

report

by Ruaidhri David Cook

FILE	4148_RUAIDHRI_DAVID_COOK_REPORT_113614_2051046288.DOCX		
TIME SUBMITTED	10-MAY-2018 11:29PM (UTC+0100)	WORD COUNT	2312
SUBMISSION ID	88919811	CHARACTER COUNT	11127

TCP/IP Client Server Implementation and Evaluation

Name: Ruaidhri Cook

Student Number: P148970

Module: SYSTEM SOFTWARE & MULTI-USER APPLICATIONS ENGINEERING

Module Code: M5X01055

Introduction

In this report the code for the program will first be identified, with the features and implementation of code being explained, this means the functionality will be explored. This report will also evaluate the complexity of the code and also it's sustainability for the development of the program, this also means parts of the code will be scrutinised and alternatives could be suggested.

Identification / Description

In this section the part of the code that have been implemented will first be identified and then their use in the program will be explained.

Chat Client Main Function

```
ChatClient::ChatClient(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this);
    nickAccepted = false;

    //Setup the timer
    timer = new QTimer();
    timer->setInterval(1000);

    //TODO activate this when your ready (i.e. you're ready to write the
    //slot associated with it!
    QWidget::connect(timer,SIGNAL(timeout()),this,SLOT(getMessages()));

    lastMessageNo = 0;
}
```

In this section of the code the User Interface is set up, which connects the UI functions the client back end. This section also initialised the nickAccepted variable so that it can be used later in the program, this variable is initially set to false as it essential for other functions in chatClient to work. In addition to this the timer variable is created, which will also be used in later functions. This part of the code also calls the QWidget::connect function which directly interacts with the ChatClient::getMessages function, calling it repeatedly so as to update the messages. In addition to this the lastMessageNo variable is set to zero as no messages have been sent on the system.

Chat client Join button clicked method

```
void ChatClient::on_joinButton_clicked()
{
    Join *j;
    Confirm *c;
    INet4Address *servaddr;
    Connection *myConnection;
    int sockfd, n;
    char* buffer; //for encoded messages
    int bytesToSend;
    int bytesToReceive;
    int bytesReceived;

    pid_t pid;
    int pipeFd[2];
    int result = pipe(pipeFd);
}
```

This screenshot shows the initialisation of variables, the first two, Join and Confirm, are actually objects that have inherited from the Message object, this means they have the same properties and are seen as a type of Message. The other objects that are initialised are INet4Address and Connection, these objects are used for establishing the connection and creating the stream used to pass messages. The bottom three variables are used in the creation of a data pipe.

```
//Get the strings out of the address, port and nickname
//boxes and convert to strings;
QString qip = ui.addressLine->text();
QString qport = ui.portLine->text();
QString qnick = ui.nicknameLine->text();

//Create new connection objects
servaddr = new INet4Address(qip.toLatin1().data(),qport.toInt());
myConnection = new Connection(servaddr);

//TODO your code here!!

//TODO Connect to server

myConnection->createSocket();
sockfd = myConnection->connectSocket();
```

This part of the code initialises the variables which store the data received from the UI when the corresponding buttons are pressed. This part of the code also initialises the servaddr object, which is passed the Port number and the IP address variables. The myConnection object is also initialised and is passed the myConnection object. This then allows myConnection to utilise it's functions, the first of these functions creates the socket needed and the second establishes the connection to the socket, whilst also returning an integer which is assigned to sockfd (the socket file descriptor).

```

if(result == -1)
{
    printf("Pipe not created");
    exit(1);
}

while (true)
{
    if (sockfd < 0)
    {
        myConnection->closeSocket();
        exit(1);
    }

    else
    {
        pid = fork();

        if (pid == 0)
        {
            //Open the file

```

This section of the code shows an IF statement being used to check the result variable to see if the pipe has been created, exiting if it has not. In addition, a while loop is created so that the following code will be repeated until set to false or if a break is used. Within this while statement the sock file descriptor is used in an IF statement that determines whether the integer is was assigned is less than 0, this is done to check that a connection to the socket was actually established, otherwise the socket would be close and the function would be exited. However if an connection had been established it would proceed to the other bit of code which creates a fork, which will check the pid variable is equal to 0 to determine if the fork was executed correctly. If it was not the code would exit the function.

```

//TODO Make a join message
j = new Join();
j->setNickname(qnick.toLatin1().data());

//TODO Send join message (as bytes)
//TODO Write size of encoded object to stream
bytesToSend = j->encode(&buffer);

while (true)
{
    if (bytesToSend == -1)
    {
        if (errno != EINTR)
        {
            //printf("Read error - not interrupted");
            break;
        }
    }

    else
    {
        buffer[bytesToSend] = 0;

        if (bytesToSend == 0)
        {
            //printf("Recieve end of stream.\n");
            break;
        }

        else
        {
            if (bytesToSend == 2)
            {
                if (buffer[0] == '\r' && buffer[1] == '\n')
                {
                    myConnection->closeSocket();
                    exit(1);
                }
            }
        }
    }
}

```

If the fork was executed correctly this part of the code would be run. Firstly a join object, with the same properties as it's parent Message object, would be created. The join object, j, would then use the setNickname function to assign the value in variable qnick to it's nickname variable which is a char*. The object j would then call it's encode function and be passed the address of the buffer variable to encode the size of the object to the stream returning an integer containing the number of bytes to send, which is assigned to the bytesToSend variable. A while loop is then used. If the bytes to send were equal to minus 1 it would be clear the encode function did not work and thus a break would be used to stop the while loop. Otherwise it would set the final byte of the buffer to 0. Then it would check if the bytes sent were equal to zero, breaking the loop if they were. Otherwise the bytes sent would be checked to see if they were equal to 2, if such a case was true it would then check to see if the first byte in the buffer was a carriage return character and the second byte was a \n which would indicate a new line, if such conditions were true that would be no actual information present in the buffer, which would cause the program to close the socket and exit the function.

However if conditions were not true that would mean the buffer did contain information and could continue.

```
//TODO Write object to stream.
myConnection->writeSize(bytesToSend);
bytesToReceive = myConnection->written(buffer, bytesToSend);
delete(buffer);

//TODO Read response - take action!
bytesReceived = myConnection->readSize();
buffer = new char[bytesReceived];
myConnection->readn(buffer, bytesReceived);

//TODO Convert to message (confirm)
c = new Confirm();

//TODO Read size of object to receive and allocate new buffer
c->decode(buffer, bytesReceived);

//TODO Read message
c->getResponse();

//TODO Destroy socket
myConnection->closeSocket();
```

In this continued part of the code the object myConnection would call it's writeSize function which sends the socket file descriptor of the socket created earlier. Once this has been sent the written function can be called, being passed the buffer variable and the bytes to send for the function to use. The function will return an integer that contains the number of bytes that were written, which is assigned to the bytes to receive variable that was included, however it can be called without assigning the return value to a variable. The buffer will then be deleted. The code then uses the readSize function which receives the socket file descriptor, returning an integer containing the number of bytes received, which is assigned to the variable bytesReceived. This variable is then used when creating a new buffer viable to change the size of memory it is allocated. The readn function will then be used which is passed the buffer variable and the bytesReceived integer. After this a type of Message object is created called c which is a Confirm object. This object will then call its decode function, being passed the buffer variable and bytesRecieved variable, this function will decode the message from the stream to it's own object, it will then call it's getResponse function which will check the response. After this the socket will be closed as it is no longer in use.

```

//check if nick accepted
if (c->getResponse() == true)
{
    nickAccepted = true;
    ui.addressLine->setEnabled(false);
    ui.portLine->setEnabled(false);
    ui.nicknameLine->setEnabled(false);
    ui.joinButton->setEnabled(false);

    //Turn on the timer
    timer->start();
}
else
{
    nickAccepted = false;
    ui.nicknameLine->setText(tr("Rejected!"));
}

```

This section of code then uses the get response function to determine if there was a response from the c object and that the message had been decoded from the stream. This then means the nickname will be accepted and the nickname field, ip address field and port field will all be disabled on the user interface aswell as the join button. This will also stop the timer which is used to request message updates. However if the response is false the nickname field will be populated with the text 'Rejected!'. After this the myConnection and servaddr objects will be deleted.

Chat client Leave button clicked method

This method is very similar to the previous one with the only main difference being the last part of the code.

```
//check if nick removed
if(c->getResponse()==true)
{
    nickAccepted = false;
    ui.addressLine->setEnabled(true);
    ui.portLine->setEnabled(true);
    ui.nicknameLine->setEnabled(true);
    ui.joinButton->setEnabled(true);
    ui.nicknameLine->setText(tr(""));

    //Stop timer
    timer->stop();
}
else
{
    nickAccepted = true;
}
```

This section of code then uses the get response function to determine if there was a response from the c object and that the message had been decoded from the stream. This then means the nickname will be rejected and the nickname field, ip address field and port field will all be enabled on the user interface aswell as the join button. In addition to this the nickname field will also be set as blank. However if the response is true the nicknameAccepted variable will be set to true.

Chat client Send button clicked method

This method is also very similar to the last but with some subtle changes at the start and at the end.

```
//TODO Make a chat message
m = new ChatMessage();
m->setMessage(qmessage.toLatin1().data());
m->setSender(qnick.toLatin1().data());

//TODO Send chat message (as bytes)
bytesToSend = m->encode(&buffer);
```

In this part of the code at the start the variables qmessage and qnick are set using the ChatMessage object's functions setMessage and setSender. The function then uses its encode function to send the message as bytes whilst writing the object to the stream.

```
//check if nick removed
if(c->getResponse()==true)
{
    ui.lineEdit->setText(tr(""));
}
```

This small section of code at the end uses the confirm object's getResponse function which returns a boolean which indicated whether the nickname has been removed or not, if it has the text entry field will be set to blank.

Chat client Get message method

This method is the one connected to the timer, it is run when the timer goes off, however the large majority of the code will not execute unless the variable nickAccepted is true. This code is slightly different to the code in the other methods.

```
int i;
QString msg;

//TODO Stop the timer
timer->stop();
```

At the start of the code this two variables are initialised that will be used, also a new function is used, this stops the timer so that the code can be executed, if the timer was not stopped the method would keep on getting called every second regardless of if the code had been fully executed or not the previous time it was called.

```
//TODO Make a Request update message
ru = new RequestUpdate(lastMessageNo);

//TODO Send Request update message (as bytes)
bytesToSend = ru->encode(&buffer);
```

In this part of the code a RequestUpdate object is created called ru, it is then encoded using its encode function is passed the address of the buffer.

```

myConnection->writeSize(bytesToSend);
bytesToReceive = myConnection->writen(buffer, bytesToSend);
delete(buffer);

//TODO Read response - take action!
//TODO Read byte count
bytesReceived = myConnection->readSize();
buffer = new char[bytesReceived];
myConnection->readn(buffer, bytesReceived);

//TODO Convert to message
um = new UpdateMessage();

//TODO allocate buffer
um->decode(buffer, bytesReceived);

//TODO Destroy socket
myConnection->closeSocket();

//Append new messages to a single string
for(i=0;i<um->getMessageCount();i++)
{
    msg += QString(um->getMessage(i));
    msg += QString("\n");
}

//Append messages to the text box and update last message count
if(um->getMessageCount() > 0)
{
    lastMessageNo += um->getMessageCount();
    ui.chatWindow->appendPlainText(msg);
}

```

This section is similar to some of the other methods however it does have some differences. After the read part of the code is done an updateMessage object, called um, is created which is a type of the Message object. After this the object is decoded using it's decode function. The socket is socket is then closed as it is no longer needed. After this a for loop is used that iterates through the code within whilst it's index number i, that is increased by one each time, is less than the integer returned by the um getMessageCount function which as the name suggests counts the number of messages in the vector containing the messages. The code that is looped uses the getMessage function to get each message from the vector individually and converts it to a QString that is added to the variable msg followed by a \n which will mean the messages are on different lines when displayed. After this and if statement is used that checks that the message count is greater than zero, if it is greater than zero the lastMessageNo variable will be updated so it contains the number of messages obtained through the getMessageCount function. Following this the chat window object will call it's appendPlainText function and pass it the msg variable meaning it will update the display with all current messages showing.

```

//TODO Restart the timer
timer->start();

```

At the end of the method the timer is started again.

Evaluation

In the program only the client part was able to be created, however whilst the code was able to compile, the application did not actually function. This is likely due to errors in the code. With this said the functionality of the interfaces and techniques used can be evaluated objectively.

A technique used in the program is the use of sockets, these as the receiving end of a connection and are opened at ports, they are used in client/server programs. They function as a way for two programs (the client and the server) to communicate as sockets on both programs can send and receive data. This means that interprocess communication can occur as the operating system can run two separate processes simultaneously.

An alternative to the use of sockets is the use of pipes and forking which were attempted to be incorporated into the program. Pipes are the first part that are created, after a pipe is created it returns a pair of descriptors that are passed on to a child process by using a fork which splits the process. This has an advantage as it allows the parent process (the original process) and the child process to perform different tasks. This is useful as tasks do not have to be done sequentially, instead they can run (almost) simultaneously. They differ from Sockets as they are essentially used like files.

A different technique associated inter-process communication and synchronisation is Semaphore, and can be used in tangent with threading, it works by controlling the access to processes or threads and allowing shared variables to be accessed exclusively by the process' or thread's using them. This means that processes wanting to use a resource can only use it if the resource isn't already in use. [1] This is advantageous as multiple processes aren't using the same resource at the same time, will help prevent corruption of said resource.

The program also used objects, although commonplace, they allow for multiple similar entities to be created that inherit the variables and functions from their parent object. This can be seen in the Message object which contained functions that allowed the encoding and decoding of variables from a stream. This meant that the message could be passed to the multiple message types, allowing each object to perform a different job helpful to the client.

The program also uses serialisation for when it encodes and decodes from the stream, in particular it uses XDR, this works by converting the data it is passed to XDR and then placing that data in the XDR stream. This is advantageous in the program as it is compatible with sockets, this means data can be encoded to the stream by one process and decoded from the stream by another.

An alternative to XDR could be XML, the key difference between them is XML is human readable text whereas XDR is binary. This type of serialisation is advantageous because it is readable to humans so verifying encoding is easier, however it is limited in it's use and cannot work if binary results are required.[2]

References

- [1] Whatls.com, "Semaphore." [Online]. Available: <https://whatis.techtarget.com/definition/semaphore>. [Accessed: 09-May-2018].
- [2] w3schools, "XML Tutorial." [Online]. Available: <https://www.w3schools.com/xml/>. [Accessed: 10-May-2018].

report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

40/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

GRADING FORM: SMU GENERIC_GJ

RUAIDHRI DAVID COOK

40

RESEARCH-INFORMED LITERATURE



A good, detailed and honest report, albeit with a relatively small number of sources.

3.5

KNOWLEDGE AND UNDERSTANDING



Good evidence of knowledge and understanding in the report. I get the feeling that you have a reasonable handle on the topic, but struggled with the time available to actually finish the implementation.

6

ANALYSIS



Good conclusions drawn, some evidence of synthesis.

6

PRACTICAL APPLICATION AND DEP.



The client is a good way through development. As you allude to in your report, however, of course it can't do too much without the server, so inevitably testing it to iron out the remaining issues would've been difficult!

21

SKILLS FOR PROFESSIONAL PRAC.



Good spelling, punctuation and grammar, and a fairly well-presented report.

3.5