Ex. No. 2A	TCP Echo Server	Karthik D
25 July 2021	UCS1511 - Networks Lab	195001047

Aim

To develop a socket program to establish client server communication. The client sends data to the server. The server in turn sends the message back to the client allowing multiple lines of text per message

Question

Develop a C program to implement a TCP driven echo 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 multiline text messages to the server. The server should then echo back the same message 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: Use the *write()* system call to send back the contents of the buffer to the client by writing to the client connection socket

(Repeat till client requests connection termination)

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

(b) **Client-side**

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 a multiline message string from the user. Use a suitable delimiter to read the user input

ii: Use the *write()* system call to send the accepted message to the server by writing to own socket stream

iii: 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

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

v: Display the received message to the user

(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

```c
#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
#define LOCALHOST_IP "127.0.0.1"
#define ADDRESS_FAMILY AF_INET
#define ADDRESS_BUFFER_SIZE 30
#define MSG_BUFFER_SIZE 100
#define IP_STRING_LEN 24
#define TERMINATION_INIT_STRING "ENDSESSION"
#define TERMINATION_ACK_STRING "ENDSESSION_ACK"
#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{
       printf("%d", errno);
       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);
}
```

2. server.c - Server-side program

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

#include "tcp_socket.h"

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;
    do{
        // BLOCKING routine to wait for a message
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        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;
        }
        // Echo back
        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 pinged: %s", msg_buffer);
        msg_size = write(client_socket, msg_buffer, msg_size);
        printf("\n(Message echoed back)\n");
    }while(1==1);

    destroy_socket(self_socket);
    return;
}
```

3. client.c - Client-side program

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

#include "tcp_socket.h"

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 ECHO-Server.\nMake sure the
server is running!\n");
        destroy_socket(self_socket);
        return;
    }
    else{
        printf("\nConnected to ECHO-Server");
    }
```

```
    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    int msg_size = 0;
    printf("\n\nDelimit Ping Messages with ';'\nEnter 'ENDSESSION;'
to terminate connection\n");
    do {
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        printf("\nEnter Ping Message: ");
        scanf(" %[^;]s", msg_buffer);
        // Consume the last newline character from read-buffer
        getchar();
        msg_size = write(self_socket, msg_buffer, MSG_BUFFER_SIZE);
        // Reading back
        bzero(msg_buffer, MSG_BUFFER_SIZE);
        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;
        }
        printf("SERVER echoed: %s\n", msg_buffer, MSG_BUFFER_SIZE);
    }while(1==1);
}
```

Sample Outputs

```
2-TCP/A_EchoServer$ ./Server

Server listening for connections from all local interfaces...
Connected to Client (127.0.0.1:38150)

CLIENT pinged: Hey server
This is a multiline message
With three lines
(Message echoed back)

CLIENT pinged: This is one line
(Message echoed back)

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

```
2-TCP/A_EchoServer$ ./Client

Enter File-Server IP Address: 127.0.0.1

Connected to ECHO-Server

Delimit Ping Messages with ';'
Enter 'ENDSESSION;' to terminate connection

Enter Ping Message: Hey server
This is a multiline message
With three lines;
SERVER echoed: Hey server
This is a multiline message
With three lines

Enter Ping Message: This is one line;
SERVER echoed: This is one line

Enter Ping Message: ENDSESSION;
```

Result

Implemented a socket program in C language to establish client server communication. An echo server is developed, wherein the client sends data to the server and the server in turn sends the message back to the client allowing multiple lines of text per message. Through this implementation, the following aspects were understood:

1. Basic functioning of the TCP protocol
2. Implementation details of socket programming using C language