Multichatting Program

Helmer Njaerheim

September 2016

1 PLEASE READ!

The main purpose of this document is to present this client server architecture for learning purposes, since I could not find many properly documented open-source client server implementations out there.

It is open for everyone to download and use and most important of all, to improve. It should be noted that this is not a complete version (there are features not implemented yet and notable bugs), and should in no circumstances be used in professional software.

2 Introduction

This document describes an architecture for implementing a general client server network. The main idea is having a server that can properly handle communication between a general number of clients.

In its current state, it works like a text-based group chat program. When a client sends a message the server makes sure that all other clients get that message and that they know which client sent the message.
This basis makes it appropriate for all kinds of software that is using network communication, since a specific message can be treated as a codeword that executes a certain action. I do not have much knowledge about how most networks in industries are implemented, but this is a very intuitive approach.

The next section describes relevant background knowledge that are vital for understanding this implementation. Afterwards, the actual architecture will be described, and in the end, the current state will be discussed.

3 Background Knowledge

This section describes different concepts that are used in this implementation.

3.1 Threads

In a traditional deterministic program, the code executes from top to bottom. When the value of a variable is set to something, it is guaranteed that the next time that variable is referenced in the code, it will be equals to that value, as described in figure 1. A thread introduces concurrency into the program.

```
int a = 0;

doSomething(); // Does not assign a to anything else

std::cout << a;

// a is guaranteed to still be 0
```

Figure 1: Deterministic program in C++

That is, it gives us the illusion that several lines of code can be executed at the same time. (Technically this does not happen on a lower level, but in a high-level programming language like C++, we treat the code like it is). Figure 2 shows a scenario where threads make a program non-deterministic. There is no guarantee for what the result is.
The reason for this is that it's possible that one thread starts accessing the variable a, while another thread is about to update its value. Then a is only incremented once, even though it should be incremented twice. The reason we need threads to implement a client server network, is because a client can be sending a message at any time. The server must be ready to receive a message from any client at any time. That means it must listen to each client at the same time.
This is not possible to do without threads, because a non-deterministic program cannot do several things at the same time.

3.1.1 Mutual Exclusions

The introduction of threads can lead to problems situations like the one in figure 2. One way to guarantee that a part of the code is not run by several threads at the same time, is by using mutual exclusions, usually shortened to mutex. Certain parts of the code where its critical that only one thread is allowed to run at any time is called a "critical section".

When a thread enters this section, the section is locked, so that no other thread can enter it before this thread is finished with it, and the section is unlocked. In figure 3, a mutex has been added to tackle this problem.

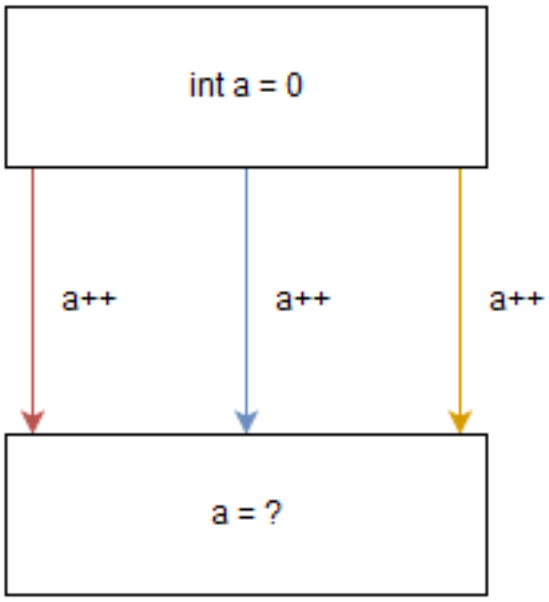

Figure 2: Three threads executing in parallel where each thread increments a by one. The value of a in the end can sometimes be 1, sometimes 2 or sometimes 3

3.2 Shared Variables

Shared memory is a piece of memory that is referenced by more than one process (in this case, threads). That means if one thread change the value of a shared variable, then all references afterwards to that variable in all threads will yield the new value.

This is essential for this approach. For example, when the server receives a message from a client it needs some way to notify other threads so that they can send the message to their respective clients. This is done by changing a variable that all threads can reference. Hence, thread communication is done through shared variables.

3.3 Sockets

Sockets are necessary to set up the connection and send information between the clients and the server in the first place. A socket receives messages from other sockets, and sends messages to other sockets. There are two protocols that most sockets follow: TCP and UDP.
Sockets using TCP first establishes a connection between them, preparing for message transfer between them. It guarantees that messages sent from one socket will arrive to another socket in the exact same order, and that they will reach their destination at all.

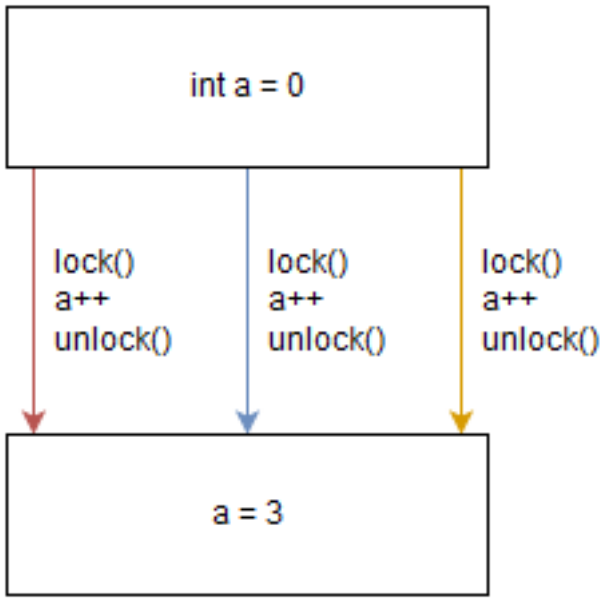

Figure 3: Three threads executing in parallel where each thread increments a by one. A mutex locks down the critical section, so that it is guaranteed that all three will run properly and a will result in 3

If two messages are sent, "Hello" first and then "World", "Hello" will arrive first and "World" second.

Sockets using UDP are not worrying about establishing a connection. They simply send a message and maybe the message will arrive at its destination in whatever order information arrive. The scenario just mentioned could result in "World" first and then "Hello". They might not even arrive at all.

UDP are typically used in multimedia applications such as video games and live-streaming, since lost data just turns into a minor glitch and if the order are wrong at some places it is not noticable to the user.
TCP are used in for example e-mails or chatting, since it is critical that the same information that arrive is the same as the one which was sent.

For this multi-chatting application, TCP is absolutely necessary, since it's all about getting the correct information in the right order through the network, and to not lose any of it.

4 Architecture and Implementation

This section describes how the system is put together.

4.1 Overall Architecture

The participants in this application are an arbitrary number of clients which is below some maximum limit, and one server. The clients will be able to talk to each other through a connection with the server. All communication will go through the server, so the server has full control of handling messages from clients and sending messages to clients.

Each client has one socket that handles incoming messages and where it sends out messages. The server has one socket that is connected to the client sockets. Figure 4 describes the architecture of the entire system.

The reason that the server has one socket per client additional to the server socket, is because the server program is referencing to each client socket. Each client has its own socket, and the server has one corresponding socket variable for each, that it connects to its own server socket. So the actual socket is set up in the client, while the server can reference it.
Hence, the server is described as having an array of sockets and its server socket connected to them.

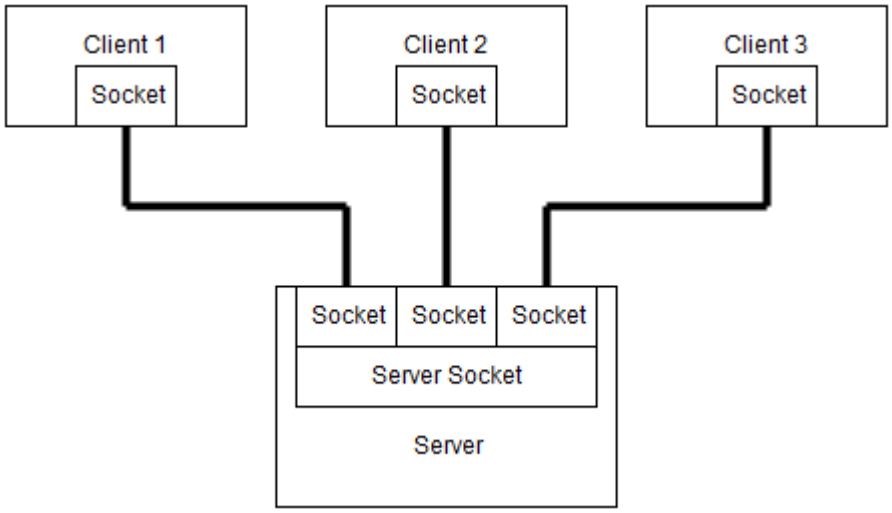

Figure 4: Overall client server architecture for three clients

4.2 Client

The client is not that complicated. In this application it can only send and receive messages. Two things are performed in the client:

- It receives a message from the server and prints it out.
- The user types a message, and the client sends it to the server.

However, these two actions happens asynchronously. A client can receive many messages and not send a single one in a specific time interval. The opposite can also happen. This is why the client needs threads, to handle scenarios like this.

One way to implement this is to create two threads, additionally to the main thread. Figure 5 shows how the threads work.

- Thread 1 is constantly waiting for incoming messages, and when it receives one, it prints it out and starts waiting for the next.
- Thread 2 is constantly waiting for the user to type in something, and when that happens, it sends that message to the server, and goes back to waiting for the user.

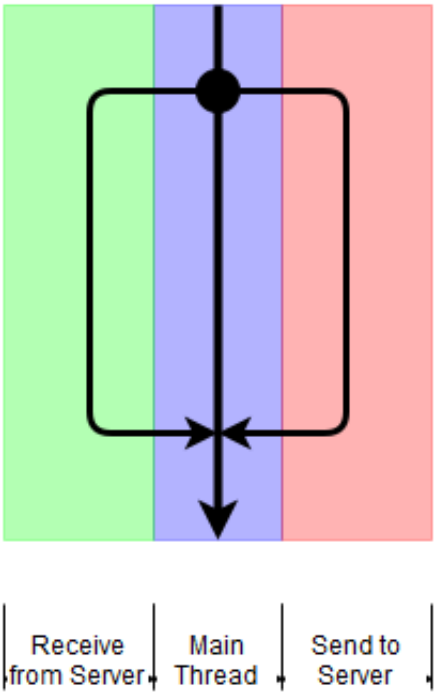

Figure 5: Implementation of threads in a client

Now the client can handle both asynchronous input and output while the main thread can still run and do all kinds of other stuff.

4.3 Server

The server is a little more complex than the client. However its main task is simple enough. It waits for an incoming message from any client connected to

it. When it receives a message, it will send that message to all other clients.

Once again, threads will be included. Naturally, there will be one thread for each client that waits for an incoming message from the corresponding client. One difference is that it will send the incoming message to all other clients except from the one it got it from.

This is implemented by having another sending thread inside each listening thread. When a message arrives all the sending threads will send the message to its client, except the thread which belongs to the client who sent the message. Figure 6 shows how the threads are organized. There's several issues with this

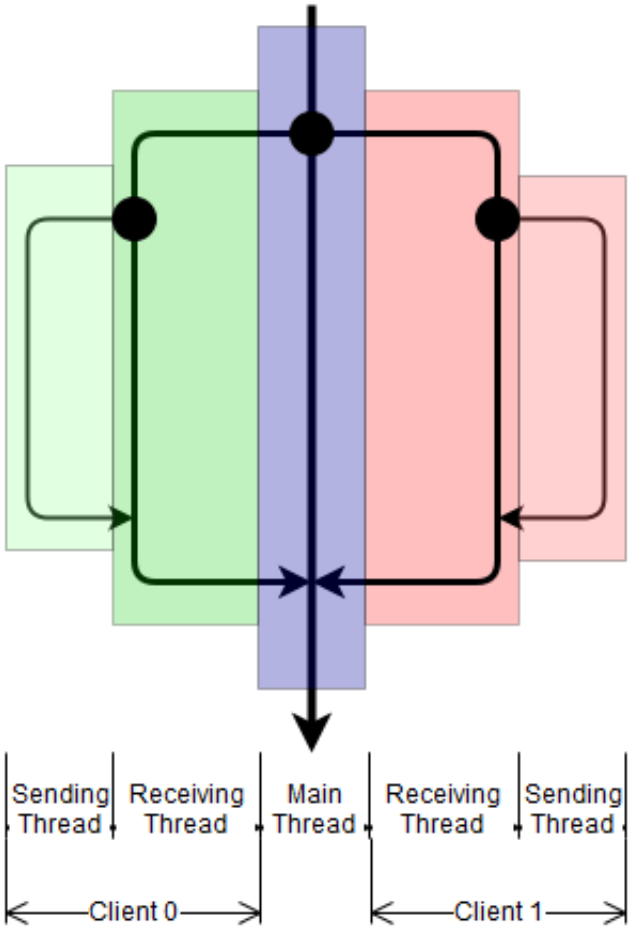

Figure 6: Implementation of threads in the server with two clients

simple approach.

1. When a client receives a message it would be essential to know which client sent that message.

2. If messages from several clients arrive at the same time, some of them might be lost, because all of them might not reach the sending part.

4.3.1 Message Structure

Problem 1 can be solved by modifying the message structure. Initially, only the message itself was sent back and forth. But when a client receives a message it needs to know which client sent that message.
Hence, the server needs to add additional information to the message, before sending it to clients. Figure 7 shows the current message format. Its practical

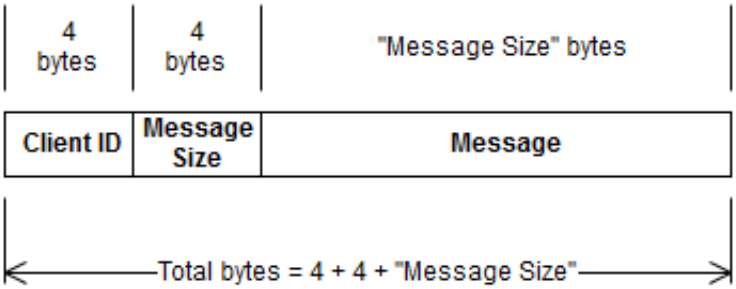

Figure 7: Structure of the message. The first 4 bytes are the ID of the client who sent the message in decimal number. The next 4 bytes says how many bytes the actual message is, in a decimal number. After that comes the message itself.

with a fixed number of bytes for the additional information, so that the server and the clients always know where in the message to find the appropriate information. The max size for a message right now is 512 bytes, but this could be easily modified.
Figure 8 shows how the message is modified, on its transfer from client to server to other clients.

4.3.2 Message Queue

Problem 2 can be solved by implementing a queue data structure in the server. When a message from a client arrives, it will be added to the queue. As long as the queue is not empty, the sender threads will send the message that is first in the queue.
So if a message arrives, while the current message is being processed for sending, it will not get lost, since it will just be added to the queue and wait for its time to be processed. Figure 9 shows an example of a scenario where the queue prevents messages from being lost.

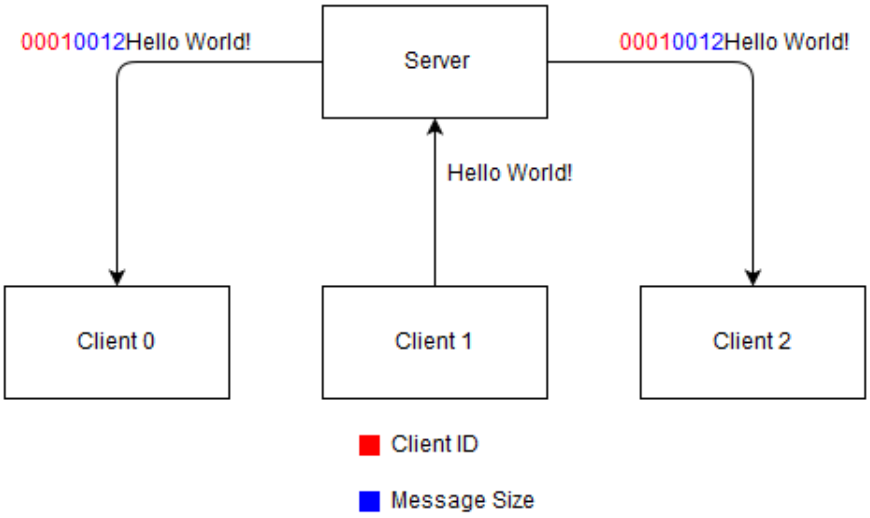

Figure 8: Client 1 sends a message "Hello World!" to other clients

5 Example

This section goes through a step-by-step example with 3 clients to give a better understanding of how the system works.

5.1 Initialization

The server and the 3 clients start up. There is no connection between the clients and the server. They are just 4 independent programs running.
However each client's socket is set up and is ready to connect to the server whose IP address it is given.
Likewise, the server's listening socket is set up and is waiting to accept connection requests from clients. It has an equal amounts of threads listening to requests as the maximum amount of clients allowed to be connected at once.

5.2 First client connects to the server

A client sends a message. This action triggers a connection request to the server. The server receives the request and accepts the client. Now the thread that accepted client 0 launches a new thread that deals with sending messages to client 0. It is now called client 0, since the server assigns 0 as its ID.

Note that since there's only one client connected to the server, there's no other client to send the message to, so the message is thrown away.

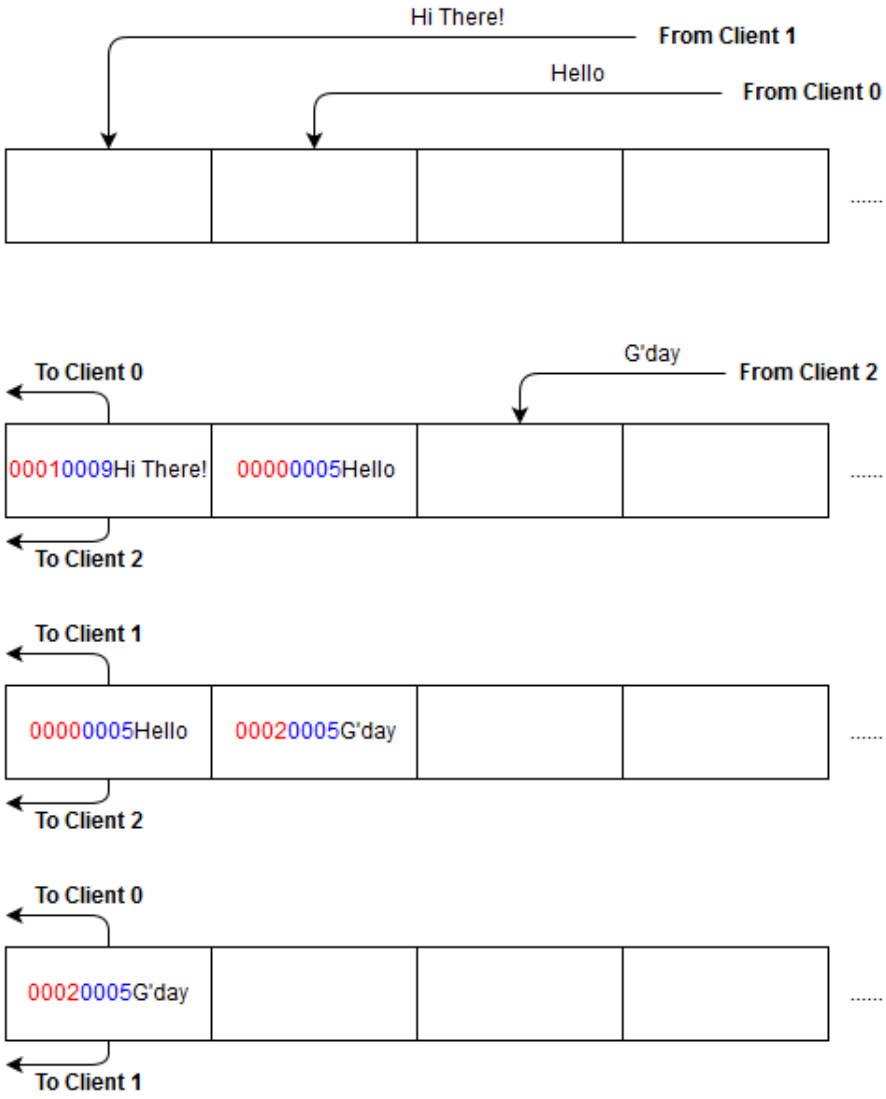

Figure 9: An example of how the message queue works with 3 clients

5.3 Another client connects to the server

Another client connects to the server, by sending a message to the server. The server accepts the request, assigns 1 as its ID, and the sending thread for client 1 is launched. Now client 0 and 1 are connected to the server.

The message that was sent by client 1 will now be added to the message queue, with the added information. Both sending threads for client 0 and 1 will notice

that the queue is no longer empty. They both check which client it came from. It came from client 1, so only the sending thread for client 0 will send the message to client 0.

Now that the queue is empty again, the sending threads keeps waiting again, and the listening threads are also waiting for messages from the clients.

5.4 Yet another client, and so on

Client 2 connects by sending a message. The server receives its message and the sending threads for client 0 and 1 send the message with added information to client 0 and 1. It keeps on going on like this for every message and for any number of clients below some maximum limit. That is all there is to this implementation.

6 Discussion

This program is still under development, so bugs are bound to happen.

6.1 Disconnection

Disconnection of sockets has not been implemented yet. I'm working on making a client successfully disconnect from the server, without crashing the whole network, which happens now.
The idea is that when a client types a specific codeword, this will make the server close down its connection to the client, and the server can wait for another connection to that spot.

6.2 Race conditions

The system seems to do well, but there are still bugs. Sometimes when a server receives a message from a client, the message is not sent to the other clients. After this happens, the server will stop completely to send messages, only receive. I suspect it's because of race conditions that are not solved yet.

6.3 Simultaneous messages

It is mentioned that the queue was implemented to handle several messages coming in at a short time span. This has not been tested properly yet, since all testing so far has happened on one machine, where it is hard to create scenarios like this. It needs proper testing with several machines to verify this.

6.4 Connection issues

As mentioned, it has only been tested on one machine, so all connections were successful and no socket errors have occurred. Socket errors have not been tested, so I don't know how it handles them.

6.5 Performance

Performance has not been tested much. It seems to do well with three clients and a server running. But when the number of clients become large, it might lead to slow performance, and further optimization may be required.

7 Conclusion

This implementation will be able to handle basic text-based communication between a general number of clients. This is a very good basis for making all kinds of applications involving network communication. The main goal with this system is to give people ideas of how it can be done and also open for discussion for better implementation alternatives.

A lot more testing needs to be done to see how robust it is, and how robust it can be. Also basic implementations like disconnection and error handling needs to be implemented properly before calling it a proper multichatting implementation.

8 Appendix

References