ECE 373 – Socket Example

This code is an example use of sockets for supporting network communication between a client and server.

The server is the common program that all clients connect to over the network. The client is what you execute on your own machine to send messages to a central server, e.g., which may then forward them to other clients that are already connected to the server.

The server may execute anywhere, but its IP address must be "routable" to all clients. In other words, the server must have a “public” IP address. The server acts as a proxy between different clients. That is, it may receive messages from one client and then relay them to other clients. Thus, the server enables communication between clients that do not have "routable" or “public” IP addresses, i.e., they may be behind a home router with IP addresses with prefixes such as 192.168.*.* or 10.0.*.*.

Within this directory, I have provided sample code for a simple client and server that uses sockets written in C++. The sample client sends one message to the server, which receives it, prints it out, and sends a reply message back to the client, which the client prints out. The client exits after printing out the message, while the server stays active, continuously listening for new connections from new clients.

The code includes comments that explain what it is doing at each specific step. To summarize, the server creates a network socket data structure to “listen” on. The socket is a “SOCK_STREAM” type which communicates reliably over the unreliable Internet using TCP. The call to **socket()** returns a socket descriptor, which is similar to a file descriptor. After the server creates a socket it must call **bind()** to “bind” the socket to a particular IP address. Servers may have many IP addresses attached to different interfaces. For example, your laptop’s wireless network connection has a different IP address than its wired connection. In addition to binding the socket to an IP address on the server, you also specify a port number. The port number enables multiple server applications to communicate using a single IP address. Clients specify an IP address and a port when connecting with the server, where different ports connect to sockets offered by different applications. For example, web servers use port 80, so web browsers always connect to port 80; ssh uses port 22, so ssh clients always connect to port 22.

After binding the socket to an IP address and port, the server then calls **listen()** to “listen” for new connections made to the socket. After calling **listen()**, the server accepts new connections by calling the **accept()** function. The **accept()** function returns a new socket descriptor that is separate from the server’s listening socket that you set up above. This socket descriptor is specific to that connection between

the client that connected to the server and the server. To send and receive data from the client, the server can call the **send()** and **recv()** system calls.

Note that **send()** and **recv()** both return the number of bytes they actually sent, which may be less than the number of bytes you tried to send when calling them. This happens because the kernel buffer for storing bytes that haven't been sent may be full, and cannot accept new bytes. As a result, you should always check these return values, and if all the bytes you tried to send were not sent, you should try to re-send them. When calling **recv()** you simply receive bytes in order from the client; if these bytes are strings, then they may not end with an end of string character '\0'. You also should define your own protocol to know when an entire message has been received when using sockets. For example, you might put a sequence of special characters at the end of each message to tell the server that the message is finished (e.g., you could have the end of each message be the sequence of characters DONE). That way, the server knows that if it hasn't received an entire message, it should remain in a loop attempting to receive the entire message.

One important thing to note about the server, is that the call to **accept()** is "blocking". It will not return until a client has connected with the server.

As a result, servers that must support 2-way communication between any number of users must use multiple **threads**, with one thread per user. Otherwise, once you called **accept(),** you would be stuck waiting on the next client to connect, while other clients might be trying to send messages. Thus, servers are typically multi-threaded, where they sit in a loop waiting to accept connections from clients. Once a client has connected, they spawn a thread for that client that sits in its own loop and receives messages from that specific client. The server maintains a table of client names to the sockets that correspond to them the connection with that client.

That way, the server can both wait for new connections from client's to accept and be able to send and receive messages from clients at the same time. Upon receiving a message from one client, the server can then forward it to a destination client using the table above: after the server receives the message, it simply looks up the socket descriptor for the destination in the table and sends the message to the proper destination using the **send()** function. When a client quits, it should send a final message informing the server that it has quit, so the server can exit the thread associated with that client and remove the client's socket descriptor from the table. The server should also close the socket, similar to how you close a file descriptor.

The client is much simpler than the server. The client need only be a single thread, and does not need to be multi-threaded. The client does not "listen" for connections on its socket. Instead, it creates a socket, also using the **socket()** function, where the IP address associated with the socket is the IP address the client wishes to connect to. In the sample code, the IP address is 127.0.0.1, since that is the default IP address for the host that you are running on. The client also needs to specify the same port that the server is listening on. In the sample code, this is port 5000. All ports less

than 1000 are only accessible by the root user, but you can use any port greater than 1000. After creating the socket and filling out the server_addr data structure (with the port and the server's IP address), you call **connect()** to connect to the server. If connect does not fail, i.e., doesn't return -1, then you are connected to the server.

The **connect()** function call does the three-way TCP handshake to establish a connection. At this point in the client, you can simply use **send()** and **recv()** to send and receive data. Your client program should read each line of characters input by the user (as described below), and then send those to the server.

While conceptually simple, there are lots of places where you can make mistakes in developing clients and servers with sockets. Check the man pages for each of these calls, and make sure to check return values to ensure the calls didn't fail (the functions generally return -1 if they fail). You should also remember to check the return values of **send()** and **recv()** as you cannot assume that the number of bytes you wanted to send or receive were actually sent and received.