Network Programming Assignment #3
ECEN 602, Fall 2017
Due November 3, 2017 NLT 5:00 pm

Introduction

In this assignment you will implement a Trivial File Transfer Protocol (TFTP) server. TFTP is a simple method of transferring files between two systems that uses the User Datagram Protocol (UDP). You will not be implementing the TFTP client. Instead, you will be using a standard TFTP client. Your server must be capable of handling multiple simultaneous clients. TFTP takes much less code to implement than the File Transfer Protocol (FTP), which has many more features and supports user authentication. TFTP was originally designed to provide a network boot capability for diskless workstations, and it is still used today to transfer firmware images to devices like VoIP phones, routers, and switches. One of the major limitations of TFTP is that it has no user authentication.

While TFTP can transfer files in both directions (i.e., server to client and client to server), your server only has to implement reading files using the Read Request function (RRQ). The basic assignment is worth 100 points, but you can earn up to 30 additional points by including support for writing files to the server using the Write Request function (WRQ) (see Bonus Feature section below). The server should allow retrieval of files in the local directory from where the server is executed. *You will want to create a working directory with your test files and server executable that is separate from your home directory so that someone cannot access your home directory via TFTP. Also, be sure to terminate your TFTP server when you are not testing for the same reason.*

To test your server, you are to use an existing TFTP client. Because of the lack of user authentication in TFTP, most system managers disable the native TFTP server and client by default. The ECE Linux system managers have installed the TFTP client on both the apollo and hera Linux clusters for you to use in your testing. There are also native TFTP clients for both the Windows and Mac operating systems, but neither native client appears to support the port option, which we need to connect to something besides the well-known port for TFTP (i.e., port # 69). There are also free TFTP clients you can download for Windows and MacOS that support the port option, although if you do this be careful that you download a reputable one and check it for viruses. Another alternative is to install a virtual machine (e.g., Oracle VirtualBox, VMware Player, or Cloud 9) in order to run Linux and then use the native Linux TFTP client. The Teaching Assistant will go over these options with you if you are interested. The easiest approach will be to test using the TFTP client on apollo and hera.

TFTP is implemented over UDP, which is a connectionless protocol; consequently, there are very few guarantees. You will have to resolve these issues with the protocol's retransmission mechanism. TFTP is basically a Stop-and-Wait protocol with timeout similar to what we discussed in class.

Assignments must be written in C or C++, and they are to be compiled and tested in a Linux environment. Because the goal of the exercise is to understand system calls to the socket layer, you are prohibited from using any socket "wrapper" libraries; however, you may use libraries for simple data-structures. It is also acceptable, to use the Unix Network Programming, Vol. 1, 3rd Edition "wrappers" for the basic networking function calls (e.g., `socket`, `bind`, `listen`, `accept`, `connect`, `close`, etc.…). These "wrapper" functions check for error returns from the network functions (see the Supplementary References handout).

Protocol Specification

The TFTP protocol specification is given in IETF RFC 1350. There are a number of later RFP's related to TFTP that describe extensions to the protocol. You are not required to support any of these extended options. The RFC is clear on most details of the protocol, but this section includes clarifying information on the following: (1) Protocol Overview, (2) Timeout, (3) Modes of transfer, and (4) Sequence number wrap-around and the maximum file size.

1. Protocol Overview

TFTP uses UDP. The *Implementation Notes* section below discusses the differences between UDP and TCP socket programming. The well-known socket for the TFTP server is port number 69, but your server will obviously need to use another port number you specify on the command line starting the TFTP server process. Your TFTP server is required to support multiple simultaneous connections from clients, so you will need to call `fork()` to create a child process to service each client connection like you did in for the TCP Echo Server. Unlike the Echo Server, however, you will need to create a new ephemeral port for the server to use in the child process since UDP, unlike TCP, is not connection oriented. Creation of a new ephemeral port in the child process is described in the *Implementation Notes* section.

Below are the steps in a TFTP RRQ transfer (the WRQ is similar with the roles of the sender and receiver reversed):

a. The TFTP client sends a RRQ to the TFTP server on the port number you have chosen for the server. The request contains a filename and the transfer mode.
b. The TFTP server receives the client RRQ packet and then creates a child process with a call to `fork()`. First, the child server process closes the socket that is bound to the server's well-known address since the parent process will handle further requests that arrive there from new clients. Next, the child process creates a new socket and calls `bind()` with a system assigned ephemeral port number. All future communications from the child server process to this client will use the new socket (see additional details in the *Implementation Notes* section). The child process then responds directly with the first DATA packet. The DATA packets are numbered, and the first block is numbered 1. All DATA packets, except for the last, contain a full-size block of 512 bytes.
c. The TFTP client receives the first DATA packet. Before responding, the TFTP client must determine the new ephemeral port number for the server and use that as the destination port number for all packets sent to the TFTP child server (see additional details in the *Implementation Notes* section). The client sends an ACK with a block number of 1 in response to the first block (Note: This is different from the protocols we considered in class where RN = *next* packet number expected). (Aside: A nice touch in the client is to check future packets to make sure they came from the same server ephemeral port number).
d. File transfer proceeds similar to the Stop-and-Wait protocol we discussed in class. The TFTP server must receive a correctly numbered ACK before it can send the next DATA packet. The TFTP server, as the sender of DATA packets in a RRQ, is responsible for setting a timeout timer and retransmitting the current DATA packet in the event of a timeout, which indicates that a DATA or ACK packet was lost or damaged. It is important that UDP checksums be enabled in order to detect errors.
e. The final DATA packet must contain less than a full-sized block of data to signal the client that this is the last packet. If the size of the transfer is an exact multiple of the block-size, the final DATA packet will contain 0 bytes. On receiving an ACK for the final packet, the TFTP child process should clean up and exit.

2. Timeout

The sender of a data packet sets a timer when a block is first sent, and it retransmits the data block on timeout. When the server is responding to a Read Request (RRQ) from the client, it will be the server that implements the timeout since it is the one sending the DATA packets; whereas, the client will be sending

ACK's. When the server is responding to a Write Request (Bonus Feature), however, it will be the client that implements the timeout since it is sending DATA packets; whereas, the server will be sending ACK's. The *Implementation Notes* section below, includes information on two approaches to setting a timer and servicing timeouts.

The receiver of a data packet sends an ACK in response to a correctly received data packet. The sender must receive an ACK for the DATA block just sent before it can send the next block of data. If an ACK is lost, the sender will eventually timeout and retransmit the last DATA packet. The receiver discards a duplicate DATA packet and retransmits the ACK. If the sender receives a duplicate ACK, it does not retransmit the DATA packet (it will eventually timeout and retransmit if recovery is needed). The block numbers in DATA and ACK packets are 16-bits in length. From our earlier work with Stop-and-Wait, we know that only one-bit send and receive sequence numbers are necessary, but here we have 16-bit Stop-and-Wait sequence numbers.

The original version of the TFTP protocol used timeouts at both the sender and receiver, and each side retransmitted what they had last sent on timeout. This original version of the protocol was subject to the *sorcerer's apprentice* problem we looked at in Homework #4 problem 2 (P&D 2.28). The fix to avoid this problem was to have a single timeout and for the sender to never retransmit a data packet if it receives a duplicate ACK (If you look back at Lecture 9 slide #21, you will see that the stop-and-wait protocol we studied worked this same way).

3. Modes of Transfer

There are two forms of transfer supported by TFTP: netascii and octet. The netascii format is used for transferring text files, and the octet format is used for transferring binary files. The mode is sent in the RRQ or WRQ packet as a text string (in any combination of lower or upper case characters) and is terminated by a null (0) byte.

The netascii format uses the standard ASCII character set with one ASCII character stored per byte. The end of each text line is designated by a carriage return (CR) character (octal 15) followed by a line fee (LF) character (octal 12). If there is a CR character not followed by a LF character, it is transferred as a CR followed by a null byte (octal 0). The presence of a CR followed by anything other than a LF or a null byte in a datagram is undefined. If TFTP is running on a computer that does not represent text in ASCII (e.g., an IBM mainframe using EBCDIC), it is required to translate to netascii. In addition, different operating systems represent end-of-lines differently. For example, Unix/Linux systems only

store a LF; whereas, Windows uses CR LF. Since your TFTP server will be running in a Linux environment, your TFTP server will have to insert a CR before every LF in the source and also insert a null character if you encounter any CR's. The *Implementation Notes* section describes complications due to the need to insert a LF or a null byte into the outgoing datagram. Obviously, the client has to translate from netascii into its standard text file format as well.

The octet format is used to transfer binary. There are two uses of TFTP to transfer binary data. First, two systems that are of the same architecture (e.g., both X86 architecture) can exchange a binary file without problem. Second, binary transfers between two systems of different architectures must function in the following manner: 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 service. Clients send their files to the server in octet mode and then later retrieve the files in octet mode. As long as the server uses the same techniques to store and later read the binary file, the file contents will not have changed.

4. Sequence Number Wrap-Around and Maximum File Size

The original version of TFTP was limited to transferring a file no larger than 32 MB (MB = 2^{20} bytes), which comes from 512 bytes/block x 2^{16} blocks = 32 MB. The maximum number of blocks was limited by the 16-bit block number field. Today, however, most servers and clients support block number wrap-around where the block number goes from 65,535 (i.e., $2^{16} – 1$) to 0, which results in an unlimited size. Recall that a Stop-and-Wait protocol only requires a one-bit sequence number.

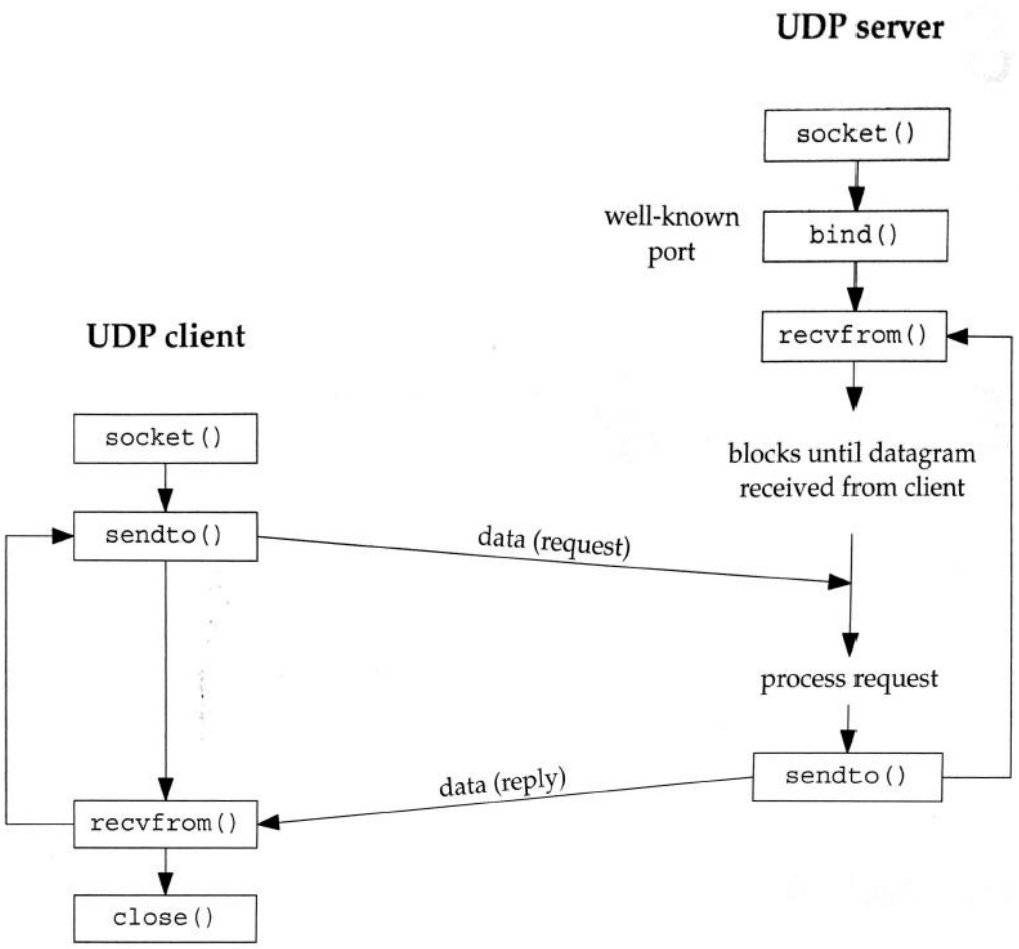
Please implement your TFTP server using block number wrap-around. During your testing, you should try transferring a file bigger than 32MB to see if the client you are using also supports wrap-around.

Implementation Notes

In this section we discuss the following topics: (1) UDP Programming, (2) TFTP Server Architecture, (3) UDP Checksum Errors, (3) Implementing timeouts, and (4) Local file I/O for octet and netascii.

1. User Datagram Protocol (UDP) Programming

There are differences in UDP and TCP programming. Recall that UDP is a connectionless, unreliable, datagram protocol. In contrast, TCP is connection-oriented and provides a reliable byte stream. The figure below shows typical socket functions for a UDP client and server [UNP vol 1 3rd Ed., Ch 8]. When compared to the similar diagram for the TCP client and server (Lecture 3, slide #5), we can see that the UDP client does not establish a connection with the server (i.e., no call to `connect()`). Instead, the UDP client just sends a datagram to the server using the `sendto()` function (described below), which requires the address of the destination (IP address and port number) as a parameter. Similarly, the UDP server does not issue a `listen()` or an `accept()`. Instead, the UDP server just calls the `recvfrom()` function, which blocks until data arrives from some client. The `recvfrom()` function returns the address of the client (IP address and port number), as well as the datagram, so the server can respond to the client.



The `sendto()` and `recvfrom()` system calls are described in section 5.8 of Beej's Guide. These two functions are similar to the standard `read()` and `write()` functions, but there are three additional arguments [UNP vol1 3rd Ed, Ch 8]:

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, –1 on error

The first three arguments, *sockfd*, *buff*, and *nbytes,* are identical to the first three arguments to `read()` and `write()`. The flags field can be set to zero for this assignment.

The *to* argument in `sendto()` is a socket address structure containing the IP address and port number of where the data is to be sent. The size of the socket address structure is specified by *addrlen*. The `recvfrom()` function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*. Note that the final argument to `sendto()` is an integer value, while the final argument to `recvfrom()` is a pointer to an integer value (the final argument to `recvfrom()` is called a *value-result* argument because the size is both a *value* when the function is called and a *result* when the function returns).

The final two arguments to `recvfrom()` are similar to the final two arguments to `accept()`: the contents of the socket address structure upon return tells us who sent the UDP datagram. The final two arguments of `sendto()` are similar to the final two arguments of `connect()` : we fill in the socket address structure with the IP address and port number of where to send the UDP datagram.

When your TFTP server receives the first UDP packet from a TFTP client, it will re-use the socket address structure returned from the `recvfrom()` in subsequent `sendto()` calls when sending data.

Unlike TCP, UDP writes generate a datagram and UDP reads return a single datagram. Not all implementations of UDP have the same behavior if a datagram arrives that is larger than the application's buffer. Some implementations discard the excess bytes and return a `MSG_TRUNC` flag (the flag can only be read by calling `recvmsg()`), other versions discard the excess bytes but do not tell the application, and finally there are even some versions that keep the excess bytes and return them in a subsequent read operation. One way to deal with this issue is to always allocate an application buffer that is one byte greater that the largest datagram the application should ever receive. If a datagram is ever received whose length equals this buffer size, consider it an error [UNP vol 1 3rd Ed, p. 594].

2. TFTP Server Architecture [UNP vol 1 3rd Ed, Chapter 22 – Concurrent UDP Servers]

Since your TFTP server must support multiple simultaneous connections, it is easiest to `fork()` a child process to handle each client transfer, similar to your TCP Echo Server. Alternatively, you could implement the server as a single process using `select()`, but this would be complicated for the TFTP server because, unlike TCP, you would have to keep a data structure for all the client IP addresses and port numbers.

With TCP, it is simple to just `fork()` a new child and let the child handle the new client. What simplifies server concurrency when TCP is being used is that every client connection is unique. Recall that TCP socket pairs are unique for each connection (e.g., the ephemeral port for the client is unique even if there are multiple clients on the same machine connecting to the same server). With UDP, however, things are different. For UDP servers that need to exchange multiple datagrams with the client, which is the case we have with the TFTP server, the problem is that the only port that the client knows for the server is the well-known port. The client sends the first datagram of its request to that well-known port, but how does the server distinguish between subsequent datagrams from that client and new requests? The typical solution to this problem is as follows: (1) the child server process closes the socket bound to the server's well-known address, (2) the child server process creates a new socket for this client, (3) the child process then calls `bind()` to assign an ephemeral port to that socket (the operating system will assign an ephemeral port in your call to `bind()` if you specify a port number of 0, e.g., `servaddr.sin_port = htons(0)`), and (4) the child server process uses this new socket for all future datagram transmissions to the client. This change also requires that the TFTP client look at the port number of the server's first reply and send subsequent datagrams for this request to that ephemeral port

on the server. The figure below depicts what has been described above [UNP vol 1 3rd Ed, pp. 612-614]:

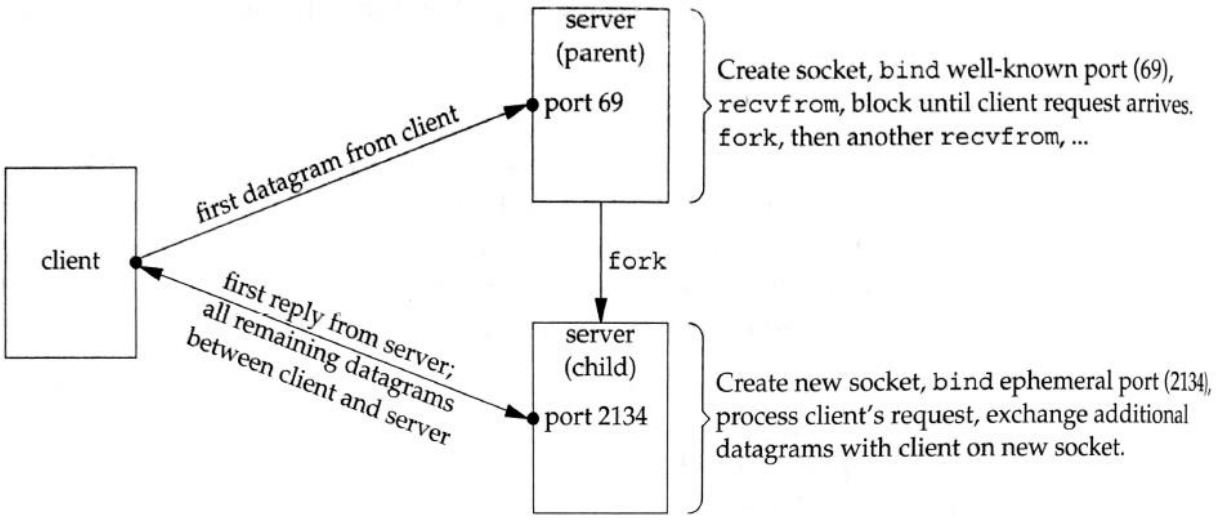


Figure 22.19 Processes involved in standalone concurrent UDP server.

3. UDP Checksum Errors

In spite of the fact that the UDP Checksum is optional in IPv4, UDP Checksums are enabled by default in all systems today (you could use a socket option to disable them, but you do not want to do this!). Recall that the UDP Checksum is an end-to-end error detection mechanism, and the UDP Checksum is not modified in transit (except when the packets pass through a NAT device). The UDP Checksum covers the data, the UDP header, and the UDP pseudo-header.

If the receiving UDP kernel process detects a UDP Checksum error, the datagram is silently discarded, and no error message is generated. Consequently, the only way for your TFTP server to recover from an ACK datagram received with a Checksum error, or a lost ACK for that matter, is via the protocol's timeout function.

4. Implementing Timeouts

The sender of a TFTP DATA packet (the TFTP Server in the case of a RRQ request or the TFTP Client in the case of a WRQ request) needs to (1) send a DATA packet, (2) set a timeout timer, and (3) wait for an ACK. If the ACK arrives before the timeout timer expires, the timer is cancelled and you then process the next packet. If the timeout timer expires, you must retransmit the last DATA packet.

There are two ways to implement the timeout timer functionality:

- Call `select()` to wait for either of the following two events: (1) data ready for reading on the socket descriptor created by the child server process and (2) a timeout of say one second from when `select()` is called.
- First, set up a signal handler for `SIGALARM` with `sigaction()` (this is the POSIX way to set up a signal handler), or use the simpler but system dependent `signal()` function. After sending a DATA packet and before issuing a `readfrom()`, call `alarm(1)`. Cancel the alarm if the if the ACK arrives by calling `alarm(0)`.

If you implement the TFTP server as recommended in section 2 above, I think using the `select()` method is the simpler of the two approaches. Below is a function [UNP vol. 1 3rd Ed, p.385] for setting a timeout using `select()`. The `readable_timeo()` function waits for up to *sec* seconds for a socket descriptor to become readable.

lib/readable_timeo.c

```
 1 #include    "unp.h"

 2 int
 3 readable_timeo(int fd, int sec)
 4 {
 5     fd_set  rset;
 6     struct timeval tv;

 7     FD_ZERO(&rset);
 8     FD_SET(fd, &rset);

 9     tv.tv_sec = sec;
10     tv.tv_usec = 0;

11     return (select(fd + 1, &rset, NULL, NULL, &tv));
12         /* > 0 if descriptor is readable */
13 }
```

lib/readable_timeo.c

Figure 14.3 `readable_timeo` function: waits for a descriptor to become readable.

If the TFTP client is aborted, the TFTP server will *not* be notified. Your server code should count consecutive timeouts, and then clean up and exit the child process, if you have more than say 10 timeouts in a row.

The choice of a one second timeout is arbitrary. In most cases, TFTP is being used on a local network where the RTT is small. If you were using your TFTP server on a network where the RTT > 1 second, you would need to use a larger

timeout interval. The standard Unix/Linux TFTP servers and clients have a command line argument that allows you to specify the timeout. The default is typically one second.

5. Local File I/O for octet and netascii

For octet transmission, you can use the standard Unix/Linux `open()`, `read()` (RRQ), and `write()` (Bonus WRQ) system calls.

For netascii transmission, however, you will want to use the Unix/Linux `fopen()`, `getc()` and `putc()` system calls. The challenge in dealing with netascii files on Unix/Linux systems has to do with the need to insert a CR before a LF and a null character after a CR. It is possible that the second byte of a two-byte CR LF or CR <null> sequence will not fit in the current TFTP datagram. If that turns out to be the case, the second character will have to go in the first character of the next TFTP block. The code fragment below from [UNP, vol 1, 1st Ed, p. 499] implements this logic, where the variable `nextchar` needs to be global if the code fragment is in a function in order to "remember" if the next character to output is linefeed or a null.

```
for (count = 0; count < maxnbytes; count++) {
        if (nextchar >= 0) {
                *ptr++ = nextchar;
                nextchar = -1;
                continue;
        }

        c = getc(fp);

        if (c == EOF) { /* EOF return means eof or error */
                if (ferror(fp))
                   err_dump("read err from getc on local file");
                return(count);

        } else if (c == '\n') {
                c = '\r';               /* newline -> CR,LF */
                nextchar = '\n';

        } else if (c == '\r') {
                nextchar = '\0';        /* CR -> CR,NULL */

        } else
                nextchar = -1;

        *ptr++ = c;
}
```

Bonus Feature

For the assignment you must implement RRQ support in your TFTP server for a possible 100 points. For a Bonus, you may implement WRQ support in your TFTP server for an additional 30 points. If you implement the Bonus Feature, test that you can send both netascii and binary files from your client to the server, and then transfer back the same files from the server to the client. Compare the original and the copied versions to make sure they are identical.

Implementation of WRQ support is easier than RRQ since you do not have to use a timer (the TFTP client is responsible for the timeout).

Submission Guidelines

1. My expectation is that each team member will contribute equally to the network programming assignments. Please include a statement in your README that describes the role of each team member in completing the assignment.

2. Test cases – Please develop a set of test cases to demonstrate that your server works with an off-the-shelf TFTP client. Submit a short report describing how you tested your final implementation, and a description of your test cases along with screen captures of the test cases to document correct operation. At a minimum, include the following test cases: (1) transfer a binary file of 2048 bytes and check that it matches the source file, (2) transfer a binary file of 2047 bytes and check that it matches the source file, (3) transfer a netascii file that includes two CR's and check that the resulting file matches the input file, (4) transfer a binary file of 34 MB and see if block number wrap-around works, (5) check that you receive an error message if you try to transfer a file that does not exist and that your server cleans up and the child process exits, (6) Connect to the TFTP server with three clients simultaneously and test that the transfers work correctly (you will probably need a big file to have them all running at the same time), (7) terminate the TFTP client in the middle of a transfer and see if your TFTP server recognizes after 10 timeouts that the client is no longer there (you will need a big file), and (8) separate test cases for the bonus feature (see the Bonus Feature section).

3. The network programming assignments are due by 5:00 pm on the due date.

4. Your source code must be submitted to Turnitin.com using eCampus by 5:00 pm on the due date. Turnitin.com is plagiarism detection software that will compare your code to files on the Internet as well as your peers' code. Additional details on how to submit your code will be provided prior to the submission date.

5. All programming assignments must include the following: makefile, README, and the code.

6. The README should contain a description of your code: architecture, usage, errata, etc.

7. Make sure all binaries and object code have been cleaned from your project before you submit.

8. Your project must compile on a standard Linux development system. Your code will be graded on a Linux testbed.

9. Explanation on the submission procedure will be provided by the TA prior to the submission date.

Notes

1. Just as in the previous assignments, it will be helpful for you to break the project up into parts and implement and test each independently.

2. The standard Unix/Linux TFTP client has a “trace” mode that will show all the packets and help you in debugging your server.

3. You are required to implement TFTP error messages. You server should respond with an error message (opcode = 5) if a read request (RRQ) or write request (Bonus WRQ) command cannot be processed. Read (RRQ) or write (WRQ) errors during file transmission also cause this message to be sent, and transmission is then terminated

4. When a file transfer is complete or you encounter an error condition, your TFTP server child process should clean up and exit. The comment about Zombie processes in the description of Network Programming Assignment 1 is applicable to this assignment as well.

5. TFTP uses port number 69 by default. But, you will not be able to bind the server socket to this port. Therefore, any other port number (ephemeral port) can be used by both the server and client.