جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

ANNEE UNIVERSITAIRE

2024-2025

الكلية المتعددة التخصصات بالعرائش
FACULTE POLYDISCIPLINAIRE A LARACHE

Miniprojet 1 Group 1

Tronc Commun National

Info.Appliquée

Développement d'une application de messagerie

Réalise par :

Chichi Ilyas SMI0062/23

Ali kaioua SMI0072/22

Abdellatif Ait Ouakrim SMI0040/23

sous la supervision de :

Pr. Essaid Elhaji

Présentation du projet :

Notre projet consiste à développer une application de messagerie instantanée en architecture client-serveur. L'objectif est de permettre aux utilisateurs de s'échanger des messages en temps réel via le protocole TCP/IP. Ce document présente les principales caractéristiques et fonctionnalités attendues de l'application ainsi que les étapes à suivre pour sa réalisation.

Fonctionnalités principales :

1. **Connexion utilisateur**

   L'application doit permettre aux utilisateurs de se connecter à un serveur centralisé à l'aide de leurs identifiants. Une fois connectés, ils doivent pouvoir accéder à leur liste de contacts et débuter des conversations.

2. **Envoi et réception de messages**

   Les messages doivent être envoyés et reçus en temps réel. L'application doit gérer les différents formats de messages, notamment le texte, les images et les fichiers.

3. **Notifications en temps réel**

   L'application n'envoie pas de notifications aux utilisateurs pour les messages reçus.

4. **Historique des conversations**

   L'historique des conversations est limité à la session en cours, permettant aux utilisateurs de consulter uniquement les messages récents.

5. **Sécurité des données**

   Toutes les communications doivent être sécurisées par chiffrement pour protéger la confidentialité des utilisateurs.

Étapes de développement :

1. **Analyse des besoins :**

   Recueillir et documenter les exigences fonctionnelles et non fonctionnelles de l'application.

2. **Conception de l'architecture :**

   Concevoir la structure de l'application, y compris les schémas de base de données et les interfaces utilisateur.

3. **Développement :**

   Programmer les fonctionnalités de l'application client et serveur en suivant les spécifications techniques.

4. **Tests :**

   Effectuer des tests unitaires et d'intégration pour s'assurer que chaque composant fonctionne correctement.

5. **Déploiement :**

   Mettre l'application en production et assurer la maintenance continue pour résoudre les bugs et améliorer les performances.

6. **Support et maintenance :**

   Fournir une assistance aux utilisateurs et effectuer des mises à jour régulières pour ajouter de nouvelles fonctionnalités.

Architecture client-serveur :

Composants :

- **Client** : L'application installée sur l'appareil de l'utilisateur. Elle se connecte au serveur pour envoyer et recevoir des messages.
- **Serveur** : Une application centrale qui gère les connexions des utilisateurs, l'acheminement des messages et le stockage des données.

Protocole de communication :

L'application utilisera le protocole TCP/IP pour assurer une communication fiable entre le client et le serveur. Ce protocole garantit que les messages sont reçus dans l'ordre dans lequel ils sont envoyés et sans pertes.

Socket :

Un socket est un point de communication bidirectionnel permettant à des processus (locaux ou distants) de nous échanger des données via un protocole (TCP, UDP, etc.).

Analogie :

Un socket, c’est l’alliance d’une IP (immeuble) et d’un port (boîte aux lettres), permettant au kernel (Le facteur) de savoir quel processus (propriétaire d'une maison) doit recevoir quel paquet.

Introduction aux Bibliothèques Qt : QTcpServer, QTcpSocket, et QThread

Qt est un cadre de développement très populaire pour créer des applications multiplateformes. Parmi ses nombreuses fonctionnalités, Qt offre plusieurs bibliothèques utiles pour la programmation réseau et de threads. Dans cet article, nous explorerons les bibliothèques **QTcpServer**, **QTcpSocket**, et **QThread**.

QTcpServer

QTcpServer est une classe qui permet de créer des applications serveur TCP. Elle est utilisée pour écouter les connexions entrantes sur un port spécifié et accepter ces connexions. Voici quelques fonctionnalités clés de **QTcpServer** :

- **Écoute des connexions** : L'application peut écouter sur un port spécifique pour des connexions entrantes.
- **Gestion des connexions multiples** : Elle peut gérer de multiples connexions entrantes simultanément.
- **Facilité d'utilisation** : Avec des signaux et des slots, elle permet une gestion efficace des événements.

QTcpSocket

QTcpSocket fournit les fonctionnalités nécessaires pour établir une connexion TCP à partir d'un client. Voici quelques caractéristiques importantes de **QTcpSocket** :

- **Connexion à un serveur** : Il permet d'établir une connexion avec un serveur en utilisant une adresse IP et un numéro de port.
- **Échange de données** : Envoie et reçoit des flux de données.
- **Gestion des erreurs** : Offre des mécanismes pour détecter et gérer les erreurs de connexion.

QThread

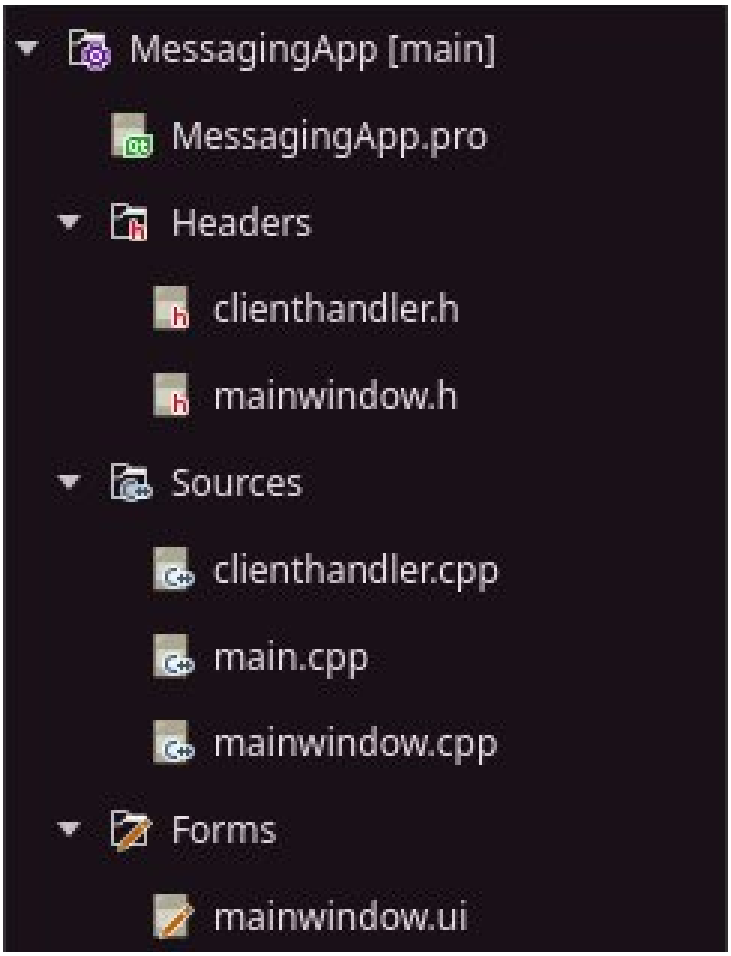
QThread est une classe qui permet de gérer des threads, c'est-à-dire des processus d'exécution parallèles dans une application Qt. Voici quelques usages principaux de **QThread** :

- **Exécution parallèle** : Elle permet d'effectuer des tâches en parallèle pour améliorer la performance de l'application.
- **Gestion des signaux et slots** : En tant que partie intégrante du système d'événements de Qt, elle permet une communication efficace entre les threads.
- **Synchronisation** : Facilite la synchronisation des données entre différents threads.

Implémentation

Serveur:

hiérarchie:



Le serveur consite à une fenêtre principale mainwindow, fichier main content la fonction main qui joue le role de créer la fenêtre principale utilisant QApplication, et clienthandler qui gére la relation client serveur.

main.cpp

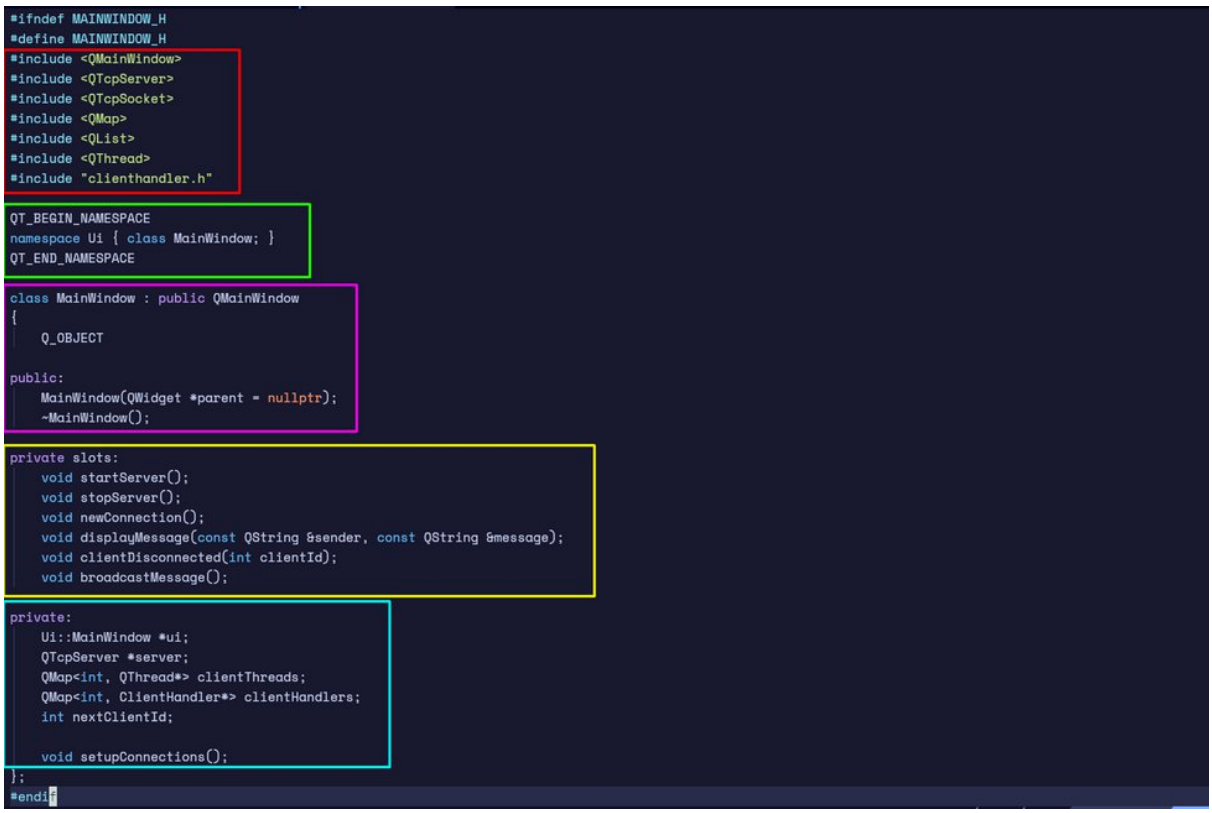
C’est la fonction plus simple dans le cote serveur, il sert à créer une objet QApplication et une objet Mainwindow w, on utilise la method w.show() pour afficher cette fenêtre et la ligne return a.exec(); laisse l’application affichée jusqu’il reçoit un signale de fermeture.

```cpp
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

mainwindow.cpp / mainwindow.h / mainwindow.ui

C’est la fenêtre d’affichage de serveur, il consiste à 3 fichiers, un fichier header qui contient les variables, les prototypes des fonctions et les classes, un fichier source qui définie ces fonctions et ces variables et un fichier ui qui gére le design de fenêtre

mainwindow.h



```h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QTcpServer>
#include <QTcpSocket>
#include <QMap>
#include <QList>
#include <QThread>
#include "clienthandler.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void startServer();
    void stopServer();
    void newConnection();
    void displayMessage(const QString &sender, const QString &message);
    void clientDisconnected(int clientId);
    void broadcastMessage();

private:
    Ui::MainWindow *ui;
    QTcpServer *server;
    QMap<int, QThread*> clientThreads;
    QMap<int, ClientHandler*> clientHandlers;
    int nextClientId;

    void setupConnections();
};
#endif
```

//////////////////////////

La section mis en mis en évidence en rouge signifie les bibliothéques utilisés dans ce cas on a utilisé **QMainWindow QTcpServer QTcpSocket QMap QList QThread** et **clienthandler.h** qui est le fichier header de clienthandler, since on prend certain méthodes de ce fichier

//////////////////////////

La section mis en évidence en vert signifie le namespace Ui, c'est le namespace pour les boutons, slots et les signales

//////////////////////////

La section mis en évidence en rose signifie le class de fenêtre principale, sont constructeur et son destructeur

//////////////////

La section mis en évidence en jaune signifie les méthodes et les signales associés à des boutons implémentés dans la fenêtre principale

////////////

La section mis en évidence en turquoise signifie les objets et les variables utilisée ainsi que la fonction setupConnections() qui joue le role de montage des connections

mainwindow.ui/mainwindow.cpp

Méthodes utilisée

Constructeur & Déstructeur

```cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
    , nextClientId(1)
{
    ui->setupUi(this);
    server = new QTcpServer(this);

    setupConnections();
}

MainWindow::~MainWindow()
{
    for (auto thread : clientThreads.values()) {
        thread->quit();
        thread->wait();
    }

    qDeleteAll(clientThreads);
    qDeleteAll(clientHandlers);

    delete ui;
}
```

Liaison des boutons et leurs fonctionalités

```cpp
void MainWindow::setupConnections()
{
    connect(ui->startButton, &QPushButton::clicked, this, &MainWindow::startServer);
    connect(ui->stopButton, &QPushButton::clicked, this, &MainWindow::stopServer);
    connect(ui->sendButton, &QPushButton::clicked, this, &MainWindow::broadcastMessage);

    ui->stopButton->setEnabled(false);
}
```

fonctions responsables à commencer le serveur et l'arrêter

```cpp
void MainWindow::startServer()
{
    int port = ui->portSpinBox->value();

    if (!server->listen(QHostAddress::Any, port)) {
        QMessageBox::critical(this, tr("Server Error"),
                              tr("Unable to start the server: %1.")
                                  .arg(server->errorString()));
        return;
    }

    connect(server, &QTcpServer::newConnection, this, &MainWindow::newConnection);
    ui->statusLabel->setText(tr("Server is running on port %1").arg(port));
    ui->startButton->setEnabled(false);
    ui->stopButton->setEnabled(true);
    ui->portSpinBox->setEnabled(false);

    ui->chatDisplay->append(tr("Server started at %1")
                                .arg(QDateTime::currentDateTime().toString()));
}

void MainWindow::stopServer()
{

    for (auto handler : clientHandlers.values()) {
        handler->disconnectClient();
    }
    server->close();
    disconnect(server, &QTcpServer::newConnection, this, &MainWindow::newConnection);

    ui->statusLabel->setText(tr("Server is stopped"));
    ui->startButton->setEnabled(true);
    ui->stopButton->setEnabled(false);
    ui->portSpinBox->setEnabled(true);

    ui->chatDisplay->append(tr("Server stopped at %1")
                                .arg(QDateTime::currentDateTime().toString()));
}
```

fonction qui gère une nouvelle connection

```
void MainWindow::newConnection()
{
    QTcpSocket *clientSocket = server->nextPendingConnection();
    int clientId = nextClientId++;

    QThread *clientThread = new QThread();
    clientThreads[clientId] = clientThread;


    ClientHandler *handler = new ClientHandler(clientId, clientSocket);
    clientHandlers[clientId] = handler;


    handler->moveToThread(clientThread);


    connect(handler, &ClientHandler::messageReceived,
            this, &MainWindow::displayMessage);
    connect(handler, &ClientHandler::clientDisconnected,
            this, &MainWindow::clientDisconnected);
    connect(clientThread, &QThread::finished,
            handler, &QObject::deleteLater);


    clientThread->start();

    ui->chatDisplay->append(tr("New client connected (ID: %1) at %2")
                            .arg(clientId)
                            .arg(QDateTime::currentDateTime().toString()));
}
```

fonction qui gère une déconnection

```
void MainWindow::clientDisconnected(int clientId)
{

    if (clientThreads.contains(clientId)) {
        QThread *thread = clientThreads.take(clientId);
        thread->quit();
        thread->wait();
        delete thread;
    }


    if (clientHandlers.contains(clientId)) {
        clientHandlers.remove(clientId);
    }

    ui->chatDisplay->append(tr("Client disconnected (ID: %1) at %2")
                                .arg(clientId)
                                .arg(QDateTime::currentDateTime().toString()));
}
```

fonction d’affichage des messages

```
void MainWindow::displayMessage(const QString &sender, const QString &message)
{
    QString formattedMessage = QString("[%1] %2: %3")
    .arg(QDateTime::currentDateTime().toString("hh:mm:ss"))
        .arg(sender)
        .arg(message);

    ui->chatDisplay->append(formattedMessage);
}
```

fonction d’envoi d’un message

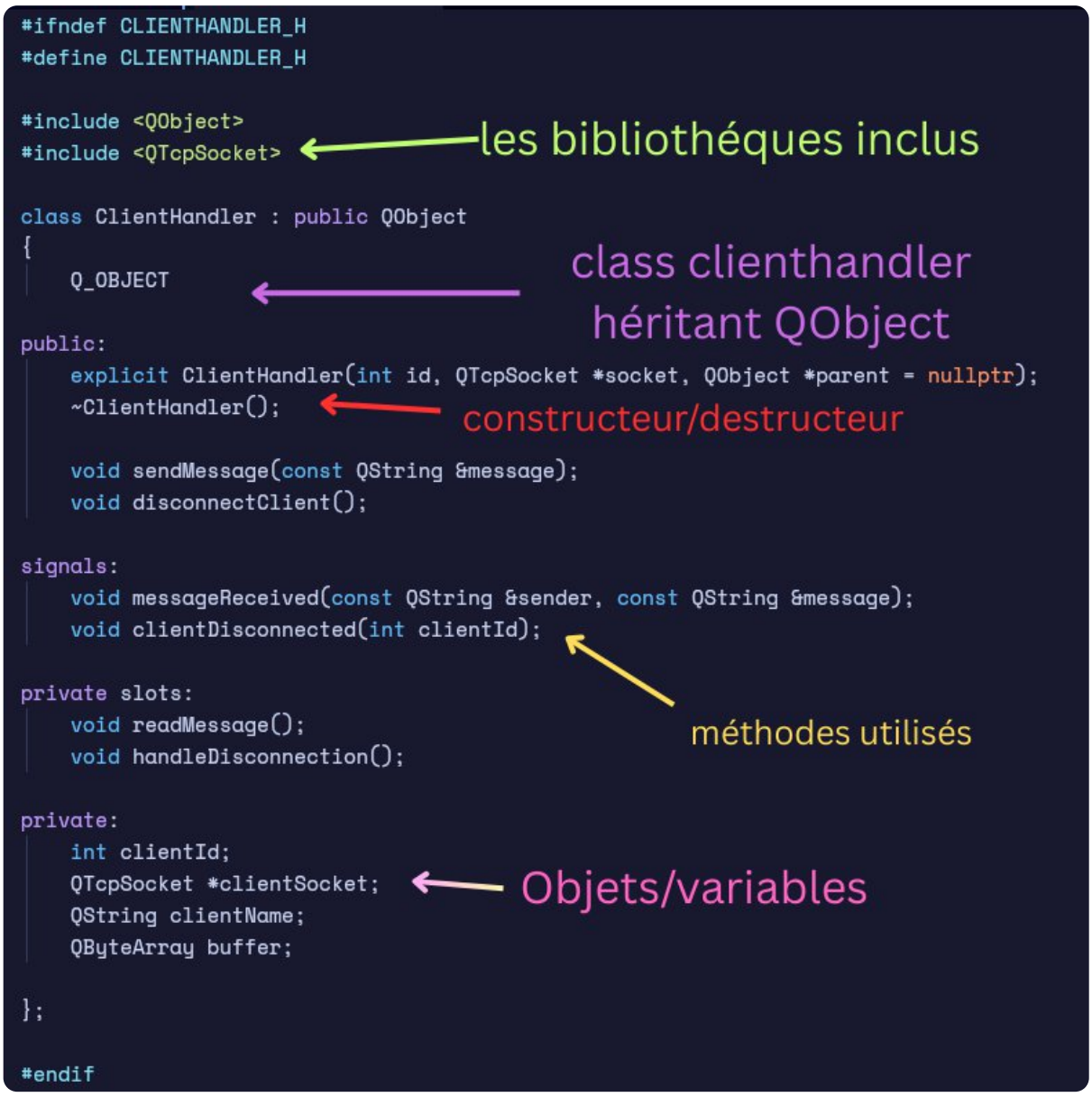
```
void MainWindow::broadcastMessage()
{
    QString message = ui->messageInput->text().trimmed();

    if (message.isEmpty()) {
        return;
    }


    displayMessage("Server", message);


    for (auto handler : clientHandlers.values()) {
        handler->sendMessage(message);
    }


    ui->messageInput->clear();
}
```

en générale la philosophie utilisé est de remplir le variable port par la valeur entrée dans le box specifié et créer un serveur hosté dans l'adresse ip locale, on a implementé une fonction lorsqu'on clique la bouton "start server" il la grisaille et avant cliquer ce bouton, "stop server" est grisaillée de méme on a implementé un string pour afficher le status de serveur dans tous le temps.

clienthandler.cpp / clienthandler.h

Ce code gère la communication avec un seul client connecté. Il gère les messages entrants, permettant aux clients de définir leur nom via /name, et émet un signal messageReceived contenant le nom et le message du client. Il gère également les déconnexions client et permet d'envoyer des messages au client. Le code prend en charge deux formats de message : texte délimité par des sauts de ligne et format binaire préfixé par la taille pour les données plus complexes.

clienthandler.h




```h
#ifndef CLIENTHANDLER_H
#define CLIENTHANDLER_H

#include <QObject>
#include <QTcpSocket>

class ClientHandler : public QObject
{
    Q_OBJECT

public:
    explicit ClientHandler(int id, QTcpSocket *socket, QObject *parent = nullptr);
    ~ClientHandler();

    void sendMessage(const QString &message);
    void disconnectClient();

signals:
    void messageReceived(const QString &sender, const QString &message);
    void clientDisconnected(int clientId);

private slots:
    void readMessage();
    void handleDisconnection();

private:
    int clientId;
    QTcpSocket *clientSocket;
    QString clientName;
    QByteArray buffer;

};

#endif
```

clienthandler.cpp

constructeur & destructeur:

```cpp
ClientHandler::ClientHandler(int id, QTcpSocket *socket, QObject *parent)
    : QObject(parent)
    , clientId(id)
    , clientSocket(socket)
    , clientName(QString("Client_%1").arg(id))
{
    connect(clientSocket, &QTcpSocket::readyRead,
            this, &ClientHandler::readMessage);
    connect(clientSocket, &QTcpSocket::disconnected,
            this, &ClientHandler::handleDisconnection);
}

ClientHandler::~ClientHandler()
{
    if (clientSocket) {
        clientSocket->disconnectFromHost();
        clientSocket->deleteLater();
    }
}
```

fonction responsable de lire les messages:

```cpp
void ClientHandler::readMessage()
{
    QByteArray data = clientSocket->readAll();
    buffer.append(data);

    while (true) {
        int newlineIndex = buffer.indexOf('\n');

        if (newlineIndex != -1) {
            QByteArray rawLine = buffer.left(newlineIndex).trimmed();
            buffer.remove(0, newlineIndex + 1);
            QString message = QString::fromUtf8(rawLine);
            if (message.startsWith("/name ")) {
                clientName = message.mid(6).trimmed();
                if (clientName.isEmpty()) {
                    clientName = QString("Client_%1").arg(clientId);
                }
                continue;
            }

            emit messageReceived(clientName, message);
            continue;
        }

        if (buffer.size() < static_cast<int>(sizeof(quint32))) {
            return;
        }
```

```
        QDataStream in(&buffer, QIODevice::ReadOnly);

        in.setVersion(QDataStream::Qt_5_15);

        in.device()->seek(0);

        quint32 blockSize;
        in >> blockSize;

        if (buffer.size() < static_cast<int>(sizeof(quint32) + blockSize)) {
            return;
        }
        QString message;

        in >> message;

        buffer.remove(0, sizeof(quint32) + blockSize);

        if (message.startsWith("/name ")) {
            clientName = message.mid(6).trimmed();
            if (clientName.isEmpty()) {
                clientName = QString("Client_%1").arg(clientId);
            }
            continue;
        }

        emit messageReceived(clientName, message);

    }
}
```

en essence la fonction de lire les messages sert à lire deux types de messages, des messages génériques qui finissent avec un neuveau ligne \n et des messages de type QString qui sont mis en jeux pour la communication avec notre client et serveur, on a fait cette distinction pour qu'on peut communiquer avec autres clients et pas nécessairement notre client, cela est aprés un long débat de sécurité!

fonction responsable d'envoyer des messages

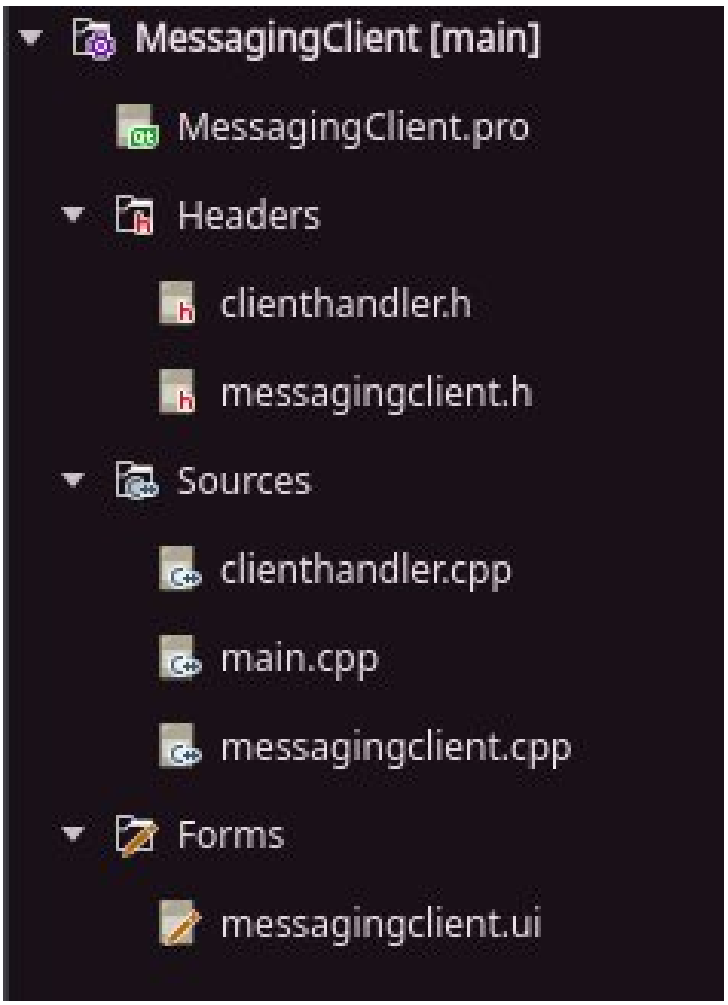
```cpp
void ClientHandler::sendMessage(const QString &message)
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_15);

    out << quint32(0);
    out << message;
    out.device()->seek(0);
    out << quint32(block.size() - sizeof(quint32));

    clientSocket->write(block);
}
```

fonctions responsables pour gérer une déconnection et déconnecter un client:

```cpp
void ClientHandler::handleDisconnection()
{
    emit clientDisconnected(clientId);
}

void ClientHandler::disconnectClient()
{
    if (clientSocket && clientSocket->state() == QAbstractSocket::ConnectedState) {
        clientSocket->disconnectFromHost();
    }
}
```

Client

Hiérarchie:



Le client consiste à des fichier header **clienthandler.h messagingclient.h** des fichier source **clienthandler.cpp main.cpp messagingclient.cpp** et un fichier .ui **messagingclient.ui**

REMARQUE:

le client utilise les mêmes principes que le serveurs pour clienthandler et le fichier main alors on va pas l'éxpliquer une autre fois, ils utilisent des fonctions IDENTIQUES au fonctions utilisés dans le serveur

messagingclient.h


```h
#ifndef MESSAGINGCLIENT_H
#define MESSAGINGCLIENT_H

#include <QMainWindow>
#include <QTcpSocket>

namespace Ui {
class MessagingClient;
}

class MessagingClient : public QMainWindow
{
    Q_OBJECT

public:
    explicit MessagingClient(QWidget *parent = nullptr);
    ~MessagingClient();

private slots:
    void connectToServer();
    void disconnectFromServer();
    void sendMessage();
    void readMessage();
    void displayError(QAbstractSocket::SocketError socketError);
    void connectionEstablished();
    void connectionClosed();

private:
    Ui::MessagingClient *ui;
    QTcpSocket *socket;
    quint32 blockSize;

    void setupConnections();
    void setConnectionStatus(bool connected);
};

#endif
```

messagingclient.cpp

Constructeur & Destructeur

```
MessagingClient::MessagingClient(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MessagingClient),
    blockSize(0)
{
    ui->setupUi(this);
    socket = new QTcpSocket(this);

    setupConnections();
    setConnectionStatus(false);
}

MessagingClient::~MessagingClient()
{
    delete ui;
}
```

Fonction responsable de la liaison des boutons et leurs fonctionalités

```
void MessagingClient::setupConnections()
{

    connect(ui->connectButton, SIGNAL(clicked()), this, SLOT(connectToServer()));
    connect(ui->disconnectButton, SIGNAL(clicked()), this, SLOT(disconnectFromServer()));
    connect(ui->sendButton, SIGNAL(clicked()), this, SLOT(sendMessage()));
    connect(ui->messageInput, SIGNAL(returnPressed()), this, SLOT(sendMessage()));


    connect(socket, SIGNAL(readyRead()), this, SLOT(readMessage()));
    connect(socket, SIGNAL(connected()), this, SLOT(connectionEstablished()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(connectionClosed()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)),
            this, SLOT(displayError(QAbstractSocket::SocketError)));
}
```

Fonctions responsables de connection et déconnection au serveur

```cpp
void MessagingClient::connectToServer()
{
    QString hostname = ui->serverInput->text();
    int port = ui->portSpinBox->value();
    socket->connectToHost(hostname, port);
}

void MessagingClient::disconnectFromServer()
{
    socket->disconnectFromHost();
}
```

Fonction responsable pour envoyer un message

```cpp
void MessagingClient::sendMessage()
{
    QString message = ui->messageInput->text().trimmed();

    if (message.isEmpty()) {
        return;
    }

    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_15);


    out << quint32(0);


    out << message;


    out.device()->seek(0);
    out << quint32(block.size() - sizeof(quint32));


    socket->write(block);


    ui->chatDisplay->append(QString("[%1] You: %2")
                                .arg(QDateTime::currentDateTime().toString("hh:mm:ss"))
                                .arg(message));



    ui->messageInput->clear();
}
```

comme expliquée dans le coté serveur, ici on peut voir que les messages envoyés par notre client sont sous la forme QString

Fonction responsable pour lire les messages

```
void MessagingClient::readMessage()
{
    QDataStream in(socket);
    in.setVersion(QDataStream::Qt_5_15);

    if (blockSize == 0) {

        if (socket->bytesAvailable() < static_cast<qint64>(sizeof(quint32))) {
            return;
        }

        in >> blockSize;
    }


    if (socket->bytesAvailable() < blockSize) {
        return;
    }

    QString message;
    in >> message;


    ui->chatDisplay->append(QString("[%1] Server: %2")
                                .arg(QDateTime::currentDateTime().toString("hh:mm:ss"))
                                .arg(message));


    blockSize = 0;
}
```

Fonction responsable pour la gestion des erreurs

```
void MessagingClient::displayError(QAbstractSocket::SocketError socketError)
{
    switch (socketError) {
    case QAbstractSocket::RemoteHostClosedError:
        break;
    case QAbstractSocket::HostNotFoundError:
        QMessageBox::information(this, tr("Client"),
                                 tr("The host was not found. Please check the host name and port settings."));
        break;
    case QAbstractSocket::ConnectionRefusedError:
        QMessageBox::information(this, tr("Client"),
                                 tr("The connection was refused by the peer. Make sure the server is running."));
        break;
    default:
        QMessageBox::information(this, tr("Client"),
                                 tr("The following error occurred: %1.")
                                     .arg(socket->errorString()));
    }

    setConnectionStatus(false);
}
```

Fonction responsable pour l’établissement d’une connection et la fermeture

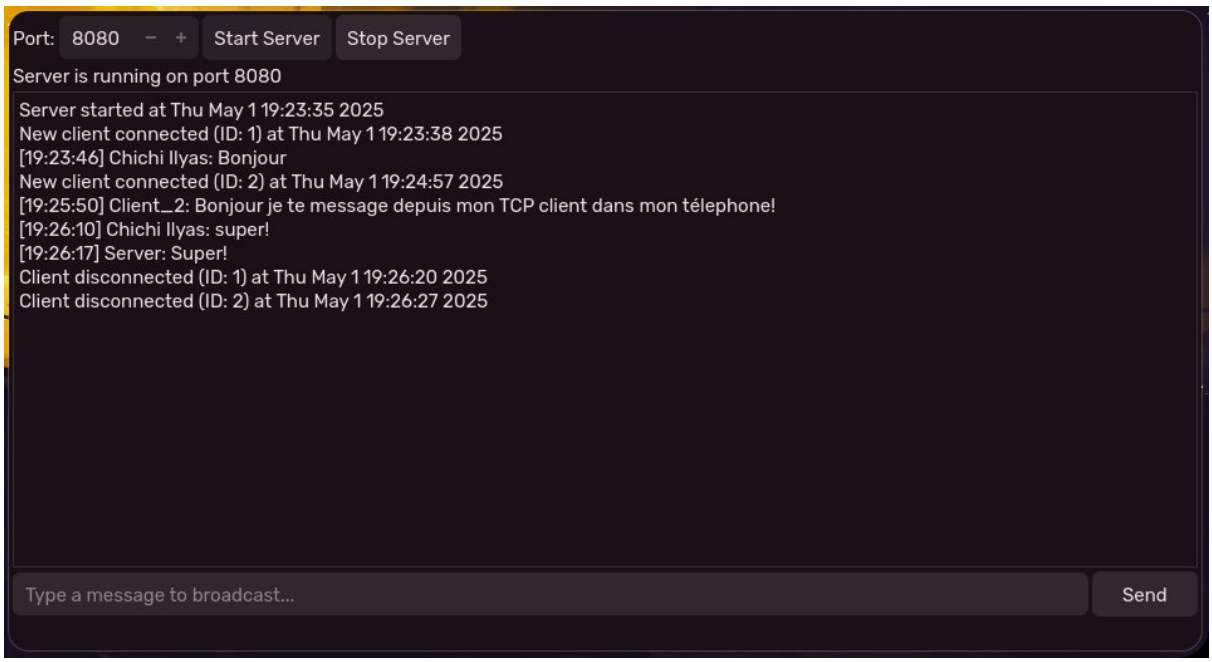
```
void MessagingClient::connectionEstablished()
{
    ui->chatDisplay->append(tr("Connected to server at %1")
                            .arg(QDateTime::currentDateTime().toString()));
    setConnectionStatus(true);

    QString username = ui->usernameInput->text().trimmed();
    if (!username.isEmpty()) {
        QByteArray block;
        QDataStream out(&block, QIODevice::WriteOnly);
        out.setVersion(QDataStream::Qt_5_15);
        out << quint32(0);
        out << QString("/name %1").arg(username);
        out.device()->seek(0);
        out << quint32(block.size() - sizeof(quint32));
        socket->write(block);
    }
}

void MessagingClient::connectionClosed()
{
    ui->chatDisplay->append(tr("Disconnected from server at %1")
                            .arg(QDateTime::currentDateTime().toString()));
    setConnectionStatus(false);
}
```

Fonction pour la gestion des boutons lorsqu'on établie une connection

```
void MessagingClient::setConnectionStatus(bool connected)
{
    ui->connectButton->setEnabled(!connected);
    ui->disconnectButton->setEnabled(connected);
    ui->serverInput->setEnabled(!connected);
    ui->portSpinBox->setEnabled(!connected);
    ui->usernameInput->setEnabled(!connected);
    ui->sendButton->setEnabled(connected);
    ui->messageInput->setEnabled(connected);

    if (connected) {
        ui->statusLabel->setText(tr("Connected to %1:%2")
                                 .arg(ui->serverInput->text())
                                 .arg(ui->portSpinBox->value()));
    } else {
        ui->statusLabel->setText(tr("Disconnected"));
    }
}
```

messagingclient.ui

Exemple de fonctionalité



Conclusion:

Nous sommes fiers d'avoir réussi à réaliser un projet de cette ampleur et nous remercions sérieusement, honnêtement et absolument notre superviseur Essaid Elhaji de nous avoir donné l'opportunité d'apprendre et de devenir ce que nous sommes aujourd'hui dans le domaine de la programmation.