles reliability, or each client accesses its server on a local area network that exhibits extremely low loss and no packet reordering. Connection oriented transport should be used whenever a wide area network separates the client and server.

SNR has three different data transport modes; mode 0, mode 1, and mode 2. SNR provides all the reliability needed to communicate across an inter network with the mode 2, which has a flow control and error checking. Interaction of server and client in mode 2 is in connection oriented style, as in TCP/IP. On the other hand, mode 0, with no flow control and error checking (i.e., retransmission), leads to a connectionless style interaction between client and server, as in UDP/IP. Mode 1, with the flow control, is a connection

oriented style interaction with no error checking. Mode 0 is the fastest mode with less overhead, whereas, mode 2 is the most reliable mode with a disadvantage of lower performance when compared to mode 0. Data transfer in mode 0 works well in a local environment because reliability errors seldom occur in a local environment. Errors usually arise only when communication spans a wide area network where packets may be duplicated, dropped or delivered out of order.

3. Stateless Vs. Stateful Servers

Information that a server maintains about the status of ongoing interactions with clients is called state information. Servers that do not keep any state information are called stateless servers; others are called stateful servers. Stateless server scheme is adopted in the implementation of SNR FTP since one connection at a time can be made to the server.

B. OVERVIEW OF THE TFTP

Any transfer begins with a request to read or write a file, which also serves to request a connection. If the server grants the request, the connection is opened and the file is sent in fixed length blocks of 512 bytes. Each data packet contains one block of data and must be acknowledged by an acknowledgment packet before the next packet can be sent. A data packet of less than 512 bytes signals termination of a transfer. If a packet gets lost in the network, the intended recipient will time-out and may retransmit his last packet (which may be data or an acknowledgment), thus, causing the sender of the lost packet to retransmit that lost packet. The sender has to keep just one packet on hand for retransmission, since the lock step acknowledgment guarantees that all older packets have been received. Both machines involved in a transfer are considered senders and receivers. One sends data and receives acknowledgments, the other sends acknowledgments and receives data.

Most errors cause termination of the connection and are signalled by sending an error packet. This packet is not acknowledged and not retransmitted (i.e., a TFTP server or user may terminate after sending an error message), so the other end of the connection may

not receive it. Therefore, time-outs are used to detect such a termination when the error packet has been lost. Errors are caused by three types of events:

- Not being able to satisfy the request (e.g., file not found, access violation, or no such user).
- Receiving a packet which cannot be explained by a delay or duplication in the network (e.g., an incorrectly formed packet).
- Losing access to a necessary resource (e.g., disk full or access denied during a transfer).

This protocol is very restrictive in order to simplify implementation. For example, the fixed length blocks make allocation straight forward, and the lock step acknowledgment provides flow control and eliminates the need to reorder incoming data packets.

C. DATA FORMATS IN TFTP

There are two formats of data transfer supported by TFTP: netascii and octet. The netascii format is used for transferring text files between the client and server. The standard ASCII character set is used and the end of each text line is designated by a carriage return (octal 15) followed by a line feed (octal 12). If there is a carriage return in the text file, it is transferred as a carriage return followed by a null byte (octal 0). The presence of a carriage return followed by any other character is undefined.

By defining a standard format for the text file that is being transferred, it is possible to transfer data between two different systems. It is the responsibility of the client and the server to convert the local file representation to and from netascii. For example, to transfer a text file between an IBM 370 and a VAX UNIX system, the IBM implementation of TFTP has to convert a file from EBCDIC to ASCII and insert the carriage return, line feed pairs at the end of every line. The VAX UNIX system has to look for the carriage return-line feed pairs and remove the carriage return, since UNIX stores text files with only a line feed separating the lines. The receiving UNIX system also has to look for a carriage return followed by a null, then remove the null. The octet data format is used for transferring binary files. There are two primary uses of TFTP to transfer binary data between systems.

First, two systems with the same architecture can exchange a binary file without any problems. Second, if a system receives a binary file and then returns it to the system that sent it originally, the format of the file must not change. This scenario can be used to provide a file server. The clients send their files to the server in binary mode and retrieve them later in binary mode- The server would not be trying to interpret the contents of the binary file. It is merely storing it on its local file system. As long as it uses the same storage technique to store and retrieve a binary file, the actual contents of the file won't change.

D. RELATION TO IMPLEMENTATION OF SNR FTP PROTOCOL

TFTP is implemented on the UDP protocol, hence, packets will have an Internet header, a Datagram header, and a TFTP header. Additionally, the packets may have a header (Ethernet, FDDI, etc.) to allow them through the local transport medium. As shown in Figure 6, the order of the contents of a packet will be; local medium header, Internet header, Datagram header, TFTP header, followed by the remainder of the TFTP packet, which may or may not be data depending on the type of packet, as specified in the TFTP header. TFTP does not specify any of the values in the Internet header. As mentioned, TFTP can be implemented on top of any desired transport protocol. We replaced the UDP header with SNR header in our implementation, as depicted in Figure 7.

The TFTP header consists of a two-byte Opcode field, which indicates the packet's type (e.g., DATA, ERROR, etc.). These opcodes and the formats of the various types of packets are discussed further in the section on SNR FTP packets.

Local Medium	IP	UDP	TFTP
--------------	----	-----	------

Figure 6. Order of Headers in Implementation of TFTP

Local Medium	IP	SNR	TFTP
--------------	----	-----	------

Figure 7. Order of Headers in Implementation of SNR FTP

1. Initial Connection Protocol

A file transfer is established by sending a request (WRQ to write onto a foreign file system, or RRQ to read from it) and receiving a positive reply, an acknowledgment packet for write, or the first data packet for read. In general, an acknowledgment packet will contain the block number of the data packet being acknowledged. Block numbers are associated with each data packet and are consecutive and begin with one. Since the positive response to a write request is an acknowledgment packet, in this special case the block number will be zero. (Normally, since an acknowledgment packet is acknowledging a data packet, the acknowledgment packet will contain the block number of the data packet being acknowledged). If the reply is an error packet, then the request has been denied.

2. SNR FTP Packets

SNR FTP supports six types of packets, one more than the five types of packets in TFTP. We have added directory listing request packet type to our implementation as opcode six. The SNR FTP header of a packet contains the Opcode associated with that packet, as listed in Table 1.

RRQ and WRQ packets have the format shown in Figure 8. The file name is a sequence of bytes in netascii terminated by a zero byte. The mode field contains the string "netascii", "octet", or "mail" (or any combination of upper and lower case, such as "NETASCII", Netascii", etc.). A host which receives netascii mode data must translate the data to its own format. Octet mode is used to transfer a file that is in the eight-bit format of the machine from which the file is being transferred. It is assumed that each type of machine has a single eight-bit format that is more common, and that format is chosen. If a host receives a octet file and then returns it, the returned file must be identical to the original.

Opcode	Operation
1	Read request (RRQ)
2	Write request (WRQ)
3	Data (DATA)
4	Acknowledgment (ACK)
5	Error (ERROR)
6	Directory listing request (DRQ)

Table 1. Opcodes Used in the SNR FTP Header. After Sollins, 1981.

2 bytes	string	1 byte	string	1 byte
Opcode	File Name	0	Mode	0

Figure 8. Read Request (RRQ) / Write Request (WRQ) Packet

Data is actually transferred in DATA packets, as depicted in Figure 9. DATA packets (Opcode = 3) have a block number and data field. The block numbers on data packets begin with one and increase by one for each new block of data. This restriction allows the program to use a single number to discriminate between new packets and duplicates. The data field is from zero to 512 bytes long. If it is 512 bytes long, the block is not the last block of data; if it is from zero to 511 bytes long, it signals the end of the transfer.

2 bytes	2 bytes	n bytes
Opcode (3)	Block No	Data

Figure 9. Data Packet

All packets other than those used for termination are acknowledged individually, unless a time-out occurs. Sending a DATA packet is an acknowledgment for the ACK packet of the previous DATA packet. The WRQ and DATA packets are acknowledged by ACK or ERROR packets, while RRQ and ACK packets are acknowledged by DATA or ERROR packets. Figure 10 depicts an ACK packet; the Opcode is four.

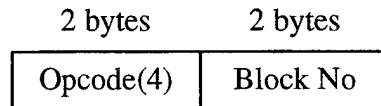


Figure 10. Acknowledgment (ACK) Packet

The block number in an ACK echoes the block number of the DATA packet being acknowledged. A WRQ is acknowledged with an ACK packet having a block number of zero.

An ERROR packet (opcode five) takes the form depicted in Figure 11. An ERROR packet can be the acknowledgment of any other type of packet. The error code is an integer indicating the nature of the error.

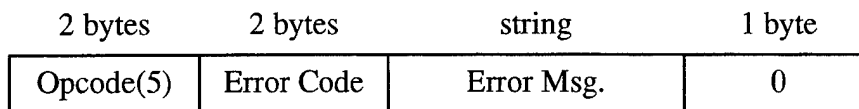


Figure 11. Error (ERROR) Packet

A directory listing request that is added to SNR FTP, DRQ (Opcode six), takes the form in Figure 12. The directory listing formed as data packet is the acknowledgment for a DRQ packet sent by the client.

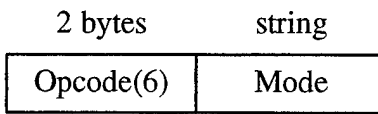


Figure 12. Directory Listing Request (DRQ) Packet

A table of error values and their meanings is given in Table 2. The error message is intended for user comprehension, and should be in netascii. Like all other strings, it is terminated with a zero byte.

Value	Meaning
0	Not defined, see error message (if any)
1	File not found
2	Access violation
3	Disk full or allocation exceeded
4	Illegal TFTP operation
5	Unknown transfer ID
6	File already exists
7	No such user

Table 2. Error Codes

E. MAXIMUM TRANSMISSION UNIT (MTU)

As we can see from Figure 13, there is a limit on the size of the Ethernet frame. This limits the maximum number of bytes of data to 1500. This characteristic of the link layer is called the maximum transmission unit (MTU).

6 bytes	6 bytes	2 bytes	46-1500 bytes	4 bytes
Dest. Addr.	Src. Addr.	Type	Data	CRC

Figure 13. Ethernet Encapsulation

If IP has a datagram to send, and the datagram is larger than the link layer's MTU, IP performs fragmentation. This breaks the datagram into smaller pieces (fragments), so that each fragment is smaller than the MTU. We used the *ifconfig* command available to the UNIX user to examine the MTU of the interfaces on the hosts on which the experiments were performed. Following is the screen output of *ifconfig -au* command on machine *gold*:

```
/n/gold/work/sezgin/% ifconfig -au
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
    inet 127.0.0.1 netmask ff000000
le0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
    inet 131.120.1.17 netmask ffffff00 broadcast 131.120.1.255
    ether 8:0:20:1b:9:7b
sbf0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 4352
    inet 131.120.10.1 netmask ffffff00 broadcast 131.120.10.255
/n/gold/work/sezgin/%
```

The first network interface, namely *lo0*, is the loopback interface, which has a MTU of 8232. The second network interface, which is *le0*, is the Ethernet interface with a MTU of 1500, the same size of an typical Ethernet packet. Finally, the third network interface, *sbf0*, is the FDDI interface with a MTU of 4352. We have used both Ethernet and FDDI interfaces in our experiments.

Most type of networks have an upper limit on MTU. Table 3 lists some typical MTU values taken from (Stevens, 1994, Vol. 1).

Network	MTU (bytes)
Hyperchannel (maximum IP)	65535
16 Mbits/sec token ring (IBM)	17914
4 Mbits/sec token ring (IEEE 802.5)	4464
FDDI	4500
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576

Table 3. Typical Implementation MTUs. From Stevens, 1994, Vol. 1.

When two hosts on the same network are communicating with each other, it is the MTU of the network that is important. However, when two hosts are communicating across multiple networks, each link can have a different MTU. The important numbers are not the MTUs of the two networks that the two hosts connect, but rather the smallest MTU of any data link that packets traverse between the two hosts. This is called the *path MTU*.

The *path MTU* between any two hosts need not be constant. It depends on the route being used at any time. Also, routing need not be symmetric (the route from A to B may not be the reverse of the route from B to A), hence, the *path MTU* need not be the same in the two directions. (Mogul, et al., 1990) specifies the “path MTU discovery mechanism,” a way to determine the path MTU at any time. We used a version of the *traceroute* program (Stevens, 1994, Vol. 1) that uses the *ICMP unreachable* error to determine the *path MTU* of our LAN.

As an experiment, this modified version of *traceroute* was run numerous times to various hosts around the world (Stevens, 1994, Vol. 1). Fifteen countries (including Antarctica) were reached, and various transatlantic and transpacific links were used. Out of 18 runs, only 2 had a path MTU of less than 1500.

This experiment concludes that many, but not all, wide area networks today can handle packets larger than 512 bytes. Using the path MTU discovery feature may allow SNR applications to take advantage of these larger MTUs.

F. SHARED MEMORY SIZE

The shared memory scheme is kept as inter-process communication in our implementation for the reasons explained in (Mezhoud, 1995).

The SunOS 5.3 kernel modules are automatically loaded when needed. There are parameters¹ for the kernel and kernel modules that can be tuned. Many parameters automatically scale as a function of the value assigned to the *maxusers*² parameter in */etc/system* file.

In SNR receiver (Wan, 1995) and transmitter (Mezhoud, 1995) implementations, the packets are implemented as 64 bytes considering the default maximum shared memory size, which is one mega byte, available to the processes. The maximum shared memory segment size parameters of the computers on our LAN are modified³ and increased from one mega-byte to ten mega-bytes. The increase in shared memory allowed to have FTP data packets of 1024 bytes (not including the ftp header) based on the MTU discussion above.

G. ORGANIZATION OF THE PROCESSES IN SNR FTP

The timing dependencies and parent-child relation of the processes in the SNR FTP client implementation is shown in Figure 14. The main process on the client side spawns one set of transmitter and receiver process per host. The main procedure of transmitter (T4) and receiver (snr_r) are responsible for spawning their own set of machines with the name listed under them. The transmitter is in charge of checking if the connection is established

1. To see a complete list of the tunable kernel parameters run the *nm* command on the appropriate module. For example:

`% /usr/ccs/bin/nm /kernel/unix`

2. To see the current values assigned to the kernel parameters type *sysdef -i* and press Return

3. We have entered a line in the */etc/system* file in the form of:

`set shmsys:shminfo_shmmax 10485760`

or not. We transferred the raw IP socket creation from receiver (transmitter) to the main procedures of server and client.

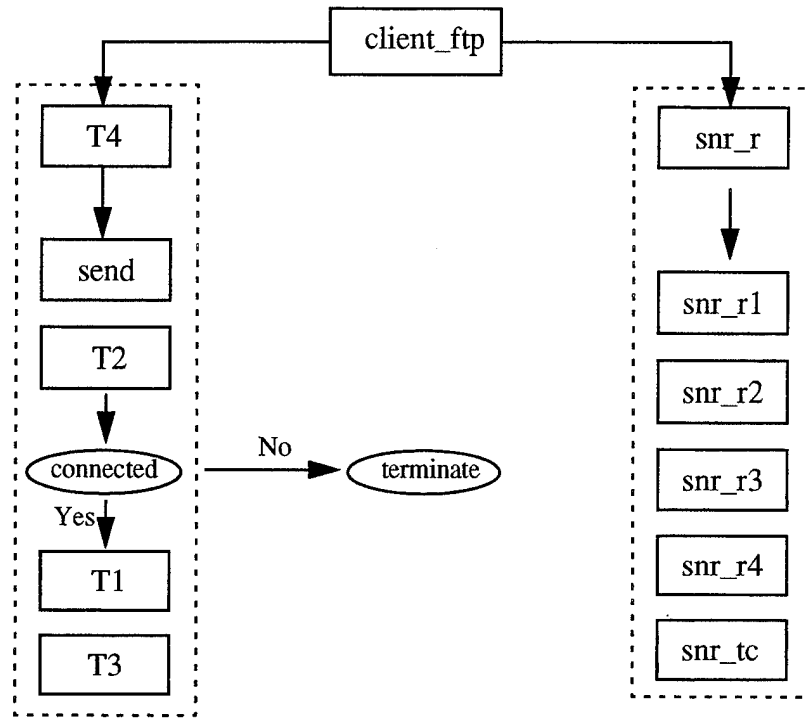


Figure 14. Processes Executed by the “client_ftp”

The timing dependencies and parent-child relation of the processes in the implementation is shown in Figure 15. The server spawns the receiver and waits for a FTP connection request to arrive. Once the connection request is gotten, the transmitter is spawned. The server receiver accepts the raw socket IP packets with a protocol number of 192, which is sent by the client transmitter. The server transmitter receives only the raw socket IP packets with a protocol number of 191, which is sent by the client receiver. So, the corresponding transmitter-receiver pairs of client and server exchange the SNR packets with different protocol numbers. This scheme discards the packets that should not be passed to an unrelated process due to handling of raw socket IP packets. Otherwise, server transmitter could get the client transmitter’s control packets, which forces them to be

discarded in the SNR layer and could increase the overhead of the implementation. The protocol number fields of packets used in the implementation is shown in Figure 16.

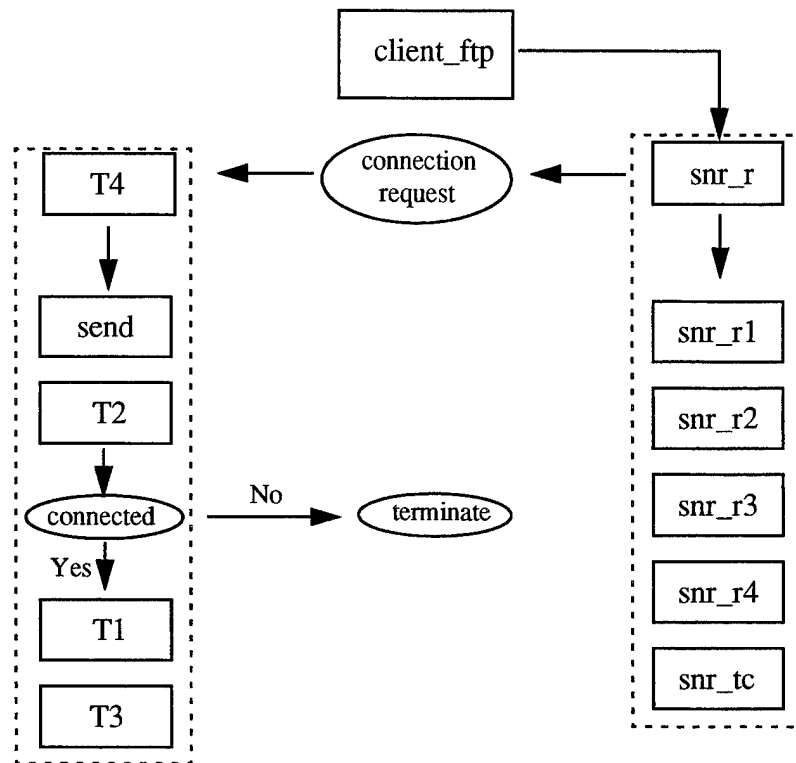


Figure 15. Processes Executed by the "server_ftp"

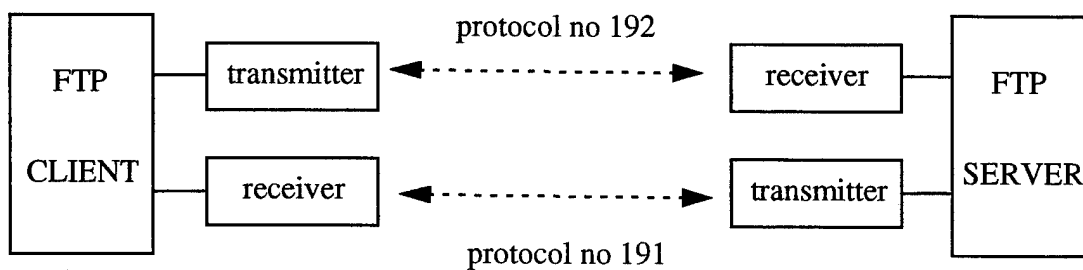


Figure 16. Protocol Numbers of Exchanged Raw Socket IP Packets

H. AN EXAMPLE

Whenever the command *client_ftp white* is typed at machine *gold* the following steps take place. These numbered steps are shown in Figure 17.

1. The application, *client_ftp*, calls the function *gethostbyname* to convert the hostname “*white*” into its 32-bit IP address. This function is called a *resolver* in the DNS (Domain Name System). This conversion is done using the DNS, or on smaller networks, as in our LAN, a static hosts file “*/etc/hosts*”.
2. The FTP client forks its SNR Transmitter to establish a connection with the SNR Receiver at that IP address.
3. The client’s SNR Transmitter sends a connection request packet to the remote host by sending a raw IP datagram to its IP address.
4. If the destination host is on a locally attached network (e.g., Ethernet, FDDI), the raw IP datagram can be sent directly to that host. If the destination host is on a remote network, the IP routing function determines the Internet address of a locally attached next hop router in order to send the IP datagram. In either case, the raw IP datagram is sent to a host or router on a locally attached network. In our LAN, IP datagram is sent directly to machine *white*.
5. Since we use the Ethernet driver, the machine *gold* must convert the 32-bit IP address into a 48-bit Ethernet address. A translation is required from the *logical* Internet address to its corresponding *physical* hardware address. This is the function of Address Resolution Protocol (ARP).
6. With the assumption that there is no entry in our */etc/hosts* file for the machine name *white*, ARP sends an Ethernet frame called an ARP *request* to every host on the network. This is called a *broadcast*. We show the broadcast in Figure 16 with dashed lines. The ARP request contains the IP address of the destination host (whose name is *white*) and is the request “if you are the owner of the IP address, please respond to me with your hardware address.”
7. The destination host’s ARP layer receives this broadcast, recognizes that the sender is asking for its IP address, and replies with an ARP reply. This reply contains the IP address and the corresponding hardware (Ethernet) address.
8. The ARP reply is received and the IP datagram that forced the ARP request-reply to be exchanged can now be sent.
9. The IP datagram is sent to the destination host.

The fundamental concept behind ARP is that the network interface has a hardware address (a 48-bit value for an Ethernet interface). Frames exchanged at the hardware level must be addressed to the correct interface. SNR/IP applications work with its own addresses: 32-bit IP addresses. Knowing a host's IP address doesn't let the operating system send a frame to that host. The operating system (i.e., Ethernet driver, FDDI driver) must know the destination's hardware address to send it data. The function of ARP is to provide a dynamic mapping between 32-bit IP addresses and the hardware addresses used by various network technologies. This requires keeping a periodically updated local *"etc/hosts"* file entries for a better performance of SNR/IP applications.

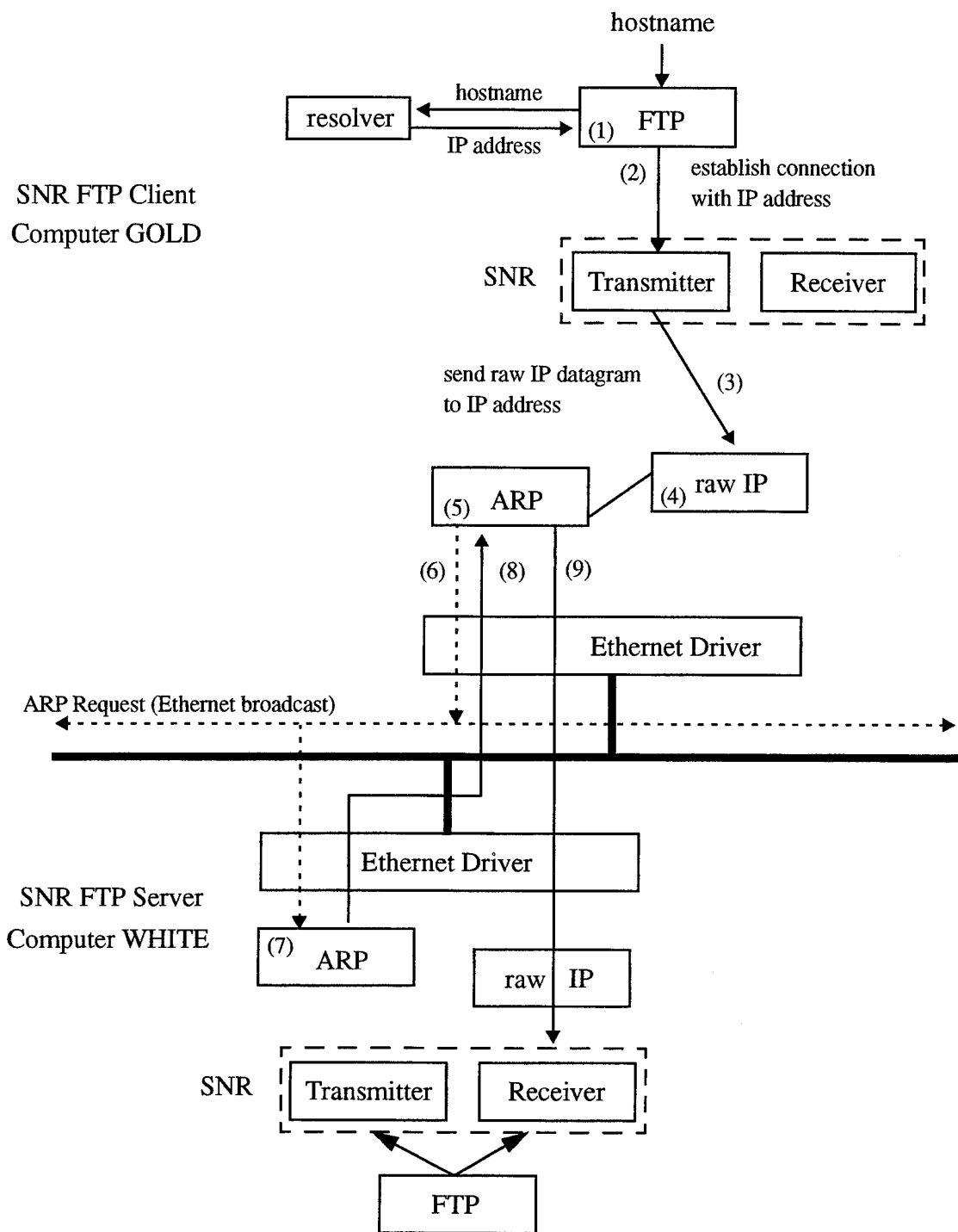


Figure 17. Operation of the command "client_ftp white". After Stevens, 1994, Vol. 1

IV. LIMITATIONS OF SNR/IP APPLICATIONS

A. IMPLEMENTATION LIMITATIONS

The performance of SNR/IP applications are limited by the decisions made during implementation of the SNR Receiver (Wan, 1995) and Transmitter (Mezhoud, 1995). The block number and the packet number fields are implemented as unsigned character, which is one byte. These two fields affect the performance of any SNR/IP application using mode one or two, where data is transferred in blocks of data packets. Hence, the largest file that can be transferred in one connection is:

$$\begin{aligned} &= (\text{max. block number}) \times (\text{max. packet number}) \times (\text{max. packet data length}) \\ &= 255 * 255 * (\text{max. packet data length}) \end{aligned}$$

The maximum packet data length is closely related to the maximum shared memory size that can be allocated for the SNR Transmitter and Receiver in a host. Using packet size of 1024 bytes consumes around eight mega-byte shared memory in the host.

Since mode zero transfer (no error checking, reordering and duplicate detection of packets) is used in SNR FTP, the performance is affected significantly by running eleven processes in single CPU host. The output from a simple “ls” command (directory listing request from server) takes around one second in the experimental LAN, regardless of which FDDI or Ethernet driver is used to reach the server. The only reason for this large amount of latency is the context switching done by the hosts in scheduling eleven processes.

B. THEORETICAL SNR PERFORMANCE

Since Ethernet is the most common LAN topology today, we calculate the theoretical throughput in an Ethernet with basic assumptions. We have used the same methodology as in (Stevens, 1994, Vol. 1). We show the basics for this calculation in Table 4. This table shows the total number of bytes exchanged for a full-sized data packet, and a receiver (or transmitter) control packet is used to acknowledge it. Even though file transfer is done using SNR transfer mode-zero (data is passed to the application layer with no buffering, reordering or error checking), we use the transfer mode-two packet

specifications in our calculations. This will enable us to evaluate the throughput with the most overhead.

Ethernet Packet Field	Data	ACK
Preamble	8	8
Destination address	6	6
Source address	6	6
Type	2	2
IP header	20	20
SNR header	6	6
User data	1474	0
Pad (to Ethernet minimum)	0	20
CRC	4	4
Inter-packet gap (9.6 micro-sec)	12	12
Total	1538	84

Table 4. Field Sizes for SNR Packet on Ethernet. After Stevens, 1995, Vol. 1.

We must account for all the overhead; the preamble, the PAD bytes that are added to the acknowledgment, the CRC, and the minimum inter-packet gap (9.6 microseconds, which equals 12 bytes at 10 Mbits/sec). We first assume the transmitter in SNR client transmits two back-to-back full-sized data blocks, and then the receiver in SNR server sends a receiver control packet acknowledging these two blocks. The maximum throughput (user data) is then,

$$throughput = \frac{(2 \times 255 \times 1474) \text{ bytes}}{((2 \times 255 \times 1538) + 84) \text{ bytes}} \times \frac{10,000,000 \text{ bits/sec}}{8 \text{ bits/byte}} = 1,197,856 \text{ bytes/sec}$$

If the SNR window is opened to its maximum size, which is the maximum number of blocks that can be transferred in one connection, this allows a window of 255 1480-byte segments. If the receiver acknowledges every 128th block, the calculation becomes:

$$throughput = \frac{(128 \times 255 \times 1474) \text{ bytes}}{((128 \times 255 \times 1538) + 84) \text{ bytes}} \times \frac{10,000,000 \text{ bits/sec}}{8 \text{ bits/byte}} = 1,197,982 \text{ bytes/sec}$$

This is the theoretical limit and makes certain assumptions: a receiver state control packet sent by the receiver doesn't collide on the Ethernet with one of the sender's packets; the sender can transmit two blocks with the minimum Ethernet spacing; and the receiver can generate the receiver state control packet within the minimum Ethernet spacing.

As seen in the results of both throughput calculations, SNR takes advantage of transferring data in blocks. Throughput stays practically the same. On the other hand with the same assumptions, TCP has a theoretical performance of 1,155,063 bytes/sec, where sender transmits two back-to-back data packets and receiver sends an acknowledgment for these two packets. If the TCP window is opened to its maximum size (65535), throughput increases to 1,183,667 bytes/sec, less than the maximum throughput of SNR.

Moving to faster networks, such as FDDI (100 Mbits/sec), (Stevens, 1995, Vol. 1) indicates that three commercial vendors have demonstrated TCP over FDDI between 80 and 98 Mbits/sec.

The following practical limits apply for any real-world scenario (Borman, 1991).

- The speed of the slowest link determines the running speed of protocol.
- The memory bandwidth of the slowest machine determines the speed of implementation. This assumes the implementation makes a single pass over the data. If not (i.e., the implementation makes one pass over the data to copy it from user space into the kernel, then another pass over the data to calculate the SNR checksum), implementation run even slower. (Dalton, et al., 1993) describe performance improvements to the standard Berkeley sources that reduce the number of data copies to one. (Partridge, et al., 1993) applied the same "copy-and-checksum" change to UDP, along with other performance improvements, and improved UDP performance by about 30%.

- The window size offered by the receiver, divided by the round-trip time is another limiting factor for the implementation speed. (This is the bandwidth-delay product equation, using the window size as the bandwidth-delay product, and solving for the bandwidth).

The bottom line in all these numbers is that the real upper limit on how fast SNR can run is determined by the size of the SNR window and the speed of light. As concluded by (Partridge, et al., 1993), many protocol performance problems are implementation deficiencies rather than inherent protocol limits.

V. EVALUATION

A. GENERAL

This implementation complies with the TFTP specifications. Some differences exist due to the independent execution of the SNR Receiver and Transmitter processes along with the TFTP process. Extra efforts in almost every way were put on program source code documentation and for the possible improvement of this implementation in the future. The TFTP client and server controls the execution of their Transmitter and Receiver machines, as they are shown in Figure 14 and 15 in Chapter III.

File transfers are performed in both LANs, depicted in Figure 3 in Chapter I, in order to compare the performance of FDDI and Ethernet network drivers with a packet size of 1024 bytes. Since the client and the server each have 11 processes running in one-CPU host, a sound comparison could not be done. The overhead of context switching done among these processes and the handling of IP packets by *raw sockets*, as it is explained in Chapter II, resulted in approximately the same amount of time in file transfers regardless of the network interface used. The other overhead is caused by the number of signals available to any user application in the UNIX operating system. Since two user defined signals in the UNIX, namely *SIGUSR1* and *SIGUSR2*, are used for sending signals among the processes of the Transmitter and the Receiver implementations, the client and the server processes do busy-waiting to check if any FTP packets has arrived, rather than blocking until a signal arrives. This also wastes system resources.

B. FUTURE IMPROVEMENTS

In this implementation, file transfers are performed in data transfer mode-zero of SNR protocol and each FTP data packet is acknowledged separately in a stop-and-go fashion. In order to take advantage of the high speed and reliability of fiber optic networks, a file can be treated as a block and the packet size can be calculated in accordance with the file size. Once a packet size is agreed upon dynamically during connection establishment, data transfer can be performed in block mode with much less acknowledgments compared to stop-and-go interaction.

The latency observed in transferring files can be reduced significantly by transporting the implementation to more than one CPU hosts. This would bring the SNR transport protocol to the same level with the TCP so that its performance can be compared.

The path MTU discovery feature, explained in Chapter III, can be incorporated into the SNR connection establishment phase, and the result can be used in deciding on the SNR packet size dynamically. This would enable the SNR take advantage of MTUs larger than the conventional limit of 576 bytes.

In order to get a good comparable performance against TCP transport protocol, the SNR Receiver and Transmitter implementations should be transported into the operating system. This improvement will prevent the overhead due to processing of *raw socket* IP packets.

LIST OF REFERENCES

- Comer, D. E., Stevens, D. L., *Internetworking With TCP/IP*, Volume III, Prentice Hall, 1991.
- Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., Lumley, J., *Afterburner*, IEEE Network, Vol. 7, No. 4, July 1993.
- Lundy, G. M., Tipici, H., *Specification and Analysis of the SNR High Speed Transport Protocol*, IEEE/ACM Transactions on Networking, Vol. 2, No. 5, October 1994.
- Mezhoud, F., *An Implementation of the SNR Protocol (Transmitter Part)*, Master's Thesis, Naval Post Graduate School, Monterey, CA, March 1995.
- Mogul J. C., Deering, S. E., *Path MTU Discovery*, RFC 1191, April 1990.
- Netravali, A. N., Roome, W. D., Sabnani, K., *Design and Implementation of A High-Speed Transport Protocol*, IEEE Transactions on Communications, Vol. 38, No. 11, November 1991.
- Partridge, C., Mendez, T., Milliken, W., *A Faster UDP*, IEEE/ACM Transactions on Networking, Vol. 1, No. 4, August 1993.
- Sollins, K. R., *Trivial File Transfer Protocol (TFTP) - Revision 2*, RFC 783, June 1981.
- Stevens, W. R., *UNIX Network Programming*, Prentice Hall, 1990.
- Stevens, W. R., *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.
- Stevens, W. R., Wright, G. R., *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.
- Wan, W. J., *An Implementation of the SNR Protocol (Receiver Part)*, Master's Thesis, Naval Post Graduate School, Monterey, CA, March 1995.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center
8725 John J. Kingman Rd. STE 0944
Ft. Belvoir, VA 22060-6218 | 2 |
| 2. | Dudley Knox Library
Naval Postgraduate School
Monterey, CA 93943-5002 | 2 |
| 3. | Chairman, Code CS
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 4. | Prof. Gilbert Lundy, Code CS/LN
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 5. | Prof. Lou Stevens, Code CS/ST
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 6. | V. Eran Sezgin
Emlak Bankasi Konutlari
B-5 Blok 8 Giris Daire 7
Gaziemir, Izmir TURKEY | 2 |