BTN415 Lab 4

Introduction to Multithreading – TCP/IP Communication

In this lab, you will learn how to create applications using multiple sockets running in different threads.

LEARNING OUTCOMES

Upon successful completion of this lab, you will have demonstrated the ability to:

- Create an application spanning multiple threads
- Use multiple sockets to allow applications to send and receive data asynchronously
- Implement a simple chat application

SPECIFICATIONS

In this lab we will create a simple chat system. This system comprises two different applications, a **client** application and a **server** application. The client application creates two sockets, one to send messages to the server and one to receive messages from the server. Conversely, the server application creates two sockets, one to receive the messages sent by clients, and another one to broadcast messages sent by one client to all clients currently connected.

Download/Clone the Visual Studio solution from the class Github and perform the following:

Part A

In the **client.cpp** file, you will need to create a second **winsock_client** object. This object should have port **27001** and IP: "**127.0.0.1**". This second socket should only attempt to connect to the server after the first socket, the one using port 27000, has successfully connected to the server and has received a "**Welcome**" message. After connecting to the server, this second socket should receive one message from the server[1]. Following, you should start a thread that calls the **get_messages()** method for **winsock_client** objects described in what follows.

The **get_messages()** method is nothing but an infinite loop that keeps calling the **receive_message()** method and printing any received messages on the terminal. It takes no arguments and returns no values. Remember that you need to define this method in the **oop_winsock.h** file as well for the **winsock_client** class.

Part B

In the **server.cpp** file, you will need to create a second **winsock_server** object. This object, which will be used to transmit data to clients, should have port **27001** and IP: "**127.0.01**". This transmitting socket

This message is nothing but the "**Welcome**" message that the server always sends after a successful connection. In contrast with the "Welcome" message received by the first socket, which is used to check if we should connect the second socket or not, this one is simply ignored.

should only attempt to accept a connection if the first socket, the one used to received data from clients using port 27000, has successfully accepted a connection and the returned socket number was smaller than **MAX_SOCKETS**.

After accepting both connections, your server should start two threads, one on the transmitting socket calling the method **send_mod()**, and one on the receiving socket calling the method **rec_mode()**. This methods are described in what follows. Note that you should not create threads to call the **echo_mode()** method, as done in the original multithreaded server application.

PART C

The **rec_mode()** method implements an infinite loop that keeps receiving messages from clients, and copy these messages to a global variable called **char message_buffer[128]**. Once a message is copied to **message_buffer**, a boolean global variable called **buffer_full** should be set to true. In case the received message was "quit", the **rec_mode()** method should close the socket and set its **active_sockets** position to 0.

PART D

The **send_mode()** method implements an infinite loop that keeps checking if there is a message in **message_buffer** by checking the boolean variable **buffer_full.** In case there is a message in message_buffer, this method should send this message to all currently connected clients, clear the **message_buffer**, using **memset(message_buffer,0,128)**, and set **buffer_full** to false. In case **message_buffer** contains "quit", this method should close the socket and set its **active_sockets** position to 0.

Take Home

Change your code so that each client starts by sending a **username** to the server. Then, edit your code so that the send_mode() sends the username of the client who originally sent the message prior to displaying the message itself. For example, if user "john.smith" sends "hello". All users should receive: "john.smith: hello".

SUBMISSION REQUIREMENTS

Once you have completed your lab create and upload the following files:

- Create a single ZIP file that contains all your source code files (*.h and *.cpp)
- The output.txt file generated by the lab
- Any additional information you feel necessary for me to mark your lab