**EE533 Network Processor Design & Programming**

**Lab #1: Familiarity with VM & Sockets**

Instructor: Prof. Young Cho, PhD

***Name: Archit Sethi***

***University of Southern California***

***Los Angeles, CA 90007***

GITHUB LINK to my REPOSITORY

Aarch0811/EE533/LAB1/

Archit’s GITHUB: [Link](https://github.com/Aarch0811/EE533/tree/main/LAB1)

1. **Setting up two virtual machine nodes on Vmware/VirtualBox**

I’ve downloaded all the links from lab-1 pdf uploaded on USC Brightspace account.

The first steps involved downloading Virtual Machine on our systems & then downloading a Linux-based OS (e.g. ubuntu).

I’ve used VirtualBox to work with Ubuntu & not VMware Workstation Pro. I was unable to establish my connection with internet, so I took the advise of TA Pavan to use VirtualBox.

Procedure, I followed for Part-1:

1. Download & install VMware/VirtualBox on our systems.
2. Create 2 Linux VMs using VMware/VirtualBox, naming them ‘server’ & ‘client’ respectively.
3. Configure networking between the 2 VMs to allow communication between them.

Just, to explore a simpler windows experience in Linux-based Operating System (OS), I’ve used Linux Cinnamon Mint. It is lighter than Ubuntu and occupies less space.

Managed to get the complete resolution as per my screen.

* 1. **Server Virtual Machine**

As you can see in the image below, I’ve added the *bridged adapter* ON, which will allow me to communicate with the client VM.

A screenshot of a computer

Description automatically generated

* 1. **Client Virtual Machine**

Similarly, with the client VM, I’ve added the *Network Adapter* to communicate its messages to the server VM.

A screenshot of a computer

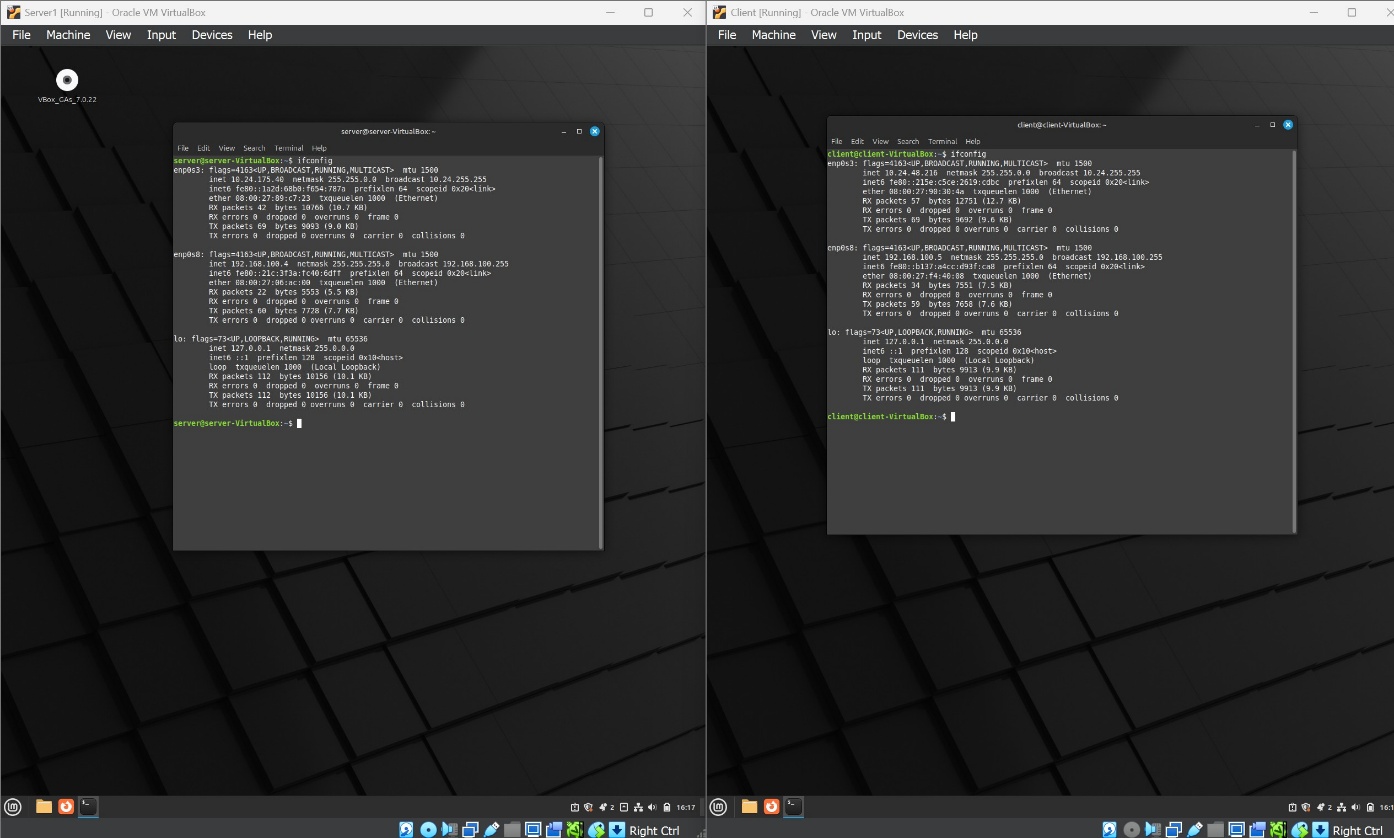
Description automatically generated

Result, I’ve shown the pinned version of the 2 VMs (server & client) side-by-side with there ip addresses to display that both are working and communicating (pinging) perfectly.

A screenshot of a computer

Description automatically generated

Configuration settings for 2-way communication between 2 VMs: Kindly please look at the network modifications that I made in the settings of virtual machines so that there is ‘ping’ between the server & client.



1. **Implement Socket Communication**
   1. **Establish socket-based communication between the server and client.**

They are follows:  
NOTE: The client needs to know that the server exists and its address (server’s address), but the opposite doesn’t need to the know the existence of client prior to the connection being established.

Procedure for socket creation for ‘server.c’:

1. Create socket() with system call *socket()*
2. Bind the socket to an address using *bind()* system call. The port number on the host machine is the address of the server socket.
3. Listen for connections of a port number using *listen()* system call.
4. Accept an incoming connection using accept() system call.
5. Send and receive data using *read()* and *write()* calls
   1. **Using the provided sample code on Brightspace *‘server.c’* & *‘client.c’* to implement a TCP-based communication system.**

To implement socket-based communication, we first must import the relevant libraries into our ‘server.c’ & ‘client.c’ codes.

Libraries added in the server & client codes for socket programming:

#include <stdio.h> 🡪 C programs uses input and output declarations

#include <sys/types.h> 🡪 defines no. of data types used in system calls.

#include <sys/socket.h> 🡪 includes no. of definitions of structures needed for sockets.

#include <netinet.h> 🡪constraints and structures used for internet domain addresses.

* 1. **Compile the server and client programs into executables named *‘server’* & *‘client’***

We compile our server.c & client.c using “g++ -o <executable\_name> <file.c>” where file.c is my C source code (server.c & client.c) & ‘-o <executable\_name>’ specifies the output name of the executable (server & client)

1. **Testing the communication:**
   1. **Run the server program with specifies port number (like I’ve used server 51000)**

The server is specified with port number 51000. We can select any port number withing a certain range of addresses.

Right now the server is in listen state. Waiting a message from the client.

**A screenshot of a computer

Description automatically generated**

* 1. **Run the client code, providing the server hostname & port number**

‘client’ executable was created and while run-time server’s IP address was specified as 192.168.100.4 with port number 51000

A screenshot of a computer

Description automatically generated

* 1. **Verify that the server displays the message sent by client and sends an acknowledgement back to the client.**

Server’s response:

The server displays the message sent by client. Here I had written in the message, ‘Hi, My name is Archit. My client is able to communicate properly with the server. This message is being displayed in the output window shown below.

A screenshot of a computer

Description automatically generated

Client’s response: Client acknowledges send by the server. It displays ‘I got your message’.

A screenshot of a computer

Description automatically generated

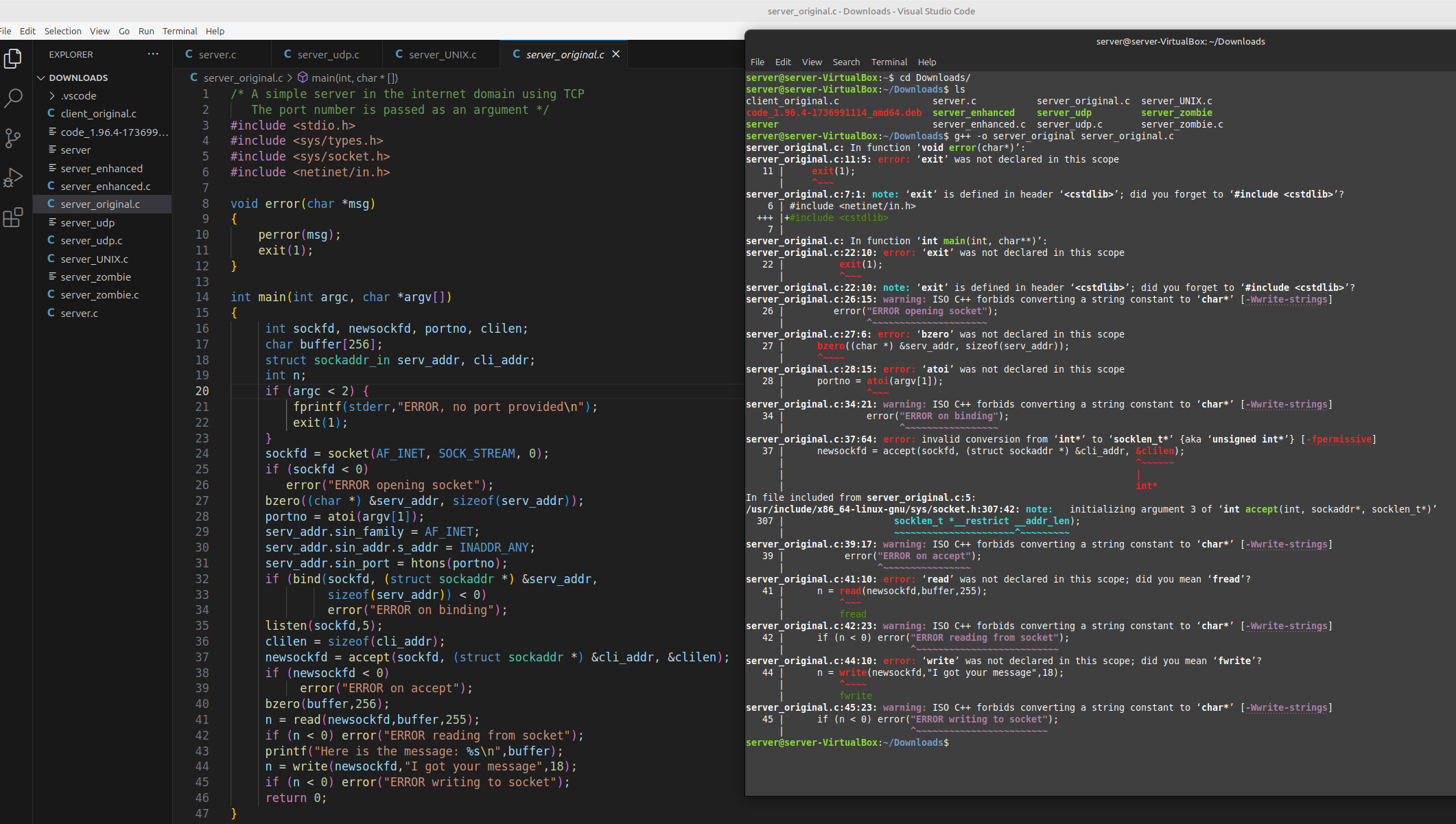
Let us play a little with the codes given to me. I’ll try to connect the client to a port number which doesn’t exist (e.g. 55122). The host port number is 55121.

It throws an exception by displaying the message as “ERROR connecting: Connection refused”

A screenshot of a computer

Description automatically generated

1. **Sample Code**
   1. **Source codes provided to us on USC Brightspace:**
      1. **Server.c changes made by me**

****

As you can see clearly the codes provided to us on Brightspace were full of bugs. There were many errors which I had resolved. These were library issues, variable declarations, wrong parameters used, etc.,

The updated and b ug-fee source code has been depicted in the snipper below. It was compiled properly and was able to generate the executable file.

* + 1. **Client.c changes made by me**

As you can see clearly the codes provided to us on Brightspace were full of bugs. There were many errors which I had resolved. These were library issues, variable declarations, wrong parameters used, etc.,

The updated and bug-fee source code has been depicted in the snipper below. It was compiled properly and was able to generate the executable file.

A screen shot of a computer screen

Description automatically generated

* 1. **The SERVER code**

1. /\* A simple server in the internet domain using TCP \*/
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <netdb.h>
6. #include <stdio.h>
7. #include <sys/types.h>
8. #include <sys/socket.h>
9. #include <netinet/in.h>
10. void error(const char \*msg)
11. {
12. perror(msg);
13. exit(1);
14. }
15. int main(int argc, char \*argv[])
16. {
17. int sockfd, newsockfd, portno;
18. socklen\_t clilen;
19. char buffer[256];
20. struct sockaddr\_in serv\_addr, cli\_addr;
21. int n;
22. if (argc < 2) {
23. fprintf(stderr, "ERROR, no port provided\n");
24. exit(1);
25. }
26. sockfd = socket(AF\_INET, SOCK\_STREAM, 0);
27. if (sockfd < 0)
28. error("ERROR opening socket");
29. bzero((char \*)&serv\_addr, sizeof(serv\_addr));
30. portno = atoi(argv[1]);
31. serv\_addr.sin\_family = AF\_INET;
32. serv\_addr.sin\_addr.s\_addr = INADDR\_ANY;
33. serv\_addr.sin\_port = htons(portno);
34. if (bind(sockfd, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)
35. error("ERROR on binding");
36. listen(sockfd, 5);
37. clilen = sizeof(cli\_addr);
38. newsockfd = accept(sockfd, (struct sockaddr \*)&cli\_addr, &clilen);
39. if (newsockfd < 0)
40. error("ERROR on accept");
41. // Corrected code block
42. bzero(buffer, 256);
43. n = read(newsockfd, buffer, 255);
44. if (n <= 0) {
45. error("ERROR reading from socket");
46. }
47. buffer[n] = '\0'; // Null-terminate the received string
48. printf("Here is the message: %s\n", buffer);
49. n = write(newsockfd, "I got your message", 18);
50. if (n < 0) {
51. error("ERROR writing to socket");
52. }
53. // Cleanup
54. close(newsockfd);
55. close(sockfd);
56. return 0;
57. }
    1. **The CLIENT code**
58. #include <stdlib.h>
59. #include <string.h>
60. #include <unistd.h>
61. #include <netdb.h>
62. #include <stdio.h>
63. #include <sys/types.h>
64. #include <sys/socket.h>
65. #include <netinet/in.h>
66. #include <netdb.h>
67. void error(const char \*msg)
68. {
69. perror(msg);
70. exit(0);
71. }
72. int main(int argc, char \*argv[])
73. {
74. int sockfd, portno, n;
75. struct sockaddr\_in serv\_addr;
76. struct hostent \*server;
77. char buffer[256];
78. if (argc < 3) {
79. fprintf(stderr,"usage %s hostname port\n", argv[0]);
80. exit(0);
81. }
82. portno = atoi(argv[2]);
83. sockfd = socket(AF\_INET, SOCK\_STREAM, 0);
84. if (sockfd < 0)
85. error("ERROR opening socket");
86. server = gethostbyname(argv[1]);
87. if (server == NULL) {
88. fprintf(stderr,"ERROR, no such host\n");
89. exit(0);
90. }
91. bzero((char \*) &serv\_addr, sizeof(serv\_addr));
92. serv\_addr.sin\_family = AF\_INET;
93. bcopy((char \*)server->h\_addr,
94. (char \*)&serv\_addr.sin\_addr.s\_addr,
95. server->h\_length);
96. serv\_addr.sin\_port = htons(portno);
97. if (connect(sockfd,(struct sockaddr \*)&serv\_addr,sizeof(serv\_addr)) < 0)
98. error("ERROR connecting");
100. printf("Please enter the message: ");
101. bzero(buffer, 256);
102. fgets(buffer, 255, stdin);
103. buffer[strcspn(buffer, "\n")] = '\0'; // Remove trailing newline
104. n = write(sockfd, buffer, strlen(buffer));
105. if (n <= 0) {
106. error("ERROR writing to socket");
107. }
108. bzero(buffer, 256);
109. n = read(sockfd, buffer, 255);
110. if (n <= 0) {
111. error("ERROR reading from socket");
112. }
113. buffer[n] = '\0'; // Null-terminate the received string
114. printf("%s\n", buffer);
115. // Close the socket
116. close(sockfd);
117. }

***Explanation for Server code ‘server.c’:***

* 1. Import relevant libraries essential for socket programming, input/output operations, and error handling.

A screenshot of a computer code

Description automatically generated

* 1. There is an error handling function that whenever a system call fails, making debugging easier.

A screen shot of a computer code

Description automatically generated

* 1. Main function: The program expects the port number as a Command-Line Argument. If NO port is provided, it prints an error message and exits.
  2. Socket Creation: Server creates a socket, binds it to the port, and listens for incoming connections.
     1. socket(AF\_INET, SOCK\_STREAM, 0):

AF\_INET: Specifies IPv4.

SOCK\_STREAM: Specifies TCP (stream-based protocol).

The return value is the file descriptor for the created socket.

If socket() fails, it calls error() to terminate the program.

* 1. Server Address Initialization:
     1. bzero: Clears the serv\_addr structure.
     2. portno: Converts the port argument (string) to an integer.
     3. sin\_family: Specifies IPv4.
     4. sin\_addr.s\_addr: Specifies the server's IP address. INADDR\_ANY allows connections from any network interface.
     5. htons: Converts the port number to big-endian byte number.
  2. Binding the socket:
     1. bind: Associates the socket (sockfd) with the server’s address (serv\_addr) and port.
     2. If bind() fails, then it prints an error and exits.
  3. Listen for connections: listen(sockfd, 5)
     1. listen: Marks the socket as passive i.e. it will accept incoming connecitons.
     2. The 2nd parameter specifies the maximum number of pending connections.
  4. Accept a connection: When a client connects, the server accepts the connection, creating a new socket.
     1. Accept: Waits for an incoming client connection.
     2. cli\_addr: Stores the address of the client
     3. newsockfd: A new file descriptor for the connection with the client/
     4. If no connection can be accepted, it prints an error and exits.
  5. Send data to client:

A black background with white text

Description automatically generated

write: Sends a message back to client.

n: stores the number of bytes

* 1. Close the sockets: Properly close the client and server to release resources.
     1. close(newsockfd)
     2. close(sockfd)

Why we use this server program?

The server.c program is simple TCP server that listens for incoming connections, exchanges messages with client and terminates after handling the communication. It servers as a base for understanding how servers operate in networked environment.

***Explanation for Client code ‘client.c’:***

1. Resolve Server Address

gethostname(argv[1]): resolves the hostname to an IP address.

Returns a struct hostent containing the server’s IP information. If host is invalid, the program exits with an error.

1. connect: Establishes a connection to server using the initialized serv\_addr.

If connection fails, the program exits with error.

1. Send a message: prompts the user to input a message. fgets: Reads up to 255 characters from stdin into buffer. Strcspn: removes the trailing newline added by fgets. write: sends the message to the server through the socket.

Why we use this client program?

For testing the server, establish TCP communication ensuring reliable, ordered, and error-checked delivery, reusable design i.e. the program can connect to any server by changing the hostname and port, making it different testing scenarios, error handling, learning and debugging. The client is critical for validating and interacting with the server in socket programming projects.

**Enhanced code for handling multiple clients and doesn’t die after the process is terminated:**We will use fork() off a new process to handle each new connection.

Following changes were made in the server code to make the required enhancements:

* 1. Put the accept statement and the following code in an infinite loop.
  2. After a connection is established, call fork()#### to create a new process.
  3. The child process will close sockfd#### and call #dostuff#####, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by dostuff()#### returning, this process simply exits.
  4. The parent process closes newsockfd####. Because all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection

The code which has been provided to us is:

while (1){

newsockfd = accept(sockfd,

(struct sockaddr \*) &cli\_addr, &clilen);

if (newsockfd < 0)

error("ERROR on accept");

pid = fork();

if (pid < 0)

error("ERROR on fork");

if (pid == 0){

close(sockfd);

dostuff(newsockfd);

exit(0);}

else

close(newsockfd);

} /\* end of while \*/

Below is the snippet showing that the code is working fine for the forking methodology.

A screenshot of a computer

Description automatically generated

*Zombie problem:* The above output has a zombie problem. Zombie problem means that if the parent runs for a long time and accepts many connections, each of these connections will create a zombie when the connection is terminated. A zombie is a process which has terminated but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a wait and would want information about the death of the child. Zombies clog up the process table in the kernel, and so they should be prevented.

Unfortunately, the code which prevents zombies is not consistent across different architectures. When a child dies, it sends a SIGCHLD signal to its parent. On systems such as AIX, the following code in main() is all that is needed.

signal(SIGCHLD,SIG\_IGN);

This says to ignore the SIGCHLD signal. However, on systems running SunOS, you have to use the following code:

void \*SigCatcher(int n)

{

wait3(NULL,WNOHANG,NULL);

}

...

int main()

{

...

signal(SIGCHLD,SigCatcher);

...

The function SigCatcher() will be called whenever the parent receives a SIGCHLD signal (i.e. whenever a child dies). This will in turn call wait3 which will receive the signal.

In the below-mentioned snippet, you can see that the server is listening on port number 52897. The client’s message is displayed accordingly. Then the server is again goes in the accept state and listens for another client’s message. As you can see the server has not terminated the process.

A screenshot of a computer

Description automatically generated

1. **User-Datagram Protocol Sockets**

UDP sockets can be compiled and run in exactly the same way as the server and client using the stream socket.

The major differences are depicted below in the form of code snippets:

Changes in the server.c code to form server\_udp.c:

1. ‘clilen’ is replaced with ‘formlen’ & cli\_addr is replaced with ‘from\_addr’.

A screen shot of a computer code

Description automatically generated

1. The following parts of server.c socket have been removed to make it usable for UDP protocol.
   1. accept(), connect(), listen(), read(), write() have been removed to make it usable for UDP.
   2. bind() is NOT removed.

A computer screen with text and images

Description automatically generated

* 1. SOCK\_STREAM was replaced with SOCK\_DGRAM

A computer screen with text

Description automatically generated

Changes in the client.c code to form client\_udp.c:

1. The same changes were made in the client.c  
   A screen shot of a computer program

   Description automatically generated
2. **Sockets in the Unix Domain**

A UNIX Domain server facilitates inter-process communication (IPC) on the same system using UNIX domain sockets. Unlike Internet domain sockets that rely on IP and port numbers, UNIX domain sockets use the file system for addressing. This ensures fast and reliable communication between processes running on the same machine.

Server\_UNIX: This is the server in UNIX domain source code in c.

A screen shot of a computer program

Description automatically generated

Client\_UNIX:

This is the client\_UNIX source code in c.

A screen shot of a computer program

Description automatically generated

**Output with executable:**

**A screenshot of a computer

Description automatically generated**

1. **Github Time-stamps with History**

Please find below the complete github history with time stamps to my repository for Lab-1.

**A screenshot of a computer

Description automatically generated**