Computer Systems Lab - II
Assignment 2

(Computer Network)

Poushali Chakraborty (2222208)
M-Tech Computer Science

Department of Computer Science
NIT Silchar
DATE SUBMITTED
[February 23, 2023]

Chapter 1

Problem 1

Modify TCPEchoServer.c to receive and send only a single byte at a time, sleeping one second between each byte. Verify that TCPEchoClient.c requires multiple receives to successfully receive the entire echo string, even though it sent the echo string with one send(). Donahoo and Calvert (2009)

Solution: To solve this problem we need TCPEchoServer and TCPEchoClient program. The programs are based on the simple TCP client and server model. The server runs an infinite loop and client takes user input string and sends it to the server. The server receives the string and print and send it to client one byte at a time, sleeping one second between sending each byte. Client require multiple recieve to get the full string.

1.1 TCPEchoServer

Basic flow of this progran will be like this,

1. Define the port number over which communication will take place, and a buffer of a certain size for reading and storing string data.
2. Create a socket address structure for the server, and set its members using memset() and htons()/htonl() functions.
3. Create a TCP socket using the socket() function. Specify the PF-INET address family and SOCK-STREAM type, and a protocol of 0.
4. Bind the socket descriptor to the server address structure using the bind() function.
5. Mark the socket as a listening socket using the listen() function.
6. Enter an infinite loop and wait for clients to connect using the accept() function.
7. Once a client has connected, retrieve its IP address and port number and print it to the console.
8. Pass the client socket descriptor to a function called HandleTCPClient().
9. HandleTCPClient() is going to read data from the client socket using the recv() function. If the number of bytes received is less than 0 than it's an error.
10. Send the received data back to the client one byte at a time, sleeping for one second between each byte.
11. Loop through steps 9-10 until the end of the stream is reached (i.e., the recv() function returns 0).
12. Close the client socket using the close() function.

Algorithm 1 TCP Echo Server

```
1: SERVER−PORT ← 9877
2: MAXPENDING ← 5
3: buffer ← character array of size 100
4: server−address ← sockaddr-in structure
5: Set all bytes of server−address to 0
6: Set server−address family to AF−INET
7: Set server−address port to SERVER−PORT in network byte order
8: Set server−address IP address to INADDR−ANY in network byte order
9: listen−sock ← create TCP socket with PF−INET family, SOCK−STREAM type, and default
   protocol
10: if listen−sock < 0 then
11:     print "could not create listen socket"
12:     exit with status 1
13: Bind listen−sock to server−address
14: if binding fails then
15:     DieWithSystemMessage("bind() failed")
16: Mark listen−sock as listening for incoming connections
17: if marking fails then
18:     DieWithSystemMessage("listen() failed")
19: while true do
20:     clntAddr ← sock-addr-in structure
21:     Set clntAddrLen to the size of clntAddr
22:     Wait for a client to connect and create clntSock
23:     if clntSock < 0 then
24:         DieWithSystemMessage("accept() failed")
25:     Convert clntAddr to a string clntName and print "Handling client clntName/clntPort"
26:     Call HandleTCPClient(clntSock)
```

Algorithm 2 HandleTCPClient(*clntSocket*)

```
1: buffer ← character array of size 100
2: temp ← character array of size 2
3: Set temp[1] to null character
4: Receive message from client into buffer
5: if receiving fails then
6:     DieWithSystemMessage("recv() failed")
7: i ← 0
8: while true do
9:     if buffer[i] ==' \0' then
10:         break
11:     Set temp[0] to buffer[i]
12:     Send temp to client
13:     Sleep for 1 second
14:     i ← i + 1
15: Close clntSocket
```

1.2 TCPEchoClient

Basic flow of this progran will be like this,

1. The program starts by including the necessary header files, defining a constant BUFSIZE, and declaring the main() function.
2. The program prompts the user to enter a string.
3. The program sets the server's IP address and port number in a sockaddr-in struct.

4. The program creates a socket using socket() function, specifying the IPv4 address family and the stream socket type. If the socket creation fails, the program terminates and show an error message.
5. The program establishes a connection to the server using the connect() function. If the connection fails, the program terminates with an error message.
6. The program determines the length of the input string and sends it to the server using the send() function. If the send fails or sends an unexpected number of bytes, the program terminates with an error message.
7. The program waits for the server to echo the same string back to the client using the recv() function. The program receives up to BUFSIZE-1 bytes from the server and keeps track of the total number of bytes received.
8. The program terminates the echoed string with a null terminator and prints it to the console.
9. The program closes the socket and exits with a return value of 0.

   Note that this program assumes that there is a server running on the same machine (localhost) and listening on port 9877. If the server is running on a different machine or listening on a different port, the server-address struct needs to be modified accordingly.

Algorithm 3 TCP Echo Client

```
1: Include the necessary header files.
2: Prompt the user to enter a string.
3: Set the server's IP address and port number in a sockaddr-in struct.
4: Create a socket using socket(PF-INET, SOCK-STREAM, 0).
5: if socket creation fails then
6:     Terminate the program with an error message.
7: Establish a connection to the server using connect().
8: if connection fails then
9:     Terminate the program with an error message.
10: Determine the length of the input string.
11: Send the string to the server using send().
12: if send fails or sends an unexpected number of bytes then
13:     Terminate the program with an error message.
14: Wait for the server to echo the same string back to the client using recv().
15: while totalBytesRcvd < echoStringLen do
16:     Receive up to BUFSIZE-1 bytes from the server using recv().
17:     if recv fails or connection is closed prematurely then
18:         Terminate the program with an error message.
19:     Keep tally of the total bytes received.
20:     Terminate the echoed string with a null terminator.
21:     Print the echoed string to the console.
22: Close the socket.
23: Exit the program with a return value of 0.
```

1.3 Result

We first run the TCPEchoServer program and than run TCPEchoClient program as shown in images below,

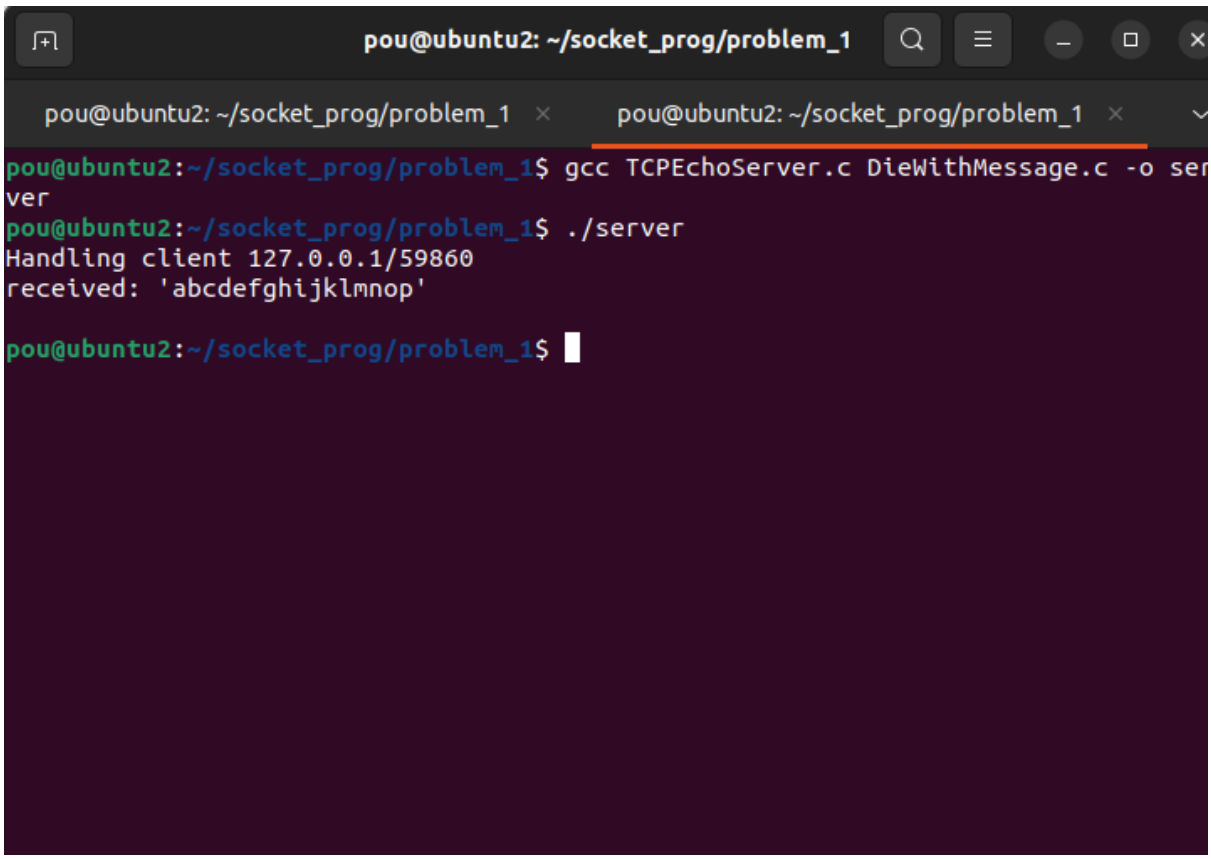

Figure 1.1: Output of TCPEchoServer

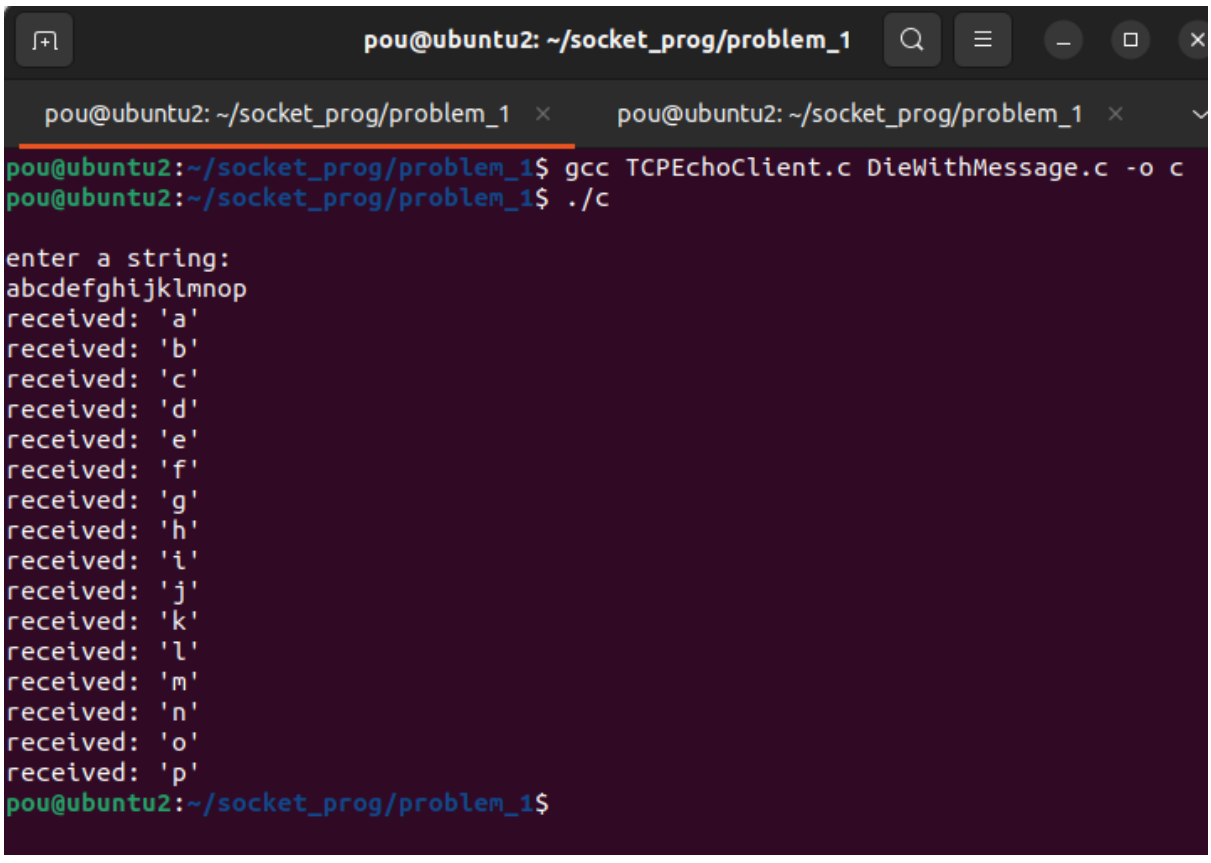

Figure 1.2: Output of TCPEchoClient

Chapter 2

Problem 2

Modify TCPEchoServer. c to read and write a single byte and then close the socket. What happens when the TCPEchoClient send a multibyte string to this server? (Note that the response could vary by operating system.) Donahoo and Calvert (2009)

Solution: To solve this problem we need TCPEchoServer and TCPEchoClient program. The programs are based on the simple TCP client and server model. The server runs an infinite loop and client takes user input string and sends it to the server. The server receives only one byte character and print and send it to client. After one receive it closes the client socket.

2.1 TCPEchoServer

Basic flow of this progran will be like this,

1. Define the port number over which communication will take place, and a buffer of a certain size for reading and storing string data.
2. Create a socket address structure for the server, and set its members using memset() and htons()/htonl() functions.
3. Create a TCP socket using the socket() function. Specify the PF-INET address family and SOCK-STREAM type, and a protocol of 0.
4. Bind the socket descriptor to the server address structure using the bind() function.
5. Mark the socket as a listening socket using the listen() function.
6. Enter an infinite loop and wait for clients to connect using the accept() function.
7. Once a client has connected, retrieve its IP address and port number and print it to the console.
8. Pass the client socket descriptor to a function called HandleTCPClient().
9. In HandleTCPClient(), read data from the client socket using the recv() function only one byte. If the number of bytes received is less than 0, an error occurred.
10. Send the received data back to the client.
11. Loop through steps 9-10 until the end of the stream is reached (i.e., the recv() function returns 0).
12. Close the client socket using the close() function.

Algorithm 4 TCP Echo Server

1: $SERVER-PORT \leftarrow 9877$
2: $MAXPENDING \leftarrow 5$
3: $buffer \leftarrow$ character array of size 100
4: $server-address \leftarrow$ sockaddr-in structure
5: Set all bytes of $server-address$ to 0
6: Set $server-address$ family to $AF-INET$
7: Set $server-address$ port to $SERVER-PORT$ in network byte order
8: Set $server-address$ IP address to $INADDR-ANY$ in network byte order
9: $listen-sock \leftarrow$ create TCP socket with $PF-INET$ family, $SOCK-STREAM$ type, and default protocol
10: **if** $listen-sock < 0$ **then**
11: print "could not create listen socket"
12: exit with status 1
13: Bind $listen-sock$ to $server-address$
14: **if** binding fails **then**
15: DieWithSystemMessage("bind() failed")
16: Mark $listen-sock$ as listening for incoming connections
17: **if** marking fails **then**
18: DieWithSystemMessage("listen() failed")
19: **while** true **do**
20: $clntAddr \leftarrow$ sock-addr-in structure
21: Set $clntAddrLen$ to the size of $clntAddr$
22: Wait for a client to connect and create $clntSock$
23: **if** $clntSock < 0$ **then**
24: DieWithSystemMessage("accept() failed")
25: Convert $clntAddr$ to a string $clntName$ and print "Handling client $clntName/clntPort$"
26: Call HandleTCPClient($clntSock$)

Algorithm 5 HandleTCPClient($clntSocket$)

1: $buffer \leftarrow$ character array of size 100
2: Receive message from client into $buffer$ only one byte
3: **if** receiving fails **then**
4: DieWithSystemMessage("recv() failed")
5: Send $buffer$ to client
6: Close $clntSocket$

2.2 TCPEchoClient

Same as the previous problem.

2.3 Result

We first run the TCPEchoServer program and than run TCPEchoClient program as shown in images below,

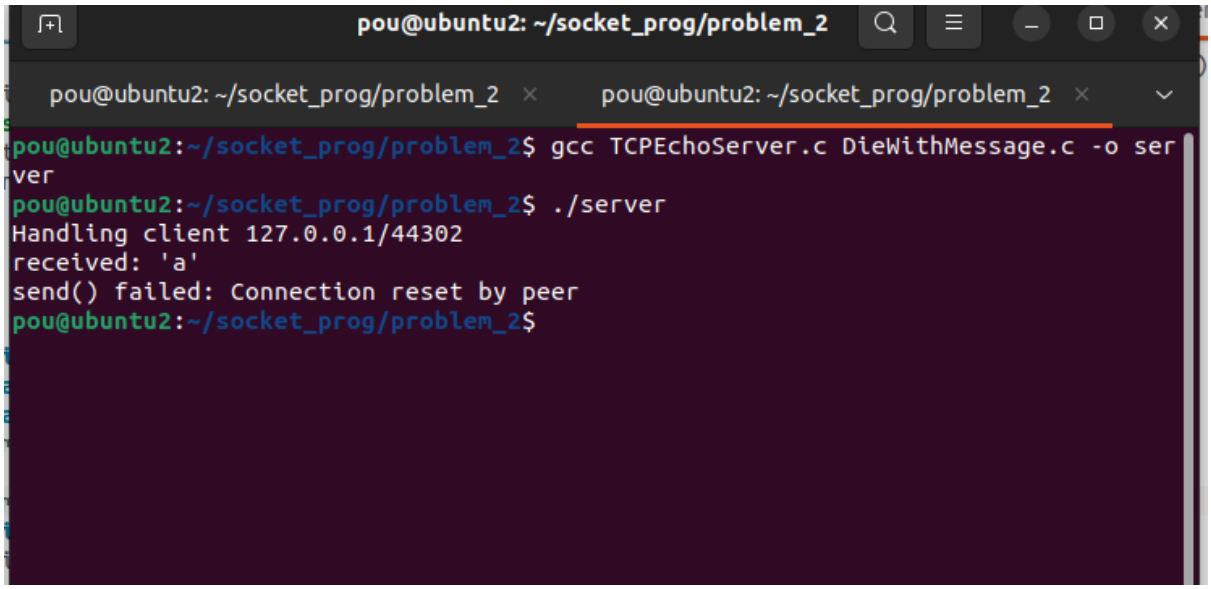

Figure 2.1: Output of TCPEchoServer

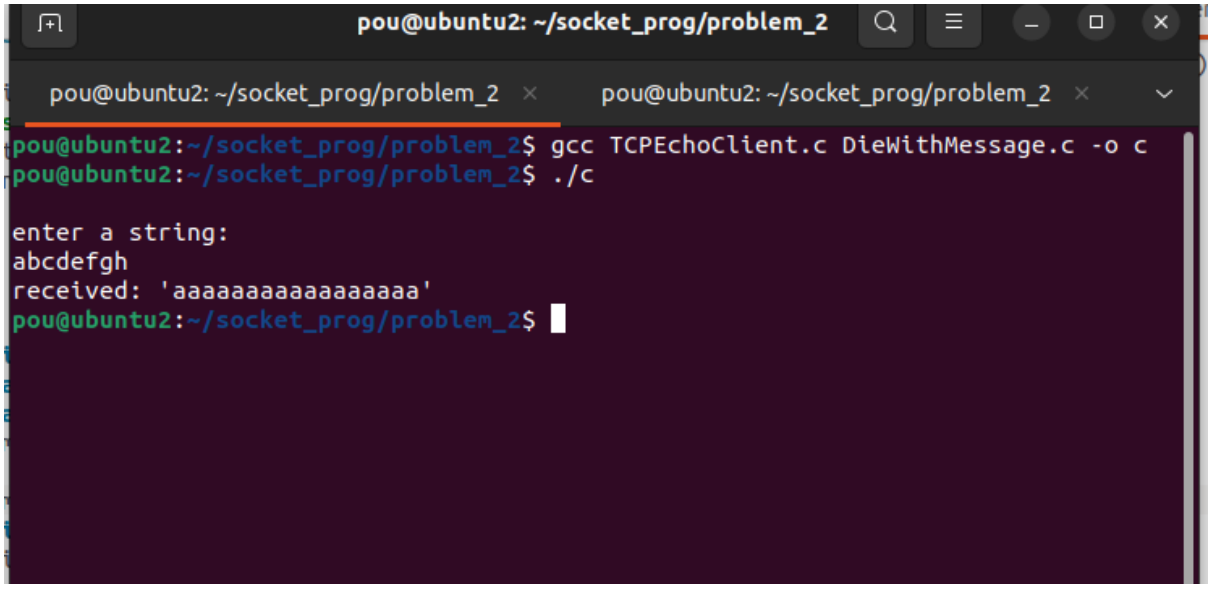

Figure 2.2: Output of TCPEchoClient

Chapter 3

Problem 3

Modify UDPEchoServer.c so that ECHOMAX is much shorter (say 5 bytes). Then use UDPEchoClient.c to send an echo string that is too long. What happens? Donahoo and Calvert (2009)

Solution: To solve this problem we need UDPEchoServer and UDPEchoClient program. The programs are based on the simple UDP client and server model. The server runs an infinite loop and client takes user input string and sends it to the server. The server receives the string and print. But server can echo at most 5 bytes and client is sending more bytes.

Algorithm 6 UDP ECHO Server

1: Set SERVER-PORT to the port number the server should run on
2: Set ECHO-MAX to the maximum number of bytes that can be echoed
3: Initialize server-address as a sockaddr-in struct and set its sin-family to AF-INET
4: Use htons to set the sin-port of server-address to SERVER-PORT in network byte order
5: Use htonl to set the sin-addr of server-address to INADDR-ANY in network byte order
6: Create a UDP socket using socket(PF-INET, SOCK-DGRAM, 0) and store the file descriptor in sock
7: **if** sock is less than 0 **then**
8: Print an error message and exit the program with status code 1
9: Bind the socket to the server address using bind$(sock, (structsockaddr*)\&server-address, sizeof(server-address))$
10: **if** bind returns a value less than 0 **then**
11: Print an error message and exit the program with status code 1
12: Initialize client-address as a sockaddr-in struct and set client-address-len to 0
13: **while** true **do**
14: Initialize buffer as a character array of length ECHO-MAX
15: Use recvfrom to read incoming data into buffer from a client, and store the number of bytes read in len
16: Use inet-ntoa to print the IP address of the client that sent the data, along with the data itself
17: Add a null terminator to the end of the data in buffer
18: Use sendto to send the data back to the client that sent it
19: Exit the program with status code 0

Algorithm 7 UDP Client

1: Initialize server name and port number
2: Create a `sockaddr-in` struct to represent the server address and set it to all zeros
3: Set the `sin-family` field of the `sockaddr-in` struct to `AF-INET`
4: Convert the server name from a string to a binary representation using `inet-pton()` and store it in the `sin-addr` field of the `sockaddr-in` struct
5: Convert the port number to network byte order using `htons()` and store it in the `sin-port` field of the `sockaddr-in` struct
6: Create a UDP socket using `socket()` with the `PF-INET` family and `SOCK-DGRAM` type
7: Prompt the user to enter a message to send to the server and store it in `data-to-send`
8: Send the message using `sendto()` with the socket, message, message length, server address, and server address length as arguments
9: Close the socket using `close()`

3.1 Result

We first run the Server program and than run Client program as shown in images below,

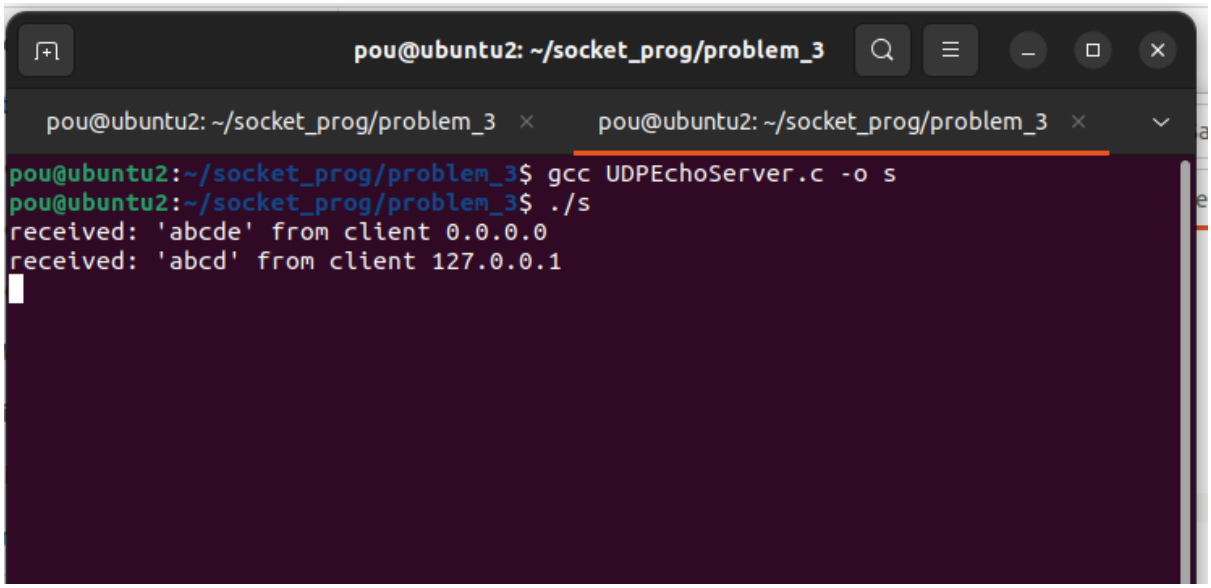

Figure 3.1: Output of UDPEchoServer

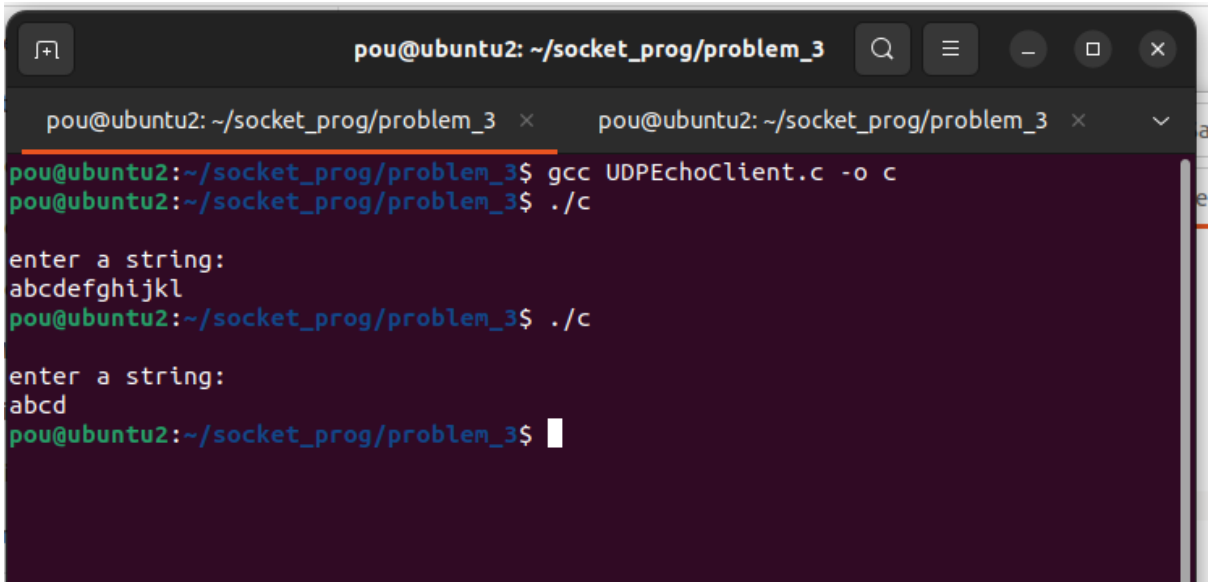

Figure 3.2: Output of UDPEchoClient

Chapter 4

Problem 4

Program for Remote Command Execution using sockets. Description: The client sends command and data parameters (if required) to server. The server will execute the command and send back the result to the client.

Solution: This code is an implementation of a remote command execution server using the UDP protocol. It runs an infinite loop in which it listens for incoming commands from clients over a specific port. Once a command is received, it executes it and sends the output back to the client. The server is implemented using socket programming.

Algorithm 8 Remote Command Execution using UDP Client

1: $PORT \leftarrow 10000$
2: $sockfd \leftarrow createUDPsocket$
3: **if** $sockfd < 0$ **then**
4: $\quad Printerrormessageandexitprogram$
5: $server_addr \leftarrow$ struct sockaddr$_i nwithfields$:sin$_f amily \leftarrow AF-INET$
$sin_port \leftarrow htons(PORT)$
$sin_addr.s_addr \leftarrow htonl(INADDR-ANY)$
6: $connect(sockfd, (structsockaddr*)server_addr, sizeof(server-addr))$
7: **while** $True$ **do**
8: $\quad print\ "EnterCommandToBeExecutedRemotely:"$
9: $\quad fgets(send-msg, sizeof(send-msg), stdin)$
10: $\quad sendto(sockfd, send_msg, sizeof(send-msg), 0, (structsockaddr*)server-addr, sizeof(server-addr))$
11: $\quad recvfrom(sockfd, recv_msg, sizeof(recv-msg), 0, (structsockaddr*)server-addr, serverLength)$
12: $\quad print\ "ServerReply:"$
13: $\quad print\ recv-msg$
14: $close(sockfd)$

This algorithm represents the main function of the remote-command-exec-udp-client.c program, which demonstrates remote command execution using the server-client model and socket programming with UDP. The algorithm uses various functions and structures to create and bind a socket to an internet address and port, connect the client to the server, and communicate between the client and server using the sendto() and recvfrom() functions. The algorithm also includes error handling for cases where the socket creation fails.

Algorithm 9 Remote Command Execution UDP Server

```
1: PORT is the port over which communication will take place
2: sockfd ← socket descriptor
3: recv_msg ← character array to read and store incoming data
4: success_message ← character array to store success message
5: server_addr ← sockaddr_in structure for server address
6: client_addr ← sockaddr_in structure for client address
7: clientLength ← size of the client address
8: function EXECUTE_COMMAND(command)
9:     buffer ← character array to read and store output of command
10:    result ← NULL
11:    pipe ← execute command with popen
12:    if pipe = NULL then
13:        print "Error executing command: command"
14:        return NULL
15:    while fgets(buffer, MAX_BUFFER_SIZE, pipe) ≠ NULL do
16:        if result = NULL then
17:            result ← duplicate of buffer
18:        else
19:            len ← length of result + length of buffer + 1
20:            temp ← reallocate memory for result
21:            if temp = NULL then
22:                free result
23:                close pipe with pclose
24:                print "Error allocating memory"
25:                return NULL
26:            result ← temp
27:            concatenate buffer to result
28:    close pipe with pclose
29:    return result
30: procedure ERROR
31:    print an error message to stderr using perror
32:    exit with failure status
33: sockfd ← create a UDP socket with socket
34: if sockfd < 0 then
35:    call ERROR
36: clear server_addr with bzero
37: set the fields of server_addr
38: bind the socket descriptor to the server address with bind
39: if binding is unsuccessful then
40:    call ERROR
41: print "Server is Connected Successfully..."
42: while true do
43:    receive data from client with recvfrom
44:    execute the received command with EXECUTE_COMMAND
45:    send the output of the command to the client with sendto
46:    print success_message
```

4.1 Result

We first run the Server program and than run Client program as shown in images below,

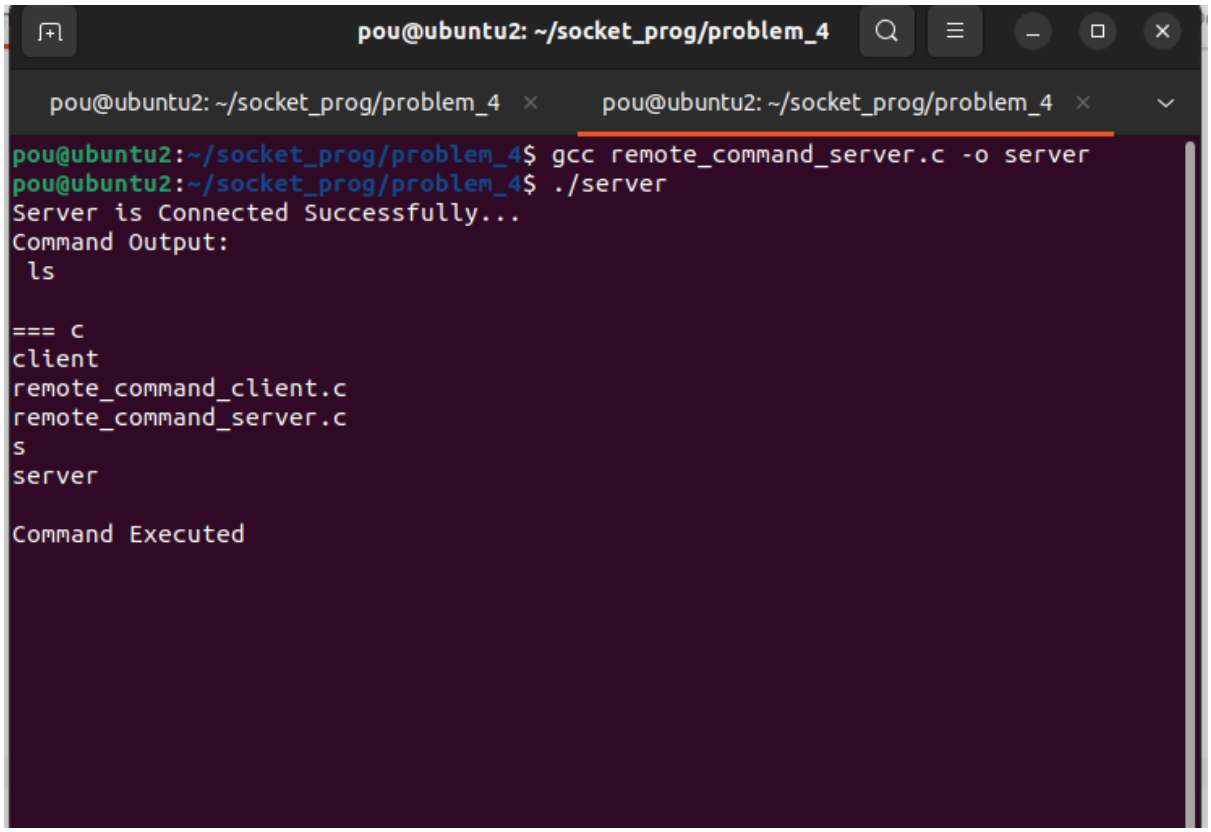

Figure 4.1: Output of remote command exec udp server

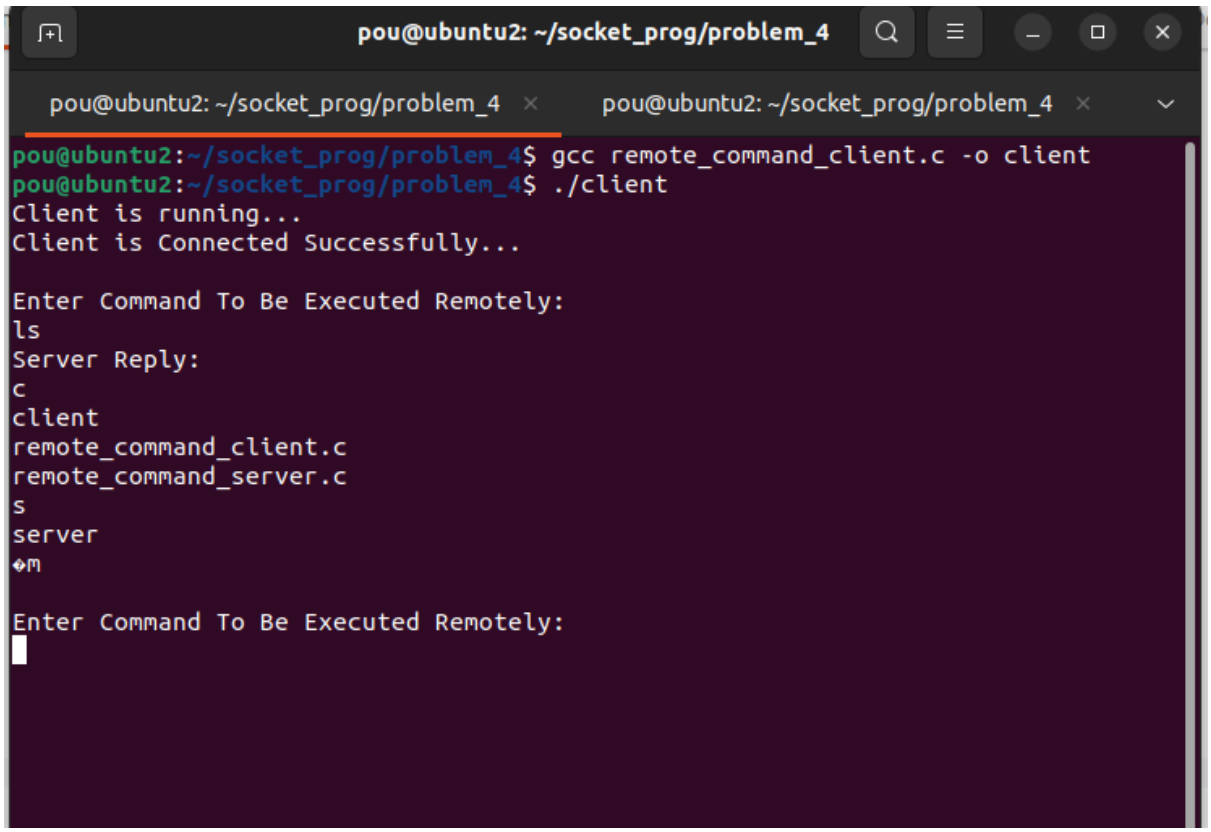

Figure 4.2: Output of remote command exec udp client.c

Chapter 5

Problem 5

Write a program to implement Web Server. Description: The Client will be requesting a web page to be accessed which resides at the Server side.

Solution: This is a basic C program that sets up a simple web server that listens on port 8080 and returns a hardcoded HTTP response.

Algorithm 10 Web Server

```
1:  PORT ← 8080
2:  BUFFER_SIZE ← 1024
3:  resp ← "HTTP/1.0200OKServer : webserver − cContent − type : text/html < html >
    code < /html > "
4:  sockfd ← SOCKET(AF_INET, SOCK_STREAM, 0)
5:  if sockfd == −1 then
6:      PERROR(webserver (socket))
7:      return 1
8:  PRINTF(socket created successfully)
9:  host_addr.sin_family ← AF_INET
10: host_addr.sin_port ← HTONS(PORT)
11: host_addr.sin_addr.s_addr ← HTONL(INADDR_ANY)
12: if BIND(sockfd, (struct sockaddr *)host_addr, sizeof(host_addr)) != 0 then
13:     PERROR(webserver (bind))
14:     return 1
15: PRINTF(socket successfully bound to address)
16: if LISTEN(sockfd, SOMAXCONN) != 0 then
17:     PERROR(webserver (listen))
18:     return 1
19: PRINTF(server listening for connections)
20: while true do
21:     newsockfd ← accept(sockfd, (struct sockaddr *)&host_addr, (socklen_t
    *)&sizeof(host_addr))  if newsockfd < 0 then
23:     perror(webserver (accept))
24:
25:     print connection accepted
26:     valread ← read(newsockfd, buffer, BUFFER_SIZE)
27:     if valread < 0 then
28:         perror(webserver (read))
29:
30:     valwrite ← write(newsockfd, resp, strlen(resp))
31:     if valwrite < 0 then
32:         perror(webserver (write))
33:
34:     close(newsockfd)
```

5.1 Result

We first run the Server program and then open browser and go to localhost:8080 as shown in images below,

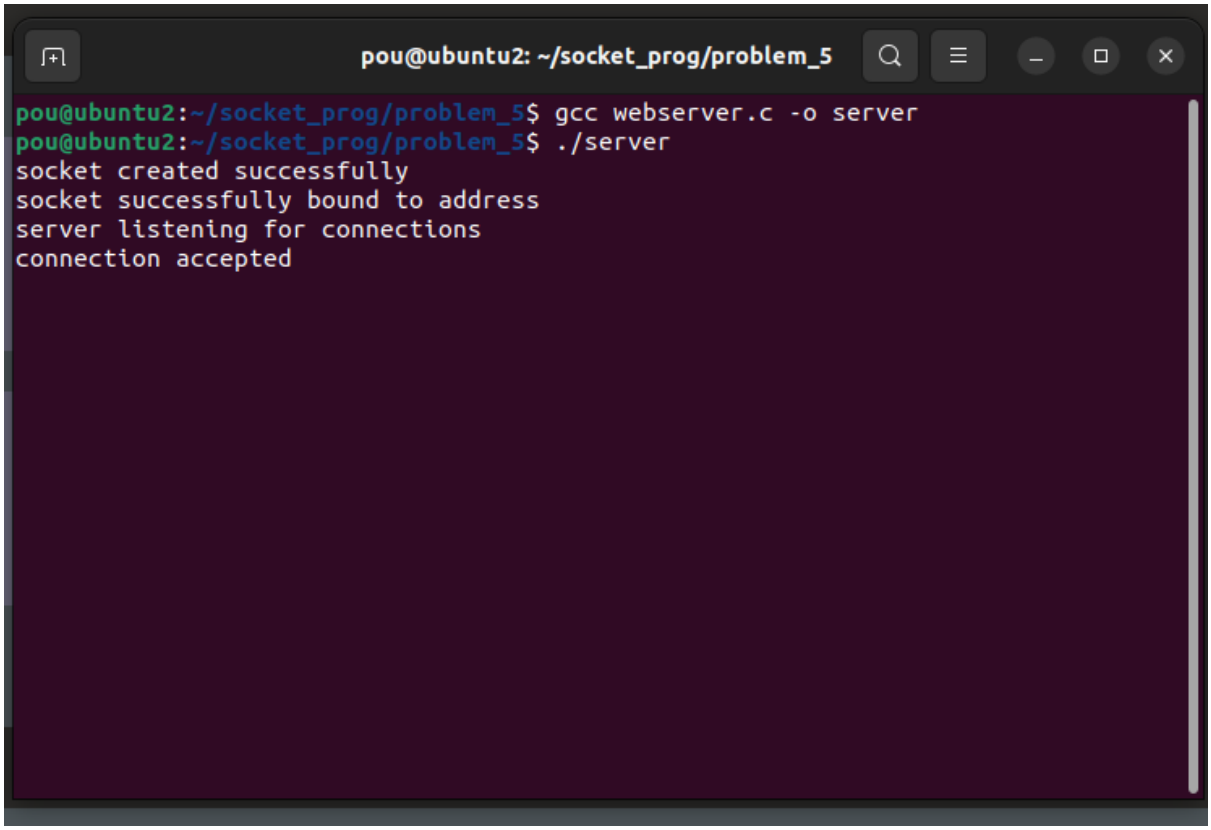

Figure 5.1: Output of webserver

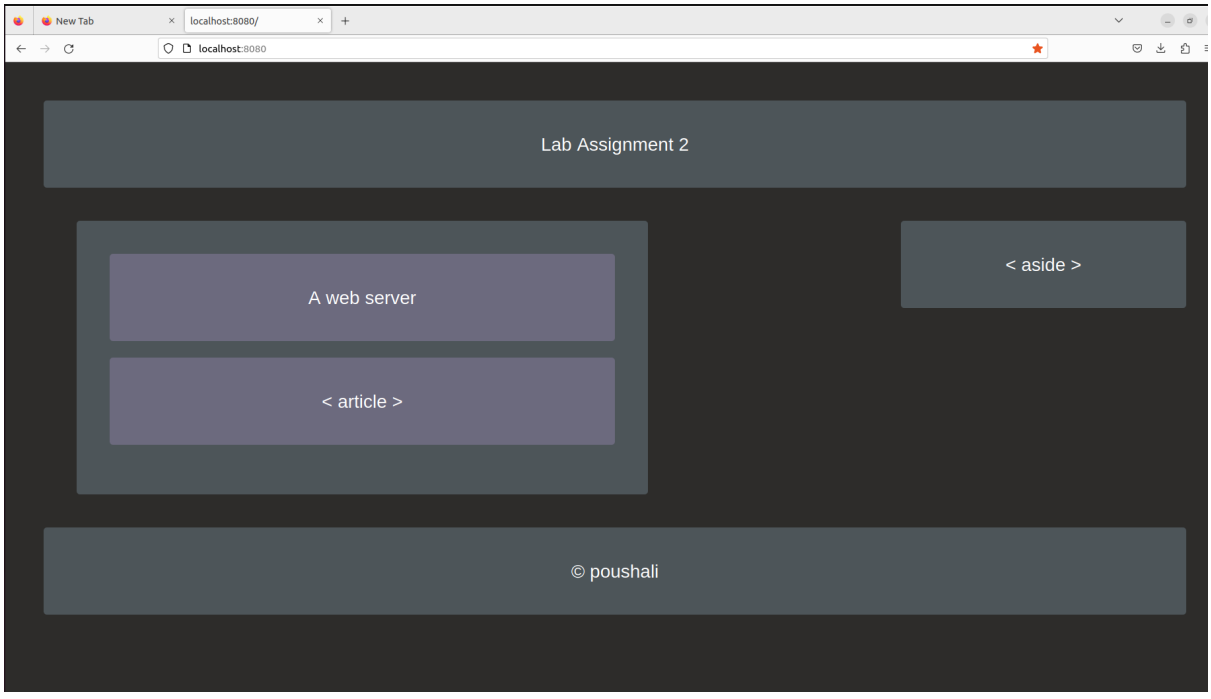

Figure 5.2: Output of browser client

Bibliography

Donahoo, M.J. and K.L. Calvert. 2009. *Tcp/ip sockets in c: Practical guide for programmers* (TCP/IP Sockets in C Bundle 1). Elsevier Science. URL `https://books.google.co.in/books?id=dmt_mERzxV4C`.