```
private: System::Windows::Forms::Label^    label1;
private: System::Windows::Forms::Label^    label2;
private: System::Windows::Forms::TextBox^  textBox1;
private: System::Windows::Forms::Button^   button1;
private: System::Windows::Forms::Button^   button2;

private: CAppServer^ m_cAppServer;

// Form1_misc.cpp
private: void Init();

private: void ServerReport(Object^, CAppServer::ServerReportEventArgs^);

private: MessageDelegate^ m_MsgTxtBoxDlgt;
private: void SetMsgInTxtBox(String^);
private: void SetMsgInTxtBoxIhm(String^);

// inline
private: void End(){ m_cAppServer->StopServer(); }

// Form1_ctrlevent.cpp
private: System::Void button1_Click(System::Object^, System::EventArgs^);
private: System::Void button2_Click(System::Object^, System::EventArgs^);
```

Au début, nous pouvons voir les déclarations des contrôles de la form. Ensuite une variable membre de la classe CAppServer. Dans « Form1_misc.cpp », nous allons regrouper les fonctions génériques. Il y a une fonction (Init) qui regroupe toutes les initialisations à faire au chargement de la form. Ensuite la fonction ServerReport, sera la fonction exécutée lorsqu'un événement sera déclenché par le serveur. Le premier paramètre contiendra l'objet qui a déclenché l'évènement, et le deuxième paramètre contiendra les arguments de l'évènement.

Ensuite nous déclarons un délégué et deux fonctions (m_MsgTxtBoxDlgt, SetMsgInTxtBox, SetMsgInTxtBox Ihm). Ceci nous permettra de gérer l'aspect crossthread de la réception de messages en provenance de la classe CAppServer.
Puis une fonction de terminaison, (End) qui s'occupe juste de stopper le serveur, dans l'éventualité où celui-ci est actif.

Enfin, la gestion de nos deux évènements de clics sur les boutons de la form, qui seront regroupés dans le fichier « Form1_ctrlevent.cpp ».

Maintenant que nous avons vu les déclarations de classe, passons au code et à son explication.

I-F Description des fonctions dans cserver.cpp

```
//-----------------------------------------------------------------------
// cserver.cpp
//-----------------------------------------------------------------------
#include "stdafx.h"

CServer::CServer(const int iPort) : m_bActive(false){

  m_cListener = gcnew TcpListener(IPAddress::Any, iPort);

  m_SyncCtx = SynchronizationContext::Current;

  m_asCallBck = gcnew AsyncCallback(&CServer::AcceptTcpClientCallback);
}
```

Nous initialisons à false le booléen qui gère le statut de la connexion. Nous instancions le TcpListener en précisant que nous acceptons les connexions sur toutes les interfaces réseaux (IPAdress::Any). L'écoute se fera sur le numéro de port passé en paramètre. Nous initialisons notre contexte de synchronisation sur le thread courant. Enfin m_asCallBck est initialisé de façon à conserver une référence sur notre fonction AcceptTcpClientCallback .

```
bool CServer::InitServer(){

  msclr::lock l(m_oLockServer);

  if(m_bActive)
    return m_bActive;

  try{

    m_cListener->Start();

    m_cListener->BeginAcceptTcpClient(m_asCallBck, this);

    m_bActive = true;
  }
  catch(Exception^ e){

    OnServerReport(gcnew ServerReportEventArgs("InitServer : " + e->Message));
  }
  return m_bActive;
}
```

Au départ nous utilisons la fonction lock qui nécessite d'inclure le fichier <msclr\lock.h>. C'est un système d'exclusion mutuelle. La même méthode sur le même objet sera appelée dans la fonction StopServer. Ces deux fonctions ne pourront s'exécuter que lorsque l'une des deux aura terminée. Bien sûr dans le cas où une des deux fonctions seraient en train d'être exécutée. Ceci nous assure surtout que le booléen m_bActive possède une valeur identifiant réellement l'état de la connexion. Cela bloquera bien évidemment deux appels successifs à la fonction InitServer.

Ensuite, nous vérifions que la connexion n'est pas déjà active, auquel cas on ne fait rien.

Puis nous lançons l'écoute réseau avec m_cListener->Start(); .A ce moment-là, la socket sous-jacente est initialisée et les demandes de connexion sont mises en file d'attente. Elles sont transmises dans l'ordre d'arrivée à la fonction de réception suivante :

m_cListener->BeginAcceptTcpClient(m_asCallBck, this);

Lors de cet appel nous lançons une opération asynchrone de réception des données réseaux. Le premier paramètre est la référence vers notre fonction de callback que nous avons initialisé dans le constructeur de cette classe. En effet, si des données réseaux sont reçues, la fonction de callback sera appelée (AcceptTcpClientCallback). Le deuxième paramètre est une référence sur la classe CServer elle-même. Ceci nous servira à manipuler certain membre de cette classe. En effet la fonction AcceptTcpClientCallback est statique et donc les membres de CServer n'y sont pas accessibles.

Si tout s'est bien passé, alors la connexion est active m_bActive = true; .

Comme vous pouvez le voir, les appels précédents sont encadrés dans un bloc try/catch. Les fonctions Start et BeginAcceptTcpClient peuvent générées une SocketException. Si une exception se produit nous appelons la méthode OnServerReport afin de transmettre l'erreur aux objets qui seraient abonnés à cet évènement. Le message en question identifie la fonction dans laquelle se déroule l'erreur (InitServer) plus le contenu du message (e->Message).

```
bool CServer::StopServer(){

  msclr::lock l(m_oLockServer);

  if(m_bActive == false)
    return !m_bActive;

  try{

    m_cListener->Stop();
    m_bActive = false;
  }
  catch(Exception^ e){

    OnServerReport(gcnew ServerReportEventArgs("StopServer : " + e->Message));
  }
  return !m_bActive;
}
```

Ici, toujours notre appel à lock dès le début de la fonction. Si m_oLockServer a été acquis soit dans cette fonction, soit dans InitServer, alors la fonction attendra que m_oLockServer redevienne dans un état signalé.

Ensuite si la connexion est inactive, nous retournons true.

Puis nous stoppons la connexion m_cListener->Stop(); et si tout se passe bien nous mettons m_bActive à false.

Le gestionnaire d'exception se comporte comme pour la méthode InitServer.
Nous retournons l'inverse de la valeur m_bActive afin d'être cohérent avec notre fonction. En effet la valeur vaut false mais nous retournons true pour dire que la connexion est bien arrêtée.

```
void CServer::AcceptTcpClientCallback(IAsyncResult^ ar){

  TcpClient^        client = nullptr;
  NetworkStream^    stream = nullptr;
  CServer^          Server = nullptr;

  try{

    Server = (CServer^)ar->AsyncState;

    Server->m_cListener->BeginAcceptTcpClient(Server->m_asCallBck, Server);

    client = Server->m_cListener->EndAcceptTcpClient(ar);

    stream = client->GetStream();

    IPEndPoint^ remoteIP = (IPEndPoint^)client->Client->RemoteEndPoint;
    IPAddress^ ip = remoteIP->Address;

    Server->ShellNetMessage(stream, ip);
  }
  catch(Exception^ e){

    if(Server)
      Server->OnServerReport(gcnew ServerReportEventArgs("AcceptTcpClientCallback : " + e->Message));
  }
  finally{

    if(stream)
      stream->Close();

    if(client)
      client->Close();
  }
}
```

Nous déclarons à l'entrée de cette fonction un TcpClient. C'est une classe du framework qui va nous permettre de communiquer avec le client qui vient de demander une connexion. Puis nous déclarons un NetworkStream, classe du Framework pour lire et écrire des données réseaux sur la connexion du client. Enfin nous déclarons une référence sur notre classe elle-même. Tous ces objets sont initialisés à nullptr, ceci afin de vérifier avant de les utiliser qu'ils références bien un objet.

Rappelez-vous, lors de l'appel à AcceptTcpClientCallback, nous avons mis en deuxième paramètre this. Ce paramètre, nous le retrouvons dans AsyncState de la variable ar. Nous récupérons donc cette référence de cette manière : Server = (Cserver^)ar->AsyncState; .

Juste après avoir récupéré notre référence sur la classe elle-même, nous ne perdons pas de temps et nous relançons aussitôt une écoute asynchrone : Server->m_cListener->BeginAcceptTcpClient(Server->m_asCallBck, Server); . Si une autre demande de connexion est en attente, elle sera aussitôt exécutée et nous entrerons à nouveau dans AcceptTcpClientCallback.

Toutes les méthodes qui commencent par Begin... doivent avoir leur équivalent End... C'est une façon de dire au Framework qu'il doit libérer les ressources générées lors de l'appel à une méthode Begin... Et c'est ce que nous faisons avec : client = Server->m_cListener->EndAcceptTcpClient(ar); . De plus cette méthode nous retournes une référence sur la connexion du client. Cette référence nous permet de nous connecter au flux réseau du client : stream = client->GetStream(); . Nous pourrons alors lire et écrire des données.

Les deux appels suivants nous permettent de récupérer l'adresse IP du client. Cette valeur se trouve dans le membre RemoteEndPoint->Address de la classe TcpClient.

Enfin, nous dispatchons le NetxorkStream et l'adresse IP dans la fonction ShellNetMessage, là où serons traitées les données réseaux.