

Tutoriel Boost.Asio

par Gwenaël Dunand ([Page personnelle](#))

Date de publication : 19 mars 2009

Dernière mise à jour :

Cet article introduit la programmation réseau en C++ à l'aide de Boost.Asio. Après un rapide tour d'horizon de l'architecture globale de Boost.Asio et des possibilités offertes par cette bibliothèque (opérations synchrones et asynchrones), cet article présentera les Timers, la communication TCP et UDP. Des exemples concrets de clients et serveurs seront étudiés. Enfin, un projet réseau réaliste avec un code robuste sera présenté en dernière partie.

I - Introduction.....	3
I-A - Réseau et langage de programmation.....	3
I-B - Pré-requis.....	3
I-C - Installation de Boost.Asio.....	3
I-D - Code et commentaires dans ce tutoriel.....	3
II - Les bases de Boost.Asio.....	4
II-A - Opérations synchrones.....	4
II-B - Opérations asynchrones.....	5
II-C - Synchrone / Asynchrone ?.....	7
II-D - Architecture de Boost.Asio.....	7
III - Les Timers.....	8
III-A - Timers synchrones.....	8
III-B - Timers asynchrones.....	8
IV - Le protocole TCP.....	9
IV-A - Introduction.....	9
IV-B - Lecture / écriture courte et transfert complet.....	11
IV-C - Exemple d'un client synchrone.....	11
IV-D - Exemple d'un serveur synchrone.....	13
IV-E - Exemple d'un serveur asynchrone.....	14
IV-E-1 - Premier essai.....	14
IV-E-2 - Avec des pointeurs intelligents.....	16
IV-E-3 - Intégration d'un timer.....	17
IV-F - Exemple d'un client asynchrone.....	19
V - Le protocole UDP.....	21
V-A - Exemple d'un client synchrone.....	21
V-B - Exemple d'un serveur synchrone.....	22
VI - Les sockets iostreams.....	23
VI-A - Exemple de client synchrone.....	23
VI-B - Exemple de serveur synchrone.....	24
VII - Projet démo : réalisation d'un 'chat' avec Boost.Asio.....	24
VII-A - Cahier des charges.....	24
VII-B - Conception.....	24
VII-B-1 - Abstractions.....	25
VII-B-2 - Architecture.....	25
VII-C - Implémentation du serveur.....	26
VII-C-1 - Classe chat_message.....	26
VII-C-2 - Classe tcp_connection.....	26
VII-C-3 - Classe chat_server.....	30
VII-C-4 - Classe chat_session.....	31
VII-C-5 - Classe chat_room.....	33
VII-C-6 - Bilan.....	35
VII-D - Bilan.....	35
VIII - Conclusion.....	35
IX - Remerciements.....	35

I - Introduction

I-A - Réseau et langage de programmation

En langage C++ (et C), la programmation réseau est dépendante du type des machines et systèmes d'exploitation utilisés. Ce qui ne rend pas facile la tâche du programmeur... Par exemple, il est souvent fastidieux de réaliser une version Windows et Linux, car s'il existe de nombreux points communs avec POSIX (socket, bind, connect, listen, accept), il existe aussi de nombreuses divergences dans les en-têtes et les fonctions d'initialisations. Si bien que développer une application réseau portable peut rapidement devenir un véritable challenge pour le programmeur.

Boost.Asio répond à ces problèmes en proposant une bibliothèque de haut niveau portable, facile à utiliser et dans un style C++ très élégant.

I-B - Pré-requis

Les concepts C++ présentés dans une première partie sont relativement simples. La dernière partie fait toutefois appel à des notions intermédiaires nécessaires au développement d'applications robustes, comme les **pointeurs intelligents** [Loïc Joly] ou encore la **sérialisation** [Pierre Schwartz]. Le lecteur est vivement encouragé à lire les articles correspondants en cas de difficultés.

La compréhension globale du fonctionnement d'un réseau n'est pas obligatoire, mais conseillée. Le lecteur trouvera parmi les liens suivant de bonnes références :

- **les sockets en C** [Benjamin Roux],
- **la théorie des réseaux locaux et étendus** [Patrick Hautrive],
- **Initiation à la programmation réseau sous Windows** [Jessee Edouard]

I-C - Installation de Boost.Asio

Boost.Asio fait parti de la grande bibliothèque Boost. Les exemples de cet article ont été compilés avec VC++ Express 2008 et Boost 1.37. Pour pouvoir utiliser Boost.Asio, il est conseillé d'installer boost.regex, boost.thread, boost.date_time et boost.serialization. Voici quelques liens expliquant comment installer boost, soit à l'aide d'un exécutable (Windows et Visual uniquement), ou bien en compilant à l'aide de bjam:

- **Installer et utiliser Boost/Boost.TR1 avec Visual C++** [Aurélien Regat-Barrel]
- **Compilation de Boost** [ram-0000]

Pour ne prendre que l'utile pour ce tutoriel, compilez boost avec la ligne suivante:

```
bjam --with-system --with-thread --with-date_time --with-regex --with-serialization stage
```

I-D - Code et commentaires dans ce tutoriel

Dans la mesure du possible, le code ne sera pas coupé par des commentaires, afin de ne pas nuire à la lisibilité du programme dans son ensemble. Ainsi, j'ai choisi dans la plupart des cas de décrire le programme d'abord, avec des références sur l'endroit du code entre parenthèses. Cela n'empêche évidemment pas de laisser quelques commentaires dans le code, comme tout bon programme.

Pour plus de lisibilité, les noms de fonctions/classes seront en italiques dans tout l'article (excepté dans le code) ainsi que les termes anglais.

Exemple:

La fonction `ma_fonction()` prend deux chaînes de caractères en paramètres (`std::string`).

On commence par afficher les deux chaînes (1).

On stocke la somme des deux chaînes dans une variable temporaire (2).

On lance la fonction `cherche_char` sur chaque caractère (3).

```
void ma_fonction(const std::string& str1, const std::string& str2)
{
    // On affiche les chaînes // (1)
    std::cout << str1 << std::endl;
    std::cout << str2 << std::endl;

    // On stocke le tout dans une autre chaîne // (2)
    std::string chaine = str1 + str2;

    // On recherche des caractères précis // (3)
    std::for_each(chaine.begin(), chaine.end(), cherche_char);
}
```

II - Les bases de Boost.Asio

Dans cette partie, nous allons découvrir ensemble les bases de Boost.Asio, à savoir les classes principales, les opérations synchrones et les opérations asynchrones.



*Quelque soit la partie de Boost.Asio que l'on utilise, le programme devra toujours posséder sa pièce centrale : un **io_service**. C'est en quelque sorte le "cœur" de la bibliothèque...*

```
#include <boost/asio.hpp>

int main()
{
    boost::asio::io_service io_service;
    //Code...
}
```

Nous pouvons également spécifier explicitement à Boost Asio la plateforme cible. Il se peut qu'elle soit différente de la plateforme de développement.

Boost.Asio et Windows XP

```
#define _WIN32_WINNT 0x0501
```

Boost.Asio et Windows 2000

```
#define _WIN32_WINNT 0x0500
```

Pour rappel, il existe deux grands types d'opérations réseaux : les opérations synchrones et les opérations asynchrones. Nous allons détailler ces deux types d'opérations dans cette partie, avec leurs avantages et inconvénients.

II-A - Opérations synchrones

Une opération synchrone bloque la fonction appelante jusqu'à ce qu'elle ait terminé. Considérons le morceau de code suivant, qui envoie un `msg` par l'intermédiaire d'une `socket`:

Envoi synchrone

```
void Mafonction()
{
    //Code...
    boost::asio::send(socket, boost::asio::buffer(msg)); // Envoi des données par le socket
    std::cout << "Terminé" << std::endl;
}
```

Le message "Terminé" ne sera affiché à l'écran que lorsque la fonction d'envoi de donnée sera terminée. On dit que la fonction est **bloquante**. Dans cet exemple, la fonction `send()` pourrait ne pas bloquer longtemps car il n'y a pas besoin d'attendre pour envoyer des données. Cependant, on pourrait imaginer des cas où on fait des demandes de `send` plus rapidement qu'elles ne sont transmises...

Autre exemple, que va t-il se passer si on effectue non pas un envoi, mais une réception de données?

Réception synchrone

```
void Mafonction()
{
    //Code...
    boost::asio::read(socket, boost::asio::buffer(msg)); // Réception des données sur le socket
    std::cout << "Terminé" << std::endl;
}
```

Dans ce code, la fonction `read()` va attendre la réception de donnée sur un port précis et on peut parfois attendre longtemps... Tant que la fonction `read()` n'a pas reçu une certaine quantité de donnée, la fonction apelante `Mafonction()` va rester bloquée.

Pour pallier à ce mode de fonctionnement, on pourrait lancer la réception de données dans un thread par exemple. On s'affranchirait ainsi du problème de la réception synchrone. Par contre, le développeur devra dans ce cas gérer les accès concurrents lui-même. Il existe une solution beaucoup sûre et plus élégante pour résoudre ce problème : les opérations asynchrones.

II-B - Opérations asynchrones

Des opérations asynchrones sont des opérations qui rendent la main immédiatement à l'appelant même si elles ne sont pas encore terminées. On sait quand elles commencent, mais on ne sait pas quand elles finissent, dans l'absolu. Par contre, elles appellent une fonction *callback* lorsqu'elles ont terminé. En effet, une opération asynchrone prend en général en paramètre un *completion_handler* (cf. exemple suivant) qu'elle va appeler lorsqu'elle a terminé son travail. Elle n'empêche pas la fonction apelante de continuer d'exécuter son code, on dit donc que l'opération est **non bloquante**.

Les *callback* sont gérés dans une **boucle de traitement** des événements. Cette boucle de traitement est instanciée par l'utilisateur à l'aide de la fonction `io_service::run()`. Cette fonction est bloquante tant qu'il reste des opérations asynchrones en cours et rend la main lorsqu'il n'y a plus de *callback* à exécuter. Il faut donc impérativement donner du travail asynchrone à réaliser avant d'appeler `io_service::run()`. Pour maintenir une IHM active, `io_service::run` peut être exécuté dans un thread dédié.





*Sur certaines plateformes, l'implémentation de Boost.Asio peut éventuellement utiliser des threads **en interne** pour émuler l'asynchronicité. Ces threads restent invisibles pour l'utilisateur et la bibliothèque s'utilise comme si toutes les opérations étaient lancées dans un thread unique. Tant que le programme ne possède qu'un seul `io_service`, la boucle de traitement des événements est traitée de manière **strictement séquentielle**. Le développeur n'a donc **pas** à gérer les accès concurrents.*

Le prototype de la fonction *callback* de Boost.Asio varie suivant les opérations asynchrones. La fonction *async_read* prend deux paramètres, alors que la fonction *async_connect* n'en prend qu'un. Afin de mieux contrôler soi même les arguments passés au *callback*, on utilisera *boost::bind*.

Réception asynchrone

```
void completion_handler(const boost::system::error_code& error)
{
    std::cout << "Terminé" << std::endl;
}
void Mafonction()
{
    //Lecture asynchrone
    boost::asio::async_read(socket, boost::asio::buffer(msg),
        boost::bind(&completion_handler, boost::asio::placeholders::error)
    );
    std::cout << "Ca s'affiche !" << std::endl;
}
```

 **Attention, il est de la responsabilité du développeur de s'assurer que la durée de vie des structures/variables utilisées lors d'opérations asynchrones sont suffisantes. Même si Boost.Asio effectue des copies de buffer, la responsabilité de la mémoire sous jacente revient à l'appelant. La mémoire doit vivre jusqu'à l'appel du callback!**
Le non respect de cette règle peut entrainer violation de mémoire et/ou comportement indéterminé parfois difficiles à déboguer.

 **En particulier, le buffer de réception ne doit pas être un buffer temporaire alloué sur la pile s'il est libéré avant la fin de l'exécution et le buffer d'envoi ne doit pas être libéré prématurément.**

async_read() ne bloque pas la fonction appelante *Mafonction()*. Une fois que l'opération *async_read()* est terminée, notre fonction *completion_handler()* sera appelée pour nous signaler qu'*async_read()* a bien effectué son travail. Pour nous informer du bon déroulement ou non de l'opération asynchrone, il nous suffit d'inspecter le paramètre *error_code* de notre *callback*. Avec ce paramètre, on peut traiter de nombreux cas comme :


- Connexion refusée
- Connexion perdue
- Argument invalide
- Message trop long
- etc..

Réception asynchrone avec retour de code d'erreur

```
void completion_handler(const boost::system::error_code& e)
{
    if (!error)
    {
        std::cout << "Tout s'est bien passé" << std::endl;
    }
    else {
        // Problème que l'on peut identifier
        // access_denied, connection_aborted, ...
    }
}
```

Pour pouvoir effectuer des opérations asynchrones, nous devons lancer la boucle de gestion des événements par la fonction *io_service::run()*:

```
boost::asio::io_service ios;
// .... opérations asynchrones
ios.run();
```

 **Rappel** : la fonction `run()` est bloquante continue de "travailler" tant qu'il reste des opérations asynchrones en cours.

II-C - Synchrone / Asynchrone ?

Inconvénients des opérations asynchrones:

- Complexité du programme
- Consommation mémoire (car les buffers de réception doivent tous être alloués et indépendants)

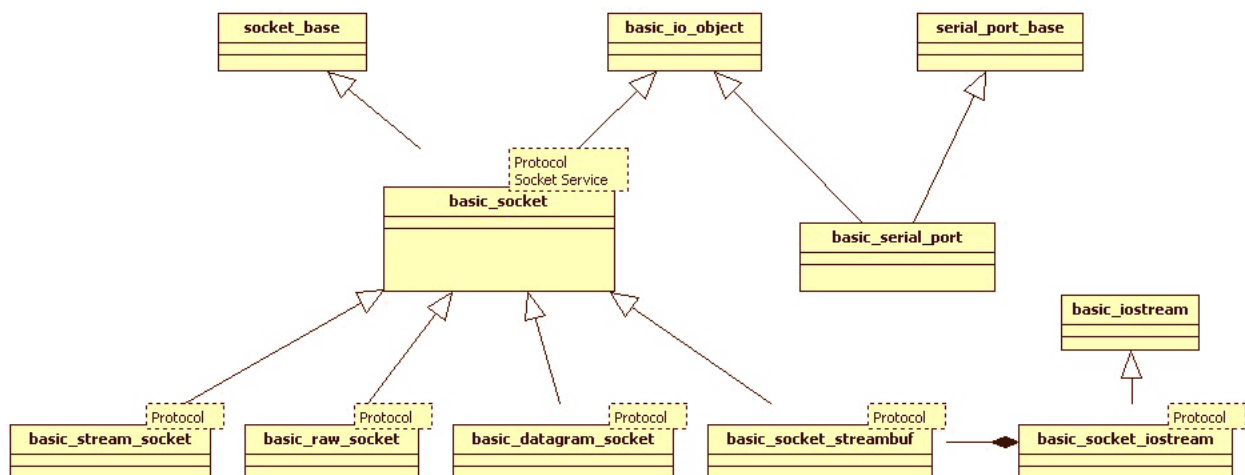
Avantages des opérations asynchrones:

- Performance
- Opérations non bloquantes

Le mode asynchrone sera beaucoup utilisé par la suite, en raison de ses performances et de ses appels non bloquants.

II-D - Architecture de Boost.Asio

Boost.Asio permet de gérer deux types de périphériques de communications : port Ethernet et port série. En ce qui concerne les réseaux, Boost.Asio permet l'utilisation de plusieurs protocoles : TCP (mode connecté), UDP (mode non connecté), ICMP. Le diagramme ci dessous permet d'apprécier la diversité des modes de communication qu'offre Boost.Asio.



Diagrammes des classes de Boost.Asio

Les classes d'utilisation communes sont fournies par Boost à l'aide d'un `typedef`, dans le bon `namespace`:

```
namespace tcp {
//...
typedef basic_stream_socket< tcp > socket;
typedef basic_socket_iostream< tcp > iostream;
```

```

}

namespace udp
{
    //...
    typedef basic_datagram_socket< udp > socket;
}

```

III - Les Timers

Boost.Asio propose un seul timer (le `deadline_timer`) qui fait tout ce qu'on lui demande, c'est à dire compter le temps, que ce soit de manière synchrone ou asynchrone. Le `deadline_timer` prend en paramètre un `io_service`, toujours indispensable, ainsi qu'une durée de référence.

Construction du timer Boost.Asio

```

// Construction d'un timer d'une seconde.
boost::asio::deadline_timer t(io, boost::posix_time::seconds(1));

```



Le timer se déclenche au moment de sa construction et non pas à partir de l'attente !

III-A - Timers synchrones

Nous allons créer dans cette section un simple timer bloquant qui attend 5 secondes, puis rend la main au programme.

Timer synchrone

```

#include <iostream>
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

int main()
{
    boost::asio::io_service io; // Service principal

    boost::asio::deadline_timer t(io, boost::posix_time::seconds(5)); // Commence à compter dès sa création
    t.wait(); // On attend que le timer expire

    std::cout << "Terminé !" << std::endl;

    return 0;
}

```

La programme affiche donc "Terminé !" après avoir attendu les 5 secondes du timer.

III-B - Timers asynchrones

On souhaite maintenant que notre timer de la section précédente soit non bloquant. Pour cela, on va utiliser le mode asynchrone, déjà vu à la section précédente.

Le fonctionnement est très semblable à ce que nous avons déjà vu :

- 1 Création d'un timer avec un durée de compte à rebours de 5 secondes
- 2 Lancement en mode asynchrone du timer
- 3 Lorsque le timer est expiré, il appelle son *callback* : la fonction `print()`
- 4 La fonction `print()` affiche "Terminé !"

Timer asynchrone

```

#include <iostream>
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

void print(const boost::system::error_code& /*error*/) // (3)
{
    std::cout << "Terminé !" << std::endl; // (4)
}

int main()
{
    boost::asio::io_service io;

    boost::asio::deadline_timer t(io, boost::posix_time::seconds(5)); // (1)
    t.async_wait(print); // Attente asynchrone (2)

    io.run();

    return 0;
}

```



Rappel : Il est important de toujours donner du travail asynchrone à effectuer AVANT d'appeler `io_service::run()`, sinon `io_service::run()` rendra la main immédiatement

Il peut arriver que l'on crée un timer pour s'en servir plus tard. Dans ce cas, il a de forte chance d'être dans l'état "expiré". La fonction membre `expires_at()` permet de changer la date d'expiration du timer, comme ceci:

```

boost::asio::deadline_timer t(io, boost::posix_time::seconds(5));
// Code... le temps passe... le timer est déjà expiré.

// On redonne au timer une nouvelle deadline:
t.expires_at(t.expires_at() + boost::posix_time::seconds(5));

// Et notre timer est de nouveau en train de compte ses 5 secondes.
t.async_wait(print); // Attente asynchrone

```



Nous verrons [dans la prochaine section](#) comment un timer peut être utilisé pour arrêter des opérations asynchrones en cours.

IV - Le protocole TCP

Dans cette section, nous allons étudier en détail comment communiquer en TCP/IP avec Boost.Asio, en mode synchrone et asynchrone. Nous étudierons deux exemples d'architecture client/serveur. Un exemple beaucoup plus poussé sera présenté dans la dernière partie de ce tutoriel.

IV-A - Introduction

Le protocole TCP s'utilise uniquement en mode connecté. Les fonctions et options classiques bas niveau POSIX sont disponibles, (`bind()`, `listen()`, `SO_REUSEADDR`, etc.) pour permettre au développeur de contrôler plus finement son application. Pour des applications "classiques", l'API haut niveau de Boost.Asio avec les valeurs par défaut suffisent amplement. C'est ce que nous allons utiliser ici.

Listons dans le tableau suivant les classes TCP de Boost.Asio les plus utilisées.

Les classes Boost.Asio	Ce qu'elles représentent
<code>boost::asio::ip::tcp::endpoint</code>	Représente un couple {adresse IP, port}
<code>boost::asio::ip::tcp::resolver</code>	Le résolveur permet de construire un TCP endpoint à partir du nom d'un serveur
<code>boost::asio::ip::tcp::socket</code>	une socket, interface de communication avec le monde extérieur. Elle possède les fonctions membres <i>connect()</i> pour se connecter à un serveur. <i>send()</i> et <i>async_send()</i> pour envoyer des données par cette socket et <i>receive()</i> et <i>async_receive()</i> pour en recevoir.
<code>boost::asio::ip::tcp::acceptor</code>	C'est elle qui possède la fonction membre <i>accept()</i> et <i>async_accept()</i> pour accepter les connexions entrantes sur un serveur.

Un *endpoint* prend une adresse et un port dans le constructeur. Il existe plusieurs moyens pour le remplir correctement:

- Si on connaît l'adresse IP du serveur, on va lui passer par une chaîne de caractère.

Construction d'un endpoint à partir d'une adresse

```
tcp::endpoint endpoint(boost::asio::ip::address::from_string("192.168.0.4"), 13);
```

- Si on ne connaît que le nom DNS, il va falloir passer par un *resolver*. Dans l'exemple de code ci-dessous, on va récupérer une liste d'adresses IP correspondant à l'acronyme recherché.

On commence par créer un *resolver* (1) que l'on va utiliser pour récupérer **toutes les IP** correspondant au nom DNS fourni en entrée (ici www.developpez.com). Dans notre exemple, on va se connecter sur le port 80, qui est le port standard HTTP (2).

On lance la résolution (3). La fonction *resolver::resolve()* retourne un itérateur sur le premier *endpoint*. Libre à nous d'en faire ce que l'on veut, comme l'afficher par exemple (4).

Construction d'un ou plusieurs endpoint à partir d'un acronyme

```
#define _WIN32_WINNT 0x0501 // Asio et Windows XP
#include <boost/asio.hpp>
#include <iostream>

int main()
{
    // Création du service principal et du résolveur.
    boost::asio::io_service ios;
    boost::asio::ip::tcp::resolver resolver(ios); // (1)

    // Paramétrage du resolver sur Developpez.com
    boost::asio::ip::tcp::resolver::query query("www.developpez.com", "80"); // (2)

    // On récupère une "liste" d'itérateur
    boost::asio::ip::tcp::resolver::iterator iter = resolver.resolve(query); // (3)
    boost::asio::ip::tcp::resolver::iterator end; //Marqueur de fin
    while (iter != end) // On itère le long des endpoints
    {
        boost::asio::ip::tcp::endpoint endpoint = *iter++;
        std::cout << endpoint << std::endl; // on affiche (4)
    }

    return 0;
}
```

www.developpez.com possède une seul IP, contre 3 pour www.google.fr, ce qu'on peut observer dans le tableau suivant :


Acronyme	Sortie écran
www.developpez.com	87.98.130.52:80
www.google.fr	<ul style="list-style-type: none"> 66.102.9.104:80 66.102.9.147:80 66.102.9.99:80

 Les endpoint sont très utiles et importants puisqu'ils permettent de travailler indépendamment du type d'adresses utilisées : IPv4 et IPv6.

IV-B - Lecture / écriture courte et transfert complet

Que ce soit en mode synchrone ou asynchrone, il existe deux types de communication *via* les sockets:

- **Les transferts courts** : Les messages transmis ne comportent pas de délimitations. Ces appels réseaux peuvent transmettre moins d'informations que le contenu original. Ce phénomène peut être dû au contrôle du flux du protocole de transport, ou bien à la bufferisation des données. Au final, on récupérera bien l'intégralité des données, mais en plus ou moins de morceaux qu'au départ. A titre d'exemple, 2 trames envoyées peuvent se traduire aussi bien par 1 ou 2 ou 3 réceptions.
- **Les transferts complets**: Dans ces appels réseaux, il faut préciser exactement la taille des données que l'on souhaite envoyer et recevoir. Dans le cas de la réception particulièrement, les données ne sont rendues disponibles (appel du *callback*) que lorsque le buffer de réception est **plein**. Ce mode de fonctionnement est particulièrement indiqué pour des messages de taille fixe ou lorsqu'on indique d'abord par un premier message de taille fixe la taille des données que l'on doit recevoir. Beaucoup d'applications réseaux ont besoin de ces garanties pour fonctionner correctement, lorsqu'on doit disposer d'un paquet intégral (pour décompression ou désérialisation).

 Le mode transfert complet est implémenté comme une succession de transferts courts, jusqu'au remplissage du buffer de réception. Il évite par conséquent au développeur de rassembler les trames applicatives reçues.


	transfert court	transfert complet
Synchrone	socket::read_some() socket::write_some() socket::receive() socket::send()	boost::asio::read() boost::asio::write()
Asynchrone	socket::async_read_some() socket::async_write_some() socket::async_receive() socket::async_send()	boost::asio::async_read() boost::asio::async_write()

IV-C - Exemple d'un client synchrone

Etudions maintenant un exemple concret d'un client synchrone se connectant à un serveur. Une fois connecté, le client va récupérer ce que le serveur lui envoie et se déconnecter lorsque le serveur n'aura plus rien à dire.

Nous avons déjà étudié dans les sections précédentes le début du code : création du service principal, et création du *endpoint*. Reste à créer une socket TCP, pour pouvoir communiquer avec l'extérieur (1). On connecte (2) ensuite la socket fraîchement créée au serveur à l'aide du *endpoint*.

Ensuite, il faut disposer d'un buffer pour la réception des données (3). Nous avons le choix : `std::vector`, `boost::array`, ou bien encore tableau "C" classique. Ici j'ai choisi un `boost::array`, de type `char` et de taille fixe 128. Le buffer est de taille confortable et permettra de récupérer en une fois les données reçues, *a priori*. Toutefois, il se peut que le message envoyé par le serveur soit très long. D'où la question : que se passe t-il si la taille du buffer est trop petite par rapport aux données envoyées par le serveur? Pas de problèmes, dans ce cas Boost.Asio lira le message en plusieurs fois, la fonction `read_some()` ne lira pas plus que la taille du buffer.

 *`std::vector` et `boost::array` font parti des classes utilisables directement dans le constructeur de `boost::asio::buffer`, ils sont donc particulièrement indiqués, je vous encourage à les utiliser.*

Les données sont récupérées *via* une communication courte synchrone (4). Dans le cas présent, on ne tient pas particulièrement à connaître la taille des données reçues et on ne fait pas grand chose à part les afficher. Cette voie de communication était donc tout indiquée... On notera qu'il faut utiliser la valeur de retour de la fonction `read_some()` pour savoir combien d'octets ont été lus.

Il existe deux méthodes pour savoir si on est arrivé à la fin de la lecture:

- Soit on fourni une variable (5) qui sera remplie par la fonction `read_some()` à la valeur *end of file*
- Soit n'utilise pas de codes d'erreur. Dans ce cas une exception sera lancée. A nous de l'attraper!

Je préfère nettement la première solution, il est en effet inutile de faire appel aux exceptions juste pour être prévenus d'une fin de lecture. Gardons les pour des choses plus utiles...

Enfin, on affiche au cours de chaque boucle le contenu de notre buffer (6).

Voici le code avec les références au texte ci dessus entre parenthèses:

Client synchrone

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/array.hpp>

int main()
{
    // Création du service principal et du résolveur.
    boost::asio::io_service ios;

    // On veut se connecter sur la machine locale, port 7171
    tcp::endpoint endpoint(boost::asio::ip::address::from_string("127.0.0.1"), 7171);

    // On crée une socket // (1)
    tcp::socket socket(ios);

    // Tentative de connexion, bloquante // (2)
    socket.connect(endpoint);

    // Création du buffer de réception // (3)
    boost::array<char, 128> buf;
    while (1)
    {
        boost::system::error_code error;
        // Réception des données, len = nombre d'octets reçus // (4)
        int len = socket.read_some(boost::asio::buffer(buf), error);

        if (error == boost::asio::error::eof) // (5)
        {
            std::cout << "\nTerminé !" << std::endl;
            break;
        }

        // On affiche (6)
        std::cout.write(buf.data(), len);
    }
}
```

Client synchrone

```
}  
  
return 0;  
}
```

Etudions maintenant le serveur correspondant.

IV-D - Exemple d'un serveur synchrone

On commence par la création d'un accepteur (1), qui prend en paramètre n'importe quelle adresse IP V4, sur le port 7171. `tcp::v4()` est l'équivalent du `INADDR_ANY` du standard POSIX et cela correspond à un bind sur toutes les interfaces réseau du serveur.

On crée le message de bienvenue (2).

Ensuite on réalise une boucle dans laquelle on effectue une attente bloquante d'un client. On commence par créer une socket (3) puis l'accepteur va réaliser l'attente bloquante d'une connexion. Lorsqu'une connexion est établie, on affiche "Client reçu !" sur notre serveur, bien que ce ne soit pas très utile.

On utilise ensuite la socket connectée au client pour lui envoyer un message (5). Notez qu'on utilise ici aussi une écriture courte et donc le message envoyé pourrait très bien arriver en plusieurs morceaux ! Bien sûr, dans le cas présent d'un si petit message, on peut présumer l'envoyer en un seul morceau. Mais on ne peut en avoir la garantie. Dès la fin de la boucle, on perd la connexion avec le client ici pour commencer une attente bloquante sur un autre client.

Serveur synchrone

```
#define _WIN32_WINNT 0x0501 // Windows XP  
  
#include <boost/asio.hpp>  
#include <boost/array.hpp>  
#include <iostream>  
  
using boost::asio::ip::tcp;  
  
int main()  
{  
    // Création du service principal et du résolveur.  
    boost::asio::io_service ios;  
  
    // Création de l'accepteur avec le port d'écoute 7171 et une adresse quelconque de type IPv4 // (1)  
    tcp::acceptor acceptor(ios, tcp::endpoint(tcp::v4(), 7171));  
  
    std::string msg ("Bienvenue sur le serveur !"); // (2)  
    // On attend la venue d'un client  
    while (1)  
    {  
        // Création d'une socket  
        tcp::socket socket(ios); // (3)  
  
        // On accepte la connexion  
        acceptor.accept(socket); // (4)  
        std::cout << "Client reçu ! " << std::endl;  
  
        // On envoi un message de bienvenue  
        socket.send(boost::asio::buffer(msg)); // (5)  
    }  
  
    return 0;  
}
```

Ce code illustre de manière simple comment réaliser un serveur minimal. Bien sûr avec cette technique on ne peut accueillir qu'un seul client à la fois, ce n'est pas acceptable. La prochaine partie propose de résoudre ce problème en utilisant une connexion asynchrone.

IV-E - Exemple d'un serveur asynchrone

Dans cette partie, nous allons traiter plusieurs clients simultanément, en modélisant un serveur et une connexion TCP. La fonction `main()` sera réduite à sa plus simple expression :

```
int main()
{
    try
    {
        boost::asio::io_service io_service;

        // Création d'un serveur
        tcp_server server(io_service, 7171);
        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```


IV-E-1 - Premier essai

Créons une classe `tcp_connection` dont le rôle est de gérer l'interaction avec le client : envoi du message de bienvenue. La gestion de la première connexion sera faite par un serveur.

Le constructeur de la classe (1) prend en paramètre un `io_service` avec lequel il construit sa socket. Cette socket doit pouvoir être accessible de l'extérieur de la classe afin d'initialiser la connexion entre le client et le serveur sur cette socket.

On pourrait faire autrement et passer une socket en paramètre du constructeur par exemple, un fois la connexion établie par le serveur. Toutefois, les classes `socket` sont **non copiables**, donc pour ce faire il faudrait passer une référence et `tcp_connection` ne serait donc pas propriétaire de la socket. Le mieux est donc de laisser `tcp_connection` posséder cette socket et donner la possibilité au serveur de s'en servir directement pour l'initialisation de la connexion. Pour cela, il nous faut donc retourner la socket par référence.

Comme dans les exemples précédents, nous avons choisis de faire le démarrage de la connexion avec un envoi de bienvenue. Cet appel se fera depuis le serveur *via* la fonction membre `start()` (3). Dans cette fonction, nous assignons le message à la variable membre, puisque le buffer doit être accessible pendant toute la durée d'une opération asynchrone. La fonction `handle_write` sera appelée à la fin de l'opération asynchrone `async_write`. (4)

 *Les fonctions membres non statiques ont un paramètre implicite `this` qui correspond en fait à l'instance de la classe concernée par la fonction membre. C'est pourquoi dans le `boost::bind` nous devons préciser à quelle instance de classe `tcp_connection` nous appliquons la fonction membre `handle_write`.*

classe `tcp_connection`, premier essai

```
class tcp_connection
{
public:
    tcp_connection(boost::asio::io_service& io_service) // (1)
```

classe tcp_connection, premier essai

```
: m_socket(io_service)
{
}

tcp::socket& socket() // (2)
{
    return m_socket;
}

void start() // (3)
{
    m_message = "Bienvenue sur le serveur!";

    boost::asio::async_write(m_socket, boost::asio::buffer(m_message), // (4)
        boost::bind(&tcp_connection::handle_write, this,
            boost::asio::placeholders::error)
    );
}

private:

void handle_write(const boost::system::error_code& error)
{
    if (!error)
    {
        // Autres actions éventuelles
    }
}

tcp::socket m_socket;
std::string m_message;
};
```

Voyons maintenant le code du serveur. Il doit écouter et accepter les connexions des clients. Lorsqu'un client se connecte, il charge *tcp_connection* de gérer le reste de la connexion et relance immédiatement une écoute pour les nouveaux clients potentiels. Il n'attend pas que *tcp_connection* ait terminé, c'est l'avantage du mode asynchrone.

Le constructeur de *tcp_server* (1) doit prendre en paramètre un *io_service* et un numéro de port. On peut donc initialiser notre *acceptor* avec et lancer une écoute avec la fonction *start_accept()*.

Dans la fonction *start_accept()*, on crée une nouvelle connexion TCP (2) et on attend une connexion entrante sur la socket de la connexion TCP fraîchement créée (3).

Une fois le client connecté, l'opération asynchrone appelle la fonction *callback* que l'on avait précisée, c'est-à-dire *handle_accept()* (4). S'il n'y a pas d'erreurs, on affiche que l'on a reçu un client, on fait appel à *tcp_connection::start()* vu précédemment pour qu'elle gère la suite de la connexion, c'est-à-dire envoyer un message de bienvenue.

Très important ensuite, on relance l'écoute de nouvelles connexions client avec *start_accept()* (5).

classe tcp_server, premier essai, code non fonctionnel

```
class tcp_server
{
public:
    tcp_server(boost::asio::io_service& io_service, int port) // (1)
        : m_acceptor(io_service, tcp::endpoint(tcp::v4(), port))
    {
        start_accept();
    }

private:
    void start_accept() // ATTENTION : ce code ne fonctionne pas, voir la suite... (6)
    {
        tcp_connection new_connection(m_acceptor.io_service()); // (2)
    }
};
```

classe tcp_server, premier essai, code non fonctionnel

```
m_acceptor.async_accept(new_connection.socket(), // (3)
    boost::bind(&tcp_server::handle_accept, this, new_connection,
    boost::asio::placeholders::error));
}

void handle_accept(tcp_connection& new_connection, const boost::system::error_code& error) // (4)
{
    if (!error)
    {
        std::cout << "Reçu un client!" << std::endl;
        new_connection.start();
        start_accept(); // (5)
    }
}

tcp::acceptor m_acceptor;
};
```

Ce code donne bien l'idée générale, mais ne fonctionne pas. Pourquoi ? Eh bien, dans la fonction *start_accept()* du serveur, la connexion est créée sur la pile et va donc disparaître dès que l'on va sortir de la fonction. En résumé, elle va être détruite à peine construite et le programme va planter lamentablement. La solution, c'est d'allouer *tcp_connection* sur le tas, via un *new*. Un nouveau problème se pose à nous : dès lors, qui va libérer la mémoire ? Certainement pas le serveur puisqu'il ne "possède" pas la connexion. Et la connexion pouvant lancer tout un tas d'opérations asynchrones, elle ne saurait être responsable d'elle même...

C'est là qu'interviennent les pointeurs intelligents. *boost::shared_ptr* pour être plus précis. Il va nous permettre de garder en vie la connexion tant qu'il reste des opérations asynchrones à réaliser. Et lorsque le compteur de référence s'apprête à devenir nul, la ressource est libérée. Découvrons cette implémentation dans la section suivante.

IV-E-2 - Avec des pointeurs intelligents

On va retrouver quasiment le même code, avec quelques modifications.

On ne peut pas faire les choses à moitié, il faut qu'on force l'utilisateur de la classe *tcp_connection* à la manipuler avec des pointeurs intelligents. Pour cela, on fait hériter de *boost::enable_shared_from_this* afin de pouvoir réaliser l'appel asynchrone d'écriture en augmentant le comptage de référence (2). Le constructeur de la classe est déclaré privé (3) et une fonction statique *tcp_connection::create()* va retourner une instance tout neuve enveloppée dans un *shared_ptr*.

tcp_connection, Version finale

```
using boost::asio::ip::tcp;

class tcp_connection : public boost::enable_shared_from_this<tcp_connection>
{
public:
    typedef boost::shared_ptr<tcp_connection> pointer;

    static pointer create(boost::asio::io_service& ios) // (1)
    {
        return pointer(new tcp_connection(ios));
    }

    tcp::socket& socket()
    {
        return m_socket;
    }

    void start()
    {
        m_message = "Bienvenue sur le serveur!";
    }
};
```


tcp_connection, Version finale

```

boost::asio::async_write(m_socket, boost::asio::buffer(m_message),
    boost::bind(&tcp_connection::handle_write, shared_from_this(), // (2)
        boost::asio::placeholders::error)
    );
}

private:
tcp_connection(boost::asio::io_service& io_service) // (3)
    : m_socket(io_service)
{
}

void handle_write(const boost::system::error_code& error)
{
    if (!error)
    {
        // Autres actions éventuelles
    }
}

tcp::socket m_socket;
std::string m_message;
};

```

Le code du serveur n'aura presque pas changé. Les seules fonctions qui changent, c'est le *start_accept()*, et le *handle_accept()*. La nouvelle connexion est faite via la fonction *create()* (1). Et maintenant, on est garanti que notre instance ne sera pas détruite à la fin de cette fonction. Qui plus est, cette instance ne sera détruite qu'à la fin de vie de la connexion!

```

void start_accept()
{
    tcp_connection::pointer new_connection = tcp_connection::create(m_acceptor.io_service());

    m_acceptor.async_accept(new_connection->socket(),
        boost::bind(&tcp_server::handle_accept, this, new_connection,
            boost::asio::placeholders::error));
}

void handle_accept(tcp_connection::pointer new_connection, const boost::system::error_code&
    error) // (4)
{
    if (!error)
    {
        std::cout << "Reçu un client!" << std::endl;
        new_connection->start();
        start_accept(); // (5)
    }
}

```

IV-E-3 - Intégration d'un timer

Si on lance une connexion qui écoute après avoir écrit le message de bienvenue, il peut être intéressant de fixer un temps d'attente maximal au delà duquel on ferme la connexion... Pour cela nous allons intégrer un timer.

Il faut créer une nouvelle fonction *do_read* (1) qui va écouter les données entrantes sur la socket. On lance cette fonction juste après la fin de l'envoi du message de bienvenue (2). La fonction va lancer une écoute asynchrone (3). Juste derrière on reparamètre le timer (4) et on le lance (5).

Si la lecture réussit, on relance une autre lecture (6), sinon on ferme (7). Si le temps d'attente dépasse 5 secondes pour la lecture, alors on ferme la socket (7).

Le timer permet de fermer proprement la connexion en cas d'attente trop longue. Ce système peut s'appliquer dans beaucoup d'autres cas comme l'attente de connexion, etc...

```
class tcp_connection : public boost::enable_shared_from_this<tcp_connection>
{
public:
    typedef boost::shared_ptr<tcp_connection> pointer;

    static pointer create(boost::asio::io_service& ios)
    {
        return pointer(new tcp_connection(ios));
    }

    tcp::socket& socket()
    {
        return m_socket;
    }

    void do_read() // (1)
    {
        // On lance une écoute
        boost::asio::async_read(m_socket, boost::asio::buffer(m_buffer), // (3)
            boost::bind(&tcp_connection::handle_read, shared_from_this(),
                boost::asio::placeholders::error)
            );
        timer.expires_from_now(boost::posix_time::seconds(5)); // (4)
        timer.async_wait(boost::bind(&tcp_connection::close, shared_from_this() )); // (5)
    }

    void start()
    {
        m_message = "Bienvenue sur le serveur!";

        boost::asio::async_write(m_socket, boost::asio::buffer(m_message),
            boost::bind(&tcp_connection::handle_write, shared_from_this(),
                boost::asio::placeholders::error)
            );
    }

private:
    tcp_connection(boost::asio::io_service& io_service)
        : m_socket(io_service),
          timer(io_service, boost::posix_time::seconds(5)) // Commence à compter dès sa création
    {
    }

    void handle_write(const boost::system::error_code& error)
    {
        if (!error)
        {
            do_read(); // (2)
        }
        else {
            std::cout << error.message() << std::endl;
        }
    }

    void handle_read(const boost::system::error_code& error) // (6)
    {
        if (!error)
        {
            // On réécoute
            do_read();
        }
        else
        {
            close();
        }
    }
}
```

```
void close() // (7)
{
    m_socket.close();
}

boost::asio::deadline_timer timer;
tcp::socket m_socket;
std::string m_message;
boost::array<char, 128> m_buffer;
};
```

Télécharger le code serveur avec timer

IV-F - Exemple d'un client asynchrone

On va dans cette partie réaliser un client asynchrone travaillant avec le serveur temporisé vu précédemment. Le code va ressembler beaucoup à ce que nous avons déjà vu auparavant.

Le *main* ci-dessous ne devrait pas poser problème, nous lançons un client pour se connecter sur le serveur.

```
main_client.cpp

#define _WIN32_WINNT 0x0501

#include "tcp_client.h"

int main()
{
    try
    {
        boost::asio::io_service io_service;
        tcp::endpoint endpoint(boost::asio::ip::address::from_string("127.0.0.1"), 7171);

        tcp_client client(io_service, endpoint);
        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

Ensuite vient *tcp_connection*, qui va être légèrement modifiée par rapport à celui du serveur.

En effet, nous avons maintenant une fonction *read* (1) qui va réaliser une lecture asynchrone sur la socket. Notez qu'on peut récupérer le nombre d'octets reçus avec *boost::asio::placeholders::bytes_transferred*.

Le principal problème ici, c'est qu'en faisant appel à un transfert complet via *async_read*, l'appel n'aboutit que lorsque le buffer de réception est rempli. Autrement dit ici, il ne se passera rien. Nous avons deux solutions pour résoudre ce problème : passer en transfert court, ou bien spécifier à l'appel synchrone que nous ne voulons pas forcément remplir tout notre buffer. Cette dernière solution a été retenue dans notre cas en spécifiant *boost::asio::transfer_at_least* pour imposer de recevoir au moins 20 octets (2). Passé ces vingt octets, l'appel asynchrone peut appeler le *handler* correspondant (3), quand le transfert est terminé pour lui.



La solution la plus "propre" serait de d'abord transférer la taille du message puis d'envoyer le message lui-même. Cette technique sera illustrée plus tard dans le projet chat.

S'il n'y a pas d'erreurs (4), on affiche le message. Sinon on affiche le message d'erreur (5).

tcp_connection.h, client

```
using boost::asio::ip::tcp;

class tcp_connection : public boost::enable_shared_from_this<tcp_connection>
{
public:
    typedef boost::shared_ptr<tcp_connection> pointer;

    static pointer create(boost::asio::io_service& ios)
    {
        return pointer(new tcp_connection(ios));
    }

    tcp::socket& socket()
    {
        return m_socket;
    }

    void read() // (1)
    {
        boost::asio::async_read(m_socket, boost::asio::buffer(m_network_buffer),
            boost::asio::transfer_at_least(20), // (2)
            boost::bind(&tcp_connection::handle_read, shared_from_this(),
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred)
            );
    }

private:
    tcp_connection(boost::asio::io_service& io_service)
        : m_socket(io_service)
    {
    }

    void handle_read(const boost::system::error_code& error, size_t number_bytes_read) // (3)
    {
        if (!error) // (4)
        {
            std::cout.write(&m_network_buffer[0], number_bytes_read);
            read();
        }
        else {
            std::cout << error.message() ; // (5)
        }
    }

    tcp::socket m_socket;
    boost::array<char, 128> m_network_buffer;
};
```

Voyons maintenant le code du client TCP.

Ici encore, pas grand chose de complètement nouveau! Le constructeur de *tcp_client* lance la fonction membre *connect* pour tenter de se connecter au serveur (1).

La fonction *connect* commence par créer une nouvelle connexion (2), puis récupère la socket sous-jacente pour lancer une connexion asynchrone sur le serveur (3).

Lorsque la connexion est établie, le *handle_connect* (4) lance l'opération de lecture de données en provenance du serveur.

tcp_client.h

```
class tcp_client
{
public:
```

tcp_client.h

```
tcp_client(boost::asio::io_service& io_service, tcp::endpoint& endpoint)
:m_io_service (io_service)
{
    // On tente de se connecter au serveur // (1)
    connect(endpoint);
}

private:

void connect(tcp::endpoint& endpoint)
{
    tcp_connection::pointer new_connection = tcp_connection::create(m_io_service); // (2)
    tcp::socket& socket = new_connection->socket();
    socket.async_connect(endpoint, // (3)
        boost::bind(&tcp_client::handle_connect, this,
            new_connection,
            boost::asio::placeholders::error)
        );
}

void handle_connect(tcp_connection::pointer new_connection, const boost::system::error_code&
error) // (4)
{
    if (!error)
    {
        new_connection->read();
    }
}

boost::asio::io_service& m_io_service;
};
```

Ce code fonctionne correctement avec le serveur de la section précédente. C'est-à-dire que le client va recevoir un message provenant du serveur et va continuer à écouter. Le serveur écoute aussi et va fermer la connexion au bout de 5 secondes s'il n'a rien reçu... C'est ce qui va se passer : le client va être déconnecté !

Télécharger le code client asynchrone

V - Le protocole UDP

Le protocole UDP s'utilise en mode non connecté. Il est plus rapide, mais moins fiable que le protocole TCP. on retrouve les opérations réseau *send_to* et *receive_from* du standard POSIX.

On peut toutefois utiliser un mode "pseudo-connecté", avec les fonctions de transferts courts des sockets. Par contre, il n'existe pas de transfert complet en UDP dans Boost.Asio. C'est impossible à cause de l'ordre d'arrivée des paquets qui n'est pas garanti, pas plus que la bonne réception des données.

V-A - Exemple d'un client synchrone

Comme vous pouvez le constater, le début du code est similaire à celui rencontré pour le protocole TCP : On définit un point d'arrivée (*endpoint*), soit en lui passant directement d'adresse, soit en utilisant un résolveur. Ici j'ai choisi l'adresse locale 127.0.0.1.

On crée ensuite une socket UDP pour l'envoi et la réception de données sous forme de datagram (1). La socket peut envoyer n'importe où, elle n'est **pas connectée**, donc on lui attribue juste le standard d'adresses avec lequel elle va travailler (IPv4 ici).

On crée un mini buffer vide d'un octet qui va nous servir à donner notre adresse au serveur pour qu'il puisse nous répondre (2). On envoie ensuite ce petit octet au serveur (3).

Le serveur va bientôt nous répondre, on alloue un buffer de réception plus conséquent (128 octets) (4). Puis on s'apprête à recevoir des données en provenance de n'importe qui (5). Notez qu'on récupère le nombre d'octets lus et le *endpoint* de l'envoyeur...

Enfin, on écrit les données reçues sur le flux de sortie standard (6).

```
#include <iostream>
#include <boost/array.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::udp;

int main()
{
    try
    {
        boost::asio::io_service io_service;

        udp::endpoint receiver_endpoint (boost::asio::ip::address::from_string("127.0.0.1"), 7171);

        udp::socket socket(io_service); // (1)
        socket.open(udp::v4());

        boost::array<char, 1> send_buf = { 0 }; // (2)
        socket.send_to(boost::asio::buffer(send_buf), receiver_endpoint); // (3)

        boost::array<char, 128> recv_buf; // (4)
        udp::endpoint sender_endpoint;
        size_t len = socket.receive_from(boost::asio::buffer(recv_buf), sender_endpoint); // (5)

        std::cout.write(recv_buf.data(), len); // (6)
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```

Cet exemple tout simple montre bien l'utilisation différente du mode non connecté avec UDP par rapport au mode connecté avec TCP. On envoie à n'importe qui et on reçoit de n'importe où.

V-B - Exemple d'un serveur synchrone

Comme le client, le serveur n'a rien d'exceptionnel, il attend un message d'un octet en provenance de n'importe qui, puis renvoie un message à cette adresse. Toujours sans notion de connexion, bien entendu!

On commence par créer une socket UDP qui écoute sur le port 7171 pour une adresse IPv4 (1).

On reçoit à partir de ce type d'adresse un octet (2), qu'on ignore. Par contre, on stocke bien soigneusement l'adresse IP (*endpoint*) de l'envoyeur pour pouvoir lui répondre (3).

Si l'erreur est due à un message trop long, ce n'est pas grave. On ignore le reste et on continue le programme. Par contre, toute autre erreur lève une exception (4).

On crée ensuite un message de bienvenue qu'on envoie à l'adresse IP précédemment reçue (5). Les erreurs sont ignorées dans ce cas.

```
int main()
{
```

```
try
{
    boost::asio::io_service io_service;

    udp::socket socket(io_service, udp::endpoint(udp::v4(), 7171)); // (1)

    while (1)
    {
        boost::array<char, 1> recv_buf; // (2)
        udp::endpoint remote_endpoint;
        boost::system::error_code error;
        socket.receive_from(boost::asio::buffer(recv_buf), remote_endpoint, 0, error); // (3)

        if (error && error != boost::asio::error::message_size) // (4)
            throw boost::system::system_error(error);

        std::string message = "Bienvenue sur le serveur ! Mode non connecté.";

        boost::system::error_code ignored_error;
        socket.send_to(boost::asio::buffer(message), remote_endpoint, 0, ignored_error); // (5)
    }
}
catch (std::exception& e)
{
    std::cerr << e.what() << std::endl;
}

return 0;
}
```

Si le mode non connecté est fondamentalement différent du mode connecté, la manière de gérer l'UDP et le TCP se ressemble fortement avec Boost.Asio. Ainsi, il ne sera pas difficile pour le lecteur de réaliser un serveur asynchrone à partir des exemples TCP qu'on peut trouver dans la section précédente.

VI - Les sockets iostreams

tcp::iostream est une classe de Boost.Asio qui implémente les iostreams comme surcouche des sockets. Plus de résolution de noms DNS, plus de problèmes de protocole. Bref, une classe très simple à utiliser.

VI-A - Exemple de client synchrone

On crée une classe iostream que l'on affecte à l'IP locale, sur le port 7171.

On utilise *std::getline* (comme pour les flux standard) pour récupérer dans une chaîne de caractère ce qui provient de *tcp::iostream*...

Et c'est tout!

```
#include <iostream>
#include <string>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

int main()
{
    tcp::iostream s("127.0.0.1", "7171"); // (1)
    std::string line;
    std::getline(s, line); // (2)
    std::cout << line << std::endl;

    return 0;
}
```

Ce code extrêmement simple fonctionne très bien avec les serveurs synchrones et asynchrones TCP que l'on a pu construire jusqu'ici. On récupère bien "Bienvenue sur le serveur !"

Mais évidemment on n'a pas autant de contrôle qu'en choisissant nous-mêmes le protocole, l'architecture, etc... Tout de même, avec quelques lignes de code, on peut réaliser d'étonnantes opérations relativement complexes.

VI-B - Exemple de serveur synchrone

`tcp::iostream` permet également de créer des serveurs très simples. Voici un serveur synchrone équivalent à tout ce qu'on a vu précédemment.

Comme d'habitude, le serveur prend tout type d'adresse IPv4.

L'*acceptor* accepte la connexion d'un client (2).

rdbuf() renvoie un pointeur sur le *streambuf* sous-jacent, c'est-à-dire *basic_socket_streambuf* ici (voir [diagramme des classes](#)). L'acceptor reçoit donc un paramètre valide !

On peut donc enfin envoyer un message via l'opérateur de flux standard.

```
int main()
{
    boost::asio::io_service io_service;

    tcp::endpoint endpoint(tcp::v4(), 7171); // (1)
    tcp::acceptor acceptor(io_service, endpoint);

    while (1)
    {
        tcp::iostream stream;
        acceptor.accept(*stream.rdbuf()); // (2)
        stream << "Bienvenue sur le serveur ! " << std::endl;
    }

    return 0;
}
```

VII - Projet démo : réalisation d'un 'chat' avec Boost.Asio

Le but de cette section est de réaliser un programme de *chat* (genre MSN) assez générique en utilisant des principes de bonne programmation. Nous allons mettre en œuvre une bonne partie de ce que nous avons vu jusqu'à présent côté réseau, en insistant encore un peu sur le côté généricité et abstraction des données. Nous allons pour cela utiliser d'autres excellentes bibliothèques de boost : sérialisation, pointeurs intelligents...

VII-A - Cahier des charges

Le cahier des charges ressemble beaucoup à ce que fût une des premières versions de MSN Messenger. Le programme devra comporter un serveur vers lequel les clients peuvent se connecter. Par défaut, tous les nouveaux clients arrivent dans une salle de chat unique et peuvent ainsi se parler directement. Les participants de la salle de chat devront être averti de l'arrivée au départ d'un tiers.

VII-B - Conception

Décomposons le problème proprement : analyse descendante, puis conception ascendante!

VII-B-1 - Abstractions

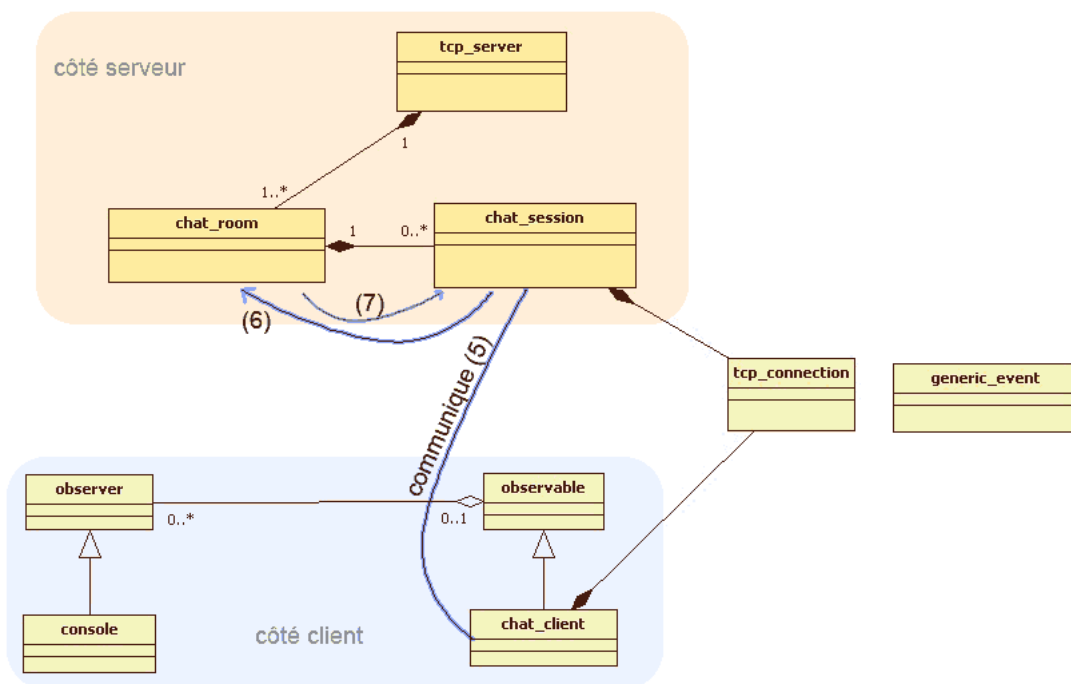
Faisons ressortir les abstractions de notre cahier des charges. On va bien sûr retrouver des éléments de notre exemple TCP de la partie IV.

- chat_client : client se connectant au serveur
- chat_server : Serveur pour gérer les connexions entrantes
- chat_room : salle de chat pour les participants
- chat_session : représente une connexion côté serveur
- tcp_connection : gère la connexion, écriture lecture sur une socket.
- chat_message : structure pour échanger des messages et des informations

Illustrons un peu ces abstractions à l'aide d'un petit schéma.

VII-B-2 - Architecture

Voici donc un schéma UML mélangeant un peu diagramme de classe et Use Case pour essayer d'être le plus clair possible sans faire 15 schémas. On a en haut le côté serveur, programme à part qui accepte les nouvelles connexions. En bas le côté client avec un programme à part également. La communication entre le client et le serveur se fait exclusivement par le réseau.



Projet chat en image

Détaillons un peu les étapes clefs du *chat*

- 1 chat_client se connecte à chat_server : c'est une demande de connexion.
- 2 chat_server accepte et crée une chat_session côté serveur. C'est désormais avec chat_session que chat_client communiquera.
- 3 chat_server ajoute le chat_session dans chat_room, il en perd le contrôle. Seul la room est responsable de ses participants.
- 4 chat_server reprend l'écoute de nouveaux clients.
- 5 Un chat_client envoie un message par le réseau, réceptionné par chat_session.
- 6 chat_session le transmet à chat_room
- 7 chat_room le diffuse à tous ses participants (y compris lui même)

8 Tous les chat_client reçoivent le message (5).

Le principe est assez simple, voyons comment le mettre en œuvre avec Boost.Asio et Boost.Serialization.

VII-C - Implémentation du serveur

VII-C-1 - Classe chat_message

On va commencer simple, cette classe, ou structure plutôt, est le message que l'on va faire transiter par le réseau. Elle va contenir des champs de données divers suivant nos envies.

Dans notre cas, on va y voir figurer :

- 1 le type de message : nouveau message, nouveau client connecté, etc.
- 2 une chaîne de caractère pour le message
- 3 une chaîne de caractère pour le login de l'envoyeur
- 4 une liste de chaîne de caractère pour récupérer toutes les personnes connectées (non utilisée ici, mais pourrait l'être si on pousse le projet plus loin...)

chat_message.h

```
class chat_message
{
public:

    void reset()
    {
        m_list_string.clear();
        m_message.clear();
        m_login.clear();
    }

    int m_type; // (1) Type d'événement : NEW_MSG, etc.

    // Generic datas
    std::list<std::string> m_list_string; // (4)

    std::string m_message; // (2)
    std::string m_login; // (3)

    template<class Archive>
    void serialize(Archive& ar, const unsigned int version){
        ar & m_type & m_list_string & m_message & m_login;
    }

    enum {
        NEW_MSG = 0, // Nouveau message
        PERSON_LEFT = 1, // Information : personne ayant quittée la room
        PERSON_CONNECTED = 2, // Information : nouvelle personne connectée à la room
    };
};
```

VII-C-2 - Classe tcp_connection

La classe tcp_connection va encapsuler l'envoi et la réception de données via une socket. Nous avons pu remarquer que nous écrivions souvent dans le cadre d'application réseau :

```
boost::async_read(socket, boost::asio::buffer(network_buffer),
    boost::bind(&ma_classe::handler, this,
    boost::asio::placeholders::error)
```


```
);
```

Il existe plusieurs inconvénients à utiliser toujours cette même méthode pour lire / écrire des données. D'abord on est toujours obligé d'écrire cette longue ligne de code juste parce que le *callback* a changé. Ensuite, comment allouer un buffer de réception lorsqu'on ne connaît pas la taille des données qui vont arriver ?

Le but serait d'encapsuler tous ces appels dans une abstraction suffisamment haute pour qu'elle couvre toutes nos "variations". Nos principales variations, ce sont les messages et leurs tailles (il ne faut pas qu'on dépende de la structure) et le *callback*.

Pour le *callback*, je vois deux solutions : une dynamique (`boost::function`) et une statique (les *template*). On aura un net gain de performance à utiliser les *template*, c'est la solution que j'ai choisie ici.

Ensuite, la taille des données. Comme nous effectuons une sérialisation, nous avons besoin de connaître la taille des données à récupérer. La solution est d'envoyer la taille des données suivant un format bien précis, avant d'envoyer les données elle-même pour l'envoi. En ce qui concerne la réception, il faut d'abord lire la taille des données à recevoir avant d'allouer le buffer de réception et de demander à recevoir les données. Bien entendu, nous devons utiliser les transferts complets (à l'aide `boost::asio::async_read` et `boost::asio::async_write`).

 *Le code est relativement long, surtout à cause du fait que `boost::bind` ne supporte pas l'appel récursif lorsqu'il est templaté. C'est-à-dire que `boost::bind` ne peut pas (pour l'instant sans doute) binder une fonction passée via un *template* où `boost::bind` est utilisé. Du coup, on est obligé de passer par des tuple, qui n'ont normalement rien à voir là dedans. Je les ai gardés pour avoir un code qui compile. Faites en abstraction et gardez à l'esprit que le tuple en question est juste le *callback*.*

Le constructeur de `tcp_connection` prend un *io_service* comme seul et unique paramètre, qui sert à initialiser une socket. Du côté des variables privées, on retrouve la socket (1) et la taille du header (2). Le reste étant des variables membres (donc à durée de vie aussi longue que la durée de vie de l'instance de la classe) pour pouvoir effectuer des opérations asynchrones dessus.

Côté fonction membre, les deux fonctions principales sont *async_write* et *sync_read*.

la fonction membre *aync_read* (3) prend deux paramètres : une structure à remplir et la fonction de *callback* à appeler à la fin du remplissage. Elles sont toutes les deux templâtées, si bien que la seule condition pour la structure, c'est d'être sérialisable au sens de boost, c'est-à-dire implémenter la fonction *serialize*. La classe `tcp_connection` fait donc le minimum d'hypothèse sur les structures de données utilisées.

Dans la fonction en elle-même, on va retrouver une sérialisation des données (4). On écrit ensuite la taille des données dans un header au format fixe (5). Puis on envoie le header suivi des données dans le réseau, en appelant le *callback* passé en paramètre lorsque le dernier *async_write* (6) sera terminé.

La fonction *async_read* suit à peu près le même schéma. On commence par récupérer le header (10). S'il n'y a pas d'erreurs, on lit la taille du message à récupérer (11). On change la taille du buffer de réception puis on lance une lecture du message derrière (12).

Dans le handler après lecture (13), on désérialise les données (14), on affecte le résultat de la désérialisation à la structure passé en paramètre d'entrée. On termine en appelant le *callback* passé en paramètre de la fonction *async_read* (15).

tcp_connection.h

```
class tcp_connection
{
public:
    tcp_connection(boost::asio::io_service& io_service)
        : m_socket(io_service)
    {
    }
}
```

tcp_connection.h

```
boost::asio::ip::tcp::socket& socket()
{
    return m_socket;
}

// Ecriture asynchrone sur le socket
template <typename T, typename Handler>
void async_write(const T& t, Handler handler) // (3)
{
    // On sérialise. (4)
    std::ostream archive_stream;
    boost::archive::text_oarchive archive(archive_stream);
    archive << t;
    m_outbound_data = archive_stream.str();

    // On écrit un header. // (5)
    std::ostream header_stream;
    header_stream << std::setw(header_length)
        << std::hex << m_outbound_data.size();
    if (!header_stream || header_stream.str().size() != header_length)
    {
        // En cas de problème, on informe l'appelant.
        boost::system::error_code error(boost::asio::error::invalid_argument);
        m_socket.io_service().post(boost::bind(handler, error));
        return;
    }
    m_outbound_header = header_stream.str();

    // On écrit les données sérialisées dans le socket. (6)
    std::vector<boost::asio::const_buffer> buffers;
    buffers.push_back(boost::asio::buffer(m_outbound_header));
    buffers.push_back(boost::asio::buffer(m_outbound_data));
    boost::asio::async_write(m_socket, buffers, handler); // (7)
}

// Lecture asynchrone de données depuis le socket
template <typename T, typename Handler>
void async_read(T& t, Handler handler)
{
    // On récupère le header (10)
    void (tcp_connection::*f)(const boost::system::error_code&, T&, boost::tuple<Handler>)
        = &tcp_connection::handle_read_header<T, Handler>;
    boost::asio::async_read(m_socket, boost::asio::buffer(m_inbound_header),
        boost::bind(f,
            this, boost::asio::placeholders::error, boost::ref(t),
            boost::make_tuple(handler)));
}

//Interprétation du header
template <typename T, typename Handler>
void handle_read_header(const boost::system::error_code& e,
    T& t, boost::tuple<Handler> handler)
{
    if (e)
    {
        boost::get<0>(handler)(e);
    }
    else
    {
        // Détermine la longueur du vrai message (11)
        std::istream is(std::string(m_inbound_header, header_length));
        std::size_t m_inbound_datasize = 0;
        if (!(is >> std::hex >> m_inbound_datasize))
        {
            // Header non valide, on informe la fonction appelante
            boost::system::error_code error(boost::asio::error::invalid_argument);
            boost::get<0>(handler)(error);
            return;
        }

        // On récupère les données (12)
    }
}
```

tcp_connection.h

```

m_inbound_data.resize(m_inbound_datasize);
void (tcp_connection::*f)(const boost::system::error_code&, T&, boost::tuple<Handler>)
    = &tcp_connection::handle_read_data<T, Handler>;

boost::asio::async_read(m_socket, boost::asio::buffer(m_inbound_data),
    boost::bind(f, this,
        boost::asio::placeholders::error, boost::ref(t), handler));
}
}

// Les données reçues, on les déserialise (13)
template <typename T, typename Handler>
void handle_read_data(const boost::system::error_code& e,
    T& t, boost::tuple<Handler> handler)
{
    if (e)
    {
        boost::get<0>(handler)(e);
    }
    else
    {
        // On extrait (14)
        try
        {
            std::string archive_data(&m_inbound_data[0], m_inbound_data.size());
            std::istringstream archive_stream(archive_data);
            boost::archive::text_iarchive archive(archive_stream);
            archive >> t;
        }
        catch (std::exception& e)
        {
            // En cas d'échec
            boost::system::error_code error(boost::asio::error::invalid_argument);
            boost::get<0>(handler)(error);
            return;
        }

        // On informe l'appelant que tout s'est bien passé. (15)
        boost::get<0>(handler)(e);
    }
}

private:
// le socket membre sous jacent.
boost::asio::ip::tcp::socket m_socket; // (1)

// Taille de l'header.
enum { header_length = 8 }; // (2)

std::string m_outbound_header;
std::string m_outbound_data;
char m_inbound_header[header_length];
std::vector<char> m_inbound_data;
};

typedef boost::shared_ptr<tcp_connection> connection_ptr;

#endif

```

On dispose maintenant d'un code indépendant de la structure des messages. *tcp_connection* peut envoyer et recevoir simplement des données. La fonction appelant est informé via un *callback* fournit en paramètre par le développeur lui même !

Exemple simple de réception d'un message réseau:

```

chat_message msg;
m_tcp_connection->async_read(msg, my_handler);

```

Dans lequel `my_handler` est un *callback* défini par nos soins. Quoi de plus naturel de s'affranchir dans nos programmes des sérialisations / désérialisations et récupérer directement le message sous la structure attendue!

VII-C-3 - Classe `chat_server`

Le serveur a pour rôle d'accepter les connexions entrantes. Découvrons son interface.

Le constructeur du serveur prend pour paramètres un *io_service* et un *endpoint* (1).

La fonction principale est d'attendre les connexions entrantes (2), puis de les accepter grâce à l'acceptor (5) dans le *handler* (3).

On notera que `chat_server` possède une chambre de chat, sous forme de *shared_ptr* (6). Il est responsable de sa durée de vie.

chat_server.h

```
using boost::asio::ip::tcp;

class chat_server
{
public:
    chat_server(boost::asio::io_service& io_service, const tcp::endpoint& endpoint); // (1)

    void wait_for_connection (); // (2)

private:
    void handle_accept (const boost::system::error_code& error, connection_ptr); // (3)

    boost::asio::io_service& m_io_service; // (4)
    tcp::acceptor m_acceptor; // (5)
    chat_room_ptr m_room; // (6)
};
```

L'interface est relativement simple et l'implémentation aussi!

On lance une écoute dans le constructeur (1). On commence par créer une connexion TCP (2) vue précédemment, puis on attend une nouvelle connexion en asynchrone via la socket de la connexion fraîchement créée (3).

Lorsqu'un client se connecte, le *callback* (4) est appelé. On crée une session qui sera responsable de l'échange des données *via* le réseau avec le client (5).

On ajoute cette session dans la salle de chat (6). Puis on réécoute à nouveau les connexions entrantes (7).

char_server.cpp

```
#include "chat_server.h"
#include "chat_session.h"

chat_server::chat_server(boost::asio::io_service& io_service, const tcp::endpoint& endpoint)
:m_io_service(io_service),
m_acceptor(io_service, endpoint),
m_room(new chat_room(*this))
{
    std::cout << "Creation d'un serveur " << std::endl;

    wait_for_connection(); // (1)
}

// Attente d'un nouveau client
void chat_server::wait_for_connection()
```

char_server.cpp

```
{
    connection_ptr new_connection(new tcp_connection(m_io_service)); // (2)

    // Attente d'une nouvelle connection
    m_acceptor.async_accept(new_connection->socket(), // (3)
        boost::bind(&chat_server::handle_accept, this,
            boost::asio::placeholders::error,
            new_connection)
        );
}

void chat_server::handle_accept(const boost::system::error_code& error, connection_ptr
    new_connection) // (4)
{
    if (!error)
    {
        std::cout << "Connection acceptée" << std::endl;
        chat_session_ptr session = chat_session::create(new_connection, m_room); // (5)
        m_room->join(session); // (6)
        wait_for_connection(); // (7)
    }
    else {
        std::cerr << "Connection refusée" << std::endl;
    }
}
```

Le code de *chat_server* est relativement simple, la gestion de la connexion avec le client va s'effectuer à travers *chat_session*, que nous découvrir sans attendre dans la prochaine section.

VII-C-4 - Classe chat_session

Elle représente la connexion côté serveur, avec le client. Découvrons tout d'abord son interface.

Parmi les fonctions membres, nous retrouvons du connu : une fonction *create()* pour renvoyer un *shared_ptr* d'une nouvelle instance. On oblige ainsi *chat_session* à être manipulée par des pointeurs intelligents. On en profite également pour lancer l'écoute (la fonction *wait_for_data()*). En effet, la fonction *wait_for_data()* nécessite *un shared_from_this()* dans l'appel asynchrone. Nous ne pouvons donc l'inclure directement dans le constructeur, mais bien passer par une sorte de *factory*.

On retrouve également une fonction *deliver* (2) pour envoyer un message au client.

La session possède un lien vers sa salle de chat (3), en *weak_ptr* parce qu'elle n'en est pas responsable.

chat_session.h

```
using boost::asio::ip::tcp;

class chat_server;

class chat_session
    : public boost::enable_shared_from_this<chat_session>
{
public:
    ~chat_session();

    static chat_session_ptr create(connection_ptr tcp_connection, chat_room_ptr room) // (1)
    {
        chat_session_ptr session (new chat_session(tcp_connection, room));
        session->wait_for_data();
        return session;
    }
}
```

chat_session.h

```
void deliver (const chat_message& msg); // (2)

private:
chat_session(connection_ptr tcp_connection, chat_room_ptr room);
void wait_for_data ();


void handle_write (const boost::system::error_code& error);
void handle_read (const boost::system::error_code& error);

connection_ptr    m_tcp_connection;
chat_room_wptr    m_room; // (3)
chat_message      m_message;

bool              is_leaving;
};

typedef boost::shared_ptr<chat_session> chat_session_ptr;
```

Nous disposons ici d'un code avec une interface très simple, voyons maintenant l'implémentation.

 *Grâce à l'effort fourni pour encapsuler la connexion dans `tcp_connection`, vous noterez la simplicité des appels réseaux désormais, puisqu'on peut jouer directement avec nos structures de message (`chat_message` ici).*

Comme nous l'avons vu quelques lignes au dessus, la fonction `create()` appelle `wait_for_data()` pour commencer l'écoute de données en provenance du client. On utilise pour cela `tcp_connection`, en précisant juste le `chat_message` de réception et le `callback` (`handle_read` ici).

Dans la fonction `handle_read` (2), on récupère un `shared_ptr` sur la salle de chat à partir du `weak_ptr`. Puis on demande à la room de faire passer le message (3). Enfin, on demande pour recevoir à nouveau (4).

Si jamais une erreur survient (dans 99% des cas il s'agit d'une déconnexion du client), alors on quitte la room (5).

La fonction pour acheminer les messages vers le client est relativement simple (6), toujours grâce à notre `tcp_connection`. On demande une écriture via `async_write`. Nul besoin de stocker une variable de "longue durée de vie", puisque c'est `tcp_connection` qui s'en charge. En cas d'erreur, la fonction `callback` (7) nous retire de la room.

On note l'existence d'une variable booléenne `is_leaving`. En effet, nous avons souvent deux opérations asynchrones menées en parallèles : la lecture et l'écriture. En cas de déconnexion du client, il est fort probable que les deux se passent mal. Le mieux est donc de placer un garde fou pour ne pas nous retirer deux fois de la room.

chat_session.cpp

```
#include "chat_session.h"
#include "chat_server.h"

chat_session::chat_session(connection_ptr tcp_connection, chat_room_ptr room)
: m_tcp_connection(tcp_connection),
m_room(room)
{
    is_leaving = false;
    std::cout << "New chat_session ! " << std::endl;
}

chat_session::~chat_session()
{
    std::cout << "Session détruite" << std::endl;
}

void chat_session::wait_for_data() // (1)
{
    // On lance l'écoute d'événements
    m_tcp_connection->async_read(m_message,
        boost::bind(&chat_session::handle_read, shared_from_this(),
```


chat_session.cpp

```

boost::asio::placeholders::error)
);
}

void chat_session::handle_read(const boost::system::error_code &error) // (2)
{
    chat_room_ptr room = m_room.lock();
    if (room)
    {
        if (!error)
        {
            // On demande à la room de transmettre le message à tout le monde
            room->deliver(m_message); // (3)

            // On relance une écoute
            wait_for_data(); // (4)
        }

        else
        {
            if (!is_leaving)
            {
                is_leaving = true;
                room->leave(shared_from_this() ); // (5)
            }
        }
    }
}

void chat_session::deliver(const chat_message& msg) // (6)
{
    m_tcp_connection->async_write(msg,
    boost::bind(&chat_session::handle_write, shared_from_this(),
    boost::asio::placeholders::error)
    );
}

void chat_session::handle_write(const boost::system::error_code &error) // (7)
{
    chat_room_ptr room = m_room.lock();
    if (room && error && (!is_leaving) )
    {
        is_leaving = true;
        room->leave(shared_from_this() );
    }
}

```

`chat_session` communique donc directement avec le client, mais aussi avec la room pour qu'elle passe le message aux autres clients connectés.

VII-C-5 - Classe chat_room

chat_room est une salle de chat contenant des participants. Elle reçoit des messages en provenance d'un participant (un *chat_session*) et le transmet à tout le monde (tous les *chat_session*).

L'interface aurait pu être simplifiée. J'ai choisis de laisser à la room un lien vers le serveur qu'il l'a créée. Ce n'était pas nécessaire dans notre cas, mais en cas d'ajouts de fonctionnalités importantes, ce sera indispensable.

Quoiqu'il en soit, nous avons donc une fonction *join* pour ajouter un participant, une fonction *leave* pour retirer un participant et *deliver* pour passer un message à tous les participants. Les participants sont stockés dans un *set* sous forme de *shared_ptr*.

chat_room.h

```

class chat_session;

```

chat_room.h

```
class chat_server;
typedef boost::shared_ptr<chat_session> chat_session_ptr;

class chat_room
{
public:
    chat_room(chat_server& server); // (1)

    void join (chat_session_ptr participant); // (1)
    void leave (chat_session_ptr participant); // (2)
    void deliver (const chat_message& msg); // (3)

private:
    std::set<chat_session_ptr> m_participants; // (4)
    chat_server& m_server;
};

typedef boost::shared_ptr<chat_room> chat_room_ptr;
typedef boost::weak_ptr<chat_room> chat_room_wptr;
```

L'implémentation est vraiment triviale et ne concerne pas vraiment le réseau.

On notera toutefois la création d'un message d'information lorsqu'un client se connecte (1) ou s'en va (2).

chat_room.cpp

```
#include "chat_room.h"
#include "chat_session.h"
#include "chat_server.h"

chat_room::chat_room(chat_server& server)
:m_server(server)
{
    std::cout << "New room" << std::endl;
}

void chat_room::join(chat_session_ptr participant)
{
    m_participants.insert(participant);

    // On informe les sessions de la room // (1)
    chat_message e;
    e.m_type = chat_message::PERSON_CONNECTED;
    deliver(e);
}

void chat_room::leave(chat_session_ptr participant)
{
    // On informe les sessions de la room // (2)
    chat_message e;
    e.m_type = chat_message::PERSON_LEFT;
    deliver(e);

    m_participants.erase(participant); // puis on le détruit
}

void chat_room::deliver(const chat_message& msg)
{
    std::for_each(m_participants.begin(), m_participants.end(),
        boost::bind(&chat_session::deliver, _1, boost::ref(msg)));
}
```

VII-C-6 - Bilan

Le serveur est prêt. Nous avons créé des classes et affecté des responsabilités à chacune d'entre elles. Le niveau d'abstraction proposé permet de découpler structure de données à envoyer, réseau et gestion de l'application. Boost.Serialization et Boost.Asio fonctionne particulièrement bien ensemble en envoyant sur le réseau n'importe quelle structure sérialisable.

VII-D - Bilan

Le client reste beaucoup plus simple que le serveur. C'est un choix de ne pas le détailler. L'esprit du code est exactement le même, donc la compréhension devrait être assez facile. A noter tout de même, j'utilise un *pattern Observer* pour découpler événements reçus et traitement des événements. En effet, le client transmet à une liste d'observateurs les événements qu'il a reçu. C'est aux observateurs d'implémenter une queue d'événements ou autre structure pour les traiter.

A travers cet exemple de chat, nous avons pu mettre en œuvre des appels réseaux asynchrones avec Boost.Asio, que ce soit en lecture ou en écriture. L'asynchronicité ne nous a pas gêné dans notre développement puisque Boost.Asio nous décharge de la gestion de la concurrence.

Boost.Serialization nous a permis de ne pas se soucier de la structure des données, elle peut être modifiée comme bon nous semble et à n'importe quel moment du développement du projet. Cette souplesse est très appréciable dans du développement réseau.

Enfin, j'ai essayé autant que possible d'utiliser des bons principes de programmation, comme les pointeurs intelligents, le découplage du code avec le *pattern* observateur. Ces bons principes doivent être présents dans tout programme ayant la prétention de devenir robuste et de résister à l'épreuve du temps, et des changements de cahier des charges!

Téléchargez le code source

VIII - Conclusion

Au cours de ce tutoriel, nous avons pu apprécier les nombreuses fonctionnalités proposées par Boost.Asio, à savoir la gestion des opérations synchrones et surtout asynchrones, les timers, le support de plusieurs protocoles réseaux (TCP, UDP, ...).

Nous avons pu voir la puissance et la simplicité des opérations asynchrones mises en place avec Boost.Asio, ne nécessitant pas d'attention particulière pour les accès concurrents. Couplé avec Boost.Serialization, Boost.Asio nous permet d'envoyer et recevoir presque n'importe quoi par le réseau et ce, sans se soucier ni dépendre des structures de données.

Certains aspects comme la communication via un **port série** n'ont pas été traités ici. Cependant, le fonctionnement reste le même puisque les écritures et lectures se font non plus via une socket, mais un port série. Le lecteur ayant compris le fonctionnement des sockets de Boost.Asio ici n'aura aucune difficulté à le transférer sur un port série.

Au final, l'élégance, la puissance et la portabilité de Boost.Asio en font une bibliothèque incontournable pour programmer des applications réseaux en C++.

IX - Remerciements

Un grand merci à **3DArchi**, **ram-0000** et **arzar** pour la relecture et la correction de ce tutoriel.