Whiteboard Chat Applictaion with Simplex Communication

Joesph Butterworth
jdb1g20@soton.ac.uk
MEng Electrical and Electronic Engineering
Personal Tutor: Tracy Melvin

Abstract:

1. Introduction

2. Theory

2.1. QPainter

How will the send-window allow users to draw diagrams? How will it display diagrams as they are being drawn, and how will it retain these diagrams so that they don't disappear when the window is repainted?

QPainter is a part of the qt library family that allows you to draw pixel graphics. For this application, QPainter will be used to modify a QImage.

2.2. Threads and Mutexes

Threads - threads are a set of calculations run in the order of using a process. Threads are run in parallel meaning more operations can be run at any given time.

pthreads (POSIX threads) - is a library in C style languages that allows for thread control.This allows multiple processes to be run simultaneously in a program

QThread - is a class that uses pthreads in qt create. It has member functions that enable access to pthread functions in an object orientated method.

How will you use threads to send and receive these packets, while the rest of the application keeps running? How will you use mutexes to make any relevant collections "thread-safe"?

2.3. Serialisation

How will you serialize these commands into packets to be sent from the send window?

How will you convert binary packets into a stream of 1's and 0's? How will you transmit this stream in a reliable way? For example, you may need to signal when a bit is ready to be read, and when the receive window has finished reading the current bit.

2.4. Simplex, Half-Duplex, and Duplex Communication

How will you represent the drawing commands so that they can be sent to the other user whilst they are being drawn?

How will you receive and buffer packets at the other end? How will you deserialize them? How will you draw them on the receive window? How will you retain the currently received diagram so that when the window is repainted the diagram isn't lost?

2.5. Class Architecture of the Whiteboard Applicatoin

3. Implementation

3.1. The GUI - Send and Receive Windows

On initiasation, the program creates new instances of the sendWindow and receiveWindow classes. The new windows are then given defined parameters, they are named and resized to a proportion of the screen they are being displayed on.

```
QScreen *screen =
       QGuiApplication::primaryScreen();
QRect screenGeometry = screen->geometry();
int height = screenGeometry.height();
int width = screenGeometry.width();

// Creating send window
QPixmap sendpix(QDir::currentPath()
```

```
        + "/Icons/send_icon.png");

sendWindow sendW(nullptr);
sendW.setWindowIcon(sendpix);
sendW.setWindowTitle("Send␣Window");
sendW.resize(width/3, height/2);
sendW.move(width/9, height/4);
sendW.show();
```

The sendWindow and receiveWindow classes inhereted from QMainWindow. To enable either the user or the draw commands the central widgets were set to be their respective canvas classes, sendCanvas and receiveCanvas.

```
sendCanvas::sendCanvas(QWidget *parent)
        : QWidget(parent)
```

The central canvases would draw by taking the position of an input, for the send window a mouse click and for the receive window the position data passed to it by the send window, and drawing a rectangle between them. For the send window the mousePressEvent(), mouseMoveEvent() and mouseReleaseEvent() were all overwritten. The program is able to draw curves and circles owing to the mouseMoveEvent() function being overwritten, this receives a lot of information from the mouse as it is constantly polling, hence all the rectangles are really small and to create a smooth line.

```
QPainter painter(&drawImage);

// Create pen and draw line
QPen newPen(areaBrushStyle, areaPenWidth,
        areaPenStyle, areaCapStyle,
        Qt::RoundJoin);
newPen.setColor(areaColour);
painter.setPen(newPen);
painter.drawLine(prevPoint, endPoint);

// Maintain radius whilst drawing
int radius = (areaPenWidth / 2) + 2;
update(QRect(prevPoint, endPoint)
        .normalized().adjusted(-radius,
         -radius, +radius, +radius));

// Update point
prevPoint = endPoint;
```

When using the drawLine() function the program uses QPainter to draw a line from the last known position to the new position. It maintains the previous drawings by loading an image of the previous inputs, the private variable drawImage, as the canvas and overlaying the new rectangle on top of that.

In addition to the central UI of the canvas, the send window also has a menubar and toolbar that allows users to easily interact with the program as in Figure 2, in an easy to use fashion. The user inputs used the QMenubar and QToolbar functionality, to easily add buttons to the menubar and toolbar.

```
// Toolbar
auto *colour = new QAction("&Colour", this);
auto *width = new QAction("&Pen␣Width", this);

QToolBar *toolbar =
        addToolBar("main␣toolbar");
toolbar->addAction(colour);
toolbar->addSeparator();
toolbar->addAction(width);

connect(colour, &QAction::triggered, this,
        &sendWindow::colour);
connect(width, &QAction::triggered, this,
        &sendWindow::penWidth);
```

These buttons were connected to member functions, that used the QFileDialog, QInputDialog and QColorDialog, this allowed for effective handling of opening files, saving the screen as an image, getting the pen width and the pen colour respectively, shown in Figures 3 & 4. Using the Qt library family allowed the program to use dialog boxes in a way that would be familiar to any user.

```
// Get image file formats
const QList<QByteArray> imageFormats =
        QImageWriter::supportedImageFormats();

// Create file filter from QList
QString fileFilter = "";
for(const QByteArray &format : imageFormats)
{
     fileFilter.append(tr("%1␣Files␣(*.%2)")
        .arg(QString::fromLatin1(format)
        .toUpper(),
        QString::fromLatin1(format)));
    fileFilter.append(";;");
}
//qDebug() << fileFilter;

// Set filename from dialog box
QString fileName =
        QFileDialog::getSaveFileName(this,
        tr("Save␣File"), QDir::currentPath()
```

```
        + "/untitled", fileFilter);

// Set fileformat from dialog box
char *format = fileName.split(".").last()
        .toUtf8().data();

// Pass to draw area function
if (!fileName.isEmpty())
        canvas->saveArea(fileName, format);
```

3.2. The Serialise and Deserialise Drawing Commands

To pass the positional data between classes a struct using uint16_ts was used. This was very similar to passing a QPoint between objects. However, this limited the size of the information, so that it is larger than the screen is likely to be but to smaller than passing entire ints across.

```
typedef struct drawInfoPosition
{
    // Opcodes
    // Bit 1 for odd-on parity
    uint8_t opcode;

    // Vertical and Horizontal Positions
    uint16_t xPosition;
    uint16_t yPosition;

} drawInfoPosition;
```

The drawInfoPosition struct could easily be serialised by loopiing over every element and pushing it to a boolean array. This is now the sequential series that can be pushed across a pin or onto a threadsafe queue.

```
uint16_t temp[3] = {serialData.opcode,
        serialData.xPosition,
        serialData.yPosition};
bool serialsedArray[48];

// Loop over each element of the
//temporary array
for(int i = 0; i < 3; i++)
{
    for(int j = 15; j>= 0; j--)
    {
        // Convert to binary
        if (temp[i] >= pow(2, j))
       {
            temp[i] -= pow(2, j);

            // Increase Local Count
            serialsedArray[i*j+j] = true;
        }
    }
}
```

Likewise the boolean array can be deserialised when it is received back into a drawInfoPosition struct, from which a QPoint can be constructed.

```
drawInfoPosition serialData;
uint16_t temp[3] = {serialData.opcode,
        serialData.xPosition,
        serialData.yPosition};

// Loop over each element of the
//temporary array
for(int i = 0; i < 3; i++)
{
    for(int j = 15; j>= 0; j--)
   {
        // Convert to decimal
        if (serialsedArray[i*i+j])
        {
            temp[i] += pow(2,j);
         }
    }
}
```

3.3. The Send and Receive Threads

To pass the serialised information across without using a large thread overhead worker threads have to be used. As the send and receive canvases are the classes that use the positional information it would be effeceint to send the information directly between them. Hence, the worker threads to the send and receive window should be instantiated in the send and receive canvases.

```
sendThread *worker = new sendThread();
worker->moveToThread(&sender);
connect(&sender, &QThread::finished, worker,
        &QObject::deleteLater);
connect(this,
        SIGNAL(drawSignal(drawInfoPosition)),
        worker, SLOT(pushSerialStruct(
        drawInfoPosition)));
sender.start();
```

The send and receive threads cannot directly communicate with each other using signals and slots as they cannot be referenced to each other. As such they require an entity in common that they may both interact with. To allow this communication a thread safe queue class template was to be implemented to allow easy communication between the threads.

However, the threadsafe queue that was implemented failed to initialise correctly and either created a compile error *undefiened reference to vtable* or worse would compile then give a SIGSEGV error when opening. A large portion of the time spent on this project was spent trying to resolve this issue. The errors in the queue class template were never resolved. Therefore, the send and receive thread were never able to communicate.

3.4. Communication Protocol Using Booleans

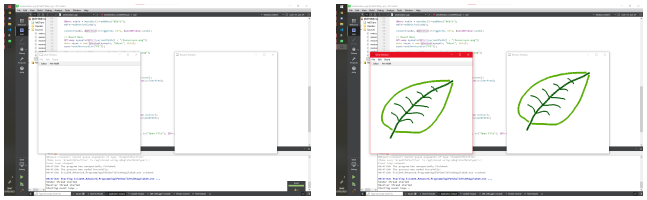
Despite the inability for the threads to communicate there was still progress in creating a communication protocol between them. For a communication to occur there was to be shared bits in the queue class that could be toggled by either thread. One to say the data packet had been received and one to say that the parity check had failed.

The send thread was going to set the first bit in the serialData.opcode if there was an odd number of bits, so that there was always an even number of bits in the transmition. It would do this by looping over the entire drawInfoPosition struct and keep a count of every set bit, if that count was zero it would then perform a bitwise OR operation to set the parity bit.

```
void sendThread::setParityBit(drawInfoPosition
        serialData)
{
    int bitCount = 0;
    uint16_t temp[3] = {serialData.opcode,
        serialData.xPosition,
        serialData.yPosition};

    // Loop over each element of the
    //temporary array
    for(int i = 0; i < 3; i++)
    {
        for(int j = 15; j>= 0; j--)
        {
            // Convert to binary
            if (temp[i] >= pow(2, j))
            {
                temp[i] -= pow(2, j);

                // Increase Local Count
                bitCount++;
            }
        }
    }

    // If odd number of bits
    if(bitCount % 2 != 0)
    {
        // Set Parity Bit
        serialData.opcode |= (1 << 7);
    }
}
```

The receive thread would perform a similar operation, counting every bit in the drawInfoPosition it received and setting the paritycheckfailed bit if there were an odd number of bits.

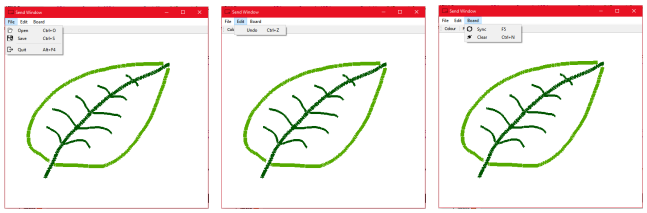
4. Final Application



(a) Send and receive windows when application is opened

(b) Send and receive windows after drawing

Figure 1: Send and receive windows open on screen and allow drawing between them

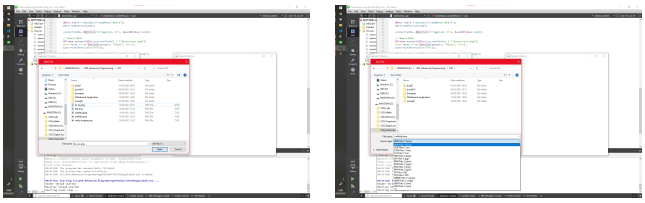


(a) File Menu with opening saving and closing

(b) Edit Menu with the undo functionality, redo was to be added later

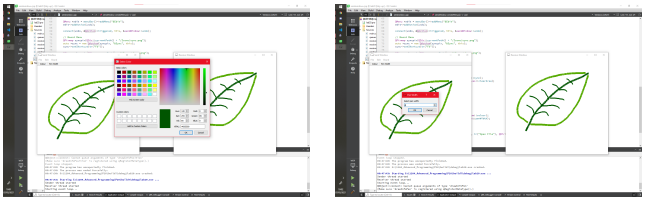
(c) Board menu allows the screen to be cleared or windows to be synced

Figure 2: Menus allow the user to control the program



(a) Open file dialog (b) Save file dialog

Figure 3: File handling dialogs using QFileDialog



(a) Colour selection dialog box (b) Pen width control dialog box

Figure 4: Dialog boxes to control the QPen characteristics

5. Discussion

5.1. GUI - Send and Receive Windows

The send and receive window were initiasised in a relatively standard way. However, the use of getting the screen geometry information allowed the program to scale better across different users. This program was written on a 1080p monitor, a visually pleasing size on this display is likely to be too small for any end user that has a 2160p monitor.

The send window was well featured with the ability to change the pen and, save and load the canvas as images. This gave the send window a feature set similar to a simple paint application. This is a good range of utility for a whiteboard application to be able to do.

In addition, the layout of features is similar in format to other contemporary desktop applications. This makes the program intuitive and easy to use as any user will know where to look for the tool that they are looking for.

The addition of an undo feature is a nice quality of life feature that makes the application more pleasurable to use. This was implemented by snapshoting the draw area image every time the mouse was released. The lack of a redo feature is apparent. The redo functionality could be implemented using a lot of the same code as the undo feature.

5.2. The Serialise and Deserialise Drawing Commands

The method to serialise and deserialise the drawing commands is a relatively common method in C++ programming. This makes it a reliable method as it is unlikely to break from a new version of Qt being used. In addition, this means that if there are found to be any issues they are likely to be well documented reducing the time needed to fix the issue. However, not using any features from Qt or Vector means that the code is not as succinct as it otherwise may have been.

Furthermore, the use of the drawInfoPosition struct is another instance of bloat in the code, as any conversion from QPoint or QPen could be carried out at the serialisation stage. This is the result of previous implementation of drawing and could be streamlined. Although bloated, this does not apppear to have anylarge detrement to the code itself.

5.3. The Send and Receive Threads

The send and receive threads successfully initialised inside the instances of the canvases. The canvas was able to communicate with the thread using signals and slots. This meant that information could easily be passed down into the thread to go to the receive window.

The use of QMutex was a useful way of ensuring that functions were thread safe. QMutex meant that a lot of the the complicated threading issues were conveniently handled by the program. This removed a level of risk from a race condition occuring.

The lack of a functioning queue was a severe detrement to this project. Lacking a queue there was no common object that the send and receive threads could communicate through. This made the threads practically pointless as they were unable to fulfill the communication they were there to enable.

Althought the implemented queue appeared to be thread safe and controlled the items put onto and pulled off of it correctly the inability to compile it with the rest of the program suggests that there is likely a problem with the use of QMutex in the class.

5.4. Communicaiton Protocol Using Booleans

Parity calculations are a useful way of checking that the received information has been received correctly. The use of parity meant that the communication between threads would have been consided a half-duplex protocol.

The parity calculations used in the implemented

protocol are relatively simple and do not accaount for two bit flips in a signal. Nor does the parity method used in this protocol tell you what part of the information has an error. However, this is likely to be sufficient for the communication between two devices with GPIO as there is likely to be very little noise on the short wired connection.

The lack of a functioning queue was again an issue. As there was no shared class between the threads so they were unable to flip shared bits, emulating the role of a GPIO pin.

6. Conclusion