

MultiUserEngineering-2-JEvans

by Jonathan Ieuan Evans

FILE	4766_JONATHAN_IEUAN_EVANS_MULTIUSERENGINEERING-2-JEVANS_113614_1005235365.DOCX		
TIME SUBMITTED	10-MAY-2018 06:49PM (UTC+0100)	WORD COUNT	2134
SUBMISSION ID	88897749	CHARACTER COUNT	11714

Chat server and client application

Jonathan Evans

P151713

10/5/18

Contents

Contents	1
1. Introduction.....	2
2. Chat Server.....	3
3. Chat Client	6
4. Additional features.....	8
5. Conclusion	9
6. References.....	10

1. Introduction

For this assignment, a chat server and chat client had to be created. This report will explain the functionalities of the chat server and chat client programs. It will show how these two components were created and how they interact.

The following sections will consist of a section that will evaluate the chat server and another that will evaluate the chat client. Justifications and explanations have been made where necessary.

2. Chat Server

The chat server is the component that is responsible for responding to messages that have been passed to it. As a result, this will allow users of the chat client to connect to the server with a nickname and read and write messages on the server, which will be viewable by other users.

The server runs on a port provided to it from the launch arguments. This allows the user deploying the server to use their own port numbers. This is a desirable feature as the server administrator will get to choose their own ports as apposed to a system where the port is hard-coded (non-changeable) where that port may be required for something else.

After the server has been launched, the following procedure is carried out:

- A socket file descriptor (FD) is created and is bound to a given port. [1] This socket FD represents a value which is the number of the socket. The socket is a communication mechanism that allows from data to be read from it and written to it.

This program will be using Berkeley Sockets (BSD Sockets) which is a socket API created for use with UNIX systems (Windows machines would have to use the Winsock API to interact with a BSD socket).

- Server is set to listen on the socket. At this point, the program enters an infinite loop. This is used to continuously check the socket for messages.

Once a message is received, a thread is created to deal with the request. Introducing threads allow multiple requests to be handled through context switching, creating the illusion of parallelism (unless the thread is set to run in parallel with other threads, which would provide true parallel processing).

The threading library used in this program is known as pthreads (POSIX threads) – this is a threading library for C [2]. C++ 11 introduces support for threading (std::thread [3]) which allows threading to be implemented through a number of classes.

An alternative to creating threads, would be to fork a process or handling the client requests within the for loop itself. There are issues with both approaches. Forking a process introduces over-head as memory spaces are duplicated, resulting in more resources being used. If the client requests are handled within the for loop itself, the program will not be able to address multiple clients at once, blocking other messages from being received.

As threads are lightweight, in that the memory spaces are shared amongst them, the use of them has been opted-for for this program as the server may be required to process many connections seemingly at the same time. [2]

A thread requires a function to be passed to it along with the arguments. The function that is passed to the thread in this program is the clientHandler function which passes the client socket as an argument.

Client Handler function

The client handler function consists of recovering the socket provided to it, reading from the given socket, and using XDR functions to decode the type of message that is on the socket and encoding responses to send back to a client.

XDR (External Data Representation) is a binary format used for serializing and deserializing messages sent across a network [3]. The process of serialization/deserialization in this program

consists of sending data from a class from machine to another, using methods to encode and decode the streams.

An alternative to XDR could be XML (Extensive Mark-up language). The problem with sending this over the network is that it will send text over the network as opposed to binary. The use of XDR may result in more responsive delivery of packets as the data is in binary.

On the server, once a message is received it is decoded firstly as a message and a function is used to determine the kind of message it is. Pending on the result, the message will be decoded into a given type of message (which are listed below). Following this is a response that is sent back to client pending on the kind of message that has been received on the server.

There are four kinds of messages: JOIN, LEAVE, MESSAGE, UPDATE, each have their own handler function associated with them that are detailed below.

joinHandler

The joinHandler reads the nickname provided to it and returns a true or false response pending on if the nickname is present or not in the user list vector. To do this, a for loop checks each username in the user vector if the name exists or not. If it does not, the name is pushed to the back of the vector.

The complexity of this function is linear $O(n)$ as a username needs to be compared against other usernames in the user list vector until a match is found or not. The speed in which this is completed depends mostly on the number of users within the vector.

leaveHandler

The leaveHandler works similarly to the joinHandler, except a name is erased from the vector, if it is found.

The complexity for this function is quadratic $O(n^2)$, that is because the process consists of a nested for loop. One for checking if the name exists, and the other which is called with the erase function, which is a linear function as it removes the element at the location specified and resizes the vector after it the element has been removed [4].

messageHandler

The messageHandler function takes a message provided to it and pushes it to the back of a messages vector, which is done in constant time $O(1)$.

A Mutual Exclusion (Mutex) lock is used here. This is an atomic lock that is used to lock a variable (in this case the message list vector) to prevent other threads from accessing it. The code within this lock is also known as the critical section, this is code that is not parallelizable [5].

Without a mutex, a race condition could occur – this is a problem where a thread could access a variable at a point in time that may make the data inconsistent as another thread may be ready to insert data that is out of date, thus a synchronisation method such as a Mutex is needed.

UpdateHandler

The updateHandler provides the client with an updated list of messages. On the client-side of the application, an update request is sent every second to retrieve new updates. This will allow users connected to the server to see a list of new messages the second the update is sent to the user.

A mutex is also used here to ensure that the currentMessageNo variable can be updated by only one thread at a time.

After any of the handlers shown above are triggered, a given message is sent back to the client and the socket and thread are closed and any dynamically allocated memory is freed.

3. Chat Client

The chat client is the user-facing component of the application. With this, the user will be presented with a QT interface that will consist of a join, leave, and send button, along with text boxes that will be used to specify the server address, port, and desired nickname.

By creating this application with Qt, the application can be distributed freely if it is not being distributed for commercial use – which would require a commercial license [6]. The QT library itself introduces several interface components which can be compiled for many platforms – Windows, Linux, Mac, BSD, and many mobile platforms.

Main

The main function of the program calls the function to display the QT Application. This draws the interface to the screen in a window. This call is a blocking call and as a result, nothing can be done in the command line while the program is running.

Chat client

The main of the program runs the chat client. As a result, most of the interesting code is stored in the chat client implementation, which contains functionalities associated with the user interface.

Join

The join button is responsible for retrieving the text entered in the IP, port, and nickname box, and creating a join message that will be sent to the server.

This is done by creating and connecting a socket, encoding the message to an XDR stream, and writing the given XDR bytes to the socket.

Once the message is sent to a socket, the server would decode the message and send a response. At this stage, the client would read the response (which will tell the client if they have been accepted to join the server or not) and enable/disable user interface elements – such as disabling the join button as the connection has already been established at this point.

It is possible that a username already exists, thus a false response is returned – in this scenario, the message “Rejected!” will be displayed in the nickname box.

Important note: This should be a functionality that should be changed in a later revision as the user may accidentally join the server as “Rejected!”.

Leave

The leave button is responsible for leaving server. The process is like that of joining the server, apart from this will disconnect the user from the server. After the user has left the server, all user interface elements will be reverted to the state they were in when the program was launched.

Send

This sends the message in the text box and clears what was entered in the box for more things to be entered.

getMessages

This function sends a request to the server every second, which is done through the QT timer signal. Every second this will request an update on the messages that have been entered. As a result, as soon as the messages list has been updated, new messages will be updated in the text box of the clients who are connected.

4. Additional features

Two extra features have been added to this program to help with usability.

The first feature was an implementation of consistently formatted print messages in the terminal. As a result, it is easy to read when someone is joining the server, leaving the server, or some other event that requires a message to be printed.

A screenshot of a terminal window titled "ChatServer_skeleton: a.out — Konsole". The terminal shows the execution of a program with various status messages. The messages are formatted with color-coded prefixes: green for status, red for errors, and blue for user actions. The output includes: a green prompt, a green status message "IRC Server is running on port: 50019...", a green "Current users:" header, a blue join message "[+] jon has joined the server", a green "Current users:" header, a red error message "[!] Another jon tried to join. Denying request", a green "Current users:" header, a blue join message "[+] jon2 has joined the server", a blue leave message "[+] jon2 has left the server", and a blue leave message "[+] jon has left the server".

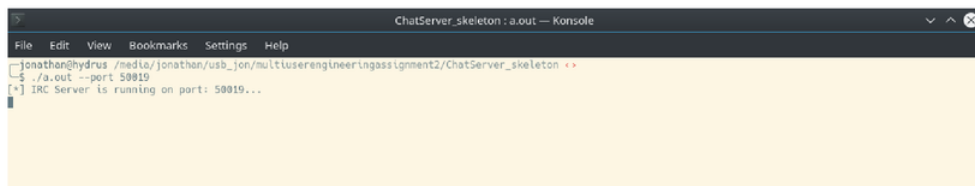
```
jonathan@hydrus /media/jonathan/usb_jon/multiuserengineeringassignment2/ChatServer_skeleton <>
$ ./a.out --port 50019
[+] IRC Server is running on port: 50019...
Current users:
[+] jon has joined the server
Current users:
[!] Another jon tried to join. Denying request
Current users:
[+] jon2 has joined the server
[+] jon2 has left the server
[+] jon has left the server
```

Figure 1 Picture showing consistent terminal messages

The second useful feature added was the incorporation of a command line flag that allows the user to specify the port they would like to use.

Alternatively, it would be convenient if a port number can be specified in a config file (which is not yet implemented).

Being able to change the port on each run from the command line flag was useful during development as the port that a server was running on can be changed if the socket has not been closed properly if the program closes unexpectedly.

A screenshot of a terminal window titled "ChatServer_skeleton: a.out — Konsole". The terminal shows the execution of a program with a status message. The message is formatted with a green prefix. The output includes: a green prompt, a green status message "IRC Server is running on port: 50019...", and a green "Current users:" header.

```
jonathan@hydrus /media/jonathan/usb_jon/multiuserengineeringassignment2/ChatServer_skeleton <>
$ ./a.out --port 50019
[+] IRC Server is running on port: 50019...
Current users:
```

Figure 2 Picture showing the chat client running on a specified port

Future features

Two extra features that could be implemented in a future revision could be the incorporation of a user interface element to show the connected users as this is a common feature in IRC clients.

Another feature would be to add another type of message – maybe an administration message to broadcast important server warnings to the user.

5. Conclusion

For this project, a chat client and server application were created and the functionalities of them have been documented. The incorporation of threading into such an application has been beneficial. Without threading, the program will not be able to accept multiple messages.

As a result, the final program is almost representational of what an IRC (Internet Relay Chat) client – the only differences are that IRC clients typically have a list of users that can be seen.

6. References

- [1] man7, “Linux Programmer's Manual - SOCKET(2),” [Online]. Available: <http://man7.org/linux/man-pages/man2/socket.2.html>. [Accessed 10 May 2018].
- [2] Blaise Barney, Lawrence Livermore National Laboratory, “What are Pthreads?,” [Online]. Available: <https://computing.llnl.gov/tutorials/pthreads/#Pthread>. [Accessed 10 May 2018].
- [3] cppreference.com, “std::thread,” [Online]. Available: <http://en.cppreference.com/w/cpp/thread/thread>. [Accessed 10 May 2018].
- [4] P. B. G. G. Abraham Silberschatz, “Threads,” in *Operating System Concepts, 9th Ed*, Wiley, 2013, pp. 163-164.
- [5] Network Working Group, “XDR: External Data Representation Standard,” May 2006. [Online]. Available: <https://tools.ietf.org/html/rfc4506#section-1>. [Accessed 10 May 2018].
- [6] cplusplus.com, “std::vector::erase,” [Online]. Available: <http://www.cplusplus.com/reference/vector/vector/erase/>. [Accessed 10 May 2018].
- [7] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Process Synchronization,” in *Operating System Concepts, 9th Ed*, Wiley, 2013, pp. 203-207.
- [8] Qt, “Licensing Comparison,” [Online]. Available: <https://www1.qt.io/licensing-comparison/>. [Accessed 10 May 2018].

FINAL GRADE

GENERAL COMMENTS

Instructor

91 /100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

GRADING FORM: SMU GENERIC_GJ

JONATHAN IEUAN EVANS

91

RESEARCH-INFORMED LITERATURE



An excellent report, Jonathan. Well-written and reference, really well done.

4.5

KNOWLEDGE AND UNDERSTANDING

- | | | |
|---|---|----------|
|  | Excellent evidence of knowledge and understanding - you clearly have a robust grasp of this topic, displayed clearly throughout the report. | 9 |
|---|---|----------|

ANALYSIS

- | | | |
|---|--|----------|
|  | Good evidence of synthesis and valid conclusions drawn. Comparison to IRC is apt - thinking of that was the basis for this assignment. | 8 |
|---|--|----------|

PRACTICAL APPLICATION AND DEP.

- | | | |
|---|---|-----------|
|  | A complete implementation, with two valid additional features. Really excellent work, Jonathan - you've obviously worked really hard on this, and it's paid off in the marks. | 65 |
|---|---|-----------|

SKILLS FOR PROFESSIONAL PRAC.

- | | | |
|---|--|------------|
|  | A very well-written, well thought out and complete report. | 4.5 |
|---|--|------------|