CLIENT/SERVEUR EN C++/CLI

Cet article a pour but d'expliquer l'utilisation des classes TcpListener et TcpClient dans un environnement windows forms. Ces deux dernières classes permettent de faire du développement réseau au niveau de la couche Tcp/Ip. Elles permettent de s'abstraire de la programmation des sockets, un peu plus technique et orientée protocole.
TcpListener propose une méthode asynchrone pour écouter sur un port. Ceci permet de ne pas figer la form, et de recevoir simultanément des messages en provenance de plusieurs clients. Nous verrons une utilisation des event/delagate afin de faire remonter les messages du thread réseau vers le thread de la form. En effet les contrôles de la form n'autorise pas les opérations crossthreads. Nous verrons aussi l'utilisation des delegate pour le déclenchement d'une routine dans un thread autre que celui de la form, avec toujours la même idée, ne pas figer l'IHM.

Cet article s'adresse aux débutants qui souhaitent découvrir un cas pratique de code qui gère la problématique des threads dans les windows forms, ainsi que l'émission/réception de données sur le réseau.

A ajouter : liens vers la MSDN des classes du Framework.

INTRODUCTION

Cet article se décompose en deux parties. Une partie pour le serveur et une partie pour le client. Le code source complet de la solution sera fourni (Framework 3.5 et compilateur Visual C++ 2008).

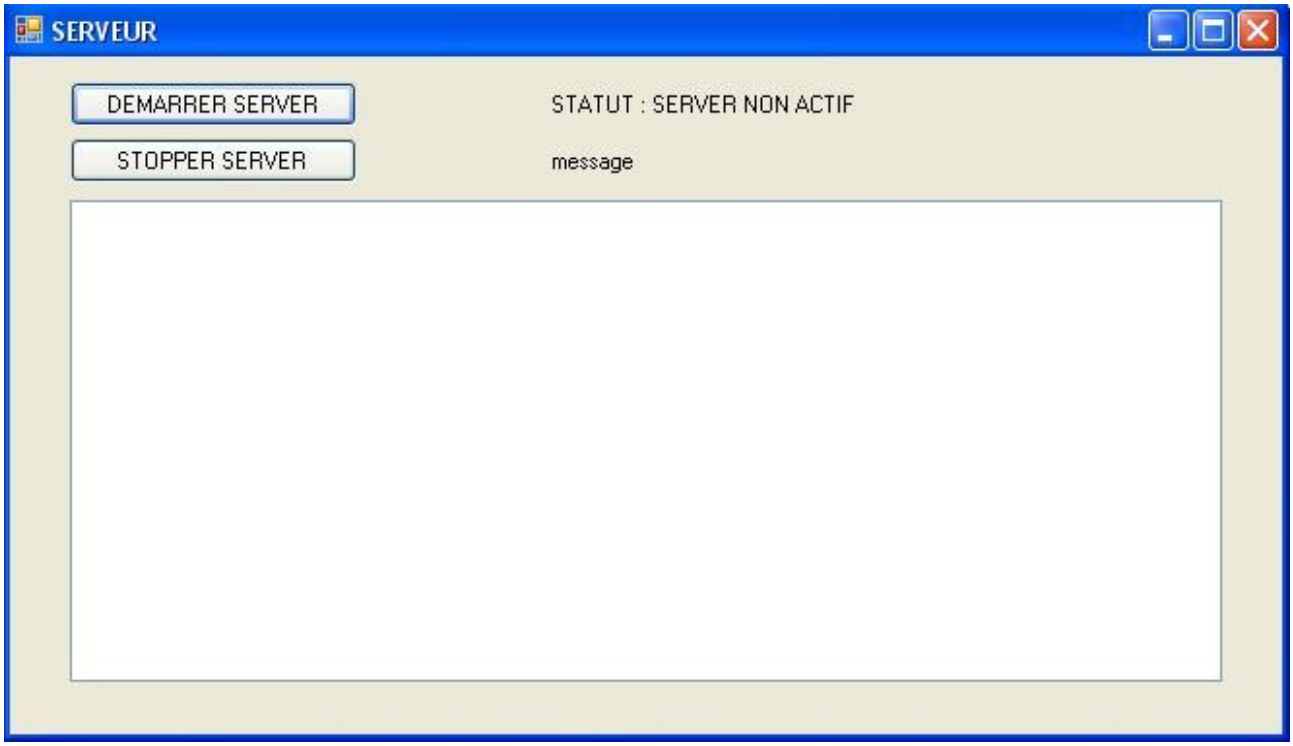
Pour le serveur, nous décrirons l'interface graphique, puis la classe « Cserver » qui nous servira à écouter sur le réseau en mode asynchrone. Cette classe est développée pour être autonome et réutilisable. Nous aurons donc une deuxième classe CAppServer qui dérivera de cette première, et qui en étendra les fonctionnalités propres à nos besoins. Nous décrirons la classe « CformEvent », un singleton pour l'envoi des messages réseaux à l'IHM. Nous verrons aussi comment récupérer l'adresse IP de l'émetteur.

Pour le client, nous décrirons l'interface graphique, puis la façon de transmettre des messages réseaux au serveur. Nous utiliserons une classe CformEvent, mais beaucoup plus simple que pour le serveur. Elle nous permettra d'avoir les éléments afin de lancer un thread qui ne figera pas l'IHM lors des opérations réseaux. La classe CArgument nous aidera à transmettre des informations au delegate qui lance le thread.

Les classes TcpClient et TcpListener font référence au namespace System::Net::Sockets.

I. Le Serveur

I-A Interface Graphique



Rien de compliqué : deux boutons, un pour démarrer l'écoute sur le réseau, un autre pour arrêter le serveur. Un label nous signale le statut du serveur et un autre différents messages (erreurs, adresse IP de l'émetteur). Et enfin une TextBox pour l'affichage des messages du client. Ce seront des messages au format texte. Cette interface est obtenue en utilisant le designer de Visual Studio. Un glissé-déposé des composants et le tour est joué. Les deux boutons gèrent l'évènement Click que vous obtenez en double-cliquant sur les contrôle dans la fenêtre du designer.

I-B La classe CServer

```
//---------------------------------------------------------------------
// cserver.h
//---------------------------------------------------------------------
#ifndef CSERVER_H
#define CSERVER_H

ref class CServer abstract{

  public:

    CServer(const int iPort);
    ~CServer(){ StopServer(); }

    // cserver.cpp
    bool InitServer();
    bool StopServer();

    ref class ServerReportEventArgs  : EventArgs{

      public:

        ServerReportEventArgs(String^ szMsg){ m_szMsg = szMsg; }

        property String^ MESSAGE{

          String^ get(){ return m_szMsg; }
          void set(String^ sz){ m_szMsg = sz; }
        }

     private:
         String^ m_szMsg;
  };

  delegate void ServerReportHandler(Object^, ServerReportEventArgs^);

  event ServerReportHandler^ OnServerReportEvent;

protected:

  virtual void ShellNetMessage(NetworkStream^, IPAddress^) = 0;
  void OnServerReport(ServerReportEventArgs^);

private:

  TcpListener^   m_cListener;
  AsyncCallback^ m_asCallBck;
  static Object^ m_oLockServer = gcnew Object;
  bool m_bActive;

  SynchronizationContext^ m_SyncCtx;

  // cserver.cpp
  static void AcceptTcpClientCallback(IAsyncResult^);

  void ServerReport(Object^);
};

#endif
```

Cette classe est déclarée abstract. En effet elle ne pourra pas être utilisée ainsi mais devra être dérivée par une autre classe.

Les variables membres privées:

- TcpListener^ m_cListener : cette classe va nous permettre de contrôler l'écoute réseau.
- Object^ m_LockServer : Cette objet va nous servir à synchroniser deux appels de fonction.
- Bool m_bActive : un booléen qui nous permettra de connaître l'état de la connexion.
- SynchronizationContext^ m_SyncCtx : cette classe du framework nous permettra de basculer d'un thread à un autre.

Le constructeur/destructeur :

Le constructeur prend en paramètre le numéro du port sur lequel nous allons faire l'écoute réseau.

Le destructeur prend l'initiative d'arrêter le serveur dans le cas où celui-ci est actif.

Les fonctions membres publics :

Une fonction d'initialisation et une fonction d'arrêt (InitServer/StopServer). Ce sont ces deux fonctions que nous allons synchroniser avec m_LockServer. En effet il ne serait pas souhaitable que ces deux fonctions s'exécutent en même temps. Dans notre programme, cette situation n'existe pas, mais c'est surtout à titre d'exemple, et aussi parce que la classe CServer pourrait être utilisée par un autre programme qui lui pourrait connaître cette situation. Nous développons les classes dans une perspective objet (POO), avec ici pour objectif la réutilisation du code de la classe.
Ces deux fonctions retournent un booléen pour nous signaler l'échec ou la réussite de l'appel.

Les objets publics de gestion d'évènements :

Nous allons donner à cette classe le même comportement que tous les contrôles du framework ont. Il s'agit de proposer un événement (event), auquel les classes qui utiliseront notre serveur pourront s'abonner. Nous pourrons alors transmettre des messages aux objets qui seront abonnés, et sans que ceux-ci n'aient à gérer l'aspect crossthread ; d'où le SynchronizationContext. Pour transmettre des arguments aux objets abonnés, nous avons besoin d'une classe qui hérite de EventArgs, et nous l'appelerons ServerReportEventArgs. C'est une façon de créer un système d'évènement avec des arguments personnalisés. ServerReportEventArgs ne fera que transmettre une String. Nous déclarons ensuite la signature de notre événement : delegate void ServerReportHandler(Object^, ServerReportEventArgs^); . Puis l'évènement auquel les objets pourront s'abonner : event ServerReportHandler^ OnServerReportEvent; . Tout ceci est en public afin que les classes qui voudront s'abonner puissent avoir accès aux définitions.

Les fonctions membres protégées :

virtual void ShellNetMessage(NetworkStream^, IPAddress^) = 0; Cette fonction est déclarée virtuelle pure. En effet nous obligeons ainsi à la classe qui héritera de Cserver à définir cette fonction. Cette fonction est le point d'entrée de tous les flux réseaux qui seront interceptés par le serveur. Le premier paramètre est un NetworkStream. A l'intérieur se trouveront toutes les données réseaux transmises par un client. Le deuxième paramètre est l'adresse IP du client émetteur. Cette fonction permet surtout de séparer la logique d'écoute du serveur de la logique de traitement des données des connexions.

void OnServerReport(ServerReportEventArgs^); C'est la fonction qui va permettre de déclencher un événement. Le seul paramètre est notre argument personnalisé.

Les fonctions membres privées :

La fonction statique AcceptTcpClientCallback va recevoir toutes les demandes de connexion au serveur de manière asynchrone. Deux ou plusieurs clients pourraient se connecter en même temps au serveur. Le paramètre IasyncResult représente l'état de l'opération asynchrone.

void ServerReport(Object^); C'est la fonction de déclenchement de l'évènement, mais au niveau du thread de l'interface graphique.

I-C La classe CAppServer

```
//---------------------------------------------------------------------------
// cappserver.h
//---------------------------------------------------------------------------
#ifndef CAPPSERVER_H
#define CAPPSERVER_H

ref class CAppServer : CServer{

  public:

    CAppServer(const int iPort) : CServer(iPort){}

  protected:

    virtual void ShellNetMessage(NetworkStream^, IPAddress^) override;
};

#endif
```

Ceci est est la classe serveur de notre application. Elle dérive de Cserver. Nous définissons la fonction ShellNetMessage(override). Depuis cette fonction nous allons gérer les messages réseaux des clients. Le constructeur se charge de transmettre le paramètre à la classe de base.

I-D La classe CFormEvent

Lorsque la classe CAppServer reçoit des messages réseaux, elle doit les transmettre à la form. La form elle, doit pouvoir écouter les messages en provenance du réseau. La classe CAppServer ne connaît rien de la form. Pour résoudre ces deux dilemmes, nous allons utiliser deux Design Patterns. Le Pattern Observateur et Singleton. Pour le Pattern Observateur, la form va se mettre en observation des messages émis par la classe CAppServer, qui elle va les déclencher. Le Pattern Singleton lui, va permettre à la classe CAppServer de communiquer indirectement avec la form. Le singleton servira alors d'intermédiaire. Son code sera accessible à tous les fichiers incluant « cformevent.h ». Cette méthode de communication entre deux classes est une autre façon de faire. Nous pourrions en effet utilser le concept d'évènement comme pour la classe CServer. Je présente cette autre manière de faire, car cela permettra de comprendre et d'appliquer la gestion crossthread au niveau de la form et non plus au niveau de la classe CServer.

```
//---------------------------------------------------------------------
// cformevent.h
//---------------------------------------------------------------------
#ifndef CFORMEVENT_H
#define CFORMEVENT_H

delegate void MessageDelegate(String^);

ref class CFormEvent sealed{

  public:

    static CFormEvent^ GetCFormEvent(){ return m_cFormEvent; }

    event MessageDelegate^ OnMsgIhmDlgt;

    void  MsgIhm(String^ szMsg){ OnMsgIhmDlgt(szMsg); }

  private:

    CFormEvent(){}

    static CFormEvent^ m_cFormEvent = gcnew CFormEvent;
};

#endif
```

Premièrement déclarer un delegate « delegate void MessageDelegate(String^); » Ceci nous permet d'officialiser une signature de fonction qui permettra au compilateur de vérifier que les appels et utilisations de fonction de ce délégué sont conformes dans le code.

En C++/Cli, un moyen de déclarer un singleton est le suivant : on utilise le mot clé sealed (la classe ne peut pas être héritée), le constructeur est privé et on utilise un membre static, référence de cette classe, que l'on initialise au plus tôt. (à mettre plusieurs façons de gérer le pointeur).

En public, nous avons une méthode qui permet de récupérer un pointeur de cette classe (GetCFormEvent). Ensuite nous déclarons notre délégué (OnMsgIhmDlgt) de type event. Ce type nous permet d'avoir un gestionnaire d'évènements pour le Pattern Observateur.
La fonction MsgIhm est la fonction de déclenchement de l'évènement.

I-E La classe Form1

Le designer de Visual Studio a la fâcheuse tendance à désorganiser le fichier « Form1.h », lorsque l'on ajoute ou retire des contrôles. Le C++/Cli ne dispose pas des classes partielles. Je réorganise souvent ce fichier. Regardons maintenant une partie du contenu de ce fichier :