8. Programmation réseau avec Qt

8.1. Introduction

La programmation réseau est une discipline qui permet à des applications, réparties sur différents dispositifs, de communiquer entre elles via un réseau. Chaque dispositif sur ce réseau est identifié de manière unique par une **adresse IP**, tandis qu'un **numéro de port** permet de cibler un processus ou une application spécifique sur ce dispositif.

Exemple : Considérons un ordinateur disposant de l'adresse IP `172.18.58.14`. Sur cette machine :

- Le **port 80** va souvent associer à une application serveur web (HTTP).
- Le **port 3306** est utilisé par défaut pour une application serveur de base de données (MySQL).

  Une seule machine peut héberger plusieurs applications capables de communiquer via le réseau. Le numéro de port est crucial pour différencier ces applications. Cela permet, par exemple, à un navigateur web de se connecter au serveur web sur le port 80, tandis qu'une application de gestion de bases de données accède au serveur MySQL via le port 3306.

Dans ce contexte, deux types d'applications sont généralement définis : les **applications dites serveur** et les **applications dites client**.

La programmation réseau repose sur des protocoles définis pour chaque couche du modèle OSI (Open Systems Interconnection), en particulier :

- **Application** : Cette couche Interagit avec le logiciel utilisateur. C'est à ce niveau que se situent les protocoles comme HTTP (pour la navigation web), FTP (pour le transfert de fichiers), ou SMTP (pour l'envoi d'e-mails). C'est également à ce niveau que va se situer les applications à développer.
- **Transport** : Cette couche assure la transmission des données entre dispositifs en utilisant des protocoles comme TCP (Transmission Control Protocol) et UDP (User Datagram Protocol).
- **Réseau** : Cette couche gère le routage des paquets à travers différents réseaux, en s'appuyant sur l'adresse IP de chaque dispositif pour l'identifier sur le réseau.

8.2. Utilisation des sockets

Un **socket** est un point de communication logiciel entre deux machines, permettant d'envoyer et de recevoir des données via le réseau. Les sockets sont créés par des appels système et utilisent principalement les deux protocoles :

- **TCP** (Transmission Control Protocol) : Il fournit une communication fiable et orientée connexion. Ce protocole garantit la livraison des paquets dans le bon ordre et permet la retransmission en cas de perte. Il est adapté aux applications où la transmission doit être fiable, comme les serveurs web ou les bases de données.
- **UDP** (User Datagram Protocol) : Ce protocole est plus léger, sans connexion, il permet une communication rapide. Il n'y a ni garanti de livraison, ni d'ordre des paquets, ce qui rend UDP adapté aux applications sensibles à la latence, comme la diffusion vidéo, le gaming en ligne, ou toute application en temps réel. Sa faible surcharge le rend aussi utilisable pour des applications qui tolèrent une perte de données occasionnelle.

8.3. Fonctionnement des sockets TCP

Les sockets TCP permettent une communication fiable, en établissant une connexion entre le client et le serveur avant de transmettre les données. Les étapes clés sont :

La connexion

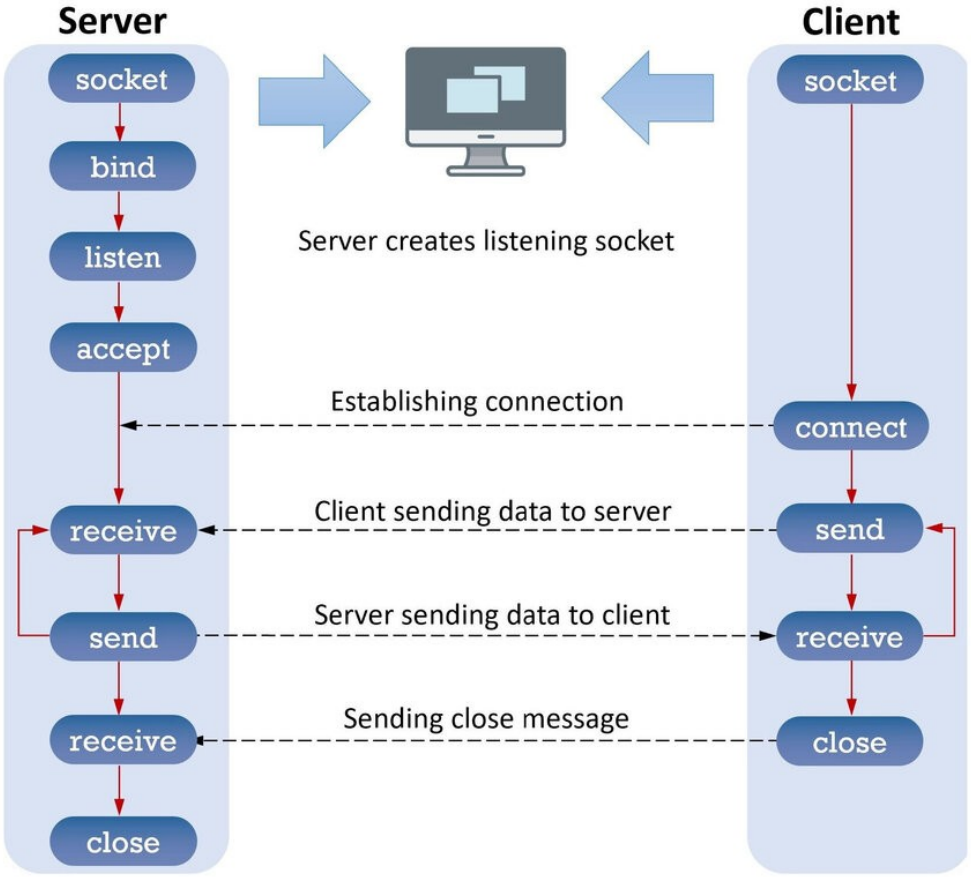
- **Le serveur** utilise **`bind()`** pour assigner une adresse et un port, **`listen()`** pour attendre les connexions entrantes. Lorsqu'un client se connecte, le serveur appelle **`accept()`** pour établir la connexion.
- **Le client** utilise **`connect()`** pour se lier à l'adresse IP et au port du serveur.

La communication

- La communication entre le client et le serveur se fait à l'aide de **`send()`** pour l'envoi et **`recv()`** pour la réception de données. TCP garantit que les données sont reçues dans le même ordre que celui de leur envoi.

La fermeture de la connexion

- Une fois la communication terminée, chaque côté ferme sa connexion avec **`close()`** pour libérer les ressources allouées.



8.4. Fonctionnement des Sockets UDP

8.4.1. Principes

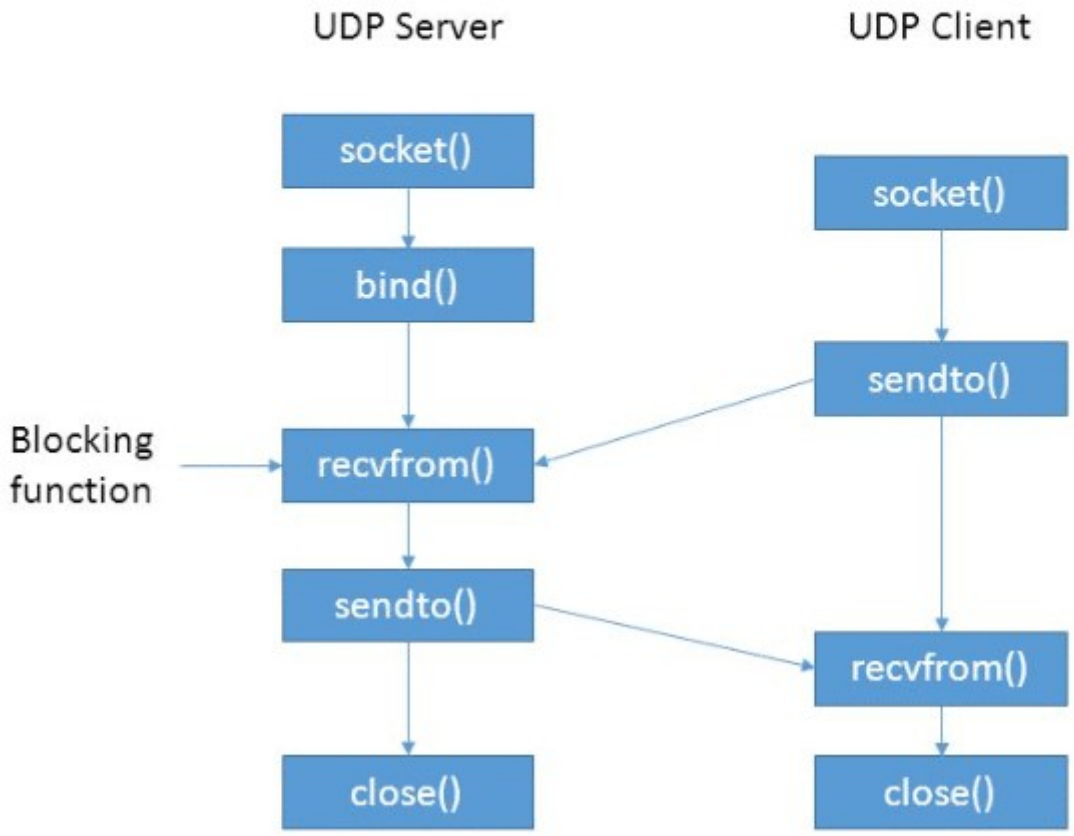
Les sockets UDP sont plus légers que les sockets TCP, mais ils ne fournissent pas les mêmes garanties de fiabilité. Le fonctionnement des sockets UDP est le suivant :

Envoi et réception de datagrammes

- Contrairement à TCP, UDP n'établit pas de connexion formelle entre le client et le serveur. Cela signifie que le client et le serveur peuvent envoyer et recevoir des messages (datagrammes) de manière indépendante, sans confirmation préalable.
- Le client et le serveur envoient les données à l'aide de **`sendto()`** et reçoivent les données avec **`recvfrom()`**. Ces fonctions incluent les informations d'adresse et de port, car chaque datagramme est autonome.

Absence de fiabilité

- UDP n'assure pas la livraison ni l'ordre des paquets. Les paquets peuvent être perdus en chemin ou arriver dans un ordre différent de celui dans lequel ils ont été envoyés.
- Il n'y a pas non plus de mécanisme de retransmission automatique des données en cas de perte, ce qui fait qu'UDP est plus rapide, mais moins fiable.



8.4.2. Types de diffusion : Unicast, Multicast et Broadcast

En **UDP**, les modes de différents mode de diffusion permettent d'envoyer un message soit à un destinataire unique, à un groupe de destinataire, soit à l'ensemble du réseau local.

- **Unicast :** Ce mode est une transmission d'un seul émetteur vers un unique récepteur identifié par son adresse IP et un port associé.
- **Broadcast :** Utile pour envoyer des annonces ou des messages d'état sur tout un réseau local.
- **Multicast :** Avec ce mode, un message est envoyé à un **groupe d'adresses** spécifiques. Il impose de rejoindre un groupe multicast en utilisant une adresse IP dans la plage d'adresses multicast réservées (224.0.0.0 à 239.255.255.255). Ce mode permet de transmettre un message à plusieurs récepteurs en une seule opération, contrairement au mode unicast où chaque récepteur reçoit un message séparé.

  Les adresses multicast sous IPv4 sont dans la plage d'adresses **224.0.0.0 à 239.255.255.255** (bloc d'adresses 224.0.0.0/4). Ces adresses sont spécifiquement réservées pour la communication multicast. Ces adresses sont divisées en plusieurs sous-plages qui ont des rôles spécifiques :

  - **224.0.0.0 à 224.0.0.255 (Multicast local)** : Ces adresses sont utilisées pour des communications multicast locales sur un réseau ou un segment de réseau. Elles ne sont pas routables au-delà du réseau local.

    Exemples :
    `224.0.0.1` : Multicast vers **tous les hôtes** dans un réseau local.
    `224.0.0.2` : Multicast vers **tous les routeurs** dans un réseau local.
  - **224.0.1.0 à 238.255.255.255 (Multicast global)** : Plage utilisée pour des groupes multicast **roulables** à travers des réseaux plus larges, y compris Internet. Ce type de groupe peut être utilisé pour la diffusion de contenu vidéo ou audio à plusieurs clients dans un réseau large.

    Exemple :
    `233.252.0.0` pour des services comme le **Session Description Protocol (SDP)**.
  - **239.0.0.0 à 239.255.255.255 (Administratively Scoped)** : Plage réservée pour des groupes multicast **administrativement restreints** à des réseaux ou organisations spécifiques. Ces adresses sont souvent utilisées pour des communications internes au sein d'une organisation.

  Les adresses multicast peuvent être gérées par des protocoles comme **IGMP** (Internet Group Management Protocol), qui permettent aux récepteurs de **rejoindre ou quitter un groupe multicast**.

8.5. Socket UDP sous Qt

Qt fournit des classes spécifiques pour gérer les communications réseau et faciliter l'utilisation des sockets. Ces classes permettent de construire aussi bien des applications client que des applications serveur de manière efficace et portable. Avec Qt, les sockets sont encapsulés dans des objets qui suivent les principes de la programmation orientée objet, rendant leur utilisation intuitive et intégrée aux applications.

Ces classes de Qt offrent une interface simplifiée pour la gestion des sockets, incluant des fonctionnalités asynchrones basées sur les signaux et slots de Qt. Cela permet de créer des applications réactives sans bloquer le thread principal, essentiel pour les interfaces utilisateur. Qt propose également des fonctionnalités pour la gestion des événements réseau, la surveillance des états de connexion, et le traitement des erreurs, ce qui en fait un framework complet pour la programmation réseau.

Pour les sockets UDP, **`QUdpSocket`** est la classe Qt dédiée aux communications réseau utilisant le protocole UDP (User Datagram Protocol).

8.5.1. Principales méthodes de `QUdpSocket` :

- **Envoi de datagrammes** : La méthode **`writeDatagram`** permet d'envoyer un datagramme à une adresse et un port spécifique.

Envoi d'un datagramme

```
QByteArray data("Message à envoyer");
QHostAddress address("192.168.1.1");
quint16 port = 1234;
udpSocket.writeDatagram(data, address, port);
```

- **Réception de datagrammes** : La méthode **`bind`** est utilisée pour associer le socket à une adresse et un port spécifiques. Cette peut également être utilisée pour associer un socket à une adresse IP spécifique, comme **`QHostAddress::Any`** pour accepter des connexions sur toutes les interfaces réseau. Ensuite, le signal **`readyRead`** est utilisé pour notifier dès qu'un datagramme est disponible. La méthode **`readDatagram`** permet ensuite de le lire.

Réception d'un datagram

```
#include <QUdpSocket>
#include <QHostAddress>
#include <QByteArray>
#include <QDebug>

QUdpSocket udpSocket;

// Utilisation de QHostAddress::Any pour accepter sur toutes les interfaces
// connexion du socket à un port pour écouter les datagrammes entrants (par exemple, 12345)
bool bindSuccess = udpSocket.bind(QHostAddress::Any, 12345);
if (!bindSuccess) {
    qDebug() << "Échec de l'association du socket.";
}
else
{
    // Connexion du signal readyRead pour traiter les datagrammes entrants
    connect(&udpSocket, &QUdpSocket::readyRead, this, &MonObjet::lireDatagram);
}

void MonObjet::lireDatagram()
{
    while (udpSocket.hasPendingDatagrams())
    {  // Vérifie si des datagrammes sont prêts à être lus
        QByteArray buffer;
        buffer.resize(udpSocket.pendingDatagramSize());
        // Redimensionne le buffer pour recevoir le datagramme
        QHostAddress sender;
        quint16 senderPort;
        // Lecture du datagramme
        udpSocket.readDatagram(buffer.data(), buffer.size(), &sender, &senderPort);
        qDebug() << "Reçu de" << sender.toString() << ":" << senderPort << "Données:" << buffer;
    }
}
```

La méthode **`hasPendingDatagrams`** est utilisée pour savoir s'il y a des données dans le tampon de réception, et qu'elle permet d'éviter de lire un datagramme vide ou d'entrer dans une boucle infinie si aucun message n'est disponible.

Dans un programme asynchrone, comme le précédent, il n'est pas nécessaire de lire les datagrammes en permanence. Une interrogation du socket de manière efficace, uniquement lorsque des données sont prêtes à être lues améliore les performances et évite de gaspiller des ressources en attendant indéfiniment de nouvelles données. Le programme ne reste pas bloqué jusqu'à l'arrivée de nouvelle donnée.

La méthode **pendingDatagramSize** permet de connaître la taille du prochain datagramme à lire dans le tampon. Elle est utilisée pour ajuster la taille du tampon avant d'appeler **`readDatagram`**.

8.5.2. Utilisation de la classe QNetworkDatagram

`QNetworkDatagram` est un objet pratique pour manipuler les datagrammes réseau, particulièrement utile avec **`QUdpSocket`** pour centraliser les informations du datagramme comme l'adresse source, le port, et les données elles-mêmes.

- **Création et envoi :** Il est initialisé avec un **`QByteArray`**, puis il est nécessaire de définir son adresse et son port.

Création et envoi

```
QByteArray message("Message à envoyer");
QNetworkDatagram datagram(message, QHostAddress("192.168.1.1"), 1234);
udpSocket.writeDatagram(datagram);
```

- **Lecture et extraction :** Après réception, on peut directement lire un datagramme en tant qu'objet **`QNetworkDatagram`** via **`receiveDatagram`**, ce qui simplifie l'accès aux détails comme l'adresse source.

Lecture et extraction

```
while (udpSocket.hasPendingDatagrams())
{
    QNetworkDatagram datagram = udpSocket.receiveDatagram();
    qDebug() << "Données:" << datagram.data();
    qDebug() << "Adresse source:" << datagram.senderAddress().toString();
}
```

L'utilisation de **`QNetworkDatagram`** en association avec **`QUdpSocket`** simplifie la communication réseau avec le protocole UDP

8.6. Socket TCP sous Qt

8.6.1. Introduction

Les sockets TCP sous Qt permettent de gérer des communications réseau basées sur le protocole TCP, garantissant une connexion fiable et orientée connexion entre un client et un serveur. Qt fournit des classes spécialement conçues pour manipuler les sockets TCP : principalement QTcpSocket et QtcpServer.

La première est utilisée pour représenter un socket client. La classe QTcpSocket permet de se connecter à un serveur, envoyer des données, recevoir des données et gérer les erreurs de connexion.

La seconde est utilisée pour représenter un serveur. La classe QTcpServer écoute les connexions entrantes et émet un signal lorsque des clients se connectent. Elle gère les sockets individuels à l'aide de QTcpSocket.

8.6.2. Fonctionnement de base côté client

Le client utilise QTcpSocket pour se connecter à un serveur, envoyer et recevoir des données, en voici les Étapes principales :

1. **Créer un QTcpSocket :**

```
QTcpSocket *socketClient = new QTcpSocket(this);
```

2. **Se connecter au serveur :**

```
socketClient->connectToHost("127.0.0.1", 1234); // Adresse IP et port
```

3. **Écouter les signaux importants :**

Signaux	Description
`connected()`	La connexion au serveur est établie.
`disconnected()`	La connexion est fermée par le serveur.
`readyRead()`	Des données sont disponibles en lecture.
`stateChanged(QAbstractSocket::SocketState)`	Le socket change d'état (connecté, fermé, etc.).
`errorOccurred(QAbstractSocket::SocketError)`	Une erreur s'est produite.

4. **Envoyer des données :**

```
socketClient->write("Hello, Server");
```

5. **Lire les données reçues :**

```
QByteArray data = socketClient->readAll();
```

6. **Gérer les erreurs :**

```
connect(socket, &QTcpSocket::errorOccurred, this, &Client::onQTcpSocket_ErrorOccured);

void   Client::onQTcpSocket_ErrorOccured(QAbstractSocket::SocketError socketError)
{
         qDebug() << socketClient -> errorString();
}
```

8.6.3. Exemple d'application client

Classe Client

```
#ifndef Client_H
#define Client_H
#include <QObject>
#include <QTcpSocket>
class Client : public QObject
{
    Q_OBJECT
public:
    explicit Client(QObject *parent = nullptr);
    void ConnecterAuServeur(const QString &_host,const quint16 _port);
    void EnvoyerMessage(const QString &_message);
private slots:
    void onQTcpSocket_Connected();
    void onQTcpSocket_Disconnected();
    void onQTcpSocket_ReadyRead();
    void onQTcpSocket_ErrorOccurred(QAbstractSocket::SocketError _socketError);
private:
    QTcpSocket *socket;
};
#endif // Client_H
```

Implémentation de la classe Client

```
#include "Client.h"

Client::Client(QObject *parent)
    : QObject{parent}
    ,socket(new QTcpSocket(this))
{
    connect(socket,&QTcpSocket::connected, this,&Client::onQTcpSocket_Connected);
    connect(socket,&QTcpSocket::disconnected, this,&Client::onQTcpSocket_Disconnected);
    connect(socket,&QTcpSocket::readyRead, this,&Client::onQTcpSocket_ReadyRead);
    connect(socket,&QTcpSocket::errorOccurred, this,&Client::onQTcpSocket_ErrorOccurred);
}

void Client::ConnecterAuServeur(const QString &_host, const quint16 _port)
{
    qDebug() << "Attente de connexion avec : " << _host << " sur le port : "<< _port;
    socket->connectToHost(_host,_port);
}

void Client::EnvoyerMessage(const QString &_message)
{
    if(socket->state() == QAbstractSocket::ConnectedState)
    {
        socket->write(_message.toLatin1());
        qDebug() << "Message envoyé : " << _message;
    }
    else
        qDebug() << "Le message ne peut pas être envoyé au serveur";
}

void Client::onQTcpSocket_Connected()
{
    qDebug() << "Connecté au serveur";
}

void Client::onQTcpSocket_Disconnected()
{
    qDebug() << "Déconnecté du serveur";
}

void Client::onQTcpSocket_ReadyRead()
{
    QByteArray data = socket->readAll();
    qDebug() << "Données reçues du serveur :" << QString(data);
}
```

Implémentation de la classe Client (suite)

```
void Client::onQTcpSocket_ErrorOccurred(QAbstractSocket::SocketError _socketError)
{
    QString messageErreur;

    switch (_socketError) {
        case QAbstractSocket::HostNotFoundError:
            messageErreur = "Hôte non trouvé. Vérifier l'adresse du serveur et le port";
            break;
        case QAbstractSocket::ConnectionRefusedError:
            messageErreur = "Connexion refusée par le serveur";
            break;
        case QAbstractSocket::RemoteHostClosedError:
            messageErreur = "L'hôte distant a fermé la connexion.";
            break;
        case QAbstractSocket::SocketTimeoutError:
            messageErreur = "La connexion a expiré";
            break;
        default:
            messageErreur = "Erreur inattendue : " + socket->errorString();
            break;
    }
    qDebug() << messageErreur;
}
```

Dans le programme principal ci-dessous, après s'être connecté au serveur, le programme attend 2 secondes afin de laisser le temps à la connexion de s'établir.

Programme principal

```
#include <QCoreApplication>
#include "client.h"
#include <QTimer>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Client client;
    client.ConnecterAuServeur("127.0.0.1",1234);
    QTimer::singleShot(2000, [&]() {
            client.EnvoyerMessage("Hello, Serveur !");
        });

    return a.exec();
}
```

Remarque : le serveur doit être lancé avant le client pour que la communication puisse se faire.

8.6.4. Fonctionnement de base côté serveur

Le serveur quant a lui, utilise QTcpServer pour écouter les connexions entrantes. Il gére les clients avec des objets QTcpSocket, en voici les étapes principales :

1. **Créer un QTcpServer :**

```
QTcpServer *serveur = new QtcpServer(this);
```

il est nécessaire de prévoir au moins un attribut QTcpSocket pour gérer la communication avec le client.

```
QTcpSocket *clientSocket;
```

2. **Démarrer l'écoute :**

```
if (!serveur->listen(QHostAddress::Any, 1234)) {
    qDebug() << "Le serveur ne peut pas être lancé : " << serveur->errorString();
} else {
    qDebug() << "Le serveur écoute le port : " << serveur->serverPort();
}
```

3. **Accepter les connexions :**

Le signal `newConnection()` du serveur doit être connecté à un slot pour gérer les nouveaux clients :

```
connect(serveur, &QTcpServer::newConnection, this, &Serveur::onQTcpServer _NewConnection);

void Serveur::  onQTcpServer _NewConnection()
{
   clientSocket = server->nextPendingConnection();
   connect(clientSocket, &QTcpSocket::disconnected, this, &Serveur::onClientDisconnected);
   connect(clientSocket, &QTcpSocket::readyRead, this, &Serveur::onClientReadyRead);
   connect(clientSocket, &QTcpSocket::errorOccurred, this, &Serveur::handleClientError);
}
```

8.6.5. Exemple d'application serveur

L'exemple suivant présente un serveur acceptant un unique client à la fois. Attention, aucune sécurité n'est mis en place en cas de connexion d'un nouveau client, le pointeur sur le client courant est perdu.

Classe Serveur

```
#ifndef Serveur_H
#define Serveur_H
#include <QObject>
#include <QTcpSocket>
#include <QTcpServer>
class Serveur : public QObject
{
    Q_OBJECT
public:
    explicit Serveur(QObject *parent = nullptr);
    void LancerServeur(quint16 _port);
private slots:
    void onQTcpServer_NewConnection();
    void onQTcpSocket_Disconnected();
    void onQTcpSocket_ReadyRead();
    void onQTcpSocket_ErrorOccurred(QAbstractSocket::SocketError _socketError);
private:
    QTcpServer *serveur;
    QTcpSocket *client;
};
#endif // Serveur_H
```

Implémentation de la classe Serveur

```
#include "Serveur.h"

Serveur::Serveur(QObject *parent)
    : QObject{parent}
    , serveur(new QTcpServer(this))
    ,client(nullptr)
{
  connect(serveur, &QTcpServer::newConnection,this,&Serveur::onQTcpServer_NewConnection);
}
void Serveur::LancerServeur(quint16 _port)
{
    if(!serveur->listen(QHostAddress::Any, _port))
        qDebug() << "Echec au lancement du serveur : " << serveur->errorString();
    else
        qDebug() << "Serveur lancé sur le port : "<< QString::number(_port);
}
void Serveur::onQTcpServer_NewConnection()
{
    client = serveur->nextPendingConnection();
    connect(client,&QTcpSocket::readyRead,this,&Serveur::onQTcpSocket_ReadyRead);
    connect(client,&QTcpSocket::disconnected,this,&Serveur::onQTcpSocket_Disconnected);
    connect(client,&QTcpSocket::errorOccurred,this,&Serveur::onQTcpSocket_ErrorOccurred);
}
void Serveur::onQTcpSocket_Disconnected()
{
    qDebug() << "Client déconnecté";
    client->deleteLater();
    client = nullptr;
}
void Serveur::onQTcpSocket_ReadyRead()
{
    if(client)
    {
        QByteArray data = client->readAll();
        qDebug() << "Données reçues : " << QString(data);
        client->write("Message reçu sur le serveur : " + data);
    }
}
void Serveur::onQTcpSocket_ErrorOccurred(QAbstractSocket::SocketError _socketError)
{
    if(client)
    {
        QString messageErreur;
        switch (_socketError) {
        case QAbstractSocket::RemoteHostClosedError:
            messageErreur = "Le client a fermé la connexion";
            break;
        default:
            messageErreur = "Erreur inattendue : " + client->errorString();
            break;
        }
        qDebug() << "Erreur client : " << messageErreur ;
        client->disconnect();
    }
}
```

Programme principal

```
#include <QCoreApplication>
#include "serveur.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Serveur serveur;
    serveur.LancerServeur(1234);

    return a.exec();
}
```

8.6.6. Gestion multiclient avec QTcpServer

La gestion multiclient consiste à permettre à un serveur de gérer plusieurs connexions simultanément. Dans Qt, cela est rendu possible grâce à l'utilisation de `QTcpServer`, qui émet un signal `newConnection` chaque fois qu'un client se connecte. Le serveur doit alors maintenir une liste de clients actifs, généralement en utilisant un conteneur comme `QList<QTcpSocket *>`. Voici quelques principes de base pour implémenter un serveur multiclient :

- **Accepter plusieurs connexions**

  À chaque newConnection, le serveur extrait le socket du client via nextPendingConnection et l'ajoute à une liste.

Acceptation de plusieurs connexions

```
void onNewConnection() {
    QTcpSocket *clientSocket = server->nextPendingConnection();
    clients.append(clientSocket);

    // Connexion des signaux du client
    connect(clientSocket, &QTcpSocket::readyRead, this, &Server::onClientReadyRead);
    connect(clientSocket, &QTcpSocket::disconnected, this, &Server::onClientDisconnected);
    connect(clientSocket, &QTcpSocket::errorOccurred, this, &Server::onClientError);

    qDebug() << "Nouveau client connecté: " << clientSocket->peerAddress().toString();
}
```

- **Gérer les messages de chaque client**

  Un signal readyRead est émis lorsqu'un client envoie des données. Chaque socket peut être distingué grâce au pointeur sender().

Gestion des messages de chaque client

```
void onClientReadyRead() {
    QTcpSocket *clientSocket = qobject_cast<QTcpSocket *>(sender());
    if (clientSocket) {
       QByteArray data = clientSocket->readAll();
       qDebug() << "Données reçues du client : " << QString::fromUtf8(data);
       // Répondre au client
       clientSocket->write("Message reçu : " + data);
    }
}
```

- **Déconnexion et nettoyage**

  Lorsqu'un client se déconnecte, il est important de le retirer de la liste et de nettoyer les ressources associées.

Gestion des messages de chaque client

```
void onClientDisconnected() {
    QTcpSocket *clientSocket = qobject_cast<QTcpSocket *>(sender());
    if (clientSocket) {
       clients.removeAll(clientSocket);
       clientSocket->deleteLater();
       qDebug() << "Client déconnecté !";
    }
}
```

- **Gestion des erreurs pour chaque client**

  Chaque client peut générer des erreurs indépendantes, qui doivent être gérées individuellement.

Gestion des erreurs de chaque client

```
void onClientError() {
    QTcpSocket *clientSocket = qobject_cast<QTcpSocket *>(sender());
    if (clientSocket) {
       qDebug() << "Erreur avec le client :" << clientSocket->errorString();;
    }
}
```

Remarque

Pour obtenir un serveur multiclient robuste, il peut être nécessaire, dans le cas d'une charge intensive, de déplacer chaque QTcpSocket vers un thread distinct afin d'éviter de bloquer l'interface utilisateur ou le flux principal.
Dans tous les cas, il est indispensable de connecter le signal disconnected pour retirer proprement le client de la liste et d'implémenter une logique limitant le nombre maximum de connexions acceptées.

8.6.7. Gestion des nouveaux client avec un serveur TCP mono-client.

Pour garantir une gestion robuste des connexions sur un serveur mono-client, il peut être préférable de refuser proprement une nouvelle connexion plutôt que de permettre au nouveau client de remplacer l'ancien. Voici un exemple de code illustrant cette approche :

Gestion des erreurs de chaque client

```
void Serveur::onNewConnection() {
    QTcpSocket *newClient = serveur->nextPendingConnection();
    if (clientSocket) { // Si un client est déjà connecté, refuser la nouvelle connexion
        newClient->disconnectFromHost();
        if (newClient->state() != QAbstractSocket::UnconnectedState) {
           newClient->waitForDisconnected();
        }
        newClient->deleteLater(); // Libérer les ressources
    }
    else {
       clientSocket = newClient;
       // connexion des différents signaux
    }
}
```