Création d'un serveur multi-threads en C++

par khayyam90

Date de publication : 24/11/2005

Dernière mise à jour : 21/05/2005

Cet article montre l'architecture d'un serveur multi-threads sous Windows. L'implémentation sera faite en C++ en utilisant la bibliothèque WinSock de Windows. Prérequis : connaissances du C++ et utilisation des sockets.

I - Principe général

Comment fonctionnera notre serveur ?

Le serveur aura plusieurs tâches à exécuter en parallèle : il devra être à l'écoute des clients qui souhaitent se connecter, et il devra aussi s'occuper des clients déjà connectés. Je choisis d'utiliser un contexte multi-threads. Nous pourrions aussi utiliser un contexte multi-processus mais l'utilisation des threads est suffisante, de plus, ils sont moins lourds à gérer pour le système d'exploitation.

Notre serveur fonctionnera de la manière suivante : le processus principal sera en charge d'écouter les connexions entrantes et pour chacune d'elles, il va créer un thread qui sera dédié à cette connexion. Le contenu du thread dépendra entièrement du protocole utilisé (ftp, http ou autre).

J'ai choisi de créer une classe représentant notre serveur. Cette classe aura les méthodes relatives à l'initialisation et au démarrage de notre serveur.

II - Implémentation

Construction du serveur

Le constructeur de notre serveur ne fait pas grand chose, il se contente de renseigner un attribut privé : le numéro du port que le serveur va écouter. On pourrait rajouter d'autres attributs à renseigner dans le cas d'un serveur un peu plus complexe, par exemple le nom d'un fichier contenant toute la configuration du serveur.

Initialisation du serveur :

L'initialisation du serveur contient l'initialisation de la bibliothèque WinSock. C'est également dans cette partie que l'on pourrait mettre l'analyse d'un fichier (xml ?) pour en récupérer les informations de configuration (liste des utilisateurs, mots de passe, liste des IPs non autorisées ...).

Le démarrage du serveur

Il contient la création de la socket d'écoute ainsi que la boucle principale qui attend les connexions et créé un thread pour chacune d'elles. On pourrait penser que la création de la socket aurait plutôt sa place dans l'initialisation du serveur, mais en la mettant dans le démarrage, le port d'écoute reste libre tant que le serveur n'est pas démarré. Ceci permet de libérer le port en mettant le serveur en pause (le processus reste actif), puis de le réutiliser lors d'un redémarrage.

La mise en pause du serveur

Elle se contente de libérer le port d'écoute et d'arrêter l'attente des nouvelles connexions. Il devient alors possible de relancer le serveur sans avoir à le réinitialiser.

II-1 - La classe serveur

```
class serveur{
        private:
                int             port;
                SOCKET         ListeningSocket;
                bool            running;
                SOCKADDR_IN     ServerAddr;
                DWORD           ClientThread(SOCKET);
        public:
                serveur(int);
                int             init();
                int             start ();
                int             pause ();
                static DWORD WINAPI ThreadLauncher(void *p){
                        [...]
                }
};
```

Détail des attributs de la classe :

- port : contient le numéro du port que le serveur va écouter
- ListeningSocket : descripteur de la socket d'écoute
- running : booléen représentant l'état du serveur, en marche ou non
- ServerAddr : structure contenant des informations sur notre serveur

La classe serveur contient deux autres méthodes relatives aux threads : le lanceur de thread et le thread proprement dit. Ces méthodes sont détaillées dans la partie II-5.

II-2 - Initialisation du serveur

La construction du serveur

```
serveur::serveur(int p){
        port            = p;
        running         = false;
}
```

L'initialisation du serveur

```
int serveur::init(){
        struct in_addr  MyAddress;
        struct hostent  *host;
        char            HostName[100];
        WSADATA         wsaData;

        if(WSAStartup(MAKEWORD(2,2), &wsaData ) != 0 ){
                cerr <<"WSAStartup a échoué "<< endl;
                return 1;
        }

        if( gethostname( HostName, 100 ) == SOCKET_ERROR ){
                cerr<< "gethostname() a rencontré l'erreur "<< WSAGetLastError()  << endl;
                return 1;
        }

        if( (host = gethostbyname( HostName ) ) == NULL){
                cerr <<"gethostbyname() a rencontré l'erreur "<< WSAGetLastError()<< endl;
                return 1;
        }

        memcpy( &MyAddress.s_addr, host->h_addr_list[0], sizeof( MyAddress.s_addr ) );

        ServerAddr.sin_family = AF_INET;
        ServerAddr.sin_port = htons( port );
        ServerAddr.sin_addr.s_addr = inet_addr( inet_ntoa( MyAddress ) );

        cout <<"server correctement initialisé" << endl;
        return 0;
}
```

Explications :

La fonction WSAStartup initialise la bibliothèque WinSock. La macro MAKEWORD renseigne la bibliothèque sur la version que l'utilisateur souhaite utiliser (ici la version 2).

Les fonctions gethostname et gethostbyname permettent de récupérer les informations relatives à l'interface réseau comme par exemple l'adresse IP de la machine. Ces informations sont ensuite stockées dans la structure ServerAddr. Notez au passage que inet_ntoa(MyAddress) est une chaîne de caractères contenant l'adresse IP de la machine.

II-3 - Lancement du serveur

La méthode start

```
int serveur::start (){
        SOCKADDR_IN                     ClientAddr;
        int                             ClientAddrLen;
        HANDLE                          hProcessThread;
        SOCKET                          Newconnection;
        struct thread_param         p;

        if( ( ListeningSocket = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP ) ) == INVALID_SOCKET ){
                cerr <<"ne peut créer la socket. Erreur n° " << WSAGetLastError()<< endl;
```

La méthode start

```
                WSACleanup();
                return 1;
        }
```

La fonction socket va créer une socket par laquelle les connexions vont arriver. Cette fonction renvoie un descripteur de socket, qui sera stocké dans ListeningSocket.

```
        if( bind( ListeningSocket, (SOCKADDR *)&ServerAddr, sizeof( ServerAddr ) ) == SOCKET_ERROR
){
                cerr<<"bind a échoué avec l'erreur "<< WSAGetLastError()<< endl;
                cerr<<"Le port est peut-être déjà utilisé par un autre processus "<< endl;
                closesocket( ListeningSocket );
                WSACleanup();
                return 1;
        }
```

La fonction bind va associer la socket avec la structure contenant les informations sur notre serveur.

```
        if( listen( ListeningSocket, 5 ) == SOCKET_ERROR ){
                cerr<<"listen a échoué avec l'erreur "<< WSAGetLastError()<< endl;
                closesocket( ListeningSocket );
                WSACleanup();
                return 1;
        }
```

La fonction listen va écouter la socket ListeningSocket.

```
        cout <<"serveur démarré : à l'écoute du port "<<port<< endl;
        running = true;
        ClientAddrLen = sizeof( ClientAddr );

        while(running){
```

La boucle while tourne tant que le serveur est dans l'état "en marche". Le booléen running devra donc être modifié pour arrêter la boucle. Il est modifié par la méthode pause.

```
                if((Newconnection = accept( ListeningSocket, (SOCKADDR *) &ClientAddr,
&ClientAddrLen)) == INVALID_SOCKET){
                        cerr  <<"accept a échoué avec l'erreur "<< WSAGetLastError() << endl;;
                        closesocket( ListeningSocket );
                        WSACleanup();
                        return 1;
                }
```

La fonction accept va accepter les connexions entrantes et créer une nouvelle socket dont elle renverra le descripteur. C'est aussi ici que l'on peut refuser les connexions provenant de certaines IPs. Il suffit de récupérer l'IP du client dans la structure ClientAddr de la même manière que nous avons récupéré l'adresse du serveur lors de l'initialisation.

```
                p.ser = this;
                p.soc = NewConnection;

                cout << "client connecté ::  IP : "<<inet_ntoa( ClientAddr.sin_addr )<< " ,port =
"<<ntohs( ClientAddr.sin_port )<< endl;

                hProcessThread = CreateThread(NULL, 0,&serveur::ThreadLauncher, &p,0,NULL);
                if ( hProcessThread == NULL ){
                        cerr <<"CreateThread a échoué avec l'erreur "<<GetLastError()<< endl;
                }
```

Chaque fois qu'une connexion a été acceptée, un nouveau thread est créé pour s'occuper de ce nouveau client.

```
        }
        return 0;
}
```

Chaque thread aura besoin de connaître au moins le descripteur de la socket avec laquelle il va pouvoir envoyer des données au client. Pour envoyer d'autres informations, il va falloir créer une structure et envoyer cette structure.

II-4 - Mise en pause du serveur

La mise en pause du serveur doit permettre de libérer le port d'écoute. La méthode de pause doit donc détruire la socket. Ensuite, cette méthode sera étoffée selon les besoins de chacun.

```
int serveur::pause (){
        running = false;
        cout <<"Serveur en pause"<< endl;
        closesocket( ListeningSocket );
        return 0;
}
```

La méthode start() a été invoquée après l'initialisation du serveur. La méthode pause(), quant à elle, devra être invoquée après une action de l'utilisateur. Elle devra donc être associée à un évènement (clic sur un bouton, pression de touche, ...). Cela signifie aussi que les methodes start et pause peuvent tenter d'accéder en même temps au booléen running. Il va donc falloir le protéger par un système de verrous.

II-5 - Les threads des clients

Un thread est créé chaque fois qu'un client arrive. Pour qu'il puisse gérer au mieux le client dont il a la charge, chaque thread doit connaître le descripteur de la socket avec laquelle il va pouvoir échanger des données avec le client. Ce descripteur doit donc être fourni au thread sous la forme d'un paramètre de fonction.

Voici comment le thread est créé :

La fonction CreateThread ne peut pas directement invoquer un thread en tant que méthode de la classe courante. C'est pourquoi j'ai choisi d'utiliser la méthode décrite dans la FAQ C++ : je me sers d'une méthode statique pour lancer un thread. La fonction CreateThread va donc appeler le lanceur de thread. Ce lanceur est déclaré comme suit :

```
static DWORD WINAPI ThreadLauncher(void *p){
        struct thread_param *Obj = reinterpret_cast<struct thread_param*>(p);
        serveur *s = Obj->ser;
```

```
        return s->ClientThread(Obj->soc);
}
```

Notez au passage qu'il est possible d'utiliser le même lanceur pour lancer plusieurs threads même s'ils ont des traitements différents. Il suffit de rajouter un champ dans la structure, et donner une valeur (un index, en somme) permettant de lancer le bon traitement.

L'argument passé au lanceur est un pointeur sur une structure contenant un pointeur sur l'objet courant, ainsi que le descripteur de la socket du client.

```
struct thread_param{
        serveur* ser;
        SOCKET soc;
};
```

Les champs de cette structure sont renseignés juste avant la création du thread ; le champ relatif à l'objet courant contiendra toujours la même valeur, il est donc tout à fait possible de renseigner ce champ avant la boucle générale d'attente.

Une fois l'argument transmis, le lanceur récupère l'objet courant et s'en sert pour invoquer la méthode de classe correspondant au traitement à effectuer (dans notre cas, la méthode ClientThread). La méthode ClientThread contient la description des interactions entre notre serveur et nos clients.

```
DWORD serveur::ClientThread(SOCKET soc){
        cout << "thread client démarré" << endl;

        /*    A mettre ici : code relatif au protocole utilisé    */

        return 0;
}
```

Attention, la structure thread_param est régulièrement écrasée pour démarrer de nouveaux threads. C'est pourquoi, il faut dupliquer les champs de la structure dans le thread du client. De même, pour se prémunir contre les accès concurrents malencontreux, on peut utiliser un sémaphore autour de la création du thread. Sans quoi, un thread pourrait influer sur la création d'un nouveau thread en lui modifiant sa structure thread_param.

Libre à vous d'étoffer le contenu de cette fonction selon le protocole que vous souhaitez implémenter. Par exemple, pour l'implémentation d'un serveur de fichiers, notre thread pourrait attendre des requêtes décrivant des fichiers. A chaque requête reçue, le serveur répondrait en ouvrant une nouvelle connexion dans laquelle il va envoyer le fichier demandé. Si par contre vous souhaitez implémenter un serveur de chat, chaque thread devra relayer les messages écrits, mais aussi être à l'écoute des requêtes du client. Une possibilité pourrait être d'utiliser une paire de sockets (jeu de mots ? qui a parlé de jeu de mots ?). Vous pourrez donc être amenés à manipuler une seule socket synchrone, ou bien une paire de sockets synchrones, ou bien des sockets asynchrones ...

II-6 - Le reste de l'implémentation

main.cpp

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

using namespace std;

int main(){

        serveur *MyServer = new serveur(12345);
        if(MyServer->init()!=0){
                cerr << "ne peut initialiser le serveur"<< endl;
```

main.cpp

```
                return 1;
        }

        if(MyServer->start()!=0){
                cerr << "ne peut démarrer le serveur"<< endl;
                return 1;
        }

        return 0;
}
```

Le main se contente ici de créer une instance de la classe serveur et de démarrer notre serveur.

N'oubliez pas de linker la bibliothèque WinSock dans votre compilateur préféré, ni d'inclure les headers qui vont bien : <winsock2.h> et <iostream> dans les fichiers serveur.cpp et serveur.h

III - Conclusion

Nous voilà arrivés au terme de cet article pour présenter une architecture de serveur multi-threads. Il s'agit d'un serveur générique pouvant implémenter n'importe quel protocole. Dans le cas d'un serveur plus conséquent, il faudra rajouter une gestion des utilisateurs, des mots de passe ...

Le portage sur un système Unix / Linux ne demanderait que la modification des objets systèmes mis en jeu, à savoir les sockets et les threads. Mais l'architecture présentée ici resterait la même.

IV - Téléchargement

Voici les 3 fichiers mis en jeu.

- main.cpp
- serveur.cpp
- serveur.h

V - Remerciements

Je tiens à remercier toute l'équipe de la section C/C++ (nico-pyright(c) pour ne citer que lui) pour m'avoir donné leurs impressions constructives et pour la relecture de cet article.