10. Communication réseau utilisant les WebSockets

10.1. Introduction

Les WebSockets sont une technologie permettant une communication **bidirectionnelle** et **en temps réel** entre un client et un serveur via un seul canal TCP. Contrairement au protocole HTTP classique, qui suit un modèle requête-réponse, les WebSockets offrent une connexion persistante, où le **serveur peut initier une transmission** sans qu'une demande préalable du client soit nécessaire. Le protocole WebSocket a été normalisé par l'IETF sous le nom de RFC 6455 en 2011.

Il est particulièrement adapté aux scénarios nécessitant des mises à jour rapides et fréquentes, tels que :

- les **applications de messagerie instantanée**,
- les **jeux en temps réel**,
- les **tableaux de bord de surveillance** (ex. : données financières, capteurs IoT),
- les **notifications instantanées** ou alertes en direct.

Ce type de communication persistante réduit la latence et la surcharge réseau, car il évite les allers-retours continus pour établir une connexion à chaque échange.

Un autre avantage clé des WebSockets est leur capacité à **mélanger les technologies**. Par exemple :

- un serveur implémenté avec **Qt** peut communiquer avec un client Web comme un navigateur,
- ou une interface en **QML** sur un appareil Android peut interagir avec un serveur distant.

Qt fournit une API conviviale pour travailler avec les WebSockets grâce au module QtWebSockets.

10.2. Présentation de l'API Qt WebSockets

10.2.1. Configuration du projet

Pour utiliser les WebSockets dans Qt, le module QtWebSockets dans le fichier .pro :

```
QT += websockets
```

10.2.2. Les classes principales

Qt propose deux classes principales :

- **QWebSocket** est implémentée à la fois coté client et coté Serveur pour communiquer.
- **QWebSocketServer** est implémentée coté serveur afin d'établir la communication.

10.2.3. Client WebSocket sous Qt : QWebSocket

Un objet de type QWebSocket dispose des méthodes pour :

- **Se connecter au serveur** avec `open(QUrl)`.
- **Envoyer des messages** avec `sendTextMessage` ou `sendBinaryMessage`.
- **Recevoir des messages** via les signaux `textMessageReceived` et `binaryMessageReceived`.

Le fonctionnement reste similaire à celui des sockets utilisant `QTcpSocket`

Exemple de client WebSocket version minimale

```
#include <QCoreApplication>
#include <QWebSocket>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QWebSocket client;
    QObject::connect(&client, &QWebSocket::connected, [&client]() {
        qDebug() << "Connecté au serveur WebSocket !";
        client.sendTextMessage("Bonjour serveur!"); });
    QObject::connect(&client, &QWebSocket::textMessageReceived, [&client](const QString &msg) {
        qDebug() << "Message reçu:" << msg; });
    QObject::connect(&client, &QWebSocket::disconnected, &client, [&a]() {
        qDebug() << "Déconnecté du serveur.";
        a.quit(); });
    client.open(QUrl("ws://localhost:12345"));
    return a.exec();
}
```

10.2.4. Serveur WebSocket sous Qt : QWebSocketServer – QWebSocket

Un objet QWebSocketServer permet de gérer plusieurs connexions de clients, il est implémenté pour :

- **Démarrer le serveur** avec listen(QHostAddress, quint16).
- **Accepter les connexions** via le signal newConnection.

Un ou plusieurs objets QWebSocket prend alors le relais pour **gérer les communications** en fonction du nombre de connexions.

Exemple de serveur WebSocket version minimale

```
#include <QCoreApplication>
#include <QWebSocketServer>
#include <QWebSocket>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QWebSocketServer server("Serveur WebSocket", QWebSocketServer::NonSecureMode);

    if (!server.listen(QHostAddress::Any, 12345)) {
        qDebug() << "Échec du démarrage du serveur WebSocket.";
    }
    else {
        qDebug() << "Serveur WebSocket démarré sur le port" << server.serverPort();

        connect(&server, &QWebSocketServer::newConnection, &server, [&server]()
        {
            QWebSocket *client = server.nextPendingConnection();
            qDebug() << "Nouveau client connecté.";
            connect(client, &QWebSocket::textMessageReceived, client, [client](const QString &msg)
            {
                qDebug() << "Message reçu:" << msg;
                client->sendTextMessage("Reçu: " + msg);
            });
            connect(client, &QWebSocket::disconnected, client, [client]()
            {
                qDebug() << "Client déconnecté.";
                client->deleteLater();
            });
        });
    }
}
return a.exec();
}
```

Nom du serveur (arbitraire)

10.2.5. Communication binaire avec WebSockets dans Qt

La communication binaire permet d'échanger des données non textuelles entre un client et un serveur, comme des fichiers, des images, ou des flux de données compressés. Dans Qt, cela est pris en charge via les méthodes sendBinaryMessage et le signal binaryMessageReceived.

- La méthode sendBinaryMessage permet d'envoyer un message binaire sous forme de QByteArray à travers une connexion WebSocket.
- Le signal binaryMessageReceived est émis lorsqu'un message binaire est reçu par le WebSocket. Le contenu du message est transmis sous forme de QByteArray.

Les messages binaires peuvent contenir des structures de données sérialisées utilisant un QDataStream pour un protocole spécifique. Si le message binaire est volumineux, il est nécessaire d'envisager de le fragmenter pour éviter des surcharges réseau ou mémoire. Il peut-être envisagé de compresser ou de chiffrer les données contenu dans le QByteArray pour optimiser ou sécuriser leur transfert, dans ce cas les fonctions associées qCompress() et qUncompress() sont utilisables.

Exemple de transfert d'image avec une compression

```
QFile file("image.png");
if (file.open(QIODevice::ReadOnly)) {
    QByteArray imageData = file.readAll();
    QByteArray compressedData = qCompress(imageData);
    client.sendBinaryMessage(compressedData);
}
}
```

La compression est facultative, elle dépend de la nature des données.

10.2.6. Gestion des erreurs

Comme pour les sockets classiques le signal `errorOccurred` est émis après l'apparition d'une erreur le paramètre transmis avec le signal décrit le type d'erreurs, la fonction `errorString` retourne un message de description textuel.

10.2.7. Interactions entre un serveur sous Qt et une page Web

Les interactions entre un serveur Qt et une page Web sont une application courante des WebSockets. Ces interactions permettent de connecter une application Qt à une interface utilisateur dans un navigateur. Par exemples pour :

- **Un chat en temps réel**
Qt : Le serveur WebSocket reçoit les messages des clients et les rediffuse à tous.
Page Web : Les messages envoyés sont affichés instantanément.
- **Un tableau de bord en direct**
Qt : Le serveur WebSocket envoie des données en provenance de capteurs périodiquement
Page Web : Les graphiques sont mis à jour en temps réel avec une bibliothèque comme HighCharts.
- **Un streaming d'événements**
Qt : Le serveur WebSocket envoie des événements à la page Web, comme des notifications ou des alertes.
Page Web : Les notifications apparaissent instantanément sans recharger la page.

Exemple utilisant le serveur sous Qt précédent :

Page HTML
index.html

```

<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Client WebSocket</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script src="websocket.js"></script>
</head>
<body>
  <h1>Client WebSocket</h1>
  <div>
    <button id="connect">Connecter</button>
    <button id="disconnect" disabled>Déconnecter</button>
  </div>
  <br/>
  <div>
    <textarea id="messageToSend" placeholder="Tapez votre message ici"></textarea>
    <button id="send" disabled>Envoyer</button>
  </div>
  <div>
    <h2>Messages reçus :</h2>
    <pre id="receivedMessages"></pre>
  </div>
</body>
</html>

```

Client WebSocket

Connecter
Déconnecter

Envoyer

Messages reçus :

Le code `websocket.js` ci-après permet de créer une nouvelle instance de `WebSocket` et enregistre les gestionnaires d'événements pour interagir avec le serveur.

Script JavaScript avec JQuery	websocket.js
<pre> \$(document).ready(function () { if (window.WebSocket) { // L'API WebSocket est-elle installée ? var socket; \$('#connect').on('click', function () { socket = new WebSocket("ws://localhost:12345"); socket.onopen = function () { console.log("Connecté au serveur WebSocket."); \$('#connect').prop('disabled', true); \$('#disconnect, #send').prop('disabled', false); }; socket.onmessage = function (event) { console.log("Message reçu :", event.data); \$('#receivedMessages').append("Message reçu : " + event.data + "\n"); }; socket.onerror = function (error) { console.error("Erreur WebSocket :", error); }; socket.onclose = function () { console.log("Déconnecté du serveur WebSocket."); \$('#connect').prop('disabled', false); \$('#disconnect, #send').prop('disabled', true); }; }); \$('#disconnect').on('click', function () { if (socket) { socket.close(); } }); \$('#send').on('click', function () { const message = \$('#messageToSend').val(); if (socket && socket.readyState === WebSocket.OPEN) { socket.send(message); console.log("Message envoyé :", message); } else { alert("La connexion WebSocket n'est pas ouverte."); } }); } else console.log('L\'API WebSocket n\'est pas supportée par ce navigateur.');</pre>	

La classe **WebSocket de JavaScript** est une API qui permet d'établir une communication bidirectionnelle entre un client et un serveur via le protocole WebSocket. Le constructeur de `WebSocket` est utilisé pour créer une nouvelle connexion `WebSocket`.

```
const socket = new WebSocket(url, [protocols]);
```

url (obligatoire) :

- L'adresse du serveur. Elle doit commencer par `ws://` (non sécurisé) ou `wss://` (sécurisé).

protocols (optionnel) :

- Une chaîne ou un tableau de chaînes indiquant les sous-protocoles supportés.
- Permet de spécifier un protocole personnalisé pour structurer les échanges (ex. `json`, `xml`).

Description des événements :

- **onopen** : Déclenché lorsque la connexion au serveur est établie avec succès.
- **onmessage** : Déclenché lorsqu'un message est reçu du serveur.
- **onclose** : Déclenché lorsque la connexion est fermée par le serveur ou le client.
- **onerror** : Déclenché en cas d'erreur dans la connexion `WebSocket`.

État de la connexion : La propriété `readyState` donne l'état de la connexion `WebSocket`. Les valeurs possibles sont :

- 0 : **CONNECTING** – La connexion est en cours d'établissement.
- 1 : **OPEN** – La connexion est ouverte et prête à échanger des données.
- 2 : **CLOSING** – La connexion est en cours de fermeture.
- 3 : **CLOSED** – La connexion est fermée ou n'a pas pu être établie.

La méthode `send(data)` permet l'envoi des données texte ou binaire au serveur.

La méthode `close([code],[reason])` ferme la connexion `WebSocket`. Avec :

- éventuellement un code de fermeture (valeurs courantes : 1000 pour une fermeture normale).
- et éventuellement, une chaîne expliquant la raison de la fermeture.