Network Services and Applications

Take Home Assignment

Introduction

This assignment was to write a client which interacts with a server running on a remote machine using the sockets interface. The server waits for a TCP connection on the port number 48031. When the connection is established by a client, the server then waits for the client to begin communicating.

Client Socket

Setup

I created a client socket that takes in 4 arguments, the program name, the address of the server, the request string, and an optional port number. The port has a default value of 48031, with the option to also enter a port number from the command line, like the sockets we had studied in the labs.

A TCP protocol socket is used to connect with the server. An address structure of type sockaddr_in is used to store the servers address details, including address family, IP, and port number. Connect() is then called to make a connection with the server.

Getsockname()

To get the client IP Address and Port another address structure is required. I then used getsockname() to get the clients IP Address and Port number. I then used sprint() to append the address details to the request string. The request string is created in the client program unlike the labs where the string to echo was entered when running the client.

Socket Communication

The client sends the request string that was input when running the client. The server then responds with a greeting message or error message, depending on if the request string was formatted correctly.

I then use the first word of the greeting/error to decide if the client needs to carry out further sends/recvs. The server closes its connection with the client socket if the request string isn't correctly formatted.

If the request string is correctly formatted, the client prints the number of bytes received in the greeting, and sends the number of bytes received back to the server. It then receives a random number of bytes from the request string back from the server. It sends this number of bytes back to the server, to use as a comparison for the outcome.

I calculated the random bytes received by the client without the "\r\n" by using the strstr() function to locate the location of the end of line terminator in the buffer. After this number is returned to the server, I used the shutdown function with SHUT_WR to stop the client from sending. The client

can still receive the outcome string, followed by a randomly generated cookie, of 9 decimal digits from the server. The socket then closes.

Server Socket

Setup

The server takes an optional port number as an argument when running, or else it takes the default port number 48031, to listen for connections. A structure stores the local address, before the server binds. The server IP Address and Port are obtained with getsockname(). The server then listens for client connections.

Loop

An infinite loop is then used to handle client connections. Another structure is used to hold the client address, and the server waits to accept a connection. When the client connects, the Server displays the Client IP Address and Port, and immediately receives the request string. The request string can be sent from client to server in a couple of different ways. I used send() and recv() for my communication.

Parsing

Framing is used to format a message so it can be parsed on the receiving side, so the beginning and end of the message can be found, along with any field boundaries. The request string uses spaces to indicate the boundaries, and \r\n to indicate the end of the message. I used strtok() to separate the request string into the different fields separated by a space delimiter, storing the fields in variables. The fields can then be compared, such as comparing the requested address to the client or servers address.

When parsing all the required fields are delimited with a space except for reqStr5 the port number which is delimited by the end of line marker, I also stored any random data at the end in a separate variable, to stop it printing to screen.

In chapter 3.4 of the Pocket Guide book it suggests using a struct to send the information from the client to the server, with the server knowing the correct format the information should arrive in, and reading the data into a similar struct, with the correct size of the data. I tried to implement this, but with the amount of errors I was getting, there wasn't enough time.

After displaying the request string and the number of bytes in it, I copied the request string from the buffer to a separate file to split using the " " space, "-" hyphen, and "\r\n" end-of-line terminator as delimiters with the strtok() function. I creatively named them reqStr1, reqStr2 etc., and gibberish to store any random characters picked up along the way, that were causing problems with the output.

Decide Which Greeting To Send

I then used strcmp() to compare the parts of the request string with values stored on the server side, to match up to the components of the request string. The request string address and port can be compared to the client address and port obtained when binding. The other strings I hard coded

in. If they all match the individual components of the request string are displayed, and a positive greeting is set, before proceeding with the server code.

If the string is incorrectly formatted, an error message is returned to the client, the stored values are shown opposite the values received. Obviously, this is of no use to the client, but I had too many if statements to return separate messages for each possible error.

The integer value proceed flag is also set to 1 to prevent the server from trying to send and receive any more data after sending the greeting. The rest of the code is then skipped over and the client is shutdown, while the server remains open for new connections.

Random Number of Bytes

If a positive greeting is sent, the server receives the number of bytes sent back from the client. It then sends a random number of bytes. For the random number of bytes, I generated a number between 1 and 50, and used this to send back this many bytes of the request string to the client. The client then sends back the number of random bytes received.

Cookie

The server generates a cookie, another random number this time % 1,000,000,000, to get up to 9 random digits.

Decide Outcome

If the number of bytes sent back from the client matches up to the random number generated earlier in the code. The server returns an OK message with the cookie, otherwise it returns an error message and a cookie.

Send The Greeting

Because sending the random bytes was causing some problems, I used a dummy send and recv in between to return the number of bytes sent for the greeting message, before sending the random bytes.

Sending - first character check

To check that the first character of a line to be sent wasn't whitespace, I used strncmp() to compare the first character of each string to be sent which worked fine, before using the isspace() function in ctype.h, to check the first character of the string.

Conclusion

With the help and guidance of the examples in the books "The Pocket Guide to TCP/IP Sockets, C Version", and "TCP/IP Sockets in C, Practical Guide For Programmers", both written by Donahoo, Calvert, I learned how to use C sockets to communicate sending and receiving data to each other.

The hardest part of beginning the assignment was finding information on the getsockname() function, with the information I could get online, I was lead to believe the client would need to bind to a socket in order to get the local IP Address and Port Number. This along with trying to create a second socket for the client to bind to, to get the address information. After 2 days off messing with different combinations, I realised that it didn't require a separate a socket to be created, only its own address structure to hold the address information.

Another difficulty I ran into was appending the information to send as a string from the client to the server after getting the information from getsockname(). Sending the string, IP address, and port as using separate sends worked fine, but the format of the string was not entirely correct. It took a lot of messing with different functions such as strcpy(), and strcat() before I got the sprintf() function to correctly format the request string. Some of these mistakes were hard to find as the client compiled as normal, and the errors weren't noticed until running the program.

Putting 2 recv()s in a row to align with to send()s caused the program to repeat the first send, and not carry out the 2nd send. I got around this by adding a send for the value of the string received to return to the server, before recv was used again. I could then use the returned value for the first string sent from server to client, to add to a return for the second string of data, to get the total number of bytes sent by the server

Another mistake I made was using sizeof() in place of strlen() when calculating the size of the string to be sent. The client compiled and worked, but it caused an error on the server side for receiving the string.

I also tried using a struct to hold a set of variables to make up the request string, one struct on the client side, and a matching struct at the server side to read in the data, but this gave too many errors.

When the server parses the request string and checks that it is formatted correctly, It didn't seem right to continue sending any data, so I used if statements, a decision flag in the server, and the first word of the greeting/error in the client to skip over unnecessary code with additional send and receives.

I also had to add back in the code so the request string could be entered from the keyboard, so the request string could be quickly checked for errors when the program is run, by entering a broken request string.

I have included the client and server code in separate files, and screenshots of the program running.