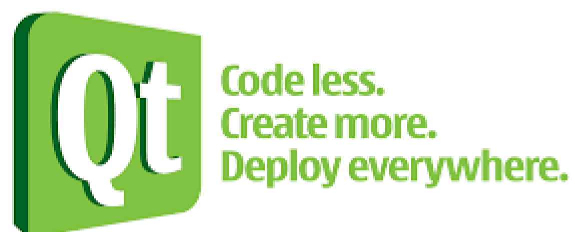
Communiquer en réseau avec Qt

Table des matières

1. Introduction....2
 1.1. Communiquer sur un réseau internet....2
 1.2. L'adresse IP....2
 1.3. Les ports....2
 1.4. Le protocole....1
2. Architecture d'un projet de Chat....2
3. Réalisation du serveur....4
 3.1. Slot nouvelleConnexion()....8
 3.2. Slot donneesRecues()....8
 3.3. Slot deconnexionClient()....10
 3.4. Méthode envoyerATous()....11
 3.5. Lancement du serveur....11
4. Réalisation du client....12
 4.1. Fichier .pro....13
 4.2. Fichier main.cpp....13
 4.3. Fichier .h....13
 4.4. Fichier .cpp....14
 4.5. Slot on_boutonConnexion_clicked()....15
 4.6. Slot on_boutonEnvoyer_clicked()....15
 4.7. Slot on_message_returnPressed()....16
 4.8. Slot donneesRecues()....16
 4.9. Slot connecte()....17
 4.10. Slot deconnecte()....17
 4.11. Slot erreurSocket()....17

Qt est une API orientée objet qui offre des composants d'interface graphique, d'accès aux données, de connexions réseaux,, d'analyse XML, etc. Qt est connu pour être la bibliothèque sur laquelle repose l'un des environnements de bureau les plus utilisés dans le monde GNU/Linux.



1. Introduction

1.1. Communiquer sur un réseau internet

Pour que 2 programmes communiquent entre eux via le réseau, il faut :

- Connaître l'adresse IP identifiant l'autre ordinateur.
- Utiliser un port libre et ouvert.
- Utiliser le même protocole de transmission des données.

1.2. L'adresse IP

Chaque ordinateur est identifié sur le réseau par une adresse IP. Cette adresse représente un ordinateur. Cependant, un ordinateur peut avoir plusieurs IP :

- Une IP interne : c'est le localhost, aussi appelé loopback. C'est une IP très pratique pour les tests.
- Une IP du réseau local : c'est une IP privée, utilisée par les ordinateurs qui communiquent entre eux sans passer par internet.
- Une IP internet : c'est uue IP publique, utilisée pour communiquer avec tous les autres ordinateurs de la planète qui sont connectés à internet.

Pour trouver son adresse IP, il existe plusieurs méthodes :

- Pour l'IP interne : c'est 127.0.0.1 (ou son équivalent texte "localhost").
- Pour l'IP locale :
  - Sous Windows, ouvrez une invite de commande et tapez ipconfig
  - Sous Linux , ouvrez une invite de commande et tapez ifconfig
- Pour l'IP internet : aller sur ce site web

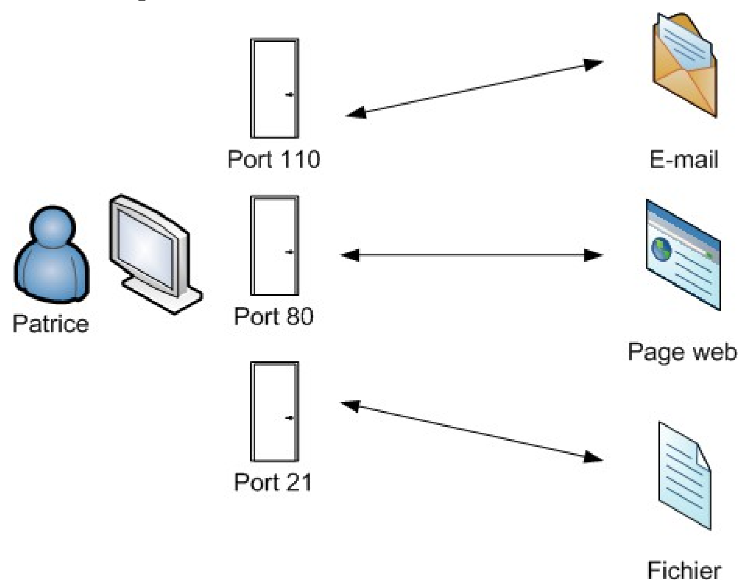
1.3. Les ports

Un ordinateur connecté à un réseau reçoit beaucoup de messages en même temps (pages web, mails, …). Pour ne pas confondre ces données, on a inventé le concept de port.

Un port est un nombre compris entre 1 et 65 536. Voici quelques ports célèbres :

- 21 : utilisé par les logiciels FTP pour envoyer et recevoir des fichiers.

- 80 : utilisé pour naviguer sur le web par votre navigateur.
- 110 : utilisé pour la réception de mails.



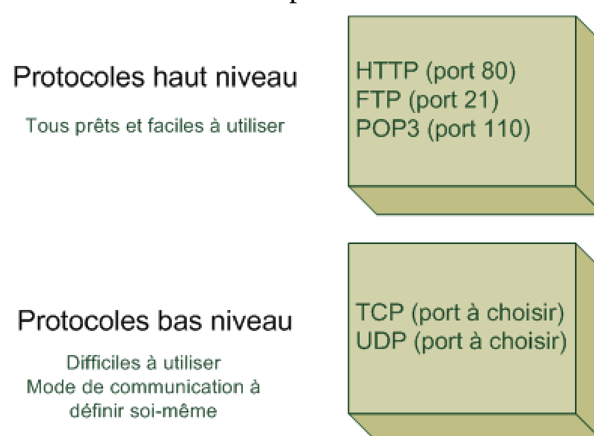
La plupart des ports dont les numéros sont inférieurs à 1 024 sont déjà réservés par votre machine. Nous ferons donc en sorte de préférence dans notre programme d'utiliser un numéro de port compris entre 1 024 et 65 536.

1.4. Le protocole

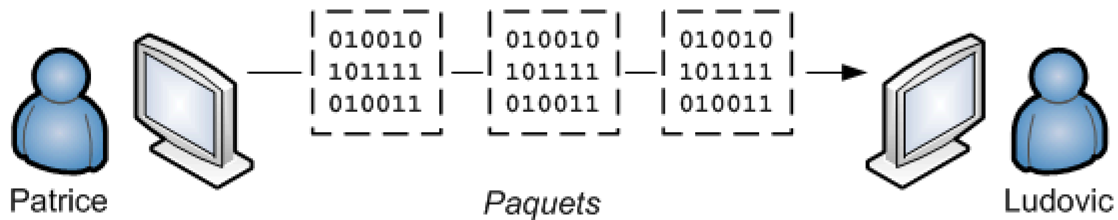
Définition : un protocole est un ensemble de règles qui permettent à 2 ordinateurs de communiquer. Il faut impérativement que les 2 ordinateurs parlent le même protocole pour que l'échange de données puisse fonctionner.

Il existe des centaines de protocoles de communication différents. Ceux-ci peuvent être très simples comme très complexes, selon si vous discutez à un "haut niveau" ou à un "bas niveau" :

- Protocoles de haut niveau : par exemple le protocole FTP, qui utilise le port 21 pour envoyer et recevoir des fichiers, est un système d'échange de données de haut niveau.
- Protocoles de bas niveau : par exemple le protocole TCP. Il est utilisé par les programmes pour lesquels aucun protocole de haut niveau ne convient. Vous devrez manipuler les données qui transitent sur le réseau octet par octet.



Les données s'envoient par paquets sur le réseau :



Ce sont les protocoles de bas niveau TCP et UDP qui s'en occupent. Il est donc impossible de connaître à l'avance la taille des paquets ou même leur nombre.

On peut envoyer ces paquets de plusieurs façons différentes :

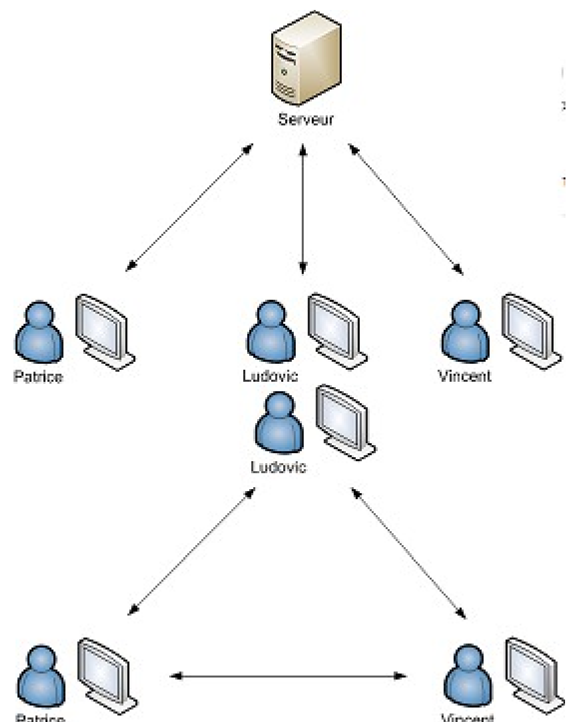
- Protocole TCP : il nécessite d'établir une connexion au préalable entre les ordinateurs. Il y a un système de contrôle qui permet de demander à renvoyer un paquet au cas où l'un d'entre eux se serait perdu sur le réseau.
- Protocole UDP : il ne nécessite pas d'établir de connexion au préalable et il est très rapide. En revanche, il n'y a aucun contrôle ce qui fait qu'un paquet de données peut très bien se perdre sans qu'on en soit informé, ou les paquets peuvent arriver dans le désordre !

2. Architecture d'un projet de Chat

On a 2 architectures possibles pour ce projet :

Une architecture client / serveur : c'est l'architecture réseau la plus classique et la plus simple à mettre en oeuvre. Les machines des utilisateurs (Patrice, Ludovic, Vincent...) sont appelées des "clients". En plus de ces machines, on utilise un autre ordinateur (appelé "serveur") qui va se charger de répartir les communications entre les clients.

Une architecture Peer-To-Peer (P2P) : ce mode plus complexe est dit décentralisé, car il n'y a pas de serveur. Chaque client peut communiquer directement avec un autre client. C'est plus direct, ça évite d'encombrer un serveur, mais c'est plus délicat à mettre en place.



Nous prendrons comme exemple une architecture client. Il va donc falloir faire deux projets :

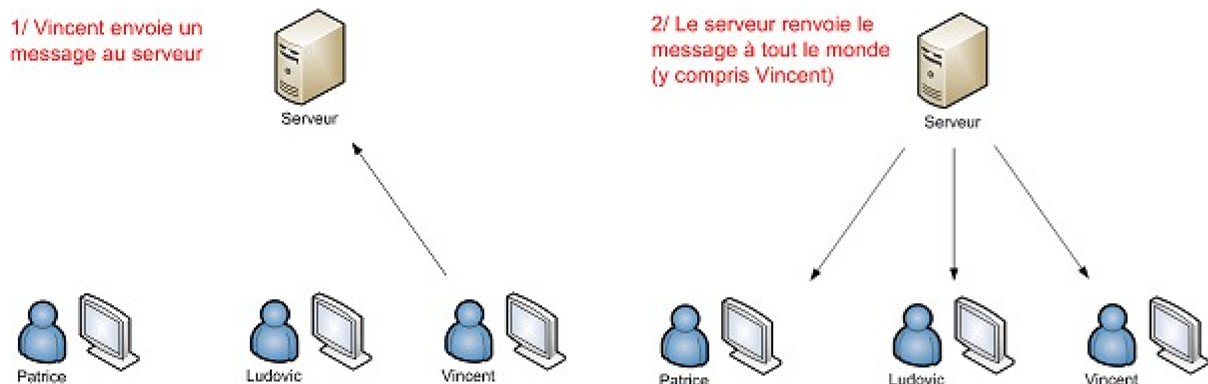
- Un projet "serveur" : pour créer le programme qui va répartir les messages entre les clients.
- Un projet "client" : pour chaque client qui participera au Chat.

L'une des machines des clients peut aussi faire office de serveur. Il suffira de faire tourner un

programme "serveur" en même temps qu'un programme "client".

En pratique donc, une seule personne lancera le programme "serveur" et le programme "client" à la fois, et toutes les autres lanceront uniquement le programme "client".

Principe de fonctionnement du Chat :

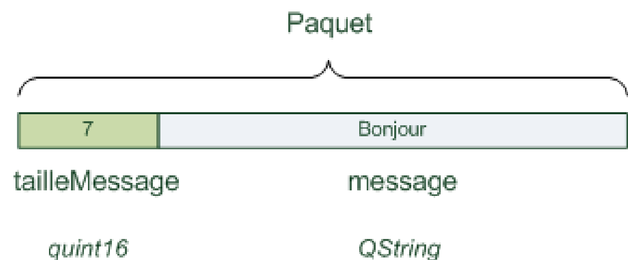


Il est plus simple de faire en sorte que le serveur renvoie le message à tout le monde sans distinction. Vincent verra donc son message s'afficher sur son écran de discussion uniquement quand le serveur l'aura reçu et le lui aura renvoyé. Cela permet de vérifier en outre que la communication sur le réseau fonctionne correctement.

Les messages qui circuleront sur le réseau seront placés dans des paquets.

Le paquet est constitué de 2 parties :

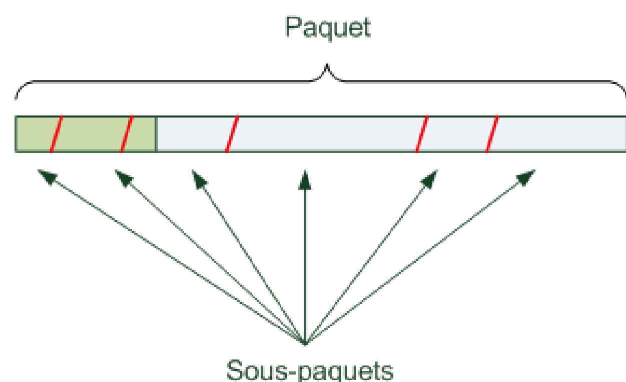
- tailleMessage : un nombre entier qui sert à indiquer la taille du message qui suit.
- message : c'est le message envoyé par le client.



Le protocole TCP va découper le paquet en sous-paquets avant de l'envoyer sur le réseau. Il n'enverra peut-être pas tout d'un coup :

On n'a aucun contrôle sur la taille de ces sous-paquets, et il n'y a aucun moyen de savoir à l'avance comment ça va être découpé.

Le serveur va recevoir ces paquets petit à petit, et non pas tout d'un coup. Il ne peut pas savoir quand la totalité du message a été reçue.



Le protocole TCP s'arrange pour que les paquets arrivent à destination dans le bon ordre contrairement au protocole UDP, qui est plus rapide, ne fait aucun contrôle sur l'ordre des paquets envoyés !

3. Réalisation du serveur

Créez un nouveau projet constitué de 3 fichiers :

- main.cpp
- FenServeur.cpp
- FenServeur.h

Editez le fichier .pro pour demander à Qt de rajouter la gestion du réseau :

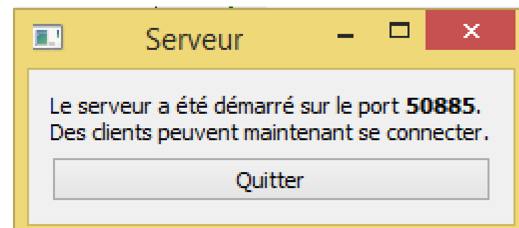
```
TEMPLATE = app

QT += widgets network
DEPENDPATH += .
INCLUDEPATH += .
# Input
HEADERS += FenServeur.h
SOURCES += FenServeur.cpp main.cpp
```

Avec QT += widgets network, Qt sait que le projet va utiliser le réseau et peut préparer un makefile approprié.

Le serveur est une application qui tourne en tâche de fond. Normalement, rien ne nous oblige à créer une fenêtre pour ce projet, mais on va quand même en faire une pour que l'utilisateur puisse arrêter le serveur en fermant la fenêtre.

Notre fenêtre sera toute simple, elle affichera le texte "Le serveur a été lancé sur le port XXXX" et un bouton "Quitter".



Fichier main.cpp :

```
#include <QApplication>

#include "FenServeur.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenServeur fenetre;
    fenetre.show();

    return app.exec();
}
```

Fichier FenServeur.h :

```
#ifndef HEADER_FENSERVEUR

#define HEADER_FENSERVEUR

#include <QtWidgets>
#include <QtNetwork>

class FenServeur : public QWidget
```

```
{
    Q_OBJECT

    public:
        FenServeur();
        void envoyerATous(const QString &message);

    private slots:
        void nouvelleConnexion();
        void donneesRecues();
        void deconnexionClient();

    private:
        QLabel *etatServeur;
        QPushButton *boutonQuitter;

        QTcpServer *serveur;
        QList<QTcpSocket *> clients;
        quint16 tailleMessage;
};

#endif
```

Notre fenêtre hérite de QWidget. Elle est constituée d'un QLabel et d'un QpushButton.

On a rajouté d'autres attributs spécifiques à la gestion du réseau :

- QTcpServer *serveur : c'est l'objet qui représente le serveur sur le réseau.
- QList<QTcpSocket *> clients : c'est un tableau qui contient la liste des clients connectés. On aurait pu utiliser un tableau classique, mais on va passer par une QList, un tableau de taille dynamique. En effet, on ne connaît pas à l'avance le nombre de clients qui se connecteront. Chaque QTcpSocket de ce tableau représentera une connexion à un client.
- quint16 tailleMessage : ce quint16 sera utilisé dans le code pour se "souvenir" de la taille du message que le serveur est en train de recevoir.

La classe est constituée de plusieurs méthodes (dont des slots) :

- Le constructeur : il initialise les widgets sur la fenêtre et initialise aussi le serveur (QTcpServer) pour qu'il démarre.
- envoyerATous() : une méthode qui se charge d'envoyer à tous les clients connectés le message passé en paramètre.
- Slot nouvelleConnexion() : appelé lorsqu'un nouveau client se connecte.
- Slot donneesRecues() : appelé lorsque le serveur reçoit des données. Attention, ce slot est appelé à chaque sous-paquet reçu. Il faudra "attendre" d'avoir reçu le nombre d'octets indiqués dans tailleMessage avant de pouvoir considérer qu'on a reçu le message entier.
- Slot deconnexionClient() : appelé lorsqu'un client se déconnecte.

Fichier FenServeur.cpp :

NB : le constructeur se charge de placer les widgets sur la fenêtre et de faire démarrer le serveur via le QTcpServer :

```
#include "FenServeur.h"

FenServeur::FenServeur()
```

```
{
    // Création et disposition des widgets de la fenêtre
    etatServeur = new QLabel;
    boutonQuitter = new QPushButton(tr("Quitter"));
    connect(boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(etatServeur);
    layout->addWidget(boutonQuitter);
    setLayout(layout);

    setWindowTitle(tr("Serveur"));

    // Gestion du serveur
    serveur = new QTcpServer(this);
    // Démarrage du serveur sur toutes les IP disponibles et sur le port 50585
    if ( !serveur->listen(QHostAddress::Any, 50885) )
    {
        // Si le serveur n'a pas été démarré correctement
        etatServeur->setText(tr("Le serveur n'a pas pu être démarré :<br />") + serveur-
>errorString());
    }
    else
    {
        // Si le serveur a été démarré correctement
        etatServeur->setText(tr("Le serveur a été démarré sur le port <strong>") +
QString::number(serveur->serverPort()) + tr("</strong>.<br />Des clients peuvent
maintenant se connecter."));
        connect(serveur, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
    }

    tailleMessage = 0;
}

void FenServeur::nouvelleConnexion()
{
    envoyerATous(tr("<em>Un nouveau client vient de se connecter</em>"));

    QTcpSocket *nouveauClient = serveur->nextPendingConnection();
    clients << nouveauClient;

    connect(nouveauClient, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(nouveauClient, SIGNAL(disconnected()), this, SLOT(deconnexionClient()));
}

void FenServeur::donneesRecues()
{
    // 1 : on reçoit un paquet (ou un sous-paquet) d'un des clients

    // On détermine quel client envoie le message (recherche du QTcpSocket du client)
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    // Si on n'a pas trouvé le client à l'origine du signal, on arrête la méthode
    if ( socket == 0 )
        return;

    // Si tout va bien, on continue : on récupère le message
    QDataStream in(socket);

    // Si on ne connaît pas encore la taille du message, on essaie de la récupérer
    if ( tailleMessage == 0 )
    {
```

```
        // On n'a pas reçu la taille du message en entier
        if ( socket->bytesAvailable() < (int)sizeof(quint16) )
             return;

        // Si on a reçu la taille du message en entier, on la récupère
        in >> tailleMessage;
    }

    // Si on connaît la taille du message, on vérifie si on a reçu le message en entier
    if ( socket->bytesAvailable() < tailleMessage )
        // Si on n'a pas encore tout reçu, on arrête la méthode
        return;

    // Si ces lignes s'exécutent, c'est qu'on a reçu tout le message : on le récupére !
    QString message;
    in >> message;

    // 2 : on renvoie le message à tous les clients
    envoyerATous(message);

    // 3 : remise de la taille du message à 0 pour la réception des futurs messages
    tailleMessage = 0;
}

void FenServeur::deconnexionClient()
{
    envoyerATous(tr("<em>Un client vient de se déconnecter</em>"));

    // On détermine quel client se déconnecte
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if ( socket == 0 )
        // Si on n'a pas trouvé le client à l'origine du signal, on arrête la méthode
        return;

    clients.removeOne(socket);

    socket->deleteLater();
}

void FenServeur::envoyerATous(const QString &message)
{
    // Préparation du paquet
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On écrit 0 au début du paquet pour réserver la place pour écrire la taille
    out << (quint16) 0;
    out << message; // On ajoute le message à la suite
    out.device()->seek(0); // On se replace au début du paquet
    // On écrase le 0 qu'on avait réservé par la longueur du message
    out << (quint16) (paquet.size() - sizeof(quint16));

    // Envoi du paquet préparé à tous les clients connectés au serveur
    for (int i = 0; i < clients.size(); i++)
    {
        clients[i]->write(paquet);
    }

}
```

La première étape consiste à disposer les widgets sur la fenêtre.

La seconde étape à démarrer le serveur : on crée un nouvel objet de type QTcpServer dans un premier temps. On lui passe en paramètre this, un pointeur vers la fenêtre, pour faire en sorte que la fenêtre soit le parent du QTcpServer.

Ensuite, on démarre le serveur grâce à serveur->listen(QHostAddress::Any, 50885) :

- IP : c'est l'IP sur laquelle le serveur "écoute" si de nouveaux clients arrivent. La mention QHostAddress::Any autorise toutes les connexions : internes, locales et externes.
- port : c'est le numéro du port sur lequel on souhaite lancer le serveur. On a choisi un numéro au hasard, compris entre 1 024 et 65 536. Sans paramètre, le serveur choisi un port libre au hasard.

La méthode listen() renvoie un booléen : vrai si le serveur a bien pu se lancer, faux s'il y a eu un problème. On affiche un message en conséquence sur la fenêtre du serveur.

Si le démarrage du serveur a fonctionné, on connecte le signal newConnection() vers notre slot personnalisé nouvelleConnexion() pour traiter l'arrivée d'un nouveau client sur le serveur.

3.1. Slot nouvelleConnexion()

Le slot nouvelleConnexion() est appelé dès qu'un nouveau client se connecte au serveur : on envoie à tous les clients déjà connectés un message comme quoi un nouveau client vient de se connecter.

Chaque client est représenté par un QTcpSocket. Pour récupérer la socket correspondant au nouveau client qui vient de se connecter, on appelle la méthode nextPendingConnection() du QTcpServer. Cette méthode retourne la QTcpSocket du nouveau client.

On conserve la liste des clients connectés dans un tableau, appelé clients.

Ce tableau est géré par la classe QList. On ajoute le nouveau client à la fin du tableau très facilement, comme ceci : `clients << nouveauClient;`

On connecte ensuite les signaux que peut envoyer le client à des slots. On va gérer 2 signaux :

- readyRead() : signale que le client a envoyé des données. Ce signal est émis pour chaque sous-paquet reçu. Lorsqu'un client enverra un message, ce signal pourra donc être émis plusieurs fois jusqu'à ce que tous les sous-paquets soient arrivés.

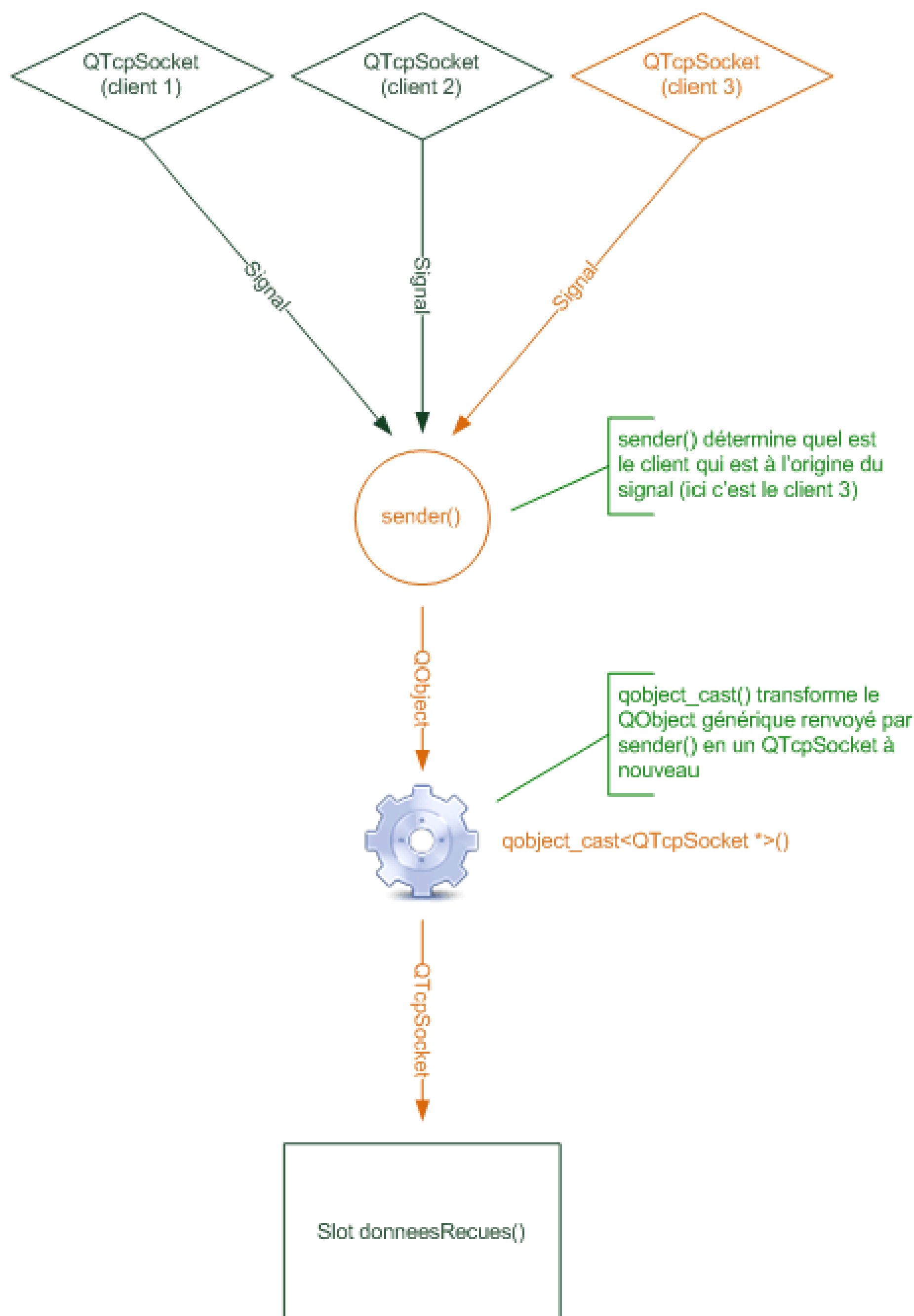
  Le slot personnalisé donneesRecues() traitera les sous-paquets.
- disconnected() : signale que le client s'est déconnecté. Le slot se chargera d'informer les autres clients de son départ et de supprimer la QTcpSocket correspondante dans la liste des clients connectés.

3.2. Slot donneesRecues()

Le slot donneesRecues() va être appelé à chaque fois qu'on reçoit un sous-paquet d'un des clients.

Il faut utiliser l'objet QTcpSocket du client pour récupérer les sous-paquets qui ont transité par le réseau. La méthode sender() de QObject dans le slot retourne un pointeur vers l'objet à l'origine du message.

Ce pointeur est de type QObject (classe générique de Qt) et il faudra le transformer QTcpSocket à l'aide de la méthode qobject_cast() : `QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());`



Remarque : la méthode qobject_cast() est similaire au dynamic_cast() de la bibliothèque standard du C++. Son rôle est de forcer la transformation d'un objet d'un type vers un autre.

NB : il se peut que le qobject_cast() n'ait pas fonctionné (par exemple parce que l'objet n'était pas de type QTcpSocket contrairement à ce qu'on attendait). Dans ce cas, il renvoie 0. Il faut que l'on teste si le qobject_cast() a fonctionné avant d'aller plus loin.

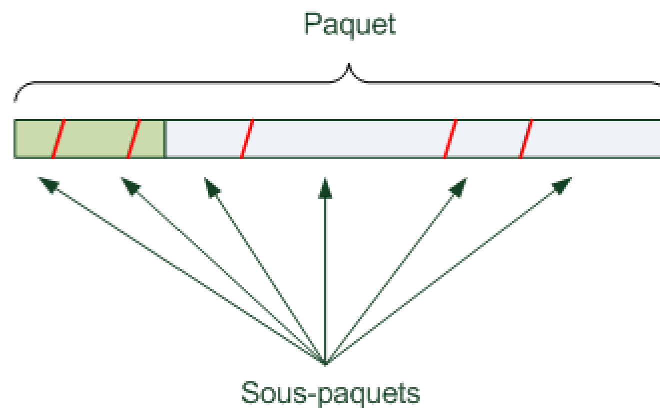
On peut ensuite travailler à récupérer les données. On commence par créer un flux de données pour lire ce que contient la socket : QDataStream in(socket);

Si lors de l'appel au slot ce tailleMessage vaut 0, cela signifie qu'on est en train de recevoir le début d'un nouveau message.

On demande à la socket combien d'octets ont été reçus dans le sous-paquet grâce à la méthode

bytesAvailable(). Si on a reçu moins d'octets que la taille d'un quint16, on arrête la méthode et on attend le prochain appel de la méthode pour vérifier à nouveau si on a reçu assez d'octets pour récupérer la taille du message.

NB : le type int peut avoir une taille différente selon les machines sur le réseau (un int peut prendre 16 bits de mémoire sur une machine, et 8 bits sur une autre). Pour résoudre le problème, on utilise un type spécial de Qt, le quint16, qui correspond à un nombre entier prenant 16 bits de mémoire quelle que soit la machine.



Le slot est appelé à chaque fois qu'un sous-paquet a été reçu.

On vérifie si on a reçu assez d'octets pour récupérer la taille du message (première section en gris foncé). La taille de la première section "tailleMessage" peut être facilement retrouvée grâce à l'opérateur sizeof().

Si on n'a pas reçu assez d'octets, on arrête la méthode (return). On attendra que le slot soit à nouveau appelé et on vérifiera alors cette fois si on a reçu assez d'octets.

Quand on a reçu la taille du message, on essaye de récupérer le message lui-même :

```
if ( socket->bytesAvailable() < tailleMessage )
       // Si on n'a pas encore tout reçu, on arrête la méthode
       return;
```

Le principe est le même. On regarde le nombre d'octets reçus, et si on en a moins que la taille annoncée du message, on arrête (return).

Si tout va bien, on peut passer à la suite de la méthode. Si ces lignes s'exécutent, c'est qu'on a reçu le message en entier, donc qu'on peut le récupérer dans une QString :

```
QString message;
in >> message;
```

Il faut maintenant renvoyer le message à tous les clients. L'envoi du message à tout le monde se fait via la méthode envoyerATous.

On remet ensuite tailleMessage à 0 pour que l'on puisse recevoir de futurs messages d'autres clients.

3.3. Slot deconnexionClient()

Ce slot est appelé lorsqu'un client se déconnecte.

On va envoyer un message à tous les clients encore connectés pour qu'ils sachent qu'un client vient de partir. Puis, on supprime la QTcpSocket correspondant au client dans notre tableau QList. Ainsi,

le serveur "oublie" ce client, il ne considère plus qu'il fait partie des connectés.

Comme plusieurs signaux sont connectés à ce slot, on ne sait pas quel est le client à l'origine de la déconnexion. Pour le retrouver, on utilise la même technique que pour le slot donneesRecues().

La méthode removeOne() de QList permet de supprimer le pointeur vers l'objet dans le tableau. Notre liste des clients est maintenant à jour.

Il ne reste plus qu'à finir de supprimer l'objet lui-même avec deleteLater() et Qt se chargera de faire le delete lui-même un peu plus tard, lorsque notre slot aura fini de s'exécuter.

3.4. Méthode envoyerATous()

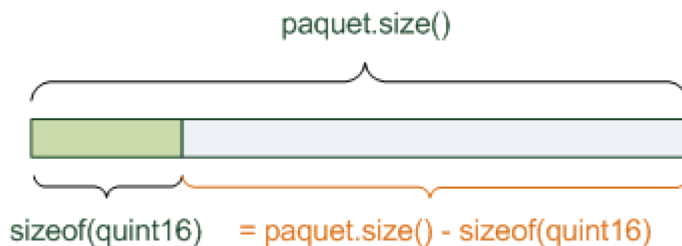
On va envoyer un message à tous les clients connectés (présents dans la QList).

On crée un QByteArray "paquet" qui va contenir le paquet à envoyer sur le réseau. La classe QByteArray représente une suite d'octets quelconque.

On utilise un QDataStream pour écrire dans le QByteArray afin d'utiliser l'opérateur "<<".

On écrit d'abord le message (QString) et ensuite on calcule sa taille qu'on écrit au début du message :

1. On écrit le nombre 0 de type quint16 pour "réserver" de la place.
2. On écrit à la suite le message, de type QString. Le message a été reçu en paramètre de la méthode envoyerATous().
3. On se replace au début du paquet (comme si on remettait le curseur au début d'un texte dans un traitement de texte).



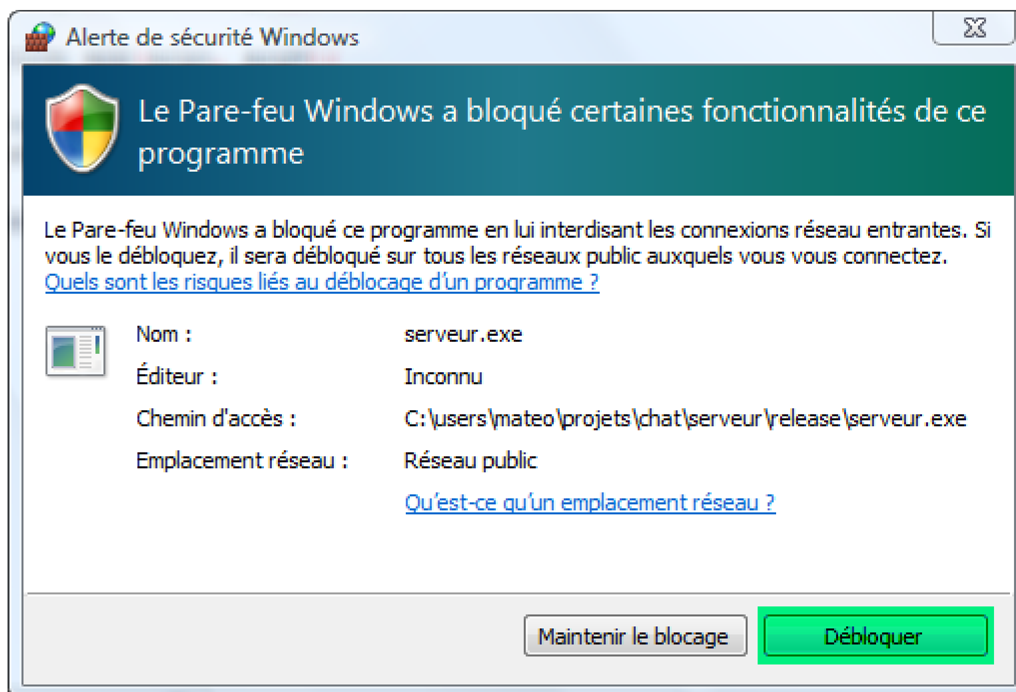
4. On écrase le 0 qu'on avait écrit pour réserver de la place par la bonne taille du message. Cette taille est calculée via une simple soustraction : la taille du message est égale à la taille du paquet moins la taille réservée pour le quint16.

Le paquet est prêt à être envoyer à tous les clients grâce à la méthode write() du socket.

Pour cela, on fait une boucle sur la QList, et on envoie le message à chaque client.

3.5. Lancement du serveur

À l'exécution, vous risquez d'avoir une alerte de votre pare-feu (firewall) :



En effet, le programme va communiquer sur le réseau. Le pare-feu nous demande si nous voulons autoriser notre programme à le faire.

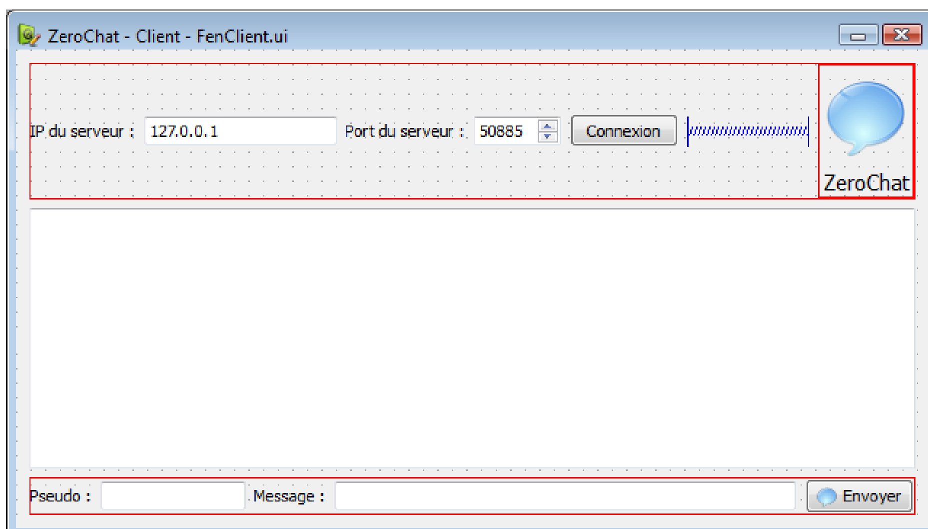
Laissez ce programme tourner en fond sur l'ordinateur (vous pouvez réduire la fenêtre). Il va servir à faire la communication entre les différents clients.

4. Réalisation du client

Nous aurons 3 fichiers à nouveau :

- main.cpp
- FenClient.h
- FenClient.cpp

On va créer l'interface de la fenêtre du client via Qt Designer :



Prenez soin à donner un nom correct à chacun des widgets via la propriété objectName.

Veillez aussi à utiliser des layouts sur la fenêtre pour la rendre redimensionnable sans problème. On sélectionne plusieurs widgets à la fois et on clique sur un des boutons de la barre d'outils en haut pour les assembler selon un layout.

4.1. Fichier .pro

Mettez mis à jour le fichier .pro pour indiquer qu'il y a un UI dans le projet et qu'on utilise le module network de Qt :

```
TEMPLATE = app

QT += widgets network
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenClient.h
FORMS += FenClient.ui
SOURCES += FenClient.cpp main.cpp
```

4.2. Fichier main.cpp

```
#include <QApplication>

#include "FenClient.h"
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    FenClient fenetre;
    fenetre.show();
    return app.exec();
}
```

4.3. Fichier .h

Notre fenêtre utilise un fichier généré avec Qt Designer. On va donc utiliser un héritage multiple pour éviter de devoir mettre le préfixe ui-> partout dans le code :

```
#ifndef HEADER_FENCLIENT
#define HEADER_FENCLIENT

#include <QtWidgets>
#include <QtNetwork>
#include "ui_FenClient.h"  //  la fenêtre générée

class FenClient : public QWidget, private Ui::FenClient
{
    Q_OBJECT
    public:
        FenClient();
    private slots:
        void on_boutonConnexion_clicked();
        void on_boutonEnvoyer_clicked();
        void on_message_returnPressed();
        void donneesRecues();
        void connecte();
        void deconnecte();
        void erreurSocket(QAbstractSocket::SocketError erreur);
    private:
        QTcpSocket *socket; // Représente le serveur
        quint16 tailleMessage;
```

```
};
#endif
```

La fenêtre comporte des slots qu'il va falloir implémenter. On utilise les autoconnect pour les 3 premiers d'entre eux pour gérer les évènements de la fenêtre :

- on_boutonConnexion_clicked() : appelé lorsqu'on clique sur le bouton "Connexion" et qu'on souhaite donc se connecter au serveur.
- on_boutonEnvoyer_clicked() : appelé lorsqu'on clique sur "Envoyer" pour envoyer un message dans le Chat.
- on_message_returnPressed() : appelé lorsqu'on appuie sur la touche "Entrée" lorsqu'on rédige un message. Comme cela revient au même que on_boutonEnvoyer_clicked(), on appellera cette méthode directement pour éviter d'avoir à écrire 2 fois le même code.
- donneesRecues() : appelé lorsqu'on reçoit un sous-paquet du serveur. Ce slot sera très similaire à celui du serveur qui possède le même nom.
- connecte() : appelé lorsqu'on vient de réussir à se connecter au serveur.
- deconnecte() : appelé lorsqu'on vient de se déconnecter du serveur.
- erreurSocket(QAbstractSocket::SocketError erreur) : appelé lorsqu'il y a eu une erreur sur le réseau (connexion au serveur impossible par exemple).

En plus de ça, on a 2 attributs à manipuler :

- QTcpSocket *socket : une socket qui représentera la connexion au serveur. On utilisera cette socket pour envoyer des paquets au serveur.
- quint16 tailleMessage : permet à l'objet de se "souvenir" de la taille du message qu'il est en train de recevoir dans son slot donneesRecues(). Il a la même utilité que sur le serveur.

4.4. Fichier .cpp

Le constructeur se doit d'appeler setupUi() dès le début pour mettre en place les widgets sur la fenêtre. C'est justement là qu'on gagne du temps grâce à Qt Designer : on n'a pas à coder le placement des widgets sur la fenêtre.

```
FenClient::FenClient()
{
    setupUi(this);

    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(socket, SIGNAL(connected()), this, SLOT(connecte()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(deconnecte()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)), this,
SLOT(erreurSocket(QAbstractSocket::SocketError)));

    tailleMessage = 0;
}
```

En plus de setupUi(), on fait quelques initialisations supplémentaires indispensables :

- On crée l'objet de type QTcpSocket qui va représenter la connexion au serveur.
- On connecte les signaux qu'il est susceptible d'envoyer à nos slots personnalisés.
- On met tailleMessage à 0 pour permettre la réception de nouveaux messages.

NB : on ne se connecte pas au serveur dans le constructeur. On prépare juste la socket, mais on ne fera la connexion que lorsque le client aura cliqué sur le bouton "Connexion".

4.5. Slot on_boutonConnexion_clicked()

Ce slot se fait appeler dès que l'on a cliqué sur le bouton "Connexion" en haut de la fenêtre :

```
// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("<em>Tentative de connexion en cours...</em>"));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il y en a
    // On se connecte au serveur demandé
    socket->connectToHost(serveurIP->text(), serveurPort->value());
}
```

1. Dans un premier temps, on affiche sur la zone de messages "listeMessages" au centre de la fenêtre que l'on est en train d'essayer de se connecter.
2. On désactive temporairement le bouton "Connexion" pour empêcher au client de retenter une connexion alors qu'une tentative de connexion est déjà en cours.
3. Si la socket est déjà connectée à un serveur, on coupe la connexion avec abort(). Si on n'était pas connecté, cela n'aura aucun effet, mais c'est par sécurité pour que l'on ne soit pas connecté à 2 serveurs à la fois.
4. On se connecte enfin au serveur avec la méthode connectToHost(). On utilise l'IP et le port demandés par l'utilisateur dans les champs de texte en haut de la fenêtre.

4.6. Slot on_boutonEnvoyer_clicked()

Ce slot est appelé lorsqu'on essaie d'envoyer un message (le bouton "Envoyer" a été cliqué) :

```
// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);
    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text() +tr("</strong> : ") +
message->text();

    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet
    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
}
```

Ce code est similaire à celui de la méthode envoyerATous() du serveur. Il s'agit d'un envoi de données sur le réseau.

1. On prépare un QByteArray dans lequel on va écrire le paquet qu'on veut envoyer.
2. On construit ensuite la QString contenant le message à envoyer. On met le nom de l'auteur et

son texte directement dans la même QString.

3. On calcule la taille du message.
4. On envoie le paquet ainsi créé au serveur en utilisant la socket qui le représente et sa méthode write().
5. On efface automatiquement la zone d'écriture des messages en bas pour qu'on puisse en écrire un nouveau et on donne le focus à cette zone immédiatement pour que le curseur soit placé dans le bon widget.

4.7. Slot on_message_returnPressed()

Ce slot est appelé lorsqu'on a appuyé sur la touche "Entrée" après avoir rédigé un message.

Cela a le même effet qu'un clic sur le bouton "Envoyer", nous appelons donc le slot que nous venons d'écrire :

```
// Appuyer sur la touche Entrée a le même effet que cliquer sur le bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}
```

4.8. Slot donneesRecues()

Il est quasiment identique à celui du serveur (la réception de données fonctionne de la même manière) :

```
// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    /* Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier
    (en se basant sur la taille annoncée tailleMessage)
    */
    QDataStream in(socket);
    if ( tailleMessage == 0 )
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
             return;
        in >> tailleMessage;
    }

    if (socket->bytesAvailable() < tailleMessage)
        return;

    // Si on arrive jusqu'à cette ligne, on peut récupérer le message entier
    QString messageRecu;
    in >> messageRecu;

    // On affiche le message sur la zone de Chat
    listeMessages->append(messageRecu);

    // On remet la taille du message à 0 pour pouvoir recevoir de futurs messages
    tailleMessage = 0;
}
```

La seule différence ici en fait, c'est qu'on affiche le message reçu dans la zone de Chat à la fin : listeMessages->append(messageRecu);

4.9. Slot connecte()

Ce slot est appelé lorsqu'on a réussi à se connecter au serveur.

```
// Ce slot est appelé lorsque la connexion au serveur a réussi
void FenClient::connecte()
{
    listeMessages->append(tr("<em>Connexion réussie !</em>"));
    boutonConnexion->setEnabled(true);
}
```

On réactive aussi le bouton "Connexion" qu'on avait désactivé, pour permettre une nouvelle connexion à un autre serveur.

4.10. Slot deconnecte()

Ce slot est appelé lorsqu'on est déconnecté du serveur.

```
// Ce slot est appelé lorsqu'on est déconnecté du serveur
void FenClient::deconnecte()
{
    listeMessages->append(tr("<em>Déconnecté du serveur</em>"));
}
```

4.11. Slot erreurSocket()

Ce slot est appelé lorsque la socket a rencontré une erreur.

```
// Ce slot est appelé lorsqu'il y a une erreur
void FenClient::erreurSocket(QAbstractSocket::SocketError erreur)
{
    switch(erreur) // On affiche un message différent selon l'erreur qu'on nous indique
    {
        case QAbstractSocket::HostNotFoundError:
            listeMessages->append(tr("<em>ERREUR : le serveur n'a pas pu être trouvé. 
Vérifiez l'IP et le port.</em>"));
            break;
        case QAbstractSocket::ConnectionRefusedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a refusé la connexion. 
Vérifiez si le programme \"serveur\" a bien été lancé. Vérifiez aussi l'IP et le 
port.</em>"));
            break;
        case QAbstractSocket::RemoteHostClosedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a coupé la 
connexion.</em>"));
            break;
        default:
            listeMessages->append(tr("<em>ERREUR : ") + socket->errorString() + 
tr("</em>"));
    }
    boutonConnexion->setEnabled(true);
}
```

NB : la plupart des erreurs gèrées ici sont liées à la connexion au serveur. Le cas "default" est appelé pour les erreurs non gérées.

Voici le code complet de FenClient.cpp :

```
#include "FenClient.h"

FenClient::FenClient()
{
    setupUi(this);
```

```
    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(socket, SIGNAL(connected()), this, SLOT(connecte()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(deconnecte()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)), this,
SLOT(erreurSocket(QAbstractSocket::SocketError)));

    tailleMessage = 0;
}

// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("<em>Tentative de connexion en cours...</em>"));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il y en a
    // On se connecte au serveur demandé
    socket->connectToHost(serveurIP->text(), serveurPort->value());
}

// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text() +tr("</strong> : ") +
message->text();
    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet

    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
}

// Appuyer sur la touche Entrée a le même effet que cliquer sur le bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}

// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    /* Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier
    (en se basant sur la taille annoncée tailleMessage)
    */
    QDataStream in(socket);
    if ( tailleMessage == 0 )
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
             return;
        in >> tailleMessage;
    }

    if ( socket->bytesAvailable() < tailleMessage )
```

```
        return;

    // Si on arrive jusqu'à cette ligne, on peut récupérer le message entier
    QString messageRecu;
    in >> messageRecu;
    // On affiche le message sur la zone de Chat

    listeMessages->append(messageRecu);
    // On remet la taille du message à 0 pour pouvoir recevoir de futurs messages
    tailleMessage = 0;
}

// Ce slot est appelé lorsque la connexion au serveur a réussi
void FenClient::connecte()
{
    listeMessages->append(tr("<em>Connexion réussie !</em>"));
    boutonConnexion->setEnabled(true);
}

// Ce slot est appelé lorsqu'on est déconnecté du serveur
void FenClient::deconnecte()
{
    listeMessages->append(tr("<em>Déconnecté du serveur</em>"));
}

// Ce slot est appelé lorsqu'il y a une erreur
void FenClient::erreurSocket(QAbstractSocket::SocketError erreur)
{
    switch(erreur) // On affiche un message différent selon l'erreur qu'on nous indique
    {
        case QAbstractSocket::HostNotFoundError:
            listeMessages->append(tr("<em>ERREUR : le serveur n'a pas pu être trouvé.
Vérifiez l'IP et le port.</em>"));
            break;
        case QAbstractSocket::ConnectionRefusedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a refusé la connexion.
Vérifiez si le programme \"serveur\" a bien été lancé. Vérifiez aussi l'IP et le
port.</em>"));
            break;
        case QAbstractSocket::RemoteHostClosedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a coupé la
connexion.</em>"));
            break;
        default:
            listeMessages->append(tr("<em>ERREUR : ") + socket->errorString() +
tr("</em>"));
    }

    boutonConnexion->setEnabled(true);
}
```