

CSCI3240 Lab 9:

Network Programming

1 Overview

In today's lab, we will be learning to use sockets for interprocess communication.

1.1 The Client Server Model

Most interprocess communication uses the client-server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

Notice also that once a connection is established, both sides can send and receive information. The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

Basically, the steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the `socket()` system call
2. Connect the socket to the address of the server using the `connect()` system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

And, the steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data

1.2 CSAPP API

The CSAPP API (csapp.c and csapp.h) provides you some functions that internally deal with some of the steps required on client side and server side to establish the connection.

1.2.1 open_clientfd and open_listenfd functions

Using CSAPP API, the client should only call the open_clientfd() function, which takes care of calling the socket() and connect() function internally, and the server should only call the open_listenfd() function, which takes care of calling the socket(), bind(), and listen() functions internally.

2 Sample Client and Server Code

2.1 Server

```
#include "csapp.h"

void serverFunction(int connfd){
    char buffer[MAXLINE]; //MAXLINE = 8192 defined in csapp.h
    char successMessage[MAXLINE] = "I got your message.\n\0";
    //TODO:
    //Add second message here
    //TODO End
    size_t n;

    //resetting the buffer
    bzero(buffer,MAXLINE);

    n = read(connfd, buffer, MAXLINE);

    printf("server received %ld bytes message\n", n);
    printf("Message from Client: %s\n",buffer);

    write(connfd,successMessage,strlen(successMessage));

    /*TODO:
        1. Add a code to receive new message from the client
        2. Send the message "I have received your second message" to the client
    */
    return;
}

int main(int argc, char *argv[])
{
    int listenfd;
    int connfd; //file descriptor to communicate with the client
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough space for any address */
```

```

char client_hostname[MAXLINE], client_port[MAXLINE];

if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);
}

listenfd = Open_listenfd(argv[1]); //wrapper function that calls getaddrinfo,
                                   //socket, bind, and listen functions
                                   //in the server side

//Server runs in the infinite loop.
//To stop the server process, it needs to be killed using the Ctrl+C key.
while (1) {
    clientlen = sizeof(struct sockaddr_storage);

    // wait for the connection from the client.
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    Getnameinfo((SA *) &clientaddr, clientlen, client_hostname,
                MAXLINE, client_port, MAXLINE, 0);

    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    //function to interact with the client
    serverFunction(connfd);

    Close(connfd);
    printf("(%s, %s) disconnected\n", client_hostname, client_port);
}
exit(0);
}

```

2.2 Client

```

#include "csapp.h"

int main(int argc, char *argv[])
{
    int clientfd; //file descriptor to communicate with the server
    char *host, *port;
    size_t n;

    char buffer[MAXLINE]; //MAXLINE = 8192 defined in csapp.h

    if (argc != 3)
    {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
    }
}

```

```

    exit(0);
}

host = argv[1];
port = argv[2];

clientfd = Open_clientfd(host, port); //wrapper function that calls getaddrinfo,
                                     //socket, and connect functions
                                     //in the server side

//getting a message from the user
printf("Please enter the message: ");

//resetting the buffer
bzero(buffer,MAXLINE);

//getting the message from the user
Fgets(buffer,MAXLINE,stdin);

//sending the message received from the user to the server
write(clientfd,buffer,strlen(buffer));

bzero(buffer,MAXLINE);
//waiting for the message from the server.
//the message will be stored in buffer variable.
read(clientfd,buffer,MAXLINE);

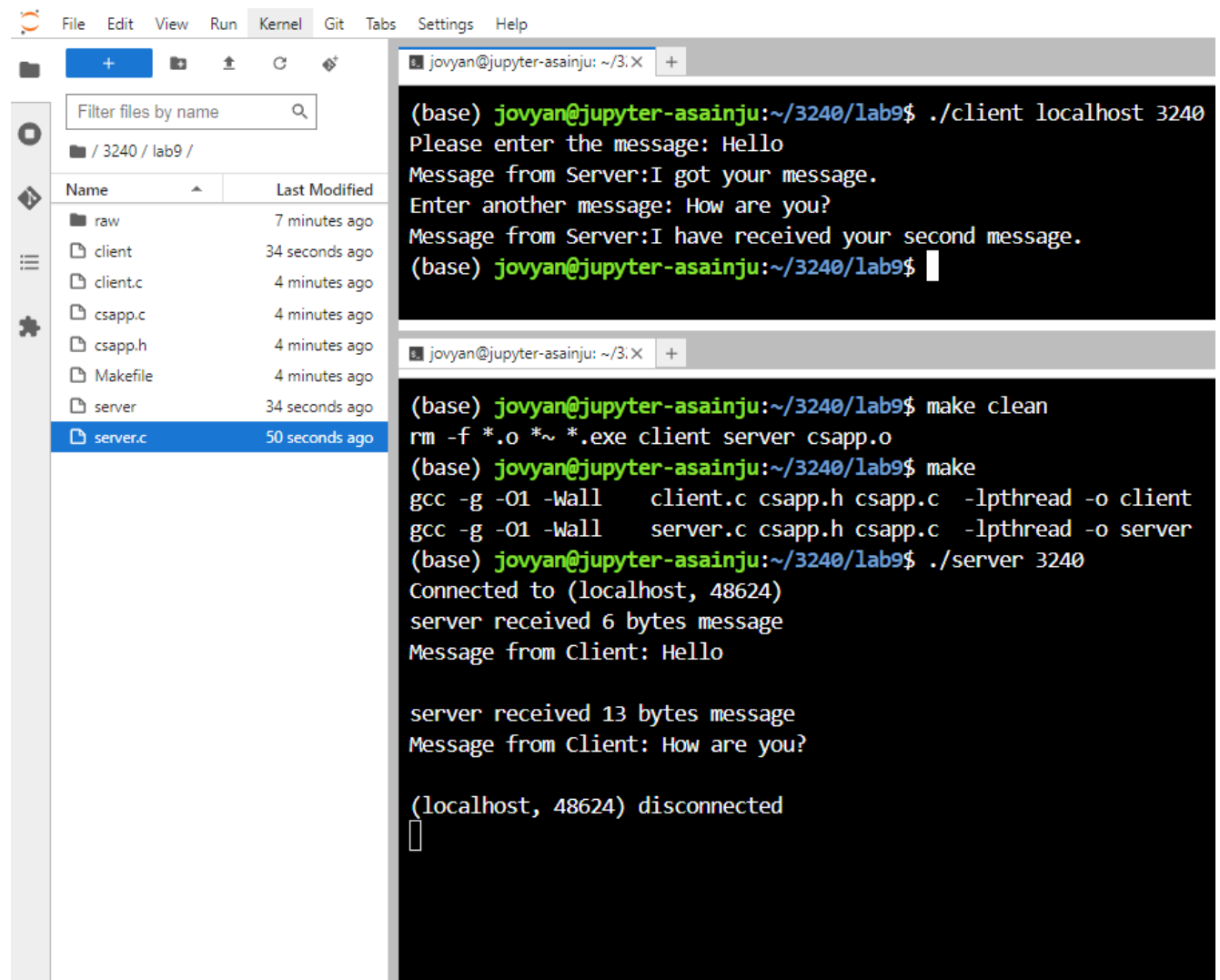
printf("Message from Server:");
//displaying the message in buffer on the console
Fputs(buffer,stdout);

/*TODO
    1. Add a code to send a new message to the server (you can ask the user
    \\to provide a new message in the terminal)
    2. Wait for the confirmation message (using read function) from the server
    \\and display it.
*/
Close(clientfd);
return 0;
}

```

3 Output of Sample Code

Here is an example of a run of the above program. We use a makefile to compile client and server programs. Link to a simple makefile tutorial: [Makefile Tutorial](#).



```
(base) jovyan@jupyter-asainju: ~/3240/lab9$ ./client localhost 3240
Please enter the message: Hello
Message from Server:I got your message.
Enter another message: How are you?
Message from Server:I have received your second message.
(base) jovyan@jupyter-asainju:~/3240/lab9$

(base) jovyan@jupyter-asainju:~/3240/lab9$ make clean
rm -f *.o *~ *.exe client server csapp.o
(base) jovyan@jupyter-asainju:~/3240/lab9$ make
gcc -g -O1 -Wall  client.c csapp.h csapp.c -lpthread -o client
gcc -g -O1 -Wall  server.c csapp.h csapp.c -lpthread -o server
(base) jovyan@jupyter-asainju:~/3240/lab9$ ./server 3240
Connected to (localhost, 48624)
server received 6 bytes message
Message from Client: Hello

server received 13 bytes message
Message from Client: How are you?

(localhost, 48624) disconnected
```

4 Submission Instructions

1. Complete the TODO sections in the client.c and server.c files. Upload a zip file: “lab9.zip” to Lab9 Dropbox in D2L. It should include:
 - (a) client.c
 - (b) server.c
 - (c) makefile
 - (d) csapp.c
 - (e) csapp.h
2. Submission Due: Check Lab9 Dropbox

5 Grading Rubrics

1. Server does not receive second message: -100
2. Client/Server compilation error: -50
3. Segmentation faults after correct execution: -30
4. If server terminates (Server should never terminate): -40
5. Client did not terminate: -25
6. Required files missing (other than client.c and server.c): -20
7. client.c or server.c missing: -100
8. Scanf used: -10
9. Incorrect file name(such as client (1).c instead of client.c): -20