**Programming Assignment 3**

Chenjie Luo (UIN: 324007289)

Yuwen Chen (UIN: 227009499)

1. **Description**

This project is roughly divided into two parts. Chenjie Luo is responsible for building up the framework and local file input and output while Yuwen Chen is responsible for the timoutouts, multiple clients handling and we verified server in different test cases together.

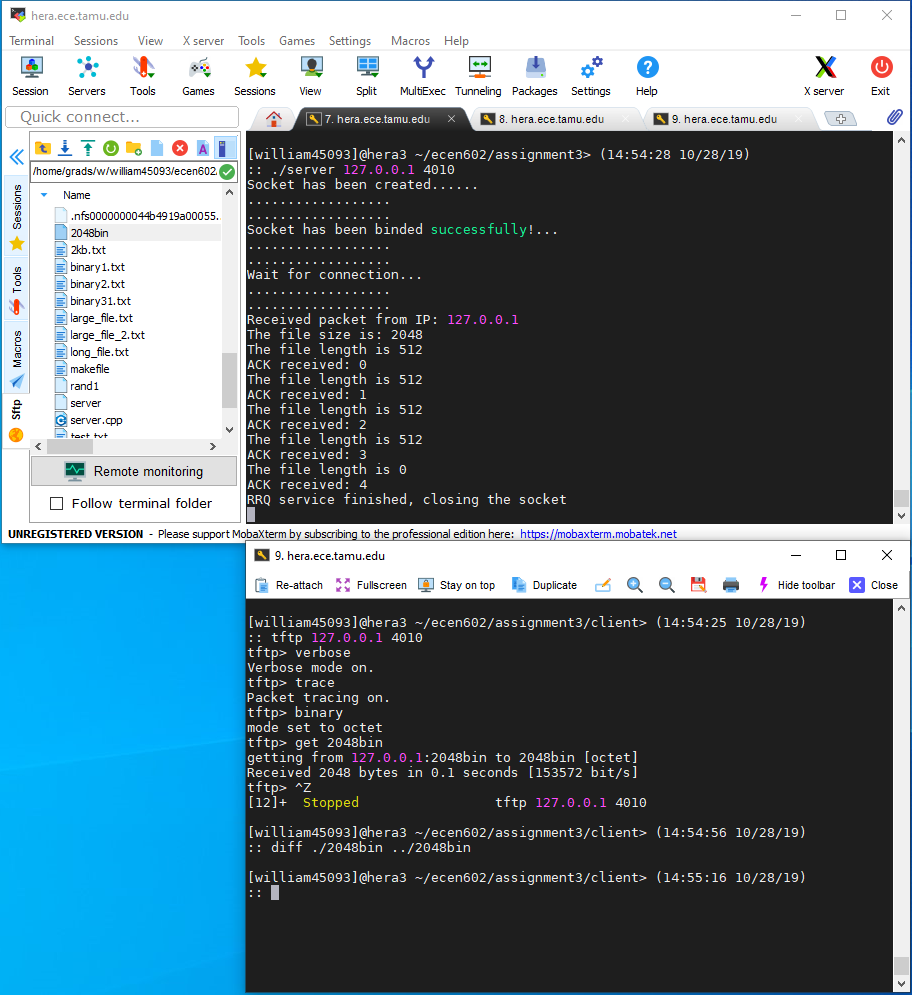
For the server’s architecture, when target file is executed, users will have to enter two inputs which are: server’s IP address as well as port number. I designed a struct to store named TFTP\_MSG to store messages read from buffer with the given form in recitation. The struct comes with four attributes: opcode, block\_num, filename and mode. The opcode is an integer indicates which type the message belong to. The filename stores the object file name we want to transmit. Block\_num is used for ACK message and it indicates the acknowledgement packet number.

To handle multiple clients, we used fork() to create child process in order to take care of each client’s transfer. When the socket is crated and bind successfully, we will listen to RRQ message. If RRQ is found in TFTP\_MSG, we will start a new child process using fork() and firstly close the socket with port number specified when opening the server. We then created a new socket and bind with the ephemeral port.

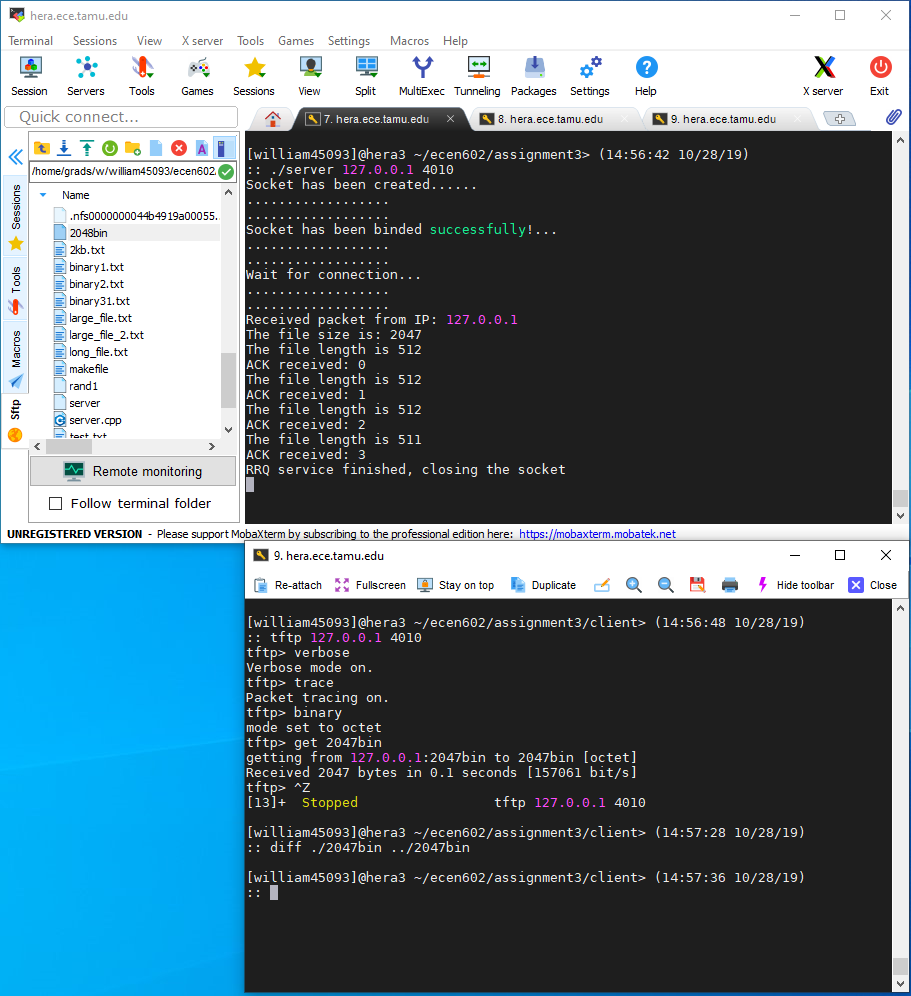
For I/O operation, FILE\* f is defined to implement I/O operations. If the f is failed to be open, an error message (opcode = 5) is generated and send back to the client. Else FILE\* f can be open successfully and file size is first recorded number of packets needed is calculated. For the I/O part, at the beginning I used EOF to detect the end of file but it seemed sometimes getc() could not find EOF when we used large random generated files. I think reason lies in MACRO of “EOF” is different on different machines. Then I switched to calculate how many remaining bytes needed to be transmitted since I recorded total file size. All the packet except the last should have 512 bytes. And this turns out to work correctly. All the packet will be wrapped and send to the client. When client received an ACK message should be echoed back. We updated the block\_num and transmit next packet then. When we came across extremely large file (> 32 MB), the default TFTP client will terminate after ACK 65536th packet. In this case we will need to reset the ACK number to make it always less than 65536. We used another variable named block\_cnt to count total number of packets right now and used it for detecting last packet as well.

For the timeout function, after the server send a packet to the client, a select function is used in the server side to wait for the acknowledgement message from the client indicating that the packet is received by the client. If the server does not receive the acknowledgement message in 1 second after sending the packet, the select function will be timeout and return value 0. In this case, a for loop is used to retransmit the last packet again until the acknowledgement message is received (select function return value larger than 0). The for loop will only retransmit a certain packet for 10 times, and if a packet is retransmitted over 10 times, and the server still not receive the corresponding acknowledgement message, the following transmission to the client will be shutdown, and both the socket bound to the client and the child process to handle this transmission will be closed.

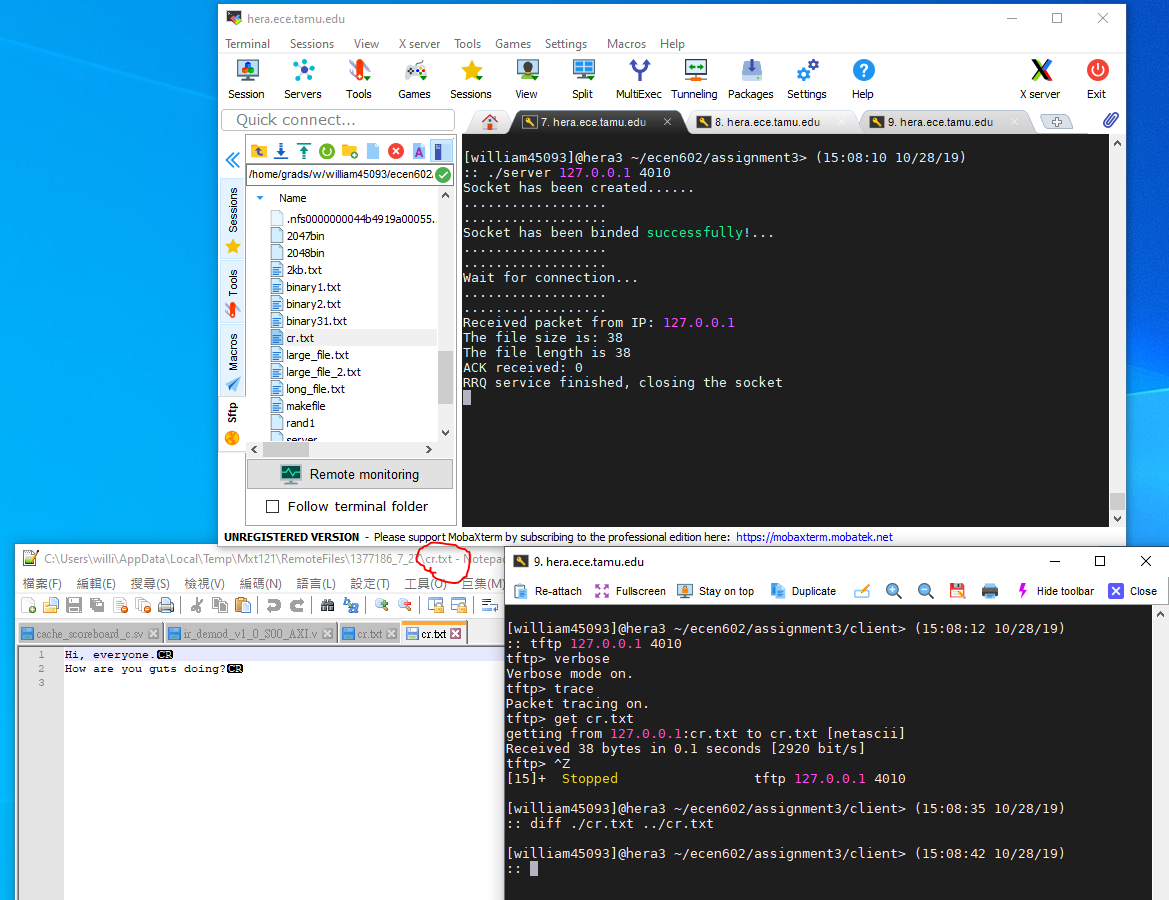
1. **Instruction to run our code**
2. After downloading the file from github, type “make” in the command line to generate the execution file: server
3. Type “./server [server IP address] [port number]” to execute the server
4. Now, users can execute the default TFTP client by typing “tftp [server IP address] [port number]” to connect to our server. After the connection is created successfully, users can send a RRQ request to our server to download files from our server.
5. **Test result**
6. transfer a binary file of 2048 bytes and check that it matches the source file



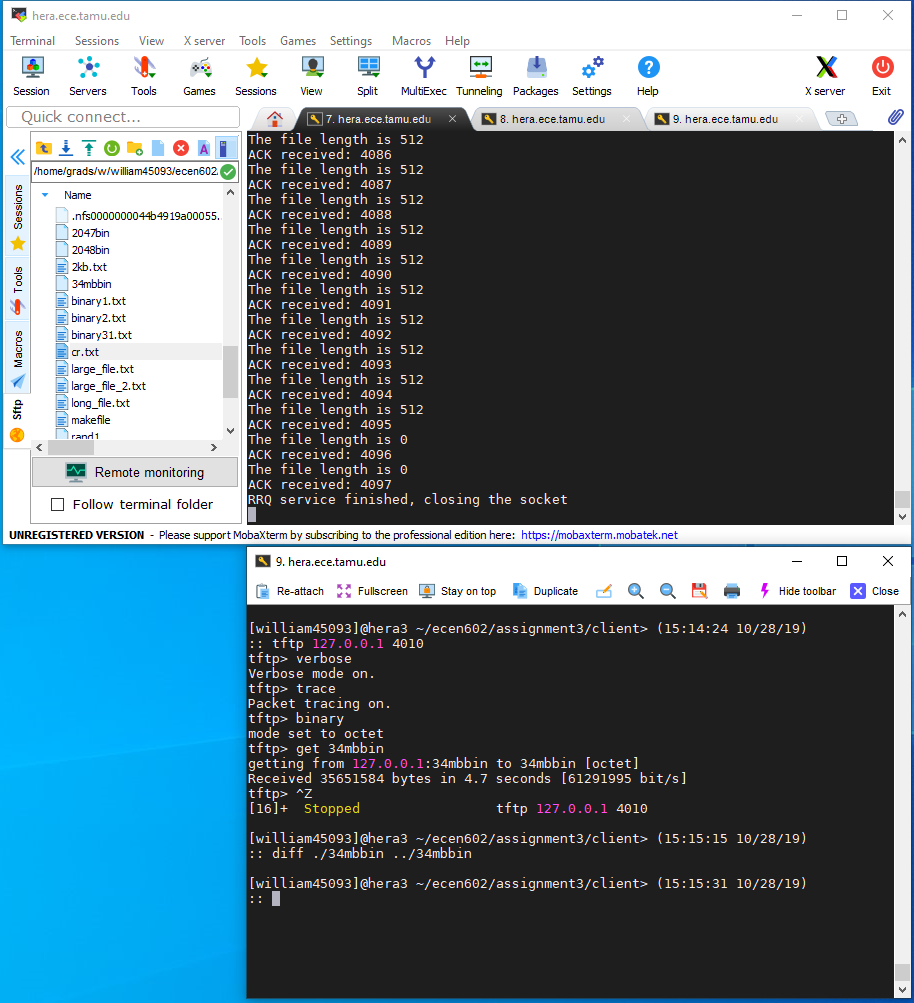
1. transfer a binary file of 2047 bytes and check that it matches the source file



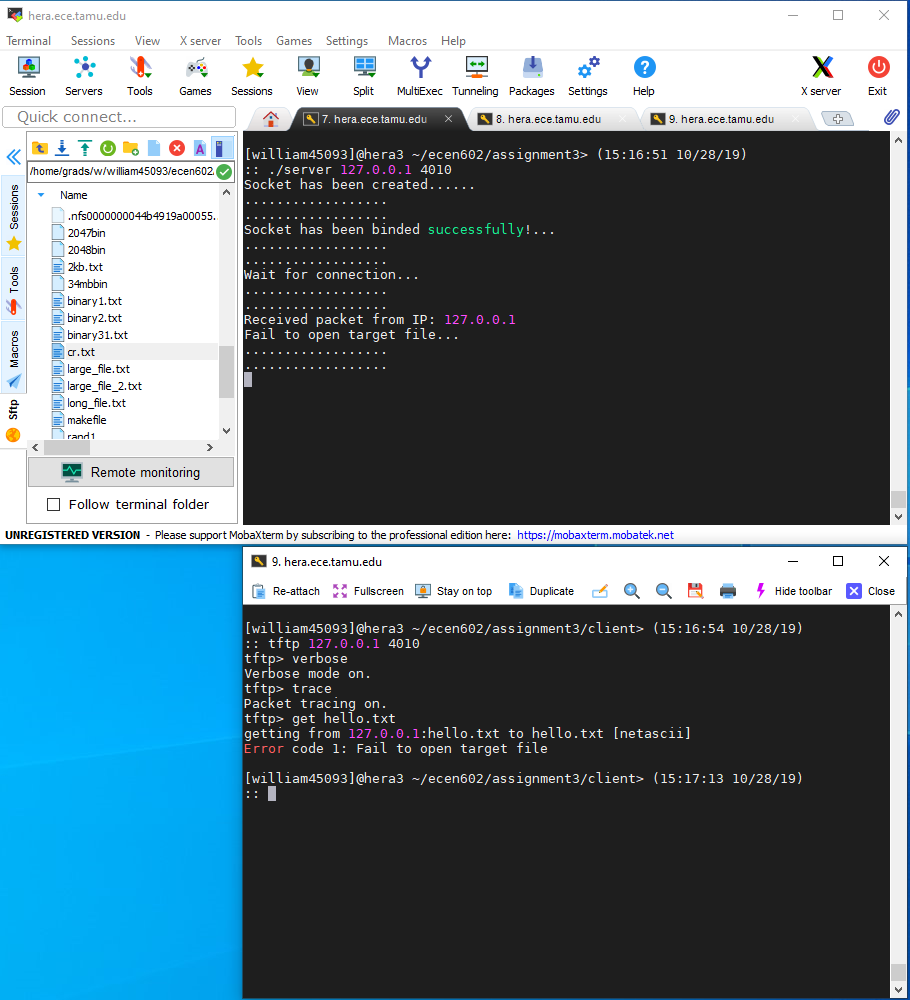
1. transfer a netascii file that includes two CR’s and check that the resulting file matches the input file



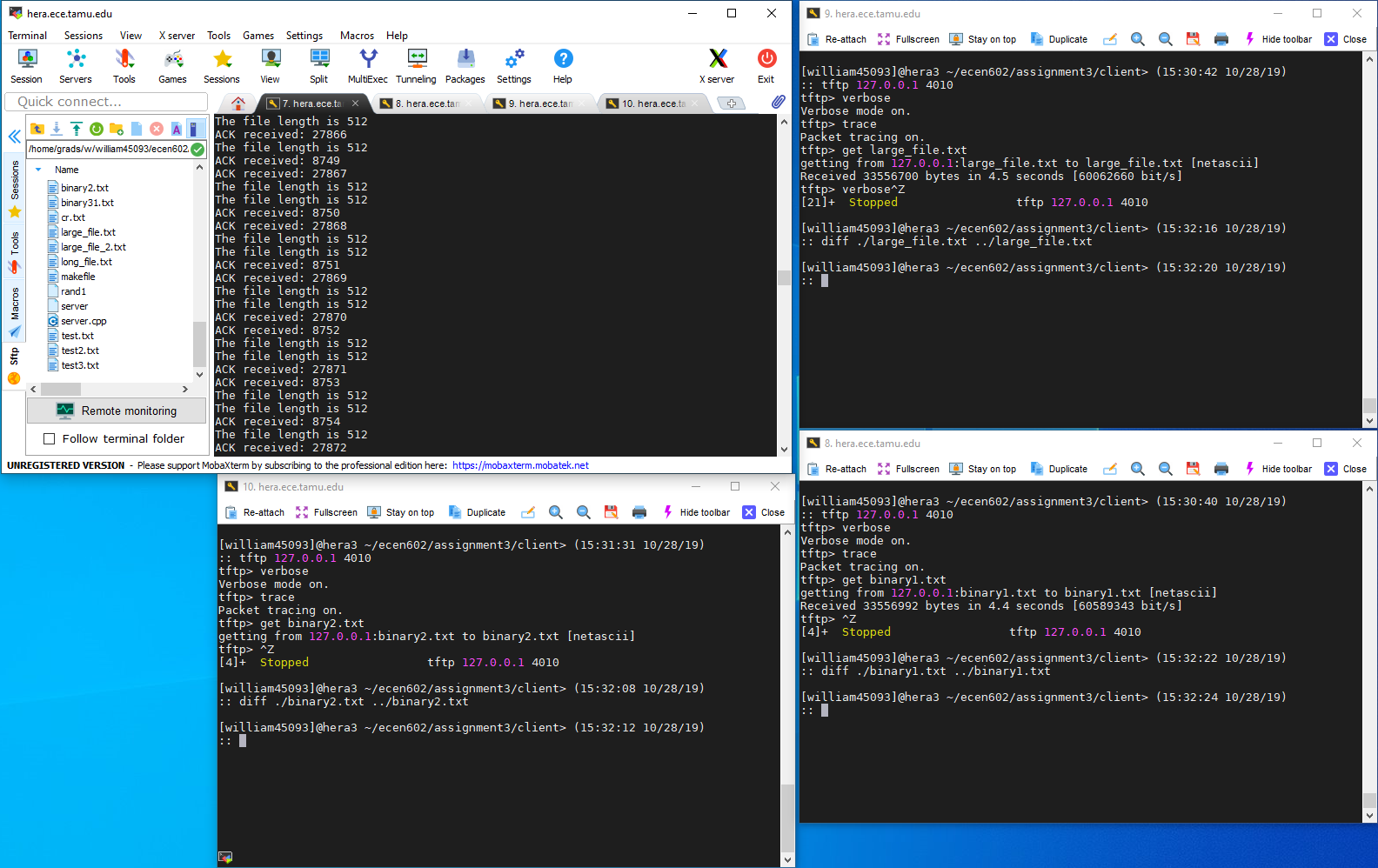
1. transfer a binary file of 34 MB and see if block number wrap-around works



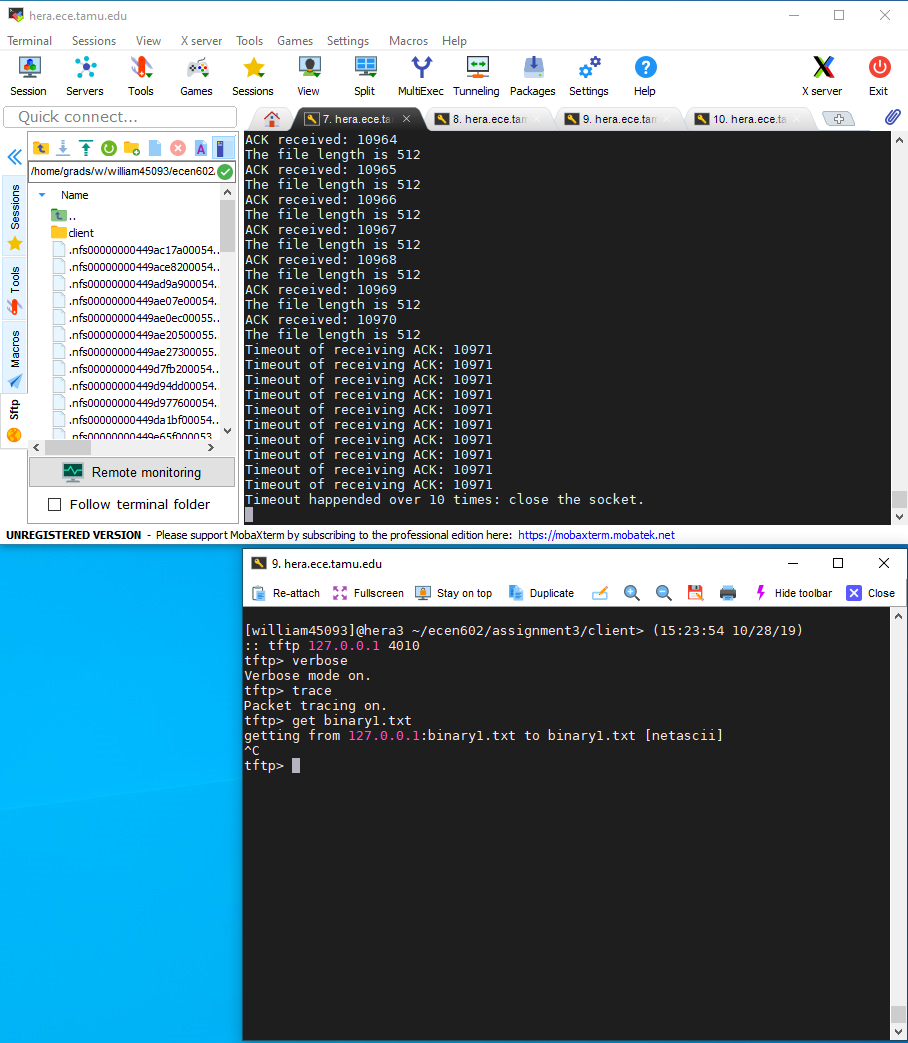
1. check that you receive an error message if you try to transfer a file that does not exist and that your server cleans up and the child process exits



1. 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)



1. 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)



1. **Code**

server.cpp

#include <stdlib.h>

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <errno.h>

#include <unistd.h>

#include <signal.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#include <sys/wait.h>

#include <string.h>

#include <sys/time.h>

#include <sys/select.h>

using namespace std;

#define RRQ 1

#define WRQ 2

#define DATA 3

#define ACK 4

#define ERROR 5

#define ERROR\_MSG\_OPEN\_FILE "Fail to open target file"

// SELF DEFINED DATA STRUCTURE NAME TFTP\_MSG, IT CONSISTS OF FOUR ATTRIBUTES

struct TFTP\_MSG{

int opcode;

char \*filename;

int block\_num;

char \*mode;

};

void print\_status(std::string s){

std::cout << s << "..." << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

void address\_set(struct sockaddr\_in &address, int &PORT){

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

}

void handle\_sigchld(int sig) {

int saved\_errno = errno;

while (waitpid((pid\_t)(-1), 0, WNOHANG) > 0) {}

errno = saved\_errno;

}

// CONVERT UNSIGNED SHORT INTEGER TO HOST BYTE ORDER

unsigned short int convert\_network\_byte(char \*buf){

unsigned short int i;

memcpy(&i, buf, 2);

i = ntohs(i);

return i;

}

// CONVERT TO NETWORK BYTE ORDER

void convert(char \*buf, unsigned short int i){

i = htons(i);

memcpy(buf,&i,2);

}

// READ FROM BUFFER AND CONVERT INTO TFTP\_MSG STRUCTURE

struct TFTP\_MSG\* readFile(char \*buffer){

struct TFTP\_MSG\* out = (struct TFTP\_MSG\*)malloc(sizeof(struct TFTP\_MSG));

out->opcode = convert\_network\_byte(buffer);

if (out->opcode == ACK){

out->block\_num = convert\_network\_byte(buffer + 2);

}

if (out->opcode == RRQ){

int i = 0;

char temp[512];

while (buffer[i + 2] != '\0'){

temp[i] = buffer[i + 2];

i += 1;

}

temp[i] = '\0';

out->filename = (char\*)malloc(strlen(temp)\*sizeof(char));

strcpy(out->filename, temp);

while(buffer[i + 2]=='\0')

i++;

int j = 0;

while (buffer[i + 2] != '\0'){

temp[j] = buffer[i + 2];

i += 1;

j += 1;

}

temp[j] = '\0';

out->mode = (char\*)malloc(strlen(temp)\*sizeof(char));

strcpy(out->mode, temp);

}

return out;

}

// WRAP THE MESSAGE FROM BUFFER INTO CHAR\* TO SEND

char \*wrap\_msg(int opcode, int block\_number, char \*file\_buffer, int length){

char \*out;

if(opcode == DATA){

out=(char \*)malloc((length + 4)\*sizeof(char));

convert(out,opcode);

convert(out + 2,block\_number);

memcpy(out + 4,file\_buffer,length);

}

if(opcode == ERROR){

out=(char \*)malloc((length+5)\*sizeof(char));

convert(out,opcode);

convert(out + 2,block\_number);

memcpy(out + 4,file\_buffer,length);

memset(out + 4 + length,'\0',1);

}

return out;

}

int readable\_timeout(int fd, int sec){

fd\_set rset;

struct timeval tv;

FD\_ZERO(&rset);

FD\_SET(fd, &rset);

tv.tv\_sec = sec;

tv.tv\_usec = 0;

return (select(fd + 1, &rset, NULL, NULL, &tv));

}

int main(int argc, char \*\*argv){

if (argc != 3){

errno = EPERM;

perror("Illegal Input! Please only enter your IP addr and server port");

exit(EXIT\_FAILURE);

}

std::string IP\_addr = argv[1];

std::string port\_str = argv[2];

// fd\_list saves all file descriptors

fd\_set fd\_list;

// curr\_fd\_list reads all current file descriptors

fd\_set curr\_fd\_list;

FD\_ZERO(&fd\_list);

FD\_ZERO(&curr\_fd\_list);

int max\_fd;

int socket\_fd; // server's socket

int new\_socketfd;

char buffer[512];

char s[INET\_ADDRSTRLEN];

int opt = 1;

socklen\_t client\_addr\_size;

struct addrinfo addressinfo;

struct addrinfo \*server\_addrinfo;

struct addrinfo \*curr\_addrinfo;

struct sockaddr\_storage client\_addr;

struct sockaddr addr;

struct sockaddr\_in \*address;

memset(&addressinfo, 0, sizeof(addressinfo));

pid\_t child;

int bytes;

addressinfo.ai\_family = AF\_INET;

addressinfo.ai\_socktype = SOCK\_DGRAM; // SET THE PROTOCOL TO UDP

if ((getaddrinfo(argv[1], argv[2], &addressinfo, &server\_addrinfo)) != 0) {

errno = EPERM;

perror("Fail to set address...");

exit(EXIT\_FAILURE);

}

for (curr\_addrinfo = server\_addrinfo; curr\_addrinfo != NULL; curr\_addrinfo = curr\_addrinfo->ai\_next){

// CREATE A SOCKET WITH A DESCRIPTER socket\_fd WHICH SUPPORT IPv4

if ((socket\_fd = socket(curr\_addrinfo->ai\_family, curr\_addrinfo->ai\_socktype, curr\_addrinfo->ai\_protocol)) < 0){

errno = ETIMEDOUT;

continue;

}

print\_status("Socket has been created...");

setsockopt(socket\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

if (::bind(socket\_fd, curr\_addrinfo->ai\_addr, curr\_addrinfo->ai\_addrlen) < 0){

errno = EADDRINUSE;

continue;

}

break;

}

if (curr\_addrinfo == NULL){

errno = EPERM;

perror("Fail to create a socket and bind...");

exit(EXIT\_FAILURE);

}

print\_status("Socket has been binded successfully!");

print\_status("Wait for connection");

// int bytes;

addr = \*(curr\_addrinfo->ai\_addr);

address = (struct sockaddr\_in \*) &addr;

address->sin\_port = htons(0);

struct sigaction sa;

sa.sa\_handler = handle\_sigchld; // wipe out all dead processes

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = SA\_RESTART;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {

perror("sigaction");

exit(EXIT\_FAILURE);

}

char nextchar = -1;

while (true){

client\_addr\_size = sizeof(client\_addr);

if (recvfrom(socket\_fd, buffer, 511, 0, (struct sockaddr \*)&client\_addr, &client\_addr\_size) < 0){

exit(EXIT\_FAILURE);

}

struct sockaddr\_in\* new\_client\_addr = (struct sockaddr\_in\*) &client\_addr;

cout << "Received packet from IP: " << inet\_ntop(client\_addr.ss\_family,&(((struct sockaddr\_in\*)address)->sin\_addr),s, sizeof s) << endl;

struct TFTP\_MSG \*to\_send;

to\_send = readFile(buffer);

if (to\_send->opcode != RRQ){

if (to\_send->opcode == WRQ)

print\_status("Currently WRQ is not valid");

else

print\_status("Invalid request");

continue;

}

if (fork() == false){

close(socket\_fd);

if ((socket\_fd = socket(curr\_addrinfo->ai\_family, curr\_addrinfo->ai\_socktype, curr\_addrinfo->ai\_protocol)) < 0){

exit(EXIT\_FAILURE);

}

setsockopt(socket\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

if (::bind(socket\_fd, &addr, sizeof(addr)) < 0){

errno = EADDRINUSE;

perror("Fail to bind...");

}

socklen\_t new\_len = sizeof(addr);

if(getsockname(socket\_fd, &addr, &new\_len) == -1){

perror("Error");

}

FILE \*f;

f = fopen(to\_send->filename, "rb");

if (f == NULL){

print\_status("Fail to open target file");

char error\_msg[512];

strcpy(error\_msg, "Fail to open target file");

int error\_length = strlen(error\_msg);

char \*error\_buffer = wrap\_msg(ERROR, 1, error\_msg, error\_length);

if ((sendto(socket\_fd, error\_buffer, error\_length + 5, 0, (struct sockaddr\*)&client\_addr, client\_addr\_size)) < 0){

perror("Fail to send error message...");

continue;

}

exit(EXIT\_FAILURE);

}

// SEARCH FOR THE FILE SIZE AND RESET THE POINTER FILE \*f TO THE BEGINNING OF FILE

fseek(f, 0, SEEK\_END);

int file\_size = ftell(f);

fseek(f, 0, SEEK\_SET);

int num\_packet = file\_size / 512 + 1;

cout << "The file size is: " << file\_size << endl;

//cout << "Number of packets needed: " << num\_packet << endl;

char file\_buffer[512];

int block\_num = 0;

int block\_cnt = 0;

int last\_ACK = 0;

int file\_length = 0;

int special\_letter = 0;

char \*ptr = file\_buffer;

int totalcnt = 0;

short int c;

int overhead = 0;

int timeout, sel\_timeout;

int opc = -1;

FD\_SET(socket\_fd, &fd\_list);

max\_fd = socket\_fd;

while (last\_ACK < num\_packet){

// RUN THE ROLL OVER TO AVOID DEFAULT CLIENT TERMINATE

if (block\_num == 65536){

block\_num = 0;

}

if (last\_ACK == block\_num){

for (int cnt = 0; cnt < 512; cnt++){

c = getc(f);

// KEEP READING EVEN COME ACROSS EOF, THE FOLLOWING LOGIC WILL RESOVE THE LAST PACKET ISSUE

if (c == EOF){

if (ferror(f))

perror("Error");

break;

}

\*ptr++ = c;

file\_length += 1;

}

cout << "The file length is " << file\_length << endl;

block\_num += 1;

block\_cnt += 1;

totalcnt += 1;

ptr = file\_buffer;

}

// THIS IF CONDITION PLAYS THE ROLE TO DETECT WHETHER THIS IS THE LAST PACKET

if (block\_cnt == num\_packet)

file\_length = file\_size % 512;

char\* data\_packet = wrap\_msg(DATA, block\_num, file\_buffer, file\_length);

char\* recv\_packet = (char\*) malloc(4 \* sizeof(char));

//cout << "Block number is " << block\_num << endl;

for (timeout = 0; timeout < 10; timeout++) { // if timeout over 10 times, the trasmission will be stopped and socket will be closed

if ((sendto(socket\_fd, data\_packet, file\_length + 4, 0, (struct sockaddr \*)&client\_addr, client\_addr\_size)) < 0){

perror("Fail to send...");

exit(EXIT\_FAILURE);

}

sel\_timeout = readable\_timeout(socket\_fd, 1); // set time out to 1 second

if (sel\_timeout < 0) {

perror("Select error: Receiving ACK...");

exit(EXIT\_FAILURE);

}

else if (sel\_timeout == 0) {

cout << "Timeout of receiving ACK: " << last\_ACK << endl;

}

else {

if (recvfrom(socket\_fd, recv\_packet, sizeof(recv\_packet), 0, (struct sockaddr \*)&client\_addr, &client\_addr\_size) < 0){

perror("Fail to receive ACK...");

exit(EXIT\_FAILURE);

}

opc = convert\_network\_byte(recv\_packet);

if (opc == 4) { // Check if the received message is ACK

cout << "ACK received: " << last\_ACK << endl;

break;

}

}

}

if (timeout == 10) { // timeout equal to 10 times, close socket and child process

cout << "Timeout happended over 10 times: close the socket." << endl;

fclose(f);

close(socket\_fd);

\_exit(2); // close child process

}

file\_length = 0;

//free(data\_packet);

//memset(file\_buffer, '\0', 512);

/\*ptr = file\_buffer;

curr\_fd\_list = fd\_list;

if (select(max\_fd + 1, &curr\_fd\_list, NULL, NULL, NULL) < 0){

perror("Fail to get update");

exit(EXIT\_FAILURE);

}

if(FD\_ISSET(socket\_fd, &curr\_fd\_list)){

struct sockaddr\_storage curr\_addr;

socklen\_t curr\_length = sizeof(curr\_addr);

if((bytes = recvfrom(socket\_fd, buffer, 4, 0, (struct sockaddr \*)&curr\_addr, &curr\_length)) == -1){

perror("Fail to receive from socket");

exit(EXIT\_FAILURE);

}

}

\*/

struct sockaddr\_in\* curr\_client\_addr = (struct sockaddr\_in\*)& client\_addr;

if(curr\_client\_addr->sin\_addr.s\_addr == new\_client\_addr->sin\_addr.s\_addr){

to\_send = readFile(recv\_packet);

last\_ACK = to\_send->block\_num;

}

//cout << "block\_cnt = " << block\_cnt << ", num\_packet = " << num\_packet <<endl;

if (block\_cnt > num\_packet)

break;

free(data\_packet);

free(recv\_packet);

}

cout << "RRQ service finished, closing the socket" << endl;

close(socket\_fd);

fclose(f);

\_exit(2); // close child process

}

}

close(socket\_fd);

}