University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2024)

Lab#	Date	Title	Due Date	Grade Release Date
Lab10	Week 10	**Client-Server Architecture**	Two-Week Lab Nov. 26, Tuesday Midnight	Dec. 02

The main objective of this lab will be to write programs that reside in different computers that are connected via a computer network and communicate. Specifically, we want to practice on a special scenario, Client-Server, where a process acts as a client and ask for some service from another process, the server. In this scenario, the client process either cannot to the task itself (does not know how to do it or the host computer does not have enough power for it), or it simply dispatches the tasks to the server or servers and merge the results.

For example, if I want to search about *`system programming'* on the Web, we ask Google to do it for me. In this case, my browser (chrome) is the client process in my computer, and Google is the server process in Google's computer. I confess that searching all web pages and retrieving relevant documents is a highly complex task. Why should I bother doing so myself? I ask search engines like Google to do it for me for free!

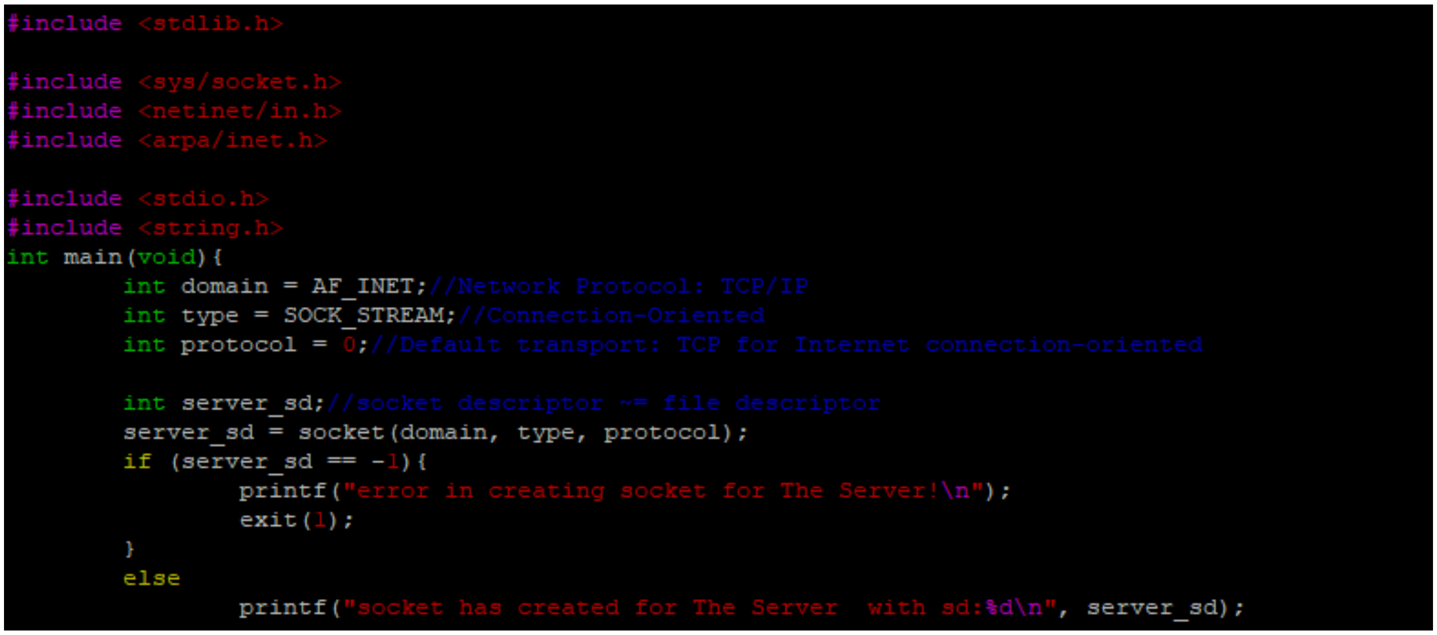
In the Client-Server scenario, we usually have many clients but a single server (*The* Server) to serve all the clients. It is a many-to-one relationship. Using TCP/IP network protocol, we can create two types of communications: i) connectionless (UDP) and ii) connection-oriented (TCP). For the client-server scenario, we use connection-oriented communication (TCP) to make sure that the server exists and is ready to answer the clients' request. To establish a connection-oriented communication, we need to do steps in the server and client parts:

In The Server part:
1) Create a socket
2) Bind to the host computer address (IP) and a Port, i.e., the endpoint IP:PORT
3) Listen for the client calls
4) If a client calls, accept the call
5) Receive the client's request
6) Send the client's answer
7) Go to (4)

In clients part:
1) Create a socket
2) [Optional] Bind to the host computer address (IP) and a Port, i.e., the endpoint IP:PORT
3) Connect to the server
4) Send the request to the server
5) Receive the answer from the server

Step 1. TCP Communication: The Server

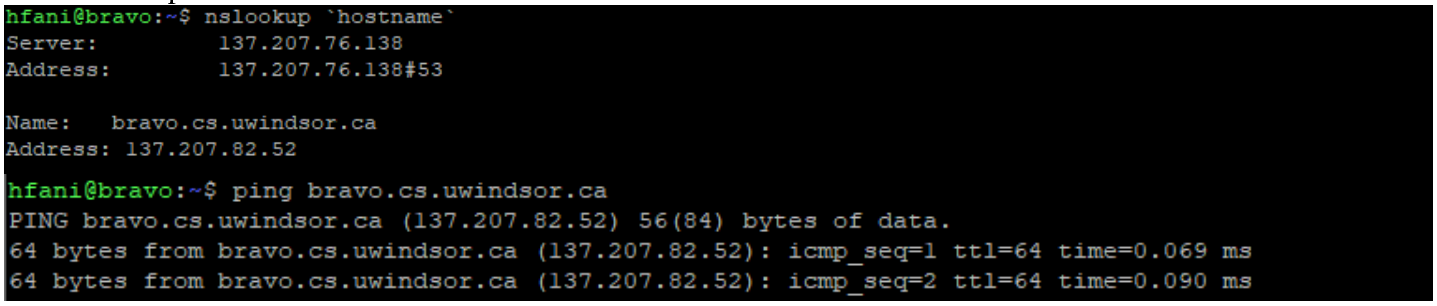
Let's do The Server part. (1) Foremost, we have to create a socket, based on TCP connection using TCP/IP network protocol:



```
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_STREAM;//Connection-Oriented
        int protocol = 0;//Default transport: TCP for Internet connection-oriented

        int server_sd;//socket descriptor ~= file descriptor
        server_sd = socket(domain, type, protocol);
        if (server_sd == -1){
                printf("error in creating socket for The Server!\n");
                exit(1);
        }
        else
                printf("socket has created for The Server  with sd:%d\n", server_sd);
```

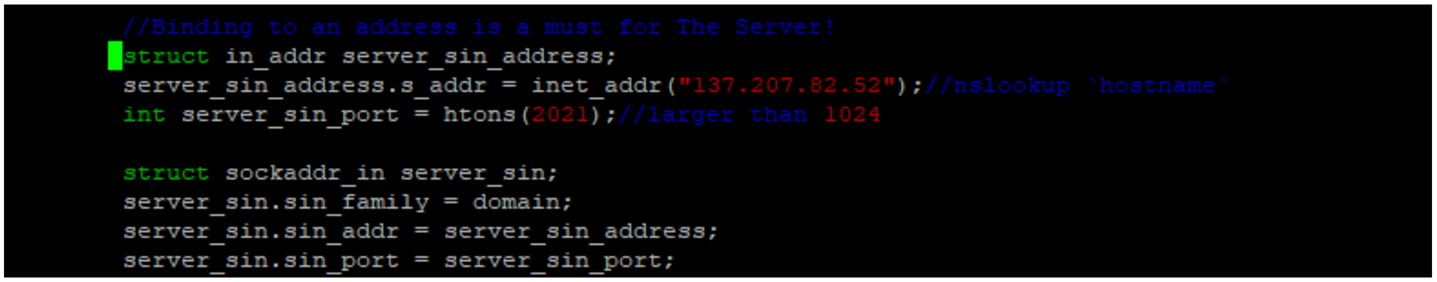
As seen, the TCP/IP network protocol can be selected by AF_INET (Address Family Internet) and the connection-oriented type of communication by SOCK_STREAM. When protocol = 0, the kernel uses the default or correct way of transportation for a network protocol which is TCP for TCP/IP protocol.

(2) Now, we need to bind the created socket server_sd to the address of the host computer and choose a port for The Server process. In TCP/IP network protocol, we need the IP address of the host computer. We can obtain the IP by running the shell command nslookup `hostname` (note ` character called back quote, SHIFT+~) or ping the host computer name:



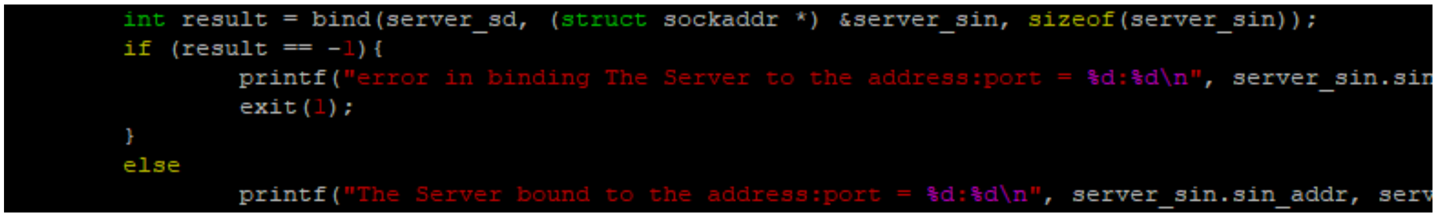
```
hfani@bravo:~$ nslookup `hostname`
Server:         137.207.76.138
Address:        137.207.76.138#53

Name:   bravo.cs.uwindsor.ca
Address: 137.207.82.52
hfani@bravo:~$ ping bravo.cs.uwindsor.ca
PING bravo.cs.uwindsor.ca (137.207.82.52) 56(84) bytes of data.
64 bytes from bravo.cs.uwindsor.ca (137.207.82.52): icmp_seq=1 ttl=64 time=0.069 ms
64 bytes from bravo.cs.uwindsor.ca (137.207.82.52): icmp_seq=2 ttl=64 time=0.090 ms
```

As seen, the IP is 137.207.82.52 for the bravo.cs.uwindsor.ca computer system. Further, we have to select a PORT number that should be easy to remember and larger than 1024 (lower numbers are reserved for superusers!) Let's pick 2021:



```
        //Binding to an address is a must for The Server!
        struct in_addr server_sin_address;
        server_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname`
        int server_sin_port = htons(2021);//larger than 1024

        struct sockaddr_in server_sin;
        server_sin.sin_family = domain;
        server_sin.sin_addr = server_sin_address;
        server_sin.sin_port = server_sin_port;
```

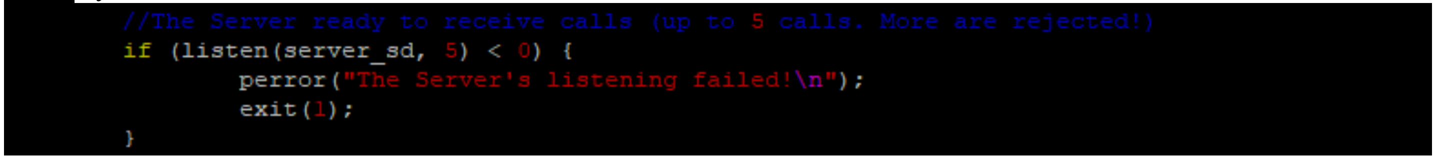
And bind the created socket server_sd to the IP:PORT end point, 137.207.82.52:2021. If successful, no other processes inside the host can be bound to the same IP:PORT.



```
        int result = bind(server_sd, (struct sockaddr *) &server_sin, sizeof(server_sin));
        if (result == -1){
                printf("error in binding The Server to the address:port = %d:%d\n", server_sin.sin
                exit(1);
        }
        else
                printf("The Server bound to the address:port = %d:%d\n", server_sin.sin_addr, serv
```

NOTE: All students are using the School's same computer but IP:PORT must be unique for a process. So, you may receive an error on binding your The Server PORT because another student's The Server may already occupy it. Try a different PORT for your The Server program!

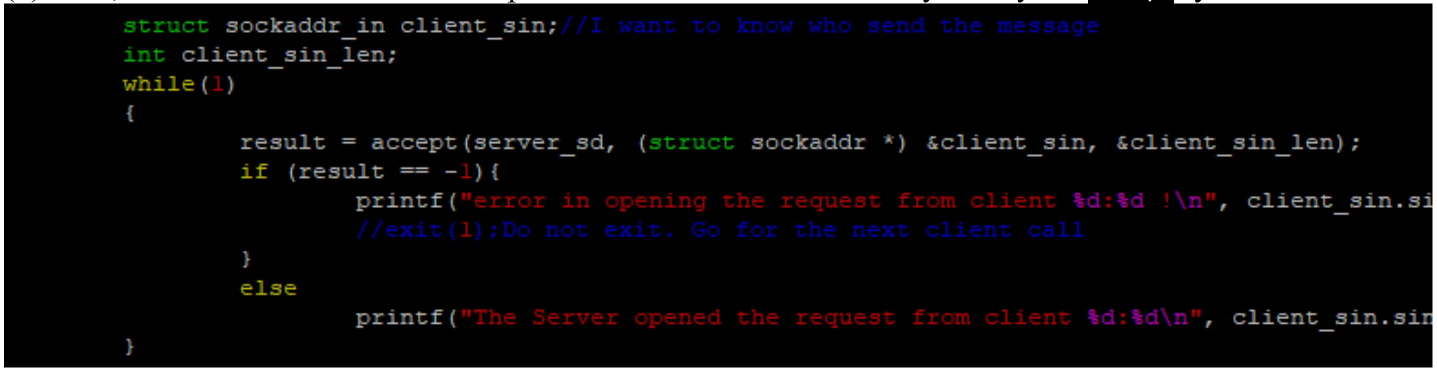
(3) Thereafter, The Server should tell the kernel that it is ready to receive calls from clients. This is done by the listen system call:



```
        //The Server ready to receive calls (up to 5 calls. More are rejected!)
        if (listen(server_sd, 5) < 0) {
                perror("The Server's listening failed!\n");
                exit(1);
        }
```

If a client calls, the kernel of the host computer puts it in the queue of The Server's socket if there is an empty space available. As seen, the listen system call accepts a number called backlog that tells the maximum size of the queue. Once the queue is full, the host kernel will reject additional client calls, so the backlog value must be chosen based on the expected load of The Server and the amount of processing it must do to accept a connection request and start the service.

(4) Now, The Server can look into the queue and serve the clients one by one by the accept system call:

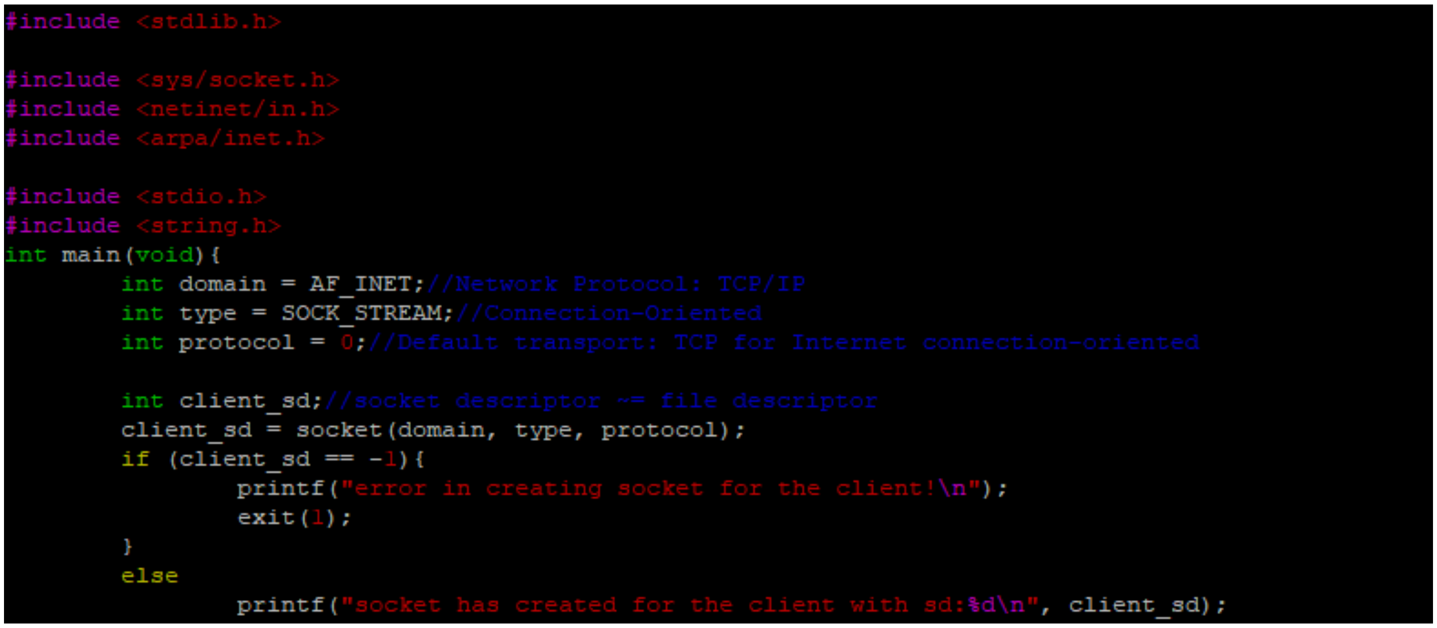


```
        struct sockaddr_in client_sin;//I want to know who send the message
        int client_sin_len;
        while(1)
        {
                result = accept(server_sd, (struct sockaddr *) &client_sin, &client_sin_len);
                if (result == -1){
                        printf("error in opening the request from client %d:%d !\n", client_sin.si
                        //exit(1);Do not exit. Go for the next client call
                }
                else
                        printf("The Server opened the request from client %d:%d\n", client_sin.sin
        }
```

Before going to the (5) and (6) steps of The Server, that is, receiving the client's request and sending back the result, let's create some clients and test The Server.

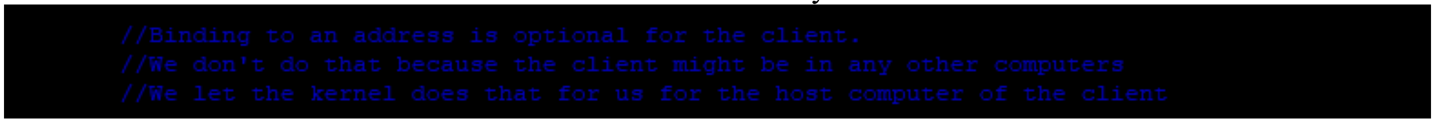
Step 2. TCP Communication: Clients

The first step for the client is the same as in The Server: (1) creating a socket:



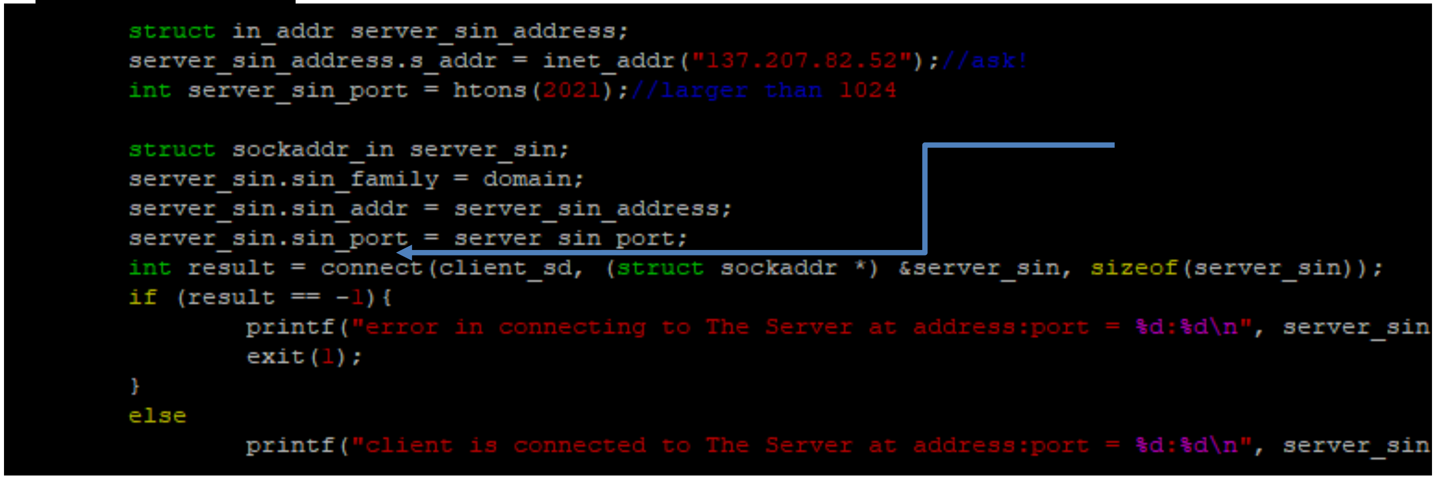
```
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_STREAM;//Connection-Oriented
        int protocol = 0;//Default transport: TCP for Internet connection-oriented

        int client_sd;//socket descriptor ~= file descriptor
        client_sd = socket(domain, type, protocol);
        if (client_sd == -1){
                printf("error in creating socket for the client!\n");
                exit(1);
        }
        else
                printf("socket has created for the client with sd:%d\n", client_sd);
```

(2) Binding the client's socket to the client's host IP and to a port is optional. Indeed, we never do that because the client program will be potentially run on different computers. So, we should not explicitly hard code the IP:PORT. We let the client's host kernel does that automatically:



```
        //Binding to an address is optional for the client.
        //We don't do that because the client might be in any other computers
        //We let the kernel does that for us for the host computer of the client
```

(3) To connect to The Server, we need to know its host computer IP and PORT. In step 1, The Server is listening at 137.207.82.52:2021

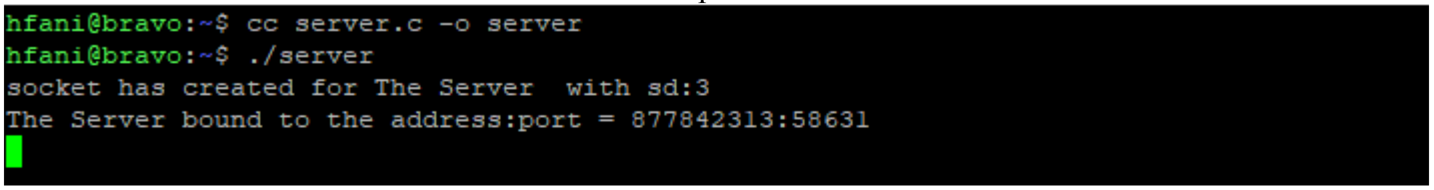


```
        struct in_addr server_sin_address;
        server_sin_address.s_addr = inet_addr("137.207.82.52");//ask!
        int server_sin_port = htons(2021);//larger than 1024

        struct sockaddr_in server_sin;
        server_sin.sin_family = domain;
        server_sin.sin_addr = server_sin_address;
        server_sin.sin_port = server_sin_port;
        int result = connect(client_sd, (struct sockaddr *) &server_sin, sizeof(server_sin));
        if (result == -1){
                printf("error in connecting to The Server at address:port = %d:%d\n", server_sin
                exit(1);
        }
        else
                printf("client is connected to The Server at address:port = %d:%d\n", server_sin
```

Before going to the (4) and (5) steps of the client, that is, sending the request and receiving the answer from The Server, let's test the clients and The Server connection first.

Step 3. TCP Communication: Clients → The Server

I built The Server code and ran it on the School's computer:

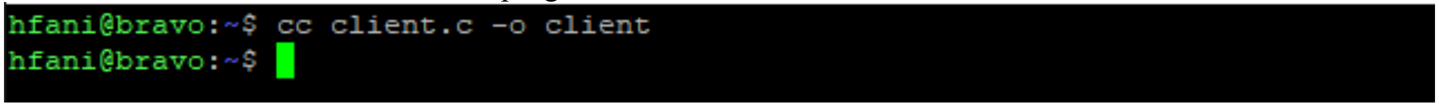


```
hfani@bravo:~$ cc server.c -o server
hfani@bravo:~$ ./server
socket has created for The Server  with sd:3
The Server bound to the address:port = 877842313:58631
```

As seen, it blocks (sleeps/waits) at the accept system call because there is no client call yet!

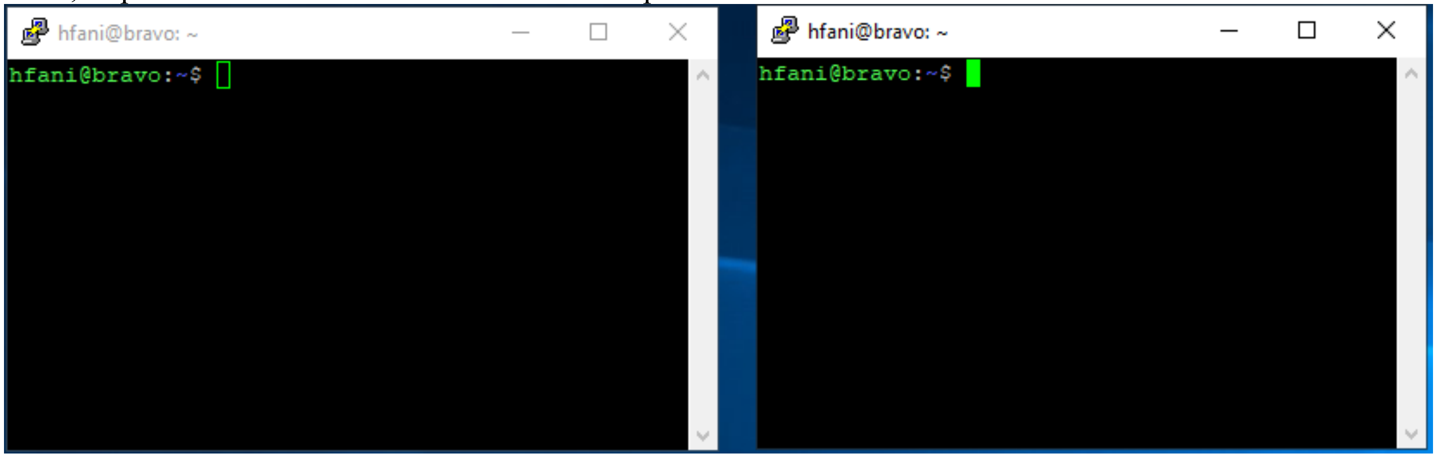
Although we write a single program for the client, we can run it on multiple computers to send their requests to

The Server. To do so, I built the client program:



```
hfani@bravo:~$ cc client.c -o client
hfani@bravo:~$ 
```

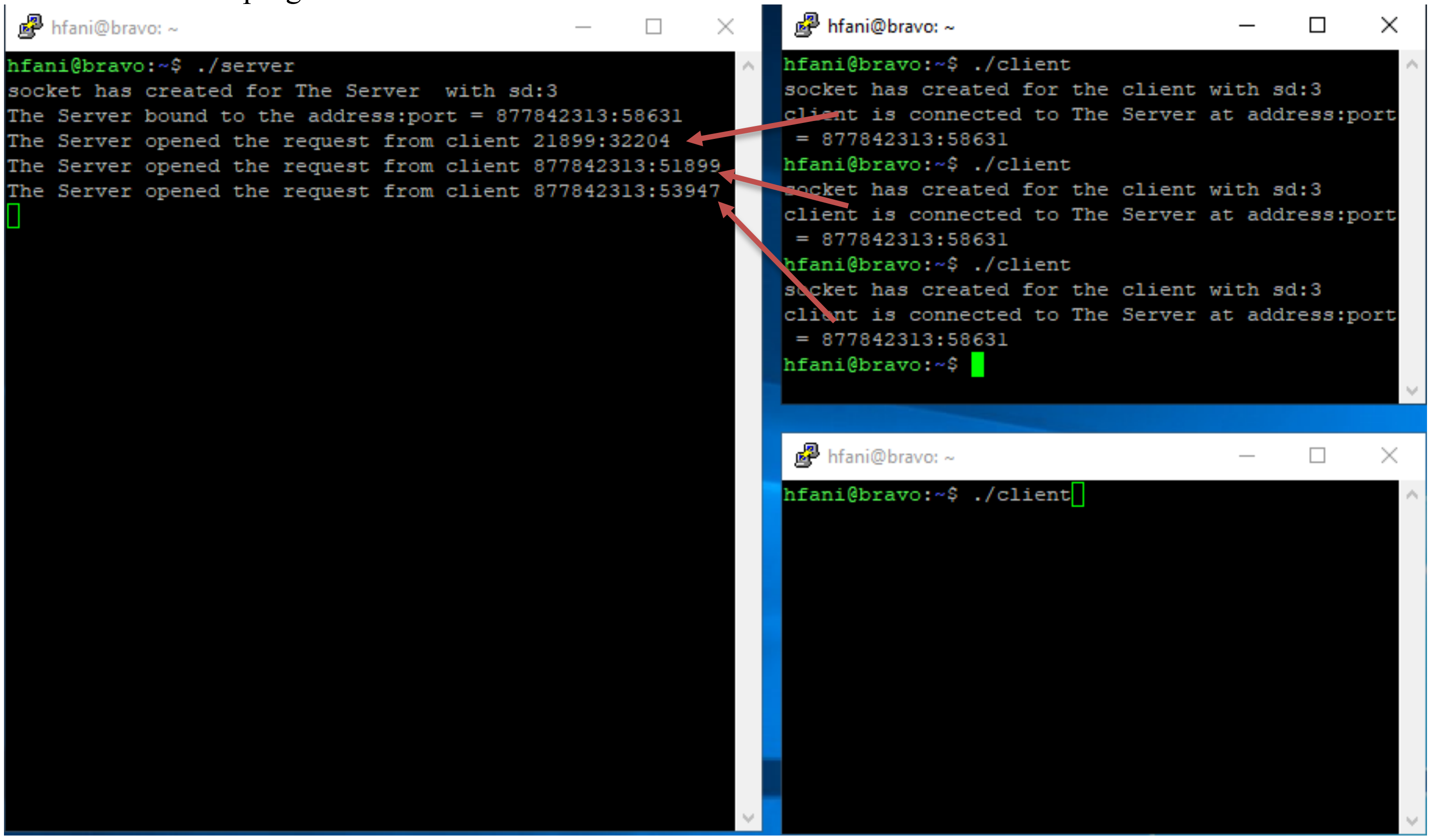
Then, I opened two terminals to the School's computer:



```
hfani@bravo: ~
hfani@bravo:~$ 
```

```
hfani@bravo: ~
hfani@bravo:~$ 
```

And run the client program in each of the terminals:

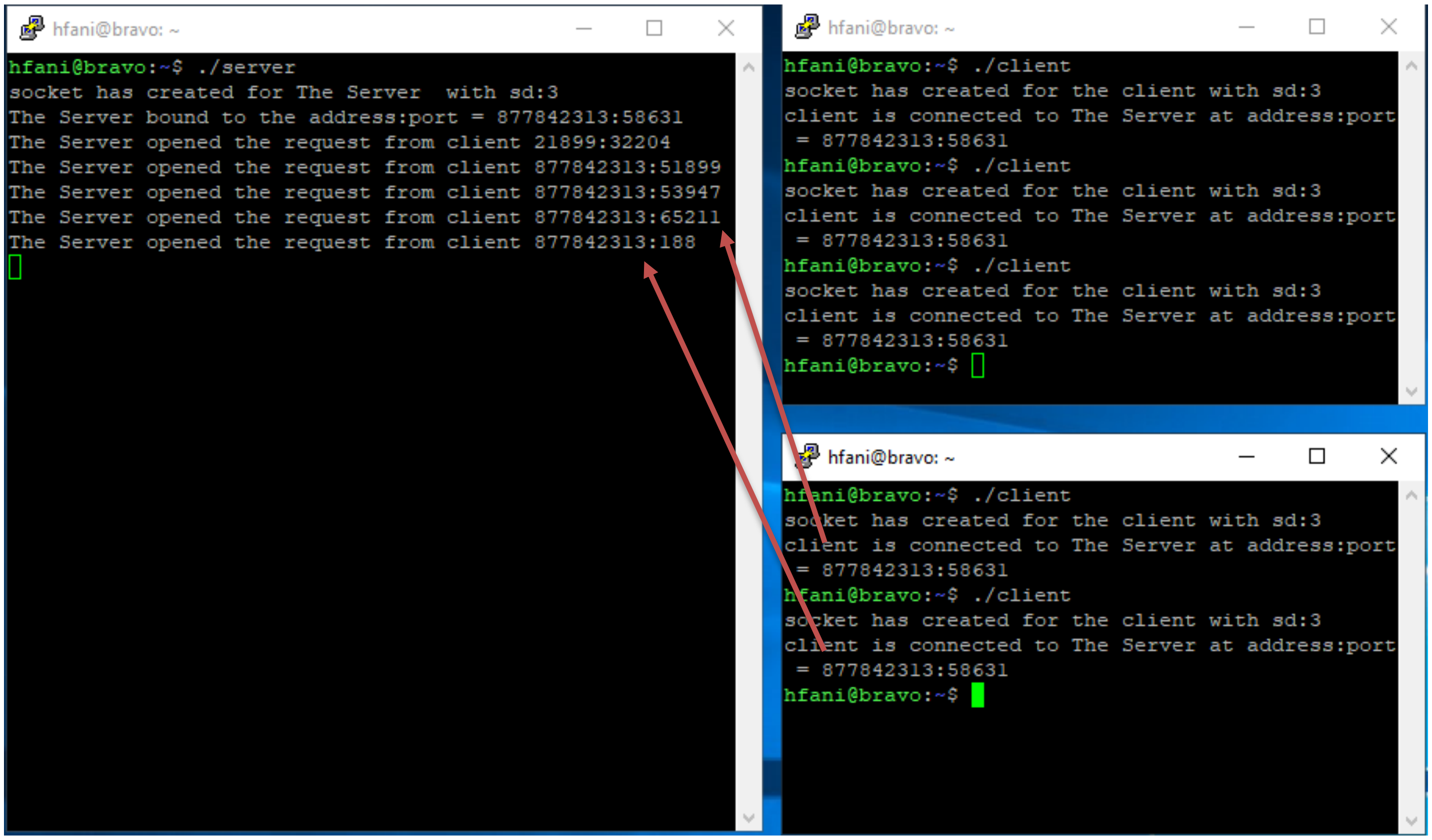


```
hfani@bravo: ~
hfani@bravo:~$ ./server
socket has created for The Server  with sd:3
The Server bound to the address:port = 877842313:58631
The Server opened the request from client 21899:32204
The Server opened the request from client 877842313:51899
The Server opened the request from client 877842313:53947
```

```
hfani@bravo: ~
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ 
```

```
hfani@bravo: ~
hfani@bravo:~$ ./client
```

Let's try it from the second client:



```
hfani@bravo:~$ ./server
socket has created for The Server  with sd:3
The Server bound to the address:port = 877842313:58631
The Server opened the request from client 21899:32204
The Server opened the request from client 877842313:51899
The Server opened the request from client 877842313:53947
The Server opened the request from client 877842313:65211
The Server opened the request from client 877842313:188
```

```
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$
```

```
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$ ./client
socket has created for the client with sd:3
client is connected to The Server at address:port
 = 877842313:58631
hfani@bravo:~$
```

As seen, The Server and the clients are all running on the same computer system. For the Network Protocol, it does not matter. A computer can communicate with itself! This is a huge advantage and allows an alternative to the IPC by pipe/FIFO/signal.

Also, the clients' IP:PORTs are handled properly by their kernels (in this case, same kernel.) Each run of a client process has been assigned to a random PORT, but the IP is the same (because the computer is the same.)

One issue is that the IPs are shown in weird numbers that are not the same as we entered for The Server. For instance, The Server IP has shown 877842313, but we entered 137.207.82.52. Can you explain it?

Step 4. Lab Assignment

You have to write The Server program that:

1) Accept IP and Port as inputs for the program (argv[1] and argv[2])
2) Receive (username, password, number x, number y) from the clients
3) Check the username == 'comp2560' and password=='f2022'
   a. If correct username and password, create a new child that adds x to y and returns the result to the client
   b. Else, send the message 'authentication failed!' to the client
4) Does (2) and (3) forever.

You have to write a client program that:

1) Accept IP and Port of The Server as input for the program (argv[1] and argv[2])
2) Ask two numbers from the user
3) Ask username and password from the user
4) Send the numbers, username and password to The Server
5) Receive The Server's answer and show it to the user
6) Does (2), (3), (4) and (5) forever until the user input -1 for the both inputs in (2).

Design Issue: The child that is assigned to a client should exit after the client does not exist anymore. You have to handle this by a `good` design.

The sample code for steps 1, 2 and 3 have been attached in a zip file named lab10_hfani.zip.

1.1. Deliverables

You will prepare and submit the program in one single zip file lab10_uwinid.zip containing the following items:

(90%) lab10_uwinid.zip

- (40%) server.c =>does the addition per client's request by a child
- (40%) client.c =>asks the user for two numbers, username, password
- (10%) results.pdf/jpg/png =>the image snapshot of communications with The Server for two clients for successful *and unsuccessful authentications*.
- (Optional) readme.txt

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of uwinid, use your own account name, e.g., mine is hfani@uwindsor.ca, so, lab10_hfani.zip