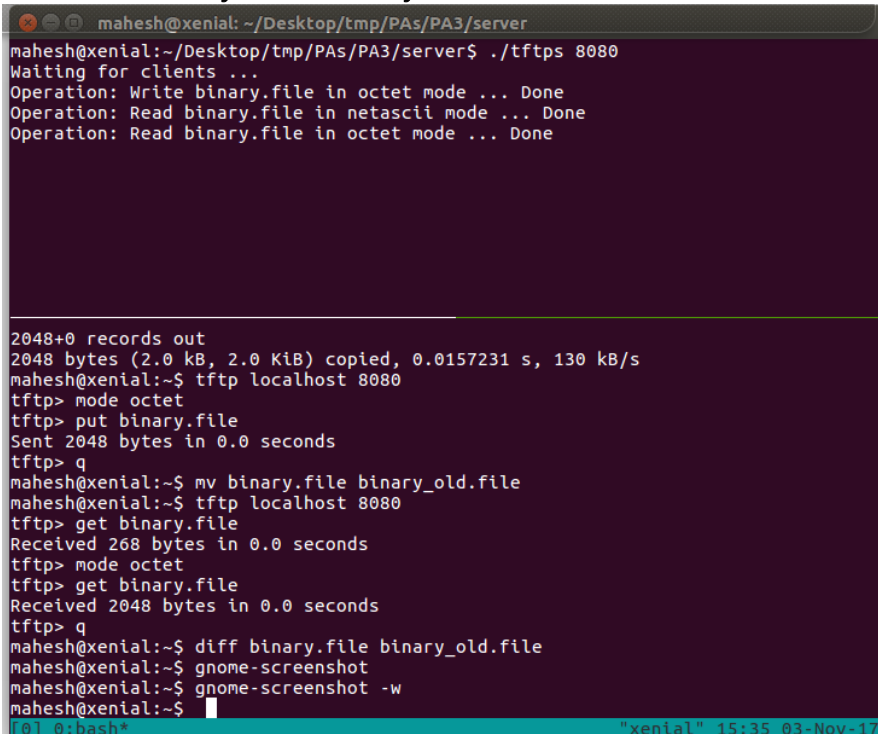


ReadMe:

In this programming assignment we implement a TFTP server. It uses the UDP to transfer file. A default TFTP client is used. Server is able to handle multiple file transfer. TFTP client makes a read request to read the files from the server. Server uses the fork function to handle multiple clients, whenever the server receives a new packet server has create a new port and start communicating on that port so that new connections can be accepted on the main port. We implement the stop and wait algorithm to ensure the reliable delivery of the messages. Timeouts are used to ensure that packets are retransmitted in case an ack is lost. Our client can also write to the server which is the bonus part of this assignment. There are two modes that are supported by the TFTP: net ASCII and Octet. 16 bit sequence numbers are used to break a large file into smaller chunks and transfer independently. As mentioned earlier TFTP uses the UDP to communicate, UDP is a connectionless unreliable protocol.

Test Cases:

1. transfer a binary file of 2048 bytes



```
maahesh@xenial: ~/Desktop/tmp/PAs/PA3/server
maahesh@xenial:~/Desktop/tmp/PAs/PA3/server$ ./tftps 8080
Waiting for clients ...
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in netascii mode ... Done
Operation: Read binary.file in octet mode ... Done

2048+0 records out
2048 bytes (2.0 kB, 2.0 KiB) copied, 0.0157231 s, 130 kB/s
maahesh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> put binary.file
Sent 2048 bytes in 0.0 seconds
tftp> q
maahesh@xenial:~$ mv binary.file binary_old.file
maahesh@xenial:~$ tftp localhost 8080
tftp> get binary.file
Received 268 bytes in 0.0 seconds
tftp> mode octet
tftp> get binary.file
Received 2048 bytes in 0.0 seconds
tftp> q
maahesh@xenial:~$ diff binary.file binary_old.file
maahesh@xenial:~$ gnome-screenshot
maahesh@xenial:~$ gnome-screenshot -w
maahesh@xenial:~$
```

2. transfer a binary file of 2047 bytes

```
maresh@xenial: ~/Desktop/tmp/PAs/PA3/server
maresh@xenial:~/Desktop/tmp/PAs/PA3/server$ ./tftps 8080
Waiting for clients ...
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in netascii mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done

maresh@xenial:~$ dd if=/dev/urandom of=binary.file bs=1 count=2047
2047+0 records in
2047+0 records out
2047 bytes (2.0 kB, 2.0 KiB) copied, 0.0153934 s, 133 kB/s
maresh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> put binary.file
Sent 2047 bytes in 0.0 seconds
tftp> q
maresh@xenial:~$ mv binary.file binary_old.file
maresh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> get binary.file
Received 2047 bytes in 0.0 seconds
tftp> q
maresh@xenial:~$ diff binary.file binary_old.file
maresh@xenial:~$
```

3. transfer a netascii file that includes two CR's

```
maresh@xenial: ~/Desktop/tmp/PAs/PA3/server
maresh@xenial:~/Desktop/tmp/PAs/PA3/server$ ./tftps 8080
Waiting for clients ...
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in netascii mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write dummy.txt in netascii mode ... Done
Operation: Read dummy.txt in netascii mode ... Done

maresh@xenial:~$ tftp localhost 8080
tftp> put dummy.txt
Sent 15 bytes in 0.0 seconds
tftp> q
maresh@xenial:~$ mv dummy.txt dummy_old.txt
maresh@xenial:~$ tftp localhost 8080
tftp> get dummy.txt
Received 15 bytes in 0.0 seconds
tftp> q
maresh@xenial:~$ diff dummy.txt dummy_old.txt
maresh@xenial:~$
```

4. transfer a binary file of 34 MB

```
maahesh@xenial: ~/Desktop/tmp/PAs/PA3/server
maahesh@xenial:~/Desktop/tmp/PAs/PA3/server$ ./tftps 8080
Waiting for clients ...
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in netascii mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write dummy.txt in netascii mode ... Done
Operation: Read dummy.txt in netascii mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done

maahesh@xenial:~$ dd if=/dev/urandom of=binary.file bs=1024 count=35000
35000+0 records in
35000+0 records out
35840000 bytes (36 MB, 34 MiB) copied, 2.14561 s, 16.7 MB/s
maahesh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> put binary.file
Sent 35840000 bytes in 1.4 seconds
tftp> q
maahesh@xenial:~$ mv binary.file binary_old.file
maahesh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> get binary.file
Received 35840000 bytes in 1.5 seconds
tftp> q
maahesh@xenial:~$ diff binary.file binary_old.file
maahesh@xenial:~$
```

[0] 0: bash* "xenial" 15:39 03-Nov-17

5. transfer a binary file of 34 MB

```
maahesh@xenial: ~/Desktop/tmp/PAs/PA3/server
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in netascii mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Write dummy.txt in netascii mode ... Done
Operation: Read dummy.txt in netascii mode ... Done
Operation: Write binary.file in octet mode ... Done
Operation: Read binary.file in octet mode ... Done
Operation: Read blah in netascii mode ... tftp.cpp:220 fopen(): No such file or
directory
Done

maahesh@xenial:~$ tftp localhost 8080
tftp> get blah
Error code 1: File Error!
tftp> q
maahesh@xenial:~$
```

[0] 0: tftp* "xenial" 15:40 03-Nov-17

6. Connect to the TFTP server with three clients simultaneously

```

M VirtualBox
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
tftp.cpp:296 Timeout Error
Done
Operation: Read binary.file in octet mode ... tftp.cpp:220 fopen(): No
such file or directory
Done
Operation: Read binary.file in octet mode ... Operation: Read binary.fi
le in octet mode ... Operation: Read binary.file in octet mode ... Done
Done
Done
Done

mahesh@xenial:~$ cd ~
mahesh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> get binary.file
^Received 102400000 bytes in 10.2 seconds
tftp>

mahesh@xenial:~/Desktop/tmp/PAs/PA3/server$ cd ~/Desktop/tmp/
mahesh@xenial:~/Desktop/tmp$ tftp localhost 8080
tftp> mode octet
tftp> get binary.file
Error code 1: File Error!
tftp> get binary.file
Received 102400000 bytes in 8.1 seconds
tftp>

Oracle VM VirtualBox Manager
File Machine Help
[0] 0:tftp*
15:43 03-Nov-17
```

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

```

mahesh@xenial: ~/Desktop/tmp/PAs/PA3/server
Operation: Read binary.file in octet mode ... Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
Timeout!
tftp.cpp:296 Timeout Error
Done

mahesh@xenial:~$ dd if=/dev/urandom of=binary.file bs=1024 count=100000
100000+0 records in
100000+0 records out
102400000 bytes (102 MB, 98 MiB) copied, 6.42155 s, 15.9 MB/s
mahesh@xenial:~$ mv binary.file Desktop/tmp/PAs/PA3/server/
mahesh@xenial:~$ tftp localhost 8080
tftp> mode octet
tftp> get binary.file
^C
tftp> █

"xenial" 15:41 03-Nov-17
```

8. Transfer binary and net ASCII file from client to server: Already included above

Code:

CC=g++

```
all: tftps
```

```
tftps: server.o tftp.o  
    $(CC) server.o tftp.o -o tftps
```

```
server.o: server.cpp  
    $(CC) -c server.cpp
```

```
tftp.o: tftp.cpp  
    $(CC) -c tftp.cpp
```

```
.PHONY: clean
```

```
clean:  
    rm -f tftps *.o
```

```
#include <stdio.h>  
#include <unistd.h>  
#include <errno.h>  
#include <string.h>  
#include <string>  
#include <netdb.h>  
#include <sys/wait.h>
```

```
#include "tftp.h"
```

```
using namespace std;
```

```
// Handles child termination  
void sigchld_handler(int signo);
```

```
int main(int argc, char const *argv[])  
{  
    int i;  
    int sd;  
    int err;  
    int opcode;  
    int bytes_rcv;  
    char buffer[TFTP_MAX_REQ_SIZE];  
    struct addrinfo addr_info, *server_info, *it;  
    struct sockaddr_storage client_addr;  
    struct sigaction sig;  
    socklen_t sockaddr_size;  
    string file, mode;
```

```

// Check command line arguments
if (argc < 2) {
    printf("Argument Error! Try executing:\n");
    printf("./tftps PORT\n");
    return 1;
}

memset(&addr_info, 0, sizeof(addr_info));
addr_info.ai_family = AF_UNSPEC;
addr_info.ai_socktype = SOCK_DGRAM;
addr_info.ai_flags = AI_PASSIVE;

// Register the signal handler for SIGCHLD event
// so that the resulting process is not a zombie while exiting
sig.sa_handler = sigchld_handler;
sigemptyset(&sig.sa_mask);
sig.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sig, NULL) == -1) {
    perror("sigaction()");
    return 1;
}

// Get IP, Port
if ((err = getaddrinfo(NULL, argv[1], &addr_info, &server_info)) != 0) {
    fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
    return 1;
}

// Iterate through output of getaddrinfo() and find a port to bind to
for (it = server_info; it != NULL; it = it->ai_next) {
    sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
    if (sd == -1) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("socket()");
        continue;
    }
    if (bind(sd, it->ai_addr, it->ai_addrlen) == -1) {
        close(sd);
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("bind()");
        // If bind was unsuccessful, try the next port
        // Note: In this case, there happens to be only one port and
        // so the list server_info only has one element
        continue;
    }
}

```

```

        }
        break;
    }

    // Check if server successfully binded to the given port
    if (it == NULL) {
        fprintf(stderr, "Server failed to bind to given port.\n");
        return 1;
    }
    // Free Memory
    freeaddrinfo(server_info);

    // Listen for data !!
    printf("Waiting for clients ...\n");
    sockaddr_size = sizeof(client_addr);

    while (true) {
        bytes_rcv = recvfrom(sd, buffer, TFTP_MAX_REQ_SIZE - 1, 0, (struct sockaddr
*)&client_addr, &sockaddr_size);
        if (bytes_rcv == -1) {
            printf("%s:%d ", __FILE__, __LINE__);
            fflush(stdout);
            perror("recvfrom()");
            return 1;
        }
        buffer[bytes_rcv] = '\0';

        // Get OPCODE
        memcpy(&opcode, buffer, 2);
        opcode = htons(opcode);

        // Get Filename and Mode
        file.clear();
        mode.clear();
        for (i = 2; buffer[i] != 0x00; i++) {
            file += buffer[i];
        }
        for (i++; buffer[i] != 0x00; i++) {
            mode += buffer[i];
        }

        // Create a child process to handle the transfer
        if (!fork()) {
            close(sd);

```

```

// Get an Ephemeral Port ("0")
if ((err = getaddrinfo(NULL, "0", &addr_info, &server_info)) != 0) {
    fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
    return 1;
}
// Iterate through output of getaddrinfo() and find a port to bind to
for (it = server_info; it != NULL; it = it->ai_next) {
    sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
    if (sd == -1) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("socket()");
        continue;
    }
    if (bind(sd, it->ai_addr, it->ai_addrlen) == -1) {
        close(sd);
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("bind()");
        // If bind was unsuccessful, try the next port
        // Note: In this case, there happens to be only one port and
        // so the list server_info only has one element
        continue;
    }
    break;
}
// Check if server successfully binded to the given port
if (it == NULL) {
    fprintf(stderr, "Server failed to bind to given port.\n");
    return 1;
}
// Free Memory
freeaddrinfo(server_info);

// If Opcode == RRQ
if (opcode == 0x01) {
    printf("Operation: Read %s in %s mode ... ", file.c_str(), mode.c_str());
    fflush(stdout);
    if (mode == "netascii") {
        err = tftp_send(sd, file, TFTP_NETASCII, (struct sockaddr
*&client_addr);

    } else {
        err = tftp_send(sd, file, TFTP_OCTET, (struct sockaddr
*&client_addr);

    }
    if (err != 0) {

```



```

        if (err == TFTP SOCK_ERROR) {
            tftp_err(sd, 0, "Socket Error!", (struct sockaddr
*)&client_addr);
        }
        else if (err = TFTP_FILE_ERROR) {
            tftp_err(sd, 1, "File Error!", (struct sockaddr
*)&client_addr);
        }
        else if (err = TFTP_TIMEOUT) {
            tftp_err(sd, 0, "Timeout Error!", (struct sockaddr
*)&client_addr);
        }
    }
}
// If Opcode == WRQ
else if (opcode == 0x02) {
    printf("Operation: Write %s in %s mode ... ", file.c_str(),
mode.c_str());
    fflush(stdout);
    if (mode == "netascii") {
        err = tftp_recv(sd, file, TFTP_NETASCII, (struct sockaddr
*)&client_addr);
    } else {
        err = tftp_recv(sd, file, TFTP_OCTET, (struct sockaddr
*)&client_addr);
    }
    if (err != 0) {
        if (err == TFTP SOCK_ERROR) {
            tftp_err(sd, 0, "Socket Error!", (struct sockaddr
*)&client_addr);
        }
        else if (err = TFTP_FILE_ERROR) {
            tftp_err(sd, 2, "File Error!", (struct sockaddr
*)&client_addr);
        }
        else if (err = TFTP_TIMEOUT) {
            tftp_err(sd, 0, "Timeout Error!", (struct sockaddr
*)&client_addr);
        }
    }
}
}
else {
    printf("Unknown Opcode!\n");
    close(sd);
    return 0;
}
}

```

```

        printf("Done\n");
        close(sd);
        return 0;
    }
}

close(sd);
return 0;
}

// SIGCHLD handler
void sigchld_handler(int signo)
{
    int saved_errno = errno;
    while(waitpid(-1, NULL, WNOHANG) > 0);
    errno = saved_errno;
}

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <string>
#include <netdb.h>
#include <signal.h>

#include "tftp.h"

using namespace std;

// Opcodes
#define OPCODE_RRQ 0x01
#define OPCODE_WRQ 0x02
#define OPCODE_DAT 0x03
#define OPCODE_ACK 0x04
#define OPCODE_ERR 0x05

#define ACK_SIZE 4

// Number of seconds to wait before timeout
#define TIMEOUT 3
// Max Retries
#define RETRIES 10

// Global Variable used by SIGALRM handler
bool timeout = false;

```

```

// Common structure used for data and err
struct data_t {
    unsigned int opcode:16;
    unsigned int block:16;
    unsigned char data[TFTP_DATA_SIZE];
};

struct ack_t {
    unsigned int opcode:16;
    unsigned int block:16;
};

// Alarm (Timeout) handler
void handle_alarm(int signo, siginfo_t *info, void *ptr)
{
    timeout = true;
}

// Send Error Packet
void tftp_err(int sd, int code, std::string msg, struct sockaddr *client)
{
    int i;
    int bytes;
    int payload_size;
    struct data_t err;
    struct sockaddr_storage client_addr;
    socklen_t size;

    size = sizeof(client_addr);
    memset(&err, 0, sizeof(err));
    err.opcode = htons(OPCODE_ERR);
    err.block = htons(code);    // block is same as error code

    // Truncate at 512 bytes
    if (msg.length() > TFTP_DATA_SIZE) {
        payload_size = TFTP_DATA_SIZE + 4;
    } else {
        payload_size = msg.length() + 4;
    }

    for (i = 0; i < (payload_size - 4); i++) {
        err.data[i] = msg[i];
    }

    // Send ERR Packet
    bytes == sendto(sd, &err, payload_size, 0, client, sizeof(struct sockaddr_storage));
}

```

```

        if (bytes == -1) {
            printf("%s:%d ", __FILE__, __LINE__);
            fflush(stdout);
            perror("sendto()");
        }
    }

int tftp_rcv(int sd, string file, int mode, struct sockaddr *client)
{
    int i;
    int id;
    int bytes;
    int retries;
    char buffer[TFTP_DATA_SIZE + 4];
    struct ack_t ack;
    struct data_t data;
    struct sockaddr_storage client_addr;
    FILE *fp;
    socklen_t size;

    id = 0;
    retries = 0;
    ack.opcode = htons(OPCODE_ACK);
    size = sizeof(client_addr);

    if (mode == TFTP_NETASCII) {
        fp = fopen(file.c_str(), "w");
    } else {
        fp = fopen(file.c_str(), "wb");
    }

    // If file can't be opened
    if (fp == NULL) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("fopen()");
        return TFTP_FILE_ERROR;
    }

    // Send ACK0
    ack.block = htons(id++);
    bytes = sendto(sd, &ack, sizeof(ack), 0, client, sizeof(struct sockaddr_storage));
    if (bytes == -1) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("sendto()");
    }
}

```

```

        return TFTP SOCK_ERROR;
    }

    while (1) {
        alarm(TIMEOUT);
        bytes = recvfrom(sd, buffer, TFTP_DATA_SIZE + 4, 0, (struct sockaddr
*)&client_addr, &size);
        if (bytes == -1) {
            if (timeout) {
                timeout = false;
                printf("Timeout!\n");
                fflush(stdout);
                retries++;
                if (retries == RETRIES) {
                    return TFTP_TIMEOUT;
                }
                continue;
            } else {
                printf("%s:%d ", __FILE__, __LINE__);
                fflush(stdout);
                perror("recvfrom()");
                return TFTP SOCK_ERROR;
            }
        }
        alarm(0);
        memcpy(&data, buffer, bytes);

        if (data.opcode == htons(OPCODE_ERR)) {
            printf("%s:%d", __FILE__, __LINE__);
            fflush(stdout);
            tftp_perror(ntohs(data.block));
            return TFTP SOCK_ERROR;
        }
        if (data.opcode != htons(OPCODE_DAT)) {
            printf("%s:%d Corrupt Data", __FILE__, __LINE__);
            fflush(stdout);
            continue;
        }
        // If block id is different, recv again
        if (data.block != htons(id)) {
            continue;
        }

        for (i = 0; i < (bytes - 4); i++) {
            fputc(data.data[i], fp);
        }
    }

```

```

        // Send ACK
        ack.block = htons(id++);
        bytes = sendto(sd, &ack, sizeof(ack), 0, client, sizeof(struct sockaddr_storage));
        if (bytes == -1) {
            printf("%s:%d ", __FILE__, __LINE__);
            fflush(stdout);
            perror("sendto()");
            return TFTP SOCK_ERROR;
        }
        retries = 0;

        if (i != 512) {
            break;
        }
    }

    fclose(fp);
    return 0;
}

```

```

int tftp_send(int sd, string file, int mode, struct sockaddr *client)
{

```

```

    int i;
    int id;
    int bytes;
    int payload_size;
    char ch;
    char buffer[ACK_SIZE];
    struct data_t data;
    struct ack_t ack;
    struct sockaddr_storage client_addr;
    struct sigaction act;
    FILE *fp;
    socklen_t size;

```

```

    data.opcode = htons(OPCODE_DAT);
    size = sizeof(client_addr);
    id = 1;
    memset(&act, 0, sizeof(act));
    act.sa_sigaction = handle_alarm;
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGALRM, &act, NULL);

```

```

    // Open the file on the server

```

```

if (mode == TFTP_NETASCII) {
    fp = fopen(file.c_str(), "r");
} else {
    fp = fopen(file.c_str(), "rb");
}

// Check if file can't be opened
if (fp == NULL) {
    printf("%s:%d ", __FILE__, __LINE__);
    fflush(stdout);
    perror("fopen()");
    return TFTP_FILE_ERROR;
}

while(1) {
    data.block = htons(id++);
    // Fill one block of data
    if (mode == TFTP_NETASCII) {
        for (i = 0; i < TFTP_DATA_SIZE; i++) {
            ch = fgetc(fp);
            if (ch == EOF) {
                break;
            }
            data.data[i] = ch;
        }
        payload_size = i + 4;
    } else {
        bytes = fread(&data.data, sizeof(unsigned char), TFTP_DATA_SIZE, fp);
        if (bytes != TFTP_DATA_SIZE) {
            ch = EOF;
        }
        payload_size = bytes + 4;
    }

    // Try to send until we receive a proper ACK
    // or until we exceed the maximum retries
    i = RETRIES;
    while(i > 0) {
        bytes = sendto(sd, &data, payload_size, 0, client, sizeof(struct
sockaddr_storage));
        if (bytes == -1) {
            printf("%s:%d ", __FILE__, __LINE__);
            fflush(stdout);
            perror("sendto()");
            return TFTP SOCK_ERROR;
        }
    }
}

```

```

    alarm(TIMEOUT);
    bytes = recvfrom(sd, buffer, ACK_SIZE, 0, (struct sockaddr *)&client_addr,
&size);

    if (bytes == -1) {
        if (timeout) {
            printf("Timeout!\n");
            fflush(stdout);
            timeout = false;
            i--;
            continue;
        } else {
            printf("%s:%d ", __FILE__, __LINE__);
            fflush(stdout);
            perror("recvfrom()");
            return TFTP SOCK_ERROR;
        }
    }
    alarm(0);

    memcpy(&ack, buffer, ACK_SIZE);
    if (ack.opcode == htons(OPCODE_ERR)) {
        printf("%s:%d", __FILE__, __LINE__);
        fflush(stdout);
        // ACK BlockID is same as ErrorCode for ERR
        tftp_perror(ntohs(ack.block));
        return TFTP SOCK_ERROR;
    }
    if (ack.opcode != htons(OPCODE_ACK)) {
        printf("%s:%d Corrupt Data", __FILE__, __LINE__);
        fflush(stdout);
        continue;
    }

    if (ack.block == htons(id-1)) {
        break;
    }
    i--;
}
if (i == 0) {
    printf("%s:%d Timeout Error\n", __FILE__, __LINE__);
    fflush(stdout);
    return TFTP_TIMEOUT;
}

// If reached end of file, terminate while(1) loop

```



```

        if (ch == EOF) {
            break;
        }
    }

    fclose(fp);
    return 0;
}

void tftp_perror(int err)
{
    switch(err) {
        case 1 : printf("File not found.\n"); break;
        case 2 : printf("Access violation.\n"); break;
        case 3 : printf("Disk full or allocation exceeded.\n"); break;
        case 4 : printf("Illegal TFTP operation.\n"); break;
        case 5 : printf("Unknown transfer ID.\n"); break;
        case 6 : printf("File already exists.\n"); break;
        case 7 : printf("No such user.\n"); break;
        default: printf("Error Code not defined, see error message.\n"); break;
    }
    fflush(stdout);
}

// Modes of Transfer
#define TFTP_OCTET 0
#define TFTP_NETASCII 1

// Max Limits
#define TFTP_MAX_REQ_SIZE 100
#define TFTP_DATA_SIZE 512

// Error Codes
#define TFTP SOCK_ERROR -1
#define TFTP FILE_ERROR -2
#define TFTP TIMEOUT -3

int tftp_send(int, std::string, int, struct sockaddr *);
int tftp_recv(int, std::string, int, struct sockaddr *);
void tftp_err(int, int, std::string, struct sockaddr *);
void tftp_perror(int);

```