ECEN 602

Computer Networks

Assignment #01

Team No: 1

Sushrut Kaul

Ishan Tyagi

Ishan did the Client side code and Sushrut did the Server side code.

The package contains 5 files:

1. **echo.c**
2. **echos.c**
3. **README.txt**
4. **Makefile**
5. **Test Cases.pdf**

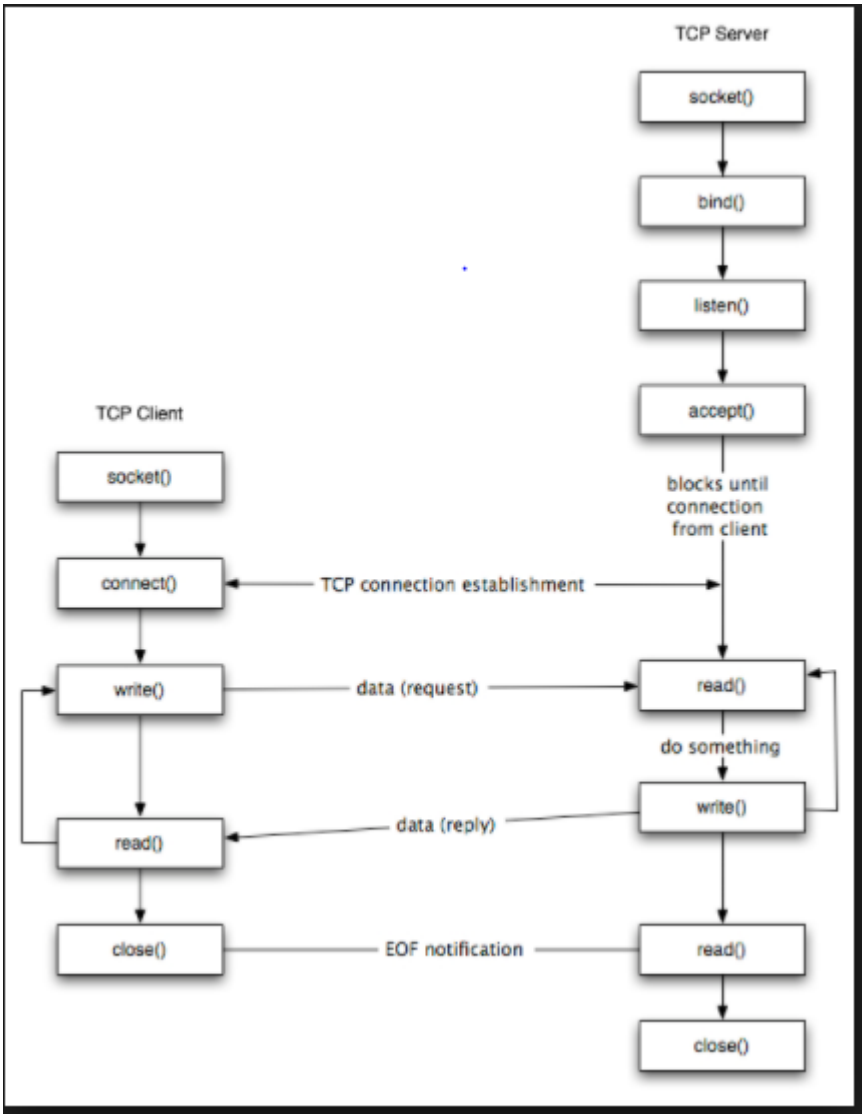
To compile the code, run the Makefile: make

Run the server by using the command line:/echos port

Run the client by using the command line:/echo localhost port

To clean the executables run: make clean

ARCHITECTURE FOLLOWED:



TCP SOCKET PROGRAMMING ARCHITECTURE

SERVER:

1) **Getaddrinfo ()** ----- This method gives us a linked-list of results. We can iterate across that linked list which contains a list of **(sock addr) structures**. We iterate until we receive a usable structure and then break out of the loop. After we get the structure, we can free the linked-list.

2) **Socket ()** ---- This function takes three parameters. The first one is ***domain*** which can be **AF_INET**, **AF_INET6** or we can leave it as **AF_UNSPEC**. We follow the third approach for our design. This is because it gives us the flexibility to accommodate both IPV4 and IPV6 addresses. We also need to

pass the socket type as an argument. Here we are focusing on TCP. So, we take socket type as **STREAM_SOCKET**. The third parameter is protocol which we keep as zero for our purposes.

3) **bind ()** **---**Bind helps us associate a port number with the socket on the machine. bind() takes three parameters. The first is the socket descriptor, the second is a pointer to a struct addr structure and the third is address length. In our program, we use one of the structures in the linked-list returned by getaddrinfo () to pass the parameters to this bind () function dynamically.

4) **listen ()—**The server then listens for incoming connections from clients. We allow multiple clients to connect to our server. ***Backlog*** is usually the number of clients that can connect to a specific server at a given time. We have set this as **10** in our example. After 10 clients have connected to the server, the next client trying to connect gets a connection timed out message.

5) **accept ()—**The server has to accept the client's connection request for communication to happen. We use **fork**() command to create child processes. In our code, the child process is getting serviced in the while () loop whereas the parent process is blocked at accept () waiting for new client connections.

6) **Read () –** The read function will try to read upto MAXDATASIZE-1 characters from the socket descriptor. If we read an END-OF-FILE from the standard input, we close the client socket. When the client socket exits, a zero is returned by the server read function and then we close the child process.

7) **Writen () -** The conventional write () function does not give us a guarantee that n bytes will be written. This is actually not an error. We need to have something more reliable. For this we write our own writen () function which guarantees to send n bytes.

8) **Close () ----** After client and server are done communicating, we close the socket and wait for new connections.

CLIENT:

1) **Socket ()** – The socket function remains the same here. Socket () returns an integer specifying the client file descriptor.

2) **connect ()** – Next, the client program uses the **connect()** function to communicate with a given destination address and a specified port. All the information required by the connect function can be obtained by using the results of the getaddrinfo () call.

3) **fgets()** -- This function is used to read from the standard input . If we enter an END-OF-FILE (Cntrl+D), then this function returns null if the end of file is not appended with some text. If the **END-OF-FILE** is appended by some text, we use another function feof (). This function returns true on the **END-OF-FILE**.

4) **Writen ()** – same as above.

5) **my_read ()** – The my_read function reads one character at a time from a static buffer that is maintained by our code. This function is being called by the readline () function.

6) **Readline ()** – This function calls the my_read function. It returns the number of bytes read from the socket. The function has following terminating conditions:

   a) **1 byte received** – If one byte is received, the function uses a pointer to store the character read from my_read in a character array and then updates the pointer. If a newline is encountered, we break out of the while ().

   b) **0 byte received**-- If we receive 0 bytes, we put the null string at the end and return the number of bytes read.

   c) **Negative return**-- Negative return value implies an error.

7) **Close ()** – After the client and server have finished communicating, we close () the socket descriptor.

Error Handling: We have taken several measures to enforce error handling in our code. These are as follows:

1) **err-sys function – The err_sys function is used to** print **easily understandable error messages**. This makes debugging the code easier. Err_sys function makes use of **perror** to print messages.

2) Errno variable is set to an appropriate value when an error occurs. We use this value to identify the error messages.

3) **EINTR** is one specific error that has been specifically handled in this assignment. **EINTR** requires us to call read again. We do that using a **goto label**.