P20 Report

Jayant Kalyan
jk5g19@soton.ac.uk

Abstract: This paper talks about the implementation of the P20 lab. P20 consists of implementing a simplex communication between a Send window and its corresponding Receive window so that the Send window can project its user's input (which in this case are digital doodles) onto the Recieve Window. This implementation was multi-threaded and used the QThread library for many of the built-in function to support the multi-threading.

1. [3.1 Design and Implementation]: How is the Image painted onto the Send Window in the Qt framework and how is it retained within the window?

This requires the use of three events; QPaintEvent for the actual drawing of the lines onto the Send Window ,MousePressEvent for the registering of when to draw and a MouseMoveEvent to know where to draw. The image is drawn onto a Pixmap in order to retain the diagrams on the Window.
This shows the implementation of the paintEvent.

```
void Window::paintEvent(QPaintEvent
*userDrawingOutput)
{
    QPainter painter(this);
    //Instantiates the painter
      painter.drawPixmap(rect(), pix);
    //Creates a PixMap for the drawing
    //to be retained
}
```

The drawing system works by making lots small straight lines in order to show a drawn curve. This works by the MousePressEvent assigning a point as an inital point whilst the MouseMoveEvent sets a current point. The MouseMoveEvent repeatedly assigns the old current point to the initial point and initialises a new current point. This is so the paintEvent can repeatedly draw lines between a continually changing current and inital point onto a PixMap (or a QImage). This culminates in a line that trails the mouse cursors current point via a series of miniscule straight lines. This line drawing stops when the left mouse button is released by the user. The clear button works by filling the Pixmap with white pixels.

The implementation of the MousePressEvent and the MouseMoveEvent respectively.

```
void Window::mousePressEvent(QMouseEvent
*userDrawingInput)
{
    if(userDrawingInput->button()
    ==Qt::LeftButton)
      {
       lastRecordedPoint=
       userDrawingInput->pos();
       //Sets point it clicked on
       //as last recorded point.
       drawing=true;
      //Lets the system know
      //that the system is drawing
     }
}
```

```
void Window::mouseMoveEvent(QMouseEvent
*userDrawingInput)
{
        QPainter pixPainter(&pix);
        //Instantiates painter which
        //paints on the pixmap
        endingPoint=
        userDrawingInput->pos();
        //assigns current point
   pixPainter.drawLine(lastRecordedPoint
        ,endingPoint);
        //draws the line based on
        //2 points
        lastRecordedPoint = endingPoint;
        update();
        //changes the pixmap
        //based on the inputs
}
```

2. [3.2 Design and Implementation]: How are the commands represented in the communication between the send and the recieve Window and how were the commands serialised into 'packets'?

2.1. The Send Window

The communication between the send and recieve happens via 'packages' of data. These 'packages' are a struct (called dataPackage) which includes; command an unsigned integer that holds the current command of the input, data is a QByteArray which holds the parameter of the command and Intructions which is an enum that hold the three possible commands that the system could call: Blank ,Enlarge and Draw.

```
struct dataPackage {
    quint8 command;
    //holds the command
    //given in a quint data type ,
    //which is a unsigned int
    QByteArray data;
    //raw binary data as parameters
    enum Instructions {Blank = 0,
        Enlarge, Draw};
    //encodes commands
    //via ennumerating them
};
```

The commands are represented as unsigned integers since it is not common practice to encode commands as negative numbers . Since Qt does not use the uint data type from the std (standard) library, the use of Qt's own unsigned integer data was necessary. This is the quint data type. The quint data type has a lot of different variants that differentiate from one other on the basis of the amounts of bits they store an unsigned integer in. According to the Qt documentation the minimum number quint variant that can be used is the 8 bit. Hence quint8 was used to store commands.
The three commands available to the user are: blank, enlarge and draw. These were stored within an enum in order to make refering to the commands in the code much easier, whilst not taking up extra memory that would be used if the system were to actually transmit the keywords.
These commands also have their own corresponding data (analogous to the opcode and the operand) which are the parameter in which they should fulfil their command. In the implementation,the parameters are appended onto a QByteArray however for each command the parameters are different.

The parameters for the enlarge command is an input of the QSize data type. This would allow the system to append the QSize at different times in the ResizeEvent into the QByteArray.

```
void Window::resizeEvent
(QResizeEvent *userResizing)
{
    pix =
    QPixmap(userResizing->size().width()
    , userResizing->size().height());
    pix.fill();
    if(resizeCount>=1)
    //since the window initially
    //resizes itself when opening,
    //this caused issues with the
    //command system and lead to crashes
   { dataPackage pkg;//
    pkg.command =
    dataPackage::Enlarge;
    pkg.data.append(
    (const char *)&userResizing->size()
    , sizeof(QSize));
    }
    resizeCount++;
}
```

The parameters for the draw command is an input of two QPoint data type variables.One of the QPoint signify the initial point of the mouse whilst the other QPoint signifies the current point of the mouse. Both of these values are appended into the QByteArray which holds the data.

```
dataPackage pkg;
pkg.command = dataPackage::Draw;
pkg.data.append(
    (const char *)&lastRecordedPoint
    , sizeof(QPoint));
pkg.data.append(
    (const char *)&endingPoint
    , sizeof(QPoint));
```

The Blank command has no parameters so only the command is taken into account.

```
dataPackage pkg;
pkg.command = dataPackage::Blank;
```

These parameters are all typecasted as char data types as they are appended into the QByteArray.

3. [3.3 Design and Implementation]: The Send and the Recieve thread and how they allow for a 'thread-safe' data transmission?

3.1. The Send Thread

This dataPackage is outputted via using the OutputThread class's function transferPackage(*dataPackage). This is called at the end of any command. This transfers the dataPackage to the sendThread which is the OutputThread instantiated.
The transferPackage() function firstly initialises the dataPackage that had been transfered from the send window as an in-class variable and then it calls the built-in start() function which is used to call the run() function since the run function is protected and cannot be directly called. So in the system whenever a input is put inside the Send window the send thread is activated.

```
void OutputThread::transferPackage
(dataPackage* pkg)
{
    this->pkgRecieve=*pkg;
        start();
}
```

The run() function is a virtual function of the QThread base class and so has to be defined by the subclass which means that OutputThread class has to override the function with its own definition. The run() function initially uses the PackageRecieve variable to initialise the local dataPackage variable named pkg. Whilst the initialisation is going on, it is good practice to apply a local Mutex when initialising to avoid concurrency issues and maximise thread safety in the system.

```
MutexRecieve.lock();
    dataPackage pkg =this->pkgRecieve;
    MutexRecieve.unlock();
```

Using a shared global Boolean which determines whether or not the command is ready to be read, the system blocked the signals if there is a command being read and allows new commands if there is no command being read. After that the command is en-queued in the global queue and then the parameters are en-queued.
Whilst this unlocking is going on, the global Mutex will prevent the other receive thread from interfering with the global queue.

Once the commands and the parameters are added the recieve thread will have access to the queue. Allowing it to access the commands that the Send Window inputted.

```
globalMutex.lock();
    cmdQueueGlobal.enqueue(pkg.command);
    for (int i = 0;
        i < pkg.data.size(); i++)
cmdQueueGlobal.enqueue(pkg.data.at(i));

    qDebug()<<"command enqueued:"<
    <pkg.command;
    cmdRead = true;
    globalMutex.unlock();
```

3.2. The receive thread

As established earlier the receive thread has access to the global command queue so long as the global Mutex is not blocking it. Hence its job is to de-queue a command and its parameters so that it transfer the command to the receive window as a dataPackage data structure via using a signal.
As the receive thread is activated directly in the main class via using the start() function. The first function of the receive thread to be called would be the run() which is a virtual function ,that all QThread subclasses have, that must be defined by the subclass.
The run() function initially make sure the command has been read by waiting for the send thread to have read the command which is signaled via the cmdRead Boolean which is global and is shared by both of the threads.

```
    while(!cmdRead)
    {
        usleep(5);
    }
```

Once the command has been read, the receive thread will declare a local dataPackage variable. This declaration will be protected by a local mutex in order to block access to a variable of a protected function. This local dataPackage will store the command and the parameters that the receive thread would get from the global command queue.
The receive thread would initially de-queue the command from global queue and use it to initialise the command variable within the dataPackage structure.

```
pkg.command = cmdQueueGlobal.dequeue();
```

Then using fixed iteration (which would be determined by the length of the queue) the thread can append the data

variable in the dataPackage structure with the parameters that are being de-queued.
In order to practice thread safety, global mutexes are used to block the send thread from using the command queue when receive thread is using it.

```
globalMutex.lock();
pkg.command = cmdQueueGlobal.dequeue();
while(!(cmdQueueGlobal.empty()))
        {
pkg.data.append(
     cmdQueueGlobal.dequeue());
        }
globalMutex.unlock();
```

Then the signal is emitted to the receive window by using the transferPackageToRecieve() function.

```
emit transferPackageToRecieve(&pkg);
    while(!cmdFinished);
   cmdRead=false;
   cmdFinished = false;
```

By using conditional iteration, the receive thread will wait until the command has been finished and the acknowledgement of the command finishing is given via the cmdFinished Boolean.After the command has been executed in the receive window, cmdFinished is again initialised as false. Also cmdRead is also assigned as false.

4. [3.4 Design and Implementation]: How bit stream communication was implemented?

4.1. Encoding the bytes into a Bit Stream in the Send thread

This was done by one of the function within the OutputThread class called sendByteValue(quint).
Prior to explaining how sendByteValue, the focus of this report would be on how this function was used in the run() function of the send thread. Since this is where the bit stream communication is truly implemented.
In the run() function, initially the startFlag is used to make sure only the transferPackage() function can activate the send thread, which is to prevent multiple sources of data in the send thread.
This in turn stops noise and other glitches from occurring in the Receive Window as it prevents data corruption.

```
while(!startFlag)
        {
            usleep(5);
        }
        dataPackage pkg =this->pkgRecieve;
```

If startFlag is true, first startFlag is again set to false and then sendByteValue is called with the parameter of the command given by the send window. After sendByteValue is used to send the size of the data or parameter of the command.

```
    startFlag = false;
    sendByteValue(pkg.command);
    int size = pkg.data.size();
    sendByteValue(size);
```

Once both the command and the size of the parameters to the receive thread, the sendByteValue is used to send the data variable of the dataPackage structure byte by byte. Since the data variable of the dataPackage structure is a QByteArray hence it is possible to input each of the QByteArray's elements into the sendByteValue() function using a form of fixed iteration.

```
for(int k=0;k<size;k++)
        {
          sendByteValue(pkg.data.at(k));
        }
```

4.2. Decoding the Bit Stream into Bytes in the Receive thread

Initially the receive thread is activated in the main class of the program by start function and so the start function in turn calls the run() function which activates the receive thread.
The data the receive thread needs to output to the receive window is the same. So the run() function's main role,in this rendition of the code, is decode the bit stream into bytes and then use the bytes to initialise the variables of the local dataPackage structure. This dataPackage structure will be emitted to the receive window to output.
At the start of the execution of the run() file the value that was received bit by bit in the command of that the send thread send. This command is used to initialise the command part of the dataPackage structure. After that recvByte() is used to initialise the number of the bytes are transmitted as the parameters of the subsequent command in a variable known as dataSize.

```
        dataPackage pkg;
        pkg.command = recvByte();
        int size = recvByte();
```

This dataSize variable is used to limit the amount of times recvByte() is used to get the data part of the instruction. This prevents the dataPackage from being too large and stops unnecessary noise from being outputted into the Receive window.

```
  for(int k=0;k<size;k++)
        {
            pkg.data.append(recvByte());
        }
```

Just like 3.3 the cmdFinished and conditional iteration is used to stop more dataPackage structures from being emitted whilst receive window completes the previously given command by making the receive thread 'sleep' for a small amount of time.

```
emit transferPackageToRecieve(&pkg);
    while(!cmdFinished)
    {
        usleep(3);
    }
   cmdFinished = false;
```

4.3. Understanding the sendByteValue() and the recvByte() functions

The sendBytevalue() function and the recvByte() functions work hand in hand with one other.
By taking advantage of the multi-threaded implementation, they can 'switch' between actively and passively running. This done by changing the cmdRead Boolean parameter in order to switch between sending and recieving a bit.
Below it shows the definition of both the functions,sendByteValue() and recvByte() respectively.
The algortihm for making a byte convert to a series of bits needs to be iterated eight times since there are eight bits in a byte. The bit can only be sent when cmdRead is true and so using by using while loop, the function can be paused until the cmdRead is true. In order to only have one thread access the global variables at once, globalMutex is used to block the corresponding thread. The globalBinaryData is the shared boolean value which is used to encode the dataPackage into a bitstream.The globalBinaryData is initialised by having the input Byte get bitwise ANDed by the number 8 in base 16 (or hexadecimal) form. The initial input is doubled (by being bit shifted to the right) and the cmdRead function is set to false to allow the recByte() function to recieve the bit via allowing to access the globalBinaryData variable. Now the sendByteValue() function is running passively whilst the recvByte() function is running actively.

```
    void OutputThread::sendByteValue
            (quint8 localByteData)
{
    for (int i = 0; i < 8; i++) {
        while(cmdRead == false);
            globalMutex.lock();
            globalBinaryData=
                localByteData &0x80;
            localByteData <<=1;
            cmdRead = false;
            globalMutex.unlock();
    }
}
```

The output quint8 variable has been declared as data and initialised as zero.The algorithm for turning a series of 8 bits into a Byte has to be iterated 8 times since there are 8 bits in a Byte. Since when in the receive thread the new command has not been read yet, the cmdRead has to be false in order to break out of the infinite loop. The data output variable is doubled by being right bit shifted. The output is bitwise ORed by the globalBinaryData.In order to change the cmdRead Boolean, the globalMutex is locked. The cmdRead is assigned as true so another bit can be sent. The globalMutex is unlocked.Now the recvByte() function is running passively whilst the sendByteValue() function is running actively.

```
quint8 InputThread::recvByte()
        { quint8 data = 0;
            for (int i = 0; i < 8; i++)
            {
                for(;;)
                { if(cmdRead != true)
                        {
                        break;
                        }
                }
                data <<=1;
                data|=globalBinaryData;
                globalMutex.lock();
                cmdRead = true;
                globalMutex.unlock();
            }
    return data; }
```

The two function switch between active and passive execution until recvByte() outputs a Byte and leaves the For loop.

5. The Output of the program

5.1. Drawing

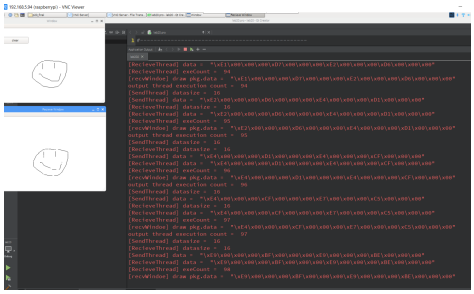

Figure 1: This is the drawing screenshot

5.2. Resizing

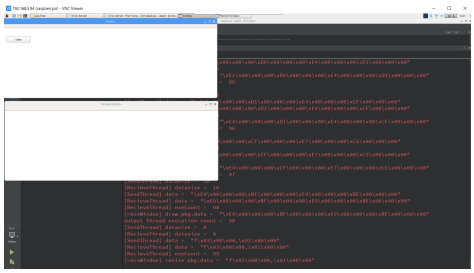

Figure 2: This is the resizing screenshot

5.3. Clearing

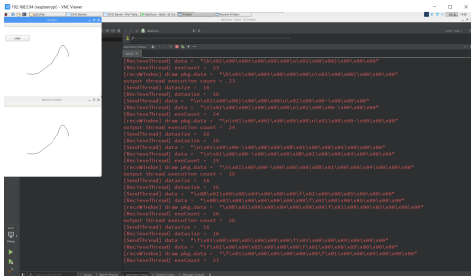

Figure 3: This is before clear pressed

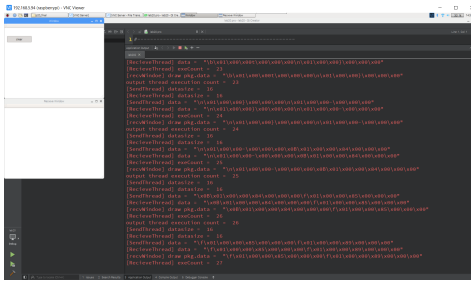

Figure 4: This is after clear pressed