COMPUTER NETWORK (CS F303)

LAB-SHEET – 4

Topic: Socket Programming Part-II

Learning Objectives:

a) **Client-server program writing using UDP sockets**
b) **File Transfer Program (with partial file sending option) using TCP sockets**
c) **Creating concurrent server (handling multiple clients) using fork() system call**

---

a) A simple ECHO server using UDP sockets

In the last lab, we created a client/server based simple application using the TCP socket (or byte stream socket). Today, as the first exercise, we will see how to create a simple ECHO server and the client using UDP (message stream) sockets. By ECHO server, we mean that the server will reply the same message back, whatever it receives from the client.

Read, understand, and save the following file as **client_udp.c.** You can also download this file from Nalanda. While reading, try to locate and understand the specific differences with the TCP based client program that we did in the previous lab.

```
/* Simple udp client */

#include<stdio.h> //printf
#include<string.h> //memset
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>
#define BUFLEN 512  //Max length of buffer

#define PORT 8888   //The port on which to send data

void die(char *s)
{
    perror(s);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_other;
    int s, i, slen=sizeof(si_other);
    char buf[BUFLEN];
    char message[BUFLEN];

    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
    {
```

```
        die("socket");
    }

    memset((char *) &si_other, 0, sizeof(si_other));
    si_other.sin_family = AF_INET;
    si_other.sin_port = htons(PORT);
    si_other.sin_addr.s_addr = inet_addr("127.0.0.1");


    while(1)
    {
        printf("Enter message : ");
        gets(message);

        //send the message
        if (sendto(s, message, strlen(message) , 0 , (struct sockaddr *) &si_other,
slen)==-1)
        {
            die("sendto()");
        }
        //receive a reply and print it
        //clear the buffer by filling null, it might have previously received data
        memset(buf,'\0', BUFLEN);
        //try to receive some data, this is a blocking call
        if (recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *) &si_other, &slen) == -
1)
        {
            die("recvfrom()");
        }

        puts(buf);
    }

    close(s);
    return 0;
}
```

Read, understand, and save the following file as **server_udp.c.** You can also download this file from Nalanda. While reading, try to locate and understand the specific differences with the TCP based server program that we did in the previous lab.

```
/*  Simple udp server */

#include<stdio.h> //printf
#include<string.h> //memset
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>

#define BUFLEN 512  //Max length of buffer
#define PORT 8888   //The port on which to listen for incoming data
```

```
void die(char *s)
{
    perror(s);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_me, si_other;
    int s, i, slen = sizeof(si_other) , recv_len;
    char buf[BUFLEN];

    //create a UDP socket
    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
    {
        die("socket");
    }

    // zero out the structure
    memset((char *) &si_me, 0, sizeof(si_me));

    si_me.sin_family = AF_INET;
    si_me.sin_port = htons(PORT);
    si_me.sin_addr.s_addr = htonl(INADDR_ANY);

    //bind socket to port
    if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
    {
        die("bind");
    }

    //keep listening for data
    while(1)
    {
        printf("Waiting for data...");
        fflush(stdout);

        //try to receive some data, this is a blocking call
        if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *) &si_other,
&slen)) == -1)
        {
            die("recvfrom()");
        }

        //print details of the client/peer and the data received
        printf("Received packet from %s:%d\n", inet_ntoa(si_other.sin_addr),
ntohs(si_other.sin_port));
        printf("Data: %s\n" , buf);

        //now reply the client with the same data
        if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &si_other, slen) == -1)
        {
            die("sendto()");
```

```
        }
    }
    close(s);
    return 0;
}
```

The following diagram showing the sequence of function calls for the client and a server participating in a TCP and UDP would help you understand the differences between TCP and UDP socket programming better.

Exercise #1

Modify the ECHO server and client programs to a guessing game, where the server will generate a number (say between 1 to 6 or the name of a famous personality with some hint) and ask the client to guess it. The user will enter the guessed number (or name) through the terminal. If the guess is correct the client will win, otherwise it will lose. An appropriate message about the outcome can be printed at the client side.

b) Designing and implementing simple FTP client and server with broken download handling capability using TCP sockets.

In the second lab, we performed few experiments with Wireshark to understand functioning of standard FTP protocol. Here, we will develop our own simple client/server based application to get a file from the server. Currently, the program given here, simply downloads a predefined file from the server. It can be extended to include many other functionalities such as directory listing, "get" and "put" commands, as present in the standard FTP program. However, our program has the broken download capability which allows the client to complete a file transfer by downloading the remaining portion of a file only, if the file to be download is already present with the client partially. For example, if the client is already having initial 100 bytes of a files then next time, instead of downloading the complete file from the server, the client can request the server to send the file starting from byte number 101.

Read, understand, and save the following file as **client_broken_ftp.c.** You can also download this file from Nalanda.

```
/* Client program Broken FTP */

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <arpa/inet.h>

int main(void)
{
    int sockfd = 0;
    int bytesReceived = 0;
    char recvBuff[256];
    unsigned char buff_offset[10];   // buffer to send the File offset value
    unsigned char buff_command[2];   // buffer to send the Complete File (0)
or Partial File Command (1).
    int offset;                      // required to get the user input for
offset in case of partial file command
    int command;              // required to get the user input for command
    memset(recvBuff, '0', sizeof(recvBuff));
    struct sockaddr_in serv_addr;

    /* Create a socket first */
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0))< 0)
    {
        printf("\n Error : Could not create socket \n");
        return 1;
    }

    /* Initialize sockaddr_in data structure */
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5001); // port
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /* Attempt a connection */
    if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
    {
        printf("\n Error : Connect Failed \n");
        return 1;
    }

   /* Create file where data will be stored */
        FILE *fp;
        fp = fopen("destination_file.txt", "ab");
        if(NULL == fp)
        {
          printf("Error opening file");
            return 1;
        }
        fseek(fp, 0, SEEK_END);
        offset = ftell(fp);
        fclose(fp);
        fp = fopen("destination_file.txt", "ab");
        if(NULL == fp)
        {
```

```
            printf("Error opening file");
              return 1;
        }

    printf("Enter (0) to get complete file, (1) to specify offset, (2)
calculate the offset value from local file\n");
    scanf("%d", &command);
    sprintf(buff_command, "%d", command);
    write(sockfd, buff_command, 2);



    if(command == 1 || command == 2)   // We need to specify the offset
    {

        if(command == 1)  // get the offset from the user
        {
        printf("Enter the value of File offset\n");
        scanf("%d", &offset);
        }
        // otherwise offset = size of local partial file, that we have
already calculated
        sprintf(buff_offset, "%d", offset);
        /* sending the value of file offset */
        write(sockfd, buff_offset, 10);
    }

    // Else { command = 0 then no need to send the value of offset }


    /* Receive data in chunks of 256 bytes */
    while((bytesReceived = read(sockfd, recvBuff, 256)) > 0)
    {
        printf("Bytes received %d\n",bytesReceived);
        // recvBuff[n] = 0;
        fwrite(recvBuff, 1,bytesReceived,fp);
        // printf("%s \n", recvBuff);
    }

    if(bytesReceived < 0)
    {
        printf("\n Read Error \n");
    }

    return 0;
}
```

Read, understand, and save the following file as **server_broken_ftp.c.** You can also download this file from Nalanda.

/* Server program for broken ftp */

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>

int main(void)
{
    int listenfd = 0;
    int connfd = 0;
    struct sockaddr_in serv_addr;
    char sendBuff[1025];
    int numrv;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    printf("Socket retrieve success\n");

    memset(&serv_addr, '0', sizeof(serv_addr));
    memset(sendBuff, '0', sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5001);

    bind(listenfd, (struct sockaddr*)&serv_addr,sizeof(serv_addr));

    if(listen(listenfd, 10) == -1)
    {
        printf("Failed to listen\n");
        return -1;
    }


    while(1)
    {
        unsigned char offset_buffer[10] = {'\0'};
        unsigned char command_buffer[2] = {'\0'};
        int offset;
```

```
        int command;
        connfd = accept(listenfd, (struct sockaddr*)NULL ,NULL);


        printf("Waiting for client to send the command (Full File (0)
Partial File (1)\n");

        while(read(connfd, command_buffer, 2) == 0);
                sscanf(command_buffer, "%d", &command);

        if(command == 0)
                offset = 0;
        else
        {
                printf("Waiting for client to send the offset\n");
                while(read(connfd, offset_buffer, 10) == 0);
                sscanf(offset_buffer, "%d", &offset);

        }


        /* Open the file that we wish to transfer */
        FILE *fp = fopen("source_file.txt","rb");
        if(fp==NULL)
        {
            printf("File opern error");
            return 1;
        }

        /* Read data from file and send it */
                 fseek(fp, offset, SEEK_SET);
        while(1)
        {
            /* First read file in chunks of 256 bytes */
            unsigned char buff[256]={0};
            int nread = fread(buff,1,256,fp);
            printf("Bytes read %d \n", nread);

            /* If read was success, send data. */
            if(nread > 0)
            {
                printf("Sending \n");
                write(connfd, buff, nread);
            }

            /*
             * There is something tricky going on with read ..
             * Either there was error, or we reached end of file.
             */
```

```
            if (nread < 256)
            {
                if (feof(fp))
                    printf("End of file\n");
                if (ferror(fp))
                    printf("Error reading\n");
                break;
            }
        }
        close(connfd);
        sleep(1);
    }
    return 0;
}
```

Exercise #2

So, now as you understand the programming with UDP sockets and the working of our simple broken FTP application running over TCP socket, modify the above broken FTP client/server programs to make it run using UDP sockets.

c) Handling multiple clients at the same time

There are two main classes of servers, iterative and concurrent. An iterative server iterates through each client, handling it one at a time. A concurrent server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the **fork** function, creating one child process for each client. An alternative technique is to use **threads** instead (i.e., light-weight processes). In this lab, we consider fork based technique only.

A typical concurrent server has the following structure. The code in **blue color** is something which is different for iterative server and concurrent server.

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);

/***fill the socket address with server's well known port***/

bind(listenfd, ...);
listen(listenfd, ...);

for ( ; ; ) {
```

```
  connfd = accept(listenfd, ...); /* blocking call */

  if ( (pid = fork()) == 0 ) {

    close(listenfd); /* child closes listening socket */

    /***process the request doing something using connfd ***/
    /* ..... */

    close(connfd);
    exit(0);  /* child terminates
   }
   close(connfd);  /*parent closes connected socket*/
}
}
```

When a connection is established, accept returns, the **server calls fork**, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd. The parent closes the connected socket since the child handles the new client. The interactions among client and server are presented.

Exercise #3

Using this code, you can extend your Lab3 server TCP server to handle multiple clients simultaneously.

----------**Good Luck**----------