

Introduction

Introduction

Par Bousk 

Date de publication : 18 mai 2016

Dernière mise à jour : 24 août 2021

Le but de cette série est de démystifier l'utilisation du réseau et de vous permettre d'en profiter dans vos applications. Il s'agit de la version réécrite et améliorée d'un cours dispensé aux étudiants de l'ESGI Paris en 2015.

Chaque partie présente simplement une composante (connexion, envoi de données...) et vous permet de comprendre son fonctionnement, puis de la mettre en pratique immédiatement, via un TP et des codes sources fournis. À la fin de cette série, vous serez en possession des briques élémentaires nécessaires à la mise en place d'échanges réseau dans votre programme, sous forme de classes C++ utilisant l'API socket de votre système.

Dans un premier temps, nous apprendrons à appréhender les échanges en TCP, d'abord en tant que simple client, puis comme serveur.

Ensuite nous verrons l'utilisation d'UDP.

Enfin certaines techniques plus spécifiques aux jeux vidéos seront présentées sous forme d'articles.

Commentez

I - Réseau : que cache ce terme ?.....	3
II - Communiquer sur le réseau : une histoire de protocoles.....	3
III - Multithreading.....	3
IV - Portée du cours.....	4
V - Chapitres.....	4
V-A - TCP.....	4
V-B - UDP.....	4
V-C - Jeux.....	4
V-D - Divers.....	4
VI - Remerciements.....	5

I - Réseau : que cache ce terme ?

La programmation réseau peut avoir une définition différente selon les personnes. Il peut s'agir de développement serveur, client, P2P, web, web services... mais aussi de base de données, d'administration de base de données, ou encore d'administration de serveurs, de mise en place de cron (appelé aussi "crontab"), d'installation des serveurs (de jeu, de lobby...) sur des machines.

Cette liste non exhaustive donne une vision des connaissances liées au domaine du réseau que l'on peut avoir. Les connaissances nécessaires dépendront essentiellement de la société/structure que vous intégrerez : une large structure pourra découper le travail entre plusieurs membres plus spécialisés, là où dans un studio indépendant de petite taille vous vous retrouverez souvent à programmer le serveur, le client, à gérer la base de données, voire installer et gérer les serveurs (machines).

II - Communiquer sur le réseau : une histoire de protocoles

La communication sur internet et sur un réseau de manière générale, est effectuée par des paquets IP (Internet Protocole). Les paquets IP sont ensuite acheminés par le routeur jusqu'à leur destination via les réseaux électriques. Bien que la création de paquets IP directement soit possible, on préférera manipuler des surcouches comme UDP ou TCP.

L'UDP (User Datagram Protocol - protocole de datagramme utilisateur) est un protocole "orienté hors connexion", ou "en mode non connecté". Il s'agit simplement d'envoyer des données à une machine distante telle une bouteille à la mer, en mode "fire and forget". Les paquets peuvent être perdus et ne jamais arriver, arriver en plusieurs exemplaires ou dans le désordre. La seule garantie est que, si un paquet arrive, il arrive entièrement et non en partie. Dans tous les cas l'expéditeur n'a aucun retour pour le lui indiquer. Pas plus que le destinataire n'a conscience d'une perte ou que ce qu'il reçoit en triple exemplaire est une duplication d'un unique message original.

C'est idéal pour les données dont l'utilité est limitée dans le temps, dans le cas d'un streaming, par exemple, ou d'un jeu vidéo où seules les dernières positions nous intéressent.

Le TCP (Transmission Control Protocol - protocole de contrôle des transmissions) est un protocole "orienté connexion", ou "en mode connecté" et est un flux bidirectionnel : quand A est connecté à B, B est également connecté à A. De plus, il s'agit d'un protocole dit "fiable" puisqu'il nous assure que toutes les données envoyées seront reçues sans perte et dans le bon ordre. Tant que la connexion n'est pas perdue entre les deux parties, elles finiront forcément par arriver.

TCP sera principalement utilisé dans les architectures client/serveur, lorsqu'une connexion continue avec le serveur distant (et donc du serveur vers le client) est nécessaire. Mais la mise en place de ce mécanisme a un coût, notamment en temps nécessaire à l'acheminement des données.

D'autres protocoles basés sur IP existent, vous pouvez en voir la liste complète ici : https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers. En pratique, vous utiliserez toujours TCP ou UDP, sauf si vous travaillez dans une branche extrêmement précise (ICMP pour la gestion des erreurs, par exemple). Aussi ils ne seront pas abordés dans ce cours qui traite uniquement de TCP et UDP.

III - Multithreading

Le code réseau se retrouve le plus souvent en milieu multithread. Pour un serveur, il pourra s'agir d'un thread qui écoute les connexions et d'un autre qui gère les données qui arrivent. Chez un client on pourra recevoir en continu les données dans un thread dédié afin qu'un autre (éventuellement le thread principal) puisse en disposer quand il le souhaite et qu'elles sont prêtes.

Comme bibliothèque de threading je vous propose **TBB** (Threading Building Block).

Ou tout simplement la bibliothèque standard qui le propose désormais.

Les seules fonctions nécessaires seront:

- la **classe thread** pour exécuter une fonction dans un thread;
- la **fonction join** pour attendre la fin de l'exécution du thread;
- les **mutex** - les **mutex simples** feront largement l'affaire - et leurs deux méthodes:
 - **lock**,
 - **unlock**.

Les verrous sont à éviter autant que possible: ils entraînent une baisse de performances, de la contention et, dans le pire des cas non maîtrisé, un deadlock. Toutefois, il existe des cas où leur utilisation est indispensable et je vous invite à savoir les utiliser plutôt qu'à les craindre.

IV - Portée du cours

Le cours vous permettra de mettre en pratique des échanges via TCP ou bien UDP en C++, en utilisant l'API socket (nommée winsock2 sur Windows), qui est en C.

Chaque partie vous présentera les fonctions nécessaires à sa réalisation, directement suivi d'un TP de mise en pratique pour les appliquer immédiatement.

Travaillant sous Windows, et puisqu'il s'agit du principal écosystème sur lequel les professionnels travaillent, je vous conseille Visual Studio comme IDE. Il est disponible gratuitement pour les étudiants d'universités appartenant à l'Academic Alliance (MSDNAA, renseignez-vous auprès de votre école), mais aussi à tous en version Express/Community.

Les codes fournis sont testés sous Visual Studio 2013 et 2015 dans les chapitres TCP, puis Visual Studio 2017 sur les premiers chapitres UDP et enfin Visual Studio 2019 à partir des chapitres de sérialisation, toujours sous Windows 10 et devraient fonctionner sous Unix et iOS, et les précédentes versions de Windows. N'hésitez pas à me contacter si un problème survient sur un certain compilateur ou OS – en ouvrant un sujet sur le forum C++ - et nous pourrions trouver une solution.

V - Chapitres

V-A - TCP

- **Premiers pas**
- **Envoi et réception de données**
- **Mise en place du protocole**
- **Premiers pas en tant que serveur**
- **Envoi et réception depuis le serveur**
- **Mode non bloquant pour le client**
- **Mode non bloquant pour le client**
- **Quelle architecture de client visée ?**
- **Un premier serveur: miniserveur**

V-B - UDP

- **Premiers pas**
- **Gérer les pertes et duplications d'identifiants**
- **S'assurer du bon fonctionnement de son code: mise en place de tests**
- **Créer son protocole par-dessus UDP**
- **Découpage et unification de paquets et création d'un protocole ordonné**
- **Envoi de paquets ordonné fiable**
- **Combiner tous les protocoles: les canaux de communication**
- **Gérer des connexions entre machines**
- **Debugger une application en réseau**

V-C - Jeux

- **Un premier jeu: Morpion / Tic-Tac-Toe**

V-D - Divers

- **Multi-threading et mutex**

- **Bases de la sérialisation**
- **Sérialisation de bits**
- **Sérialisation avancée**

VI - Remerciements

Merci à **LittleWhite**, **Francis Walter** et **François DORIN** pour les relectures techniques.

Ainsi qu'à **jacques_jean**, **f-leb**, **ClaudeLELOUP** et **Malick SECK** pour les corrections orthographiques.

Cours programmation réseau en C++

Premiers pas avec TCP

Par [Bousk](#)

Date de publication : 26 mai 2016

Dernière mise à jour : 8 mars 2019

TOUT PUBLIC

Dans cette première partie, nous allons apprendre à créer un socket et nous connecter à un serveur via son adresse IP.

Commentez

I - Introduction.....	3
I-A - Le socket TCP.....	3
II - Spécificité Windows : initialisation.....	3
II-A - WSASStartup - int WSASStartup(WORD version, WSADATA* data);.....	3
II-B - WSACleanup - int WSACleanup();.....	3
III - Gestion d'erreurs.....	3
III-A - Windows - int WSAGetLastError();.....	4
III-B - Unix - errno.....	4
IV - Présentation des fonctions utiles.....	4
IV-A - hton*.....	4
IV-B - ntoh*.....	4
V - Manipuler un socket.....	4
V-A - socket - int socket(int family, int type, int protocol);.....	4
V-B - Windows – int closesocket(SOCKET socket);.....	5
V-C - UNIX – int close(int socket);.....	5
VI - Se connecter à une machine distante.....	5
VI-A - Windows - int connect(SOCKET _socket, const struct sockaddr* server, int serverlen);.....	5
VI-B - UNIX – int connect(int _socket, const struct sockaddr* server, socklen_t serverlen);	5
VII - Proposition de corrigé.....	7

I - Introduction

Un socket, qu'il soit TCP ou UDP, sera défini par un simple entier qui devra être passé aux fonctions qui l'utilisent. Sur plateformes UNIX il s'agira d'un descripteur de fichier, mais sur Windows c'est un descripteur de socket. La différence majeure est sur le type utilisé : un descripteur de socket (sous Windows donc) est un entier non signé (`unsigned int`), tandis que sous UNIX il s'agira d'un entier signé (`int`). Cela peut entraîner des avertissements supplémentaires à la compilation lors du portage d'un code d'une plateforme à l'autre.

I-A - Le socket TCP

Un socket TCP représente un lien direct avec une machine distante et est une route d'échange de données bilatérale avec celle-ci.

Un socket TCP sera utilisé pour envoyer des données à la machine distante qu'il représente, mais également pour en recevoir de cette dernière.

Dans cette première partie, nous allons voir comment créer un socket TCP et se connecter à un serveur dont on connaît l'adresse IP et le port.



La plupart des fonctions présentées ici seront également utilisées pour un socket UDP.

II - Spécificité Windows : initialisation

Quelques spécificités existent sur plateforme Windows pour utiliser les sockets.

Il s'agit de deux méthodes particulières à appeler pour démarrer et arrêter la bibliothèque de sockets.

II-A - WSAStartup - `int WSAStartup(WORD version, WSADATA* data);`

Permet d'initialiser la DLL pour utiliser les sockets. La version actuelle est la 2.2. Derrière ce prototype, il s'agit d'effectuer un simple appel de fonction comme suit :

```
WSADATA data;  
WSAStartup(MAKEWORD(2, 2), &data);
```

Retourne un code d'erreur en cas d'échec, 0 sinon.

II-B - WSACleanup - `int WSACleanup();`

Appeler cette fonction à la fin du programme pour libérer la DLL.

```
WSACleanup();
```

Retourne `SOCKET_ERROR` en cas d'erreur, 0 sinon.

III - Gestion d'erreurs

Quand une fonction génère une erreur, dans la majorité des cas, elle retourne -1 sous UNIX ou `SOCKET_ERROR` sous Windows, et met à jour l'indicateur d'erreur de son thread (souvenez-vous qu'il s'agit d'une bibliothèque en C). Pour récupérer la valeur d'erreur correcte, il faut récupérer la dernière erreur ainsi mise à jour.

III-A - Windows - int WSAGetLastError();

Comme son nom l'indique, retourne la dernière erreur survenue dans la bibliothèque de sockets pour le thread appelant la fonction.

Attention, sur Windows, la plupart des codes d'erreurs sont également « spécifiques ». EWOULDBLOCK sera par exemple remplacé par WSAEWOULDBLOCK, etc. La liste complète est disponible sur [le site de la MSDN](#).

```
int error = WSAGetLastError();
```

III-B - Unix - errno

Sur Unix, il suffira de lire la valeur de la variable globale `errno`, disponible dans `errno.h`.

```
#include <errno.h>
int error = errno;
```

IV - Présentation des fonctions utiles

IV-A - hton*

Les fonctions de cette forme sont les fonctions **Host/Home to Network**. Elles servent à convertir les données numériques de la machine en données « réseau ».

Par convention, les communications réseau sont en **big-endian**, c'est-à-dire l'octet de poids fort en premier. On parle aussi de **network byte order**.

Il existe une méthode pour chaque type numérique existant :

Conversion local vers réseau

```
short htons(short value);
long htonl(long value);
```

IV-B - ntoh*

Il s'agit des fonctions inverses des `hton*`. Elles convertissent les données réseau en données Host/Home.

Conversion réseau vers local

```
short ntohs(short value);
long ntohl(long value);
```

V - Manipuler un socket

Avant toute chose, il faut créer le socket à manipuler.

V-A - socket - int socket(int family, int type, int protocol);

Crée un socket avec les paramètres passés.

- `family` définit la famille du socket. Les valeurs principales sont `AF_INET` pour un socket IPv4, `AF_INET6` pour un support IPv6.

- `type` spécifie le type de socket. Les valeurs principales utilisées sont `SOCK_STREAM` pour TCP, `SOCK_DGRAM` pour UDP.
- `protocol` définit le protocole à utiliser. Il sera dépendant du type de socket et de sa famille. Les valeurs principales sont `IPPROTO_TCP` pour un socket TCP, `IPPROTO_UDP` pour un socket UDP.

Retourne `INVALID_SOCKET` sous Windows, -1 sous UNIX, en cas d'erreur, le socket sinon.

Créer un socket

```
SOCKET socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  
if (socket == INVALID_SOCKET)  
    // erreur
```

Une fois qu'on en a fini avec notre socket, il faut le fermer pour indiquer au système qu'il peut disposer de celui-ci. Que le port qu'il utilisait est à nouveau disponible, que les ressources nécessaires à son utilisation peuvent être libérées.

V-B - Windows – `int closesocket(SOCKET socket);`

Ferme le socket précédemment ouvert.

- `socket` est le socket à fermer.

Retourne `SOCKET_ERROR` en cas d'erreur, 0 sinon.

V-C - UNIX – `int close(int socket);`

Ferme le socket.

- `socket` est le socket à fermer.

Retourne -1 en cas d'erreur, 0 sinon.

Effectivement, il s'agit de la simple fonction `close()` utilisée habituellement pour fermer un fichier. Mais comme indiqué en début de partie, sous UNIX les sockets sont de simples descripteurs de fichiers, de simples fichiers. Ce n'est donc pas si surprenant que ça.

VI - Se connecter à une machine distante

VI-A - Windows - `int connect(SOCKET _socket, const struct sockaddr* server, int serverlen);`

Connecte un socket précédemment créé au serveur passé en paramètre.

- `_socket` est le socket à connecter.
- `server` la structure représentant le serveur auquel se connecter.
- `serverlen` est la taille de la structure `server`. Généralement un `sizeof(server)` suffit.

Retourne 0 si la connexion réussit, `SOCKET_ERROR` sinon.

VI-B - UNIX – `int connect(int _socket, const struct sockaddr* server, socklen_t serverlen);`

- `_socket` est le socket à connecter.
- `server` la structure représentant le serveur auquel se connecter.

- `serverlen` est la taille de la structure `server`. `socklen_t` est un type spécifique aux plateformes UNIX et peut être un `int` ou `unsigned int`. Généralement un `sizeof(server)` suffit, nous ne nous attarderons donc pas sur lui pour l'instant.

L'appel à cette fonction, quelle que soit la plateforme, est bloquant tant que la connexion n'a pas été effectuée. Autrement dit : si cette fonction retourne, c'est que votre connexion a été effectuée et acceptée par l'ordinateur distant. Sauf si elle retourne une erreur bien sûr.

Une fois notre socket connecté, il agira comme un identifiant vers la machine distante. Quand nous passerons ce socket en paramètre des fonctions, ce sera pour indiquer que l'on appelle cette fonction à destination de la machine à laquelle il est connecté, pour envoyer des données à cette machine spécifiquement ou recevoir des données qu'elle nous aurait envoyées par exemple. Par abus de langage on parlera de la machine ou du socket qui sert de passerelle vers cette machine indistinctement.

Pour créer le paramètre `server`, on utilise une structure `sockaddr_in` à initialiser ainsi :

Structure de connexion au serveur

```
sockaddr_in server;
server.sin_addr.s_addr = inet_addr(const char* ipaddress);
server.sin_family = AF_INET;
server.sin_port = htons(int port);
```

Que l'on peut ensuite utiliser comme paramètre à connect :

```
If (connect(socket, &server, sizeof(server) != 0)
    // Erreur
```

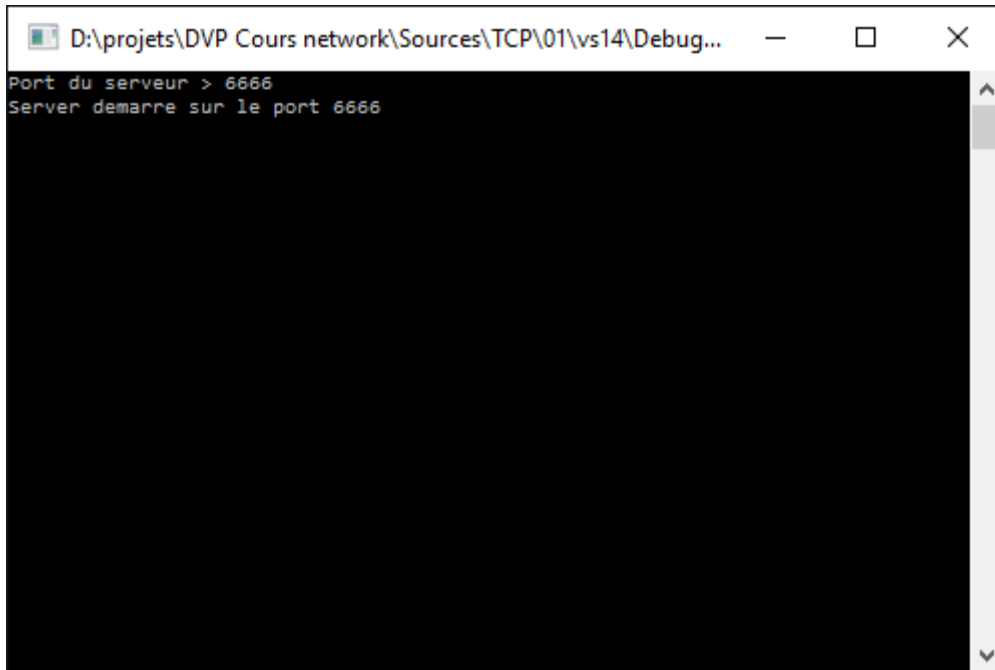
Attention, l'adresse à utiliser avec `inet_addr` est une adresse IP (v4 ou v6) et non un nom de domaine tel que `google.com`. Pour se connecter à un nom de domaine, d'autres manipulations sont à réaliser, que nous verrons plus tard.

Notez également l'utilisation de `htons` pour indiquer le port de destination auquel se connecter.



*Training Time ! Avant d'aller plus loin, pourquoi ne pas déjà s'entraîner ? Lancez ou compilez le **TD 01**. Un serveur se lancera sur le port de votre choix et créez un client capable de se connecter à celui-ci.*

Vous devriez avoir une fenêtre comme ça en lançant le serveur :



```
D:\projets\DVP Cours network\Sources\TCP\01\vs14\Debug...
Port du serveur > 6666
Server démarre sur le port 6666
```

Puis quand un client se connecte, une ligne d'information relative à celui-ci s'affichera :



```
D:\projets\DVP Cours network\Sources\TCP\01\vs14\Debug...
Port du serveur > 6666
Server démarre sur le port 6666
Connexion de 127.0.0.1:51785
-
```

Puisque le serveur est sur la même machine que le client, votre pc, l'IP du serveur sera 127.0.0.1 aussi appelée adresse locale ou de **loopback**.

VII - Proposition de corrigé

Si vous êtes parvenus à vous connecter au serveur (et voir apparaître la ligne correspondante sur sa console), alors c'est que votre code est bon. Toutefois je vous propose comment personnellement j'aurais réalisé ceci.

Tout d'abord, le plus simple (pour moi), ce que j'aurais écrit sous Windows, en travaillant sous Visual Studio :

Main.cpp - premier jet, sous Windows

```

1. #include <iostream>
2. #include <WinSock2.h>
3. #pragma comment(lib, "Ws2_32.lib")
4.
5. int main()
6. {
7.     WSADATA data;
8.     WSAStartup(MAKEWORD(2, 2), &data);
9.
10.    SOCKET socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
11.    if (socket == INVALID_SOCKET)
12.    {
13.        std::cout << "Erreur creation socket : " << WSAGetLastError() << std::endl;
14.        return 0;
15.    }
16.    sockaddr_in server;
17.    server.sin_addr.s_addr = inet_addr("127.0.0.1");
18.    server.sin_family = AF_INET;
19.    server.sin_port = htons(6666);
20.    if (connect(socket, &server, sizeof(server)) == SOCKET_ERROR)
21.    {
22.        std::cout << "Erreur connection : " << WSAGetLastError() << std::endl;
23.        return 0;
24.    }
25.    std::cout << "Socket connecte !" << std::endl;
26.    closesocket(socket);
27.    WSACleanup();
28. }

```

Quelques avertissements peuvent éventuellement survenir, notamment sur l'utilisation de `inet_addr` qui est dépréciée sur les versions récentes de Visual Studio.

Globalement, le code serait identique à peu de choses près pour les autres plateformes. Les seules spécificités à ce niveau sont l'appel à `WSAStartup` pour initialiser la DLL, `WSACleanup` pour la désinitialiser, `WSAGetLastError` pour récupérer l'erreur survenue et `closesocket` pour fermer le socket. Remarquez également que Windows déclare le type `SOCKET`, un `define` sur `unsigned int`, alors que sous UNIX on manipulerait un simple `int`. Servons-nous de la proposition de Windows pour utiliser `SOCKET` dans notre code, qui s'adaptera à la plateforme à la compilation. Faisons de même pour `INVALID_SOCKET` qui, en lisant la documentation de `socket()`, devra valoir -1 et sera un `int` :

```

#ifdef _WIN32
#define SOCKET int
#define INVALID_SOCKET ((int)-1)
#endif

```

Ainsi `SOCKET` sera un `unsigned int` sous Windows grâce à sa déclaration dans `Winsock2.h` que l'on inclut, et sera un `int` sous les autres plateformes via notre `define`. Et on utilisera désormais cette déclaration pour nos sockets.

Puisque ce cours se veut un minimum accessible sur différentes plateformes, faisons en sorte que ce soit le cas maintenant. Comme bien souvent, « l'astuce » consiste simplement à ajouter une indirection.

Ajoutons un fichier `Sockets.h` contenant ceci :

Sockets.h

```

1. #ifndef _WIN32
2. #define SOCKET int
3. #define INVALID_SOCKET ((int)-1)
4. #endif
5.
6. namespace Sockets
7. {
8.     bool Start();
9.     void Release();
10.    int GetError();
11.    bool CloseSocket(SOCKET socket);

```

Sockets.h

```
12. }
```

L'implémentation de chacune des fonctions dépendra de la plateforme, quitte à être vides, mais elles seront utilisables sur toutes. Ainsi le code initial deviendra :

Main.cpp

```
1. #include "Socket.h"
2.
3. #include <iostream>
4. #include <WinSock2.h>
5.
6. int main()
7. {
8.     if (!Sockets::Start())
9.     {
10.         std::cout << "Erreur initialisation : " << Sockets::GetError() << std::endl;
11.         return 0;
12.     }
13.
14. SOCKET socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
15. if (socket == INVALID_SOCKET)
16. {
17.     std::cout << "Erreur creation socket : " << Socket::GetError() << std::endl;
18.     return 0;
19. }
20. sockaddr_in server;
21. server.sin_addr.s_addr = inet_addr("127.0.0.1");
22. server.sin_family = AF_INET;
23. server.sin_port = htons(6666);
24. if (connect(socket, &server, sizeof(server)) == SOCKET_ERROR)
25. {
26.     std::cout << "Erreur connection : " << Socket::GetError() << std::endl;
27.     return 0;
28. }
29. std::cout << "Socket connecte !" << std::endl;
30. Sockets::CloseSocket(socket);
31. Sockets::Release();
32. }
```

Concernant l'implémentation de nos fonctions, ajoutons un fichier Sockets.cpp pour celles-ci :

Sockets.cpp

```
1. #include "Sockets.h"
2. namespace Sockets
3. {
4.     bool Start()
5.     {
6.         #ifdef _WIN32
7.             WSADATA wsaData;
8.             return WSAStartup(MAKEWORD(2, 2), &wsaData) == 0;
9.         #else
10.             return true;
11.         #endif
12.     }
13.     void Release()
14.     {
15.         #ifdef _WIN32
16.             WSACleanup();
17.         #endif
18.     }
19.     int GetError()
20.     {
21.         #ifdef _WIN32
22.             return WSAGetLastError();
23.         #else
24.             return errno;
25.         #endif
26.     }
27. }
```

Sockets.cpp

```
27. void CloseSocket(SOCKET s)
28. {
29.     #ifdef _WIN32
30.         closesocket(s);
31.     #else
32.         close(s);
33.     #endif
34. }
35. }
```

Si la vue d'instructions `ifdef` au milieu du code vous dérange, vous pouvez opter pour avoir un fichier d'implémentation différent selon la plateforme. Ici par simplicité, et préférence personnelle, j'ai choisi de n'utiliser qu'un seul fichier et ces instructions préprocesseurs.

Il ne reste plus que l'inclure de `Winsock2.h` qui traîne, puisque nous avons regroupé nos fonctions dans `Sockets.h`, et qu'il nous servira de porte d'entrée pour utiliser nos sockets, utilisons ce fichier pour inclure le header correct selon la plateforme cible :

Sockets.hpp

```
1. #ifdef _WIN32
2. #if _MSC_VER >= 1800
3. #include <WS2tcpip.h>
4. #else
5. #define inet_pton(FAMILY, IP, PTR_STRUCT_SOCKADDR) (*(PTR_STRUCT_SOCKADDR)) = inet_addr((IP))
6. typedef int socklen_t;
7. #endif
8. #include <WinSock2.h>
9. #ifdef _MSC_VER
10. #if _WIN32_WINNT >= _WIN32_WINNT_WINBLUE
11. //!< Win8.1 & higher
12. #pragma comment(lib, "ws2_32.lib")
13. #else
14. #pragma comment(lib, "wssock32.lib")
15. #endif
16. #endif
17. #else
18. #include <sys/socket.h>
19. #include <netinet/in.h> // sockaddr_in, IPPROTO_TCP
20. #include <arpa/inet.h> // hton*, ntoh*, inet_addr
21. #include <unistd.h> // close
22. #include <cerrno> // errno
23. #define SOCKET int
24. #define INVALID_SOCKET ((int)-1)
25. #endif
26.
27. namespace Sockets
28. {
29.     bool Start();
30.     void Release();
31.     int GetError();
32.     bool CloseSocket(SOCKET socket);
33. }
```

N'oubliez pas que vous devrez également lier `Ws32_2.lib` sous Windows.

Et puisque nous sommes en C++, et qu'un socket s'y prête bien, pourquoi ne pas avoir une classe `Socket` pour le manipuler plus aisément ?

TCPSocket.hpp

```
1. #ifndef TCP_SOCKET_HPP
2. #define TCP_SOCKET_HPP
3. #pragma once
4.
5. #include "Sockets.h"
6.
7. #endif
```

TCPSocket.hpp

```
7. #include <string>
8.
9. class TCPSocket
10. {
11. public:
12.     TCPSocket();
13.     ~TCPSocket();
14.
15.     bool Connect(const std::string& ipaddress, unsigned short port);
16.
17. private:
18.     SOCKET mSocket;
19. };
20.
21. #endif // TCPSOCKET_HPP
```

Avec une telle interface, notre code source initial deviendra alors :

Main.cpp

```
1. #include "TCPSocket.h"
2.
3. #include <iostream>
4.
5. int main()
6. {
7.     if (!Sockets::Start())
8.     {
9.         std::cout << "Erreur initialisation : " << Sockets::GetError() << std::endl;
10.        return 0;
11.    }
12.
13.    {
14.        TCPSocket socket;
15.        if (!socket.Connect("127.0.0.1", 6666))
16.        {
17.            std::cout << "Erreur connection : " << Sockets::GetError() << std::endl;
18.            return 0;
19.        }
20.        std::cout << "Socket connecte !" << std::endl;
21.    }
22.    Sockets::Release();
23. }
```

Beaucoup plus clair n'est-ce pas ?

L'implémentation de notre `TCPSocket` sera très simple à réaliser :

TCPSocket.cpp

```
1. #include "TCPSocket.hpp"
2.
3. TCPSocket::TCPSocket()
4. {
5.     mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
6.     if (mSocket == INVALID_SOCKET)
7.     {
8.         std::ostringstream error;
9.         error << "Erreur initialisation socket [" << Sockets::GetError() << "]";
10.        throw std::runtime_error(error.str());
11.    }
12. }
13. TCPSocket::~TCPSocket()
14. {
15.     Sockets::CloseSocket(mSocket);
16. }
17. bool TCPSocket::Connect(const std::string& ipaddress, unsigned short port)
18. {
19.     sockaddr_in server;
```


TCPSocket.cpp

```
20. server.sin_addr.s_addr = inet_addr(ipaddress.c_str());
21. server.sin_family = AF_INET;
22. server.sin_port = htons(port);
23. return connect(mSocket, &server, sizeof(server)) == 0;
24. }
```

Enfin, pour peaufiner le tout, remplaçons cet `inet_addr` par `inet_pton` comme préconisé par l'avertissement de compilation. `inet_pton` a été introduit avec l'arrivée d'IPv6 et est donc préféré puisqu'il peut traduire une adresse IPv4 ou IPv6, alors que `inet_addr` ne pouvait gérer qu'une adresse IPv4.

Un appel à `inet_addr` sera remplacé par `inet_pton` de la sorte, dans le cas de l'utilisation pour `connect` :

```
sockaddr_in server;
server.sin_addr.s_addr = inet_addr("127.0.0.1");
// équivalent à
inet_pton(AF_INET, "127.0.0.1", &server.sin_addr.s_addr);
```

Remarquez la symétrie d'écriture, nous permettant d'écrire la macro suivante pour n'utiliser que `inet_pton` dans notre code et que celui-ci effectue un appel à `inet_addr` s'il n'est pas disponible :

```
#define inet_pton(FAMILY, IP, PTR_STRUCT_SOCKADDR) (*(PTR_STRUCT_SOCKADDR)) = inet_addr((IP))
```

On peut éventuellement ajouter un `assert` pour que `FAMILY` soit toujours égale à `AF_INET` puisque la valeur `AF_INET6` et les adresses IPv6 ne sont pas permises avec `inet_addr`. Ce n'est pas trop dérangentant puisque si l'adresse est incorrecte, une valeur d'erreur sera de toute façon retournée.

Ainsi notre `TCPSocket::Connect` finale sera :

```
bool TCPSocket::Connect(const std::string& ipaddress, unsigned short port)
{
    sockaddr_in server;
    inet_pton(AF_INET, ipaddress.c_str(), &server.sin_addr.s_addr);
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    return connect(mSocket, (const sockaddr*)&server, sizeof(server)) == 0;
}
```

Nous possédons désormais les fondations pour écrire un client TCP pouvant se connecter à une IP. Ce code évoluera au fil du cours pour ajouter des possibilités à notre application.



Télécharger les codes sources du cours

Article précédent
<< Introduction

Article suivant
TCP – Envoi et réception

Cours programmation réseau en C++

TCP - Envoi et réception de données

Par [Bousk](#)

Date de publication : 21 juin 2016

Dernière mise à jour : 9 mai 2017

TOUT PUBLIC

Maintenant que nous savons nous connecter à un serveur, il est temps de communiquer avec lui : envoyer et recevoir des données.

Commentez

I - Envoyer des données.....	3
II - Recevoir des données.....	3
III - Proposition de corrigé.....	4

I - Envoyer des données

send - int send(int socket, const void datas, size_t len, int flags);*

Envoie des données au socket en paramètre.

- `socket` est le socket auquel envoyer les données.
- `datas` les données à envoyer.
- `len` est la taille maximale des données à envoyer en octets.
- `flags` un masque d'options. Généralement 0.

Retourne le nombre d'octets mis en file d'envoi dans la mémoire tampon du système. Peut retourner 0. Retourne -1 sous Unix, `SOCKET_ERROR` sous Windows, en cas d'erreur.

L'appel à cette fonction est bloquant tant que tous les octets à envoyer n'ont pas été mis en file d'envoi.

send

```
SOCKET socket;
// initialisation et connexion
char buffer[1024];
if (send(socket, buffer, 1024, 0) == -1)
    // erreur
```

II - Recevoir des données

recv - int recv(int socket, void buffer, size_t len, int flags);*

Réceptionne des données sur le socket en paramètre.

- `socket` est le socket duquel réceptionner les données.
- `buffer` est un tampon où stocker les données reçues.
- `len` est le nombre d'octets maximal à réceptionner. Typiquement, il s'agira de la place disponible dans le tampon.
- `flags` est un masque d'options. Généralement 0.

Retourne le nombre d'octets reçus et stockés dans `buffer`. Peut retourner 0 si la connexion a été terminée. Retourne -1 en cas d'erreur.

L'appel à cette fonction est bloquant tant qu'aucune donnée n'est reçue par votre socket et que la connexion n'est pas terminée.

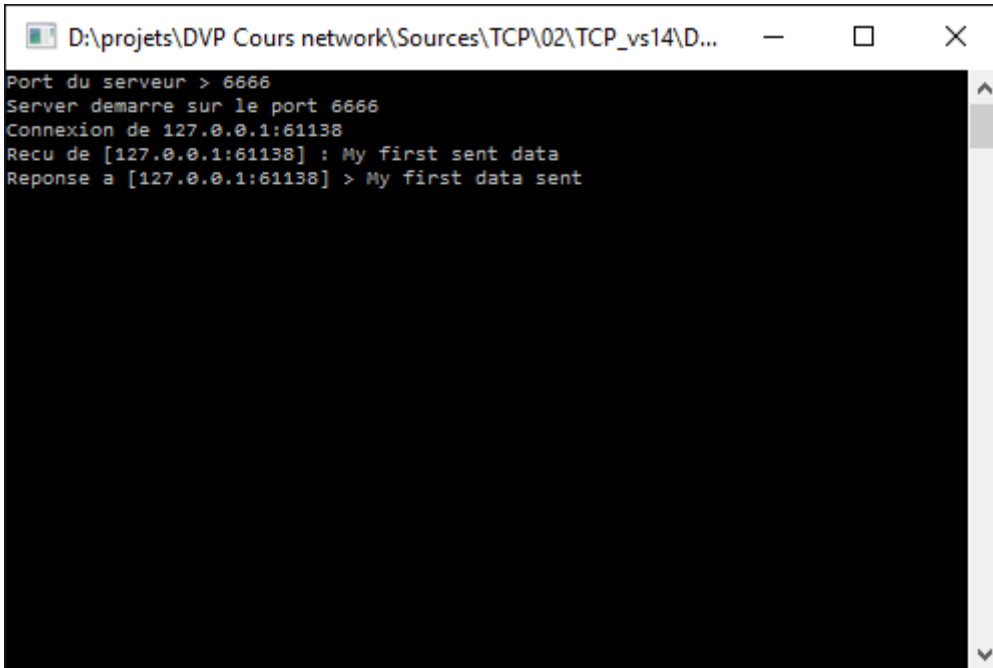
recv

```
SOCKET socket;
// initialisation et connexion
char buffer[1024];
if (recv(socket, buffer, 1024, 0) <= 0)
    // erreur ou connexion fermée
```



Training Time ! Lancez ou compilez le TD 02. Un serveur se lancera sur le port de votre choix. Créez un client pour vous y connecter (vous pouvez, voire devriez, utiliser le précédent en l'améliorant), puis à l'aide de ce client envoyez une phrase. Vous devriez voir la phrase s'afficher dans la console du serveur et il vous renverra la même phrase avec l'ordre des mots modifié aléatoirement.

Vous retrouverez les lignes de la partie précédente relative à la connexion d'un client. Quand le client envoie une phrase, celle-ci sera affichée, ainsi que la version modifiée qui lui est renvoyée.



```

D:\projets\DVP Cours network\Sources\TCP\02\TCP_vs14\D...
Port du serveur > 6666
Server démarre sur le port 6666
Connexion de 127.0.0.1:61138
Recu de [127.0.0.1:61138] : My first sent data
Reponse a [127.0.0.1:61138] > My first data sent
  
```

Vérifiez que vous recevez les données correctement sur votre client.

III - Proposition de corrigé

Nous allons simplement modifier notre classe TCPSocket précédemment créée pour y ajouter le nécessaire à l'envoi et la réception de données.

Pour l'instant, il s'agira d'un simple wrapping autour des fonctions send et recv que nous avons découvertes dans cette partie.

Les prototypes seront donc tout simplement :

prototypes

```

int Send(const char* data, unsigned int len);
int Receive(char* buffer, unsigned int len);
  
```

L'implémentation n'aura rien de particulier non plus :

implémentation

```

int TCPSocket::Send(const char* data, unsigned int len)
{
    return send(mSocket, data, len, 0);
}
int TCPSocket::Receive(char* buffer, unsigned int len)
{
    return recv(mSocket, buffer, len, 0);
}
  
```

Rien d'extraordinaire jusque là, il est temps de passer à la partie suivante :)



Télécharger les codes sources du cours

Article précédent
<< TCP - Premiers-pas

Article suivant
TCP - Mise en place du protocole

Cours programmation réseau en C++

TCP - mise en place de votre protocole

Par [Bousk](#)

Date de publication : 22 novembre 2016

Dernière mise à jour : 9 mai 2017

DÉBUTANT

Dans la courte partie précédente, nous avons appris à utiliser send et recv.

Penchons-nous un peu plus en détail sur leur fonctionnement, et sur l'utilisation de recv en particulier.

Commentez

I - Réception de données ? Quelles données ?.....	3
II - Créer son protocole.....	3
III - Mise en place du protocole.....	4
IV - Puis-je vraiment envoyer 64 ko de données ?.....	4
V - Proposition de corrigé.....	5
V-A - Pourquoi unsigned char ?.....	5

I - Réception de données ? Quelles données ?

Comme vu de par son prototype précédemment, l'appel à `recv` permet de récupérer jusqu'en octets de données.

Comment ça marche ? On ne s'en soucie pas, c'est le fonctionnement interne de TCP et dans sa boîte noire. Tout ce qu'il faut savoir pour le moment, c'est que l'appel à `recv` ne lit pas des données du réseau directement, ce n'est pas possible et c'est le matériel qui s'en charge, mais depuis un tampon système où sont stockées les données reçues de l'ordinateur distant après avoir réalisé toutes les manipulations nécessaires pour s'assurer qu'elles sont cohérentes avec ce qui a été envoyé, dans leur quantité et le bon ordre. Quand ces manipulations sont réussies, le système met les données à disposition de l'application qui peut en extraire une quantité donnée via `recv`.

Comment alors peut-on savoir si les données extraites sont le résultat d'un ou plusieurs appels à `send` ? C'est impossible !

Exemple concret : si A envoie à B « salut » puis « comment va ? » en deux temps (via deux appels à `send`), lorsque B appellera `recv` il pourra extraire « salutcomment va ? » si toutes ces données sont reçues et disponibles. Du point de vue de B, on n'a absolument aucune idée s'il s'agissait d'un ou plusieurs envois, ce qui peut faire une grosse différence (par exemple dans ce cas : « salut » et « comment » se retrouvent collés - ce qui n'est pas très grave dans cet exemple simpliste, mais peut entraîner de gros problèmes en pratique) quant au traitement à faire sur notre tampon.

II - Créer son protocole

On doit alors ajouter un minimum de logique logicielle autour de nos données. Mettre en place son protocole, son formalisme, ses règles d'échange de données.

D'abord un peu de vocabulaire pour s'assurer que l'on parle la même langue. Nous parlons généralement de paquet, ou message, réseau. Le **paquet** sera désormais notre unité d'échange, il s'agit d'un amas de données envoyées/à envoyer à l'ordinateur distant. Un mot, une phrase, le contenu d'un fichier, chacun de ces éléments sera envoyé par paquets. On enverra toute donnée sous forme d'un paquet en TCP, que l'on peut découper manuellement, par exemple si le paquet résultant était trop important, chaque fragment sera alors un paquet nécessaire à reformer la donnée finale.

Il devient donc évident qu'il nous faut un moyen de délimiter les paquets afin d'être capable de les retrouver si l'appel à `recv` nous retourne plusieurs paquets de données à notre insu, ou pour limiter les données à extraire de `recv` à la longueur du paquet attendu.

Il existe en gros trois possibilités :

- utiliser des paquets de taille fixe ;
- utiliser un délimiteur, une suite d'octets, à la fin d'un paquet qui indique sa fin ;
- indiquer au destinataire la taille des données à traiter, en préfixant le paquet de sa taille.

La première option est rejetée de par son inefficacité, en dehors du cas particulier où il s'agira de la meilleure option, et la perte énorme que ça engendre en général. Il faut que la taille choisie soit suffisante pour le paquet le plus gros, donc le gaspillage de mémoire est d'autant plus important que le paquet est petit.

La seconde option est risquée : il faut pouvoir être sûr à 100 % que la suite d'octets choisie ne sera **jamais** présente dans un paquet. Notamment si vous comptez transférer des fichiers, ou du binaire de manière générale, comment vous en assurer ?

Reste la troisième solution, celle que j'ai toujours employée et que nous utiliserons dans ce cours.

III - Mise en place du protocole

La solution retenue est donc de préfixer chaque paquet de sa taille. Pour y parvenir, nous pouvons au choix mettre le nombre en texte plein, ou le sérialiser dans sa forme binaire. Et dans ce dernier cas, quelle taille doit être utilisée ?

L'idée d'utiliser du texte est généralement alléchante, pourtant c'est de loin la moins efficace. Si vos données font 50 octets, vous devrez stocker « 50 », et le délimiteur de fin de chaîne, soit trois octets avant que les données intéressantes de votre paquet ne commencent.

Nous pouvons faire la table suivante pour connaître la taille du paquet en fonction de la longueur du préfixe :

Intervalle de préfixe	Taille du préfixe (octets)	Quantité de données min	Quantité de données max	Taille totale du paquet min (octets)	Taille min formatée	Taille totale du paquet max (octets)	Taille max formatée
1-9	2	1	9	3	3 o	11	11 o
10-99	3	10	99	13	13 o	102	102 o
100-999	4	100	999	104	104 o	1003	0,98 ko
1000-9999	5	1000	9999	1005	0,98 ko	10004	9,77 ko
10000-99999	6	10000	99999	10006	9,77 ko	100005	97,66 ko
100000-999999	7	100000	999999	100007	97,66 ko	1000006	976,57 ko
1000000-9999999	8	1000000	9999999	1000008	976,5 ko	10000007	9,53 Mo

Intéressons-nous à l'intervalle 100-999. Avec un préfixe de quatre octets, nous pouvons envoyer jusqu'à 1003 octets. Ce qui n'est certes pas mal, mais pas forcément évident pour transférer un fichier par exemple. Nous devons monter à huit octets pour pouvoir envoyer un maximum de 9.53 Mo de données.

Alors que quatre octets et huit octets peuvent être représentés respectivement par un entier 32 bits (int32) et 64 bits (int64).

Nous savons aussi que notre taille est forcément positive, il s'agira donc d'entiers non signés (uint32 et uint64). Pour lesquelles les intervalles de valeurs iront jusque 4 294 967 295 pour uint32 max permettant d'envoyer jusqu'au **4096 Mo**, et 18 446 744 073 709 551 615 pour uint64 qui représenteraient **16 777 216 To**. Ils nécessiteraient respectivement 10 et 20 octets, plus le délimiteur de fin de chaîne soit 11 et 21 octets, si on les représentait sous forme de chaîne. Soit un gâchis de 7 et 13 octets. Mais envoyer autant de données est de toute façon farfelu !

Pour ces raisons, nous opterons finalement pour un uint16, codé sur deux octets pour un maximum théorique de **64 ko par paquet**. Il s'agit d'une taille déjà colossale pour un paquet, et si vous décidez d'envoyer une information plus importante que ça (un fichier de plusieurs Mo, ou Go, par exemple) vous devriez de toute façon gérer ça finement et manuellement par souci de quantité de ram utilisée entre autres.

IV - Puis-je vraiment envoyer 64 ko de données ?

Oui et non. Il faut maintenant comprendre plus précisément le fonctionnement de TCP par le système.

Comme indiqué en début de cette partie, recv récupère les données d'un tampon fourni par le système. Ce tampon est appelé *TCP window*. Ce tampon n'est bien entendu pas illimité, et sa taille dépendra de la configuration du système, que l'on retrouvera sous l'appellation de *TCP window size*.

La norme TCP indique que la taille de ce tampon est codée sur 16 bits (2 octets), et donc que la taille maximale théorique est de 64 ko (<https://tools.ietf.org/html/rfc7323#section-1.1>). Mais il s'agit d'une taille maximale et chaque système est libre d'utiliser une taille moins importante, notamment sur les appareils mobiles où ce sera le plus souvent le cas.

Il existe également un autre paramètre *TCP window scale* permettant de multiplier cette taille sur la plupart des systèmes. Là encore, il s'agit de configuration système. Ce multiplicateur sera utilisé principalement pour configurer des réseaux spécifiques, rien qui ne nous intéresse et que l'on puisse réellement utiliser donc.

Si par hasard (ou malheur) votre tampon système se retrouve plein, aucune nouvelle donnée ne pourra être réceptionnée par le système tant que vous n'aurez pas fait appel à `recv` pour en extraire et libérer des données, ou libérer le socket correspondant. Ayant pour effet de bloquer l'envoi par les ordinateurs distants, voire de vous déconnecter. Dans le pire des cas, les ordinateurs distants peuvent également mettre trop de données en attente d'envoi dans leur système, bloquant également tout envoi de données de leur part à leurs destinataires (ce qui peut inclure d'autres machines que la vôtre).

Cette fenêtre influera également sur le débit de données que l'on a par la formule

$$\text{Throughput} \leq \frac{RWIN}{RTT}$$

où *Throughput* est ce débit, *RWIN* la taille du tampon système (défini par *TCP window* et *TCP window scale*) et *RTT* le *Round-Trip Time*, le ping.

Donc s'il est en théorie possible d'envoyer un message de 64 ko, en pratique on n'a que rarement, voire jamais, besoin d'envoyer autant de données en un unique paquet, vous ne devriez donc pas vous retrouver dans ce genre de cas particulier de si tôt. Et si le cas se présente, si vous souhaitez envoyer un fichier par exemple, il suffira de veiller à découper le paquet en plusieurs parties dans la couche logicielle, ou votre moteur réseau, afin de le reconstruire à destination.



Training Time ! Lancez ou compilez le TD 03. Il s'agit d'une version modifiée du serveur de la partie précédente, qui vous retournera votre phrase en mélangeant les mots de celle-ci, mais en prenant en compte le protocole que l'on a introduit dans cette partie.

V - Proposition de corrigé

Nous continuons d'itérer sur notre classe `TCPSocket`.

Améliorons l'interface de *Send* et *Receive* afin d'être plus que de simples wrappers.

Je propose l'interface suivante :

Prototypes

```
bool Send(const unsigned char* data, unsigned short len);
bool Receive(std::vector<unsigned char>& buffer);
```

L'idée étant que ces fonctions retournent `false` en cas d'erreur, `true` sinon. Notez également le passage à de l'`unsigned char` dans les types.

Petit aparté avant de voir l'implémentation que je propose.

V-A - Pourquoi unsigned char ?

Le fait est que le `char`, dans la norme, n'est pas défini comme étant signé ou non signé (<http://www.open-std.org/jtc1/sc22/open/n2356/basic.html#basic.fundamental>). Le fait que le `char` soit en fait un `signed char` ou `unsigned char` est laissé au choix du compilateur.

Manipuler des `unsigned char` permet donc de lever cette ambiguïté, tout en permettant d'utiliser l'un ou l'autre (ou tout autre type) via des conversions (`static_cast`, `reinterpret_cast`). Quels que soient la plateforme ou le compilateur utilisé, nous communiquerons via des `unsigned char` avec l'application.

Je trouve également d'un point de vue personnel qu'il est plus simple de réfléchir en termes d'`unsigned char`, sur un intervalle [0;255] qu'en termes de `signed char` [-128;127], que je dois rechercher quasi systématiquement pour sortir de la confusion « [-127;128] ou [-128;127] ? ».

Au niveau de l'implémentation, commençons par le plus simple, le Send :

TCPSocket::Send

```
1. bool TCPSocket::Send(const unsigned char* data, unsigned short len)
2. {
3.     unsigned short networkLen = htons(len);
4.     return send(mSocket, reinterpret_cast<const char*>(&
networkLen), sizeof(networkLen), 0) == sizeof(networkLen)
5.     && send(mSocket, reinterpret_cast<const char*>(data), len, 0) == len;
6. }
```

Comme convenu, notre protocole commence par envoyer la longueur des données, puis les données à proprement parler.

Les conversions nécessaires pour le passage des paramètres à l'API sockets, et enfin on s'assure que les données envoyées l'ont été avec succès en vérifiant que les tailles mises en file d'envoi sont celles attendues.

La seule *difficulté* réside dans la conversion de ladite longueur dans l'endianness réseau avec la fonction `htons` vue dès la première partie du cours. C'est facilement oubliable.

Voyons maintenant l'implémentation du Receive qui, sans être insurmontable, présente une subtilité :

TCPSocket::Receive

```
1. bool TCPSocket::Receive(std::vector<unsigned char>& buffer)
2. {
3.     unsigned short expectedSize;
4.     int pending = recv(mSocket, reinterpret_cast<char*>(&expectedSize), sizeof(expectedSize), 0);
5.     if ( pending <= 0 || pending != sizeof(unsigned short) )
6.     {
7.         //!< Erreur
8.         return false;
9.     }
10.
11.     expectedSize = ntohs(expectedSize);
12.     buffer.resize(expectedSize);
13.     int receivedSize = 0;
14.     do {
15.         int ret = recv(mSocket, reinterpret_cast<char*>(&buffer[receivedSize]), (expectedSize -
receivedSize) * sizeof(unsigned char), 0);
16.         if ( ret <= 0 )
17.         {
18.             //!< Erreur
19.             buffer.clear();
20.             return false;
21.         }
22.         else
23.         {
24.             receivedSize += ret;
25.         }
26.     } while ( receivedSize < expectedSize );
27.     return true;
28. }
```

Vous pouvez remarquer immédiatement la symétrie logique avec le Send.

On commence par lire notre en-tête, la taille attendue des données qui vont suivre et former notre paquet. On s'assure que tout s'est bien passé, sinon inutile d'aller plus loin et on peut retourner `false` immédiatement pour indiquer à l'utilisateur qu'un problème est survenu.

Ensuite, on lit les données. Et alors qu'on pourrait s'attendre à un simple appel à `recv`, celui-ci se trouve dans une boucle.

`recv` ne fait que transférer des données depuis un tampon système dans notre propre tampon passé en paramètre, en prenant en compte la taille maximale de l'espace disponible que l'on lui indique. Et il s'agit bien de la taille **maximale** à extraire, et non de la taille attendue. `recv` n'a aucune connaissance du fait que nous allons effectivement recevoir X données. Si nous demandons la réception de X données, mais qu'il n'en a déjà que Y disponibles, avec $Y \leq X$, il remplira notre tampon de ces Y données et nous retournera cette valeur. Si nous voulons vraiment attendre les X données, nous le réalisons à l'aide de cette boucle, dans la logique logicielle. Il ne bloquera pas jusqu'à remplir le tampon qu'on lui fournit.

Dans l'absolu, cette même boucle devrait être mise en place pour la réception de l'en-tête. Inutile d'encombrer le code pour l'instant, supposons, sans grande erreur, que l'envoi des deux octets d'en-tête sera toujours faisable sans découpage par la couche réseau pour rendre cette opération possible en un seul appel à `recv`.

Et bien entendu pour chaque appel à `recv`, il convient de vérifier qu'aucune erreur n'est survenue.

 [Télécharger les codes sources du cours](#)

Article précédent
[<< TCP - Envoi et réception](#)

Article suivant
[TCP - Premiers-pas en tant que serveur](#)

Cours programmation réseau en C++

TCP - Premiers pas en tant que serveur

Par [Bousk](#)

Date de publication : 29 novembre 2016

Dernière mise à jour : 9 mai 2017

TOUT PUBLIC

Maintenant que nous savons comment réaliser un client TCP, voyons comment réaliser un serveur.

Passons d'abord en revue les nouvelles fonctions spécifiques à un programme serveur avant de créer notre premier serveur.

Commentez

I - Fonctions spécifiques au serveur.....	3
I-A - Bind - int bind(SOCKET sckt, const struct addr* name, int namelen);.....	3
I-B - Listen - int listen(SOCKET sckt, int backlog) ;.....	3
I-C - Accept - SOCKET accept(SOCKET sckt, struct sockaddr* addr, int* addrlen);.....	4
I-D - Récupérer l'IP d'un client	4
I-D-1 - Windows - const char* inet_ntop(int family, void* src, char* dst, size_t size);.....	4
I-D-2 - Unix - const char* inet_ntop(int family, const void* src, char* dst, socklen_t size);.....	4
II - Notre premier serveur.....	4
III - Proposition de corrigé.....	6

I - Fonctions spécifiques au serveur

I-A - Bind - `int bind(SOCKET sckt, const struct addr* name, int namelen);`

La fonction `bind` est utilisée pour assigner une adresse locale à un socket.

- `sckt` est le socket auquel est assigné l'adresse.
- `name` est la structure à assigner au socket.
- `namelen` est la taille de cette structure, généralement un `sizeof` fera l'affaire.

Retourne `SOCKET_ERROR` sous Windows et -1 sous Unix en cas d'erreur, 0 sinon.

La structure `name` contiendra le port public à assigner ainsi que l'information des adresses distantes qui peuvent se connecter à notre socket. Sa création classique, pour accepter toutes les connexions entrantes, sera ainsi :

```
sockaddr_in addr;
addr.sin_addr.s_addr = INADDR_ANY; // indique que toutes les sources seront acceptées
addr.sin_port = htons(port); // toujours penser à traduire le port en réseau
addr.sin_family = AF_INET; // notre socket est TCP
int res = bind(server, (sockaddr*)&addr, sizeof(addr));
if (res != 0)
    // erreur
```

Si le port demandé est 0, le système assignera lui-même un port valide automatiquement.

Généralement les ports jusqu'à 1024 sont réservés pour le système.

La valeur `INADDR_ANY` pour `s_addr` indique au système d'assigner ce socket à toutes les interfaces disponibles sur la machine, et ainsi d'accepter toutes les sources de connexion, locales et distantes. Si vous hésitez sur la valeur à assigner, vous devriez sûrement utiliser celle-ci.

I-B - Listen - `int listen(SOCKET sckt, int backlog);`

Place le socket dans un état lui permettant d'écouter les connexions entrantes.

`sckt` est le socket auquel les clients vont se connecter.

`backlog` est le nombre de connexions en attente qui peuvent être gérées. La valeur `SOMAXCONN` peut être utilisée pour laisser le système choisir une valeur correcte selon sa configuration.

Retourne `SOCKET_ERROR` sous Windows et -1 sous Unix en cas d'erreur, 0 sinon.

```
res = listen(server, SOMAXCONN);
if (res != 0)
    // erreur
```

Si vous appelez `listen` avant `bind`, le système fera l'équivalent d'un appel à `bind` avec `INADDR_ANY` et le port 0, soit le choix d'un port aléatoire disponible.

Notez que c'est exactement ce qui se passe quand vous appelez `connect` pour un socket client. Mais dans le cadre d'un serveur, on préfère assigner un port en particulier et connu dans la majorité des cas.

I-C - Accept - SOCKET accept(SOCKET sckt, struct sockaddr* addr, int* addrlen);

Accepte une connexion entrante.

- `sckt` est le socket serveur qui attend les connexions.
- `addr` recevra l'adresse du socket qui se connecte.
- `addrlen` est la taille de la structure pointée par `addr`.

Cette fonction est bloquante en attendant qu'une connexion entrante arrive.

Retourne `INVALID_SOCKET` sous Windows et `-1` sous Unix en cas d'erreur, un socket représentant le nouveau client sinon, la structure `addr` contient alors les informations du client connecté.

```
sockaddr_in addr = { 0 };
int len = sizeof(addr);
SOCKET newClient = accept(server, (sockaddr*)&addr, &len);
if (newClient == INVALID_SOCKET)
    // erreur
```

Le socket ainsi récupéré sert d'identifiant pour communiquer, recevoir et envoyer des données, avec le client connecté. On l'appellera le socket client.

I-D - Récupérer l'IP d'un client

I-D-1 - Windows - `const char* inet_ntop(int family, void* src, char* dst, size_t size);`

I-D-2 - Unix - `const char* inet_ntop(int family, const void* src, char* dst, socklen_t size);`

Permet de récupérer l'adresse IP d'un socket, IPv4 ou IPv6, sous forme lisible.

- `family` est la famille du socket.
- `src` le pointeur vers l'adresse du socket.
- `dst` un pointeur vers un tampon où stocker l'adresse sous forme lisible.
- `size` la taille maximale du tampon.

```
char buff[INET6_ADDRSTRLEN] = {0};
return inet_ntop(addr.sin_family, (void*)&(addr.sin_addr), buff, INET6_ADDRSTRLEN);
```

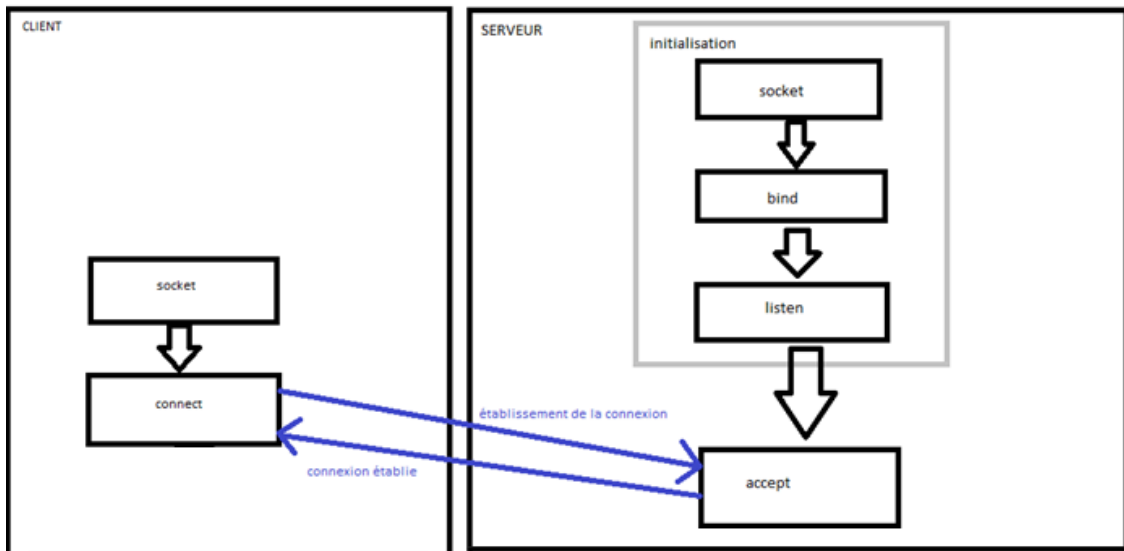
L'utilisation de `INET6_ADDRSTRLEN` permet d'utiliser cette fonction indépendamment du fait qu'il s'agisse d'une adresse IPv4 ou IPv6 puisqu'une adresse IPv6 peut être écrite en utilisant jusqu'à 45 caractères (`INET6_ADDRSTRLEN` vaut au moins 46) alors qu'une adresse IPv4 s'étend au maximum sur 15 caractères (et son équivalent `INET_ADDRSTRLEN` vaut au moins 16).

Cette fonction est bien entendu également utilisable dans du code client, mais l'intérêt paraît moindre puisqu'un client se connecte au serveur via une adresse et un port connus.

II - Notre premier serveur

Pour notre premier serveur, on se contentera uniquement d'accepter la connexion de clients.

Notre programme devrait avoir l'architecture suivante :

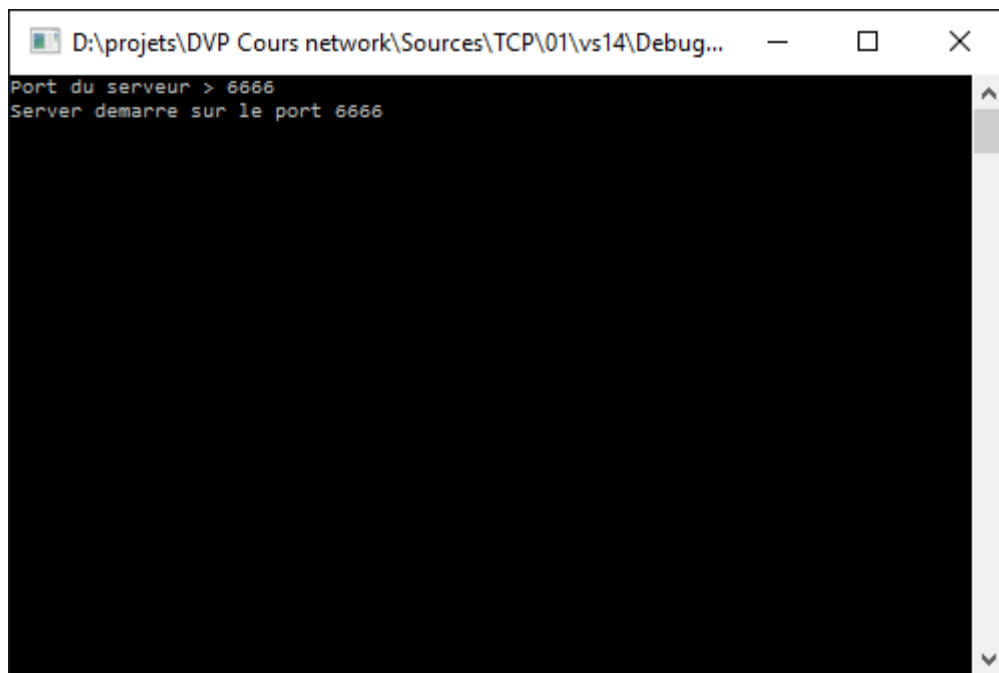


À chaque connexion, on affichera l'adresse IP et le port du client.



*Training Time ! Mettez en place un serveur simple suivant le schéma précédent auquel vous vous connectez avec le client réalisé dans la partie **TCP - Premiers pas avec TCP** .*

Vous devriez retrouver le résultat présenté dans la partie **premiers pas**, à savoir :



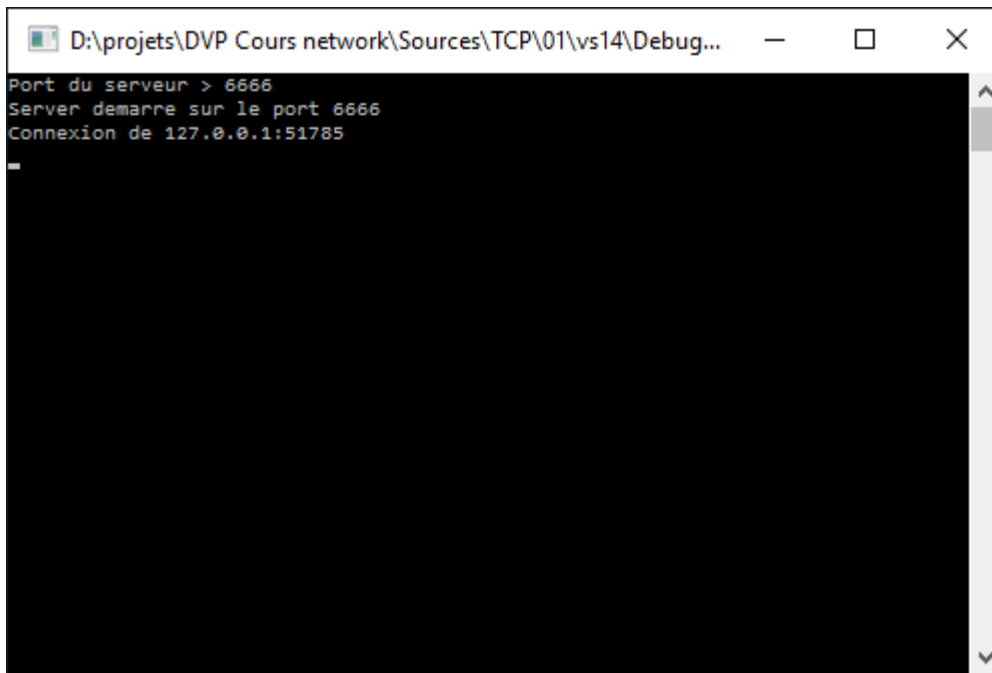
```

D:\projets\DVP Cours network\Sources\TCP\01\vs14\Debug...
Port du serveur > 6666
Server démarre sur le port 6666
  
```

The screenshot shows a Windows command prompt window with the title 'D:\projets\DVP Cours network\Sources\TCP\01\vs14\Debug...'. The prompt shows the user has entered 'Port du serveur > 6666' and the system has responded with 'Server démarre sur le port 6666'.

Quand vous démarrez le serveur.

Puis



```

D:\projets\DVP Cours network\Sources\TCP\01\vs14\Debug...
Port du serveur > 6666
Server démarre sur le port 6666
Connexion de 127.0.0.1:51785
  
```

Quand un client se connecte au serveur.

III - Proposition de corrigé

La première version devrait être assez simple à mettre en place : il suffit de dérouler le schéma en écrivant chaque étape du code correspondant.

Ainsi, le programme final devrait ressembler, sous Windows, à ceci :

Premier serveur

```

1. int main()
2. {
3.     WSADATA wsaData;
4.     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
5.     {
6.         std::cout << "Erreur initialisation WinSock : " << WSAGetLastError();
7.         return -1;
8.     }
9.
10.    SOCKET server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
11.    if (server == INVALID_SOCKET)
12.    {
13.        std::cout << "Erreur initialisation socket : " << WSAGetLastError();
14.        return -2;
15.    }
16.
17.    const unsigned short port = 9999;
18.    sockaddr_in addr;
19.    addr.sin_addr.s_addr = INADDR_ANY;
20.    addr.sin_port = htons(port);
21.    addr.sin_family = AF_INET;
22.
23.    int res = bind(server, (sockaddr*)&addr, sizeof(addr));
24.    if (res != 0)
25.    {
26.        std::cout << "Erreur bind : " << WSAGetLastError();
27.        return -3;
28.    }
29.
30.    res = listen(server, SOMAXCONN);
31.    if (res != 0)
32.    {
  
```

Premier serveur

```

33.     std::cout << "Erreur listen : " << WSAGetLastError();
34.     return -4;
35. }
36.
37.     std::cout << "Serveur démarre sur le port " << port << std::endl;
38.
39.     sockaddr_in from = { 0 };
40.     socklen_t addrlen = sizeof(addr);
41.     SOCKET newClient = accept(server, (SOCKADDR*)&from, &addrlen);
42.     if (newClient != INVALID_SOCKET)
43.     {
44.         char buff[INET6_ADDRSTRLEN] = { 0 };
45.         std::string clientAddress = inet_ntop(addr.sin_family, (void*)&(addr.sin_addr), buff,
            INET6_ADDRSTRLEN);
46.         std::cout << "Connexion de " << clientAddress.c_str() << ":" << addr.sin_port << std::endl;
47.     }
48.     closesocket(server);
49.     WSACleanup();
50.     return 0;
51. }

```

Vous retrouvez l'initialisation de la bibliothèque socket propre à Windows, puis on crée un socket de manière identique au code client.

Ensuite ça diverge puisqu'on associe un port spécifique au serveur (on peut bind le port 0 et laisser le système choisir, mais connaître le port spécifique d'un serveur est généralement plus intéressant) avant de le passer en mode écoute et prêt à accepter des connexions entrantes.

Une fois une connexion acceptée, on se contente d'afficher l'IP et le port du client avant de fermer le programme. Oui, ce code n'accepte qu'un unique client, il faut bien commencer quelque part. Il s'agit de la retranscription exacte du schéma plus haut.

Une évolution directe serait de pouvoir accepter plusieurs clients et afficher leurs informations à chaque fois, en remplaçant les lignes 40 à 49 par :

```

1. for (;;)
2. {
3.     sockaddr_in from = { 0 };
4.     socklen_t addrlen = sizeof(addr);
5.     SOCKET newClient = accept(server, (SOCKADDR*)&from, &addrlen);
6.     if (newClient != INVALID_SOCKET)
7.     {
8.         char buff[INET6_ADDRSTRLEN] = { 0 };
9.         std::string clientAddress = inet_ntop(addr.sin_family, (void*)&(addr.sin_addr), buff,
            INET6_ADDRSTRLEN);
10.        std::cout << "Connexion de " << clientAddress << ":" << addr.sin_port << std::endl;
11.    }
12.    else
13.        break;
14. }

```

Enfin, on constate que du code peut être mis en commun avec les parties précédentes, notamment déjà l'initialisation.

Puisque le but final reste d'avoir à disposition une bibliothèque réseau utilisable, commençons déjà par factoriser ces parties.

Reprenons les fichiers Sockets.hpp et Sockets.cpp à notre disposition. Ajoutons une fonction bien utile pour la récupération de l'adresse IP depuis un socket dont le prototype sera `std::string GetAddress(const sockaddr_in& addr)` :

```

std::string GetAddress(const sockaddr_in& addr)
{

```

```
char buff[INET6_ADDRSTRLEN] = { 0 };  
return inet_ntop(addr.sin_family, (void*)&(addr.sin_addr), buff, INET6_ADDRSTRLEN);  
}
```

Retourne une chaîne vide en cas d'erreur.

Le programme final devrait ressembler à :

```
1. #include "Sockets.hpp"  
2.  
3. #include <iostream>  
4. #include <string>  
5.  
6. int main()  
7. {  
8.     if (!Sockets::Start())  
9.     {  
10.         std::cout << "Erreur initialisation WinSock : " << Sockets::GetError();  
11.         return -1;  
12.     }  
13.  
14.     SOCKET server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  
15.     if (server == INVALID_SOCKET)  
16.     {  
17.         std::cout << "Erreur initialisation socket : " << Sockets::GetError();  
18.         return -2;  
19.     }  
20.  
21.     const unsigned short port = 9999;  
22.     sockaddr_in addr;  
23.     addr.sin_addr.s_addr = INADDR_ANY;  
24.     addr.sin_port = htons(port);  
25.     addr.sin_family = AF_INET;  
26.  
27.     int res = bind(server, (sockaddr*)&addr, sizeof(addr));  
28.     if (res != 0)  
29.     {  
30.         std::cout << "Erreur bind : " << Sockets::GetError();  
31.         return -3;  
32.     }  
33.  
34.     res = listen(server, SOMAXCONN);  
35.     if (res != 0)  
36.     {  
37.         std::cout << "Erreur listen : " << Sockets::GetError();  
38.         return -4;  
39.     }  
40.  
41.     std::cout << "Serveur démarre sur le port " << port << std::endl;  
42.  
43.     for (;;)   
44.     {  
45.         sockaddr_in from = { 0 };  
46.         socklen_t addrlen = sizeof(addr);  
47.         SOCKET newClient = accept(server, (SOCKADDR*)&from, &addrlen);  
48.         if (newClient != INVALID_SOCKET)  
49.         {  
50.             std::string clientAddress = Sockets::GetAddress(from);  
51.             std::cout << "Connexion de " << clientAddress << ":" << addr.sin_port << std::endl;  
52.         }  
53.         else  
54.             break;  
55.     }  
56.     Sockets::CloseSocket(server);  
57.     Sockets::Release();  
58.     return 0;  
59. }
```

Article précédent**<< TCP - Mise en place du protocole****Article suivant****Multi-threading et mutex**

Cours programmation réseau en C++

TCP - Envoi et réception depuis le serveur

Par [Bousk](#)

Date de publication : 31 mars 2017

Dernière mise à jour : 9 mai 2017

DÉBUTANT

Nous pouvons désormais nous connecter à notre serveur, il est temps de lui envoyer des données et en recevoir de sa part.

Commentez

I - Envoyer et recevoir des données depuis un socket serveur.....	3
II - send et recv.....	3
III - Architecture du serveur et ses clients.....	3
III-A - À chaque client son thread.....	3
III-B - Un unique thread : vérifier l'état des descripteurs.....	5
III-B-1 - select - int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);.....	5
III-B-1-a - Connaître l'erreur d'un socket donné.....	7
III-B-1-b - Windows - int getsockopt(SOCKET sockt, int level, int optname, char* optval, int* optlen);.....	7
III-B-1-c - Unix - int getsockopt(int sockfd, int level, int optname, void* optval, socklen_t* optlen);.....	8
III-C - Alternative à select : poll.....	8
III-C-1 - Windows - int WSAPoll(WSAPOLLFD fdarray[], unsigned long nfds, int timeout);.....	8
III-C-2 - Unix - int poll(struct pollfd* fds, nfds_t nfds, int timeout);.....	8
III-C-3 - Une utilisation portable de poll et WSAPoll.....	9
III-C-4 - poll ou select ?.....	11
III-D - Un unique thread : socket non bloquant.....	12
III-D-1 - Windows - int ioctlsocket(SOCKET socket, long command, unsigned long* parameter);.....	12
III-D-2 - Unix - int fcntl(int fd, int cmd, ...);.....	12
III-D-3 - Changements liés au mode non bloquant.....	12
III-D-3-a - Serveur à un seul thread.....	13

I - Envoyer et recevoir des données depuis un socket serveur

Pour envoyer des données à et depuis un client connecté, on utilisera la même fonction que dans le code client : `send`.

De même, la réception de données se fait via `recv`.

II - send et recv

Pour rappel, les prototypes de ces fonctions sont `int send(int socket, const void* datas, size_t len, int flags);` et `int recv(int socket, void* buffer, size_t len, int flags);`.

Je vous renvoie vers **la première partie du cours** qui traite de ces fonctions plus en détail.

Nous avons vu que le paramètre `socket` est le socket auquel nous voulons envoyer les données ou duquel nous voulons les recevoir. Ici il s'agira donc du socket du client que nous avons créé via l'appel à `accept` vu dans **la partie précédente**.

III - Architecture du serveur et ses clients

Afin de gérer ses clients, notre serveur va maintenant devoir maintenir une liste de clients connectés en enregistrant les retours de `accept` qui représente chaque client effectivement connecté à notre serveur, et en supprimant ceux dont le socket retourne une erreur indiquant qu'ils ont été déconnectés.

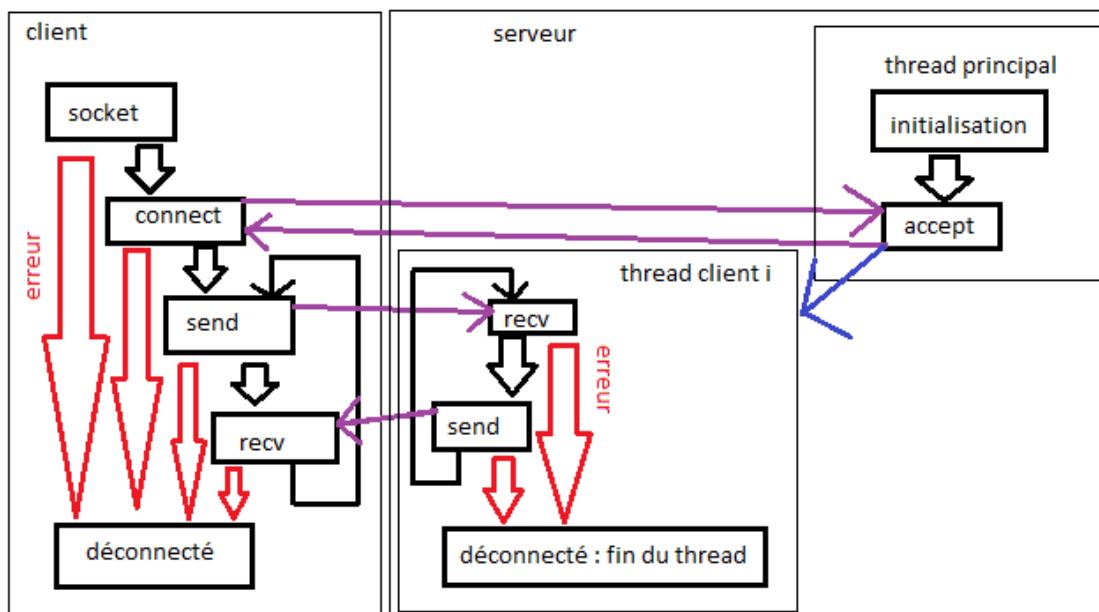
« Gérer ses clients » signifie recevoir et traiter les données qu'ils envoient, les requêtes, puis envoyer d'éventuelles réponses.

Souvenez-vous aussi que `send` et surtout `recv` sont bloquants. La première solution à laquelle on pense est généralement d'avoir un thread par client. Commençons donc par celle-ci.

III-A - À chaque client son thread

La première option sera donc de créer un thread par client, tandis que le thread principal servira à l'initialisation et à accepter les connexions entrantes.

Voilà grossièrement à quoi devrait ressembler notre programme :



Pour ce qui est du client, nous réutiliserons le client de la **partie 2 Envoi et réception**.

Pour le code serveur, repartons sur le code précédent et créons dans un premier temps un thread par client dans la boucle d'accept, en lieu et place où nous nous contentions d'afficher les informations du client connecté. Nous allons maintenant renvoyer au client ce qu'il nous a envoyé :

Boucle d'acceptation des nouveaux clients et lancement du thread pour chacun

```

1. for (;;)
2. {
3.     sockaddr_in from = { 0 };
4.     socklen_t addrlen = sizeof(from);
5.     SOCKET newClient = accept(server, (SOCKADDR*)&from, &addrlen);
6.     if (newClient != INVALID_SOCKET)
7.     {
8.         std::thread([newClient, from]() {
9.             const std::string clientAddress = Sockets::GetAddress(from);
10.            const unsigned short clientPort = ntohs(from.sin_port);
11.            std::cout << "Connexion de " << clientAddress.c_str() << ":" << clientPort << std::endl;
12.            bool connected = true;
13.            for(;;)
14.            {
15.                char buffer[200] = { 0 };
16.                int ret = recv(newClient, buffer, 199, 0);
17.                if (ret == 0 || ret == SOCKET_ERROR)
18.                    break;
19.                std::cout << "[" << clientAddress << ":" << clientPort << "]" << buffer << std::endl;
20.                ret = send(newClient, buffer, ret, 0);
21.                if (ret == 0 || ret == SOCKET_ERROR)
22.                    break;
23.            }
24.            std::cout << "Deconnexion de [" << clientAddress << ":" << clientPort << "]" << std::endl;
25.        }).detach();
26.    }
27.    else
28.        break;
29. }

```

Lignes 8 à 25, se trouvent les changements et le code qui permet de lancer un thread reproduisant le comportement du schéma précédent : chaque client exécutera son rcv puis son send dans son propre thread.

En termes de traitement de la requête, nous faisons au plus simple : il n'y a aucun traitement et nous nous contentons de retourner à l'expéditeur ce qu'il nous a envoyé.

**Télécharger le code source de l'exemple.**

III-B - Un unique thread : vérifier l'état des descripteurs

Une autre approche du serveur est d'avoir l'ensemble des traitements sur un unique thread.

D'abord, créons une structure très simple pour agréger les sockets de chaque client avec leur adresse :

```
1. struct Client {  
2.     SOCKET sckt;  
3.     sockaddr_in addr;  
4. };
```

Qui nous permettra de facilement avoir l'information d'IP et port du client en question.

Pour garder en mémoire nos clients connectés, ayons recours à un `std::vector` :

```
std::vector<Client> clients ;
```

III-B-1 - select - int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

Permet de récupérer le statut d'écriture, lecture ou erreur d'un ou plusieurs sockets, sous Windows, ou descripteurs de fichiers, sous Unix.

- `nfd` est l'identifiant du descripteur le plus élevé, plus un
- ce paramètre est ignoré sous Windows, mais présent pour compatibilité.
- `readfds` est un pointeur vers un ensemble de sockets pour lesquelles tester le statut de lecture.
- `writefds` est un pointeur vers un ensemble de sockets pour lesquelles tester le statut d'écriture.
- `exceptfds` est un pointeur vers un ensemble de sockets pour lesquelles tester le statut d'erreur.
- `timeout` est un pointeur vers une structure pour le temps maximum que `select` doit attendre et bloquer avant de retourner, une valeur de `nullptr` permet de bloquer jusqu'à ce qu'un des sockets soit prêt à lire ou écrire.

Pour les `fd_set`, on utilisera les macros de `FD_ZERO` pour l'initialiser et `FD_SET` pour mettre les valeurs des sockets, de cette forme :

```
fd_set set;  
FD_ZERO(&set);  
FD_SET(server, &set);
```

Où `server` est notre socket serveur.

`select` retourne le nombre de sockets qui sont prêts à lire, écrire ou ayant une erreur, peut retourner 0 si aucun socket n'est prêt. Retourne -1 en cas d'erreur.

Pour vérifier qu'un socket, ou descripteur de fichier, particulier a été défini dans la structure, on utilisera la macro `FD_ISSET`.

Pour vérifier qu'une connexion entrante est en attente, que l'appel à `accept` ne sera pas bloquant, on doit vérifier que notre socket serveur est prêt en écriture :

```
1. fd_set set;  
2. timeval timeout = { 0 };  
3. FD_ZERO(&set);  
4. FD_SET(server, &set);
```

```

5. int selectReady = select(server + 1, &set, nullptr, nullptr, &timeout);
6. if (selectReady == -1)
7. {
8.     std::cout << "Erreur select pour accept : " << Sockets::GetError() << std::endl;
9.     break;
10. }
11. else if (selectReady > 0)
12. {
13.     // notre socket server est prêt à être lu
14.     sockaddr_in from = { 0 };
15.     socklen_t addrlen = sizeof(from);
16.     SOCKET newClientSocket = accept(server, (SOCKADDR*)&from, &addrlen);
17.     if (newClientSocket != INVALID_SOCKET)
18.     {
19.         Client newClient;
20.         newClient.sckt = newClientSocket;
21.         newClient.addr = from;
22.         const std::string clientAddress = Sockets::GetAddress(from);
23.         const unsigned short clientPort = ntohs(from.sin_port);
24.         std::cout << "Connexion de " << clientAddress.c_str() << ":" << clientPort << std::endl;
25.     }
26. }

```

Pour vérifier qu'un de nos clients est prêt à recevoir des données, l'utilisation sera identique, mais notre structure `fd_set` sera utilisé pour vérifier tous les clients via un seul appel à `select` :

```

1. fd_set setReads;
2. fd_set setWrite;
3. fd_set setErrors;
4. int highestFd = 0;
5. timeval timeout = { 0 };
6. for (auto& client : clients)
7. {
8.     FD_SET(client.sckt, &setReads);
9.     FD_SET(client.sckt, &setWrite);
10.    FD_SET(client.sckt, &setErrors);
11.    if (client.sckt > highestFd)
12.        highestFd = client.sckt;
13. }
14. int selectResult = select(highestFd + 1, &setReads, &setWrite, &setErrors, &timeout);
15. if (selectResult == -1)
16.     // erreur
17. else if (selectResult > 0)
18.     // au moins 1 client a une action à exécuter

```

Si `selectResult` est strictement positif, au moins un de nos clients est prêt à recevoir ou envoyer des données, ou a une erreur :

```

1. auto itClient = clients.begin();
2. while (itClient != clients.end())
3. {
4.     const std::string clientAddress = Sockets::GetAddress(itClient->addr);
5.     const unsigned short clientPort = ntohs(itClient->addr.sin_port);
6.
7.     bool hasError = false;
8.     if (FD_ISSET(itClient->sckt, &setErrors))
9.     {
10.        std::cout << "Erreur" << std::endl;
11.        hasError = true;
12.    }
13.    else if (FD_ISSET(itClient->sckt, &setReads))
14.    {
15.        char buffer[200] = { 0 };
16.        int ret = recv(itClient->sckt, buffer, 199, 0);
17.        if (ret == 0 || ret == SOCKET_ERROR)
18.        {
19.            std::cout << "Erreur reception" << std::endl;
20.            hasError = true;
21.        }

```

```
22. else
23. {
24.     std::cout << "[" << clientAddress << ":" << clientPort << "]" << buffer << std::endl;
25.     if (FD_ISSET(itClient->sckt, &setWrite))
26.     {
27.         ret = send(itClient->sckt, buffer, ret, 0);
28.         if (ret == 0 || ret == SOCKET_ERROR)
29.         {
30.             std::cout << "Erreur envo" << std::endl;
31.             hasError = true;
32.         }
33.     }
34. }
35. }
36. if (hasError)
37. {
38.     //!< Déconnecté
39.     std::cout << "Deconnexion de [" << clientAddress << ":" << clientPort << "]" << std::endl;
40.     itClient = clients.erase(itClient);
41. }
42. else
43. {
44.     ++itClient;
45. }
46. }
```

Ce code nécessitera quelques `static_cast` selon la plateforme et les options de compilation.

III-B-1-a - Connaître l'erreur d'un socket donné

Puisque `select` permet de connaître l'état de plusieurs sockets à la fois, nous ne pouvons pas utiliser notre `Sockets::GetError()` pour déterminer l'erreur d'un socket en particulier.

Pour connaître l'erreur d'un socket en particulier, il faudra utiliser la fonction `getsockopt`.

III-B-1-b - Windows - `int getsockopt(SOCKET sckt, int level, int optname, char* optval, int* optlen);`

Permet de récupérer certaines informations d'un socket, dont l'erreur qui l'affecte.

- `sckt` est le socket en question.
- `level` est le niveau relatif à l'option que nous voulons récupérer. Pour les erreurs il s'agira de `SOL_SOCKET`.
- `optname` est le nom de l'option à récupérer. Pour les erreurs il s'agira de `SO_ERROR`.
- `optval` est un tampon pour récupérer la valeur de l'option.
- `optlen` est un pointeur vers la taille du tampon.

Son utilisation sera donc :

```
getsockopt
int err;
int errsize = sizeof(err);
getsockopt(sckt, SOL_SOCKET, SO_ERROR, reinterpret_cast<char*>(&err), &errsize);
```

Retourne 0 si aucune erreur est survenue, `SOCKET_ERROR` sinon.

III-B-1-c - Unix - int getsockopt(int sockfd, int level, int optname, void* optval, socklen_t* optlen);

La principale différence vient du type des paramètres, un `int` remplace le `SOCKET` pour le descripteur de socket, le tampon de la valeur de l'option sera un `void*` et la taille du tampon sera représentée par un `socklen_t`.

Puisque notre code possède déjà de quoi faire abstraction du `SOCKET` et `socklen_t`, et que le langage permet de convertir automatiquement un `char*` vers un `void*`, l'appel pourra être uniformisé entre Windows et Unix sous cette forme :

getsockopt cross-platform

```
socklen_t err;
int errsize = sizeof(err);
if (getsockopt(sckt, SOL_SOCKET, SO_ERROR, reinterpret_cast<char*>(&err), &errsize) != 0)
{
    // erreur lors de la recuperation d'erreur...
    std::cout << "Erreur lors de la determination de l'erreur : "
    << Sockets::GetError() << std::endl;
}
```



Télécharger le code source de l'exemple

III-C - Alternative à select : poll

Les systèmes plus récents (Windows Vista et supérieurs) proposent une alternative à `select` avec `poll`. Leur fonctionnement est identique : vérifier l'état d'un ensemble de descripteurs. Le principal avantage qui aura un intérêt est que `poll` peut gérer plus que 1024 descripteurs à la fois.

III-C-1 - Windows - int WSAPoll(WSAPOLLFD fdarray[], unsigned long nfds, int timeout);

Permet de récupérer l'état d'un ensemble de descripteurs de sockets.

- `fdarray` est un tableau de structures `WSAPOLLFD` (voir détails ci-dessous).
- `nfds` est le nombre de structures `WSAPOLLFD` dans `fdarray`.
- `timeout` est la durée maximale d'attente avant retour.
- `timeout < 0` indique une attente infinie : un appel bloquant.
- `timeout == 0` indique un appel non bloquant.
- `timeout > 0` pour définir un temps d'attente en millisecondes.

La structure `WSAPOLLFD` possède trois champs :

- `fd` de type `SOCKET` pour accueillir le descripteur de socket ;
- `events` de type `short` servant de champ de bits des états à vérifier ;
- `revents` de type `short` qui sera modifié par l'appel avec les flags des états trouvés pour le socket en question.

Voir la documentation plus complète de `WSAPoll` sur la MSDN.

III-C-2 - Unix - int poll(struct pollfd* fds, nfds_t nfds, int timeout);

Version Unix de `WSAPoll`.

- `fds` est un tableau de structure `pollfd` (voir détails ci-dessous).
- `nfds` est la taille du tableau `fds`.

- `timeout` est la durée maximale d'attente avant retour.
- `timeout < 0` indique une attente infinie : un appel bloquant.
- `timeout == 0` indique un appel non bloquant.
- `timeout > 0` pour définir un temps d'attente en millisecondes.

La structure `pollfd` possède également trois champs :

- `fd` de type `int` pour accueillir le descripteur de fichier ;
- `events` de type `short` servant de champ de bits des états à vérifier ;
- `revents` de type `short` qui sera modifié par l'appel avec les flags des états trouvés pour le socket en question.

III-C-3 - Une utilisation portable de poll et WSApoll

Malgré les différences de prototypes et types, les fonctions sont suffisamment similaires pour que la portabilité soit simple. Les structures `WSAPOLLFD` et `pollfd` sont identiques, et Windows définit d'ailleurs lui-même une `struct pollfd`. Inutile de passer par la déclaration `WSAPOLLFD` donc.

Le second paramètre `nfds` est déclaré comme `unsigned long` sur Windows, et est un `typedef` vers `unsigned int` sur Unix. Ayons recours à la technique habituelle, et définissons `nfds_t` pour Windows afin d'utiliser ce type dans notre code indépendamment de la plateforme.

Sockets.hpp

```
#ifndef _WIN32
...
typedef unsigned long nfds_t;
#define poll WSApoll
...
#else
...
#include <poll.h>
...
#endif
```

Les noms des flags de chaque état sont quelque peu différents d'une plateforme à l'autre, mais Windows s'en sort bien en définissant les valeurs que l'on s'attend à avoir en Posix. Les valeurs qui nous intéressent sont :

- `POLLIN` pour vérifier que le socket est prêt en lecture ;
- `POLLOUT` pour vérifier que le socket est prêt en écriture.

Les valeurs intéressantes à vérifier dans `revents` au retour de `poll` sont :

- `POLLERR` pour vérifier qu'une erreur est survenue ;
- `POLLNVAL` si le socket n'était pas initialisé ;
- `POLLHUP` si la connexion a été interrompue ;
- `POLLIN` si l'écriture est possible.
- Notez que si vous envoyez plus de données que la place disponible, le `send` sera toujours bloquant ;
- `POLLOUT` si des données sont disponibles en lecture.

Nous pouvons modifier le code précédent utilisant `select` pour utiliser `poll` :

poll remplace select

```
1. std::map<SOCKET, Client> clients;
2. std::vector<pollfd> clientsFds;
3. for (;;)
4. {
5. {
6. pollfd pollServerFd;
7. pollServerFd.fd = server;
8. pollServerFd.events = POLLIN;
```

poll replace select

```

9.  int pollReady = poll(&pollServerFd, 1, 0);
10. if (pollReady == -1)
11. {
12.     std::cout << "Erreur poll pour accept : " << Sockets::GetError() << std::endl;
13.     break;
14. }
15. if (pollReady > 0)
16. {
17.     sockaddr_in from = { 0 };
18.     socklen_t addrlen = sizeof(from);
19.     SOCKET newClientSocket = accept(server, (SOCKADDR*)&from, &addrlen);
20.     if (newClientSocket != INVALID_SOCKET)
21.     {
22.         Client newClient;
23.         newClient.sckt = newClientSocket;
24.         newClient.addr = from;
25.         const std::string clientAddress = Sockets::GetAddress(from);
26.         const unsigned short clientPort = ntohs(from.sin_port);
27.         std::cout << "Connexion de " << clientAddress.c_str() << ":" << clientPort << std::endl;
28.         clients[newClientSocket] = newClient;
29.         pollfd newClientPollFd;
30.         newClientPollFd.fd = newClientSocket;
31.         newClientPollFd.events = POLLIN | POLLOUT;
32.         clientsFds.push_back(newClientPollFd);
33.     }
34. }
35. }
36. if (!clients.empty())
37. {
38.     int pollResult = poll(clientsFds.data(), static_cast<nfds_t>(clientsFds.size()), 0);
39.     if (pollResult == -1)
40.     {
41.         std::cout << "Erreur poll pour clients : " << Sockets::GetError() << std::endl;
42.         break;
43.     }
44.     else if (pollResult > 0)
45.     {
46.         auto itPollResult = clientsFds.cbegin();
47.         while (itPollResult != clientsFds.cend())
48.         {
49.             const auto clientIt = clients.find(itPollResult->fd);
50.             if (clientIt == clients.cend())
51.             {
52.                 itPollResult = clientsFds.erase(itPollResult);
53.                 continue;
54.             }
55.             const auto& client = clientIt->second;
56.             const std::string clientAddress = Sockets::GetAddress(client.addr);
57.             const unsigned short clientPort = ntohs(client.addr.sin_port);
58.             bool disconnect = false;
59.             if (itPollResult->revents & POLLERR)
60.             {
61.                 socklen_t err;
62.                 int errsize = sizeof(err);
63.                 if (getsockopt(client.sckt, SOL_SOCKET,
64.                     SO_ERROR, reinterpret_cast<char*>(&err), &errsize) != 0)
65.                 {
66.                     std::cout << "Impossible de determiner l'erreur : " << Sockets::GetError() << std::endl;
67.                     if (err != 0)
68.                         std::cout << "Erreur : " << err << std::endl;
69.                     disconnect = true;
70.                 }
71.             }
72.             else if (itPollResult->revents & (POLLHUP | POLLNVAL))
73.             {
74.                 disconnect = true;
75.             }
76.             else if (itPollResult->revents & POLLIN)
77.             {
78.                 char buffer[200] = { 0 };

```


poll remplace select

```

78.     int ret = recv(client.sckt, buffer, 199, 0);
79.     if (ret == 0)
80.     {
81.         std::cout << "Connexion terminée" << std::endl;
82.         disconnect = true;
83.     }
84.     else if (ret == SOCKET_ERROR)
85.     {
86.         std::cout << "Erreur reception : " << Sockets::GetError() << std::endl;
87.         disconnect = true;
88.     }
89.     else
90.     {
91.         std::cout << "[" << clientAddress << ":" << clientPort << "]" << buffer << std::endl;
92.         if (itPollResult->revents & POLLOUT)
93.         {
94.             ret = send(client.sckt, buffer, ret, 0);
95.             if (ret == 0 || ret == SOCKET_ERROR)
96.             {
97.                 std::cout << "Erreur envoi : " << Sockets::GetError() << std::endl;
98.                 disconnect = true;
99.             }
100.        }
101.    }
102. }
103. if (disconnect)
104. {
105.     std::cout << "Deconnexion de " << "[" << clientAddress << ":" <<
    clientPort << "]" << std::endl;
106.     itPollResult = clientsFds.erase(itPollResult);
107.     clients.erase(clientIt);
108. }
109. else
110. {
111.     ++itPollResult;
112. }
113. }
114. }
115. }
116. }
  
```

Notez également le changement de *clients* pour un `std::map<SOCKET, Client>`. Ce changement peut également être fait dans l'exemple utilisant `select`.

III-C-4 - poll ou select ?

Un avantage de `poll` est la différenciation entre les flags d'entrée, à vérifier, le champ `events` de la structure, et les flags de sortie, retournés, le champ `revents` de la structure. Ça permet de conserver une collection de sockets et flags à appeler sans nécessiter la moindre réinitialisation après appel à `poll`.

Un autre point fort est que `poll` peut être utilisé sur plus de 1024 descripteurs à la fois, là où `select` est limité à 1024.

En dehors de ça, `select` est surtout le premier implémenté historiquement et l'utilisation de l'un ou l'autre est le plus souvent interchangeable. Pour un développement récent, si vous ne comptez pas avoir un code portable sur d'anciens systèmes, autant utiliser `poll` à mon avis. Sauf si celui-ci n'est pas disponible sur la plateforme ciblée.

D'autres systèmes existent aujourd'hui tel `epoll` ou `kqueue`. Ils feront sans doute l'objet d'articles plus avancés par la suite.



Télécharger le code source de l'exemple

III-D - Un unique thread : socket non bloquant

Une autre façon de faire, qui peut aller de pair avec l'utilisation de select, est de déclarer explicitement le socket comme non bloquant.

Cette option a ma préférence dans le cadre d'un serveur, parce que plus simple dans l'écriture et les traitements, selon moi.

III-D-1 - Windows - `int ioctlsocket(SOCKET socket, long command, unsigned long* parameter);`

Permet de changer le mode d'entrée/sortie d'un socket.

- `socket` est le socket sur lequel appliquer la modification.
- `command` est l'identifiant de la commande à appliquer au socket.
- `parameter` est un pointeur vers un paramètre à appliquer à la commande.

Retourne -1 en cas d'échec, 0 sinon.

Dans le cas qui nous intéresse, rendre le socket non bloquant, l'appel sera :

Rendre un socket non bloquant sous Windows

```
1. u_long mode = 1;  
2. ioctlsocket(s, FIONBIO, &mode);
```

Pour avoir la liste des commandes possibles et leur paramètre, référez-vous à [la documentation sur MSDN](#).

III-D-2 - Unix - `int fcntl(int fd, int cmd, ...);`

Permet de changer une propriété d'un descripteur de fichier.

- `fd` est le descripteur de fichier auquel appliquer la modification, dans notre cas le socket.
- `cmd` la commande à appliquer.
- un éventuel dernier paramètre selon la commande souhaitée.

La valeur de retour dépendra de la commande à exécuter.

```
fcntl(socket, F_SETFL, O_NONBLOCK);
```

Dans ce cas retournera -1 en cas d'erreur, autre chose sinon.

Pour avoir la liste des commandes possibles et leur paramètre, ainsi que les valeurs de retour pour chaque commande, référez-vous à [la documentation](#).

III-D-3 - Changements liés au mode non bloquant

Que se passe-t-il désormais pour les fonctions auparavant bloquantes ?

`accept` retournera `INVALID_SOCKET` si aucune nouvelle connexion n'est arrivée au lieu d'attendre la prochaine connexion, le nouveau socket sinon.

`recv` retournera une erreur (-1 ou `SOCKET_ERROR`). Il faudra alors récupérer le code erreur de la bibliothèque socket afin de vérifier s'il s'agit d'une erreur légitime à traiter en tant que telle ou la valeur de `WSAEWOULDBLOCK` (pour Windows) ou `EWOULDBLOCK` (pour Unix) indiquant que `recv` a retourné sans lire de données au lieu de bloquer.

send aura le même comportement que recv si la mise en file d'envois aurait dû être bloquante : retour d'une valeur d'erreur, puis il faudra vérifier si l'erreur de la bibliothèque socket est WSAEWOULDBLOCK/EWOULDBLOCK ou non.

III-D-3-a - Serveur à un seul thread

Puisque Windows et Unix sont ici encore différents, l'un utilisant WSAEWOULDBLOCK l'autre EWOULDBLOCK, commençons par uniformiser ceci dans notre bibliothèque. Ajoutons une énumération d'erreurs à notre code, par exemple dans un nouveau fichier :

Errors.hpp

```
1. #ifndef BOUSK_DVP_COURS_ERRORS_HPP
2. #define BOUSK_DVP_COURS_ERRORS_HPP
3.
4. #pragma once
5.
6. #ifdef _WIN32
7. #include <WinSock2.h>
8. #else
9. #include <cerrno>
10. #define SOCKET int
11. #define INVALID_SOCKET ((int)-1)
12. #define SOCKET_ERROR (int(-1))
13. #endif
14.
15. namespace Sockets
16. {
17.     int GetError();
18.     enum class Errors {
19. #ifdef _WIN32
20.         WOULDBLOCK = WSAEWOULDBLOCK
21. #else
22.         WOULDBLOCK = EWOULDBLOCK
23. #endif
24.     };
25. }
26.
27. #endif // BOUSK_DVP_COURS_ERRORS_HPP
```

J'ai également déplacé `int GetError();` de `Sockets.cpp` vers `Errors.cpp` afin de centraliser tout ce qui est lié aux erreurs dans ces nouveaux fichiers.

Nous avons maintenant une façon élégante et portable de vérifier l'aspect « erreur, opération bloquante qui n'a pas bloqué » via cette nouvelle valeur **Sockets::Errors::WOULDBLOCK**.

Il est maintenant temps de modifier notre programme principal.

D'abord, il faut définir notre socket serveur comme non bloquant :

```
if (!Sockets::SetNonBlocking(server))
{
    std::cout << "Erreur settings non bloquant : " << Sockets::GetError();
    return -3;
}
```

Ensuite, la modification de la boucle principale qui aura maintenant cette forme :

Boucle principale : acceptation des nouveaux clients et gestion des connectés et déconnectés

```
1. std::vector<Client> clients;
2. for (;;)
3. {
4.     {
5.         sockaddr_in from = { 0 };
6.         socklen_t addrlen = sizeof(from);
```

Boucle principale : acceptation des nouveaux clients et gestion des connectés et déconnectés

```

7.  SOCKET newClientSocket = accept(server, (SOCKADDR*)&from, &addrlen);
8.  if (newClientSocket != INVALID_SOCKET)
9.  {
10.     if (!Sockets::SetNonBlocking(newClientSocket))
11.     {
12.        std::cout << "Erreur settings nouveau socket non bloquant :
" << Sockets::GetError() << std::endl;
13.        Sockets::CloseSocket(newClientSocket);
14.        continue;
15.     }
16.     Client newClient;
17.     newClient.sckt = newClientSocket;
18.     newClient.addr = from;
19.     const std::string clientAddress = Sockets::GetAddress(from);
20.     const unsigned short clientPort = ntohs(from.sin_port);
21.     std::cout << "Connexion de " << clientAddress.c_str() << ":" << clientPort << std::endl;
22.     clients.push_back(newClient);
23. }
24. }
25. {
26.     auto itClient = clients.begin();
27.     while ( itClient != clients.end() )
28.     {
29.         const std::string clientAddress = Sockets::GetAddress(itClient->addr);
30.         const unsigned short clientPort = ntohs(itClient->addr.sin_port);
31.         char buffer[200] = { 0 };
32.         bool disconnect = false;
33.         int ret = recv(itClient->sckt, buffer, 199, 0);
34.         if (ret == 0)
35.         {
36.             //!< Déconnecté
37.             disconnect = true;
38.         }
39.         if (ret == SOCKET_ERROR)
40.         {
41.             int error = Sockets::GetError();
42.             if (error != static_cast<int>(Sockets::Errors::WOULDBLOCK))
43.             {
44.                 disconnect = true;
45.             }
46.             //!< il n'y avait juste rien à recevoir
47.         }
48.         std::cout << "[" << clientAddress << ":" << clientPort << "]" << buffer << std::endl;
49.         ret = send(itClient->sckt, buffer, ret, 0);
50.         if (ret == 0 || ret == SOCKET_ERROR)
51.         {
52.             disconnect = true;
53.         }
54.         if (disconnect)
55.         {
56.             std::cout << "Deconnexion de [" << clientAddress << ":" <<
clientPort << "]" << std::endl;
57.             itClient = clients.erase(itClient);
58.         }
59.         else
60.             ++itClient;
61.     }
62. }
63. }

```

Notez qu'il faut définir chaque socket accepté comme non bloquant également.

En théorie, il faudrait également vérifier que l'erreur retournée par `send` ne soit pas `WOULDBLOCK`. En pratique il est très difficile de faire bloquer `send`, il faudrait vouloir envoyer une très grande quantité de données, ce que ne fait pas ce programme. Plus d'informations sous le terme de « `send buffer size` » dans votre moteur de recherche préféré et dans une partie précédente **Puis-je vraiment envoyer 64 ko de données ?**

[Télécharger le code source de l'exemple](#)**Article précédent**[<< Multi-threading et mutex](#)**Article suivant**[TCP - Mode non bloquant pour le client](#)

Cours programmation réseau en C++

TCP - Mode non bloquant pour le client

Par [Bousk](#)

Date de publication : 27 avril 2017

Dernière mise à jour : 10 juin 2017

DÉBUTANT

Voyons rapidement comment utiliser le mode non bloquant pour le client.

Commentez

I - Mode non bloquant, select ou poll.....	3
II - connect.....	3

I - Mode non bloquant, select ou poll

Afin d'utiliser un socket non bloquant, nous pouvons avoir recours au mode non bloquant, se servir de select ou de poll, qui ont une utilisation et mise en place identiques à celles que l'on a vues dans [l'article précédent](#) dans le cadre d'un serveur.

Quelle que soit la solution choisie, un avantage direct est de ne plus avoir besoin de threads pour traiter le réseau, éliminant ainsi le besoin de synchronisation, l'utilisation de mutex et tous les problèmes habituellement rencontrés dès lors que l'on parle de multithread. De plus vous aurez un meilleur contrôle des actions en cours, en pouvant interrompre à tout moment, par exemple une connexion - chose impossible auparavant.

II - connect

Pour un client TCP, connect pose un problème avec le mode non bloquant : bien que connect retourne immédiatement, comment alors savoir quand la connexion est réellement établie ? Ou a échoué ?

Dans ce cas, il faut coupler le mode non bloquant, qui est nécessaire afin que connect ne bloque pas, à l'utilisation de select ou de poll. Quand le socket est prêt en écriture, c'est que le processus de connexion est terminé. Il faut alors récupérer l'état du socket pour vérifier s'il y a une erreur, ou si tout s'est bien passé et la connexion est établie.

Comme pour les appels à recv et send non bloquant, connect retournera une erreur, il faudra alors vérifier de quelle erreur il s'agit : s'il s'agit de EINPROGRESS, c'est juste l'erreur liée au mode non bloquant, l'équivalent de l'erreur EWOULDBLOCK pour recv et send. Sinon il s'agit d'une vraie erreur survenue. Windows aime se faire remarquer et retournera EWOULDBLOCK suite à une erreur d'appel à connect non bloquant. Faites donc attention à ce cas particulier et si votre code est à vocation d'être porté, il faudra sans doute vérifier les deux valeurs.

connect non bloquant et poll pour vérifier le résultat

```
sockaddr_in server;
inet_pton(AF_INET, "127.0.0.1", &server.sin_addr.s_addr);
server.sin_family = AF_INET;
server.sin_port = htons(9999);
SOCKET client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
Sockets::SetNonBlocking(client);
if (connect(client, reinterpret_cast<const sockaddr*>(&server), sizeof(server)) != 0)
{
    int error = Sockets::GetError();
    if (error != static_cast<int>(Sockets::Errors::INPROGRESS) && error !=
        static_cast<int>(Sockets::Errors::WOULDBLOCK))
    {
        std::cout << "Erreur connect : " << error << std::endl;
        return -2;
    }
}
bool connected = false;
pollfd fd = { 0 };
fd.fd = client;
fd.events = POLLOUT;
while(true)
{
    int ret = poll(&fd, 1, 0);
    if (ret == -1)
    {
        std::cout << "Erreur poll : " << Sockets::GetError() << std::endl;
        break;
    }
    else if (ret > 0)
    {
        if (fd.revents & POLLOUT)
        {
            connected = true;
            std::cout << "Socket connecté" << std::endl;
            break;
        }
    }
}
```


connect non bloquant et poll pour vérifier le résultat

```
}
else if (fd.revents & (POLLHUP | POLLNVAL))
{
    std::cout << "Socket deconnecte" << std::endl;
    break;
}
else if (fd.revents & POLLERR)
{
    socklen_t err;
    int errsize = sizeof(err);
    if (getsockopt(client, SOL_SOCKET, SO_ERROR, reinterpret_cast<char*>(&err), &errsize) != 0)
    {
        std::cout << "Impossible de determiner l'erreur : " << Sockets::GetError() << std::endl;
    }
    else if (err != 0)
        std::cout << "Erreur : " << err << std::endl;
    break;
}
}
}

if (connected)
{
    //!< connexion établie avec succès
}
```

Si vous êtes familier avec **le chapitre 1** et **le chapitre 5**, ce code ne devrait poser aucun souci de compréhension. Il s'agit de l'application directe de l'explication qui précède, avec réutilisation des codes des parties précédentes. `poll` peut être remplacé par `select` selon votre préférence. L'utilisation d'une boucle infinie pour attendre que la connexion soit établie n'est pas représentative d'un exemple réel, mais a juste pour but de montrer comment la logique est réalisée. Dans un code réel, il s'agira d'appeler ce code régulièrement, typiquement une fois par itération de la boucle principale du logiciel ou de la boucle de gameplay, afin de vérifier la progression de la connexion.

Article précédent**<< TCP - Envoi et réception depuis le serveur****Article suivant****TCP - Quelle architecture de client visée ?**

Cours programmation réseau en C++

TCP - Quelle architecture de client visée ?

Par [Bousk](#)

Date de publication : 30 mai 2017

Dernière mise à jour : 19 août 2017

DÉBUTANT

Dans cette partie nous allons définir l'architecture du code client que nous souhaitons utiliser.

Pour cela nous allons mettre à jour notre protocole et nos interfaces.

Commentez

I - Quel client voulons-nous ?.....	3
I-A - Propriétés souhaitées.....	3
I-B - Un message ?.....	3
II - Nouvel élément : les messages.....	4
II-A - Comment savoir de quel Message il s'agit ?.....	4
II-A-1 - Interface de conversion.....	4
III - Comment rendre notre classe non bloquante ?.....	7
III-A - recv.....	7
III-B - connect.....	7
III-C - send.....	8
III-D - Le client.....	9
IV - Implémentation des différents modules.....	10
IV-A - Propriétés de la classe Client et son socket.....	10
IV-B - Gestionnaire de connexion.....	11
IV-B-1 - bool connect(SOCKET sckt, const std::string& address, unsigned short port);.....	11
IV-B-2 - std::unique_ptr<Messages::Connection> poll();.....	12
IV-B-2-a - Changements liés à poll sous Windows.....	13
IV-B-3 - Interface finale de ConnectionHandler.....	13
IV-C - Gestionnaire de réception des données.....	13
IV-C-1 - std::unique_ptr<Messages::Base> recv();.....	13
IV-C-1-a - Utilisation des membres mBuffer et mReceived.....	14
IV-C-1-b - Implémentation de std::unique_ptr<Messages::Base> recv();.....	14
IV-C-1-b-1 - Initialiser la réception de l'en-tête.....	15
IV-C-1-b-2 - Initialiser la réception des données.....	15
IV-C-2 - Initialisation de la réception.....	16
IV-C-3 - Interface finale de ReceptionHandler.....	16
IV-D - Gestionnaire d'envoi des données.....	17
IV-D-1 - bool send(const unsigned char* data, unsigned int datalen);.....	17
IV-D-2 - void update();.....	17
IV-D-2-a - Utilisation de mSendingBuffer.....	17
IV-D-2-b - Implémentation de void update();.....	17
IV-D-2-b-1 - bool sendPendingBuffer();.....	18
IV-D-2-b-2 - Initialisation de l'envoi de l'en-tête.....	19
IV-D-2-b-3 - Initialisation de l'envoi des données.....	19
IV-D-3 - void init(SOCKET sckt);.....	19
IV-D-4 - size_t queueSize() const;.....	20
IV-D-5 - Interface finale de SendingHandler.....	20
V - Implémentation de ClientImpl.....	21
V-A - Destructeur.....	21
V-B - bool connect(const std::string& ipaddress, unsigned short port);.....	21
V-C - void disconnect();.....	22
V-D - bool send(const unsigned char* data, unsigned int len);.....	22
V-E - std::unique_ptr<Messages::Base> poll();.....	22
VI - Utilisation de Client dans le programme.....	23

I - Quel client voulons-nous ?

I-A - Propriétés souhaitées

Le client qui nous intéresse sera non bloquant pour l'application principale, et rendra les données qu'il possède disponibles à l'application sur simple requête.

Il s'agira d'un client TCP. La latence n'est donc pas une priorité et il s'adaptera plutôt pour une utilisation avec un serveur de lobby (discussions, textes...), gameplay non temps réel (Civilization, Blood Bowl) ou dans un cas où la latence n'est pas vraiment un souci et peut être absorbée. La plupart des MMO utilisent également un client TCP pour ne pas ajouter une couche supplémentaire de complexité sur la difficulté de réaliser un tel projet.

Son interface finale devrait être proche de :

Interface souhaitée pour le client

```
1. class TCPClient
2. {
3.     public:
4.         TCPClient();
5.         ~TCPClient();
6.
7.         bool connect(const std::string& ipaddress, unsigned short port);
8.         void disconnect();
9.         bool send(const unsigned char* data, unsigned int len);
10.        std::unique_ptr<Message> poll();
11. };
```

Par rapport à l'interface mise en place dans la **partie 3**, vous remarquerez que Recv a disparu, laissant sa place à poll.

poll sera appelé pour extraire un Message du socket. Tout le mécanisme de réception des données et création d'un paquet complet vu dans le **chapitre 3 Mise en place du protocole** sera désormais interne à la bibliothèque. L'utilisateur de cette classe ne s'occupera pas de réceptionner des données arbitraires, mais directement ce que nous avons appelé un paquet, sous la dénomination de message. En pratique, l'application devra appeler poll pour récupérer le dernier message, typiquement tant qu'il y en a un disponible et agir en fonction du message reçu.

Si l'utilisation via poll ne vous convient pas, vous pouvez opter pour un système de callbacks.

Ajoutez à l'interface du client une méthode par type de callback/message existant, et modifiez poll en update pour qu'il appelle ces callbacks en interne.



Veillez simplement à également fournir une fonction pour retirer une callback.

Nous resterons pour l'instant sur cette interface. La mise en place d'une solution à base de callbacks fera éventuellement l'objet d'un chapitre à part.

I-B - Un message ?

Notre Message sera la classe de base d'une hiérarchie de cette forme :

Messages réseau

```
1. class Message
2. {};
3. class Connection : public Message
4. {};
5. class Disconnection : public Message
```

Messages réseau

```
6. {};  
7. class UserData : public Message  
8. {};
```

Nous pourrions ajouter autant de types de messages que nous le souhaitons au fur et à mesure de l'évolution de notre bibliothèque, ceux-ci ne représentent que les actions basiques nécessaires jusque-là.

II - Nouvel élément : les messages

Le protocole mis en place dans le **chapitre correspondant** est toujours valable et sera utilisé en interne de notre bibliothèque. Il s'agit d'un détail d'implémentation dont l'utilisateur n'a à priori pas à se soucier.

Les messages servent d'abstraction entre la couche réseau et l'application. L'utilisateur de la bibliothèque travaillera avec ces messages. Leur utilité sera d'autant plus évidente quand nous implémenterons les sockets UDP, qui utiliseront également ce système de messages, permettant ainsi de passer d'une implémentation TCP à UDP avec très peu, voire aucun changement dans le code utilisateur.

Grâce à cette couche d'abstraction, l'écriture de tests devient également possible très simplement et vous pouvez fournir une interface pour simuler n'importe quel message sans même établir de connexion ni initialiser de socket.

II-A - Comment savoir de quel Message il s'agit ?

Puisque nous mettons en place du polymorphisme, il n'y aura pas beaucoup de solutions possibles. Soit avoir recours au `dynamic_cast`, soit proposer une interface qui le permette.

C'est cette seconde option que l'on retiendra : elle permet plus de flexibilité d'implémentation, vous pouvez vous contenter d'appeler un `dynamic_cast` en interne au début puis changer quand vous trouvez une solution plus performante.

II-A-1 - Interface de conversion

Nous aurons donc une interface template pour vérifier qu'un message est d'un type donné, et pouvoir l'y convertir.

L'interface aura cette forme :

Interface de conversion de Message

```
1. class Message  
2. {  
3. public:  
4.     template<class M>  
5.     bool is() const { ... }  
6.     template<class M>  
7.     const M* as() const { ... }  
8. };
```

Une implémentation utilisant `dynamic_cast` pourra être :

Conversion de Message utilisant dynamic_cast

```
1. class Message  
2. {  
3. public:  
4.     template<class M>  
5.     bool is() const { return dynamic_cast<const M*>(this) != nullptr; }  
6.     template<class M>  
7.     const M* as() const { return static_cast<const M*>(this); }  
8. };
```

Notez que c'est au client de s'assurer qu'une conversion est possible via un appel à `is` avant tout `as`. Vous pouvez opter pour une solution plus défensive et ajouter ce test en interne également. Une bonne pratique, qui n'apparaît pas ici, est d'ajouter ceci via un `assert`, afin que le test soit réalisé en configuration de débogage, puis retiré lors de la compilation de la version finale.

Je vous propose une autre implémentation qui n'utilise pas `dynamic_cast`, réputé lent et nécessitant le RTTI (RunTime Type Information) qui est parfois désactivé dans un souci de performance et dont l'implémentation dépend de la plateforme. (Voir [cette discussion sur StackOverflow traitant du RTTI en C++](#).)

Vu le faible nombre de messages, tous connus et internes à la couche réseau (l'application cliente ne pourra pas en définir), j'utilise une implémentation toute simple à base d'énumération, et quelques macros d'aide pour la génération de code :

Implémentation à base d'énumération

```
1. #define DECLARE_MESSAGE(name) friend class Message; static const Message::Type StaticType =  
    Message::Type::name  
2. class Message  
3. {  
4. public:  
5.     template<class M>  
6.     bool is() const { return mType == M::StaticType; }  
7.     template<class M>  
8.     const M* as() const { return static_cast<const M*>(this); }  
9.  
10. protected:  
11.     enum class Type {  
12.         Connection,  
13.         Disconnection,  
14.         UserData,  
15.     };  
16.     Message(Type type)  
17.     : mType(type)  
18.     {}  
19. private:  
20.     Type mType;  
21. };  
22. class Connection : public Message  
23. {  
24.     DECLARE_MESSAGE(Connection);  
25. public:  
26.     Connection()  
27.     : Message(Type::Connection)  
28.     {}  
29. };  
30. class Disconnection : public Message  
31. {  
32.     DECLARE_MESSAGE(Disconnection);  
33. public:  
34.     Disconnection()  
35.     : Message(Type::Disconnection)  
36.     {}  
37. };  
38. class UserData : public Message  
39. {  
40.     DECLARE_MESSAGE(UserData);  
41. public:  
42.     UserData()  
43.     : Message(Type::UserData)  
44.     {}  
45. };  
46. #undef DECLARE_MESSAGE
```

Le constructeur n'est pas public afin que l'on ne puisse pas créer un `Message` basique, mais uniquement un `Message` dérivé qui eux sont correctement définis. De même les types sont dans la portée `protected` afin que les classes dérivées puissent utiliser ces valeurs dans leur appel au constructeur de base, mais qu'ils ne soient pas accessibles par l'utilisateur.

Enfin la macro se chargera de définir une variable statique `StaticType` qui contiendra le `Type` de message de cette classe et déclarer une amitié de la classe dérivée vers la classe de base afin que cette variable soit accessible dans la fonction `is`, sans être accessible publiquement. `StaticType` est donc dans la portée `private` de la classe dérivée.

Finalement, par pure préférence personnelle, plaçons le tout dans un espace de nommage (`namespace`) approprié `Messages` et renommons `Message` en `Base`. Ajoutons également dès à présent des données supplémentaires qui nous seront utiles plus tard :

Messages.hpp

```
1. namespace Messages
2. {
3. #define DECLARE_MESSAGE(name) friend class Base; static const Base::Type StaticType =
   Base::Type::name
4. class Base
5. {
6. public:
7.     template<class M>
8.     bool is() const { return mType == M::StaticType; }
9.     template<class M>
10.    M* as() { return static_cast<M*>(this); }
11.    template<class M>
12.    const M* as() const { return static_cast<const M*>(this); }
13.
14. protected:
15.     enum class Type {
16.         Connected,
17.         ConnectionFailed,
18.         Disconnected,
19.         UserData,
20.     };
21.     Base(Type type)
22.         : mType(type)
23.     {}
24. private:
25.     Type mType;
26. };
27. class Connection : public Base
28. {
29.     DECLARE_MESSAGE(Connection);
30. public:
31.     enum class Result {
32.         Success,
33.         Failed,
34.     };
35.     Connection(Result r)
36.         : Base(Type::Connection)
37.         , result(r)
38.     {}
39.     Result result;
40. };
41. class Disconnection : public Base
42. {
43.     DECLARE_MESSAGE(Disconnection);
44. public:
45.     enum class Reason {
46.         Disconnected,
47.         Lost,
48.     };
49.     Disconnection(Reason r)
50.         : Base(Type::Disconnection)
51.         , reason(r)
52.     {}
53.     Reason reason;
54. };
55. class UserData : public Base
56. {
57.     DECLARE_MESSAGE(UserData);
58. public:
59.     UserData(std::vector<unsigned char>&& d)
60.         : Base(Type::UserData)
```

Messages.hpp

```
61.     , data(std::move(d))
62.     {}
63.     std::vector<unsigned char> data;
64. };
65. #undef DECLARE_MESSAGE
66. }
```

III - Comment rendre notre classe non bloquante ?

III-A - recv

`recv` est le plus simple aspect à traiter ici. Que l'on choisisse d'utiliser le mode non bloquant ou `select` ou `poll`, son comportement est très similaire et permet de réceptionner une partie des données demandées. Puisque les données peuvent être réceptionnées par fragments, on devra user d'un tampon interne pour les sauvegarder entre chaque appel.

Notre protocole utilise un en-tête indiquant la taille des données, avant d'envoyer les données elles-mêmes, et chacune de ces données peut être fragmentée. Une très simple machine à état sera utilisée pour indiquer si nous sommes en train de réceptionner un en-tête ou des données.

Pour simplifier la mise en place de ce mécanisme, ayons recours à une classe de cette forme :

Gestionnaire de réception des données

```
1. class ReceptionHandler
2. {
3.     enum class State {
4.         Header,
5.         Data,
6.     };
7. public:
8.     ReceptionHandler(SOCKET sockt);
9.     std::unique_ptr<Messages::Base> recv();
10.
11. private:
12.     size_t missingData() const { return mBuffer.size() - mReceived; }
13.
14. private:
15.     std::vector<unsigned char> mBuffer;
16.     unsigned int mReceived;
17.     SOCKET mSocket;
18.     State mState;
19. };
```

`recv` retourne directement le dernier message disponible, s'il y en a. En cas de déconnexion survenue, il sera capable de déterminer l'éventuelle erreur et retournera le message de déconnexion correspondant.

Tant que notre client est connecté, il suffira d'appeler cette méthode pour récupérer les données. Si un message est retourné, vérifiez s'il s'agit d'un message de déconnexion ou non pour modifier l'état de notre client avant de le retourner à l'application.

Toute la gestion de la taille du message, de la quantité de données manquantes, de la réception de l'en-tête et du passage du mode de réception de l'en-tête aux données sera interne à cette classe.

III-B - connect

Afin de rendre le `connect` non bloquant et aisément interruptible, nous utiliserons `poll` comme introduit au [chapitre précédent](#).

Et pourquoi ne pas limiter et encapsuler cette gestion dans sa propre classe ? Par exemple un premier jet pourrait être :

Gestionnaire de connexion

```
1. class ConnectionHandler
2. {
3. public:
4.     ConnectionHandler(SOCKET sckt);
5.     bool connect(const std::string& address, unsigned short port);
6.     std::unique_ptr<Messages::Connection> poll();
7.
8. private:
9.     pollfd mFd;
10.    std::string mAddress;
11.    unsigned short mPort;
12. };
```

Cette interface permet de connecter un socket donné à une adresse et port. `connect` pouvant échouer dès son appel, la valeur de retour sera utilisée pour déterminer si la connexion a été démarrée ou non. Ensuite il faut vérifier régulièrement, typiquement à chaque frame, si la connexion est finalement établie, ou a échoué en vérifiant si `poll` retourne un message et le champ `result` de celui-ci.

Grâce à cette interface à utilisation asynchrone, il sera plus simple de faire évoluer notre code par la suite. Pour l'instant nous nous sommes toujours connectés à une adresse IP directement, et nous continuerons ainsi pour le moment. Si nous voulons utiliser un nom d'hôte à la place, il faudra retrouver l'IP derrière cet hôte, ce qui est une opération asynchrone. Comme notre interface l'est déjà, le résultat sera différé pour l'utilisateur, mais il n'aura pas, ou que très peu, à changer l'utilisation qu'il en fait.

La sauvegarde de l'adresse et du port est optionnelle. Ils seront directement utilisés dans l'appel à `connect`, mais il est pratique de les garder en mémoire si on a besoin d'y accéder par la suite.

Le socket n'est apparemment pas sauvegardé, mais sera en fait enregistré dans le champ `fd` de la structure `mFd` utilisée pour l'appel à `poll`. Inutile donc de dupliquer cette information.

III-C - send

Enfin la dernière partie nécessaire à l'utilisation de notre socket : l'envoi de données. La *difficulté* réside dans la gestion des envois partiels et des erreurs non bloquantes qui nécessitent de renvoyer respectivement les données manquantes et la totalité des données.

Afin de décharger l'application de ces vérifications, ce mécanisme sera également géré en interne. Ce qui signifie qu'on devra user d'un tampon interne pour stocker les données avant que leur envoi ne soit effectif s'il n'est pas possible de les envoyer directement. Toute la mise en place du protocole, symétrique à la réception (en-tête suivi des données) sera également gérée en interne.

Ceci permettra également d'avoir certaines informations sur notre connexion : nous pourrions déterminer à chaque instant la quantité de données en file d'attente, ce qui peut être pratique pour les contrôles et éviter de noyer notre couche réseau. Vous pouvez ainsi limiter la file et générer des erreurs si elle atteint une taille critique, et avoir une idée à tout moment de la quantité de données que vous envoyez.

Vous pourrez aller plus loin en ajoutant par exemple une notion de durée de vie à chaque envoi, si des données n'ont pas été envoyées après un temps donné vous pouvez les ignorer et passer directement aux suivantes. Mais ceci sort du cadre de ce chapitre.

La structure d'une telle classe pourrait être :

Gestionnaire d'envoi des données

```
1. class SendingHandler
```

Gestionnaire d'envoi des données

```

2. {
3.   enum class State {
4.     Idle,
5.     Header,
6.     Data,
7.   };
8.   public:
9.     SendingHandler(SOCKET sckt);
10.    bool send(const unsigned char* data, unsigned int datalen);
11.    void update();
12.    size_t queueSize() const;
13.
14.   private:
15.     std::list<std::vector<unsigned char>> mQueueingBuffers;
16.     std::vector<unsigned char> mSendingBuffer;
17.     State mState;
18. };
  
```

L'appel à `send` devra mettre en file d'attente les données à envoyer, si possible. Une valeur retour `false` indiquera que les données ont été rejetées et ne seront pas envoyées.

Ensuite il convient d'appeler `update` régulièrement, typiquement à chaque frame. Cette fonction se chargera de l'envoi effectif des données quand possible, d'envoyer l'en-tête avant les données réelles et prendra en charge les envois partiels éventuels.

Contrairement aux classes `ConnectionHandler` et `ReceivingHandler`, nous n'avons aucune gestion d'erreur explicite ici. La raison est que cette classe est faite pour travailler en binôme avec la classe `ReceivingHandler`, et ce sera cette dernière qui nous informera d'une erreur.



Si vous souhaitez utiliser cette classe seule, dans le cadre d'un socket amené uniquement à envoyer des données et ne jamais en recevoir, vous pourrez par exemple changer la signature de la fonction `update` pour retourner un `bool` indiquant que la connexion a été terminée s'il vaut `false`, ou est toujours valide s'il vaut `true`.

III-D - Le client

Avec toutes ces spécificités internes, le pattern **pimpl** (pointer to implementation : pointeur vers l'implémentation) est un bon candidat : il permettra de modifier les mécanismes internes à notre guise sans avoir à changer l'interface publique de notre classe client.

Voici donc à quoi devrait ressembler notre en-tête :

En-tête du client TCP

```

1. #pragma once
2.
3. #include <string>
4. #include <memory>
5.
6. namespace Network
7. {
8.   namespace Messages {
9.     class Base;
10.   }
11.   namespace TCP
12.   {
13.     using HeaderType = uint16_t;
14.     static const unsigned int HeaderSize = sizeof(HeaderType);
15.     class ClientImpl;
16.     class Client
17.     {
  
```

En-tête du client TCP

```

18. public:
19.     Client();
20.     ~Client();
21.
22.     bool connect(const std::string& ipaddress, unsigned short port);
23.     void disconnect();
24.     bool send(const unsigned char* data, unsigned int len);
25.     std::unique_ptr<Messages::Base> poll();
26.
27. private:
28.     std::unique_ptr<ClientImpl> mImpl;
29. };
30. }
31. }

```



Les informations concernant l'en-tête d'un paquet sont également rendues disponibles publiquement dans ce fichier pour simplifier leur utilisation dans les différents modules et si un utilisateur souhaite y accéder pour quelque raison (information de débogage...).

IV - Implémentation des différents modules

IV-A - Propriétés de la classe Client et son socket

Notre classe possédera un constructeur par défaut qui n'initialisera pas le socket.

Le socket sera initialisé sur appel à `connect` uniquement et pas avant. Si l'initialisation du socket échoue, `connect` retournera directement `false` sans tenter d'amorcer la connexion.

Notre classe ne sera pas copiable. Sa destruction entraînera la fermeture du socket interne s'il était utilisé. Elle sera par contre déplaçable.

Le socket de notre classe Client utilisera le mode non bloquant. Ceci par pure préférence personnelle, vous pouvez très bien utiliser `select` ou `poll` pour l'envoi et la réception également.

Notre en-tête de Client sera de cette forme :

En-tête du client TCP

```

1. #pragma once
2.
3. #include <string>
4. #include <memory>
5.
6. namespace Network
7. {
8.     namespace Messages {
9.         class Base;
10.    }
11.    namespace TCP
12.    {
13.        using HeaderType = uint16_t;
14.        static const unsigned int HeaderSize = sizeof(HeaderType);
15.        class ClientImpl;
16.        class Client
17.        {
18.        public:
19.            Client();
20.            Client(const Client&) = delete;
21.            Client& operator=(const Client&) = delete;
22.            Client(Client&&);
23.            Client& operator=(Client&&);
24.            ~Client();

```

En-tête du client TCP

```

25.
26.     bool connect(const std::string& ipaddress, unsigned short port);
27.     void disconnect();
28.     bool send(const unsigned char* data, unsigned int len);
29.     std::unique_ptr<Messages::Base> poll();
30.
31. private:
32.     std::unique_ptr<ClientImpl> mImpl;
33. };
34. }
35. }

```

Au niveau de l'implémentation, il s'agit juste de transférer les appels au pointeur `mImpl` qui se charge d'implémenter réellement les mécanismes :

Client.cpp

```

1. namespace Network
2. {
3.     namespace TCP
4.     {
5.         Client::Client() {}
6.         Client::~~Client() {}
7.         Client::Client(Client&& other) : mImpl(std::move(other.mImpl)) {}
8.         Client& Client::operator=(Client&& other) { mImpl = std::move(other.mImpl); return *this; }
9.         bool Client::connect(const std::string& ipaddress, unsigned short port)
10.        {
11.            if (!mImpl)
12.                mImpl = std::make_unique<ClientImpl>();
13.            return mImpl && mImpl->connect(ipaddress, port);
14.        }
15.        void Client::disconnect() { if (mImpl) mImpl->disconnect(); mImpl.reset(); }
16.        bool Client::send(const unsigned char* data, unsigned int len) { return mImpl && mImpl->send(data, len); }
17.        std::unique_ptr<Messages::Base> Client::poll() { return mImpl ? mImpl->poll() : nullptr; }
18.    }
19. }

```

Assurez-vous juste de vérifier que le pointeur `mImpl` est toujours valide avant d'y accéder. Puisque notre classe propose les interfaces de déplacement, il peut se retrouver invalidé si elle a été déplacée - ou non initialisée.

L'initialisation du pointeur interne sera faite sur appel à `connect` uniquement, afin de limiter la mémoire utilisée si notre client n'est pas initialisé et pouvoir réutiliser et reconnecter un client déplacé.

IV-B - Gestionnaire de connexion

IV-B-1 - bool connect(SOCKET sckt, const std::string& address, unsigned short port);

Notre fonction `connect` prend en paramètre l'adresse, qui devra être, pour le moment, uniquement une adresse IP, et le port du serveur auquel se connecter.

Puisque notre classe `Client` initialise son socket tardivement, il n'est pas possible de créer directement un `ConnectionHandler` avec le socket sans user d'allocation dynamique. On ajoute donc le socket à utiliser en paramètre de `connect`.

Elle retournera `true` si la connexion a été amorcée, `false` sinon.

Point de vue implémentation, il s'agit d'une copie adaptée du **chapitre précédent** :

```

1. bool ConnectionHandler::connect(SOCKET sckt, const std::string& address, unsigned short port)
2. {

```

```

3.  assert(sckt != INVALID_SOCKET);
4.  mAddress = address;
5.  mPort = port;
6.  mFd.fd = sckt;
7.  mFd.events = POLLOUT;
8.  sockaddr_in server;
9.  inet_pton(AF_INET, mAddress.c_str(), &server.sin_addr.s_addr);
10. server.sin_family = AF_INET;
11. server.sin_port = htons(mPort);
12. if (::connect(sckt, (const sockaddr*)&server, sizeof(server)) != 0)
13. {
14.     int err = Errors::Get();
15.     if (err != Errors::INPROGRESS && err != Errors::WOULDBLOCK)
16.         return false;
17. }
18. return true;
19. }

```

Avec cette implémentation, seule la connexion à une adresse IPv4 est possible.



Cette classe sera étendue par la suite pour permettre de se connecter à des adresses IPv6, mais également à un nom d'hôte en recherchant l'IP de l'hôte.

IV-B-2 - std::unique_ptr<Messages::Connection> poll();

Notre fonction poll retournera `nullptr` tant que la connexion est en cours.

Une fois la connexion terminée, ou échouée, elle retournera le `Messages::Connection` correspondant en déterminant l'éventuelle erreur survenue afin de remplir le champ `Reason` du message.

Implémentation de poll

```

1. std::unique_ptr<Messages::Connection> ConnectionHandler::poll()
2. {
3.     int res = ::poll(&mFd, 1, 0);
4.     if (res < 0)
5.         return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
6.     else if (res > 0)
7.     {
8.         if (mFd.revents & POLLOUT)
9.         {
10.            return std::make_unique<Messages::Connection>(Messages::Connection::Result::Success);
11.        }
12.        else if (mFd.revents & (POLLHUP | POLLNVAL))
13.        {
14.            return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
15.        }
16.        else if (mFd.revents & POLLERR)
17.        {
18.            return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
19.        }
20.    }
21.    //!< action non terminée
22.    return nullptr;
23. }

```

Pour l'instant très simple, `Reason` peut valoir uniquement `Success` ou `Failed`. Mais par la suite nous pourrions ajouter une durée limitée et avoir un `Timeout`, ou bien échouer à traduire le nom d'hôte en IP. Libre à vous d'ajouter la granularité de vos possibilités sur les erreurs possibles selon votre plateforme par exemple.

IV-B-2-a - Changements liés à poll sous Windows

Dans le **chapitre introduisant poll**, nous nous étions contentés d'un `#define poll WSAPoll` sous Windows pour uniformiser la syntaxe d'appel avec les systèmes Unix.

Puisque `poll` est maintenant une fonction de notre interface, un simple `define` ne peut plus convenir. Ce n'était de toute façon qu'une écriture *quick & dirty* qui n'était pas faite pour durer : les macros de ce genre, à fortiori sur une fonction aussi banale et commune, peuvent vite dégénérer et modifier un code à l'insu de l'utilisateur.

Changeons donc dès à présent sa déclaration pour rediriger l'appel vers `WSAPoll` convenablement. `#define poll WSAPoll` devient donc `inline int poll(pollfd fdarray[], nfds_t nfds, int timeout) { return WSAPoll(fdarray, nfds, timeout); }`.

Je choisis de la déclarer `inline` par simplicité principalement. Vous pouvez tout aussi bien avoir le seul prototype dans l'en-tête `int poll(pollfd fdarray[], nfds_t nfds, int timeout);` et son implémentation dans un fichier cpp, en s'assurant qu'elle soit limitée à Windows :

Implémentation de poll pour Windows

```
1. #ifdef WIN32
2. int poll(pollfd fdarray[], nfds_t nfds, int timeout)
3. {
4.     return WSAPoll(fdarray, nfds, timeout);
5. }
6. #endif
```

IV-B-3 - Interface finale de ConnectionHandler

L'interface finale est très proche de celle introduite dans la première partie de ce chapitre. Le résultat avec l'initialisation des variables membres est :

Interface finale de ConnectionHandler

```
1. namespace Network
2. {
3.     namespace TCP
4.     {
5.         class ConnectionHandler
6.         {
7.         public:
8.             ConnectionHandler() = default;
9.             bool connect(SOCKET sockt, const std::string& address, unsigned short port);
10.            std::unique_ptr<Messages::Connection> poll();
11.
12.        private:
13.            pollfd mFd{ 0 };
14.            std::string mAddress;
15.            unsigned short mPort;
16.        };
17.    }
18. }
```

IV-C - Gestionnaire de réception des données

IV-C-1 - std::unique_ptr<Messages::Base> recv();

Notre fonction `recv` doit assurer la bonne mise en place de notre protocole.

Notre protocole se caractérise par l'envoi d'un en-tête qui contient la taille du paquet à suivre, codé sur 16 bits, puis les données.

Une fois la connexion effectuée, les premières données que nous recevrons seront forcément un en-tête. Après réception de l'en-tête complet, nous savons quelle quantité de données nous devons nous attendre à recevoir pour le paquet.

Une fois la réception des données terminées, nous avons un paquet entier disponible que nous pouvons transmettre à l'application via un **Messages::UserData**, et nous pouvons repasser en mode réception de l'en-tête suivant.

IV-C-1-a - Utilisation des membres mBuffer et mReceived

À chaque instant, mBuffer servira de tampon de réception. Puisque nous connaissons toutes les données à recevoir, un `sizeof(uint16_t)` pour l'en-tête et la valeur de l'en-tête pour les données, nous pouvons préparer ce tampon à la taille convenable en utilisant **std::vector::resize**.

mReceived enregistrera les données déjà reçues.

La quantité de données manquantes sera tout simplement la différence entre la taille de mBuffer et la valeur de mReceived : `int missingDataLength() const { return static_cast<int>(mBuffer.size() - mReceived); }`.

De la même manière, pour savoir où se situer dans le tampon pour la réception, il suffira de décaler de mReceived le pointeur de mBuffer : `char* missingDataStartBuffer() { return reinterpret_cast<char*>(mBuffer.data() + mReceived); }`.

IV-C-1-b - Implémentation de std::unique_ptr<Messages::Base> recv();

recv se chargera de la réception des données dans mBuffer, dont l'utilisation a été expliquée dans le paragraphe précédent et de retourner les données quand elles ont été intégralement reçues dans un **Messages::UserData**.

recv servira également à intercepter les déconnexions et retournera un **Messages::Disconnection** correspondant si cela se produit.

Finalement, si aucune donnée n'a été reçue ou que le paquet n'est pas encore prêt - pas encore réceptionné entièrement - et que la connexion est toujours valide, la valeur de `nullptr` sera retournée.

Implémentation de recv

```
1. std::unique_ptr<Messages::Base> ReceptionHandler::recv()
2. {
3.     assert(mSckt != INVALID_SOCKET);
4.     int ret = ::recv(mSckt, missingDataStartBuffer(), missingDataLength(), 0);
5.     if (ret > 0)
6.     {
7.         mReceived += ret;
8.         if (mReceived == mBuffer.size())
9.         {
10.            if (mState == State::Data)
11.            {
12.                std::unique_ptr<Messages::Base>
13.                msg = std::make_unique<Messages::UserData>(std::move(mBuffer));
14.                startHeaderReception();
15.                return msg;
16.            }
17.            else
18.            {
19.                startDataReception();
20.                //!< si jamais les données sont déjà disponibles elles seront ainsi retournées
21.                directement
22.                return recv();
23.            }
24.        }
25.        return nullptr;
26.    }
27.    else if (ret == 0)
```

Implémentation de recv

```

26. {
27.     //!< connexion terminée correctement
28.     return std::make_unique<Messages::Disconnection>(Messages::Disconnection::Reason::Disconnected);
29. }
30. else // ret < 0
31. {
32.     //!< traitement d'erreur
33.     int error = Errors::Get();
34.     if (error == Errors::WOULDBLOCK || error == Errors::AGAIN)
35.     {
36.         return nullptr;
37.     }
38.     else
39.     {
40.         return std::make_unique<Messages::Disconnection>(Messages::Disconnection::Reason::Lost);
41.     }
42. }
43. }

```

Nous commençons par réceptionner des données, en limitant à la quantité que nous attendons. Si des données ont été reçues, nous mettons à jour `mReceived`.

Si nous avons reçu la quantité de données attendues, `mReceived == mBuffer.size()`, alors le paquet est complet. Commence alors le traitement du paquet en question.

Si nous étions en mode en-tête, `mState == State::Header`, nous commençons immédiatement à récupérer les données du paquet. Si nous étions déjà en mode données, `mState == State::Data`, nous retournons au mode réception du prochain en-tête et créons le message pour l'application.

IV-C-1-b-1 - Initialiser la réception de l'en-tête

Notre en-tête sera une donnée comme une autre à réceptionner. La seule différence réside dans l'état de notre receveur qui sera en mode `State::Header`.

Sa taille est connue, puisque décidée par nous-mêmes lors de l'élaboration de la bibliothèque, et vaut 16 bits, `sizeof(uint16_t)` octets (2 octets sur à peu près toutes les machines standards).

```

1. void ReceptionHandler::startHeaderReception()
2. {
3.     mReceived = 0;
4.     mBuffer.clear();
5.     mBuffer.resize(HeaderSize, 0);
6.     mState = State::Header;
7. }

```

IV-C-1-b-2 - Initialiser la réception des données

Après réception de l'en-tête, le tampon contiendra la valeur de la quantité de données à attendre pour le prochain paquet, enregistrée en boutisme réseau qu'il faudra donc convertir en boutisme local.

```

1. void ReceptionHandler::startDataReception()
2. {
3.     assert(mBuffer.size() == sizeof(HeaderType));
4.     HeaderType networkExpectedDataLength;
5.     memcpy(&networkExpectedDataLength, mBuffer.data(), sizeof(networkExpectedDataLength));
6.     const auto expectedDataLength = ntohs(networkExpectedDataLength);
7.     mReceived = 0;
8.     mBuffer.clear();
9.     mBuffer.resize(expectedDataLength, 0);
10.    mState = State::Data;

```



```
11. }
```

Vu la similitude, sans grande surprise, de ces deux fonctions, nous pouvons les factoriser ainsi afin de limiter les erreurs :

Factorisation de la réception des en-têtes et des données

```
1. void ReceptionHandler::startHeaderReception()
2. {
3.     startReception(HeaderSize, State::Header);
4. }
5. void ReceptionHandler::startDataReception()
6. {
7.     assert(mBuffer.size() == sizeof(HeaderType));
8.     HeaderType networkExpectedDataLength;
9.     memcpy(&networkExpectedDataLength, mBuffer.data(), sizeof(networkExpectedDataLength));
10.    const auto expectedDataLength = ntohs(networkExpectedDataLength);
11.    startReception(expectedDataLength, State::Data);
12. }
13. void ReceptionHandler::startReception(unsigned int expectedDataLength, State newState)
14. {
15.     mReceived = 0;
16.     mBuffer.clear();
17.     mBuffer.resize(expectedDataLength, 0);
18.     mState = newState;
19. }
```

Ce qui renforce la remarque initiale : l'en-tête est une donnée à récupérer comme une autre.

IV-C-2 - Initialisation de la réception

Une fois notre socket connecté, nous devons initialiser la réception des données. Ajoutons donc une fonction `init` à notre `ReceptionHandler` afin d'indiquer le socket sur lequel travailler et démarrer la réception des données, ou plutôt de l'en-tête :

```
1. void ReceptionHandler::init(SOCKET sckt)
2. {
3.     assert(sckt != INVALID_SOCKET);
4.     mSckt = sckt;
5.     startHeaderReception();
6. }
```

IV-C-3 - Interface finale de ReceptionHandler

Interface finale de ReceptionHandler

```
1. class ReceptionHandler
2. {
3.     enum class State {
4.         Header,
5.         Data,
6.     };
7. public:
8.     ReceptionHandler() = default;
9.     void init(SOCKET sckt);
10.    std::unique_ptr<Messages::Base> recv();
11.
12. private:
13.    inline char* missingDataStartBuffer() { return reinterpret_cast<char*>(mBuffer.data() +
14.        mReceived); }
15.    inline int missingDataLength() const { return static_cast<int>(mBuffer.size() -
16.        mReceived); }
17.    void startHeaderReception();
18.    void startDataReception();
19.    void startReception(unsigned int expectedDataLength, State newState);
20. }
```

Interface finale de ReceptionHandler

```
19. private:
20.     std::vector<unsigned char> mBuffer;
21.     unsigned int mReceived;
22.     SOCKET mSckt{ INVALID_SOCKET };
23.     State mState;
24. };
```

IV-D - Gestionnaire d'envoi des données

IV-D-1 - bool send(const unsigned char* data, unsigned int datalen);

La fonction `send` validera les données avant de les mettre en file d'attente d'envoi et retourner `true`, ou les ignorer et retourner `false`. Les données seront dans une **liste FIFO** - First In, First Out - : elles seront envoyées dans l'ordre de leur passage à `send`.

Pour l'instant la seule limitation à l'envoi de nos données est que leur quantité ne dépasse pas la taille de l'en-tête du protocole : la valeur maximale stockable dans un `uint16_t`.

```
1. bool SendingHandler::send(const unsigned char* data, unsigned int datalen)
2. {
3.     if (datalen > std::numeric_limits<HeaderType>::max())
4.         return false;
5.     mQueueingBuffers.emplace_back(data, data + datalen);
6.     return true;
7. }
```

IV-D-2 - void update();

`update` sera le cœur de l'implémentation de l'envoi des données. Il s'agira d'envoyer autant de données que possible à chaque appel.

Si un envoi est en cours, que l'envoi précédent n'a été que partiel, nous devons renvoyer les données manquantes.

Si aucune donnée n'est en cours d'envoi, nous envoyons les données suivantes dans la file d'envoi si elle n'est pas vide.

En cas d'erreur, nous l'ignorons et abandonnons l'opération en cours. L'erreur sera interceptée et traitée par le gestionnaire de réception.

IV-D-2-a - Utilisation de mSendingBuffer

`mSendingBuffer` sera à chaque instant notre tampon à envoyer. Quand nous voudrions envoyer des données, nous les placerons dans ce tampon. Pour connaître la quantité de données restant à envoyer, il s'agira de la taille du tampon. Après chaque envoi réussi, il faudra donc l'amputer d'autant afin de garder ses informations cohérentes.

IV-D-2-b - Implémentation de void update();

```
1. void SendingHandler::update()
2. {
3.     assert(mSocket != INVALID_SOCKET);
4.     if (mState == State::Idle && !mQueueingBuffers.empty())
5.     {
6.         prepareNextHeader();
7.     }
8.     while (mState != State::Idle && sendPendingBuffer())
9.     {
```

```

10.
11.     if (mState == State::Header)
12.     {
13.         prepareNextData();
14.     }
15.     else
16.     {
17.         if (!mQueueingBuffers.empty())
18.         {
19.             prepareNextHeader();
20.         }
21.         else
22.         {
23.             mState = State::Idle;
24.         }
25.     }
26. }
27. }

```

Notre classe débute toujours en état `Idle`, la première partie commencera l'envoi des données après un arrêt et l'initialisation, si des données ont été mises en file d'envoi depuis.

Ensuite, tant que nous ne sommes pas en arrêt, nous envoyons les données en cours. `sendPendingBuffer` se charge d'envoyer les données en cours et retourne `true` si toutes les données ont été envoyées, et que nous pouvons passer à la suite, `false` sinon.

Si l'envoi en cours est terminé et que nous étions en mode d'envoi de l'en-tête, `mState == State::Header`, on démarre l'envoi des données actuellement mises en file et attendues par l'utilisateur. Si nous étions déjà en mode envoi des données, `mState == State::Data`, et qu'il reste des données en attente d'envoi, on repasse en mode en-tête et continuons l'envoi. Sinon on repasse à l'arrêt et en attente de données.

IV-D-2-b-1 - bool sendPendingBuffer();

Cette fonction devra envoyer les données manquantes de l'envoi en cours et retournera `true` si toutes les données ont été envoyées, `false` si l'envoi n'a été que partiel et que des données restent à envoyer.

```

1. bool SendingHandler::sendPendingBuffer()
2. {
3.     if (mSendingBuffer.empty())
4.         return true;
5.
6.     //!< envoi des données restantes du dernier envoi
7.     int
8.     sent = ::send(mSocket, reinterpret_cast<char*>(mSendingBuffer.data()), static_cast<int>(mSendingBuffer.size()),
9.     if (sent > 0)
10.    {
11.        if (sent == mSendingBuffer.size())
12.        {
13.            //!< toutes les données ont été envoyées
14.            mSendingBuffer.clear();
15.            return true;
16.        }
17.        else
18.        {
19.            //!< envoi partiel
20.            memmove(mSendingBuffer.data() + sent, mSendingBuffer.data(), sent);
21.            mSendingBuffer.erase(mSendingBuffer.cbegin() + sent, mSendingBuffer.cend());
22.        }
23.    }
24.    return false;
25. }

```



Petite astuce : plutôt que de supprimer les données en début de tampon après leur envoi, il est plus efficace de déplacer les données au début du vector afin d'en supprimer la fin.

IV-D-2-b-2 - Initialisation de l'envoi de l'en-tête

D'après les propriétés de notre protocole, l'en-tête est un entier `uint16_t` qui représente la taille - la quantité de données - du paquet à venir. Le paquet à venir étant le premier paquet présent dans `mQueueingBuffers`.

La préparation de l'envoi de l'en-tête sera le miroir du code de leur réception du paragraphe précédent :

Préparation de l'envoi de l'en-tête

```
1. void SendingHandler::prepareNextHeader()
2. {
3.     assert(!mQueueingBuffers.empty());
4.     auto header = static_cast<HeaderType>(mQueueingBuffers.front().size());
5.     const auto networkHeader = htons(header);
6.     mSendingBuffer.clear();
7.     mSendingBuffer.resize(HeaderSize);
8.     memcpy(mSendingBuffer.data(), &networkHeader, sizeof(HeaderType));
9.     mState = State::Header;
10. }
```

On s'assure de convertir en boutisme réseau la taille avant de la copier dans le tampon d'envoi, puis on change l'état en envoi d'en-tête.

IV-D-2-b-3 - Initialisation de l'envoi des données

Les données seront celles présentes en tête de `mQueueingBuffer`. Préparer les données à l'envoi reviendra à copier ces données dans le tampon d'envoi, puis modifier l'état de l'envoyeur en mode données :

Préparation de l'envoi des données

```
1. void SendingHandler::prepareNextData()
2. {
3.     assert(!mQueueingBuffers.empty());
4.     mSendingBuffer.swap(mQueueingBuffers.front());
5.     mQueueingBuffers.pop_front();
6.     mState = State::Data;
7. }
```



Si vous comptez appeler `send` et `update` dans des threads différents, il faudra synchroniser les opérations sur `mQueueingBuffers` avec un mutex.

IV-D-3 - void init(SOCKET sckt);

Tout comme pour la réception, il faudra initialiser notre gestionnaire d'envoi après connexion.

```
1. void SendingHandler::init(SOCKET sckt)
2. {
3.     mSocket = sckt;
4.     mQueueingBuffers.clear();
5.     mSendingBuffer.clear();
6.     mState = State::Idle;
7. }
```

Cette version purgera toutes les données qui seraient toujours en file lors de l'appel. En cas de reconnexion par exemple.

Si vous préférez pouvoir garder les données précédemment mises en file, vous pouvez opter pour cette implémentation :

```
1. void SendingHandler::init(SOCKET sckt)
```

```

2. {
3.   mSocket = sckt;
4.   if (mState == State::Header || mState == State::Data)
5.   {
6.     mSendingBuffer.clear();
7.   }
8.   mState = State::Idle;
9. }

```

Les données qui étaient en cours d'envoi ne peuvent malheureusement pas être conservées, on ignore ce qui a réellement été transmis ou non, il faudra donc les purger dans tous les cas.

De même, comme l'application n'a aucune idée de ce qui a été envoyé ou non, à moins de recevoir une réponse du serveur, il n'est pas forcément avantageux de garder les envois précédents en cas de reconnexion. Ce n'est donc pas l'implémentation que je conseille ou utiliserai dans ce cours.

IV-D-4 - size_t queueSize() const;

Voici une implémentation de la fonction utilitaire présentée plus tôt afin de connaître la quantité de données en file d'envoi :

```

1. #include <numeric>
2. size_t SendingHandler::queueSize() const
3. {
4.   size_t s = std::accumulate(mQueueingBuffers.cbegin(),
5.                               mQueueingBuffers.cend(), static_cast<size_t>(0), [](size_t n, const std::vector<unsigned char>&
6.                               queuedItem) {
7.     return n + queuedItem.size() + HeaderSize;
8.   });
9.   if (mState == State::Data)
10.    s += mSendingBuffer.size();
11.   return s;
12. }

```

Retourne la quantité de données actuellement en file d'envoi. N'oubliez pas d'ajouter la taille de l'en-tête pour chaque donnée. Si des données sont actuellement en train d'être envoyées, il faut aussi les ajouter au total.

Si le gestionnaire est en train d'envoyer un en-tête, cette implémentation retournera une valeur erronée d'une poignée d'octets, ce qui n'est pas bien grave étant donné la rareté de la chose (le gestionnaire d'envoi devrait se trouver dans un état en-tête qu'extrêmement rarement, puisqu'il enverra l'en-tête avec succès avant d'immédiatement retourner en état envoi de données l'immense majorité du temps) et que ce résultat n'est toujours qu'une information sur l'état de nos échanges plus qu'une donnée réelle.

IV-D-5 - Interface finale de SendingHandler

Interface finale de SendingHandler

```

1. class SendingHandler
2. {
3.   enum class State {
4.     Idle,
5.     Header,
6.     Data,
7.   };
8. public:
9.   SendingHandler() = default;
10.  void init(SOCKET sckt);
11.  bool send(const unsigned char* data, unsigned int datalen);
12.  void update();
13.  size_t queueSize() const;
14. private:
15.  bool sendPendingBuffer();

```

Interface finale de SendingHandler

```

17. void prepareNextHeader();
18. void prepareNextData();
19.
20. private:
21.     std::list<std::vector<unsigned char>> mQueueingBuffers;
22.     std::vector<unsigned char> mSendingBuffer;
23.     SOCKET mSocket{ INVALID_SOCKET };
24.     State mState{ State::Idle };
25. };
  
```

V - Implémentation de ClientImpl

Maintenant que nos différents modules sont implémentés, il est temps de les connecter et de les utiliser dans notre implémentation *ClientImpl*.

Notre interface devrait ressembler à :

```

1. class ClientImpl
2. {
3.     enum class State {
4.         Connecting,
5.         Connected,
6.         Disconnected,
7.     };
8.
9. public:
10.    ClientImpl() = default;
11.    ~ClientImpl();
12.
13.    bool connect(const std::string& ipaddress, unsigned short port);
14.    void disconnect();
15.    bool send(const unsigned char* data, unsigned int len);
16.    std::unique_ptr<Messages::Base> poll();
17.
18. private:
19.    ConnectionHandler mConnectionHandler;
20.    SendingHandler mSendingHandler;
21.    ReceptionHandler mReceivingHandler;
22.    SOCKET mSocket{ INVALID_SOCKET };
23.    State mState{ State::Disconnected };
24. };
  
```

Nous n'avons fait que reprendre l'interface de la classe Client et avons ajouté les différents membres pour chaque module, le socket et une machine à état, que nous initialisons aux valeurs par défaut de `INVALID_SOCKET` et `Disconnected`.

V-A - Destructeur

Le destructeur n'aura qu'à s'assurer que notre socket est correctement déconnecté et libéré :

```

1. ClientImpl::~ClientImpl()
2. {
3.     disconnect();
4. }
  
```

La logique pour vérifier que le socket est déjà déconnecté ou non sera à l'intérieur de la fonction `disconnect`.

V-B - bool connect(const std::string& ipaddress, unsigned short port);

`connect` devra initialiser le socket puis amorcer la connexion via le module de *ConnectionHandler*.

Pour l'initialisation d'un socket, souvenez-vous du **premier chapitre à ce sujet**.

```
1. bool ClientImpl::connect(const std::string& ipaddress, unsigned short port)
2. {
3.     assert(mState == State::Disconnected);
4.     assert(mSocket == INVALID_SOCKET);
5.     if (mSocket != INVALID_SOCKET)
6.         disconnect();
7.     mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
8.     if (mSocket == INVALID_SOCKET)
9.     {
10.         return false;
11.     }
12.     else if (!SetNonBlocking(mSocket))
13.     {
14.         disconnect();
15.         return false;
16.     }
17.     if (mConnectionHandler.connect(mSocket, ipaddress, port))
18.     {
19.         mState = State::Connecting;
20.         return true;
21.     }
22.     return false;
23. }
```

Cette fonction ne devrait être appelée que sur un Client déconnecté, d'où les assertions des premières lignes.

Cependant, pour simplifier l'utilisation et ne pas perturber le fonctionnement en cas de problème, on s'assurera de simplement déconnecter le socket s'il était déjà utilisé.

V-C - void disconnect();

Cette fonction aura deux objectifs : libérer le socket et réinitialiser l'état de la classe à `Disconnected`.

```
1. void ClientImpl::disconnect()
2. {
3.     if (mSocket != INVALID_SOCKET)
4.     {
5.         CloseSocket(mSocket);
6.     }
7.     mSocket = INVALID_SOCKET;
8.     mState = State::Disconnected;
9. }
```

`CloseSocket` pourrait être appelé dans tous les cas, il n'aurait aucun effet si le socket était déjà déconnecté. Mais il est toujours plus clair de réaliser le test et éviter d'avoir des erreurs inutiles si on veut récupérer une erreur.

V-D - bool send(const unsigned char* data, unsigned int len);

Ici il y a une seule chose à faire : rediriger l'appel vers le gestionnaire d'envoi :

```
1. bool ClientImpl::send(const unsigned char* data, unsigned int len)
2. {
3.     return mSendingHandler.send(data, len);
4. }
```

V-E - std::unique_ptr<Messages::Base> poll();

Enfin, le cœur du client, qui s'assure du bon fonctionnement des différents modules et changement d'état correspondant quand nécessaire.

```

1. std::unique_ptr<Messages::Base> ClientImpl::poll()
2. {
3.     switch (mState)
4.     {
5.         case State::Connecting:
6.         {
7.             auto msg = mConnectionHandler.poll();
8.             if (msg)
9.             {
10.                if (msg->result == Messages::Connection::Result::Success)
11.                {
12.                    mSendingHandler.init(mSocket);
13.                    mReceivingHandler.init(mSocket);
14.                    mState = State::Connected;
15.                }
16.            }
17.            else
18.            {
19.                disconnect();
20.            }
21.            return msg;
22.        } break;
23.        case State::Connected:
24.        {
25.            mSendingHandler.update();
26.            auto msg = mReceivingHandler.recv();
27.            if (msg && msg->is<Messages::Disconnection>())
28.            {
29.                disconnect();
30.            }
31.            return msg;
32.        } break;
33.        case State::Disconnected:
34.        {
35.        } break;
36.    }
37.    return nullptr;
38. }

```

Tant que la connexion est en cours, seul le gestionnaire de connexion est utilisé. Nous nous contentons d'appeler sa méthode `poll` pour récupérer le résultat de la connexion dès qu'il est disponible.

Si la connexion est réussie, nous initialisons le gestionnaire d'envoi et de réception et changeons l'état à `Connected`. Sinon on déconnecte et réinitialise le client.

Une fois le client connecté, nous mettons à jour le gestionnaire d'envoi pour traiter la file d'envoi, puis nous récupérons un éventuel message reçu. S'il s'agit d'un message de déconnexion, nous réinitialisons le client. Dans tous les cas, ce message est retourné au programme principal.

VI - Utilisation de Client dans le programme

Nous avons désormais accès à une écriture du code bien plus agréable et simple.

Voici un programme minimal utilisant cette nouvelle implémentation créée lors de ce chapitre :

Programme utilisant le nouveau Client

```

1. #include "Sockets.hpp"
2. #include "TCP/Client.hpp"
3. #include "Messages.hpp"
4. #include "Errors.hpp"
5.
6. #include <iostream>
7.
8. int main()
9. {

```


Programme utilisant le nouveau Client

```

10. if (!Network::Start())
11. {
12.     std::cout << "Error starting sockets : " << Network::Errors::Get() << std::endl;
13.     return -1;
14. }
15.
16. Network::TCP::Client client;
17. int port;
18. std::cout << "Port du serveur ? ";
19. std::cin >> port;
20. if (!client.connect("127.0.0.1", port))
21. {
22.     std::cout << "Impossible de se connecter au serveur [127.0.0.1:" << port << "] : "
23.     << Network::Errors::Get() << std::endl;
24. }
25. else
26. {
27.     while (1)
28.     {
29.         while (auto msg = client.poll())
30.         {
31.             if (msg->is<Network::Messages::Connection>())
32.             {
33.                 auto connection = msg->as<Network::Messages::Connection>();
34.                 if (connection->result == Network::Messages::Connection::Result::Success)
35.                 {
36.                     std::cin.ignore();
37.                     std::cout << "Connecte!" << std::endl;
38.                     std::cout << "Entrez une phrase >";
39.                     std::string phrase;
40.                     std::getline(std::cin, phrase);
41.                     if
42.                     (!client.send(reinterpret_cast<const unsigned char*>(phrase.c_str()), static_cast<unsigned int>(phrase.length())))
43.                     {
44.                         std::cout << "Erreur envoi : " << Network::Errors::Get() << std::endl;
45.                         break;
46.                     }
47.                 }
48.                 else
49.                 {
50.                     std::cout << "Connexion echoue : " << static_cast<int>(connection->
51.                     result) << std::endl;
52.                     break;
53.                 }
54.             }
55.             else if (msg->is<Network::Messages::UserData>())
56.             {
57.                 auto userdata = msg->as<Network::Messages::UserData>();
58.                 std::string reply(reinterpret_cast<const char*>(userdata->data.data()), userdata->
59.                 data.size());
60.                 std::cout << "Reponse du serveur : " << reply << std::endl;
61.                 std::cout << ">";
62.                 std::string phrase;
63.                 std::getline(std::cin, phrase);
64.                 if
65.                 (!client.send(reinterpret_cast<const unsigned char*>(phrase.c_str()), static_cast<unsigned int>(phrase.length())))
66.                 {
67.                     std::cout << "Erreur envoi : " << Network::Errors::Get() << std::endl;
68.                     break;
69.                 }
70.             }
71.             else if (msg->is<Network::Messages::Disconnection>())
72.             {
73.                 auto disconnection = msg->as<Network::Messages::Disconnection>();
74.                 std::cout << "Deconnecte : " << static_cast<int>(disconnection->reason) << std::endl;
75.                 break;
76.             }
77.         }
78.     }
79. }
80. }

```

Programme utilisant le nouveau Client

```
75. Network::Release();  
76. return 0;  
77. }
```

Vous pouvez réutiliser le serveur du **chapitre 3** qui utilise déjà ce protocole en interne (avec une implémentation très différente) afin de tester ce code.

Le `while(1)` de la ligne 26 représente la boucle principale du programme, ou votre boucle de gameplay.

Article précédent[<< TCP - Mode non bloquant pour le client](#)[TCP - Un premier serveur : miniserveur](#)

Cours programmation réseau en C++

TCP - Un premier serveur : miniserveur

Par [Bousk](#)

Date de publication : 22 août 2017

Dernière mise à jour : 19 août 2017

DÉBUTANT

Voyons une implémentation d'un premier miniserveur et les évolutions de code nécessaires pour y parvenir.

Commentez

I - De quel serveur s'agit-il ?.....	3
II - Représentation des clients sur le serveur.....	3
II-A - Initialisation depuis le serveur.....	3
III - Interface du serveur.....	4
III-A - Pourquoi une fonction update ?.....	5
IV - Implémentation de la classe Server.....	5
IV-A - Liste de clients connectés.....	5
IV-B - bool start(unsigned short _port);.....	5
IV-B-1 - Adresse réutilisable ?.....	6
IV-B-1-a - Unix - int setsockopt(int sckt, int level, int optname, const void* optval, socklen_t optlen);.....	6
IV-B-1-b - Windows - int setsockopt(SOCKET sckt, int level, int optname, const char* optval, int optlen);.....	7
IV-B-1-b-1 - Doit-on définir un socket comme réutilisable sur Windows ?.....	7
IV-C - void stop();.....	7
IV-D - void update();.....	7
IV-E - std::unique_ptr<Messages::Base> poll();.....	8
V - Qui est l'expéditeur de ce message ?.....	9
V-A - Identifiant de client.....	9
V-A-1 - uint64_t id() const ;.....	9
V-B - Mise à jour des Messages.....	9
V-C - Coordonnées du client.....	10
V-C-1 - Coordonnées du serveur.....	11
V-C-1-a - Récupérer le port d'une adresse.....	12
V-D - Ajouter les coordonnées dans le message pour le serveur.....	12
V-E - Comment joindre un client ?.....	13
V-E-1 - Mise à jour de la collection de clients.....	13
V-E-2 - Envoyer des données à un client spécifique.....	13
VI - Code final du serveur.....	14
VI-A - Server.hpp.....	14
VI-B - Server.cpp.....	15
VI-C - Exemple d'utilisation.....	17
VI-D - Code du client mis à jour.....	18
VI-D-1 - Client.hpp.....	18
VI-D-2 - Client.cpp.....	19

I - De quel serveur s'agit-il ?

Il s'agira d'un « miniserveur » : un serveur destiné à gérer un faible nombre de connexions, de l'ordre de la dizaine ou centaine - mais cela peut varier selon le matériel et la puissance possédés.

Le serveur tournera sur une machine cliente, un joueur fera office de serveur, et ne sera pas exécuté sur une machine distante dédiée. Il devra donc pouvoir tourner en parallèle du programme principal et ne devra pas bloquer celui-ci. Il s'agit du cas où un joueur fait office d'hôte.

Nous utiliserons le mode non bloquant pour y parvenir.

Si le besoin devient de déplacer le programme serveur sur sa propre machine dédiée, aucun changement ne sera nécessaire - tant que son but (accueillir une centaine de joueurs) ne change pas. Vous devriez être capable d'extraire le code spécifique au serveur dans un programme à part et supprimer l'affichage (par exemple) inutile dans ce cas.

Si le serveur devient plus gros et doit accueillir une population plus importante, l'architecture mise en place dans ce chapitre ne sera pas adaptée.

II - Représentation des clients sur le serveur

Quelle est la différence entre un client - la machine cliente - tel que nous l'avons vu et **mis en place dans le chapitre précédent** et un client - la représentation locale de la machine cliente sur un serveur - ? Quasiment aucune !

Souvenez-vous que dans le chapitre traitant de **l'envoi et réception des données depuis le serveur**, nous faisons appel aux mêmes fonctions `recv` et `send` pour la réception et l'envoi. Donc l'échange de données est identique.

La seule différence est que notre *client serveur* (appelons ainsi le socket qui représente un client sur le serveur) ne va pas créer la connexion entre le client et le serveur, mais plutôt **accepter la connexion entrante** sur le serveur, via un appel à `accept` du serveur (**voir chapitre 4**) et non `socket` pour créer le socket suivi de `connect` (**voir chapitre 1**). C'est donc uniquement l'initialisation du socket qui diffère.

Nous utiliserons donc logiquement la même classe `Client`, en modifiant comme nécessaire son fonctionnement interne et son interface afin qu'on puisse l'utiliser indifféremment comme client - comme code client sur une machine cliente - et client serveur - représentation du client sur une machine serveur.

II-A - Initialisation depuis le serveur

L'initialisation sera en fait bien plus simple : `accept` retourne directement un socket valide qu'il suffira donc de passer à notre classe `Client` afin qu'elle l'utilise.

Ajoutons donc simplement une fonction `init` à cette fin que l'on implémentera ainsi :

```
1. bool ClientImpl::init(SOCKET&& sckt)
2. {
3.     assert(sckt != INVALID_SOCKET);
4.     if (sckt == INVALID_SOCKET)
5.         return false;
6.
7.     assert(mState == State::Disconnected);
8.     assert(mSocket == INVALID_SOCKET);
9.     if (mSocket != INVALID_SOCKET)
10.        disconnect();
11. }
```

```

12. mSocket = sckt;
13. if (!SetNonBlocking(mSocket))
14. {
15.     disconnect();
16.     return false;
17. }
18. mSendingHandler.init(mSocket);
19. mReceivingHandler.init(mSocket);
20. mState = State::Connected;
21. return true;
22. }

```

Puisque notre socket ainsi initialisé est déjà connecté, il suffit de le passer en mode non bloquant uniquement - c'est le mode nécessaire à l'utilisation de notre classe `Client` suite au [chapitre précédent](#). Si l'initialisation du mode non bloquant est réussie, alors on passe notre client directement en état connecté et initialisons la réception et l'envoi de données, sinon on déconnecte le client.



Le socket en paramètre pourrait être passé par copie, mais le passer par déplacement permet de renforcer l'idée que la classe prend possession du socket lors de l'appel à cette fonction.

III - Interface du serveur

Il est maintenant temps d'avoir une classe `Server` à utiliser dans un programme principal.

L'utilisation retenue sera la même que pour notre classe `Client` : initialisation et extraction des messages via appel à `poll`.

Notre interface devrait donc ressembler à :

```

1. class Server
2. {
3.     public:
4.         Server();
5.         Server(const Server&) = delete;
6.         Server& operator=(const Server&) = delete;
7.         Server(Server&);
8.         Server& operator=(Server&);
9.         ~Server();
10.
11.         bool start(unsigned short _port);
12.         void stop();
13.         void update();
14.         std::unique_ptr<Messages::Base> poll();
15.
16.     private:
17.         std::unique_ptr<ServerImpl> mImpl;
18. };

```

Tout comme notre `Client`, un `Server` sera déplaçable, mais non copiable. Le pattern *pimpl* sera réutilisé puisqu'il est toujours un bon candidat pour cet exercice.

Pour le démarrer, nous utiliserons la méthode `start` en passant en paramètre le port souhaité. La fonction `stop` servira à arrêter le serveur et déconnecter tous les clients.

`update` servira à mettre à jour le serveur, l'état de chaque client connecté et enfin `poll` retournera le message suivant à traiter.



Le serveur sera accessible uniquement depuis une connexion IPv4 pour le moment. Le port demandé sera initialisé avec un socket IPv4.

L'utilisation d'IPv6 sera étudiée lors d'un prochain chapitre dans lequel le code Client et Server sera mis à jour.

III-A - Pourquoi une fonction update ?

Notre serveur aura une liste de clients connectés. Que se passerait-il si nous n'avions qu'une fonction `poll` pour récupérer les données des clients ? Le serveur itérerait sur sa collection de clients, et retournerait le premier message complet du premier client.

Avec une simple collection type `vector`, l'itération serait toujours dans le même ordre, donc notre serveur retournerait les messages des premiers clients en priorité. Potentiellement le dernier client arrivé n'aurait jamais aucun message traité par le serveur.

Avec l'ajout d'une fonction `update`, on s'assure de gérer tous les clients, puis de mettre en file les messages reçus lors de ce traitement. Ainsi, sans être parfaitement égaux dans la file, la priorité des premiers clients connectés est moindre sur les derniers. Une autre astuce est de limiter à un message par client à chaque appel à `update` par exemple.

Ceci est préférable à avoir une collection plus intelligente qui garderait en mémoire où l'itération s'est arrêtée la dernière fois afin de traiter le client suivant : si vous avez un grand nombre de clients, ou particulièrement rapides, vous pourriez recevoir un nouveau message du premier avant d'avoir traité le dernier, l'appel à `poll` retournerait alors toujours un message valide et vous ne sortiriez jamais de la boucle d'appel à `poll` dans votre programme principal.

On aura un problème similaire sur l'acceptation de nouveaux clients : si notre serveur se fait spammer de connexions, il passera son temps à accepter des clients sans traiter les existants. De la même manière, nous pouvons limiter, par exemple, à dix nouveaux clients par appel à `update`.

L'établissement d'une connexion est un processus réputé relativement lent, donc retarder le traitement des connexions entrantes de quelques millisecondes est acceptable et n'entraînera pas la grogne des utilisateurs. Par contre une fois connectés, ils sont en général plutôt impatients de pouvoir communiquer avec le serveur.

IV - Implémentation de la classe Server

IV-A - Liste de clients connectés

Les clients connectés seront sauvegardés dans une collection type `std::vector` pour le moment. Dans les codes suivants, `mClients` est défini comme `std::vector<Client> mClients`; où `Client` est la classe `Client` introduite dans le chapitre précédent.

IV-B - `bool start(unsigned short _port);`

L'appel à `start` déclenchera l'initialisation du serveur telle qu'on la connaît depuis le chapitre 5.

Son implémentation sera donc juste une adaptation de ce que l'on avait alors vu, en ajoutant l'étape pour rendre le socket non bloquant :

```
1. bool ServerImpl::start(unsigned short _port)
2. {
3.     assert(mSocket == INVALID_SOCKET);
4.     if (mSocket != INVALID_SOCKET)
5.         stop();
6.     mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
7.     if (mSocket == INVALID_SOCKET)
8.         return false;
```

```
9.
10. if (!SetReuseAddr(mSocket) || !SetNonBlocking(mSocket))
11. {
12.     stop();
13.     return false;
14. }
15.
16. sockaddr_in addr;
17. addr.sin_addr.s_addr = INADDR_ANY;
18. addr.sin_port = htons(_port);
19. addr.sin_family = AF_INET;
20. if (bind(mSocket, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) != 0)
21. {
22.     stop();
23.     return false;
24. }
25. if (listen(mSocket, SOMAXCONN) != 0)
26. {
27.     stop();
28.     return false;
29. }
30. return true;
31. }
```

IV-B-1 - Adresse réutilisable ?

Un serveur utilisera généralement un port spécifique sur l'ensemble des adresses disponibles. Il n'est pas rare également que la machine serveur ait des mécanismes au niveau du système pour relancer automatiquement le programme en cas de problème, erreur ou crash, type cron sur Unix ou les services Windows.

En cas d'arrêt intempestif du programme serveur, à fortiori si des clients étaient toujours connectés, il est fort à parier que des échanges étaient toujours en cours. Afin de terminer proprement les connexions, le socket TCP sera maintenu par le système dans un état nommé *TIME_WAIT*. Le socket dans cet état attendra que les derniers échanges se terminent afin que sa connexion soit terminée correctement du point de vue du protocole TCP, ou qu'il arrive à expiration (timeout), généralement après deux minutes.

Si l'on essaye de rouvrir un socket sur ce port, ce qui arrive généralement plus rapidement que les deux minutes d'expiration, à fortiori si c'est le système qui le gère, le système va retourner une erreur *EADDRINUSE* ou *EACCESS*, respectivement adresse déjà utilisée ou erreur d'accès, tant que notre socket fantôme est dans cet état, empêchant un nouveau socket d'être créé à cette adresse et utilisant le port souhaité.

Pour contrer ça, il existe le flag *SO_REUSEADDR* qui permet entre autres d'indiquer au système qu'il est autorisé à réutiliser une adresse et un port d'un socket dans l'état *TIME_WAIT*.

Pour des informations plus en détail et spécificités de chaque plateforme, voici [un excellent post sur StackOverflow](#).

IV-B-1-a - Unix - int setsockopt(int sckt, int level, int optname, const void* optval, socklen_t optlen);

Permet d'effectuer une opération sur un socket. Il s'agit de l'inverse de *getsockopt* que nous avons utilisé pour récupérer l'erreur spécifique d'un socket dans [le chapitre 5](#).

- *sckt* est le socket sur lequel agir.
- *level* est le niveau relatif à l'opération que nous voulons effectuer. Pour un socket, il s'agira toujours de *SOL_SOCKET*.
- *optname* est le nom de l'opération à effectuer. Dans notre cas présent *SO_REUSEADDR*.
- *optval* est un pointeur vers la valeur à appliquer à l'opération.
- *optlen* la taille de la valeur pointée par *optval*.


```
1. int optval = 1;
2. setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
```

Retourne -1 en cas d'échec, 0 sinon.

IV-B-1-b - Windows - `int setsockopt(SOCKET sckt, int level, int optname, const char* optval, int optlen);`

La signature change à peine, `optval` est passé en tant que `const char*` plutôt que `const void*` et `optlen` est un `int` et non un `socklen_t`.

Ce qui permettrait d'écrire ceci avec notre code actuel pour avoir un appel portable :

```
1. int optval = 1;
2. setsockopt(socket, SOL_SOCKET,
   SO_REUSEADDR, reinterpret_cast<const char*>(&optval), sizeof(optval));
```

Retourne `SOCKET_ERROR` en cas d'erreur, 0 sinon.

IV-B-1-b-1 - Doit-on définir un socket comme réutilisable sur Windows ?

Cette option n'a que peu d'intérêt sur Windows, dû à l'implémentation de celle-ci par le système.

En effet, Windows définit `SO_REUSEADDR` comme permettant au socket de s'associer (`bind`) de force à un port déjà ouvert par un autre socket. Un programme peut donc usurper l'utilisation d'un port sur une adresse locale, ou toutes les adresses locales. Puis on tombe dans un indéterminisme des plus complets : vous avez deux sockets qui utilisent le même port et travaillent sur la même adresse, les données vont aller à l'un ou l'autre sans aucune garantie et aléatoirement - mais toujours à l'un ou l'autre - sans que vous puissiez rien y faire.

L'aspect usurpation de socket a depuis été rectifié en ajoutant l'option `SO_EXCLUSIVEADDRUSE`, qui permet d'empêcher toute autre ouverture du même port. Rien qui ne nous intéresse dans notre cas pour rouvrir un socket sur un port qui serait dans l'état `TIME_WAIT` suite au crash de notre serveur. Rien d'utile à notre niveau donc.

Lire les sujets sur la MSDN à ce sujet [ici](#) et [ici](#).

IV-C - `void stop();`

`stop` aura pour effet de déconnecter les clients existants et libérer le socket du serveur.

```
1. void ServerImpl::stop()
2. {
3.     for (auto& client : mClients)
4.         client.disconnect();
5.     mClients.clear();
6.     if (mSocket != INVALID_SOCKET)
7.         CloseSocket(mSocket);
8.     mSocket = INVALID_SOCKET;
9. }
```

IV-D - `void update();`

Notre fonction aura deux objectifs : accepter les nouvelles connexions et gérer les clients existants.

Afin de limiter l'inégalité des clients face à leur ordre de connexion, nous limiterons la réception des messages à un par client.

De même, afin de limiter l'éventuel spam de connexions et continuer de traiter les clients déjà connectés, nous limiterons à dix les nouveaux clients par appel à update.

Il suffit de dérouler cet algorithme pour obtenir une implémentation :

```

1. void ServerImpl::update()
2. {
3.     if (mSocket == INVALID_SOCKET)
4.         return;
5.
6.     //!< accept jusqu'à 10 nouveaux clients
7.     for (int accepted = 0; accepted < 10; ++accepted)
8.     {
9.         sockaddr_in addr = { 0 };
10.        socklen_t addrlen = sizeof(addr);
11.        SOCKET newClientSocket = accept(mSocket, reinterpret_cast<sockaddr*>(&addr), &addrlen);
12.        if (newClientSocket == INVALID_SOCKET)
13.            break;
14.        Client newClient;
15.        if (newClient.init(std::move(newClientSocket)))
16.        {
17.            auto
18.            message = std::make_unique<Messages::Connection>(Messages::Connection::Result::Success);
19.            mMessages.push_back(std::move(message));
20.            mClients.push_back(std::move(newClient));
21.        }
22.
23.        //!< mise à jour des clients connectés
24.        //!< réceptionne au plus 1 message par client
25.        //!< supprime de la liste les clients déconnectés
26.        for (auto itClient = mClients.begin(); itClient != mClients.end(); )
27.        {
28.            auto& client = *itClient;
29.            auto msg = client.poll();
30.            if (msg)
31.            {
32.                if (msg->is<Messages::Disconnection>())
33.                    itClient = mClients.erase(itClient);
34.                else
35.                    ++itClient;
36.                mMessages.push_back(std::move(msg));
37.            }
38.            else
39.                ++itClient;
40.        }
41.    }

```

N'oubliez pas de générer un **Message::Connection** lorsqu'un nouveau client est accepté par le serveur pour que l'application puisse avoir l'information.

IV-E - std::unique_ptr<Messages::Base> poll();

Ici poll sera des plus simples à implémenter. Puisque update se charge de mettre en file les messages, poll n'aura qu'à les dépiler afin de les retourner à l'application.

```

1. std::unique_ptr<Messages::Base> ServerImpl::poll()
2. {
3.     if (mMessages.empty())
4.         return nullptr;
5.
6.     auto msg = std::move(mMessages.front());
7.     mMessages.pop_front();
8.     return msg;
9. }

```

Dans cette implémentation, les fonctions `update` et `poll` sont supposées appelées depuis le même thread.



Si vous souhaitez appeler `update` et `poll` depuis différents threads, il vous faudra synchroniser l'accès à `mMessages` à l'aide de `mutex` par exemple.

V - Qui est l'expéditeur de ce message ?

Vous vous serez sans doute déjà fait la réflexion : j'extrais un message via `poll`, mais je n'ai aucune idée de qui l'a envoyé !

En effet, notre hiérarchie de messages était utilisée jusque-là par le client uniquement, qui ne pouvait recevoir un message que du serveur auquel il était connecté. Mais notre serveur peut recevoir des messages de plusieurs clients, il faut donc ajouter quelques informations supplémentaires quant à l'origine du message reçu.

V-A - Identifiant de client

L'idée est d'ajouter un identifiant unique pour chaque client.

La bonne nouvelle c'est que nous avons déjà un tel identifiant à disposition, généré par le système et assuré unique pour chaque connexion : le socket !

Sur Unix, il s'agit d'un `int`, généralement ayant une valeur plutôt faible qui plus est.

Sur Windows c'est un type plus opaque, un entier non signé, mais qui sur toutes les versions jusque-là se retrouve être un `typedef` sur un entier non signé 64 bits ou plus petit.

Pour couvrir ces deux cas, notre identifiant sera un `uint64_t`.

V-A-1 - `uint64_t id() const ;`

Ajoutons donc cette très simple fonction à notre interface Client :

```
class ClientImpl
{
...
    uint64_t id() const { return static_cast<uint64_t>(mSocket); }
...
};

uint64_t Client::id() const { return mImpl ? mImpl->id() : (uint64_t)(-1); }
```

Notez que tous les clients non initialisés partageront l'identifiant -1. Heureusement un client n'est pas censé être utilisé non initialisé très longtemps vous ne devrez donc avoir aucun problème à cause de cela.

Si un client est déplacé, son identifiant sera également la valeur non initialisée suite au déplacement. Mais ici encore, vous ne devriez pas avoir à traiter des clients déplacés très longtemps. Une fois déplacé, vous aurez vite fait de vous en débarrasser et le libérer.

V-B - Mise à jour des Messages

Il faut maintenant mettre à jour nos Messages pour ajouter cet identifiant.

Pas besoin de chercher compliqué pour le moment, et ajoutons juste un champ à la structure de Base `uint64_t idFrom`;

```

1. namespace Network
2. {
3.     namespace Messages
4.     {
5.         class Base
6.         {
7.         public:
8.             template<class M>
9.             bool is() const { return mType == M::StaticType; }
10.            template<class M>
11.            const M* as() const { return static_cast<const M*>(this); }
12.
13.            uint64_t idFrom;
14.
15.        protected:
16.            enum class Type {
17.                Connection,
18.                Disconnection,
19.                UserData,
20.            };
21.            Base(Type type)
22.                : mType(type)
23.            {}
24.        private:
25.            Type mType;
26.        };
27.    }
28. }
```

Le mettre dans la portée publique simplifie les modifications à apporter au reste de la bibliothèque. Si vous préférez l'inclure dans le constructeur et le mettre en privé, il vous faudra modifier tous les constructeurs et leur appel, puis penser à mettre un accesseur public.

Ensuite modifiez les générations de messages afin d'ajouter l'identifiant du client correspondant à chaque fois où nécessaire.

Ou bien vous pouvez l'ajouter à un seul emplacement : dans le poll du serveur. Après tout il n'est pas vraiment utile d'ajouter l'identifiant du client si on utilise directement le client : le message vient forcément du serveur auquel il est connecté.

V-C - Coordonnées du client

Parmi les informations utiles que nous n'avons pas encore retrouvées, se trouvent l'adresse IP et le port du client.

Cette information est récupérée lors de l'appel à `accept`, mais directement oubliée puisque nous n'initialisons le client qu'avec le socket.

Mettons donc à jour notre fonction `init` pour également passer l'information d'adresse du client :

```

1. class ClientImpl
2. {
3.     ...
4.     bool init(SOCKET&& sckt, const sockaddr_in& addr);
5.     ...
6.     const sockaddr_in& destinationAddress() const { return mAddress; }
7.     ...
8.     sockaddr_in mAddress{ 0 };
9. };
10. bool ClientImpl::init(SOCKET&& sckt, const sockaddr_in& addr)
11. {
12.     assert(sckt != INVALID_SOCKET);
```

```

13. if (sckt == INVALID_SOCKET)
14.     return false;
15.
16. mSocket = sckt;
17. mAddress = addr;
18. if (!SetNonBlocking(mSocket))
19. {
20.     disconnect();
21.     return false;
22. }
23. onConnected();
24. return true;
25. }

```

N'oubliez pas de réinitialiser l'adresse dans la fonction disconnect avec un `memset(&mAddress, 0, sizeof(mAddress))`.

V-C-1 - Coordonnées du serveur

Afin de garder une utilisation cohérente lorsque notre classe Client est utilisée en tant que client de connexion à un serveur, utilisons cette variable `mAddress` pour sauvegarder les informations du serveur auquel le client est connecté. `destinationAddress()` retournera ainsi toujours l'adresse de destination du client : du client derrière le client serveur dans le cadre d'un serveur, et du serveur auquel est connecté le client dans le cadre d'un client.

Comme il s'agit de l'adresse à laquelle le client se connecte, le code qui génère cette information se retrouvera dans le *ConnectionHandler* :

```

1. class ConnectionHandler
2. {
3.     ...
4.     const sockaddr_in& getConnectedAddress() const { return mConnectedAddress; }
5.     ...
6.     sockaddr_in mConnectedAddress;
7. };
8. bool ConnectionHandler::connect(SOCKET sckt, const std::string& address, unsigned short port)
9. {
10.    assert(sckt != INVALID_SOCKET);
11.    mAddress = address;
12.    mPort = port;
13.    mFd.fd = sckt;
14.    mFd.events = POLLOUT;
15.    inet_pton(AF_INET, mAddress.c_str(), &mConnectedAddress.sin_addr.s_addr);
16.    mConnectedAddress.sin_family = AF_INET;
17.    mConnectedAddress.sin_port = htons(mPort);
18.    if (::connect(sckt, (const sockaddr*)&mConnectedAddress, sizeof(mConnectedAddress)) != 0)
19.    {
20.        int err = Errors::Get();
21.        if (err != Errors::INPROGRESS && err != Errors::WOULDBLOCK)
22.            return false;
23.    }
24.    return true;
25. }

```

Dans `connect`, nous avons juste remplacé le `sockaddr_in server` auparavant local par la variable membre `mConnectedAddress`.

L'accesseur est présent pour pouvoir initialiser `ClientImpl::mAddress` lorsque la connexion est réussie. J'opte pour factoriser les initialisations qui suivent une connexion réussie, que ce soit depuis `init` ou le gestionnaire de connexion :

```

1. bool ClientImpl::init(SOCKET&& sckt, const sockaddr_in& addr)
2. {
3.    assert(sckt != INVALID_SOCKET);
4.    if (sckt == INVALID_SOCKET)
5.        return false;

```

```

6.
7.  mSocket = sckt;
8.  if (!SetNonBlocking(mSocket))
9.  {
10.   disconnect();
11.   return false;
12. }
13. onConnected(addr);
14. return true;
15. }
16.
17. std::unique_ptr<Messages::Base> ClientImpl::poll()
18. {
19.   switch (mState)
20.   {
21.     case State::Connecting:
22.     {
23.       auto msg = mConnectionHandler.poll();
24.       if (msg)
25.       {
26.         if (msg->result == Messages::Connection::Result::Success)
27.         {
28.           onConnected(mConnectionHandler.getConnectedAddress());
29.         }
30.         else
31.         {
32.           disconnect();
33.         }
34.       }
35.       return msg;
36.     } break;
37.     ...
38.   }
39. }
40.
41. void ClientImpl::onConnected(const sockaddr_in& addr)
42. {
43.   mAddress = addr;
44.   mSendingHandler.init(mSocket);
45.   mReceivingHandler.init(mSocket);
46.   mState = State::Connected;
47. }

```

V-C-1-a - Récupérer le port d'une adresse

Nous avons déjà une fonction `GetAddress` qui retourne l'adresse IP depuis une structure `sockaddr_in`.

Voici une fonction pour récupérer le port depuis une telle structure, que vous aurez facilement devinée puisqu'elle est le miroir de l'initialisation du `sockaddr_in` pour l'appel à `connect` :

```

1. unsigned short GetPort(const sockaddr_in& addr)
2. {
3.   return ntohs(addr.sin_port);
4. }

```

V-D - Ajouter les coordonnées dans le message pour le serveur

Dans les précédents chapitres, nous affichions toujours les informations de connexion du client sur le serveur à chaque réception de message.

Avec l'identifiant seul, il faudrait ajouter une interface pour récupérer un client ou ses informations à partir de l'identifiant introduit plus haut. Ce qui n'est pas très élégant et source de problème si notre code n'est plus monothreadé.

Au lieu de modifier l'interface du serveur, ajoutons cette information à nos messages, tout comme nous avons déjà ajouté l'identifiant :

```
1. class Base
2. {
3.     ...
4.     sockaddr_in from;
5.     ...
6. };
```

Tout comme pour idFrom, il sera dans la portée publique et utilisée uniquement dans le cadre d'un serveur. Ce sera le serveur qui mettra à jour ce champ lors de la réception de données d'un client :

```
1. for (auto itClient = mClients.begin(); itClient != mClients.end(); )
2. {
3.     auto& client = *itClient;
4.     auto msg = client.poll();
5.     if (msg)
6.     {
7.         msg->from = client.destinationAddress();
8.         msg->idFrom = client.id();
9.         if (msg->is<Messages::Disconnection>())
10.        {
11.            itClient = mClients.erase(itClient);
12.        }
13.        else
14.            ++itClient;
15.        mMessages.push_back(std::move(msg));
16.    }
17.    else
18.        ++itClient;
19. }
```

V-E - Comment joindre un client ?

Notre serveur est capable de réceptionner les données de nos clients, mais pas encore de leur envoyer quoi que ce soit.

Mais nous avons désormais un identifiant pour nos clients.

V-E-1 - Mise à jour de la collection de clients

Puisque nous avons un identifiant unique et allons être amenés à retrouver un client dans notre collection, le seul vector est désormais remplacé par une map. Il sera ainsi plus simple de retrouver un client par son identifiant.

Désormais la définition de mClients est `std::map<uint64_t, Client> mClients;`. Le code à jour des parties implémentées plus haut est disponible en fin de chapitre.

V-E-2 - Envoyer des données à un client spécifique

Ajoutons donc deux méthodes pour envoyer des données à nos clients : l'une d'elles pour envoyer des données à un client précis, selon son identifiant, l'autre pour envoyer une même donnée à tous nos clients connectés - une opération toujours pratique dans le cadre d'un serveur :

```
1. bool sendTo(uint64_t clientid, const unsigned char* data, unsigned int len);
2. bool sendToAll(const unsigned char* data, unsigned int len);
3.
4. bool ServerImpl::sendTo(uint64_t clientid, const unsigned char* data, unsigned int len)
5. {
6.     auto itClient = mClients.find(clientid);
```

```
7. return itClient != mClients.end() && itClient->second.send(data, len);
8. }
9. bool ServerImpl::sendToAll(const unsigned char* data, unsigned int len)
10. {
11.     bool ret = true;
12.     for (auto& client : mClients)
13.         ret &= client.second.send(data, len);
14.     return ret;
15. }
```

`sendTo` sera utilisé pour envoyer des données à un client spécifique selon son identifiant et retournera `true` ou `false` selon que les données ont été mises en file d'envois avec succès ou non.

`sendToAll` tentera de mettre en file d'envois un même ensemble de données à tous les clients connus du serveur. Si la valeur de retour vaut `true`, tous les clients connus au moment de l'appel ont ces données en file d'envois. Sinon, au moins un client n'a pas pu les mettre en file et ne les recevra donc jamais.

Ici nous n'aurons aucune information sur quel(s) client(s) ne recevra(ont) pas les données en question. Si cette information vous importe, vous pourrez par exemple retourner un `vector` d'identifiants des clients qui ont échoué.

En pratique, puisque nous sommes en TCP, il n'y a aucune raison que les données ne soient pas mises en file d'envois, et donc reçues du moment que le client reste connecté assez longtemps pour les recevoir. En fait l'unique raison est que l'appel à notre `Client::send` retourne `false`, ce qui peut actuellement arriver pour deux raisons :

- le client n'est pas ou plus implémenté, après un déplacement par exemple et auquel cas ce client ne devrait plus être dans notre liste et sera certainement purgé prochainement ;
- `SendingHandler::send` retourne `false`, ce qui dans notre implémentation actuelle n'est possible que si la taille des données est trop élevée, et échouera donc pour tous les clients.

Vous aurez remarqué l'utilisation de `mClients`. Ces fonctions doivent donc être appelées sur le même thread que `update`.



Sinon, il vous faudra synchroniser `mClients` à l'aide d'un mutex par exemple.

VI - Code final du serveur

Nous avons maintenant un serveur fonctionnel prêt à accueillir nos utilisateurs.

VI-A - Server.hpp

TCP/Server.hpp

```
1. #pragma once
2.
3. #include <memory>
4.
5. namespace Network
6. {
7.     namespace Messages {
8.         class Base;
9.     }
10.    namespace TCP
11.    {
12.        class ServerImpl;
13.        class Server
14.        {
15.        public:
16.            Server();
17.            Server(const Server&) = delete;
```


TCP/Server.hpp

```

18.     Server& operator=(const Server&) = delete;
19.     Server(Server&&);
20.     Server& operator=(Server&&);
21.     ~Server();
22.
23.     bool start(unsigned short _port);
24.     void stop();
25.     void update();
26.     bool sendTo(uint64_t clientid, const unsigned char* data, unsigned int len);
27.     bool sendToAll(const unsigned char* data, unsigned int len);
28.     std::unique_ptr<Messages::Base> poll();
29.
30. private:
31.     std::unique_ptr<ServerImpl> mImpl;
32. };
33. }
34. }
```

VI-B - Server.cpp

TCP/Server.cpp

```

1. #include "Server.hpp"
2.
3. #include "Sockets.hpp"
4. #include "TCP/Client.hpp"
5. #include "Messages.hpp"
6.
7. #include <map>
8. #include <list>
9. #include <cassert>
10.
11. namespace Network
12. {
13.     namespace TCP
14.     {
15.         class ServerImpl
16.         {
17.         public:
18.             ServerImpl() = default;
19.             ~ServerImpl();
20.
21.             bool start(unsigned short _port);
22.             void stop();
23.             void update();
24.             bool sendTo(uint64_t clientid, const unsigned char* data, unsigned int len);
25.             bool sendToAll(const unsigned char* data, unsigned int len);
26.             std::unique_ptr<Messages::Base> poll();
27.
28.         private:
29.             std::map<uint64_t, Client> mClients;
30.             std::list<std::unique_ptr<Messages::Base>> mMessages;
31.             SOCKET mSocket{ INVALID_SOCKET };
32.         };
33.         ServerImpl::~ServerImpl()
34.         {
35.             stop();
36.         }
37.
38.         bool ServerImpl::start(unsigned short _port)
39.         {
40.             assert(mSocket == INVALID_SOCKET);
41.             if (mSocket != INVALID_SOCKET)
42.                 stop();
43.             mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
44.             if (mSocket == INVALID_SOCKET)
45.                 return false;
46.
47.             if (!SetReuseAddr(mSocket) || !SetNonBlocking(mSocket))
48.                 {
```

TCP/Server.cpp

```

49.     stop();
50.     return false;
51. }
52.
53. sockaddr_in addr;
54. addr.sin_addr.s_addr = INADDR_ANY;
55. addr.sin_port = htons(_port);
56. addr.sin_family = AF_INET;
57. if (bind(mSocket, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) != 0)
58. {
59.     stop();
60.     return false;
61. }
62. if (listen(mSocket, SOMAXCONN) != 0)
63. {
64.     stop();
65.     return false;
66. }
67. return true;
68. }
69. void ServerImpl::stop()
70. {
71.     for (auto& client : mClients)
72.         client.second.disconnect();
73.     mClients.clear();
74.     if (mSocket != INVALID_SOCKET)
75.         CloseSocket(mSocket);
76.     mSocket = INVALID_SOCKET;
77. }
78. void ServerImpl::update()
79. {
80.     if (mSocket == INVALID_SOCKET)
81.         return;
82.
83.     //!< accept jusqu'à 10 nouveaux clients
84.     for (int accepted = 0; accepted < 10; ++accepted)
85.     {
86.         sockaddr_in addr = { 0 };
87.         socklen_t addrlen = sizeof(addr);
88.         SOCKET newClientSocket = accept(mSocket, reinterpret_cast<sockaddr*>(&addr), &addrlen);
89.         if (newClientSocket == INVALID_SOCKET)
90.             break;
91.         Client newClient;
92.         if (newClient.init(std::move(newClientSocket), addr))
93.         {
94.             auto
95.             message = std::make_unique<Messages::Connection>(Messages::Connection::Result::Success);
96.             message->idFrom = newClient.id();
97.             message->from = newClient.destinationAddress();
98.             mMessages.push_back(std::move(message));
99.             mClients[newClient.id()] = std::move(newClient);
100.        }
101.
102.        //!< mise à jour des clients connectés
103.        //!< réceptionne au plus 1 message par client
104.        //!< supprime de la liste les clients déconnectés
105.        for (auto itClient = mClients.begin(); itClient != mClients.end(); )
106.        {
107.            auto& client = itClient->second;
108.            auto msg = client.poll();
109.            if (msg)
110.            {
111.                msg->from = itClient->second.destinationAddress();
112.                msg->idFrom = itClient->second.id();
113.                if (msg->is<Messages::Disconnection>())
114.                {
115.                    itClient = mClients.erase(itClient);
116.                }
117.                else
118.                    ++itClient;

```

TCP/Server.cpp

```

119.     mMessages.push_back(std::move(msg));
120. }
121. else
122.     ++itClient;
123. }
124. }
125. bool ServerImpl::sendTo(uint64_t clientid, const unsigned char* data, unsigned int len)
126. {
127.     auto itClient = mClients.find(clientid);
128.     return itClient != mClients.end() && itClient->second.send(data, len);
129. }
130. bool ServerImpl::sendToAll(const unsigned char* data, unsigned int len)
131. {
132.     bool ret = true;
133.     for (auto& client : mClients)
134.         ret &= client.second.send(data, len);
135.     return ret;
136. }
137. std::unique_ptr<Messages::Base> ServerImpl::poll()
138. {
139.     if (mMessages.empty())
140.         return nullptr;
141.
142.     auto msg = std::move(mMessages.front());
143.     mMessages.pop_front();
144.     return msg;
145. }
146. //////////////////////////////////////
147. //////////////////////////////////////
148. //////////////////////////////////////
149. //////////////////////////////////////
150. //////////////////////////////////////
151. Server::Server() {}
152. Server::~Server() {}
153. Server::Server(Server&& other)
154.     : mImpl(std::move(other.mImpl))
155. {}
156. Server& Server::operator=(Server&& other)
157. {
158.     mImpl = std::move(other.mImpl);
159.     return *this;
160. }
161. bool Server::start(unsigned short _port)
162. {
163.     if (!mImpl)
164.         mImpl = std::make_unique<ServerImpl>();
165.     return mImpl && mImpl->start(_port);
166. }
167. void Server::stop() { if (mImpl) mImpl->stop(); }
168. void Server::update() { if (mImpl) mImpl->update(); }
169. bool Server::sendTo(uint64_t clientid, const unsigned char* data, unsigned int
len) { return mImpl && mImpl->sendTo(clientid, data, len); }
170. bool Server::sendToAll(const unsigned char* data, unsigned int len) { return mImpl &&
mImpl->sendToAll(data, len); }
171. std::unique_ptr<Messages::Base> Server::poll() { return mImpl ? mImpl->poll() : nullptr; }
172. }
173. }

```

VI-C - Exemple d'utilisation

```

1. #include "Sockets.hpp"
2. #include "TCP/Server.hpp"
3. #include "Messages.hpp"
4. #include "Errors.hpp"
5.
6. #include <iostream>
7.
8. int main()
9. {

```

```

10. if (!Network::Start())
11. {
12.     std::cout << "Erreur initialisation WinSock : " << Network::Errors::Get();
13.     return -1;
14. }
15.
16. unsigned short port;
17. std::cout << "Port ? ";
18. std::cin >> port;
19.
20. Network::TCP::Server server;
21. if (!server.start(port))
22. {
23.     std::cout << "Erreur initialisation serveur : " << Network::Errors::Get();
24.     return -2;
25. }
26.
27. while(1)
28. {
29.     server.update();
30.     while (auto msg = server.poll())
31.     {
32.         if (msg->is<Network::Messages::Connection>())
33.         {
34.             std::cout << "Connexion de [" << Network::GetAddress(msg->from) << ":" << Network::GetPort(msg->from) << "]" << std::endl;
35.         }
36.         else if (msg->is<Network::Messages::Disconnection>())
37.         {
38.             std::cout << "Deconnexion de [" << Network::GetAddress(msg->from) << ":" << Network::GetPort(msg->from) << "]" << std::endl;
39.         }
40.         else if (msg->is<Network::Messages::UserData>())
41.         {
42.             auto userdata = msg->as<Network::Messages::UserData>();
43.             server.sendToAll(userdata->data.data(), static_cast<unsigned int>(userdata->data.size()));
44.         }
45.     }
46. }
47. server.stop();
48. Network::Release();
49. return 0;
50. }

```

Le `while(1)` représente la boucle principale du programme, si vous embarquez le serveur dans le code client avec un joueur qui fait office d'hôte. Si vous optez pour une machine dédiée, il s'agira vraisemblablement du `main` du programme comme dans cet exemple.

Ici le serveur se contente de transférer les données reçues à tous les clients connectés. Un programme plus significatif devrait désérialiser les données reçues, traiter la requête puis retourner un résultat ou non au(x) client(s).

VI-D - Code du client mis à jour

Le code de la classe Client a légèrement été mis à jour dans ce chapitre, voici une copie de l'implémentation actuelle.

VI-D-1 - Client.hpp

TCP/Client.hpp

```

1. #pragma once
2.
3. #include "Sockets.hpp"
4.
5. #include <string>
6. #include <memory>

```

TCP/Client.hpp

```

7.
8. namespace Network
9. {
10. namespace Messages {
11. class Base;
12. }
13. namespace TCP
14. {
15. using HeaderType = uint16_t;
16. static const unsigned int HeaderSize = sizeof(HeaderType);
17. class ClientImpl;
18. class Client
19. {
20. public:
21. Client();
22. Client(const Client&) = delete;
23. Client& operator=(const Client&) = delete;
24. Client(Client&&);
25. Client& operator=(Client&&);
26. ~Client();
27.
28. bool init(SOCKET&& sckt, const sockaddr_in& addr);
29. bool connect(const std::string& ipaddress, unsigned short port);
30. void disconnect();
31. bool send(const unsigned char* data, unsigned int len);
32. std::unique_ptr<Messages::Base> poll();
33.
34. uint64_t id() const;
35. const sockaddr_in& destinationAddress() const;
36.
37. private:
38. std::unique_ptr<ClientImpl> mImpl;
39. };
40. }
41. }
```

VI-D-2 - Client.cpp

TCP/Client.cpp

```

1. #include "TCP/Client.hpp"
2.
3. #include "Sockets.hpp"
4. #include "Messages.hpp"
5. #include "Errors.hpp"
6.
7. #include <vector>
8. #include <list>
9. #include <cassert>
10. #include <numeric>
11.
12. namespace Network
13. {
14. namespace TCP
15. {
16. class ConnectionHandler
17. {
18. public:
19. ConnectionHandler() = default;
20. bool connect(SOCKET sckt, const std::string& address, unsigned short port);
21. std::unique_ptr<Messages::Connection> poll();
22. const sockaddr_in& connectedAddress() const { return mConnectedAddress; }
23.
24. private:
25. pollfd mFd{ 0 };
26. sockaddr_in mConnectedAddress;
27. std::string mAddress;
28. unsigned short mPort;
29. };
```

TCP/Client.cpp

```

30.  bool ConnectionHandler::connect(SOCKET sckt, const std::string& address, unsigned short
    port)
31.  {
32.      assert(sckt != INVALID_SOCKET);
33.      mAddress = address;
34.      mPort = port;
35.      mFd.fd = sckt;
36.      mFd.events = POLLOUT;
37.      inet_pton(AF_INET, mAddress.c_str(), &mConnectedAddress.sin_addr.s_addr);
38.      mConnectedAddress.sin_family = AF_INET;
39.      mConnectedAddress.sin_port = htons(mPort);
40.      if (::connect(sckt, (const sockaddr*)&mConnectedAddress, sizeof(mConnectedAddress)) != 0)
41.      {
42.          int err = Errors::Get();
43.          if (err != Errors::INPROGRESS && err != Errors::WOULDBLOCK)
44.              return false;
45.      }
46.      return true;
47.  }
48.  std::unique_ptr<Messages::Connection> ConnectionHandler::poll()
49.  {
50.      int res = ::poll(&mFd, 1, 0);
51.      if (res < 0)
52.          return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
53.      else if (res > 0)
54.      {
55.          if (mFd.revents & POLLOUT)
56.          {
57.              return std::make_unique<Messages::Connection>(Messages::Connection::Result::Success);
58.          }
59.          else if (mFd.revents & (POLLHUP | POLLNVAL))
60.          {
61.              return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
62.          }
63.          else if (mFd.revents & POLLERR)
64.          {
65.              return std::make_unique<Messages::Connection>(Messages::Connection::Result::Failed);
66.          }
67.      }
68.      //!< action non terminée
69.      return nullptr;
70.  }
71.  //////////////////////////////////////
72.  //////////////////////////////////////
73.  //////////////////////////////////////
74.  //////////////////////////////////////
75.  //////////////////////////////////////
76.  class ReceptionHandler
77.  {
78.      enum class State {
79.          Header,
80.          Data,
81.      };
82.  public:
83.      ReceptionHandler() = default;
84.      void init(SOCKET sckt);
85.      std::unique_ptr<Messages::Base> recv();
86.  private:
87.      inline char* missingDataStartBuffer() { return reinterpret_cast<char*>(mBuffer.data() +
        mReceived); }
88.      inline int missingDataLength() const { return static_cast<int>(mBuffer.size() -
        mReceived); }
89.      void startHeaderReception();
90.      void startDataReception();
91.      void startReception(unsigned int expectedDataLength, State newState);
92.  private:
93.      std::vector<unsigned char> mBuffer;
94.      unsigned int mReceived;
95.      SOCKET mSckt{ INVALID_SOCKET };

```

TCP/Client.cpp

```

98.     State mState;
99. };
100. void ReceptionHandler::init(SOCKET sckt)
101. {
102.     assert(sckt != INVALID_SOCKET);
103.     mSckt = sckt;
104.     startHeaderReception();
105. }
106. void ReceptionHandler::startHeaderReception()
107. {
108.     startReception(HeaderSize, State::Header);
109. }
110. void ReceptionHandler::startDataReception()
111. {
112.     assert(mBuffer.size() == sizeof(HeaderType));
113.     HeaderType networkExpectedDataLength;
114.     memcpy(&networkExpectedDataLength, mBuffer.data(), sizeof(networkExpectedDataLength));
115.     const auto expectedDataLength = ntohs(networkExpectedDataLength);
116.     startReception(expectedDataLength, State::Data);
117. }
118. void ReceptionHandler::startReception(unsigned int expectedDataLength, State newState)
119. {
120.     mReceived = 0;
121.     mBuffer.clear();
122.     mBuffer.resize(expectedDataLength, 0);
123.     mState = newState;
124. }
125. std::unique_ptr<Messages::Base> ReceptionHandler::recv()
126. {
127.     assert(mSckt != INVALID_SOCKET);
128.     int ret = ::recv(mSckt, missingDataStartBuffer(), missingDataLength(), 0);
129.     if (ret > 0)
130.     {
131.         mReceived += ret;
132.         if (mReceived == mBuffer.size())
133.         {
134.             if (mState == State::Data)
135.             {
136.                 std::unique_ptr<Messages::Base>
137.                 msg = std::make_unique<Messages::UserData>(std::move(mBuffer));
138.                 startHeaderReception();
139.                 return msg;
140.             }
141.             else
142.             {
143.                 startDataReception();
144.                 //!< si jamais les données sont déjà disponibles elles seront ainsi retournées
145.                 //!< directement
146.                 return recv();
147.             }
148.         }
149.         else if (ret == 0)
150.         {
151.             //!< connexion terminée correctement
152.             return std::make_unique<Messages::Disconnection>(Messages::Disconnection::Reason::Disconnected);
153.         }
154.         else // ret < 0
155.         {
156.             //!< traitement d'erreur
157.             int error = Errors::Get();
158.             if (error == Errors::WOULDBLOCK || error == Errors::AGAIN)
159.             {
160.                 return nullptr;
161.             }
162.             else
163.             {
164.                 return std::make_unique<Messages::Disconnection>(Messages::Disconnection::Reason::Lost);

```

TCP/Client.cpp

```

165.     }
166.     }
167. }
168. //////////////////////////////////////
169. //////////////////////////////////////
170. //////////////////////////////////////
171. //////////////////////////////////////
172. //////////////////////////////////////
173. class SendingHandler
174. {
175.     enum class State {
176.         Idle,
177.         Header,
178.         Data,
179.     };
180. public:
181.     SendingHandler() = default;
182.     void init(SOCKET sckt);
183.     bool send(const unsigned char* data, unsigned int datalen);
184.     void update();
185.     size_t queueSize() const;
186.
187. private:
188.     bool sendPendingBuffer();
189.     void prepareNextHeader();
190.     void prepareNextData();
191.
192. private:
193.     std::list<std::vector<unsigned char>> mQueueingBuffers;
194.     std::vector<unsigned char> mSendingBuffer;
195.     SOCKET mSocket{ INVALID_SOCKET };
196.     State mState{ State::Idle };
197. };
198. void SendingHandler::init(SOCKET sckt)
199. {
200.     mSocket = sckt;
201.     if (mState == State::Header || mState == State::Data)
202.     {
203.         mSendingBuffer.clear();
204.     }
205.     mState = State::Idle;
206. }
207. bool SendingHandler::send(const unsigned char* data, unsigned int datalen)
208. {
209.     if (datalen > std::numeric_limits<HeaderType>::max())
210.         return false;
211.     mQueueingBuffers.emplace_back(data, data + datalen);
212.     return true;
213. }
214. void SendingHandler::update()
215. {
216.     assert(mSocket != INVALID_SOCKET);
217.     if (mState == State::Idle && !mQueueingBuffers.empty())
218.     {
219.         prepareNextHeader();
220.     }
221.     while (mState != State::Idle && sendPendingBuffer())
222.     {
223.         if (mState == State::Header)
224.         {
225.             prepareNextData();
226.         }
227.         else
228.         {
229.             if (!mQueueingBuffers.empty())
230.             {
231.                 prepareNextHeader();
232.             }
233.             else
234.             {
235.                 mState = State::Idle;

```


TCP/Client.cpp

```

236.     }
237.   }
238.   }
239. }
240. bool SendingHandler::sendPendingBuffer()
241. {
242.     if (mSendingBuffer.empty())
243.         return true;
244.
245.     //!< envoi des données restantes du dernier envoi
246.     int
247.     sent = ::send(mSocket, reinterpret_cast<char*>(mSendingBuffer.data()), static_cast<int>(mSendingBuffer.size()),
248.     {
249.         if (sent > 0)
250.         {
251.             if (sent == mSendingBuffer.size())
252.             {
253.                 //!< toutes les données ont été envoyées
254.                 mSendingBuffer.clear();
255.                 return true;
256.             }
257.             else
258.             {
259.                 //!< envoi partiel
260.                 memmove(mSendingBuffer.data() + sent, mSendingBuffer.data(), sent);
261.                 mSendingBuffer.erase(mSendingBuffer.cbegin() + sent, mSendingBuffer.cend());
262.             }
263.         }
264.         return false;
265.     }
266. void SendingHandler::prepareNextHeader()
267. {
268.     assert(!mQueueingBuffers.empty());
269.     const auto header = static_cast<HeaderType>(mQueueingBuffers.front().size());
270.     const auto networkHeader = htons(header);
271.     mSendingBuffer.clear();
272.     mSendingBuffer.resize(HeaderSize);
273.     memcpy(mSendingBuffer.data(), &networkHeader, sizeof(HeaderType));
274.     mState = State::Header;
275. }
276. void SendingHandler::prepareNextData()
277. {
278.     assert(!mQueueingBuffers.empty());
279.     mSendingBuffer.swap(mQueueingBuffers.front());
280.     mQueueingBuffers.pop_front();
281.     mState = State::Data;
282. }
283. size_t SendingHandler::queueSize() const
284. {
285.     size_t s = std::accumulate(mQueueingBuffers.cbegin(),
286.     mQueueingBuffers.cend(), static_cast<size_t>(0), [](size_t n, const std::vector<unsigned char>&
287.     queuedItem) {
288.         return n + queuedItem.size() + HeaderSize;
289.     });
290.     if (mState == State::Data)
291.         s += mSendingBuffer.size();
292.     return s;
293. }
294. //////////////////////////////////////
295. //////////////////////////////////////
296. //////////////////////////////////////
297. //////////////////////////////////////
298. //////////////////////////////////////
299. //////////////////////////////////////
300. class ClientImpl
301. {
302.     enum class State {
303.         Connecting,
304.         Connected,
305.         Disconnected,
306.     };
307. };
308. public:

```

TCP/Client.cpp

```

304.     ClientImpl() = default;
305.     ~ClientImpl();
306.
307.     bool init(SOCKET&& sckt, const sockaddr_in& addr);
308.     bool connect(const std::string& ipaddress, unsigned short port);
309.     void disconnect();
310.     bool send(const unsigned char* data, unsigned int len);
311.     std::unique_ptr<Messages::Base> poll();
312.
313.     uint64_t id() const { return static_cast<uint64_t>(mSocket); }
314.     const sockaddr_in& destinationAddress() const { return mAddress; }
315.
316. private:
317.     void onConnected(const sockaddr_in& addr);
318.
319. private:
320.     ConnectionHandler mConnectionHandler;
321.     SendingHandler mSendingHandler;
322.     ReceptionHandler mReceivingHandler;
323.     sockaddr_in mAddress{ 0 };
324.     SOCKET mSocket{ INVALID_SOCKET };
325.     State mState{ State::Disconnected };
326. };
327. ClientImpl::~ClientImpl()
328. {
329.     disconnect();
330. }
331. bool ClientImpl::init(SOCKET&& sckt, const sockaddr_in& addr)
332. {
333.     assert(sckt != INVALID_SOCKET);
334.     if (sckt == INVALID_SOCKET)
335.         return false;
336.
337.     assert(mState == State::Disconnected);
338.     assert(mSocket == INVALID_SOCKET);
339.     if (mSocket != INVALID_SOCKET)
340.         disconnect();
341.
342.     mSocket = sckt;
343.     if (!SetNonBlocking(mSocket))
344.     {
345.         disconnect();
346.         return false;
347.     }
348.     onConnected(addr);
349.     return true;
350. }
351. bool ClientImpl::connect(const std::string& ipaddress, unsigned short port)
352. {
353.     assert(mState == State::Disconnected);
354.     assert(mSocket == INVALID_SOCKET);
355.     if (mSocket != INVALID_SOCKET)
356.         disconnect();
357.     mSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
358.     if (mSocket == INVALID_SOCKET)
359.     {
360.         return false;
361.     }
362.     else if (!SetNonBlocking(mSocket))
363.     {
364.         disconnect();
365.         return false;
366.     }
367.     if (mConnectionHandler.connect(mSocket, ipaddress, port))
368.     {
369.         mState = State::Connecting;
370.         return true;
371.     }
372.     return false;
373. }
374. void ClientImpl::disconnect()

```

TCP/Client.cpp

```

375.     {
376.         if (mSocket != INVALID_SOCKET)
377.         {
378.             CloseSocket(mSocket);
379.         }
380.         mSocket = INVALID_SOCKET;
381.         memset(&mAddress, 0, sizeof(mAddress));
382.         mState = State::Disconnected;
383.     }
384.     bool ClientImpl::send(const unsigned char* data, unsigned int len)
385.     {
386.         return mSendingHandler.send(data, len);
387.     }
388.     std::unique_ptr<Messages::Base> ClientImpl::poll()
389.     {
390.         switch (mState)
391.         {
392.             case State::Connecting:
393.             {
394.                 auto msg = mConnectionHandler.poll();
395.                 if (msg)
396.                 {
397.                     if (msg->result == Messages::Connection::Result::Success)
398.                     {
399.                         onConnected(mConnectionHandler.connectedAddress());
400.                     }
401.                     else
402.                     {
403.                         disconnect();
404.                     }
405.                 }
406.                 return msg;
407.             } break;
408.             case State::Connected:
409.             {
410.                 mSendingHandler.update();
411.                 auto msg = mReceivingHandler.recv();
412.                 if (msg)
413.                 {
414.                     if (msg->is<Messages::Disconnection>())
415.                     {
416.                         disconnect();
417.                     }
418.                 }
419.                 return msg;
420.             } break;
421.             case State::Disconnected:
422.             {
423.                 } break;
424.             }
425.             return nullptr;
426.         }
427.         void ClientImpl::onConnected(const sockaddr_in& addr)
428.         {
429.             mAddress = addr;
430.             mSendingHandler.init(mSocket);
431.             mReceivingHandler.init(mSocket);
432.             mState = State::Connected;
433.         }
434.         //////////////////////////////////////
435.         //////////////////////////////////////
436.         //////////////////////////////////////
437.         //////////////////////////////////////
438.         //////////////////////////////////////
439.         Client::Client() {}
440.         Client::~Client() {}
441.         Client::Client(Client&& other)
442.             : mImpl(std::move(other.mImpl))
443.         {}
444.         Client& Client::operator=(Client&& other)
445.         {

```

TCP/Client.cpp

```
446.     mImpl = std::move(other.mImpl);
447.     return *this;
448. }
449. bool Client::init(SOCKET&& sckt, const sockaddr_in& addr)
450. {
451.     if (!mImpl)
452.         mImpl = std::make_unique<ClientImpl>();
453.     return mImpl && mImpl->init(std::move(sckt), addr);
454. }
455. bool Client::connect(const std::string& ipaddress, unsigned short port)
456. {
457.     if (!mImpl)
458.         mImpl = std::make_unique<ClientImpl>();
459.     return mImpl && mImpl->connect(ipaddress, port);
460. }
461. void Client::disconnect() { if (mImpl) mImpl->disconnect(); }
462. bool Client::send(const unsigned char* data, unsigned int len) { return mImpl && mImpl->send(data, len); }
463. std::unique_ptr<Messages::Base> Client::poll() { return mImpl ? mImpl->poll() : nullptr; }
464. uint64_t Client::id() const { return mImpl ? mImpl->id() : 0xffffffffffffffff; }
465. const sockaddr_in& Client::destinationAddress() const { static sockaddr_in empty{ 0 }; return mImpl ? mImpl->destinationAddress() : empty; }
466. }
467. }
```

**Langue Télécharger le code source complet.****Article précédent****<< TCP - Quelle architecture de client
visée ?**