**Ex. No. 2B** **TCP File Server** **Karthik D**

**25 July 2021** **UCS1511 - Networks Lab** **195001047**

## Aim

To develop a TCP socket program to establish client server communication. The server must be able to transfer a client requested file to the client using TCP a connection

## Question

Develop a C program to implement a TCP driven file server. The client must be able to connect to a specific server, supplying the address of the server. The client should then be able to send a text message to the server indicating the path to the required file. The server should then find the requested file and transfer it to the client.

## Algorithms

### (a) Server-side

**Step 1:** Create a network socket with parameters suitable for an end-point of TCP based communication

**Step 2:** Bind the socket to INADDR_ANY which is defined as a *zero* address, allowing the socket to be reachable by all active interfaces on the device. Set the port to a preset value, known to the targeted clients as well

**Step 3:** Set the socket status to passive i.e initiate listening on the socket to allow it to accept incoming connection requests

**Step 4:** Wait for a connection request from a client and accept the first such request. Save the file descriptor of the connection-socket. This will be used to communicate with the client

**Step 5:** Prepare a memory buffer to read and store messages from the connection.

**Step 6:** Start an infinite loop to perform the following operations,

- **i:** Use the *read()* system call to block and read a message sent from the client by reading into the client connection socket, and store the message in the buffer
- **ii:** Check if the message received is the connection termination string. If so, send back a termination acknowledgment to the client and *exit the loop*. If not, continue to step-(iii)

**iii:** Search the server for the requested file. If the file is not found, send a suitable message string to client and go back to *step-(i)*

**iv:** Maintain a character memory buffer of suitable size as per the server configuration

**v:** Send the next chunk of file data containing as many bytes as the character buffer can hold, using the *write()* system call to write the buffer contents to the client connection socket

**vi:** Block and read the file chunk acknowledgement from the client using the *read()* system call

**vii:** Repeat from *step-(v)* until the entire file has been transferred. When done, go to *step-(viii)*

**viii:** Send a message string indicating the end of file transfer to the client. Block and wait for the client to request the next file

( Repeat till client requests connection termination )

**Step 7:** Close the created socket using the *close()* system call and terminate the process

(b) **<u>Client-side</u>**

**Step 1:** Create a network socket with parameters suitable for an end-point of TCP based communication

**Step 2:** Accept the target server address as input from the user

**Step 3:** Using the accepted address and a preset port number agreed upon between the server and client, send a connection request to the server using the *connect()* system call

**Step 4:** Prepare a memory buffer to read and store messages from the connection.

**Step 5:** Start an infinite loop to perform the following operations,

**i:** Accept the path and filename of the file to be downloaded from the server

**ii:** Open a local file with the user entered path and filename in write mode using the *open()* system call

**iii:** Use the *write()* system call to send the accepted file path to the server by writing to own socket stream

**iv:** Use the *read()* system call to block and read a message sent from the server by reading into the own socket, and store the message in the buffer

**v:** If the received message is a connection termination acknowledgement from the server, terminate the loop. Otherwise, continue to *step-(vi)*

**vi:** If the received message is a message indicating that the file was not found, display a suitable message to the user and repeat from *step-(i)*. Otherwise, continue to *step-(vii)*

**vii:** If the received message is a message indicating that the end of file has been reached, go to *step-(x)*

**viii:** Write the data from the memory buffer into the local file. Send back a file chunk acknowledgement to the server using the *write()* system call and writing to own socket

**ix:** Repeat from *step-(iv)* until the entire file has been transferred and the server acknowledges the same

**x:** Close the local write file using the *close()* system call

( Repeat till server acknowledges connection termination )

**Step 6:** Close the created socket using the *close()* system call and terminate the process

## C Program Code

1. tcp_socket.h - TCP connection helper functions

```
#ifndef tcp_socket
#define tcp_socket

#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>

#define SERVER_PORT 8080
#define BACKLOG_LIMIT 5
```

```c
#define LOCALHOST_IP "127.0.0.1"
#define ADDRESS_FAMILY AF_INET
#define ADDRESS_BUFFER_SIZE 30
#define MSG_BUFFER_SIZE 20
#define IP_STRING_LEN 24
#define TERMINATION_INIT_STRING "ENDSESSION"
#define TERMINATION_ACK_STRING "ENDSESSION_ACK"
#define FILENOTFOUND_STRING "FILENOTFOUND"
#define TRANSFERFAIL_STRING "TRANFERFAILED"
#define MSG_DELIMITER ';'

/*
Use BLOCKING sockets (default configuration)
Only one client-connection
And server only echoes messages
No need to initiate messages on the server!
(i.e) Synchronous send/receive
*/

int make_socket(){
   // AF_INTER specifies IPv4
   // SOCK_STREAM specifies two-way byte-stream
   // 0 selects default protocol
   int sock_fd = socket(ADDRESS_FAMILY, SOCK_STREAM, 0);
   if (sock_fd == -1){
       return -2;    // Could not create socket
   }
   return sock_fd;
}

short check_termination_init(char *msg){
   return (strcmp(msg, TERMINATION_INIT_STRING)==0);
}

short check_termination_ack(char *msg){
   return (strcmp(msg, TERMINATION_ACK_STRING)==0);
}

short bind_server_socket(int sock_fd){
   struct sockaddr_in bind_address;
   // Set family to IPv4
   bind_address.sin_family = ADDRESS_FAMILY;
   // Set port in network byte-order to a non-privileged port (>1023)
   bind_address.sin_port = htons(SERVER_PORT);
   // Set address to 0.0.0.0 to connect to bind to all local interfaces
   bind_address.sin_addr.s_addr = htonl(INADDR_ANY);
   // Convert to generic socket address format and bind
   if (!bind(sock_fd, (struct sockaddr *)&bind_address,
sizeof(bind_address))){
       return 0;    // Success
   }
   else{
       return -3;   // Could not bind server-socket
   }
}

short connect_server(int sock_fd, char *server_ip){
   struct sockaddr_in bind_address;
   bzero((char*)&bind_address, sizeof(bind_address));
   // Set family to IPv4
   bind_address.sin_family = ADDRESS_FAMILY;
   // Set port in network byte-order to a non-privileged port (>1023)
   bind_address.sin_port = htons(SERVER_PORT);
   // Set address to server IP or 127.0.0.1 to loopback to same host
   if (server_ip == NULL){
       bind_address.sin_addr.s_addr = inet_addr(LOCALHOST_IP);
   }
   else{
       bind_address.sin_addr.s_addr = inet_addr(server_ip);
   }
   // Convert to generic socket address format and bind
   if (!connect(sock_fd, (struct sockaddr*)&bind_address,
sizeof(bind_address))){
       return 0;    // Success
   }
   else{
       return -5;   // Could not connect to the local server
   }
}
```

```
short initiate_listen(int sock_fd){
   if (!listen(sock_fd, BACKLOG_LIMIT)){
       return 0;     // Success
   }
   else{
       return -4;    // Could not initiate listening
   }
}

// BLOCKING FUNCTION
int accept_client(int sock_fd, struct sockaddr_in *client_addr, int
*client_addr_len){
   // Address of client is registered
   int client_sock_fd = accept(sock_fd, (struct sockaddr*)client_addr,
client_addr_len);
   if (client_sock_fd == -1){
       return -5;    // Did not connect to a client
   }
   else if (*client_addr_len > sizeof(struct sockaddr_in)){
       *client_addr_len = -1;     // Non-Fatal Warning: Client address was
truncated to fit in buffer
   }
   return client_sock_fd;
}

void destroy_socket(int sock_fd){
   close(sock_fd);
}

#endif
```

2. file_io.h - File transfer helper functions

```
#ifndef file_io
#define file_io

#include<fcntl.h>
#include<sys/stat.h>
```

```c
#include "tcp_socket.h"

#define FILE_CHUNK_ACK "FILECHUNKACK"
#define EOF_STRING "ENDFILEHERE"
#define PATH_SIZE 100

mode_t get_default_fileperm(){
   mode_t curr_umask = umask(022);
   umask(curr_umask);
   return (mode_t)0666-curr_umask;
}

short check_chunk_ack(char *msg){
   return (strcmp(msg, FILE_CHUNK_ACK)==0);
}

short check_eof(char *msg){
   return (strcmp(msg, EOF_STRING)==0);
}

short check_transfer_fail(char *msg){
   return (strcmp(msg, TRANSFERFAIL_STRING)==0);
}

short check_filenotfound(char *msg){
   return (strcmp(msg, FILENOTFOUND_STRING)==0);
}

short send_file(char *filepath, int sock_fd){
   int fd_in = open(filepath, O_RDONLY);
   if (fd_in<0){
       return -6;   // Could not open read-file
   }
   char *data = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
   int read_size = read(fd_in, data, MSG_BUFFER_SIZE);
   int write_size = 0;
   do{
       write_size = write(sock_fd, data, read_size);
       // Wait for acknowledgement
```

```c
        read_size = read(sock_fd, data, MSG_BUFFER_SIZE);
        if (!check_chunk_ack(data)){
            return -7;    // Corrupt occurred while sending file
        }
        read_size = read(fd_in, data, MSG_BUFFER_SIZE);
    }while(read_size!=0);
    write_size = write(sock_fd, EOF_STRING, sizeof(EOF_STRING));
    close(fd_in);
    return 0;
}


short receive_file(char *filepath, int sock_fd){
    char *working_dir = (char*)malloc(sizeof(char)*PATH_SIZE);
    getcwd(working_dir, PATH_SIZE);
    chdir("downloads");
    int fd_out = open(filepath, O_CREAT|O_WRONLY, get_default_fileperm());
    chdir(working_dir);
    if(fd_out<0){
        return -8;   // Could not open write-file
    }
    char *buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    // Block until filename is read
    int read_size = 0;
    do{
        read_size = recv(sock_fd, buffer, MSG_BUFFER_SIZE, MSG_PEEK);
    }while(strcmp(buffer, filepath)==0);
    // Start file-transfer
    read_size = read(sock_fd, buffer, MSG_BUFFER_SIZE);
    if (check_transfer_fail(buffer)){
        return -9;   // Transfer failed abruptly
    }
    else if(check_filenotfound(buffer)){
        return -10;  // File was not found
    }
    int write_size = 0;
    do{
        if (check_eof(buffer)) {
            break;
        }
```

```
       write_size = write(fd_out, buffer, read_size);
       // Send acknowledgement
       write_size = write(sock_fd, FILE_CHUNK_ACK,
sizeof(FILE_CHUNK_ACK));
       // Block until acknowledgement is read
       do{
           read_size = recv(sock_fd, buffer, MSG_BUFFER_SIZE, MSG_PEEK);
       }while(check_chunk_ack(buffer));
       // Read next chunk of data
       read_size = read(sock_fd, buffer, MSG_BUFFER_SIZE);
   }while(1==1);
   close(fd_out);
   return 0;
}

#endif
```

3. <u>server.c - Server-side program</u>

```
#include<stdio.h>
#include<stdlib.h>

#ifndef tcp_socket
   #include "tcp_socket.h"
#endif

#ifndef file_io
   #include "file_io.h"
#endif

void main(){

   int self_socket = make_socket();
   if(self_socket<0){
       printf("\nCould not create socket. Retry!\n");
       return;
   }

   if (bind_server_socket(self_socket)<0){
       printf("\nCould not bind server socket. Retry!\n");
```

```
        destroy_socket(self_socket);
        return;
    }

    if (initiate_listen(self_socket)<0){
        printf("\nCould not listen on server socket. Retry!\n");
        destroy_socket(self_socket);
        return;
    }
    else{
        printf("\nServer listening for connections from all local
interfaces...\n");
    }

    struct sockaddr_in *client_addr = malloc(sizeof(struct sockaddr_in));
    int client_addr_len = sizeof(struct sockaddr_in);
    // BLOCKING routine to accept a client
    int client_socket = accept_client(self_socket, client_addr,
&client_addr_len);
    if (client_socket<0){
        printf("\nError when connecting to client. Retry!\n");
        destroy_socket(self_socket);
        return;
    }
    else if(client_addr_len == -1){
        printf("Client connected.\nCould not read address\n");
    }
    else{
        char *client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
        // Alternatively, use inet_ntoa
        inet_ntop(ADDRESS_FAMILY, (void*)&client_addr->sin_addr,
client_addr_ip_str, ADDRESS_BUFFER_SIZE);
        int client_addr_port = (int)ntohs(client_addr->sin_port);
        if (client_addr_ip_str == NULL) {
            printf("Client connected.\nCould not read address\n");
        }
        else{
            printf("Connected to Client (%s:%d)\n", client_addr_ip_str,
client_addr_port);
```

```
        }
    }

    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    int msg_size = 0;
    int response;
    do{
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        printf("\nAwaiting file request...\n");
        msg_size = read(client_socket, msg_buffer, MSG_BUFFER_SIZE);
        if (msg_size==0){
            printf("\nClient shut-down abruptly!\n");
            destroy_socket(client_socket);
            break;
        }
        if (check_termination_init(msg_buffer)){
            printf("\nClient terminated connection\n");
            bzero(msg_buffer, MSG_BUFFER_SIZE);
            msg_size = write(client_socket, TERMINATION_ACK_STRING,
msg_size);
            destroy_socket(client_socket);
            break;
        }
        printf("\nCLIENT requested file: %s", msg_buffer);
        fflush(stdout);
        response = send_file(msg_buffer, client_socket);
        if(response==-6){
            printf("\nRequested file NOT FOUND\n");
            msg_size = write(client_socket, FILENOTFOUND_STRING,
sizeof(FILENOTFOUND_STRING));
        }
        else if(response==-7){
            printf("\nFile tranfer failed unexepectedly\n");
            msg_size = write(client_socket, TRANSFERFAIL_STRING,
sizeof(TRANSFERFAIL_STRING));
        }
        else{
            printf("\nFile transfered to client\n");
        }
    }while(1==1);
```

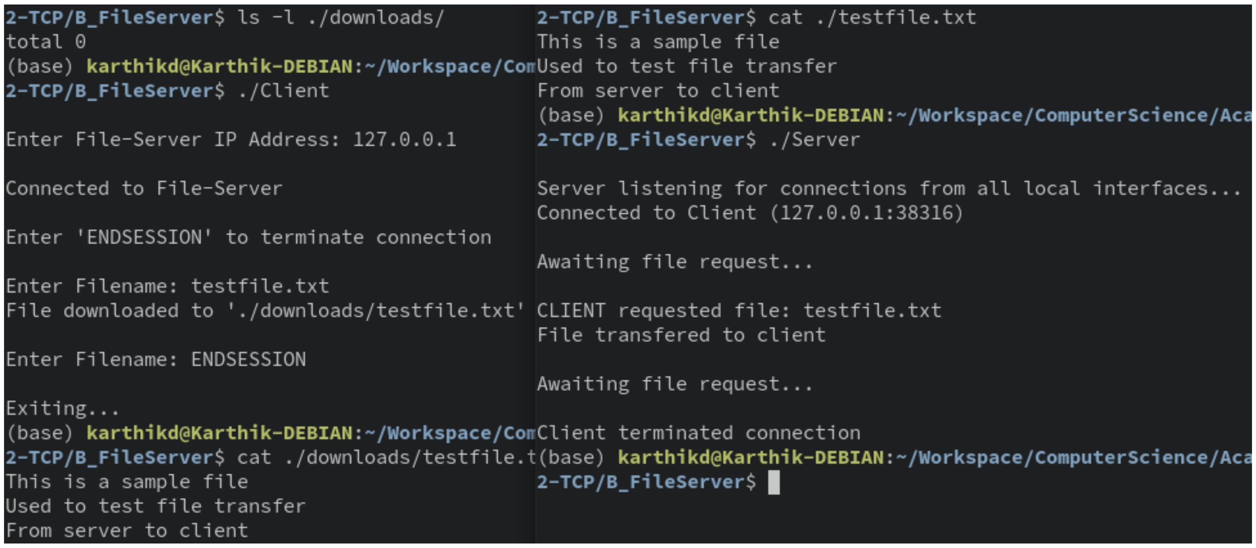
```
   destroy_socket(self_socket);
   return;
}
```

4. client.c - Client-side program

```
#include<stdio.h>
#include<stdlib.h>

#ifndef tcp_socket
   #include "tcp_socket.h"
#endif

#ifndef file_io
   #include "file_io.h"
#endif

void main(){

   int self_socket = make_socket();
   if(self_socket<0){
       printf("\nCould not create socket. Retry!\n");
       return;
   }

   char *server_ip = (char*)malloc(sizeof(char)*IP_STRING_LEN);
   printf("\nEnter File-Server IP Address: ");
   scanf(" %s", server_ip);
   if (connect_server(self_socket, server_ip) < 0){
       printf("\nCould not connect to File-Server.\nMake sure the server
is running!\n");
       destroy_socket(self_socket);
       return;
   }
   else{
       printf("\nConnected to File-Server");
   }

   char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
```

```
    int msg_size = 0;
    int response;
    printf("\n\nEnter 'ENDSESSION' to terminate connection\n");
    do {
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        printf("\nEnter Filename: ");
        scanf(" %[^\n]s", msg_buffer);
        msg_size = write(self_socket, msg_buffer, MSG_BUFFER_SIZE);
        if (check_termination_init(msg_buffer)){
            msg_size = read(self_socket, msg_buffer, MSG_BUFFER_SIZE);
            // Check if server acknowledged ENDSESSION
            if (check_termination_ack(msg_buffer)){
                printf("\nExiting...\n");
                break;
            }
            else{
                printf("\nServer did not acknowledge termination.
Retry!\n");
                continue;
            }
        }
        response = receive_file(msg_buffer, self_socket);
        if(response==-8){
            printf("\nCould not create local file\n");
        }
        else if(response==-9){
            printf("\nFile transfer failed unexpectedly\n");
        }
        else if(response==-10){
            printf("File '%s' was not found on server\n", msg_buffer);
        }
        else{
            printf("File downloaded to './downloads/%s'\n", msg_buffer);
        }
    }while(1==1);
}
```

## Sample Outputs



```
2-TCP/B_FileServer$ ls -l ./downloads/
total 0
(base) karthikd@Karthik-DEBIAN:~/Workspace/Com
2-TCP/B_FileServer$ ./Client

Enter File-Server IP Address: 127.0.0.1

Connected to File-Server

Enter 'ENDSESSION' to terminate connection

Enter Filename: testfile.txt
File downloaded to './downloads/testfile.txt'

Enter Filename: ENDSESSION

Exiting...
(base) karthikd@Karthik-DEBIAN:~/Workspace/Com
2-TCP/B_FileServer$ cat ./downloads/testfile.t
This is a sample file
Used to test file transfer
From server to client
```

```
2-TCP/B_FileServer$ cat ./testfile.txt
This is a sample file
Used to test file transfer
From server to client
(base) karthikd@Karthik-DEBIAN:~/Workspace/ComputerScience/Aca
2-TCP/B_FileServer$ ./Server

Server listening for connections from all local interfaces...
Connected to Client (127.0.0.1:38316)

Awaiting file request...

CLIENT requested file: testfile.txt
File transfered to client

Awaiting file request...

Client terminated connection
(base) karthikd@Karthik-DEBIAN:~/Workspace/ComputerScience/Aca
2-TCP/B_FileServer$ 
```

## Result

Implemented a socket program in C language to establish client server communication. A file server is developed, wherein the client sends the required file path to the server and the server in turn, transfers the file to the client. Through this implementation, the following aspects were understood:

1. Basic functioning of the TCP protocol
2. Synchronizing multiple chunks of data transfer between server and client
3. Implementation details of socket programming using C language