Subject: CAB403 Systems Programming
Title: Distributed Systems Major Assignment
Name: Shaun Kickbusch
Student Number: n9962361
Individual Group Number: 222

# Statement of Completeness

## Task 1

| Task | Completeness |
|---|---|
| **Server command line parameter – configurable port & default port** | Fully Implemented |
| **Server exits gracefully upon receiving SIGNAL (ctrl + c)** | Fully Implemented |
| **Client command line parameters** | Fully Implemented |
| **Implementation of SUB <channelid>** | Fully Implemented |
| **Implementation of CHANNELS** | Fully Implemented |
| **Implementation of UNSUB <channelid>** | Fully Implemented |
| **Implementation of SEND <channelid> <message>** | Implemented however if the user uses a space to separate words only the first word will be saved because tokens have been used to try and cut up the string. |
| **Implementation of BYE** | Fully Implemented |
| **Implementation of NEXT <channelid> and NEXT** | Fully Implemented |
| **Implementation of LIVEFEED <channelid> and LIVEFEED** | Partially implemented both. Don't have signal handlers to break out of this command however. |

## Task 2

| Task | |
|---|---|
| **Multiprocess implementation** | Fully Implemented |
| **Process synchronization** | Fully implemented through the use of mutex locks |

## Task 3

| Task | |
|---|---|
| **Thread usage** | Not Implemented |
| **Thread synchronisation** | Not implemented |

# Description of Data Structures

## Client

```
typedef struct{
	struct sockaddr_in address;
	int subbedChannels[MAX_NUMBER_OF_CHANNELS];
	int clientSocket;
	int clientID;
} myClient;
```

The above code is my implementation for the client's structure. It contains the address of the client, an array called `subbedChannels` which allows the program to see whether or not a client is subbed. Each index in the `subbedChannels` array is represented by either a 1 or 0. 1 meaning they're subbed and 0 if they're not. Since we have 255 channels the array size for subbed channels is 256. Each index is a direct representation of that channel number. For example, index 0 in the array is representative of channel 0. The client's socket number is stored in `clientSocket`. `clientID` is the number the client is given when they connect. For example, when the server's running and the first client connects, they will be given a `clientID` of `0`. For every client that connects this `clientID` variable is incremented so each instance of the `myClient` struct contains a unique `clientID`.

```
myClient *clients[MAX_CLIENTS];
```

The above piece of code is where all the clients are stored. It's essentially an array of pointers which point to each unique client's struct for each index. For example, when the first client connects to the server, their struct will be stored in the first index of `*clients`. Each subsequent client's struct will then be stored in the next available index.

## Messages

```
typedef struct{
	int channelIDForMessage[MAX_NUM_OF_MESSAGES_IN_CHANNEL];
	char
channelMessage[MAX_NUM_OF_MESSAGES_IN_CHANNEL][MAX_CHANNEL_MESSAGE_SIZE];
	int readMessages[MAX_CLIENTS][MAX_NUM_OF_MESSAGES_IN_CHANNEL];
	int amountOfMessageSent;
} messageBank;
```

The above piece of code is the implementation for where all the messages are stored. The first array inside the struct titled `channelIDForMessage` indicated which channel the messages stored in the `channelMessage` belong to. For each index in `channelMessage`, the same index in `channelIDForMessage` will represent what channel that respective message belongs to. For example, if a message is sent to channel 0, this string is stored in `channelMessage[0]`.

channelIDForMessage[0] will then be given a value of 0 to represent the correct channel that the message was stored to. The next array, channelMessage stores a maximum of 1000 messages with a length of 1024 as per the assignment task. The readMessages array represents whether or not a client has a read a message. It is a 2D array with the first index representing the clientID as seen in the previous struct for the client and the second index representing whether or not they've read the message. Once again, the same index in channelIDForMessage and channelMessage will be represented by the same index in readMessage. When the program is initialized every index in readMessages is set to -1. When a user sends a message to a channel the respective index in readMessage is set to 0 to denote an unread message. When a user reads a message the respective index value is set to 1. The next value amountOfMessageSent keeps track of the current amount of messages sent. For every message sent this value is incremented to keep track of index we write to when another message is sent. To give an example of how everything operates, we have 2 clients with IDs of 0 and 1. If the client with ID 0 sends the message "Hello" to channel 0 we will have the following values:

```
int channelIDForMessage[0] = 0;
char channelMessage[0] = "Hello";
int readMessages[0][0] = 0; int readMessages[1][0] = 0;
amountOfMessageSent = 1;
```

When client 0 and 1 read the message via the NEXT command the following values will be allocated:

```
int readMessages[0][0] = 1; int readMessages[1][0] = 1;
```

messageBank is utilised by all clients so it's imperative that it's allocated to shared the memory. The following code was implemented to achieve this:

```
int messageBankSize = sizeof(messageBank);
messageBank *messageBankPtr;
messageBankPtr = mmap(NULL, messageBankSize, PROT_WRITE | PROT_READ,
MAP_SHARED| MAP_ANONYMOUS, -1, 0);
```

## Forking Description

Forking was implemented relatively easily. For each client that connects a new child process is created. This was achieved through the typical `fork()` command which was assigned to a `pid_t` data type. The value assigned to the `pid_t` was then checked to see if it contained a value less than or equal to `0`. A `0` indicates that the fork failed and a `0` denotes a successful creation. The following piece of codes illustrates what appears in my server program:

```
pid_t childpid;
childpid = fork();
if (childpid < 0){
      printf("Creating the fork failed");
}
else if (childpid == 0){
      clientFunction((void*)clientPointer, messageBankPtr);
}
else {
      wait(NULL);
}
```

The `clientFunction()` handles everything needed for each client. It has a buffer, which contains data received from the client. Depending on this information, which is a char, we use if statements to check if it corresponds to any of the commands such as SUB or SEND. A pointer to the client is also passed to the function alongside the `messageBank's` pointer. This allows the function to access both of these datatypes. Since `clientFunction()` contains a while loop which ensures the program is checking constantly whether new data has been received it's imperative that when a client disconnects that this while loop is exited. This is simply achieved by testing to see if the client sent a BYE command or whether or not the buffer is empty. If this is the case, an integer value which acts as flag is set to `1` and an if statement, at the beginning of the while loop, will detect this and trigger the `break` command. Once the while loop is broken, that respective client's socket is closed, the pointer to their `myClient` struct removed and the memory allocated to this struct is freed. Once all this is completed, the `clientFunction()` has completed and thus the child process has finished everything. Command is then returned to the parent.

## Critical Section Problem Description

In my programs, whenever a user is reading or writing a pthread mutex is utilised. This mutex was initialised as the following:

```
pthread_mutex_t readAndWriteMutex = PTHREAD_MUTEX_INITIALIZER;
```

Between piece of code that were reading and writing from the `messageBank` the following functions were present:

```
pthread_mutex_lock(&readAndWriteMutex);
```

```
pthread_mutex_unlock(&readAndWriteMutex);
```

The use of these mutex locks only allows one child process, 1 client, to be reading or writing at a time solving the reader and writer problem.

## How to Run

Simple enter the folders directory and enter `make`. The make command will compile both the server and client. Once this is done simply type `./server` if you wish to run on the default port or `./server <port number>` if you would like to run it on a custom port. For the client, simply type `./client 127.0.0.1 <port number>`. In the case that the `make` command fails, use

```
gcc -o server server.c -lpthread
```

to compile the server and

```
gcc -o client client.c -lpthread
```

to compile the client program.