3rd Group Homework Report
Trivial File Transfer Protocol (TFTP) Server
ECEN 602 Network Programming Assignment 3

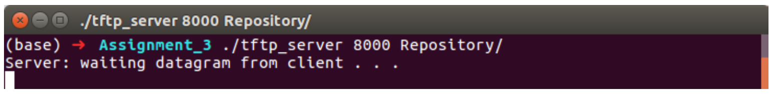
Mainly, we worked together, but here is the role what we did.
Minhwan Oh : Developed server programming, tested for integration
Sanghyeon Lee : Developed server programming, debugged for integration

File info
===============
For this program, you can see how trivial File Transfer Protocol (TFTP) server. There is one main file.
1. tftp_server.c
Path: cd Assignment_3/tftp_server.c
Main feature: Server application for TFTP server.
2. makefile
Main feature: Compile setting description

Build info
===============
Command $make

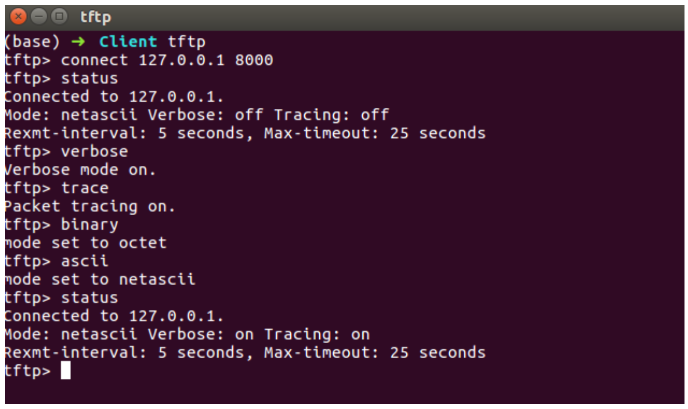
Program scenario
===============
This program includes only server for Trivial File Transfer Protocol (TFTP) because we are going to use a standard TFTP client, which does the following:

1. Start the server first with the commanding lien: $tftp_server <Port> <Repository>, where "tftp_server" is the name of the server program, Port is the port number on which the tftp server is listening, and repository is the folder that server can store or transfer files.



```
(base) ➜ Assignment_3 ./tftp_server 8000 Repository/
Server: waiting datagram from client . . .
```

2. Assume that TFTP client is installed in linux environment, after server programming runs, tftp client also needs to run. Since we put port number as 8000 in server program, client also needs to set same port name. To test properly, please go cd Assignment_3/Client then run TFTP client



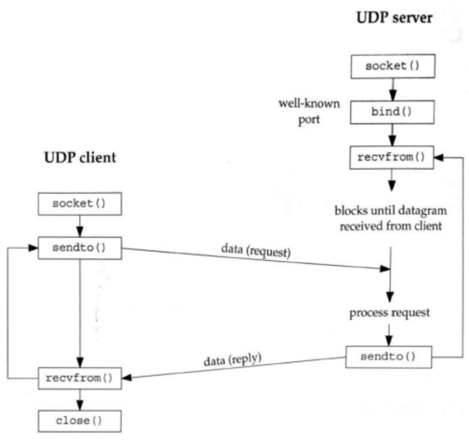
```
(base) ➜ Client tftp
tftp> connect 127.0.0.1 8000
tftp> status
Connected to 127.0.0.1.
Mode: netascii Verbose: off Tracing: off
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp> verbose
Verbose mode on.
tftp> trace
Packet tracing on.
tftp> binary
mode set to octet
tftp> ascii
mode set to netascii
tftp> status
Connected to 127.0.0.1.
Mode: netascii Verbose: on Tracing: on
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp>
```

3. Current our TFTP program can transfer files in both directions because we implemented Read Request Function (RRQ) and Write Request Function (WRQ) as well. The server allows retrieval of files in the assigned local directory.

4. There are several test cases as below, and we comply with all 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)

(8) Transfer a binary file of 2048 bytes to server and check that it matches the source file (Bonus Feature)
(9) Transfer a binary file of 2047 bytes to server and check that it matches the source file (Bonus Feature)
(10) Connect to the TFTP server with three clients simultaneously and test that the transfers work correctly – transmitting file to server simultaneously (Bonus Feature)
(11) Terminate the TFTP server in the middle of a transfer and see if your TFTP client recognizes after 10 timeouts that the client is no longer there (Bonus Feature)

Program details & File architecture
===============



1. User Datagram Protocol (UDP) Programming
- Recall that UDP is aconnectionless, unreliable, datagram protocol. In contrast, TCP is connection- oriented and provides a reliable byte stream.
- 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.

2. Multiple simultaneous connections
- To support multiple simultaneous connections, used fork() function in server which a child process to handle each client transfer, similar to TCP Echo Server.

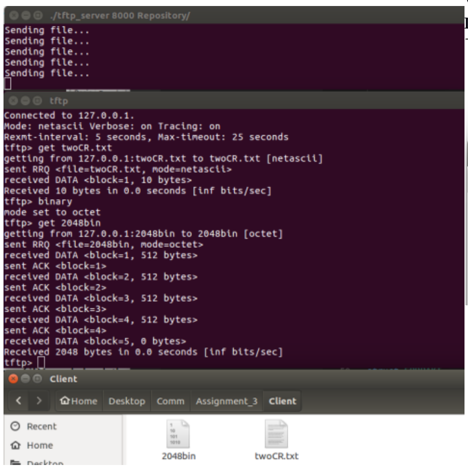
3. Timeout
- The sender of a TFTP DATA packet (the TFTP Server in the case of a RRQ request or the TFTP Client in the case of a WRQ request) needs to (1) send a DATA packet, (2) set a timeout timer, and (3) wait for an ACK. If the ACK arrives before the timeout timer expires, the timer is cancelled and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

4. File transmission
- For octet transmission, we used the standard Unix/Linux open(), read() (RRQ), and write() (Bonus WRQ) system calls.
- For netascii transmission, however, we used the Unix/Linux fopen(), getc() and putc() system calls.

File Function Explanation
===============
1. tftp_server
a) Architecture
- Scenario :
(1) Setup server_info/bind port/broadcast/listen functions
(2) Then it is going to infinite loop until the exit
- Main function :
(1) int main (int argc, char** argv) : Including main scenario
(2) void ErrSend (int errNum, struct sockaddr_in client_Addr) : Send an error message
(3) int DataSend (char* fileDir, char* mode_Str, struct sockaddr_in client_Addr) : Send the file to client
(4) int recvData (char* fileDir, char* mode_Str, struct sockaddr_in client_Addr) : Receive file from client

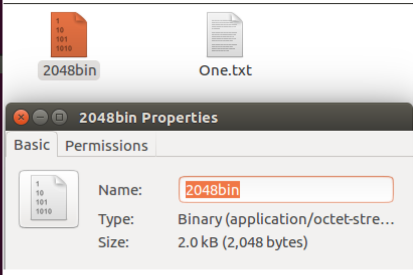
Test cases
===============

(1) Transfer a binary file of 2048 bytes and check that it matches the source file



```
./tftp_server 8000 Repository/
Sending file...
Sending file...
Sending file...
Sending file...
Sending file...
```

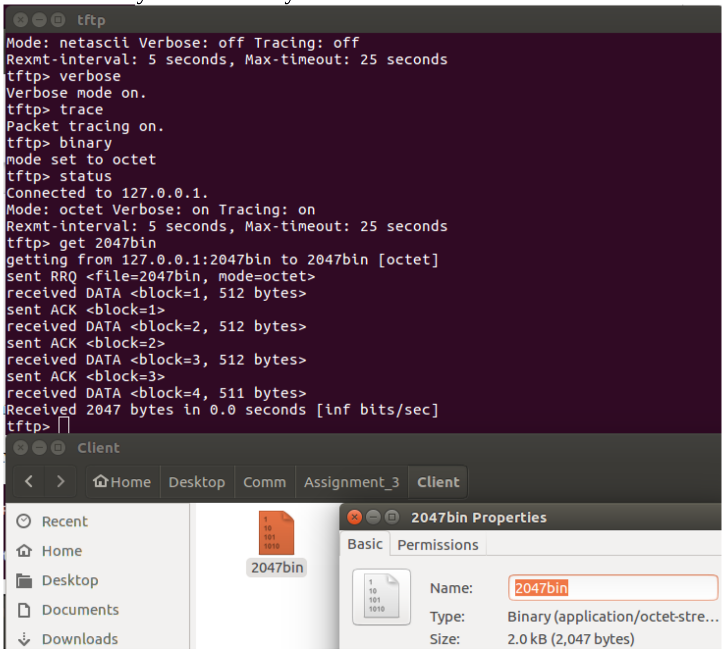
```
tftp
Connected to 127.0.0.1.
Mode: netascii Verbose: on Tracing: on
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp> get twoCR.txt
getting from 127.0.0.1:twoCR.txt to twoCR.txt [netascii]
sent RRQ <file=twoCR.txt, mode=netascii>
received DATA <block=1, 10 bytes>
Received 10 bytes in 0.0 seconds [inf bits/sec]
tftp> binary
mode set to octet
tftp> get 2048bin
getting from 127.0.0.1:2048bin to 2048bin [octet]
sent RRQ <file=2048bin, mode=octet>
received DATA <block=1, 512 bytes>
sent ACK <block=1>
received DATA <block=2, 512 bytes>
sent ACK <block=2>
received DATA <block=3, 512 bytes>
sent ACK <block=3>
received DATA <block=4, 512 bytes>
sent ACK <block=4>
received DATA <block=5, 0 bytes>
Received 2048 bytes in 0.0 seconds [inf bits/sec]
tftp>
```



Name: 2048bin
Type: Binary (application/octet-stre…
Size: 2.0 kB (2,048 bytes)

- After setting binary in client, we can see 2048 bin file to transfer from server to client properly.

(2) transfer a binary file of 2047 bytes and check that it matches the source file

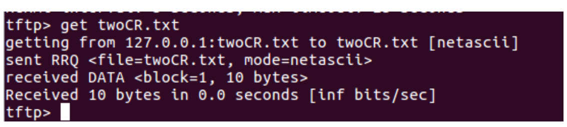


```
tftp
Mode: netascii Verbose: off Tracing: off
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp> verbose
Verbose mode on.
tftp> trace
Packet tracing on.
tftp> binary
mode set to octet
tftp> status
Connected to 127.0.0.1.
Mode: octet Verbose: on Tracing: on
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
tftp> get 2047bin
getting from 127.0.0.1:2047bin to 2047bin [octet]
sent RRQ <file=2047bin, mode=octet>
received DATA <block=1, 512 bytes>
sent ACK <block=1>
received DATA <block=2, 512 bytes>
sent ACK <block=2>
received DATA <block=3, 512 bytes>
sent ACK <block=3>
received DATA <block=4, 511 bytes>
Received 2047 bytes in 0.0 seconds [inf bits/sec]
tftp>
```

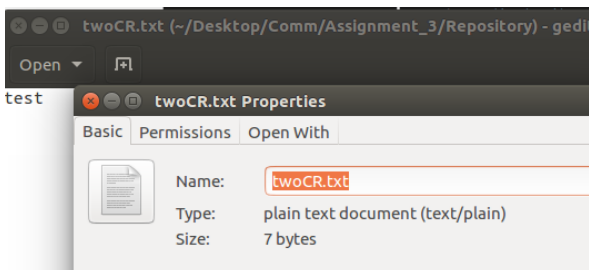
Name: 2047bin
Type: Binary (application/octet-stre…
Size: 2.0 kB (2,047 bytes)

- It works properly as expected. The different thing from test 1 is last received data for block 4 is 511 bytes.

(3) transfer a netascii file that includes two CR's and check that the resulting file matches the input file

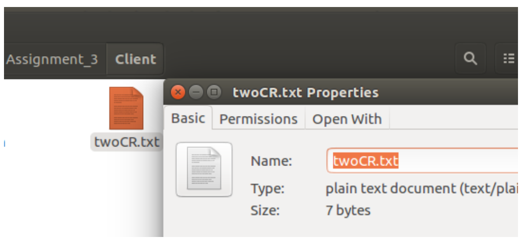


```
tftp> get twoCR.txt
getting from 127.0.0.1:twoCR.txt to twoCR.txt [netascii]
sent RRQ <file=twoCR.txt, mode=netascii>
received DATA <block=1, 10 bytes>
Received 10 bytes in 0.0 seconds [inf bits/sec]
tftp>
```



```
test
```

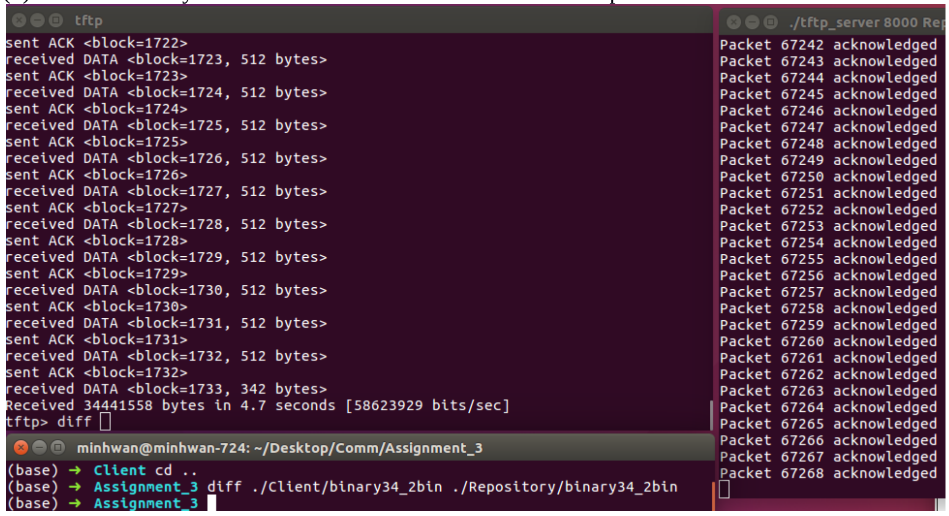
Name: twoCR.txt
Type: plain text document (text/plain)
Size: 7 bytes



Name: twoCR.txt
Type: plain text document (text/plai
Size: 7 bytes

- As test, received data was same as original one which contained CR.

(4) transfer a binary file of 34 MB and see if block number wrap-around works



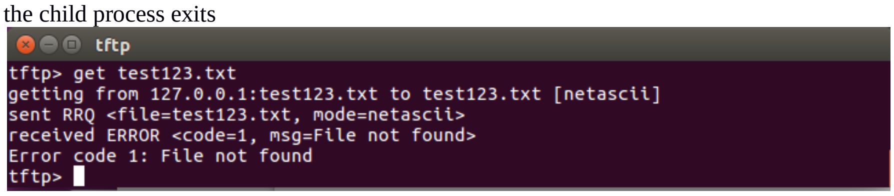
```
tftp
sent ACK <block=1722>
received DATA <block=1723, 512 bytes>
sent ACK <block=1723>
received DATA <block=1724, 512 bytes>
sent ACK <block=1724>
received DATA <block=1725, 512 bytes>
sent ACK <block=1725>
received DATA <block=1726, 512 bytes>
sent ACK <block=1726>
received DATA <block=1727, 512 bytes>
sent ACK <block=1727>
received DATA <block=1728, 512 bytes>
sent ACK <block=1728>
received DATA <block=1729, 512 bytes>
sent ACK <block=1729>
received DATA <block=1730, 512 bytes>
sent ACK <block=1730>
received DATA <block=1731, 512 bytes>
sent ACK <block=1731>
received DATA <block=1732, 512 bytes>
sent ACK <block=1732>
received DATA <block=1733, 342 bytes>
Received 34441558 bytes in 4.7 seconds [58623929 bits/sec]
tftp> diff
```

```
minhwan@minhwan-724: ~/Desktop/Comm/Assignment_3
(base) ➜ Client cd ..
(base) ➜ Assignment_3 diff ./Client/binary34_2bin ./Repository/binary34_2bin
(base) ➜ Assignment_3
```

```
./tftp_server 8000 Rep
Packet 67242 acknowledged
Packet 67243 acknowledged
Packet 67244 acknowledged
Packet 67245 acknowledged
Packet 67246 acknowledged
Packet 67247 acknowledged
Packet 67248 acknowledged
Packet 67249 acknowledged
Packet 67250 acknowledged
Packet 67251 acknowledged
Packet 67252 acknowledged
Packet 67253 acknowledged
Packet 67254 acknowledged
Packet 67255 acknowledged
Packet 67256 acknowledged
Packet 67257 acknowledged
Packet 67258 acknowledged
Packet 67259 acknowledged
Packet 67260 acknowledged
Packet 67261 acknowledged
Packet 67262 acknowledged
Packet 67263 acknowledged
Packet 67264 acknowledged
Packet 67265 acknowledged
Packet 67266 acknowledged
Packet 67267 acknowledged
Packet 67268 acknowledged
```

- As a result with diff function, we could see client could receive 34MB file properly. However, sender was marked 67268 packets as log and client was 1733 because the maximum packet client can log as 65535, so after that the packet number was reset to 0, then started again. The functional point of view, client worked fine.

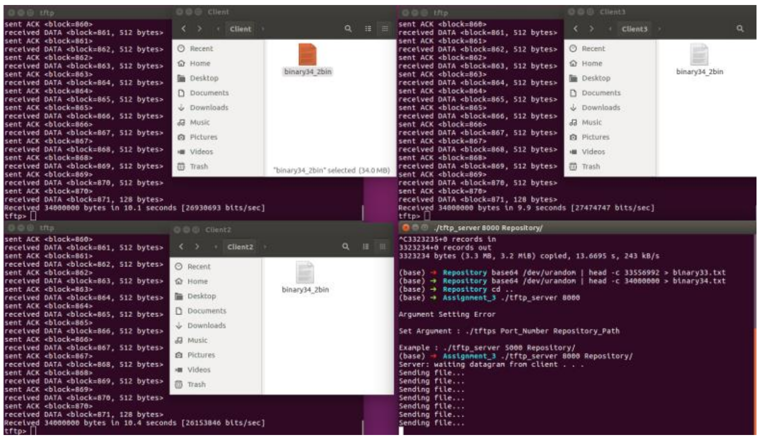
(5) check that you receive an error message if you try to transfer a file that does not exist and that your server cleans up and the child process exits



```
tftp
tftp> get test123.txt
getting from 127.0.0.1:test123.txt to test123.txt [netascii]
sent RRQ <file=test123.txt, mode=netascii>
received ERROR <code=1, msg=File not found>
Error code 1: File not found
tftp>
```

- If client requested some file that didn't exist, we could check that server cleaned up and the child process existed.

(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)
- It works properly. To have this test, three clients to receive 34MB file at same time. Comparing to running one client, running time is linearly increased because server technically can't send data at same time. Instead, server sends data to clients in order.

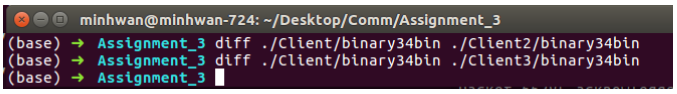


```
tftp
sent ACK <block=860>
received DATA <block=861, 512 bytes>
sent ACK <block=861>
received DATA <block=862, 512 bytes>
sent ACK <block=862>
received DATA <block=863, 512 bytes>
sent ACK <block=863>
received DATA <block=864, 512 bytes>
sent ACK <block=864>
received DATA <block=865, 512 bytes>
sent ACK <block=865>
received DATA <block=866, 512 bytes>
sent ACK <block=866>
received DATA <block=867, 512 bytes>
sent ACK <block=867>
received DATA <block=868, 512 bytes>
sent ACK <block=868>
received DATA <block=869, 512 bytes>
sent ACK <block=869>
received DATA <block=870, 512 bytes>
sent ACK <block=870>
received DATA <block=871, 128 bytes>
Received 34000000 bytes in 10.1 seconds [26930693 bits/sec]
tftp>
```

```
tftp
sent ACK <block=860>
received DATA <block=861, 512 bytes>
sent ACK <block=861>
received DATA <block=862, 512 bytes>
sent ACK <block=862>
received DATA <block=863, 512 bytes>
sent ACK <block=863>
received DATA <block=864, 512 bytes>
sent ACK <block=864>
received DATA <block=865, 512 bytes>
sent ACK <block=865>
received DATA <block=866, 512 bytes>
sent ACK <block=866>
received DATA <block=867, 512 bytes>
sent ACK <block=867>
received DATA <block=868, 512 bytes>
sent ACK <block=868>
received DATA <block=869, 512 bytes>
sent ACK <block=869>
received DATA <block=870, 512 bytes>
sent ACK <block=870>
received DATA <block=871, 128 bytes>
Received 34000000 bytes in 9.9 seconds [27474747 bits/sec]
tftp>
```

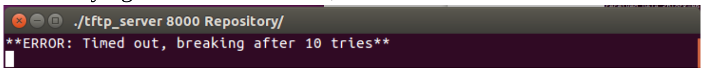
```
tftp
sent ACK <block=860>
received DATA <block=861, 512 bytes>
sent ACK <block=861>
received DATA <block=862, 512 bytes>
sent ACK <block=862>
received DATA <block=863, 512 bytes>
sent ACK <block=863>
received DATA <block=864, 512 bytes>
sent ACK <block=864>
received DATA <block=865, 512 bytes>
sent ACK <block=865>
received DATA <block=866, 512 bytes>
sent ACK <block=866>
received DATA <block=867, 512 bytes>
sent ACK <block=867>
received DATA <block=868, 512 bytes>
sent ACK <block=868>
received DATA <block=869, 512 bytes>
sent ACK <block=869>
received DATA <block=870, 512 bytes>
sent ACK <block=870>
received DATA <block=871, 128 bytes>
Received 34000000 bytes in 10.4 seconds [26153846 bits/sec]
tftp>
```

```
./tftp_server 8000 Repository/
^C3323235+0 records in
3323234+0 records out
3323234 bytes (3.3 MB, 3.2 MiB) copied, 13.6695 s, 243 kB/s

(base) ➜ Repository base64 /dev/urandom | head -c 33556992 > binary33.txt
(base) ➜ Repository base64 /dev/urandom | head -c 34000000 > binary34.txt
(base) ➜ Repository cd ..
(base) ➜ Assignment_3 ./tftp_server 8000

Argument Setting Error

Set Argument : ./tftps Port_Number Repository_Path

Example : ./tftp_server 5000 Repository/
(base) ➜ Assignment_3 ./tftp_server 8000 Repository/
Server: waiting datagram from client . . .
Sending file...
Sending file...
Sending file...
Sending file...
Sending file...
Sending file...
Sending file...
```



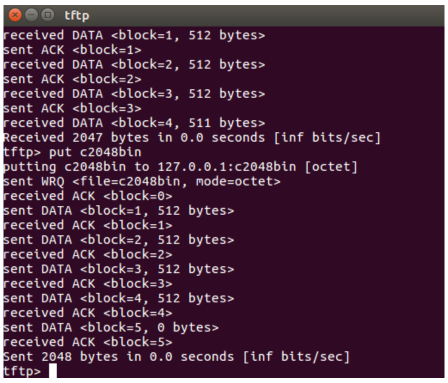
```
minhwan@minhwan-724: ~/Desktop/Comm/Assignment_3
(base) ➜ Assignment_3 diff ./Client/binary34bin ./Client2/binary34bin
(base) ➜ Assignment_3 diff ./Client/binary34bin ./Client3/binary34bin
(base) ➜ Assignment_3
```

(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)
- After trying to transmit 10 times, server occurs time outs

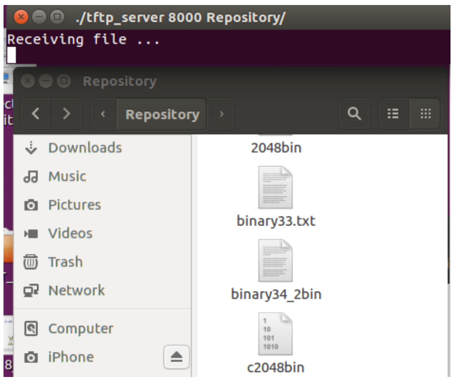


```
./tftp_server 8000 Repository/
**ERROR: Timed out, breaking after 10 tries**
```

(8) Transfer a binary file of 2048 bytes to server and check that it matches the source file (Bonus Feature)
- We can see 2048 bin file to transfer from client to server properly. Also, we can check WRQ in log as well.

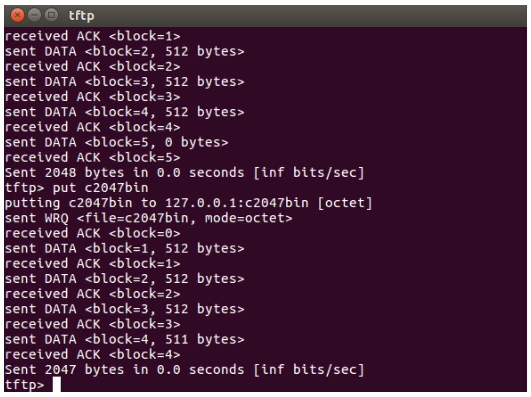


```
tftp
received DATA <block=1, 512 bytes>
sent ACK <block=1>
received DATA <block=2, 512 bytes>
sent ACK <block=2>
received DATA <block=3, 512 bytes>
sent ACK <block=3>
received DATA <block=4, 511 bytes>
Received 2047 bytes in 0.0 seconds [inf bits/sec]
tftp> put c2048bin
putting c2048bin to 127.0.0.1:c2048bin [octet]
sent WRQ <file=c2048bin, mode=octet>
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=1>
sent DATA <block=2, 512 bytes>
received ACK <block=2>
sent DATA <block=3, 512 bytes>
received ACK <block=3>
sent DATA <block=4, 512 bytes>
received ACK <block=4>
sent DATA <block=5, 0 bytes>
received ACK <block=5>
Sent 2048 bytes in 0.0 seconds [inf bits/sec]
tftp>
```



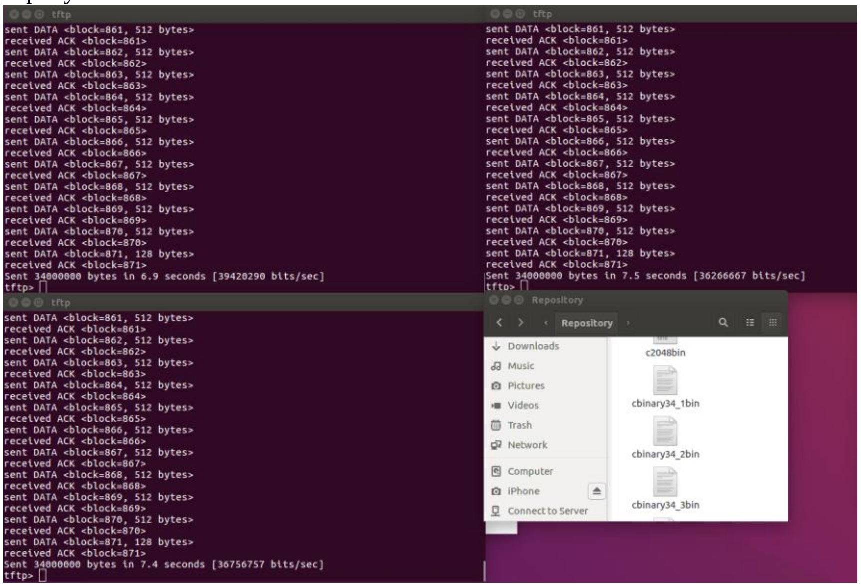
```
./tftp_server 8000 Repository/
Receiving file ...
```

(9) Transfer a binary file of 2047 bytes to server and check that it matches the source file (Bonus Feature)
- We can see 2047 bin file to transfer from client to server properly. Also, we can check WRQ in log as well. Different thing from test case 8 is last block 4 transmits 511 bytes as expected



```
tftp
received ACK <block=1>
sent DATA <block=2, 512 bytes>
received ACK <block=2>
sent DATA <block=3, 512 bytes>
received ACK <block=3>
sent DATA <block=4, 512 bytes>
received ACK <block=4>
sent DATA <block=5, 0 bytes>
received ACK <block=5>
Sent 2048 bytes in 0.0 seconds [inf bits/sec]
tftp> put c2047bin
putting c2047bin to 127.0.0.1:c2047bin [octet]
sent WRQ <file=c2047bin, mode=octet>
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=1>
sent DATA <block=2, 512 bytes>
received ACK <block=2>
sent DATA <block=3, 512 bytes>
received ACK <block=3>
sent DATA <block=4, 511 bytes>
received ACK <block=4>
Sent 2047 bytes in 0.0 seconds [inf bits/sec]
tftp>
```

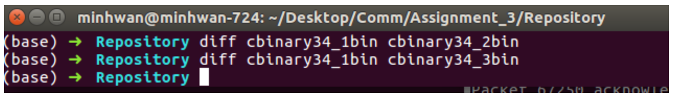
(10) Connect to the TFTP server with three clients simultaneously and test that the transfers work correctly – transmitting file to server simultaneously (Bonus Feature)
- As you can see below picture, 3 clients can send their files to server at same time. And, checked all files were sent properly.



```
sent DATA <block=861, 512 bytes>
received ACK <block=861>
sent DATA <block=862, 512 bytes>
received ACK <block=862>
sent DATA <block=863, 512 bytes>
received ACK <block=863>
sent DATA <block=864, 512 bytes>
received ACK <block=864>
sent DATA <block=865, 512 bytes>
received ACK <block=865>
sent DATA <block=866, 512 bytes>
received ACK <block=866>
sent DATA <block=867, 512 bytes>
received ACK <block=867>
sent DATA <block=868, 512 bytes>
received ACK <block=868>
sent DATA <block=869, 512 bytes>
received ACK <block=869>
sent DATA <block=870, 512 bytes>
received ACK <block=870>
sent DATA <block=871, 128 bytes>
received ACK <block=871>
Sent 34000000 bytes in 6.9 seconds [39420290 bits/sec]
tftp>
```

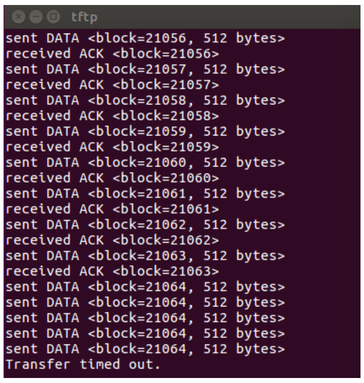
```
sent DATA <block=861, 512 bytes>
received ACK <block=861>
sent DATA <block=862, 512 bytes>
received ACK <block=862>
sent DATA <block=863, 512 bytes>
received ACK <block=863>
sent DATA <block=864, 512 bytes>
received ACK <block=864>
sent DATA <block=865, 512 bytes>
received ACK <block=865>
sent DATA <block=866, 512 bytes>
received ACK <block=866>
sent DATA <block=867, 512 bytes>
received ACK <block=867>
sent DATA <block=868, 512 bytes>
received ACK <block=868>
sent DATA <block=869, 512 bytes>
received ACK <block=869>
sent DATA <block=870, 512 bytes>
received ACK <block=870>
sent DATA <block=871, 128 bytes>
received ACK <block=871>
Sent 34000000 bytes in 7.5 seconds [36266667 bits/sec]
tftp>
```

```
sent DATA <block=861, 512 bytes>
received ACK <block=861>
sent DATA <block=862, 512 bytes>
received ACK <block=862>
sent DATA <block=863, 512 bytes>
received ACK <block=863>
sent DATA <block=864, 512 bytes>
received ACK <block=864>
sent DATA <block=865, 512 bytes>
received ACK <block=865>
sent DATA <block=866, 512 bytes>
received ACK <block=866>
sent DATA <block=867, 512 bytes>
received ACK <block=867>
sent DATA <block=868, 512 bytes>
received ACK <block=868>
sent DATA <block=869, 512 bytes>
received ACK <block=869>
sent DATA <block=870, 512 bytes>
received ACK <block=870>
sent DATA <block=871, 128 bytes>
received ACK <block=871>
Sent 34000000 bytes in 7.4 seconds [36756757 bits/sec]
tftp>
```



```
minhwan@minhwan-724: ~/Desktop/Comm/Assignment_3/Repository
(base) ➜ Repository diff cbinary34_1bin cbinary34_2bin
(base) ➜ Repository diff cbinary34_1bin cbinary34_3bin
(base) ➜ Repository
```

(11) Terminate the TFTP server in the middle of a transfer and see if your TFTP client recognizes it that the server is no longer there (Bonus Feature)
- Of course, it could be not our project range because client is not developed from us. However, for make sure, we tested client side as well. As you can see, after trying 5 times, client occurs time out.



```
sent DATA <block=21056, 512 bytes>
received ACK <block=21056>
sent DATA <block=21057, 512 bytes>
received ACK <block=21057>
sent DATA <block=21058, 512 bytes>
received ACK <block=21058>
sent DATA <block=21059, 512 bytes>
received ACK <block=21059>
sent DATA <block=21060, 512 bytes>
received ACK <block=21060>
sent DATA <block=21061, 512 bytes>
received ACK <block=21061>
sent DATA <block=21062, 512 bytes>
received ACK <block=21062>
sent DATA <block=21063, 512 bytes>
received ACK <block=21063>
sent DATA <block=21064, 512 bytes>
sent DATA <block=21064, 512 bytes>
sent DATA <block=21064, 512 bytes>
sent DATA <block=21064, 512 bytes>
sent DATA <block=21064, 512 bytes>
Transfer timed out.
```

Model codes

Makefile

```
PROGS = tftp_server
CLEANFILES = *.o
CC = gcc

all:        ${PROGS}

tftp_server:        tftp_server.o
                    ${CC} -o $@ tftp_server.o
```

```
clean:
	rm -f ${PROGS} ${CLEANFILES}
```

tftp_server.c

```
/*
*TFTP Server
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/socket.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <stdbool.h>
#include <signal.h>


#define MAX_DATA_SIZE 512
#define MAX_ERR_SIZE 512
#define MAX_PORT_NUMBER 65535
#define MAX_PORT_NUMBER_SIZE 7
#define MAX_BUFFER_SIZE 2048
#define MAX_FILE_PATH 1024
#define OPCODE_SIZE 2
#define BLK_LIMIT 65536

#define NETASCII "netascii"
#define OCTET "octet"
#define RETRY_LIMIT 10

typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;

//Packet structures
struct _ERRPAKT
{
   uint16_t opcode;
   uint16_t errCode;
   char errMsg[MAX_ERR_SIZE];
};
typedef struct _ERRPAKT ERRPAKT;

struct _ACKPAKT
{
   uint16_t opcode;
   uint16_t blkNumber;
};
typedef struct _ACKPAKT ACKPAKT;

struct _REQPAKT
{
   uint16_t opcode;
```

```
    char fileName[MAX_FILE_PATH];
};
typedef struct _REQPAKT REQPAKT;

struct _DATAPAKT
{
    uint16_t opcode;
    uint16_t blkNumber;
    char data[MAX_DATA_SIZE];
};
typedef struct _DATAPAKT DATAPAKT;


//Error Message
enum {
    NOT_DEF,
    FILE_NOT_FOUND,
    ACCESS_VIOLATION,
    DISK_FULL_ALLOCATION_EXCEEDED,
    ILLEGAL_TFTP_OPER,
    UNKNOWN_TRANSFER_ID,
    FILE_ALREADY_EXISTS,
    NO_SUCH_USER
};

static char *errMsg[] = {
    "Not defined",
    "File not found",
    "Access violation",
    "Disk full or allocation exceeded",
    "Illegal TFTP operation",
    "Unknown transfer ID",
    "File already exists",
    "No such user"
};

//Transmission mode
enum {
    MODE_NETASCII,
    MODE_OCTET
};

enum {
    NONE,
    RRQ,
    WRQ,
    DATA,
    ACK,
    ERR
};


// print errors using Error numbers
void ErrPrint (const char* Err_Name)
{
    char *errMsg = NULL;
    errMsg = strerror (errno);
    if (NULL != Err_Name)
        printf ("[Error] Server: '%s' error has occured\n", Err_Name);
    printf ("[Error] Server: %s\n", errMsg);
    return;
}


//Signal reap all death process
void sigchld_handler(int s)
{
    int tmp_errno = errno;
    while(waitpid(-1, NULL, WNOHANG) > 0);
    errno = tmp_errno;
}

//Send error message
void ErrSend (int errNum, struct sockaddr_in client_Addr)
{
    struct sockaddr_in child_Addr;
    ERRPAKT errPakt ;
    socklen_t addrlen = sizeof (struct sockaddr_in);

    int child_fd = 0;
    int errType = FILE_NOT_FOUND;
    int bufferSize = 0;

    memset (&child_Addr, 0, sizeof (struct sockaddr_in));
    memset (&errPakt, 0, sizeof (ERRPAKT));


    if ((child_fd = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        ErrPrint ("socket");
        return;
    }

    child_Addr.sin_family = AF_INET;
    child_Addr.sin_addr.s_addr = htonl (INADDR_ANY);
    child_Addr.sin_port = htons(0);

    if (bind(child_fd, (struct sockaddr*) &child_Addr, sizeof (child_Addr)) < 0)
    {
        ErrPrint ("bind");
        return;
    }

    errPakt.opcode = htons(ERR);

    if (errNum == EACCES)
        errType = ACCESS_VIOLATION;

    errPakt.errCode = htons(errType);

    snprintf (errPakt.errMsg, sizeof (errPakt.errMsg), "%s", errMsg[errType]);
    bufferSize = sizeof (errPakt.opcode) + sizeof (errPakt.errCode) + strlen (errPakt.errMsg) + 1;
    sendto (child_fd, (void *)(&errPakt), bufferSize, 0, (struct sockaddr *) &client_Addr, addrlen);
}


//Send data (return 1 for success, 0 for false)
int DataSend (char* fileDir, char* mode_Str, struct sockaddr_in client_Addr)
```

```
{
   DATAPAKT dataPakt;
   struct sockaddr_in child_Addr;
   struct timeval tv;
   ACKPAKT ackPakt;
   int retryCnt = 0;
   int pakcnt = 1;
   int mode = MODE_NETASCII;
   int child_fd = 0;
   int status = 0;
   int totalData = 0;
   char character = '\0';
   char next_Char = -1;
   // takes care of '\r' and '\n' in ascii mode

   char buffer [MAX_DATA_SIZE] = {0};
   bool isEOFReached = false;
   FILE *fp = NULL;
   socklen_t addrlen = sizeof (struct sockaddr_in);
   uint16_t blockCnt = 1;
   uint16_t cnt = 0;
   uint16_t buffLen = 0;
   uint16_t buffIndex = 0;
   size_t bufferSize = 0;
   fd_set readFds;


   if ((NULL == mode_Str) || (NULL == fileDir))
   {
      printf ("[Error] Server: file Name / sockAddr sent is NULL \n");
      return 0;
   }

   memset (&dataPakt, 0, sizeof (DATAPAKT));
   memset (&ackPakt, 0, sizeof (ACKPAKT));
   memset (&child_Addr, 0, sizeof (struct sockaddr_in));

   if (0 == strcmp (mode_Str , OCTET))
   {
      mode = MODE_OCTET;
   }

   fp = fopen(fileDir, "r");

   if (NULL == fp)
   {
      ErrSend (errno, client_Addr);
      return 0;
   }
   // Start to send the data-----------------------
   if ((child_fd = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
   {
      ErrPrint ("socket");
      return 0;
   }
   printf("Sending file...\n");

   // Child server
   child_Addr.sin_family = AF_INET;
   child_Addr.sin_addr.s_addr = htonl (INADDR_ANY);
   child_Addr.sin_port = htons(0);

   if (bind(child_fd, (struct sockaddr*) &child_Addr, sizeof (child_Addr)) < 0)
   {
      ErrPrint ("bind");
      return 0;
   }

   FD_ZERO (&readFds);
   FD_SET (child_fd, &readFds);
   tv.tv_sec = 10;
   tv.tv_usec = 0;

   while (1)
   {
      // Copy 512 bytes of data or less
      for (cnt = 0; cnt < MAX_DATA_SIZE; cnt++)
      {
         if ((buffIndex == buffLen) && (false == isEOFReached))
         {
            memset (buffer, 0, sizeof (buffer));
            buffIndex = 0;
            buffLen = fread (buffer, 1, sizeof (buffer), fp);
            totalData = totalData + buffLen;

            if (buffLen < MAX_DATA_SIZE)
               isEOFReached = true;
            if (ferror (fp))
               printf ("[Error] Server: Read error\n");
         }
         if (next_Char >= 0) {
            dataPakt.data [cnt] = next_Char;
            next_Char = -1;
            continue;
         }
         // take care of the data set which is less than MAX_DATA
         if (buffIndex >= buffLen)
         {
            break;
         }
         character = buffer [buffIndex];
         buffIndex ++;
         if (MODE_NETASCII == mode)
         {
            if ('\n' == character)
            {
               next_Char = character;
               character = '\r';
            }
            else if ('\r' == character)
            {
               next_Char = '\0';
            }
            else
            {
               next_Char = -1;
            }
         }
```

```
          dataPakt.data [cnt] = character;
       }
                dataPakt.opcode = htons (DATA);
       dataPakt.blkNumber = htons (blockCnt % BLK_LIMIT);
       bufferSize = sizeof(dataPakt.opcode) + sizeof (dataPakt.blkNumber) + (cnt);
       do
       {
          sendto (child_fd, (void *)(&dataPakt), bufferSize, 0, (struct sockaddr *) &client_Addr, addrlen);
          if ((status = select (child_fd + 1, &readFds, NULL, NULL, &tv)) <= 0)
          {
             retryCnt ++;
             continue;
          }
          else
          {
                 if ((status = recvfrom (child_fd, (void *)&ackPakt, sizeof (ackPakt), 0, (struct sockaddr*)
&client_Addr, &addrlen)) >= 0)
             {
                if ((ACK == ntohs (ackPakt.opcode)) && (blockCnt == (ntohs (ackPakt.blkNumber))))
                   break;
             }
          }
          memset (&ackPakt, 0, sizeof (ACKPAKT));
       }while (retryCnt < RETRY_LIMIT);
       if (retryCnt >= RETRY_LIMIT)
       {
          printf ("[Error] Server: Timed out, breaking after %d tries \n", retryCnt);
          break;
       }
       blockCnt++;
       memset (&dataPakt, 0, sizeof (DATAPAKT));
       retryCnt = 0;
       if ((cnt < MAX_DATA_SIZE) && (buffIndex == buffLen))
       {
          break;
       }

       printf("Packet %d acknowledged\n", pakcnt);
       pakcnt++;
    }

    fclose (fp);

    return (1);
}


//Receives file from Client (return 1 for success, 0 for false)
int recvData (char* fileDir, char* mode_Str, struct sockaddr_in client_Addr)
{
    DATAPAKT dataPakt;
    ACKPAKT ackPakt;
    struct sockaddr_in child_Addr;
    struct timeval tv;
    int status = 0;
    int child_fd = 0;
    int mode = MODE_NETASCII;
    char character = '\0';
    FILE *fp = NULL;
```

```
uint16_t blkCnt = 0;
uint16_t cnt = 0;
uint16_t byteLen = 0;
uint16_t buffLen = MAX_DATA_SIZE;
socklen_t addrlen = sizeof (struct sockaddr_in);
fd_set readFds;
bool err_Occr = false;
bool lastCharWasCR = false;

if ((NULL == fileDir) || (NULL == mode_Str))
{
   printf ("[Error] Server: '%s' file Name / sockAddr sent is NULL \n", __FILE__);

   return 0;
}

memset (&dataPakt, 0, sizeof (DATAPAKT));
memset (&ackPakt, 0, sizeof (ACKPAKT));
memset (&child_Addr, 0, sizeof (struct sockaddr_in));

if (0 == strcmp (mode_Str , OCTET))
{
   mode = MODE_OCTET;
}

fp = fopen(fileDir, "w");

if (NULL == fp)
{
   ErrSend (errno, client_Addr);
   return 0;
}

// Start to recv ----------------------
if ((child_fd = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
{
   ErrPrint ("socket");
   return 0;
}

// Child server
child_Addr.sin_family = AF_INET;
child_Addr.sin_addr.s_addr = htonl (INADDR_ANY);
child_Addr.sin_port = htons(0);

if (bind(child_fd, (struct sockaddr*) &child_Addr, sizeof (child_Addr)) < 0)
{
   ErrPrint ("bind");

   return 0;
}
printf("Receiving file ...\n");


         /*******************************************************
          *
          * FD is file descriptor, and can use it to include select.h
          * below is FD structure function
          *
          * #defineFD_SET(fd, fdsetp)__FD_SET (fd, fdsetp)
          * #defineFD_CLR(fd, fdsetp)              __FD_CLR (fd, fdsetp)
          * #defineFD_ISSET(fd, fdsetp)            __FD_ISSET (fd, fdsetp)
          * #defineFD_ZERO(fdsetp)                 __FD_ZERO (fdsetp)
          *
          *
          *****************************************************/

   //FD initialization
   FD_ZERO (&readFds);
   FD_SET (child_fd, &readFds);
   tv.tv_sec = 10;
   tv.tv_usec = 0;

   while (1)
   {
      ackPakt.opcode = htons (ACK);
      ackPakt.blkNumber = blkCnt;
      sendto (child_fd, (void *)(&ackPakt), sizeof(ackPakt), 0, (struct sockaddr *) &client_Addr, addrlen);
      if ((buffLen != MAX_DATA_SIZE) || (true == err_Occr))
      {
         break;
      }
      if ((status = select (child_fd + 1, &readFds, NULL, NULL, &tv)) <= 0)
      {
         if (0 == status)
         {
            printf ("Server: Timedout while waiting for packet from the host \n");
            break;
         }
         else if (0 > status)
         {
            ErrPrint("select");
            break;
         }
      }
      else
      {
         if ((byteLen = recvfrom (child_fd, (void *)&dataPakt, sizeof (dataPakt), 0, (struct sockaddr*)
&client_Addr, &addrlen)) >= 0)
         {
            if (DATA != ntohs (dataPakt.opcode))
            {
               continue;
            }
            buffLen = byteLen - sizeof(dataPakt.opcode) - sizeof(dataPakt.blkNumber);
            blkCnt = dataPakt.blkNumber;

            while (cnt < buffLen)
            {
               if (MODE_NETASCII == mode)
               {
                  if (true == lastCharWasCR)
                  {
                     character = dataPakt.data [cnt];
                     if ((0 == cnt) && ('\0' == dataPakt.data [cnt]))
                     {
                        character = '\r';
                     }
```

```
                    cnt ++;
                    lastCharWasCR = false;
                }
                else if ('\r' == dataPakt.data [cnt])
                {
                    cnt ++;
                    if (('\n' == dataPakt.data [cnt]) && (cnt < buffLen))
                    {
                        character = dataPakt.data [cnt];
                        cnt ++;
                    }
                    else if (('\0' == dataPakt.data [cnt]) && (cnt < buffLen))
                    {
                        character = dataPakt.data [cnt - 1];
                        cnt ++;
                    }
                    else
                    {
                      if (cnt >= buffLen)
                      {
                         if (dataPakt.data [cnt - 1] == '\r')
                         {
                             lastCharWasCR = true;
                         }
                         continue;
                      }
                    }
                }
                else
                {
                    character = dataPakt.data [cnt];
                    cnt ++;
                }
            }
            else if (MODE_OCTET == mode)
            {
                character = dataPakt.data [cnt];
                cnt ++;
            }

            if (EOF == putc (character, fp))
            {

                ErrPrint("putc");
                printf ("[Error] Server: : There was an error while writing to the file \n");
                err_Occr = true;
                break;
            }
        }
        cnt  = 0;
        memset (&dataPakt, 0, sizeof (dataPakt));
      }
    }
  }

  fclose (fp);

  return 1;
}
```

```
//Request handling
void handleRequest (void* buffer, struct sockaddr_in client_Addr, char *tftpFolderPath)
{
   int requestType = NONE;
   REQPAKT *reqPakt = buffer;
   char fileDir [MAX_FILE_PATH]= {0};

   //Parse request
   if ((NULL == buffer) || (NULL == reqPakt))
      return;
   reqPakt->opcode = ntohs(*((uint16_t*)buffer));


   if ((NULL == reqPakt) || (NULL == tftpFolderPath))
   {
      ErrPrint ("handleRequest");
      return;
   }

   switch (reqPakt->opcode)
   {

      case WRQ:
         {
            strncat (fileDir, tftpFolderPath, sizeof (fileDir));
            strcat (fileDir, "/");
            strncat (fileDir, reqPakt->fileName, (sizeof (fileDir) - strlen (fileDir)));
            recvData (fileDir, (char *)(buffer + OPCODE_SIZE + strlen (reqPakt->fileName) + 1),
client_Addr);
         }
         break;

      case RRQ:
         {
            strncat (fileDir, tftpFolderPath, sizeof (fileDir));
            strcat (fileDir, "/");
            strncat (fileDir, reqPakt->fileName, (sizeof (fileDir) - strlen (fileDir)));
            DataSend (fileDir, (char *)(buffer + OPCODE_SIZE + strlen (reqPakt->fileName) + 1),
client_Addr);
         }
         break;

      default :
         printf ("**Invalid request from the client**\n");
   }
}

int main (int argc, char** argv)
{
   int listenFd = 0;
   int pid = 0;
   int ret = 0;
   struct sigaction sa;
   struct sockaddr_in servAddr;
   struct sockaddr_in client_Addr;
   socklen_t addrlen = sizeof(struct sockaddr_in);
   char port[MAX_PORT_NUMBER_SIZE] = {0};
char buffer[MAX_BUFFER_SIZE] = {0};
char ip[INET_ADDRSTRLEN] = {0};

if (argc != 3)
{
   printf ("\nArgument Setting Error \n"
           "\nSet Argument : ./tftps Port_Number Repository_Path\n"
           "\nExample : ./tftp_server 5000 Repository/\n");
   return 1;
}

if ((listenFd = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
{
   ErrPrint ("socket");
   return 1;
}

memset (&servAddr, 0, sizeof (servAddr));
memset (&client_Addr, 0, sizeof (client_Addr));

if (atoi (argv[1]) > MAX_PORT_NUMBER)
{
   printf ("[Error] Server: The Port number exceeded the maximum range 65535 \n"
           "        A port number is within the range of 1025~65535 \n");
   return 1;
}
else if (atoi (argv[1]) <= 0)
{
   printf ("[Error] Server: The Port number is less than 1\n"
           "        A port number is within the range of 1025~65535 \n");
   return 1;
}

// Server Information
servAddr.sin_family = AF_INET;
           //servaddr.sin_family      = AF_INET6;
servAddr.sin_addr.s_addr = htonl (INADDR_ANY);
servAddr.sin_port = htons(atoi(argv[1]));
           //inet_pton(AF_INET6, argv[1], &servaddr.sin_addr);

sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
   perror("sigaction");
   exit(1);
}

//take care of the scenario where the PORT number provided is already in use
while (bind(listenFd, (struct sockaddr*) &servAddr, sizeof (servAddr)) < 0)
{
   // Check for port already in use error
   if (EADDRINUSE == errno)
   {
      printf ("[Port] : %s is already in use by another process, choose any other free port\n", argv [1]);
      printf ("[Port] : ");
      scanf ("%s", port);
      if (MAX_PORT_NUMBER < atoi (port))
      {
            printf ("[Error] Server: The Port number exceeded the maximum range 65535 \n"
                    "        A port number is within the range of 1025~65535 \n");
            return 1;
        }
        else if (0 >= atoi (port))
        {
            printf ("[Error] Server: The Port number is less than 1\n"
                    "        A port number is within the range of 1025~65535 \n");
            return 1;
        }
        servAddr.sin_port = htons(atoi(port));
    }
    else
    {
        ErrPrint ("bind");
        return 1;
    }
  }

  printf("Server: waiting datagram from client . . .\n");
  while (1)
  {
    // Initializing the variable for every loop
    memset (&client_Addr, 0, sizeof (client_Addr));
    memset (&buffer, 0, sizeof (buffer));
    addrlen = sizeof (client_Addr);
    recvfrom (listenFd, (void *)buffer, sizeof(buffer), 0, (struct sockaddr *)&client_Addr, &addrlen);
    inet_ntop( AF_INET, &(client_Addr.sin_addr), ip, INET_ADDRSTRLEN);

    if ((pid = fork()) ==  -1)
    {
        ErrPrint ("fork");
    }
    else if (pid == 0)
    {
        handleRequest ((void *)buffer, client_Addr, argv [2]);
        break;
    }
  }

  return 0;
}
```