Synopsis

TFTP is a simple protocol to transfer files. It has been implemented on top of the Internet User Datagram protocol (UDP) so it may be used to move files between machines on different networks implementing UDP. (This should not exclude the possibility of implementing TFTP on top of other datagram protocols.) In common with other Internet protocols, it passes 8 bit bytes of data. It is designed to be small and easy to implement. Therefore, it lacks most of the features of a regular FTP. The only thing it can do is read and write files (or mail) from/to a remote server. It cannot list directories, and currently has no provisions for user authentication.

Remote Command Execution is when a process on a host causes a program to be executed on another host. Usually the invoking process wants to pass data to the remote program, and capture its output also. We want to be able to write data that becomes the standard input of the remote process and be able to read this on a different channel from its standard output. Additionally, we would like to be able to read what the remote process writes to its standard error, and be able to read this on a different channel from its standard output.

Contents

1. Introduction

File transfer is an important part of any network. TFTP is a simple protocol to transfer files. It has been implemented on top of the Internet User Datagram protocol (UDP), so it may be used to move files between machines on different networks implementing UDP. In common with other Internet protocols, it passes 8 bit bytes of data. The only thing it can do is read and write files (or mail) from/to a remote server. It cannot list directories, and currently has no provisions for user authentication. Here the protocol has been implemented using C language. The RFC followed is RFC 783 [Sollins 1981]. The operating system used is Linux on Intel platform.

Remote Command Execution is when a process on a host causes a program to be executed on another host. Usually the invoking process wants to pass data to the remote program, and capture its output also. We want to be able to write data that becomes the standard input of the remote process and be able to read data on a different channel from its standard output. Additionally, we would like to be able to read what the remote process writes to its standard error, and be able to read this on a different channel from its standard output. If we don't separate these two output streams from the remote process, it'll be impossible to know which output corresponds to standard output or standard error. Finally, we would like a way to be able to send a signal to the remote process, as another way of controlling its execution. We have implemented remote command execution here using C language, with Linux as Operating system on Intel platform.

2.a. The OSI model

In networking, the communication task done by different modules called layers. The task is divided into pieces and each layer concentrate on providing a particular function. The open systems interconnection modal (OSI) consists of 7 layers as shown.

1. Application Layer
2. Presentation Layer
3. Session Layer
4. Transport Layer
5. Network Layer
6. Data Layer
7. Physical Layer

Layering leads to definition of protocols at each layer. The various layers in its order of hierarchy are shown in fig 2.1.

7	Application Layer
6	Presentation Layer
5	Session Layer
3	Transport Layer
2	Network Layer
1	Data Layer
	Physical Layer

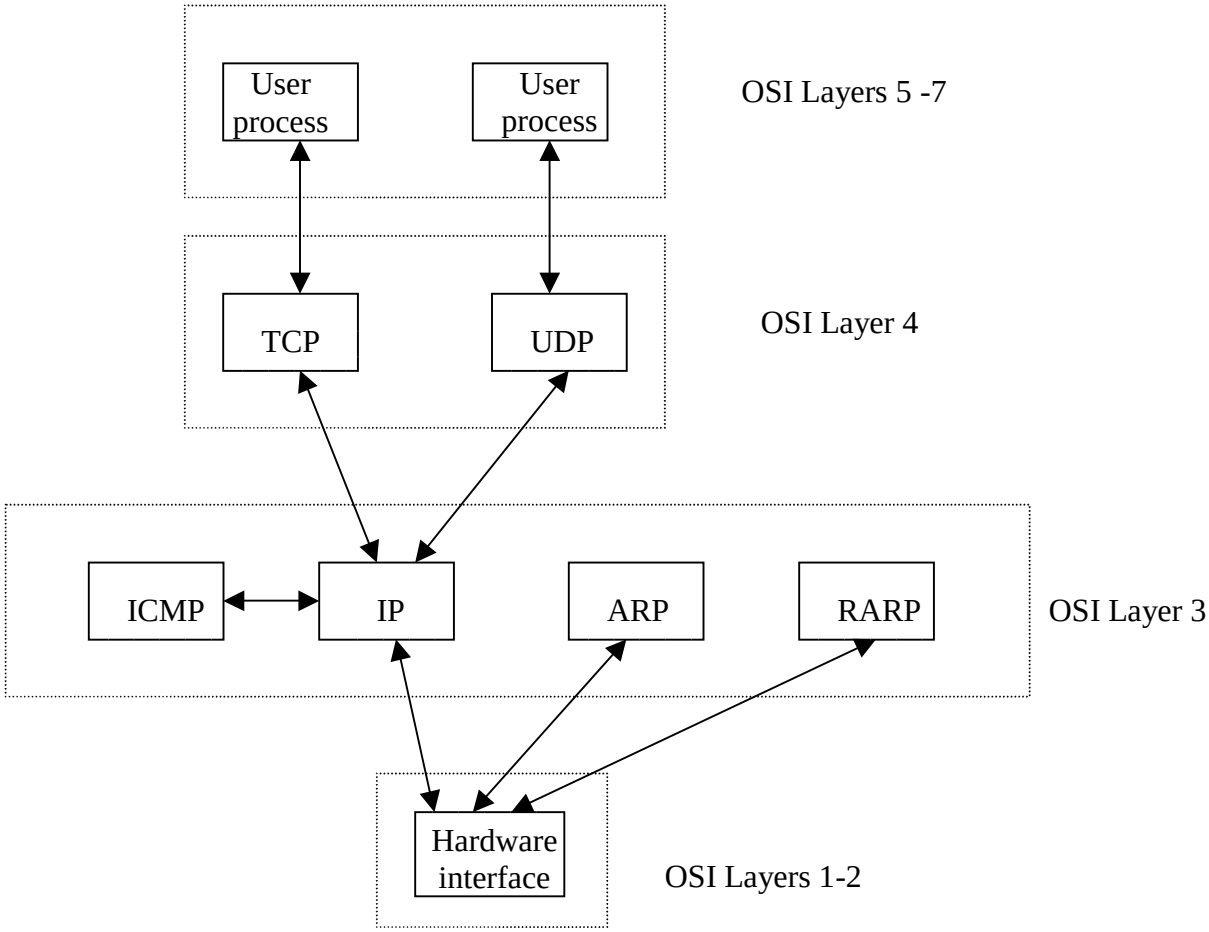
OSI model

Fig. 2.1

2b. Example Of implementation.

2b.i TCP/IP - OVERVIEW

Although the protocol family is referred to as TCP/IP, there are more members of this family than TCP and IP. Figure 2.2 shows the relationship of the protocols in the protocol suite with their approximate mapping into the OSI model.



Layering in the Internet protocol suite

Fig2.2

TCP T*ransmission Control Protocol.* A connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process. Most Internet application programs use TCP. Since TCP uses IP the entire Internet protocol suite is often called the TCP/IP protocol family.

UDP *User Datagram Protocol.* A connectionless protocol for user processes. Unlike TCP, which is a reliable protocol, there is no guarantee that UDP datagrams ever reach their intended destination.

ICMP *Internet Control Message Protocol.* The protocol to handle error and control information between gateways and hosts. While ICMP messages are transmitted using IP datagrams, these messages are normally generated by and processed by the TCP/IP networking software itself, not user processes.

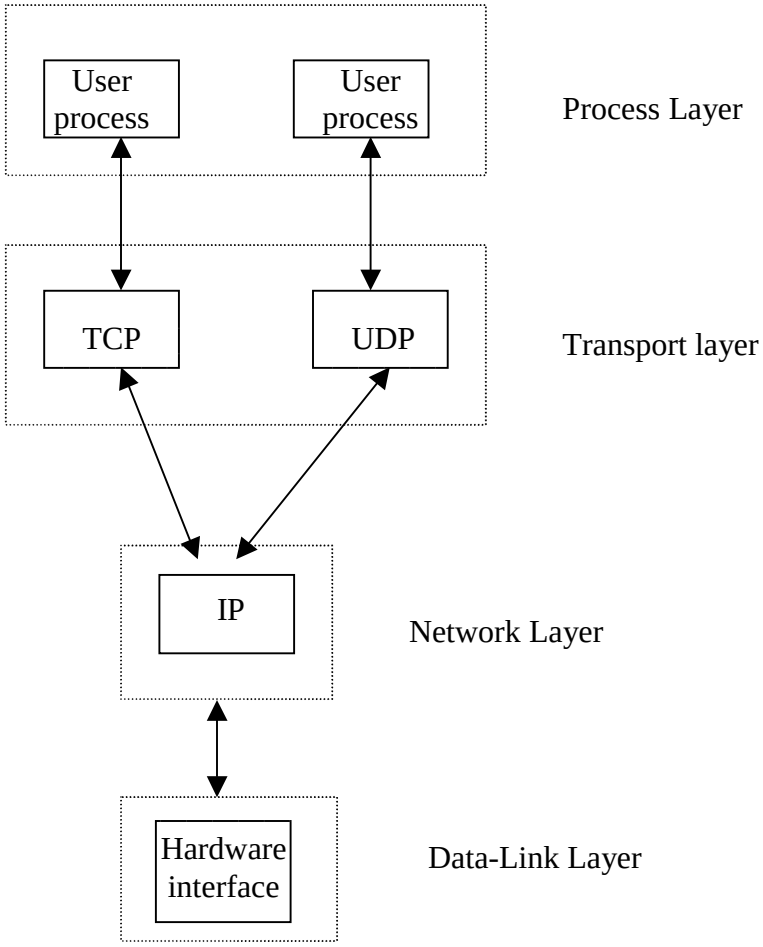
IP *Internet Protocol.* IP is the protocol that provides the packet delivery service for TCP, UDP, and ICMP. Note from figure 2.2 that user processes normally do not need to be involved with the IP layer.

ARP *Address Resolution Protocol.* The protocol that maps an Internet address into a hardware address. This protocol and the next, RARP, are not used on all networks. Only some networks need it.

RARP *Reverse Address Resolution Protocol.* The protocol that maps hardware address into an Internet address.

2b.ii Transport Layer – UDP and TCP

Use processes interact with the TCP/IP protocol suite by sending and receiving either TCP data or UDP data. The relationship of TCP and UDP to our simplified 4-layer model is shown in figure 2.3.



4 Layer model showing UDP, TCP, and IP

Fig2.3

These two protocols are sometimes referred to as TCP/IP or UDP/IP, to indicate that both use IP also.

TCP provides a connection-oriented, reliable, full-duplex, byte-stream service to an application program. UDP, on the other hand, provides a connectionless, unreliable datagram service. Figure 2.4 compares IP, UDP and TCP against the modes of service.

	IP	UDP	TCP
Connection oriented?	No	No	Yes
Message boundaries?	Yes	Yes	No
Data checksum?	No	Opt	Yes
Positive ack.?	No	No	Yes
Timeout and rexmit?	No	No	Yes
Duplicate direction?	No	No	Yes
Sequencing?	No	No	Yes
Flow control?	No	No	Yes

Fig2.4

2b.iii Comparison of protocol features for IP, UDP, and TCP.

Since the IP layer provides an unreliable, connectionless delivery service for TCP, it is the TCP module that contains the logic necessary to provide a reliable, virtual circuit for a user process. TCP handles the establishment and termination of connections between processes, the sequencing of data that might be received out of order, the end-to-end reliability (checksums, positive acknowledgements, and time outs) and the end-to-end flow control.

UDP provides only two features that are not provided by IP, port numbers and an optional checksum to verify the contents of the UDP datagram but these two features are enough reason for a user process to use UDP instead of trying to use IP directly, when a connectionless datagram protocol is required.

2c. Network Programming

In this section we describe some concepts of network programming we have used in our implementation.

2c.i Application Program Interfaces (APIs)

The API is the interface available to a programmer. The availability of an API depends on both the operating system being used, and the programming language. The two most relevant APIs for networking in Unix systems are Berkeley sockets and System V transport layer interface (TLI). Both of these interfaces were developed for the C language. In our implementation we have used the Berkeley socket interface.

Berkeley Sockets – Overview

The socket interface used here corresponds to 4.3 BSD VAX release from 1986. This release supported the following communication protocols.

- Unix domain – Unix domain protocols provide both connection oriented and connection less interface for Unix systems.

- Internet domain (TCP/IP) (refer section 2)

- Xerox NS domain – XNS is the network architecture that Xerox has published and is similar in structure to the TCP/IP protocol suite.

Figure 2.5 shows a time line of the typical scenario that takes place for a connection oriented transfer— first the server is started, then sometime later the client is started that connects to the server.

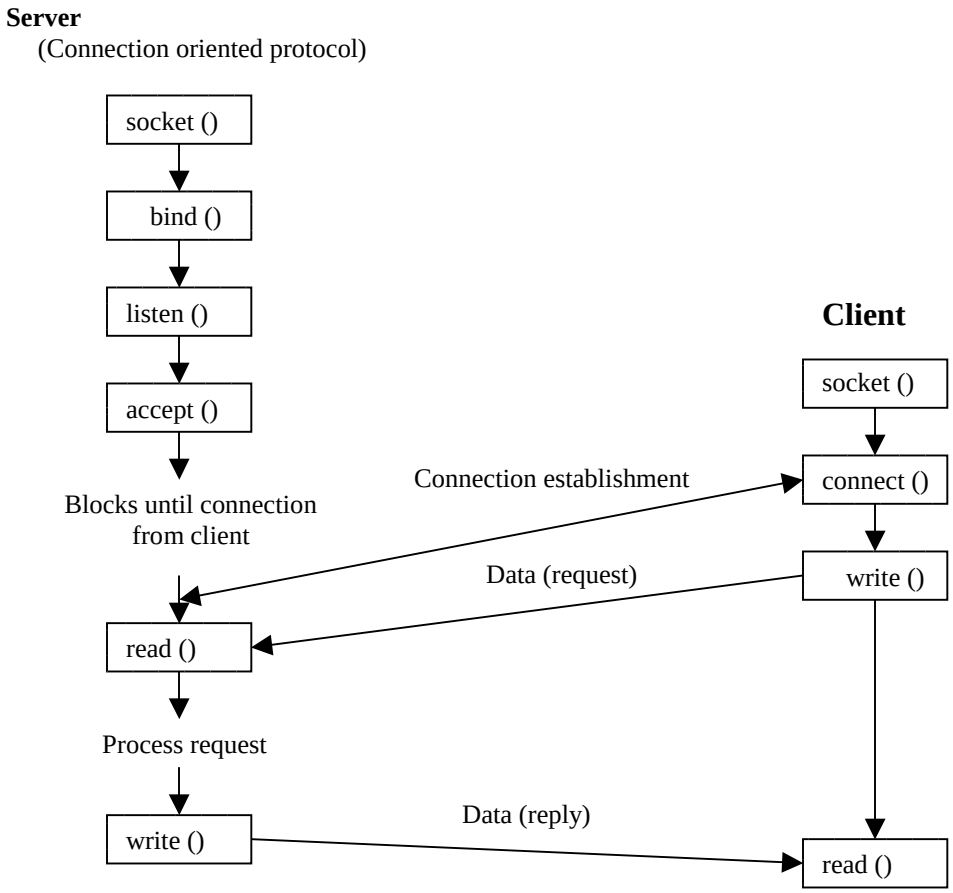


Fig2.5

Socket system calls for connection- oriented protocol

Figure 2.6 shows the system calls for a client server using a connectionless protocol. The client does not establish a connection with the server. Instead the client just sends a datagram to the server using the *sendto* system call, which requires the address of the destination (server) as a parameter. Similarly the server does not have to accept a connection from a client. Instead the server just issues a *recvfrom* system call that waits until data arrives from the same client. The *recvfrom* returns the network address of the client process, along with the datagram, so the server can send its response to the correct process.

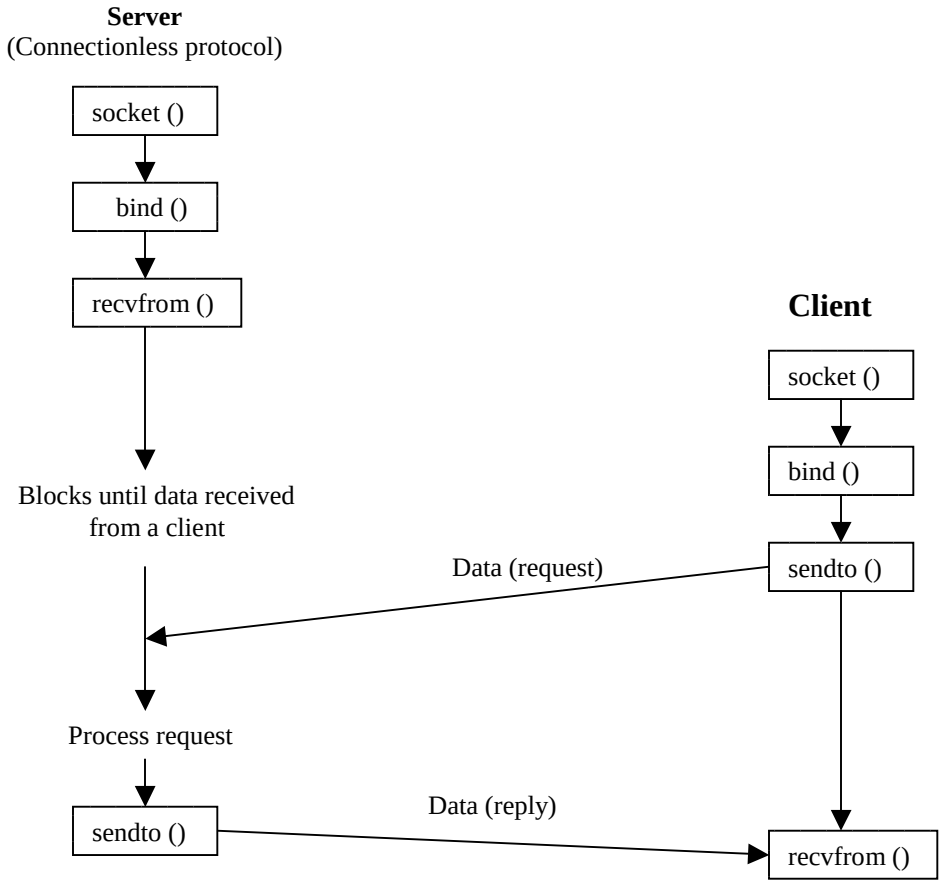


Fig2.6

Socket system calls for connectionless protocol

Elementary Socket System Call

socket System Call

To do network I/O first thing the process must do is call the socket system call specifying the type of communication protocol decided.

Eg: # include <sys/types.h>

include < sys/socket.h>

int socket (int family, int type, int protocol)

socketpair **system call**

This system call is implemented only for the Unix domain. It returns two socket descriptors sockvec [0] and sockvec [1] that are unnamed and corrected. The systems call similar to the pipe system call, but ***socketpair*** returns a pair of socket descriptors, not file descriptors. Additionally the two-socket descriptors returns by ***socketpair*** are bi-directional, unlike pipes, which are unidirectional.

Eg: # include <sys/types.h>
include < sys/socket.h>
int socketpair (int family, int type, int protocol, int sockvec [2]);

The *bind* system call

This assigns a name to an unnamed socket.

Eg: # include <sys/types.h>
include < sys/socket.h>
int bind (int sockfd, struct, sockaddr *miaddr, int addrlen)

the bind system call fills in the local-addr and local-process elements of the association 5-tuple.

connect **system call**

A client process uses this to establish a connection with the server. By connecting a socket descriptor following the socket system call

Eg: # include <sys/types.h>
include < sys/socket.h>
int connect(int sockfd, struct, sockaddr *servaddr,int addrlen)

for most connection oriented protocols the connect system call results in the actual establishment of a connection between the local system and the foreign system.

Messages are typically exchanged between the two systems and specific parameters relating to the conversation might be agreed on. In these cases the connect system call does not return until the connection is established, or an error is written to the process. A connectionless client can also use the connect system call, but the scenario is different from what was described. For a connectionless protocol all that is done by the connect system call is to store the servaddr specified by the process, so that the system knows where to send any future data that the process writes to the sockfd descriptor. Also only datagrams from this address will be received by socket. In this case the connect system call returns immediately and there is not an actual exchange of messages between the local network systems.

listen system call

A connection-oriented server to indicate that it is willing to receive connections uses this system call.

Eg: int listen(int sockfd, int backlog)

It is usually executed after both the socket and bind system calls and immediately before the accept system call.

accept system call

After a connection oriented server executes a listen system call an actual connection from some client process is awaited by having the server execute the accept system call

Eg: # include <sys/types.h>

include < sys/socket.h>

int accept (int sockfd, struct sock addr * peer, int *addrlen)

accept is the first connection request on the queue and created another socket with the same properties as sockfd. If there are no connection requests pending, this call blocks the caller until one arrives.

close **system call**

The normal Unix close system call is used to close the socket.

Eg: int close (int fd);

If the socket being closed is associated with a protocol that promises reliable delivery, the system must assure that any data within the kernel that still has to be transmitted or acknowledged is sent.

2c.ii Byte ordering routines

The following four functions handle the potential byte order differences between different computer architectures and different network protocols.

```
# include <sys/types.h>
# include < sys/socket.h>

  u_long htonl(u_long hostlong);
  u_short htons(u _short hostshort);
  u_long ntohl(u_long netlong);
  u_short ntohs(u_short netshort);
```

htonl	convert host – to – network , long integer
stons	convert host – to – network, short integer
ntohl	convert host – to – host, long integer
ntohs	convert host – to – host, short integer

Fig2.7

2c.iii Reserved Ports

There are two ways for a process to have an Internet port assigned to a socket.

- The process can request a specific port. This is typical for servers that need to assign a well-known port to a socket.
- The process can let the system automatically assign a port. Specifying a port no of 0 before calling bind requests the system to do this.

In Internet domain any TCP or UDP port in the range 1- 1023 is reserved. A process is not allowed to bind unless its effective user id is 0. (Super user).

4.3 BSD provides library function that assigns a reserved TCP stream socket to its caller:

int rresvport (int *aport);

This function creates an Internet stream socket and binds a reserved port to the socket. The socket descriptor is returned as the value of the function, unless an error occurs, in which case –1 is returned. The argument to this function is the address of an integer not an integer value. The integer pointed to by a port is the first pointer number that the function attempts to bind. The caller typically initializes the starting port number to IPPORT_RESERVED –1. (The value of the constant IPPORT_RESERVED is defined to be 1024 in < netinet/in.h>). If this bind fails with an errno of EADDRINUSE, then this function decrements the port number and tries again. If it finally reaches port 512 and finds it already in use, it sets errno to EAGAIN, and returns –1. If the function returns successfully, it not only returns the socket as the value of the function, but the port number is also returned in the location pointed to by a port. Figure 2.8 summarizes the assignment of ports

	Internet
Reserved ports	1-1023
Ports automatically assigned by the system	1024 – 5000
Ports assigned by rresvport ( )	512-1023

Fig2.8

The concept of reserved ports only handles the binding of ports to unbound sockets by the system. It is the application program (the server) that receives a request from the client with a reserved port, to consider the request as special or not. The server can obtain the address of the client using the getpeername system call. A connection oriented server can also obtain the clients address from the peer argument of the accept system call, and datagram server can obtain the address of the client from the from argument of the recvfrom system call.

2c.iv Daemon Process

A Daemon Process is a process that executes " in the background " (ie; without an associated terminal or login shell). Either waiting for some event to occur, or waiting to perform some specified task on a periodic basis.

Eg: A remote login program that allows users to login to one system from another system on the network, would have Daemon Process waiting for a request to come across the network for someone to login.

The different ways a daemon process can be started are:

1. During system startup
2. From system's / usr/lib/crontab file on a periodic basis
3. From a user's crontab on a periodic basis.
4. By executing the at command, which schedules a job for execution at some later time

5. From a user terminal as a foreground job or as a background job

Typical system daemons have the following characteristics.

- They are started once, when system is initialized
- Their lifetime is the entire time that the system is operating; usually they do not die and get restarted later.
- They spend most of the time waiting for some event to occur at which time they perform their service.
- They frequently spawn other process to handle service requests.

The following session looks at the details of initiating and executing daemon processes.

Close All Open File Descriptors

All unnecessary file descriptors should be closed. This especially applies to standard input, standard output, and the standard error, since they were probably inherited from the process that exceeds the daemon. The code fragment

```
# include < sys/param.h>
  for ( i= 0;i<NOFILE;i++)
  close (i);
```

will close all open files. The constant NOFILE defines maximum number of open files in the process.

Change Current Working Directory

The kernel for the life of the process holds the current working directory for a process open. The mounted file system to which the directory belongs cannot be unmounted while an active process has its current working directory in that file system. To allow the system administrator to unmount a file system , a daemon should execute

```
chdir(“ /”);
```

to change its current working directory to the root directory.

Reset the file access creation mask.

A process inherits its file access creation mask from its parent. A daemon should execute

umask(0);

to reset this mask. This prevents any files created by the daemon from having their access bits modified.

Run In the background

If a daemon is started from a login session without being placed in the background, the daemon will tie up the terminal while it is executing. If the daemon is started from a shell script without being placed in the background, the daemon will cause the processing of the shell script to stop until the daemon finishes. To avoid either of these the daemon should do a fork and have the parent exit, allowing the actual daemon to continue in the child process.

Disassociate from process group

Any process inherits its process group id from the process that execed it. By belonging to some process group, the daemon is susceptible to signals send to the entire process group. To prevent this from affecting a daemon, the daemon should dissociate from its inherited process group and form its own process group. The daemon calls the setpgrp system call to set its process group id equals to its process id, making the calling process a process group leader, in a new process group. The call is

setpgrp( );

Ignore Terminal I/O signals

On systems that support job control, you control the ability of a background to produce output on the control terminal with an *stty* option. If background writes are not allowed, the process is sent the SIGTTOU signal when it attempts to write to the control terminal. The best solution is to dissociate the daemon process from its controlling terminal.

The *inetd* daemon

On a system there could be several daemons in existence, just waiting for a request to arrive at a time. All these process are started at boot time from /etc/rc startup file , and each process does nearly identical startup tasks : create a socket, bind the server's well-known address to the socket, wait for connection, then fork. The child process performs the request while the parent waits for another request.

The inetd daemon simplifies this process by providing the following two features:

1. It allows a single process (inetd) to be waiting to service multiple connection requests, instead of one process for each potential service. This reduces the total number of processes in the system.

2. It simplifies the writing of the daemon process to handle the requests, since many of the startup details are handled by inetd. All the details to be handled by typical daemon, are dine by inetd, before the actual server is invoked.

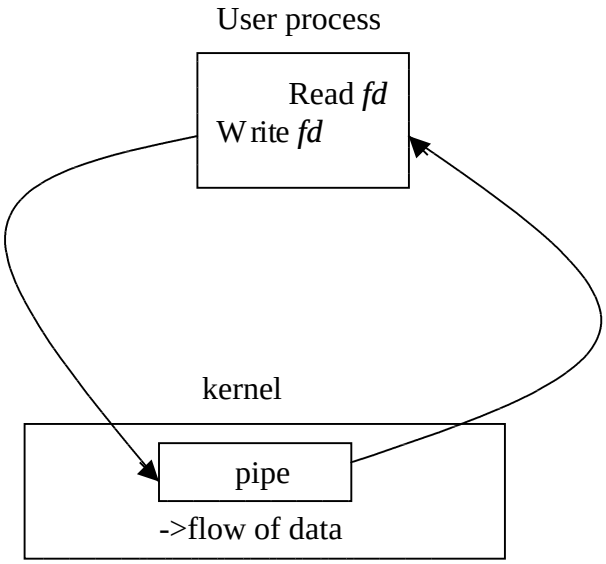
2c.v PIPES

A pipe provides a one-way flow of data. A pipe is created by the *pipe* system call.

int pipe(int *filedes);

Two file descriptors are returned-*filedes[0*] which is open for reading, and *filedes[1]* which open for writing.

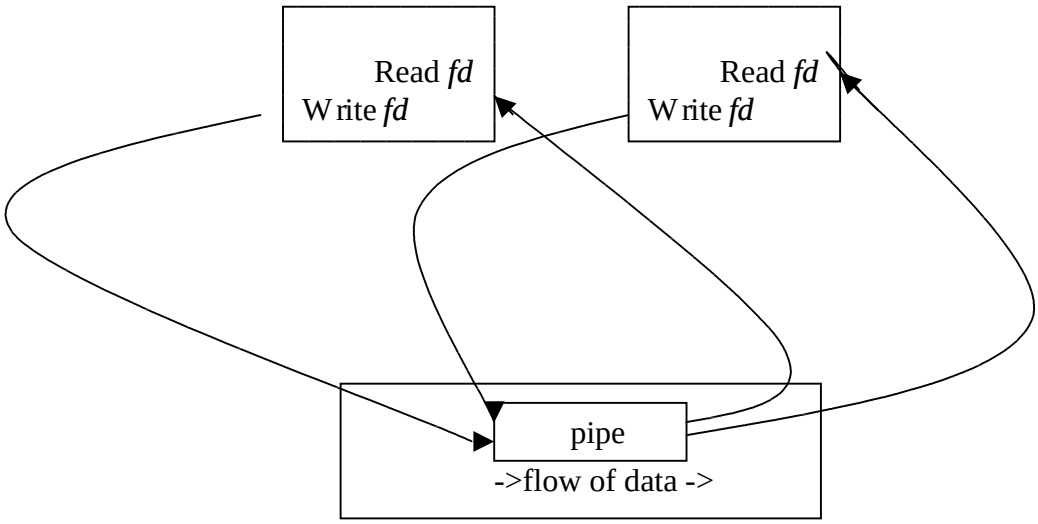
A diagram of what a pipe looks like in a single process is shown in figure 2.9. A pipe has a finite size always a least 4096 bytes.



Pipe in a single process

Fig2.9

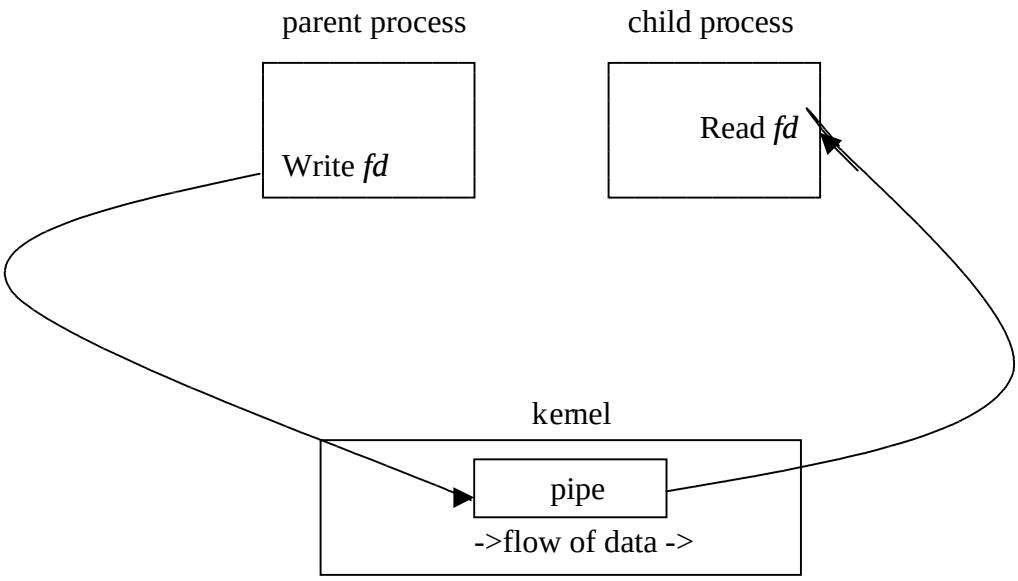
Pipes are typically used to communicate between two different processes in the following way. First, a process creates a pipe and then *forks* to create a copy of itself, as shown.



Pipe in a single process, immediately after fork

Fig2.10

Next the parent process closes the read end of the pipe and the child process closes the write end of the pipe. This provides a one0way flow of data between the two processes as shown in figure 2.11.



Pipe between two process

Fig2.11

When a user enters a command such as

Who | sort | lpr

to a Unix shell, the shell does the steps shown above to create three processes with two pipes between them. We show this pipeline as shown in figure 2.12.

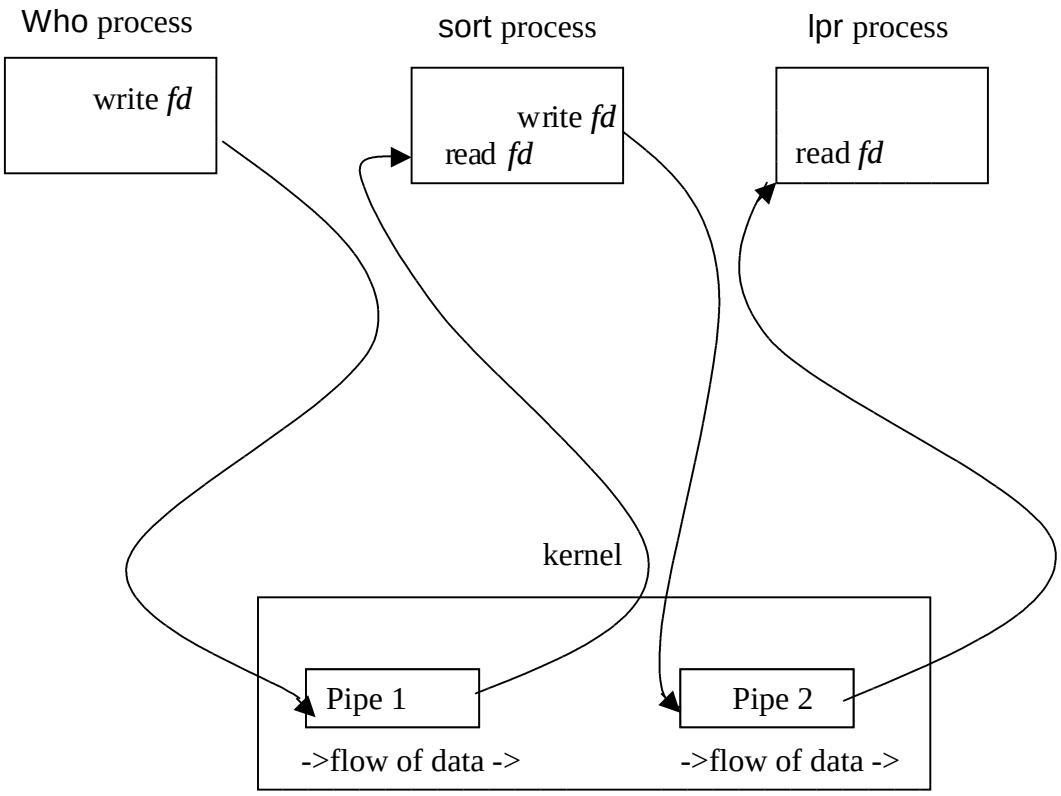


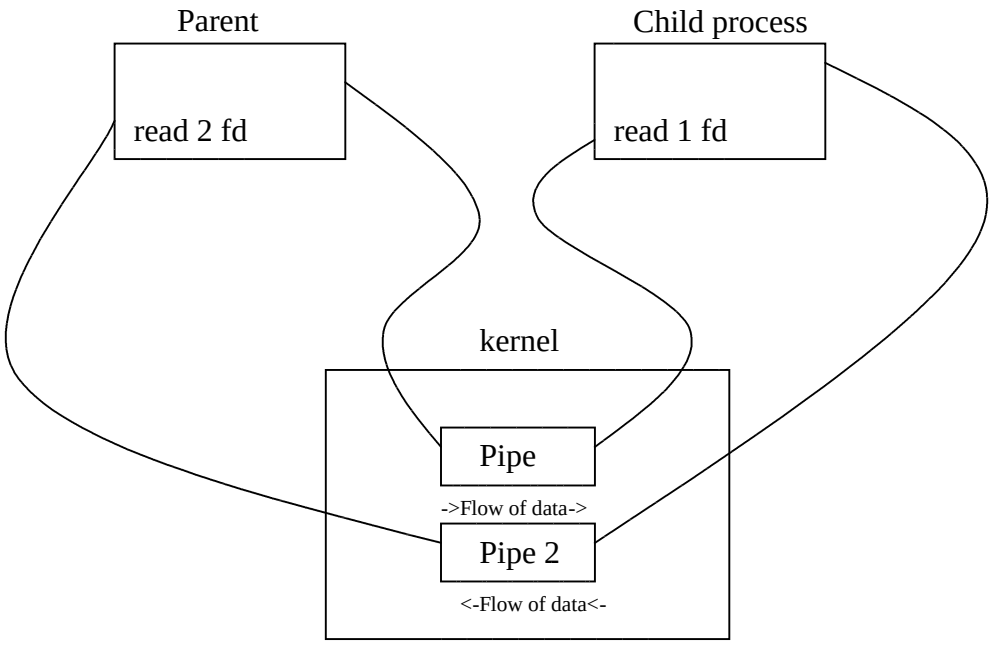
Fig2.12

When a two-way flow of data is desired, we must create two pipes and use one for each direction. The actual steps are

- Create pipe 1,create pipe 2

- Fork,
- Parent closes read end of pipe1,
- Parent closes write end of pipe2,
- Child closes write end of pipe1,
- Child closes read end of pipe2.

This generates a picture as shown.



Two pipes to provide a bidirectional flow of data

Fig 2.13

2c.vi Signals

A signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur asynchronously. Signals can be sent

- By one process to another process (or to itself)
- By the kernel to a process

  Every signal has a name, specified in the header file < signal.h>. Appendix C given summarizes the names of all signals along with their description and default action.

A process can do the following with a signal:

1. A process can provide a function that is called whenever a specific type of signal occurs. This function, called signal handler, can do whatever the process wants to handle the condition. This is called catching the signal.

2. A process can choose ignore a signal . All signals, other than SIGKILL, can be ignored. The SIGKILL signal is special, since the system administrator must have a guaranteed way of terminating any process.

3. A process can allow the default to happen as indicated in the column labeled default in the figure. Normally a process is terminated on receipt of signal with certain signals generating a core image of the process in its current working directory.

3. THE TFTP PROTOCOL

3a. Overview of the Protocol

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 timeout 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. An error is signaled 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 get it. Therefore timeouts 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), and losing access to a necessary resource (e.g., disk full or access denied during a transfer).

TFTP recognizes only one error condition that does not cause termination, the source port of a received packet being incorrect. In this case, an error packet is sent to the originating host.

3b. Relation to other Protocols

Since Datagram is implemented on the Internet protocol, packets will have an Internet header, a Datagram header, and a TFTP header. Additionally, the packets may have a header (LNI, ARPA header, etc.) to allow them through the local transport medium. As shown in Figure 3.1 , the order of the contents of a packet will be: local medium header, if used, Internet header, Datagram header, TFTP header, followed by the remainder of the TFTP packet.

The TFTP header consists of a 2 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 TFTP packets.

Order of Headers

Fig 3.1

3c. Initial Connection Protocol

A 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. Each data packet has associated with it a block number; block numbers 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.

In order to create a connection, each end of the connection chooses a TID for itself, to be used for the duration of that connection. The TID's chosen for a connection should be randomly chosen, so that the probability that the same number is chosen twice in immediate succession is very low. Every packet has associated with it the two TID's of the ends of the connection, the source TID and the destination TID. These TID's are handed to the supporting UDP (or other datagram protocol) as the source and destination ports. A requesting host chooses its source TID as described above, and sends its initial request to the known TID 69 decimal (105 octal) on the serving host. The response to the request, under normal operation, uses a TID chosen by the server as its source TID and the TID chosen for the previous message by the requestor as its destination TID. The two chosen TID's are then used for the remainder of the transfer.

3d. TFTP Packets

TFTP supports five types of packets, all of which have been mentioned above:

opcode operation

1 Read request (RRQ)

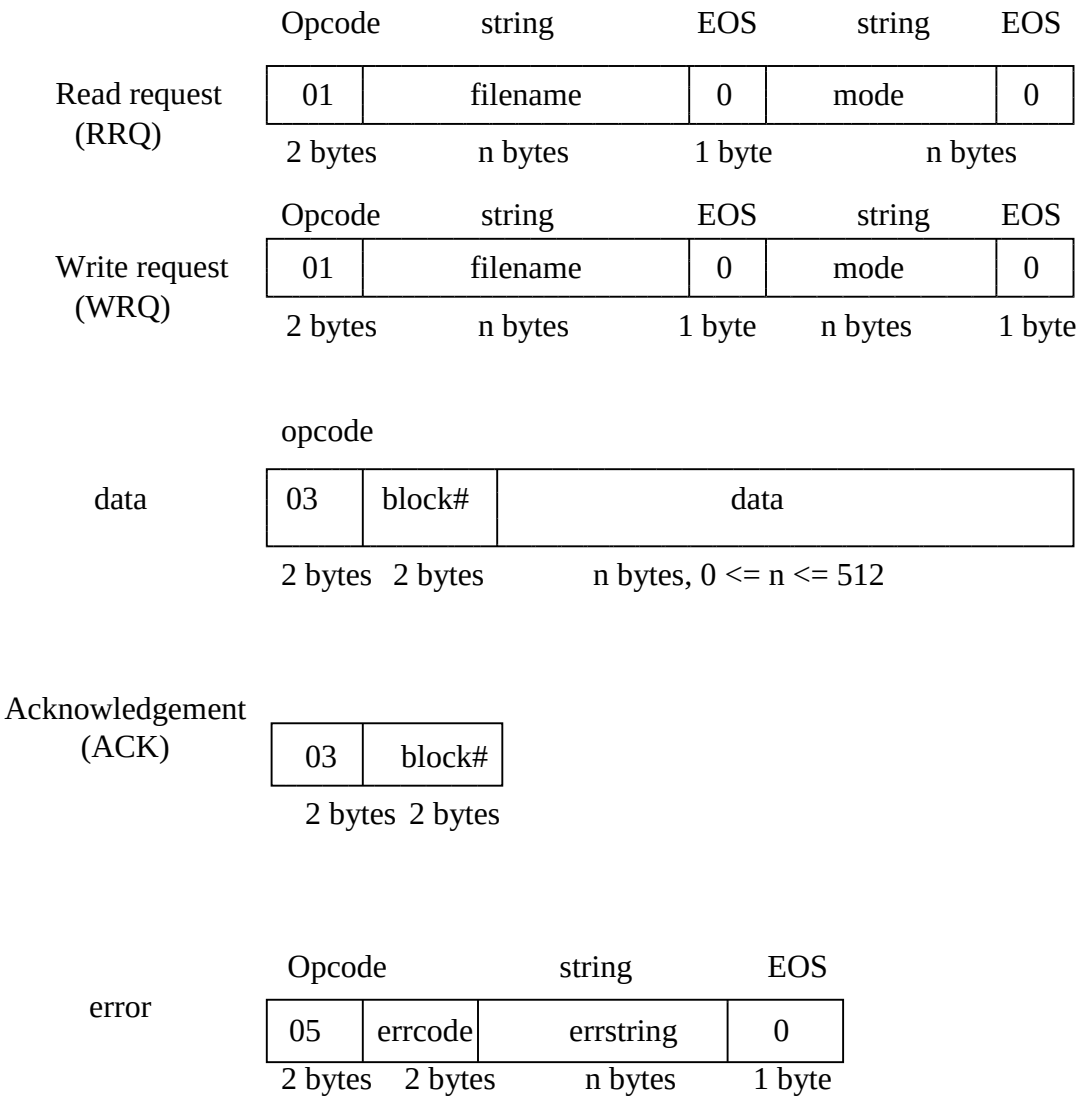
2 Write request (WRQ)

3 Data (DATA)

4 Acknowledgment (ACK)

5 Error (ERROR)

The TFTP header of a packet contains the opcode associated with that packet.



TFTP packet formats

Fig 3.2

RRQ and WRQ packets (opcodes 1 and 2 respectively) have the format shown in Figure 3.2. 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.) in netascii indicating the three

modes defined in the protocol. 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 8-bit format of the machine from which the file is being transferred. It is assumed that each type of machine has a single 8-bit format that is more common, and that that format is chosen. If a host receives a octet file and then returns it, the returned file must be identical to the original. Mail mode uses the name of a mail recipient in place of a file and must begin with a WRQ. Otherwise it is identical to netascii mode. The mail recipient string should be of the form "username" or "username@hostname". If the second form is used, it allows the option of mail forwarding by a relay computer.

The discussion above assumes that both the sender and recipient are operating in the same mode, but there is no reason that this has to be the case. For example, one might build a storage server. There is no reason that such a machine needs to translate netascii into its own form of text. Rather, the sender might send files in netascii, but the storage server might simply store them without translation in 8-bit format. Another such situation is a problem that currently exists on DEC-20 systems. Neither netascii nor octet accesses all the bits in a word. One might create a special mode for such a machine which read all the bits in a word, but in which the receiver stored the information in 8-bit format. When such a file is retrieved from the storage site, it must be restored to its original form to be useful, so the reverse mode must also be implemented. The user site will have to remember some information to achieve this. In both of these examples, the request packets would specify octet mode to the foreign host, but the local host would be in some other mode. No such machine or application specific modes have been specified in TFTP, but one would be compatible with this specification.

It is also possible to define other modes for cooperating pairs of hosts, although this must be done with care. There is no requirement that any other hosts implement these. There is no central authority that will define these modes or assign them names. Data is actually transferred in DATA packets depicted in figure 3.2.

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.

All packets other than duplicate ACK's and those used for termination are acknowledged unless a timeout occurs. Sending a DATA packet is an acknowledgment for the first 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 3.2 depicts an ACK packet; the opcode is 4. 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 5) takes the form depicted in Figure 3.2 . 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. The error message is intended for human consumption, and should be in netascii. Like all other strings, it is terminated with a zero byte.

3e. Normal Terminations

The end of a transfer is marked by a DATA packet that contains between 0 and 511 bytes of data (i.e., Datagram length < 516). This packet is acknowledged by an ACK packet like all other DATA packets. The host acknowledging the final DATA packet may terminate its side of the connection on sending the final ACK. On the other hand, dallying is encouraged. This means that the host sending the final ACK will wait for a while before terminating in order to retransmit the final ACK if it has been lost. The acknowledger will know that the ACK has been lost if it receives the final DATA packet again. The host sending the last DATA must retransmit it until the packet is acknowledged or the sending host times out. If the response is an ACK, the transmission was completed successfully. If the sender of the data times out and is not prepared to

retransmit any more, the transfer may still have been completed successfully, after which the acknowledger or network may have experienced a problem. It is also possible in this case that the transfer was unsuccessful. In any case, the connection has been closed.

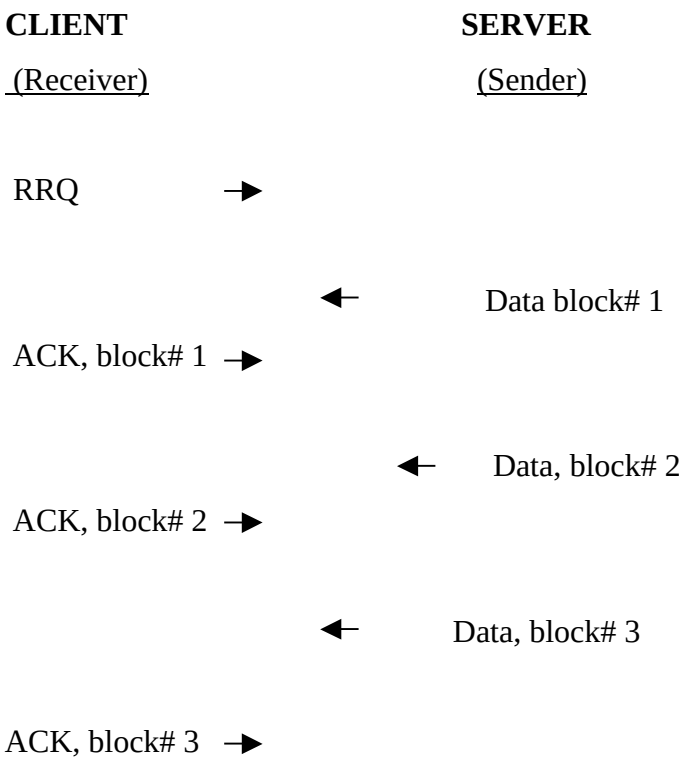
3f. Premature Termination

If a request cannot be granted, or some error occurs during the transfer, then an ERROR packet (opcode 5) is sent. This is only a courtesy since it will not be retransmitted or acknowledged, so it may never be received. Timeouts must also be used to detect errors.

3g. Communication between Server and client

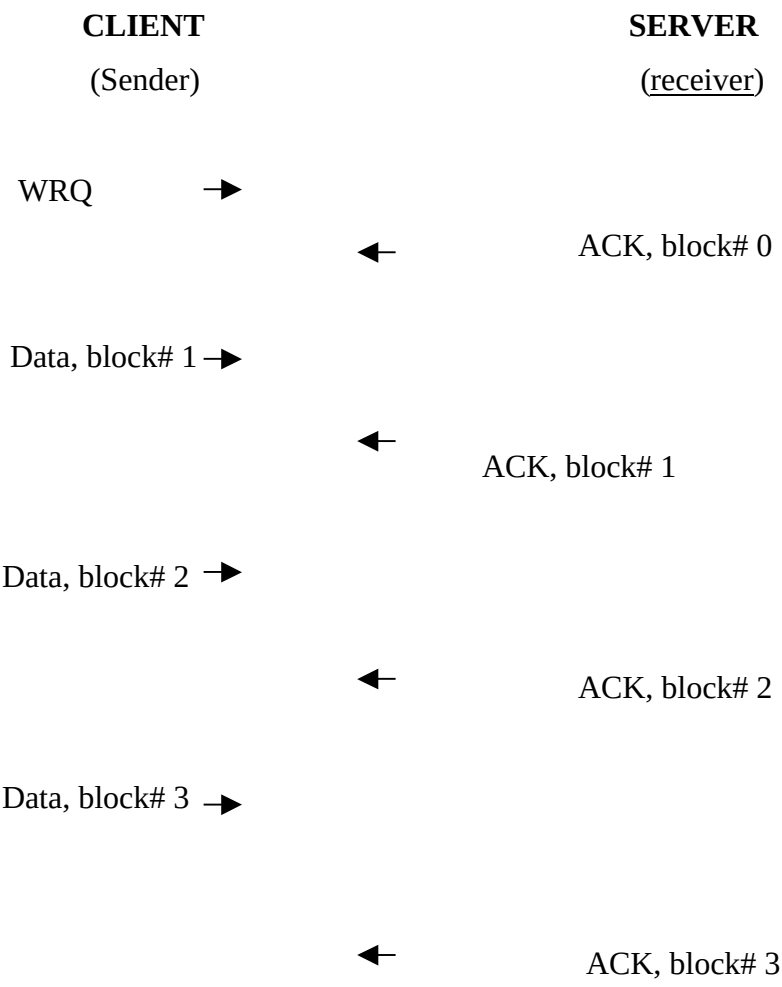
All the 2-byte fields in these packets, the *opcode, block#,* and *errcode,* are stored in network byte order.

1. The client asks to receive a file from the server.



With TFTP we use the term receiver to designate the end that is receiving the data packets, and the term sender for the end that is sending the data packets.

2. The client asks to send a file to the server.



The protocol handles lost packets by having the sender of the data packets use a timeout with retransmission. If a data packet is lost, the sender eventually times out when an acknowledgement is not received, and transmits the packet. Note that either the client or the server can be the sender of packets, depending

upon whether the client has issued a read request or a write request. If an acknowledgement packet is lost, the sender of the data packets still times out and retransmits the data packet. In this case the duplicate, so it ignores the duplicate data packet and retransmits the acknowledgement. For TFTP, the block number in every data packet and in every acknowledgement packet serves to detect missing or duplicate packets.

Most errors other than timeout cause termination of the process. When one side sends an error packet, the other side does not acknowledge it, nor does the side that sent the error packet retransmit it. The most common type of errors are "File not found" for an RRQ request, and "Access violation" for WRQ request. Once a file transfer is in progress, it is possible to get the error "Disk full or allocation exceeded." Most other errors signify some fatal condition that aborts the transfer.

Note that once the initial RRQ or WRQ packet is sent by the client, the remainder of the protocol is symmetrical between the client and the server. Both can send and receive data packets, acknowledgements, and error packets. We 'll take the advantage of this fact n our implementation to use as much of the source code as possible in both client and server.

If both the sender and receiver use a timeout with retransmission, and both retransmit whatever they last sent, a condition termed the source's apprentice syndrome results. Fig 3.3 shows a time line of packets between a client and server.

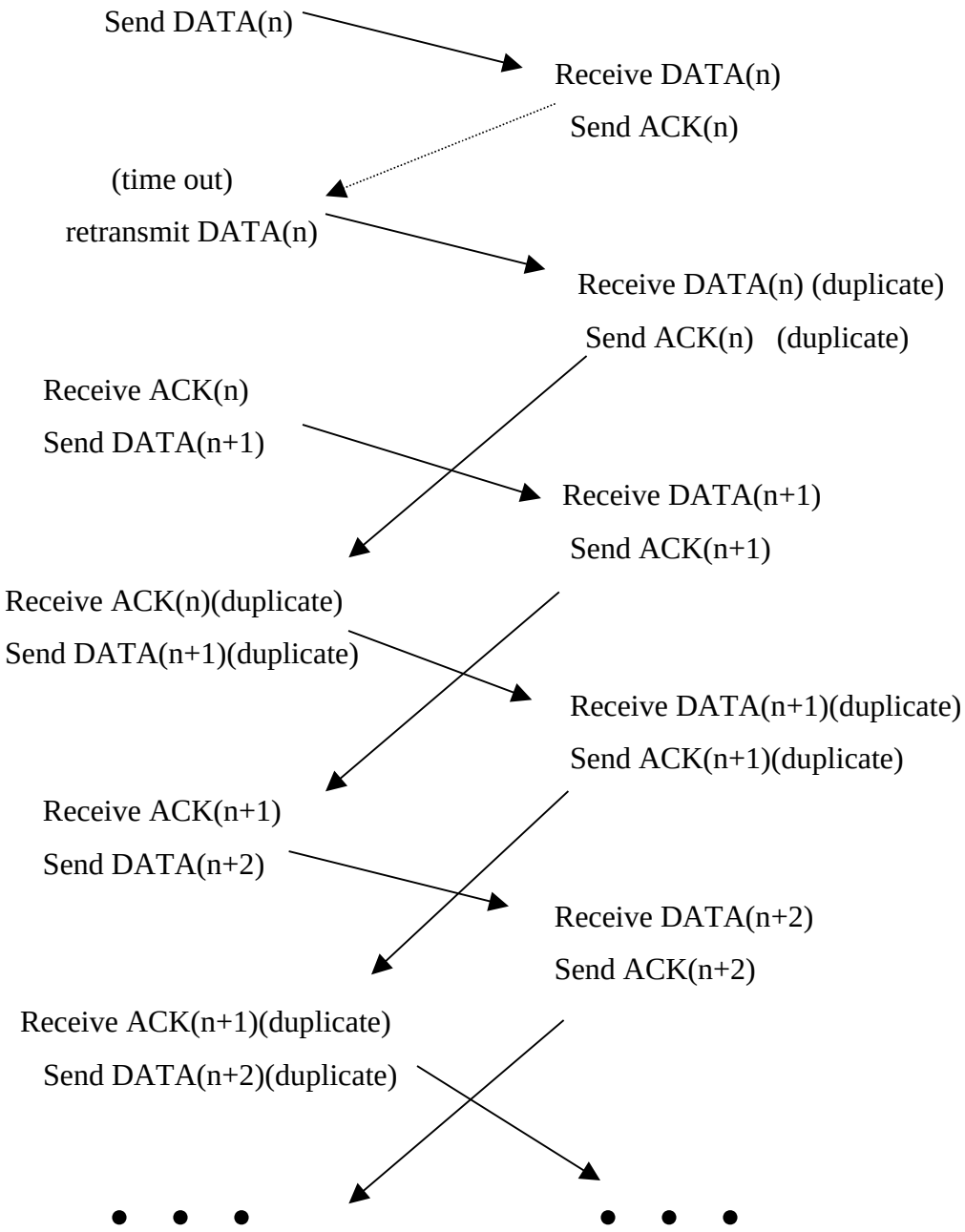


Fig 3.3

Assume that the ack(n) sent by the receiver is delayed somehow in the network. We show this as the dashed arrow in figure3.3. What results is that every data packet and every acknowledgement from that point on is sent twice. The file is still transferred correctly, assuming the remainder of the transfer doesn't abort for some other reason. The correction for this syndrome is for the sender never to retransmit a data packet if it receives a duplicate acknowledgement. If we implement this change, the new time line is shown in figure 3.4 .

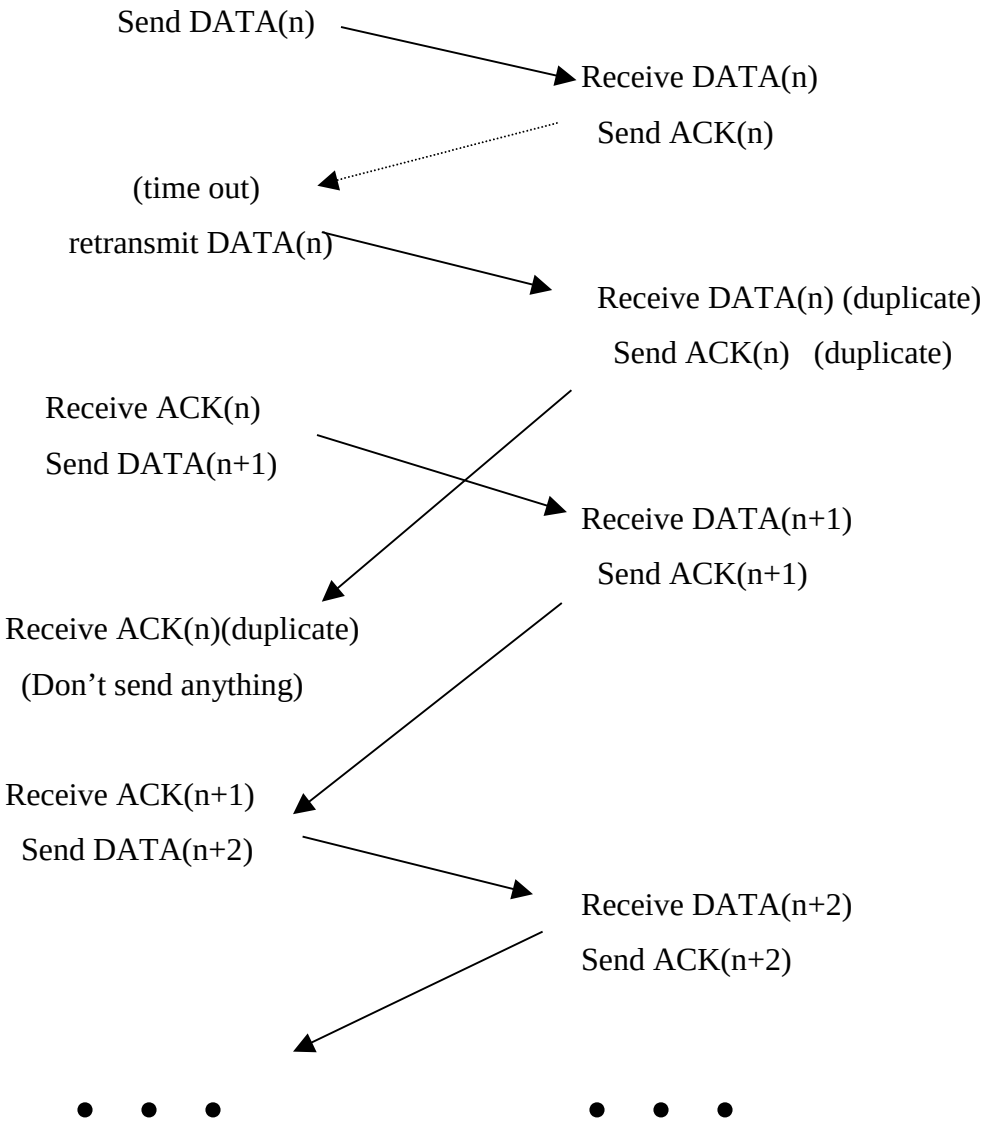


Fig 3.4

An effect of this correction is that the receiver of the data packets does not need a retransmission timer. This can simplify the implementation of TFTP if it is being used as a bootstrap program. When TFTP is used as a bootstrap program, it would only issue RRQs and would always be the receiver of data.

3h. SECURITY.

Our strategy is for the server only to allow read access (RRQ requests) for files that have read access to all users. Our server also restricts write access (WRQ requests) to those directories that have write access to all users. Also note that the sever is running as a detached process that specifically sets its home directory to '/' (the root directory). It therefore requires that all filenames requested by a client be fully qualified pathnames that begin with a slash.

TFTP server is started by the *inetd* daemon. Before the *inetd* superserver *execs* the TFTP server, it first sets the login name to *nobody.* This is an entry in the password file that has a numeric user ID that is not of 32767, a value that no normal user has. This entry also specifies a numeric group ID that is not normally used (9999). This is done to assure that server didn't specify this login name in the *inetd.conf* file, it would run with superuser privileges. The intent of TFTP is to provide a simple file transfer program, but its use should be restricted to cooperating host systems.

3i. DATA FORMATS.

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 the server. The standard ASCII 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.

The octet data format is used for transferring binary files. An octet is an 8-bit quantity, which call as a byte. There are two primary uses of TFTP to transfer binary data between systems. First, two systems with the same architecture can obviously exchange a binary file without any problems. Second if a system receives a binary file and then returns it to 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 fetch a binary file, the actual contents of the file wouldn't change.

3j. CONNECTIONS.

TFTP is unique in its handling of a "pseudo" connection between the client and server. TFTP is based on UDP, but UDP is a connectionless protocol. Few files are small enough to be transferred in a single packet, so there must be some way for the client and server to establish a connection between themselves. This allows the multiple packets that comprise a typical file to be handled between a single invocation of a client and server.

Since the server is listening for a UDP packet on a well-known port, the server is initially contacted by the client on that port. But multiple clients can contact the server on that well-known port during the time it takes the server to transfer a single file with some client. We need a concurrent server. But the child process that the server spawns can't continue using the well-known port number. The technique use by TFTP is as follows:

1.The client ***binds*** a local address that forms a unique socket 3-tuple (protocol, local-address, and local-process) on the client's system.

2. The client sends its first datagram (an RRQ or a WRQ request) to the server at the server's well-known port. This datagram contains the client's address from the previous step, so that the server knows where to send its response.

3. The server receives the datagram from the new client and spawns a child process to handle the request. This child process *bind*s a local address on its system that forms a unique socket 3-tuple.

4.The child process sends its response to the RRQ or WRQ packet that the client sent. This datagram is sent to the client's address from step 1 above. The return address in the datagram is the new port that the child process obtained in step 3.

5. The client and server continue exchanging packets, with the client sending packets to the server's address from step 3.

This technique provides a connection between the client and server. It is not a true connection, in the sense of a TCP connection but it is a pseudo-connection that is handled by the two application processes. This protocol requires that the client look at the return address in the first received response from the server, and use that address as its destination address in all future exchanges with the server.

The commands provided to an interactive user are:

Connect host [port]

Set the name of the host for future transfers. This command is optional, since the name of the other host systems can be specified in both the *get* and *put* commands. The host can be either a dotted-decimal number, or a name. An optional *port* number can also be specified, but its use depends on the underlying network protocol being used. In a typical UDP implementation, this *port* specifies the UDP port to use to contact the server, instead of the default UDP port of 69.

Mode transfer-mode

Set the mode for file transfer. The *transfer-mode* must be either *ascii* (for a net-ascii transfer) or *binary* (for an octet transfer). The default file transfer mode is ascii.

Binary

Set the mode for file transfer to binary (octet). This command is shorthand for *mode binary*.

Ascii

Set the mode for file transfer to ascii (neasciit). This command is shorthand for *mode ascii.*

Get remotename localname

Get a file from the server. The remotename can be either the name of a file, in which case the name of the host is taken from the most previous *connect* command, or it can be of the form *host:filename* to specify both the name of the host and the name of the file. The *host* can be either a dotted-decimal number, or a name. The mode for the file transfer depends on the most previous *mode* command.

Put localname remotename

Send a file to the server. The remotename can be either the name of a file, in which case the name of the host is taken from the most previous *connect* command, or it can be of the form *host:filename* to specify both the name of the host and the name of the file. The *host* can be either a dotted-decimal number, or a name. The mode for the file transfer depends on the most previous *mode* command.

Exit

Terminate the program. Equivalent to the *quit* command.

Quit

Terminate the program. Equivalent to the ***exit*** command.

Trace

Toggle the packet tracing facility. By default the packet trace is disabled. It can be enabled either by this command or by the –t command line option when the ***tftp*** client program is started

Verbose

Toggle the verbose option. By default the packet trace is disabled. It can be enabled either by this command or by the –t command line option when the ***tftp*** client program is started

status

shows the current status of the program.

Help

Print a 1-line summery of each command. Equivalent to the ? command.

?

Print a 1-line summery of each command. Equivalent to the ***help*** command.

3k. UDP IMPLEMENTATION.

The TFTP program consists of both a client program and a server program. Since the client and server protocols are identical, once a file transfer is started, the same source code files and functions are used whenever possible. For the UDP version, the files are listed in fig 3.5.

File name	client	server	notes
cmd.c	yes		only client processes user Commands
cmdgetput.c	yes		only client processes user Commands
cmdsubr.c	yes		only client processes user Commands
file.c	yes	yes	client and server file I/0
fsm.c	yes	yes	finite state machine to drive system.
Initvars.c	yes	yes	initialize all variables
Maincli.c	yes		client main function
Mainserv.c		yes	server main function
Net udp.c	yes	yes	client & server n/w I/O
Sendrecv.c	yes	yes	client &server TFTP funcs

Fig 3.5

3l. Implementation

The actual implementation of the code is explained in the proceeding section. The different files are given and their explanations given alongwith.

1. *defs.h* :- This is the file containing all the global definitions.

2. *netudp.c* :- This file defines five functions that are used by both the client and server.

net_init	initialize a network connection
net_open	open a network connection.
net_close	close a network connection.
net_send	send a packet
net_recv	receive a packet

.

Fig 3.6

3. *maincli.c* :- The file *maincli.c* contains the main function for the client. The while loop at the bottom of the *mainloop* function calls the *docmd* function to process every user command.

4. *cmd.h* :- The cmd.h is a header file that is used by the files *cmdsubr.c* and cmd.c mentioned later.

5. *cmdsubr.c* :- The file *cmdsubr.c* contains the miscellaneous functions for command processing . The function *docmd* that is called from the main loop is in this file.

6. *cmd.c*:- The file *cmd.c* contains one function for every user command. These functions are called by the *docmd* function from the previous file.

7. *cmdgetput.c* :- The file *cmdgetput.c* has additional processing for the *get* and *put* commands.

8. *file*.c :- The file *file.c* handles all the Unix file I/O. This includes any required conversions between the TFTP formats, netascii and octet, and the Unix file format.

9. *fsm.c* :- The file *fsm.c* contains the finite state machine that drives the protocol processing. When a user enters either a get or a put command, the functions do_get and do_put that we showed in the file *cmdgetput.c* send the first packet to the server. Then the function fsm_loop is called to do the rest of the protocol processing. The fsm_loop function is also called by the server's main function to do all its protocol processing.

The finite state machine in the following file can be represented as a table; similar to the state transition matrices. The table for the client is shown below.

Packet type sent	Packet type received				
	RRQ	WRQ	DATA	ACK	ERROR
RRQ			•		•
WRQ				•	•
DATA				•	
ACK			•		
ERROR					•

Fig 3.7

Table for the server is shown below.

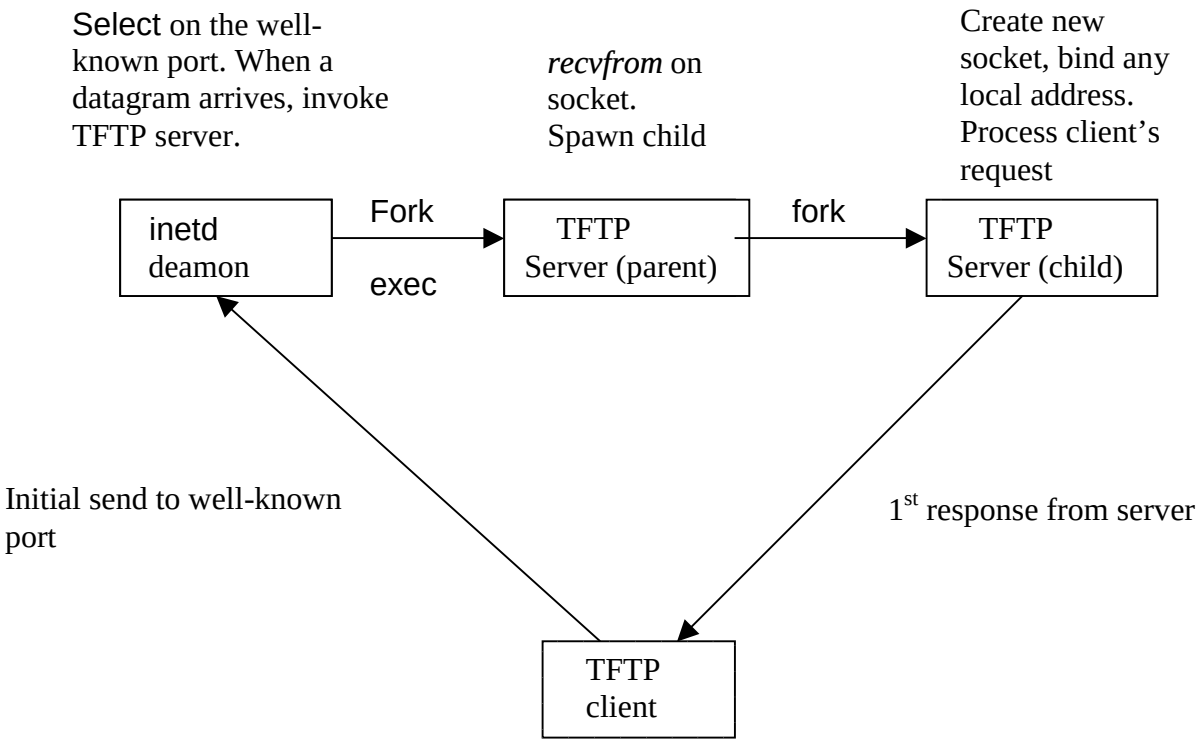
Packet type sent	Packet type received				
	RRQ	WRQ	DATA	ACK	ERROR
(nothing)	•	•			
RRQ					
WRQ				•	
DATA			•		
ACK					
ERROR					

Fig 3.8

For the server we have added another row to handle the initial state-the first packet that the server receives. We use the identifier DATAGRAM to include the code required for a datagram protocol such as UDP.

10. *sendrecv.c* :- The file *sendrecv.c* contains all the functions that send and receive packets to the peer process. Some of these functions are used by the client or server, and both use some.

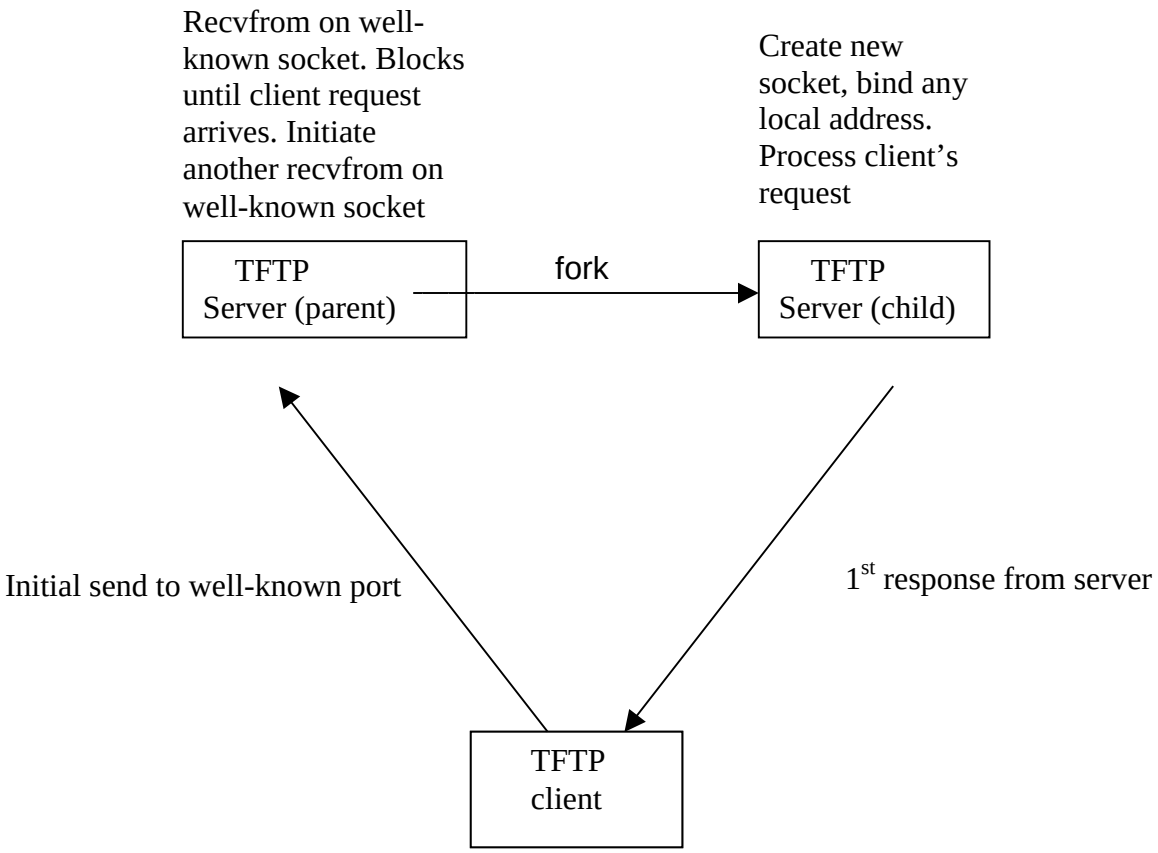
11. *mainserv.c* :- The file *mainserv.c* contains the main function for the server. The server is complicated by allowing it to be started either by the 4.3BSD *inetd* supervisor, or independently. When started by the *inet*d daemon(refer section 2c.iv) , the wait mode is specified, as we described. Once the TFTP server has read the datagram from the client on its well-known port, we want the TFTP server to fork with the parent exiting. This allows the *inetd* daemon to start another read on the server's well-known port to process the next request from some other client. The sequence of steps shown below.



TFTP server when invoked by inetd.

Fig 3.9

When the server is started independently of the *inetd* superserver, the TFTP server must provide the concurrency. The TFTP parent must start another *recvfrom* after a child process is spawned. Here we have the processes shown.



TFTP server when run as daemon

Fig 3.10

12. *initvars.c* :- The file *initvars.c* declares all the global variables and initializes them.