Chapitre 05 : Programmation UDP

Table des matières

Objectifs....2
Rappels....2
Interfaces de programmation....2
  Les sockets de Berkeley....2
  Les sockets de Windows (*Winsock*)....3
  Bibliothèques externes....3
    SFML....3
Utilisation des sockets UDP de SFML....4
  Classes UdpSocket et Packet....4
  Exemple pas-à-pas : Écho UDP....5
    Serveur....5
    Client....10
  Limiter le temps d’attente pour des données....16
  Exemple complet....18
Index....18
Références....18

Objectifs

- Comprendre la notion de socket
- Comprendre comment implémenter une communication client-serveur en UDP à l'aide du module réseau de SFML

Rappels

Dans le chapitre précédent, nous avons vu que les protocoles de la couche Application sont construits par-dessus les protocoles de la couche Transport (le plus souvent TCP ou UDP). Un protocole de la couche Application détermine donc quelles données sont échangées en TCP ou UDP, dans quels formats et selon quelles règles. Par ailleurs, si la couche Application comprend de nombreux protocoles standardisés (comme HTTP), il est aussi possible de créer son propre protocole selon les besoins de l'application qu'on développe. Dans ce cas, il faut faire le choix du protocole de la couche Transport à utiliser, soit choisir le compromis qui nous convient le mieux entre la fiabilité et la rapidité de transmission.

Nous avons aussi vu que l'échange de données entre différents ordinateurs pose certains défis, puisque la représentation des données peut varier d'un ordinateur à l'autre. Par exemple, un même type numérique en C++ peut être représenté sur un nombre de bits différent selon le compilateur utilisé, et l'ordre dans lequel les octets sont lus (l'*endianness*) peut différer d'une architecture de processeur à une autre. En réponse à ces problèmes, la couche Application utilise la sérialisation, qui consiste à convertir les données dans un format commun avant de les acheminer entre les hôtes.

Finalement, rappelons également que chaque couche du modèle TCP/IP ou du modèle OSI offre un service à sa couche supérieure par le biais d'une interface, qui peut être matérielle ou logicielle. Dans le cas de la couche Transport, l'interface qu'elle met à disposition de la couche Application est logicielle : il s'agit d'une interface de programmation habituellement fournie par le système d'exploitation. Cette interface permet d'envoyer et recevoir des données en UDP ou TCP.

Interfaces de programmation

Les systèmes d'exploitation modernes offrent des interfaces de programmation réseau appelés **sockets**. Un socket permet d'attacher un programme à un port en UDP ou TCP et d'échanger des données sur un réseau à l'aide de ces protocoles.

Les sockets de Berkeley

Les sockets de Berkeley sont la source d'inspiration derrière les interfaces de programmation réseau de la plupart des systèmes d'exploitation existant aujourd'hui. Ils sont apparus au début des années 1980 sur BSD (*Berkeley Software Distribution*), soit le système d'exploitation Unix de l'*Université de Californie à Berkeley.*

Cette interface comprend les fonctions suivantes :

- *socket* → Crée un socket.
- *bind* → Attache un socket à une adresse IP locale et à un port.
- *listen* → Écoute sur un socket pour recevoir les connexions entrantes.
- *connect* → Sur un client, tente d'établir une connexion avec un serveur (TCP).
- *accept* → Sur un serveur, accepte une demande de connexion provenant d'un client (TCP).
- *send* et *sendto* → Envoyer des données (*send* pour TCP et *sendto* pour UDP).
- *recv* et *recvfrom* → Recevoir des données.
- *close* → Ferme le socket. En TCP, ferme la connexion.

Les sockets de Windows (*Winsock*)

Windows possède sa propre implémentation des sockets, l'interface *Winsock*, fortement inspirée des sockets de Berkeley. Ces deux interfaces possèdent donc les mêmes fonctions principales sans toutefois être identiques.

Bibliothèques externes

Les sockets natifs, que ce soit ceux des systèmes de type Unix ou de Windows, sont relativement complexes et fastidieux à utiliser. Heureusement, il existe des bibliothèques (librairies) logicielles qui fournissent leurs propres interfaces de sockets, plus simples à utiliser, puisqu'elles font le travail le plus difficile à notre place.

Dans le cadre du cours, nous utiliserons les sockets de la bibliothèque **SFML**.

SFML

Le site officiel de SFML (pour *Simple and Fast Multimedia Library*) décrit cette bibliothèque ainsi :

> « SFML offre une interface simple vers les différents composants de votre PC, afin de faciliter le développement de jeux ou d'applications multimédia. Elle se compose de cinq modules : système, fenêtrage, graphisme, audio et réseau. »[1]

Cette bibliothèque, écrite en C++, mais disponible pour d'autres langages également, est surtout utilisée pour développer des jeux vidéo en 2D. Dans le cadre du cours, nous utiliserons seulement son module réseau, qui fournit sa propre version des sockets UDP et TCP.

Cette bibliothèque a été choisie pour la grande simplicité d'utilisation de son module réseau, même par rapport à d'autres bibliothèques offrant les mêmes fonctionnalités. En effet, plusieurs bibliothèques de sockets nécessitent la connaissance de concepts de programmation que vous n'avez pas vus dans votre

1 https://www.sfml-dev.org/index-fr.php

parcours jusqu'à maintenant, tels que la programmation asynchrone et le *multithreading*. Ce n'est pas le cas de SFML. Un autre intérêt de SFML est qu'elle fournit son propre mécanisme de sérialisation des données.

Pour utiliser SFML dans un projet, il faut d'abord l'installer. Dans le cadre du cours, des solutions *Visual Studio* pré-configurées pour utiliser le module réseau de SFML vous seront fournies, afin que vous puissiez vous concentrer sur la programmation d'applications réseau sans avoir à perdre du temps pour faire fonctionner une bibliothèque.

La documentation officielle de SFML est disponible ici : https://www.sfml-dev.org/learn-fr.php

Utilisation des sockets UDP de SFML

Classes UdpSocket et Packet

Pour programmer des échanges entre deux programmes en UDP, on utilise principalement deux classes de SFML : **UdpSocket** et **Packet**. La classe UdpSocket, comme son nom l'indique, permet de créer des sockets UDP. La classe Packet, pour sa part, permet d'encapsuler des données pour les transmettre sur le réseau. Il s'agit d'une abstraction, et une instance de Packet ne correspond donc pas directement à un paquet IP.

Le fonctionnement de la classe Packet est assez simple. On instancie un Packet via son constructeur, puis on y insère des données à l'aide de l'opérateur d'entrée (« << »), comme ceci :

```
string prenom = "Bart", nom = "Simpson";
sf::Packet paquet;
paquet << prenom << nom;
```

Les classes de SFML appartiennent à l'espace de noms **sf**, on doit donc préfixer leurs noms de **sf::** dans notre code (ou ajouter une directive « *using namespace sf;* » au début du fichier).

On peut envoyer un Packet directement à un hôte de destination via un socket (nous verrons plus loin comment faire cela). À la destination, on utilise l'opérateur de sortie (« >> ») pour extraire les données :

```
string prenom, nom;
sf::Packet paquet;
paquet >> prenom >> nom;
```

La classe Packet gère la sérialisation pour nous. Cependant, pour qu'elle fonctionne correctement avec les nombres entiers, il faut utiliser les types à taille fixe fournis par la bibliothèque, et non les types natifs de C++. Ces types sont **sf::Int8**, **sf::Int16**, **sf::Int32**, **sf::Int64**, et leurs équivalents non signés (ex : **sf::Uint8**). Le nombre à la fin de chaque type indique la taille du type en bits. Voici un exemple d'utilisation d'un de ces types :

```
string prenom = "Bart", nom = "Simpson";
int age = 10;
sf::Packet paquet;
paquet << prenom << nom << sf::Uint8(age);
```

SFML ne fournit pas de types à virgule (float et double), puisque les machines capables d'exécuter SFML représentent ces types de la même façon nativement.

La classe Packet étant propre à SFML, on ne peut pas l'utiliser dans un programme qui doit communiquer avec des programmes qui n'utilisent pas SFML. Dans ce cas, on peut aussi envoyer et recevoir directement des tableaux d'octets sur les sockets. Nous n'aurons cependant pas à faire cela dans le cadre du cours.

Exemple pas-à-pas : Écho UDP

Serveur

Imaginons que nous voulons programmer un serveur de type « écho » en UDP, c'est-à-dire un serveur qui reçoit un message et le retourne tel quel au client. Commençons par créer la structure de base de notre programme :

```
#include <iostream>
#include <SFML/Network.hpp> ← importe le module réseau de SFML

using namespace std;

int main()
{
}
```

Ajoutons maintenant les variables dont nous aurons besoin :

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket; ← Le socket UDP que nous utiliserons
    sf::IpAddress adresseClient; ← Stockera l'adresse source du client
    sf::Packet paquetEntrant; ← Packet provenant d'un client
    sf::Packet paquetSortant; ← Packet à envoyer à un client

    unsigned short portServeur = 54000; ← Le port sur lequel on écoutera
    unsigned short portClient; ← Stockera le port source du client
```

```
    string message; ← Stockera le message extrait du paquet entrant
}
```

On veut maintenant faire écouter notre socket sur un port. Nous utiliserons le port 54000.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseClient;
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    unsigned short portServeur = 54000;
    unsigned short portClient;

    string message;

    setlocale(LC_ALL, "");

    // Attacher le socket au port du serveur
    if (socket.bind(portServeur) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }

    cout << "Le serveur écoute sur le port " << portServeur
         << "." << endl;
}
```

Ensuite, on veut recevoir des données sur le socket.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
```

```
    sf::IpAddress adresseClient;
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    unsigned short portServeur = 54000;
    unsigned short portClient;

    string message;

    setlocale(LC_ALL, "");

    // Attacher le socket au port du serveur
    if (socket.bind(portServeur) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }

    cout << "Le serveur écoute sur le port " << portServeur
         << "." << endl;

    // socket.receive est bloquant.
    // L'exécution du programme ne continuera pas tant que
    // des données ne seront pas reçues.
    socket.receive(paquetEntrant, adresseClient, portClient);

    Rendu ici, paquetEntrant contiendra les données envoyées par le
    client, adresseClient contiendra l'adresse IP source du client,
    et portClient contiendra le port source.
}
```

La prochaine étape est d'extraire le contenu du Packet :

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseClient;
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    unsigned short portServeur = 54000;
```

```
    unsigned short portClient;

    string message;

    setlocale(LC_ALL, "");

    // Attacher le socket au port du serveur
    if (socket.bind(portServeur) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }

    cout << "Le serveur écoute sur le port " << portServeur
         << "." << endl;

    // socket.receive est bloquant.
    // L'exécution du programme ne continuera pas tant que
    // des données ne seront pas reçues.
    socket.receive(paquetEntrant, adresseClient, portClient);

    paquetEntrant >> message;

    cout << adresseClient << " a envoyé: " << message << endl;
}
```

On peut maintenant renvoyer le même message au client.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseClient;
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    unsigned short portServeur = 54000;
    unsigned short portClient;

    string message;

    setlocale(LC_ALL, "");
```

```
    // Attacher le socket au port du serveur
    if (socket.bind(portServeur) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }

    cout << "Le serveur écoute sur le port " << portServeur
         << "." << endl;

    // socket.receive est bloquant.
    // L'exécution du programme ne continuera pas tant que
    // des données ne seront pas reçues.
    socket.receive(paquetEntrant, adresseClient, portClient);

    paquetEntrant >> message;

    cout << adresseClient << " a envoyé: " << message << endl;

    // Renvoyer le même message au client
    paquetSortant << message;
    socket.send(paquetSortant, adresseClient, portClient);
}
```

Dans son état actuel, notre serveur accepte un seul message, puis s'arrête après avoir renvoyé ce message au client l'ayant envoyé. On veut plutôt qu'il continue de traiter les messages des clients jusqu'à ce qu'on l'arrête. Nous allons donc placer la logique du serveur dans une boucle infinie.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseClient;
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    unsigned short portServeur = 54000;
    unsigned short portClient;

    string message;
```

```
    setlocale(LC_ALL, "");

    // Attacher le socket au port du serveur
    if (socket.bind(portServeur) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }

    cout << "Le serveur écoute sur le port " << portServeur
         << "." << endl;

    // Tant que le programme n'est pas arrêté par l'utilisateur
    while (true) {
        // socket.receive est bloquant.
        // L'exécution du programme ne continuera pas tant que
        // des données ne seront pas reçues.
        socket.receive(paquetEntrant, adresseClient, portClient);

        paquetEntrant >> message;

        cout << adresseClient << " a envoyé: " << message << endl;

        // Renvoyer le même message au client
        paquetSortant << message;
        socket.send(paquetSortant, adresseClient, portClient);

        // On efface le contenu de paquetSortant pour pouvoir le
        // réutiliser à la prochaine itération
        paquetSortant.clear();
    }
}
```

Notre serveur est maintenant entièrement fonctionnel. On peut donc passer au client.

Client

Voici un exemple de l’exécution souhaitée pour le client :

```
Écho Réseau
===========

Le client écoute sur le port 59876.

Entrer un message: J’aime la programmation!
Le serveur (127.0.0.1:54000) a retourné: J’aime la programmation!
```

```
Entrer un message:
```

Commençons par mettre en place la structure de base du programme et créer les variables dont nous aurons besoin :

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseServeur = "127.0.0.1";
                                    On utilise l'adresse 127.0.0.1 pour
                                    le serveur, car on l'exécute sur
                                    la même machine que le client.
    sf::IpAddress adressePaquetEntrant; ← Stockera l'adresse source des
                                          données reçues (normalement
                                          l'adresse du serveur)

    unsigned short portServeur = 54000; ← Doit être le même qu'indiqué
                                          dans le code du serveur
    unsigned short portClient; ← Stockera le port source qui sera utilisé
                                 pour envoyer des messages au serveur
    unsigned short portPaquetEntrant; ← Stockera le port source des
                                        données reçues (normalement
                                        le port du serveur)

    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;
}
```

On doit maintenant attacher le socket à un port. Puisqu'il s'agit du client, on ne veut pas utiliser un port en particulier, on veut plutôt s'en faire attribuer un par le système d'exploitation.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
```

```
{
    sf::UdpSocket socket;
    sf::IpAddress adresseServeur = "127.0.0.1";
    sf::IpAddress adressePaquetEntrant;

    unsigned short portServeur = 54000;
    unsigned short portClient;
    unsigned short portPaquetEntrant;

    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;

    setlocale(LC_ALL, "");

    // Attacher le socket à un port
    // AnyPort indique d'utiliser un port assigné par
    // le système d'exploitation
    if (socket.bind(sf::Socket::AnyPort) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }
    portClient = socket.getLocalPort();


    cout << "Écho Réseau" << endl;
    cout << "===========" << endl << endl;

    cout << "Le client écoute sur le port " << portClient << "."
         << endl << endl;
}
```

Ensuite, on veut lire un message au clavier, puis l'envoyer au serveur.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseServeur = "127.0.0.1";
```

```
    sf::IpAddress adressePaquetEntrant;

    unsigned short portServeur = 54000;
    unsigned short portClient;
    unsigned short portPaquetEntrant;

    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;

    setlocale(LC_ALL, "");

    // Attacher le socket à un port
    // AnyPort indique d'utiliser un port assigné par
    // le système d'exploitation
    if (socket.bind(sf::Socket::AnyPort) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }
    portClient = socket.getLocalPort();

    cout << "Écho Réseau" << endl;
    cout << "===========" << endl << endl;

    cout << "Le client écoute sur le port " << portClient << "."
         << endl << endl;

    cout << "Entrer un message: ";
    getline(cin, messageSortant);
    paquetSortant << messageSortant;

    socket.send(paquetSortant, adresseServeur, portServeur);
}
```

Une fois qu'on a envoyé un message au serveur, on veut attendre de recevoir une réponse de celui-ci, puis on veut afficher cette réponse.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
```

```
{
    sf::UdpSocket socket;
    sf::IpAddress adresseServeur = "127.0.0.1";
    sf::IpAddress adressePaquetEntrant;

    unsigned short portServeur = 54000;
    unsigned short portClient;
    unsigned short portPaquetEntrant;

    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;

    setlocale(LC_ALL, "");

    // Attacher le socket à un port
    // AnyPort indique d'utiliser un port assigné par
    // le système d'exploitation
    if (socket.bind(sf::Socket::AnyPort) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }
    portClient = socket.getLocalPort();

    cout << "Écho Réseau" << endl;
    cout << "===========" << endl << endl;

    cout << "Le client écoute sur le port " << portClient << "."
         << endl << endl;

    cout << "Entrer un message: ";
    getline(cin, messageSortant);
    paquetSortant << messageSortant;

    socket.send(paquetSortant, adresseServeur, portServeur);

    socket.receive(paquetEntrant, adressePaquetEntrant,
                   portPaquetEntrant);

    paquetEntrant >> messageEntrant;

    cout << "Le serveur ("
         << adressePaquetEntrant << ":" << portPaquetEntrant
```

```
        << ") a retourné : " << messageEntrant << endl;
}
```

Finalement, plaçons le tout dans une bouche « while (true) » afin de redemander un message à l'utilisateur après chaque réponse du serveur.

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::IpAddress adresseServeur = "127.0.0.1";
    sf::IpAddress adressePaquetEntrant;

    unsigned short portServeur = 54000;
    unsigned short portClient;
    unsigned short portPaquetEntrant;

    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;

    setlocale(LC_ALL, "");

    // Attacher le socket à un port
    // AnyPort indique d'utiliser un port assigné par
    // le système d'exploitation
    if (socket.bind(sf::Socket::AnyPort) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }
    portClient = socket.getLocalPort();

    cout << "Écho Réseau" << endl;
    cout << "===========" << endl << endl;

    cout << "Le client écoute sur le port " << portClient << "."
         << endl << endl;

    // Tant que le programme n'est pas arrêté par l'utilisateur
```

```
    while (true) {
        cout << "Entrer un message: ";
        getline(cin, messageSortant);
        paquetSortant << messageSortant;

        socket.send(paquetSortant, adresseServeur, portServeur);

        socket.receive(paquetEntrant, adressePaquetEntrant,
                  portPaquetEntrant);

        paquetEntrant >> messageEntrant;

        cout << "Le serveur ("
             << adressePaquetEntrant << ":" << portPaquetEntrant
             << ") a retourné : " << messageEntrant << endl;

        // On efface le contenu de paquetSortant pour pouvoir le
        // réutiliser à la prochaine itération
        paquetSortant.clear();
    }
}
```

Limiter le temps d'attente pour des données

Le client dans notre exemple a une faiblesse : une fois qu'on a envoyé un message, le programme bloque indéfiniment jusqu'à ce qu'il ait reçu une réponse du serveur. Or, puisqu'on est en UDP, il est possible qu'on ne reçoive jamais de réponse! Il serait donc plus judicieux de se donner un temps d'attente maximum au-delà duquel on passe à autre chose si on n'a pas reçu de réponse. Pour ce faire, nous pouvons utiliser la classe **SocketSelector** :

```
#include <iostream>
#include <SFML/Network.hpp>

using namespace std;

int main()
{
    sf::UdpSocket socket;
    sf::SocketSelector selecteur;
    sf::IpAddress adresseServeur = "127.0.0.1";
    sf::IpAddress adressePaquetEntrant;

    unsigned short portServeur = 54000;
    unsigned short portClient;
    unsigned short portPaquetEntrant;
```

```
    sf::Packet paquetEntrant;
    sf::Packet paquetSortant;

    string messageSortant;
    string messageEntrant;

    setlocale(LC_ALL, "");

    // Attacher le socket à un port
    // AnyPort indique d'utiliser un port assigné par
    // le système d'exploitation
    if (socket.bind(sf::Socket::AnyPort) != sf::Socket::Done) {
        cout << "Une erreur est survenue lors de la création du socket."
             << endl;
        return 1;
    }
    portClient = socket.getLocalPort();

    // On associe le socket à un sélecteur.
    // Celui-ci permettra de mettre un temps d'attente maximal
    // sur la réception de données.
    selecteur.add(socket);

    cout << "Écho Réseau" << endl;
    cout << "===========" << endl << endl;

    cout << "Le client écoute sur le port " << portClient << "."
         << endl << endl;

    // Tant que le programme n'est pas arrêté par l'utilisateur
    while (true) {
        cout << "Entrer un message: ";
        getline(cin, messageSortant);
        paquetSortant << messageSortant;

        socket.send(paquetSortant, adresseServeur, portServeur);

        // On attend la réception de données sur le socket
        // pour un maximum de 5 secondes.
        // Passé ce délai, la méthode `wait` retournera false.
        if (selecteur.wait(sf::seconds(5))) {
            socket.receive(paquetEntrant, adressePaquetEntrant,
                   portPaquetEntrant);

            paquetEntrant >> messageEntrant;
```

```
            cout << "Le serveur ("
                 << adressePaquetEntrant << ":" << portPaquetEntrant
                 << ") a retourné : " << messageEntrant << endl;
        }
        else {
            cout << "Le serveur n'a retourné aucune réponse "
                 << "après 5 secondes." << endl;
        }

        // On efface le contenu de paquetSortant pour pouvoir le
        // réutiliser à la prochaine itération
        paquetSortant.clear();
    }
}
```

Exemple complet

L'exemple complet du client et du serveur d'écho UDP est disponible sur l'organisation GitHub du cours.

Index

sockets....2
UdpSocket....4
Packet....4
sf....4
SocketSelector....18

Références

- TANENBAUM, Andrew et WETHERALL, David, *Réseaux, 5e édition*, Paris, Pearson Education France, 2011, p. 535-537
- https://fr.wikipedia.org/wiki/Berkeley_sockets
- https://en.wikipedia.org/wiki/Berkeley_sockets
- https://www.man7.org/linux/man-pages/man2/socket.2.html
- https://learn.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock
- https://www.sfml-dev.org/index-fr.php
- https://www.sfml-dev.org/tutorials/2.5/index-fr.php#module-rceseau
- https://www.sfml-dev.org/documentation/2.5.1-fr/group__network.php
- https://github.com/420-PB1-SH/exemple-udp