CS 371 – Second Project

Deadlines: April 18th First report, April 27th Second report

Design of a Small Online Chat

Maximum points 100

General information

Maximum group size 3.

The project first project report should be submitted by email to the TA by April 18th. The entire package including first and second reports, and the code, should be submitted on Canvas as a single Zip file by April 27th.

The project should be discussed with the TA within 1 week from the deadline of the second report, the suggested dates are between May 2nd and May 4th. All grades should be finalized and uploaded by May 7th. Contact the TA to set up an appointment. The TA will run your program to make sure it works.

Small Online Chat

Online chats are very common software that allow several users to discuss in real time through the Internet. Common examples are Skype, Microsoft Messenger, etc.

In this project you will design a small chat system. The system includes a Chat Server, that is responsible for accepting connections from Clients and receiving/forwarding messages, and Clients, that connect to the server and send/receive messages.

The project will be developed in C using socket programming and thread.

We will use the library <sys/socket.h> for socket programming and a description can be found here: http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html .

We can use <pthread.h> for thread and a description can be found here: http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html . However, there are many other methods you can use to implement the thread part, for example, you can also choose to use fork() function in <unistd.h> to achieve the same goal.

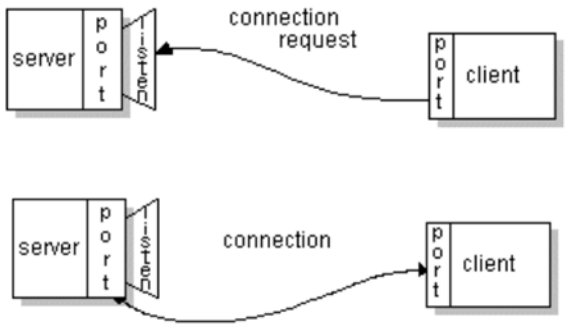
Background on Socket Programming

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. On the client side, the client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to connect to the server using the server's IP address

and port number. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets. The figure below shows the process.



The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data.

The primitives including socket settings, connect, read, write, bind, listen and accept can be found in <sys/socket.h>, which means we don't need to define our own primitives.

Socket Creation: int sockfd = socket(domain, type, protocol). For the first parameter domain, we set it to AF_INET if we use IPv4 protocol, and AF_INET6 if IPv6 protocol. For the second parameter, we set it to SOCK_STREAM, which means we use TCP for reliable connection. For the third parameter, we set it to 0.

Bind: int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen). After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure).

Listen: int listen(int sockfd, int backlog). It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow.

Accept: int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Connect: int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen). The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

Background on Threading Programming

The sample server code above has the limitation that it only handles one connection, and then dies. A "real world" server should run indefinitely and should have the capability of handling a number of simultaneous connections, each in its own process. This is typically done by creating a new process to handle each new connection.

If you use <pthread.h>, then you may need to use pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)) function to create a new thread every time a new client is accepted. For the first parameter, we set it to a thread we want to create. For the second parameter, we set it to NULL. For the third parameter, we set it to a function which we define by ourselves to handle clients. The fourth parameter should be the new accepted client.

Note that we may need to use threads also for the design of the client, in order to be able to send and receive messages, preventing the blocking nature of the receive function not to allow to send messages.

Online chat design

The chat is composed by a Chat Server and a Chat Client, their functionalities are described below.

<u>Chat Server</u>

The Chat Servers is the core of the application. It has three main components: the first component (Server Listener) listens to incoming connections, the second component (Server Connection Thread) is responsible for the connection of an individual client, while the third component (Message Forwarding Thread) receives and forward clients' messages. Since listening to incoming connections is a blocking activity, these components need to run on three separate threads of the same process.

Server Listener

The first component listens to incoming connections on the IP address of the server, on a port that you can choose (do not use well-known ports such as 80). This can be realized with the listen(int sockfd, int backlog) function of the library <sys/socket.h>. Once a connection is accepted, a new thread is created to handle that connection.

Server Connection Thread (SCT)

A connection thread is responsible for the connection of a single client and interacts with the Message Forwarding thread (MFT) of the Server. Intuitively, the connection thread will receive messages from the MFT and forward it to the client, as well as receive messages from the client and forward them to the MFT. The exchange of messages can be realized through shared memory (but be careful of race conditions!) or through message passing, between the threads. An option could be a list of lists, where each element of the list contains the list of messages to be forwarded to a certain client. Semaphores should be used to prevent race conditions.

The SCT is also responsible for closing the connection once the client times out (e.g. after few minutes of inactivity), or explicitly closes the connection.

Message Forwarding Thread (MFT)

The MFT receives messages from the clients and forwards them to the other clients. In this simplified chat we will have one single room, so when a message is received from one client it is forwarded to all the other ones.

Chat Client

The client establishes the connection with the Server, and specifically with the Server Listener, using the IP and port. We can assume that the IP and port are known to the client. Once the connection is established, the client can send messages and receive messages. Each client should ask a nickname before starting the connection, that name will be used in the chat to identify the client. Nicknames do not need to be unique (but you can think about how to ensure unique names if you want to extend the project).

The client can work on a simple terminal, and thus the messages exchanged are text messages input by the user. Messages that are sent should include the nick name and the message.

The client can also close the connection with the server, for example when a specific string is input by the user.

Evaluation

This project does not have a quantitative nature such as the other one. As a result, there will be no graphs to be generated. Nevertheless, you should provide several use cases and corresponding snapshot of your application to show that it works.

Use case 1: Client connects to the server

Show that your client can successfully connect and receive a hello message from the server.

Use case 2: Two clients exchange a message

Consider two clients connected, and exchange a message between them.

Use case 3: Three clients, one logs out

Consider three clients exchanging messages, then one logs out, and show that the other two can still exchange messages.

Use case 4: Two clients, one logs in

Consider two clients exchanging messages, then one logs in, and show that the three can still exchange messages.

Reports

Two reports should be prepared. The first report does not require implementation and should include at least the following sections:

- Abstract: Summary of the following sections
- Introduction and motivation: Overview of Online Chats
- Proposed solutions: Diagram and Pseudo-code explaining the design of the chat and all its components, important data structures, called functions. You should describe how the design issues of the chat are going to be addressed (e.g. threading, shared memory, etc.) both on the client and server side.
- Plan of experiments: Description of the methodology which is planned to use in the experiments, significance and expected results.

The second report requires implementation and describes the implementation and use cases. It should include at least the following sections:

- Abstract: Summary of the following sections and results
- Implementation: Description of the methodology used for implementing the proposed solutions (e.g. relevant classes and data structures, structure of the program, etc.).
- Experiments: Description of the experiments performed and obtained results. Results should be represented through snapshots in a graphical form as well as discussed in the writing.
- Conclusions: Summary of the work and final considerations.