DISTRIBUTED SYSTEMS FINAL PROJECT REPORT (PART 3)

Malak Sadek - 900140107,
Mohamed Hegab – 900142811,
Ibrahim Abdelrahman - 900131254,
Baher Mahmoud - 900133218,
Mohamed Badreldin - 900140813

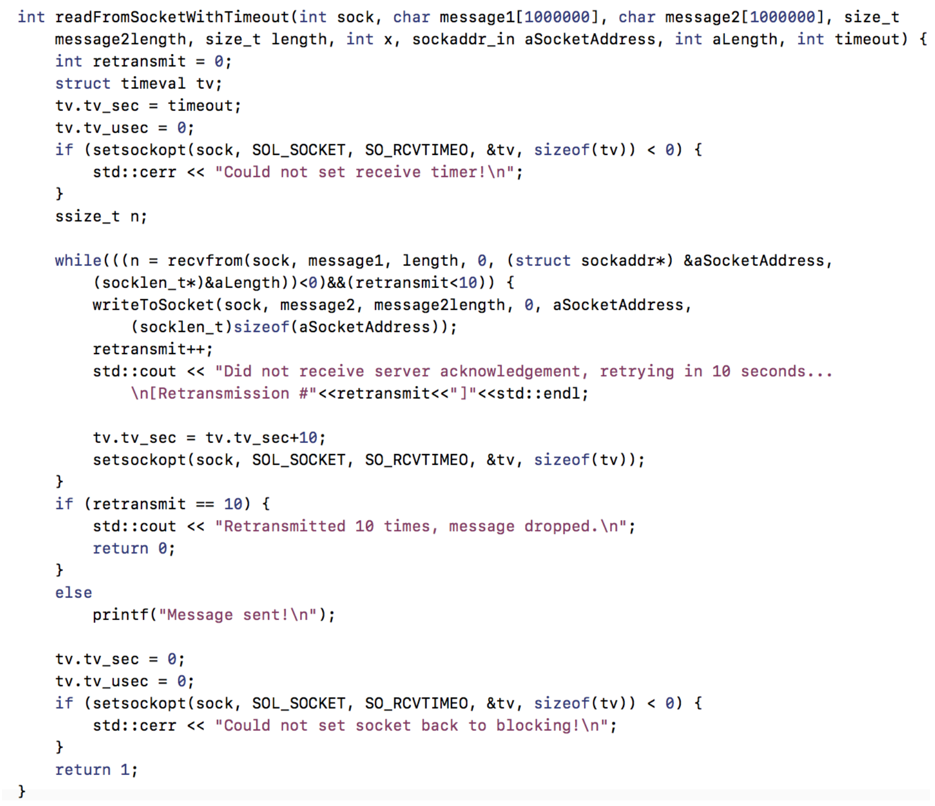
For exercise 3 of the project, we tried to prevent or mask as many failures as possible.
Below is a detailed list of:
1. Possible failure scenarios,
2. The methods we used to either mask or correct them,
3. The experiments used to test our methods and how our system reacted to the experiments.

Message Transmission Failure Handling

1. A message is lost /does not arrive at destination

How We Handled It:

- A client can receive two types of messages. It can either receive a message that is smaller than 500 characters, which would be a client list, picture list, or views. Or a message that is larger than 500 characters, which would be a picture fragment.
- For messages under 500 characters, clients await a reply from the server by reading from their socket with a timeout. If the timeout period (which is a different value based on the kind of request obtained through experimentation to find the normal response time and then adding a few milliseconds to it) has elapsed with no reply (either because the request or the reply were lost), the client then retransmits the request. These kinds of requests are not checked for duplicates as they are idempotent.
- For messages over 500 characters, the client will have previously issued a request for an image and resumed normal operation. If the other client (server in this case) rejects the request, nothing will happen, however if they accepted the request, then the reply would be the first fragment of the image. As soon as this first packet is received, the client begins issuing acknowledgements. Consecutive communication takes the form of
fragment – acknowledgement – next fragment – etc.
- If a packet containing an image fragment is lost, it will be retransmitted, this is due to the fact that the sender will wait for an acknowledgement, if it is not received, then a retransmission will occur. Similarly, if an acknowledgement is lost, it will also be retransmitted since the receiver will be waiting for the next packet and if it is not received then the acknowledgement is retransmitted.
- **At any point of the process, retransmission occurs 10 times, and then the current operation is aborted. Each retransmission has a timeout interval that is 10 seconds longer than the prior attempt.**



```
int readFromSocketWithTimeout(int sock, char message1[1000000], char message2[1000000], size_t
    message2length, size_t length, int x, sockaddr_in aSocketAddress, int aLength, int timeout) {
    int retransmit = 0;
    struct timeval tv;
    tv.tv_sec = timeout;
    tv.tv_usec = 0;
    if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
        std::cerr << "Could not set receive timer!\n";
    }
    ssize_t n;

    while(((n = recvfrom(sock, message1, length, 0, (struct sockaddr*) &aSocketAddress,
        (socklen_t*)&aLength))<0)&&(retransmit<10)) {
        writeToSocket(sock, message2, message2length, 0, aSocketAddress,
            (socklen_t)sizeof(aSocketAddress));
        retransmit++;
        std::cout << "Did not receive server acknowledgement, retrying in 10 seconds...
            \n[Retransmission #"<<retransmit<<"]"<<std::endl;

        tv.tv_sec = tv.tv_sec+10;
        setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
    }
    if (retransmit == 10) {
        std::cout << "Retransmitted 10 times, message dropped.\n";
        return 0;
    }
    else
        printf("Message sent!\n");

    tv.tv_sec = 0;
    tv.tv_usec = 0;
    if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
        std::cerr << "Could not set socket back to blocking!\n";
    }
    return 1;
}
```

Setting the socket to timeout mode, trying to read, and rewriting 10 times while increasing the timeout interval by 10 seconds each time, then aborting operation.

Experiments to Test Handling & Results:

- Client issues a request for a client list while centralized server is down:

The client reissued the request 10 times and then stopped trying, if the server came up before the 10th request, it replied only once and the client received the reply.

- Client issues a request for a picture list or views while the other client is down:

The client reissued the request 10 times and then stopped trying, if the other client came up before the 10th request, it replied only once and the client received the reply.

- Client issues a request for a client list, picture list, or views and falls before receiving a reply:

The receiver does not retransmit the reply as it does not wait for an acknowledgement since all these requests are idempotent. The client can reissue the request at another time.

- Client falls before receiving an image fragment:

The server attempted to resend the fragment 10 times without getting an acknowledgement, it then restarted and attempted to send the first fragment all over again 10 times without an acknowledgement, then aborted the operation. If the client came up before the 10th attempt, it received either the current fragment or the first fragment again depending on when it came up. If it received the first fragment again, it restarts the receiving and combining process on its side.

- Server (another client) falls before receiving an acknowledgement for an image fragment:

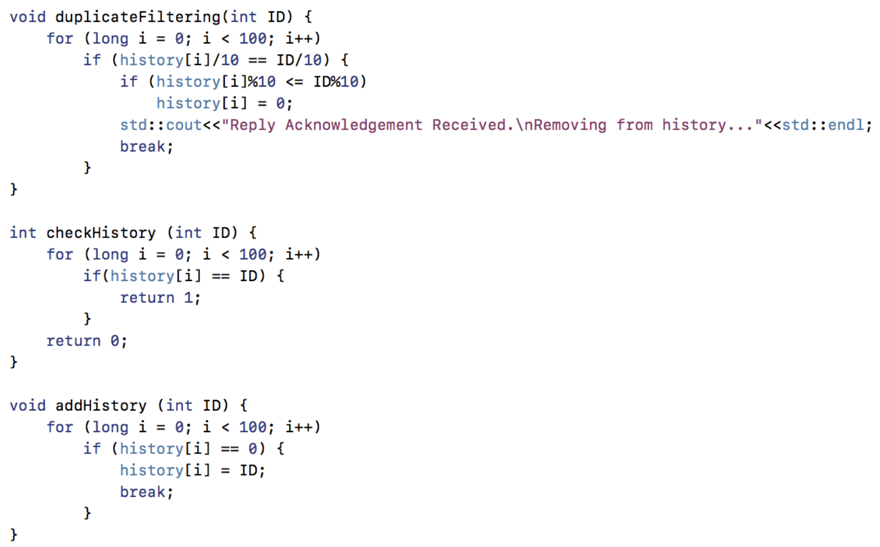
The client retransmitted the acknowledgement for the last packet it received 10 times, and then aborted the operation and discarded any received image fragments.

Communication Failure Handling

2. A duplicate request is received as the server replied after the timeout period

How We Handled It:
- Both the centralized server, and the server thread of the client, maintain history tables. These tables store the RPCId of requests received. The RPCId is a combination of a unique client ID and a request count. Once a new request is received, it is checked against these tables, if it is found, then the request is considered a duplicate and is ignored. The only way to remove a RPCId from the tables is to receive an acknowledgement with the same RPCId, in which case the entry, and all entries with the same unique client ID and smaller request counts (since they must have been acknowledged if the client made a subsequent request), are removed from the tables. This mechanism, combined with acknowledgement and reply retransmissions, ensures that if both parties are up and running, all messages are eventually delivered.

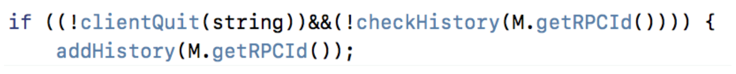
Experiments to Test Handling & Results:
- Client given extremely small timeout period to force retransmission even though the server received the request:
Even when the client retransmitted the request up to 10 times, the server only replied to the very first request and ignored all the rest. It kept the RPCId of the first request in its history table. If that request was for a picture, then the entry was removed when the acknowledgment was received, if it was for something else, the entry was removed when the next request was made and acknowledged. If the history table becomes full, entries are replaced using a FIFO policy.



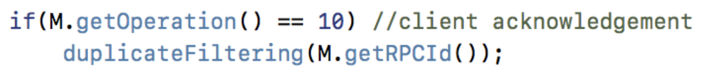
```
void duplicateFiltering(int ID) {
    for (long i = 0; i < 100; i++)
        if (history[i]/10 == ID/10) {
            if (history[i]%10 <= ID%10)
                history[i] = 0;
            std::cout<<"Reply Acknowledgement Received.\nRemoving from history..."<<std::endl;
            break;
        }
}

int checkHistory (int ID) {
    for (long i = 0; i < 100; i++)
        if(history[i] == ID) {
            return 1;
        }
    return 0;
}

void addHistory (int ID) {
    for (long i = 0; i < 100; i++)
        if (history[i] == 0) {
            history[i] = ID;
            break;
        }
}
```



```
if ((!clientQuit(string))&&(!checkHistory(M.getRPCId()))) {
    addHistory(M.getRPCId());
```



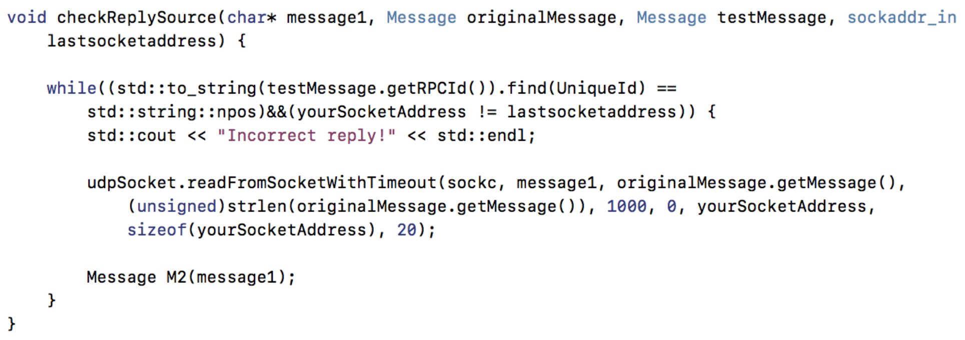
```
if(M.getOperation() == 10) //client acknowledgement
    duplicateFiltering(M.getRPCId());
```

The methods for adding to history, checking if an entry exists, and removing from history, as well as 2 examples of their usage within other functions.

3. A server incorrectly sent a reply to the wrong client

How We Handled It:

- Whenever a client receives a request, it checks the source of the reply and its RPCId. It matches both of these against the destination and RPCId of the last request it made. If they do not match, then it will ignore this reply and await the correct one and retransmit if it is not received in time.



```
void checkReplySource(char* message1, Message originalMessage, Message testMessage, sockaddr_in
    lastsocketaddress) {

    while((std::to_string(testMessage.getRPCId()).find(UniqueId) ==
        std::string::npos)&&(yourSocketAddress != lastsocketaddress)) {
        std::cout << "Incorrect reply!" << std::endl;

        udpSocket.readFromSocketWithTimeout(sockc, message1, originalMessage.getMessage(),
            (unsigned)strlen(originalMessage.getMessage()), 1000, 0, yourSocketAddress,
            sizeof(yourSocketAddress), 20);

        Message M2(message1);
    }
}
```

This function compares the incoming address and RPCId with those of the last request made.

Experiments to Test Handling & Results:

- Client made a request, then the destination is closed, and a third client is hard-coded to send the original sender a false reply:

The client ignored the reply as it did not match the request's credentials, retransmitted 10 times, and then aborted the operation, or successfully received a reply if the correct destination came back online before the 10th retransmission.

4. Two clients issue requests to the same server at the same time

How We Handled It:

- This will essentially create a race condition, where whichever request will arrive first will be serviced first. The other client will retransmit and then give up and can try again at a later time. The duplicate requests will be ignored and idempotent operations can simply be requested again. If the client had requested an image, it can receive it at any time after the request is made since it will not block afterwards.

Experiments to Test Handling & Results:

- Two clients issued requests to a third client at the same time:

The client serviced one of them while the other one attempted to retransmit and gave up. After the third client had finished the first request, it then performed the second one if it was a picture request, or else the second image later reissued the request and received a reply.

5. A client quits unexpectedly (due to power outage or local error) without informing central server

How We Handled It:

- Normally, when a client quits, it will send a message to the centralized server, so that it can remove that client from its currently connected clients list. If the client quits abnormally, it might not send this message. To accommodate this, whenever a client requests a client list from the centralized server, the server first updates its client list. It does this by sending a message to all the clients currently in the list. If they reply, then they are kept on the list, if they do not, then they are removed. By doing this, the server always sends an up to date version of the list.

```
void updateClientList() {
    char reply[5];
    reply[0]='U';
    reply[1]='C';
    reply[2]='C';
    reply[3]='\0';
    char* message1 = new char [50];

    for (int i = 0; i < 5; i++) {

        if ((connectedClients[i].sin_port !=
            2222)&&(connectedClients[i].sin_addr.s_addr!=2222))

            udpServerSocket.writeToSocket(sock, reply, strlen(reply), 0,
                connectedClients[i], sizeof(struct sockaddr));

        if (!udpServerSocket.ServerReadFromSocketWithTimeout(sock, message1,
            strlen(message1), aSocketAddress, sizeof(struct sockaddr), 10)) {

            connectedClients[i].sin_port = 2222;
            connectedClients[i].sin_addr.s_addr = 2222;
        }
    }
}
```

This function sends messages to all sockets in the connected clients list and awaits a reply, if it does not receive one, it removes the client (by setting the entry's port and address to 2222).

Experiments to Test Handling & Results:

- A client is forcibly terminated and then another client requests a client list:

As soon as the other client requested the list, the centralized server sent a message to all the clients on the currently connected list. It awaits replies with a timeout, but does not retransmit. Those who reply are kept on the list, while the others are removed. If a client replies after the timeout, it can reconnect to the server as if it was its first time (as this happens automatically when a new client is running and given a username) and is then added to the list again.

6. The central server falls at any time

How We Handled It:

- If the central server falls, the clients can still communicate with each other directly if a client list was previously requested, however it will no longer have a valid list of currently connected clients. If a client requests a client list and this client is not stored in the server's currently connected clients list (which must mean the server crashed since the fact that the client is making a request to the server means that it previously connected to it so it must be on the currently connected list), the server informs it that it has crashed and that it must restart.

Experiments to Test Handling & Results:

- Clients connect to the central server, and then the central server is closed and rebooted:

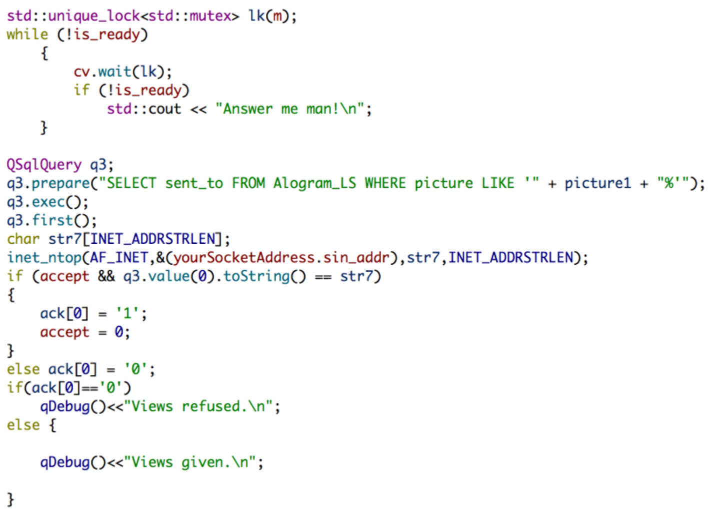
The server recognizes that a client is making a request to it even though it is not on its currently connected list and asks the client to restart, this is repeated until all clients are restarted and thus reconnected.

Security Failure Handling

7. A client requests views for a picture they do not own

How We Handled It:

- The server thread of each client maintains a database of the other clients that it has sent images to. Whenever a request for more views is received, the server thread will first check that the source of the request has received this image before. This database does not get deleted if a client quits normally or abnormally.



```
std::unique_lock<std::mutex> lk(m);
while (!is_ready)
    {
        cv.wait(lk);
        if (!is_ready)
            std::cout << "Answer me man!\n";
    }

QSqlQuery q3;
q3.prepare("SELECT sent_to FROM Alogram_LS WHERE picture LIKE '" + picture1 + "%'");
q3.exec();
q3.first();
char str7[INET_ADDRSTRLEN];
inet_ntop(AF_INET,&(yourSocketAddress.sin_addr),str7,INET_ADDRSTRLEN);
if (accept && q3.value(0).toString() == str7)
{
    ack[0] = '1';
    accept = 0;
}
else ack[0] = '0';
if(ack[0]=='0')
    qDebug()<<"Views refused.\n";
else {

    qDebug()<<"Views given.\n";

}
```

The function first checks the database to make sure the picture that a client is requesting views for has been sent to them before.

<u>Experiments to Test Handling & Results:</u>

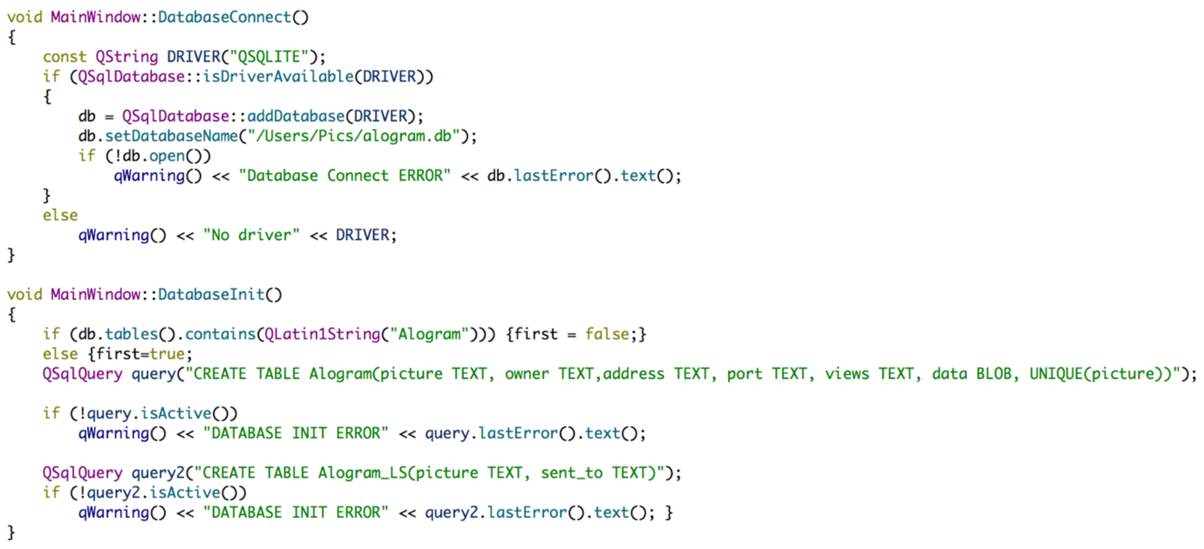
- A client requests views for an image that it did not receive from another client:

The other client does not find the first client in its database, so it does not send the views, it does not send a reply at all. The other client retransmits 10 times and gives up while the first client ignores the duplicate requests.

8. A client requests a picture that the server does not have

<u>How We Handled It:</u>

- Requests for pictures include the picture's name, and the number of views requested for the picture. This name is checked against the images that the client has and if they do not have an image with that name, they will ignore the request.



```
void MainWindow::DatabaseConnect()
{
    const QString DRIVER("QSQLITE");
    if (QSqlDatabase::isDriverAvailable(DRIVER))
    {
        db = QSqlDatabase::addDatabase(DRIVER);
        db.setDatabaseName("/Users/Pics/alogram.db");
        if (!db.open())
            qWarning() << "Database Connect ERROR" << db.lastError().text();
    }
    else
        qWarning() << "No driver" << DRIVER;
}

void MainWindow::DatabaseInit()
{
    if (db.tables().contains(QLatin1String("Alogram"))) {first = false;}
    else {first=true;
    QSqlQuery query("CREATE TABLE Alogram(picture TEXT, owner TEXT,address TEXT, port TEXT, views TEXT, data BLOB, UNIQUE(picture))");

    if (!query.isActive())
        qWarning() << "DATABASE INIT ERROR" << query.lastError().text();

    QSqlQuery query2("CREATE TABLE Alogram_LS(picture TEXT, sent_to TEXT)");
    if (!query2.isActive())
        qWarning() << "DATABASE INIT ERROR" << query2.lastError().text(); }
}
```

All photos owned by the user and received from other clients are stored in a database (the pictures received can only be accessed from the database in their marshaled form to make sure they cannot be seen outside the application).

Experiments to Test Handling & Results:

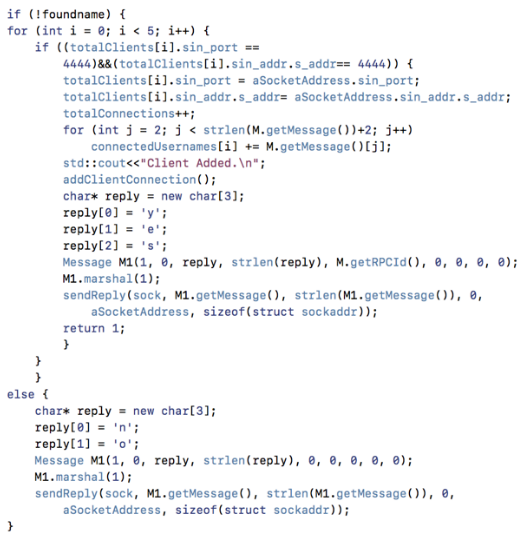
- A client requests an image using a name that the other client does not have stored:

The receiver does not find an image with that name stored locally, so it ignores the request. The sender does not wait for a reply and resumes normal operation, it can request another image at another time.

9. A client uses a duplicate username

How We Handled It:

- When a client first starts running, the user is prompted to enter a username. This username is automatically sent to the centralized server which checks it against the list of currently connected clients while also sending the "connected clients update" message discussed previously to make sure the list is up to date. If the name already exists, the centralized server replies with a negative message and the user must enter a different username, if the name is not found, it is added to the list and the client can start making requests. A list of all the usernames to ever connect to the server is also maintained (even if they are not currently connected) to keep track of the system's users.



```
if (!foundname) {
for (int i = 0; i < 5; i++) {
    if ((totalClients[i].sin_port ==
        4444)&&(totalClients[i].sin_addr.s_addr== 4444)) {
        totalClients[i].sin_port = aSocketAddress.sin_port;
        totalClients[i].sin_addr.s_addr= aSocketAddress.sin_addr.s_addr;
        totalConnections++;
        for (int j = 2; j < strlen(M.getMessage())+2; j++)
            connectedUsernames[i] += M.getMessage()[j];
        std::cout<<"Client Added.\n";
        addClientConnection();
        char* reply = new char[3];
        reply[0] = 'y';
        reply[1] = 'e';
        reply[2] = 's';
        Message M1(1, 0, reply, strlen(reply), M.getRPCId(), 0, 0, 0, 0);
        M1.marshal(1);
        sendReply(sock, M1.getMessage(), strlen(M1.getMessage()), 0,
            aSocketAddress, sizeof(struct sockaddr));
        return 1;
        }
    }
    }
else {
    char* reply = new char[3];
    reply[0] = 'n';
    reply[1] = 'o';
    Message M1(1, 0, reply, strlen(reply), 0, 0, 0, 0, 0);
    M1.marshal(1);
    sendReply(sock, M1.getMessage(), strlen(M1.getMessage()), 0,
        aSocketAddress, sizeof(struct sockaddr));
}
```

This function replies according to whether a duplicate username was found in the currently connected clients list or not.

Experiments to Test Handling & Results:

- A client logs in with a username, then a second client attempts to log in with the same username while the first client is still running:

The first client is allowed to log in, the second client is refused and asked to enter another username. When the first client closed, the second client was allowed to log in using the first client's old username.