Développement d'un logiciel de messagerie instantanée avec Dotnet (version complète)

Propriétés	Description
Intitulé long	Développement d'un logiciel de messagerie instantanée
Formation concernée	Classes de BTS Services informatiques aux organisations
Matière	SLAM4 - Réalisation et maintenance de composants logiciels
Présentation	Développement d'un logiciel de messagerie instantanée. La communication est basée sur les sockets entre deux applications dotnet.
Notions	▪ Programmation à l'aide d'objets ▪ Utilisation d'une classe ▪ Utilisation d'un composant logiciel commun à plusieurs projets ▪ Développement orienté réseau ▪ Sérialisation ▪ Communication un à plusieurs (*multicast*)
Transversalité	Le cours sur les protocoles réseau
Pré-requis	Le langage C#, les applications WinForms.
Outils	Les applications fournies ont été réalisées avec Visual Studio 2008 et sont destinées au framework dotnet 3.5, elles peuvent facilement être recompilées avec un autre outil et/ou cibler une autre version du framework. - CoursSockets.doc : support de cours élève. - Exemple01 : un premier exemple d'application simple. - Exemple02 : exemple de réception asynchrone. - Exemple03 : utilisation d'un objet *BackgroundWorker*. - Exemple04 : utilisation d'une classe *MessageReseau*. - Chat : application simple de chat. - ExempleCommun : utilisation d'une DLL commune à plusieurs projets. - Serialisation : exemple de sérialisation binaire. - SerialisationCommun : sérialisation utilisant une DLL commune. - ChatMulticast : application chat, le serveur émet en multicast.=
Mots-clés	C # , DotNet
Durée	10 heures
Auteur(es)	Pierre Loisel
Version	v 1.0
Date de publication	Novembre 2008

Ce document n'est pas une étude exhaustive du domaine. Il présente simplement une manière de réaliser des applications communicantes en C#.

Les applications fournies ont été réalisées avec Visual Studio 2008 et sont destinées au framework dotnet 3.5, elles peuvent facilement être recompilées avec un autre outil et/ou cibler une autre version du framework.

L'exécution d'une application utilisant les sockets à partir d'un disque réseau peut poser des problèmes de sécurité. Si vous obtenez une exception concernant la sécurité, configurez les paramètres de sécurité du framework. Vous pouvez consulter la page http://msdn.microsoft.com/fr-

fr/library/2bc0cxhc.aspx pour obtenir des informations concernant l'outil .NET Framework Configuration (Mscorcfg.msc).

A/ Présentation

1. Les sockets

Les sockets fournissent un mécanisme générique de communication entre les processus. Elles sont apparues pour la première fois en 1986 dans la version UNIX de l'université de Berkeley.

Un processus peut être sommairement défini comme une application (ou un service) en cours d'exécution.

Un socket permet à un processus (le client) d'envoyer un message à un autre processus (le serveur). Le serveur qui reçoit ce message peut alors accomplir toutes sortes de tâche et éventuellement retourner un résultat au processus client.

Lors de la création d'un socket, il faut préciser le type d'adressage, le type de message et le protocole transport utilisés. Nous utiliseront : IPV4, les datagrammes simples et le protocole UDP.

2. Point de terminaison

Un point de terminaison (EndPoint) est défini par une adresse IP et un numéro de port. Une communication s'établit entre deux points de terminaison.

Un processus serveur reçoit les messages destinés à un numéro de port et à un protocole transport (UDP, TCP, ...) déterminés. Un client désirant envoyer un message à un serveur doit donc créer un point de terminaison représentant le récepteur du message en fournissant l'adresse IP du serveur et le numéro de port.

Pour envoyer un message, un processus doit également créer un point de terminaison représentant l'émetteur du message en fournissant sa propre adresse IP. Il ne fournit pas de numéro de port, c'est le système qui attribuera un numéro de port libre pour la communication.

3. Principe de communication

Lors de l'envoi d'un message, il faut :

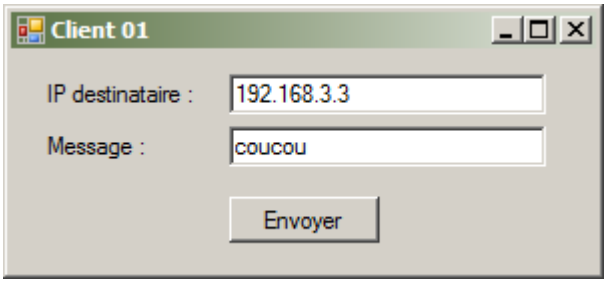
- Créer un socket,
- Créer le point de terminaison émetteur,
- Lier le socket au point de terminaison émetteur (ce qui précisera le protocole transport utilisé pour l'émission).
- Créer le point de terminaison récepteur,
- Envoyer le message à l'aide du socket vers le point de terminaison récepteur.

Pour recevoir un message, le processus serveur doit :

- Créer un socket,
- Créer le point de terminaison récepteur (lui-même donc),
- Lier le socket au point de terminaison récepteur (ce qui précisera le protocole transport utilisé pour la réception).
- Créer le point de terminaison émetteur (sans préciser l'adresse IP ni le numéro de port puisqu'ils ne sont pas connus à ce stade).
- Mettre le socket en état de réception en lui fournissant un buffer pour les données à recevoir, et une référence au point de terminaison émetteur.
- Lorsque le message est effectivement reçu, le point de terminaison émetteur est renseigné.

4. Premier exemple (exemple01)

Le client :




```
public Fm_client()
{
    InitializeComponent();
    adrIpLocale = getAdrIpLocaleV4();
}
private IPAddress adrIpLocale;
private IPAddress getAdrIpLocaleV4()
{
    string hote = Dns.GetHostName();
    IPHostEntry ipLocales = Dns.GetHostEntry(hote);
    foreach (IPAddress ip in ipLocales.AddressList)
    {
      if (ip.AddressFamily == AddressFamily.InterNetwork)
      {
        return ip;
      }
    }
    return null; // aucune adresse IP V4
}

private void bt_envoyer_Click(object sender, EventArgs e)
{
    byte[] message;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur =
                new IPEndPoint(  IPAddress.Parse(tb_ipDestinataire.Text),
                                 33000);
    message = Encoding.Unicode.GetBytes(tb_message.Text);
    sock.SendTo(message, epRecepteur);
    sock.Close();
}
```

Remarque : la chaîne à envoyer est transformée en un tableau de bytes.

Le serveur :


```
public Fm_serveur()
{
    InitializeComponent();
    adrIpLocale = getAdrIpLocaleV4();
}
private IPAddress adrIpLocale;
private IPAddress getAdrIpLocaleV4()
{
    string hote = Dns.GetHostName();
    IPHostEntry ipLocales = Dns.GetHostEntry(hote);
    foreach (IPAddress ip in ipLocales.AddressList)
    {
      if (ip.AddressFamily == AddressFamily.InterNetwork)
      {
        return ip;
      }
    }
    return null; // aucune adresse IP V4
}
private void bt_recevoir_Click(object sender, EventArgs e)
{
    byte[] message = new byte[40];
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epRecepteur = new IPEndPoint(adrIpLocale, 33000);
    sock.Bind(epRecepteur);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.ReceiveFrom(message, ref epTemp);
    IPEndPoint epEmetteur = (IPEndPoint)epTemp;
    string strMessage = Encoding.Unicode.GetString(message);
    MessageBox.Show( epEmetteur.Address.ToString()
                    + " -> " + strMessage);
}
```

Remarques :

- La fonction *getAdrIpLocaleV4* retourne l'adresse IP V4 de l'hôte en utilisant le service DNS. Le nom de l'hôte local est récupéré en utilisant le DNS.
- Elle est appelée dans le constructeur du formulaire.
- La méthode *ReceiveFrom* de la classe Socket attend un paramètre de type EndPoint. Il faut donc opérer une conversion de type pour récupérer le point de terminaison émetteur.
- Le message est un tableau de bytes. Il faut le transformer en chaîne de caractères.

Mise en œuvre :

- Créer l'application client et l'application serveur.
- Lancer les deux applications (sur le même poste ou sur deux postes différents).
- Cliquer sur le bouton *recevoir* du serveur.
- Renseigner l'adresse IP du serveur et le message à envoyer dans l'application client.
- Cliquer sur le bouton *envoyer* du client.
- Le serveur affiche l'adresse IP et le message du client.

B/ Réception asynchrone

1. Principe

Lorsque l'on clique sur le bouton *recevoir* du serveur, celui-ci se trouve bloqué en attente du message et ne peut accomplir aucune autre tâche. Pour y remédier, il est possible d'utiliser le mécanisme de réception asynchrone fourni par Dotnet. Ce mécanisme est basé sur les threads. Un thread peut être vu comme un « mini processus » interne à une application. Une application peut avoir plusieurs threads et donc exécuter plusieurs tâches simultanément (cette simultanéité n'est que fictive, en tous cas sur une machine mono-processeur).

Le principe est le suivant :

> Mettre le socket en état de réception en utilisant la méthode ci-dessous :

sock.BeginReceiveFrom(message,0,40,SocketFlags.None, ref epTemp, new AsyncCallback(recevoir),null);

- Le socket est mis en état de réception dans un thread de réception différent du thread principal. Le thread principal poursuit immédiatement son exécution, le programme n'est pas bloqué par cet appel.
- L'avant dernier paramètre indique que la méthode *recevoir* doit être appelée dès la réception d'un message par le thread de réception.

> Gérer le message reçu :

Quand le thread de réception reçoit le message, il le prend en charge (l'affiche par exemple), et se termine. L'idéal est de remettre le socket en état de réception immédiatement, de manière à accepter plusieurs messages les uns après les autres.

2. Modification du serveur (exemple02)


```
public Fm_serveur()
{
    InitializeComponent();
    init();
}
private int lgMessage = 40;
private IPAddress adrIpLocale;
private Socket sock;
private IPEndPoint epRecepteur;
byte[] message;
private IPAddress getAdrIpLocaleV4()
{
    // Idem version pécédente
}
```

```
private void init()
{
    message = new byte[lgMessage];
    adrIpLocale = getAdrIpLocaleV4();
    sock = new Socket( AddressFamily.InterNetwork, SocketType.Dgram,
                       ProtocolType.Udp);
    epRecepteur = new IPEndPoint(adrIpLocale, 33000);
    sock.Bind(epRecepteur);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.BeginReceiveFrom(  message, 0, lgMessage, SocketFlags.None,
                            ref epTemp,
                            new AsyncCallback(recevoir), null);
}
private void recevoir(IAsyncResult AR)
{
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.EndReceiveFrom(AR, ref epTemp);
    IPEndPoint epEmetteur = (IPEndPoint)epTemp;
    string strMessage;
    strMessage = Encoding.Unicode.GetString(message, 0, message.Length);
    Array.Clear(message, 0, message.Length);
    sock.BeginReceiveFrom(  message, 0, lgMessage, SocketFlags.None,
                            ref epTemp,
                            new AsyncCallback(recevoir), null);
    MessageBox.Show( epEmetteur.Address.ToString()
                    + " -> " + strMessage);
}
```

Remarques :

- Le serveur se met en réception dès le lancement de l'application, il n'y a donc plus besoin d'un bouton *recevoir*.
- Dès la réception d'un message, le serveur relance un nouveau thread de réception.
- L'attente de réception ne s'arrête qu'à la fermeture du programme.
- Vous pouvez tester l'envoi de messages à partir de deux clients différents.

2.Visualiser l'activité réseau

Il est possible de visualiser ce qui se passe au niveau du réseau de la manière suivante :

- Ouvrir une fenêtre de commandes (exécuter, cdm.exe) sur le serveur.
- Lancer la commande « netstat – a », le port UDP ouvert doit apparaître.
- Lancer la commande « netstat –s –p UDP », noter les informations affichées.
- Envoyer quelques messages au serveur depuis le client.
- Lancer à nouveau la commande « netstat –s –p UDP », relever les nouvelles valeurs.

C/ Un travailleur…

1. Problème potentiel

Modifiez votre application serveur de la manière suivante :

- Ajouter un contrôle de type *ListBox* (lb_messages) sur le formulaire.
- Au lieu d'afficher le message reçu, la méthode *recevoir* doit maintenant ajouter ce message à la liste *lb_messages*.
- Testez votre application… Problème ! Lisez bien le message d'erreur.

La liste a été créée par le thread principal de l'application. C'est le thread de réception qui exécute la méthode *recevoir* et tente donc de mettre à jour cette liste. Ceci n'est pas permis car cela pourrait conduire à une situation de blocage si plusieurs threads tentaient de le faire en même temps (ou plutôt si un thread tentait de commencer à le faire alors qu'un autre thread n'aurait pas terminé de le faire).

2. Une solution

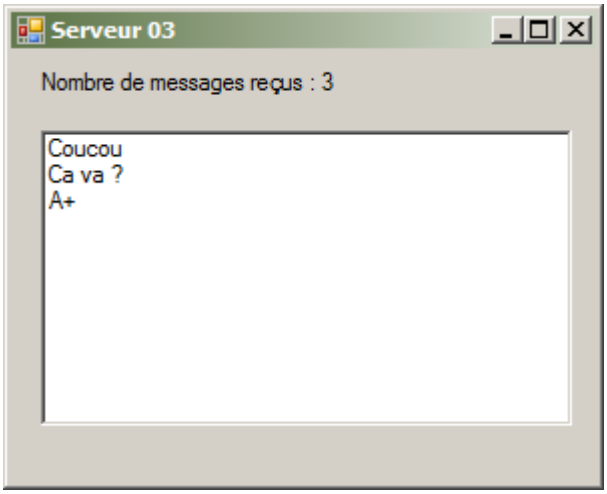
Il s'agit de créer un troisième thread chargé de traiter le message lors de son arrivée. Cette opération peut être réalisée à l'aide d'un objet de la classe *BackgroundWorker*.

Les étapes sont les suivantes :

- Créer un « travailleur » lors de l'arrivée du message (dans la méthode *recevoir* donc).
- Indiquer quelle méthode doit exécuter ce travailleur lorsqu'il est lancé (appelons la *worker_DoWork*). Cette méthode peut recevoir un paramètre (le message reçu par exemple), et peut accomplir une tâche quelconque sauf la mise à jour de l'interface utilisateur. En effet, si ce travailleur mettait à jour l'interface, le problème de concurrence ne serait pas résolu. Cette méthode peut également produire un résultat.
- Indiquer quelle méthode doit être appelée lorsque le travailleur a terminé son travail (appelons la *worker_RunWorkerCompleted*). Cette méthode sera exécutée par le thread principal après l'exécution du thread travailleur. Il est possible de récupérer le résultat produit par la méthode *worker_DoWork*. La méthode étant exécutée par le thread principal, il n'y a plus de problème de concurrence et elle peut mettre à jour l'interface utilisateur.

3. Modification du serveur (exemple03)

- Ajouter une propriété *nbMessages* à la classe *Fm_serveur*. Cette propriété sera destinée à compter le nombre de messages reçus (c'est juste un prétexte permettant d'illustrer le fonctionnement d'un travailleur, on pourrait compter les messages plus simplement).
- Ce nombre de messages sera affiché dans un label *lbl_nbMessages*.
- Mettre en place le mécanisme du travailleur dans la méthode *recevoir*. La méthode *worker_DoWork* reçoit en paramètre le message reçu, incrémente le nombre de messages et renseigne son résultat (le message lui-même).
- A la fin de l'exécution du travailleur, la méthode *worker_RunWorkerCompleted* récupère le résultat de l'exécution du travailleur (*worker_DoWork*) et met à jour l'interface utilisateur.




```
private int nbMessages;
// idem …
private void init()
{
    nbMessages = 0;
    // idem …
}
private void recevoir(IAsyncResult AR)
{
    BackgroundWorker worker = new BackgroundWorker();
    worker.DoWork += new DoWorkEventHandler(worker_DoWork);
    worker.RunWorkerCompleted +=
         new RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.EndReceiveFrom(AR, ref epTemp);
    string strMessage;
    strMessage = Encoding.Unicode.GetString(message, 0, message.Length);
    worker.RunWorkerAsync(strMessage);
    Array.Clear(message, 0, message.Length);
    sock.BeginReceiveFrom(  message, 0, lgMessage, SocketFlags.None,
                            ref epTemp,
                            new AsyncCallback(recevoir), null);
}
void worker_DoWork(object sender, DoWorkEventArgs e)
{
    nbMessages++;
    e.Result = e.Argument;
}
void worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    string element = (string)e.Result;
    lb_messages.Items.Add(element);
    lbl_nbMessages.Text =   "Nombre de messages reçus : "
                            + nbMessages.ToString();
}
```

D/ Gérer un message plus complexe

1. Principe

On peut avoir à transmettre des messages plus complexes qu'une simple chaîne. Au moins deux solutions :

- Créer une classe *MessageReseau* sérialisable et utiliser la sérialisation d'objets. Cette solution sera examinée plus loin dans ce document.
- Transmettre les différents éléments du message dans une seule chaîne de caractères en utilisant un séparateur. Le mieux est de créer une classe *MessageReseau* encapsulant les opérations nécessaires.

2. Classe MessageReseau

```
class MessageReseau
{
    private static char separateur = '#';
    private IPAddress ipEmetteur;
    public IPAddress IpEmetteur
    {
      get { return ipEmetteur; }
      set { ipEmetteur = value; }
    }
    private string texte;
    public string Texte
    {
      get { return texte; }
      set { texte = value; }
    }
    public MessageReseau(IPAddress p_ipEmetteur, string p_texte)
    {
      ipEmetteur = p_ipEmetteur;
      texte = p_texte;
    }
    public byte[] GetInfos()
    {
      string infos =   ipEmetteur.ToString() + separateur.ToString()
                     + texte + separateur.ToString();
      return Encoding.Unicode.GetBytes(infos);
    }
    public MessageReseau(byte[] p_infos)
    {
      string infos = Encoding.Unicode.GetString(p_infos);
      string[] tabInfos = infos.Split(new char[] { separateur });
      ipEmetteur = IPAddress.Parse(tabInfos[0]);
      texte = tabInfos[1];
    }
}
```

- La méthode *GetInfos* permettra de créer le tableau de bytes lors de l'envoi du message par le client.
- Le constructeur *MessageReseau(byte[] p_infos)* permettra de construire l'objet *MessageReseau* lors de la réception par le serveur.
- Le caractère séparateur en fin de message dans *GetInfos* est important, il permet de ne récupérer que le texte réellement envoyé dans le constructeur *MessageReseau(byte[] p_infos)*.

3. Modification du client (exemple04)

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
   byte[] messageBytes;
   Socket sock = new Socket( AddressFamily.InterNetwork,
                             SocketType.Dgram, ProtocolType.Udp);
   IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
   sock.Bind(epEmetteur);
   IPEndPoint epRecepteur =
               new IPEndPoint(  IPAddress.Parse(tb_ipDestinataire.Text),
                                33000);
   MessageReseau leMessage = new MessageReseau ( adrIpLocale,
                                                  tb_message.Text);
   messageBytes = leMessage.GetInfos();
   sock.SendTo(messageBytes, epRecepteur);
   sock.Close();
}
```

Le client envoie cette fois un message plus complexe.

4. Modification du serveur (exemple04)

```
private void recevoir(IAsyncResult AR)
{
   BackgroundWorker worker = new BackgroundWorker();
   worker.DoWork += new DoWorkEventHandler(worker_DoWork);
   worker.RunWorkerCompleted +=
        new RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sock.EndReceiveFrom(AR, ref epTemp);
   MessageReseau leMessage = new MessageReseau (messageBytes);
   worker.RunWorkerAsync(leMessage);
   Array.Clear(messageBytes, 0, messageBytes.Length);
   sock.BeginReceiveFrom(  messageBytes, 0, lgMessage,
                           SocketFlags.None, ref epTemp,
                           new AsyncCallback(recevoir), null);
}
void worker_DoWork(object sender, DoWorkEventArgs e)
{
   nbMessages++;
   e.Result = e.Argument;
}
void worker_RunWorkerCompleted( object sender,
RunWorkerCompletedEventArgs e)
{
   MessageReseau leMessage = (MessageReseau)e.Result;
   string element =   leMessage.IpEmetteur.ToString()
                     + " -> " + leMessage.Texte;
   lb_messages.Items.Add(element);
   lbl_nbMessages.Text =   "Nombre de messages reçus : "
                           + nbMessages.ToString();
}
```

C'est cette fois un objet de la classe *MessageReseau* qui est transmis au travailleur.

E/ Une vraie conversation

1. Transmettre un message à tout le monde

Pour transmettre un message à plusieurs hôtes, il y a au moins trois solutions :

- Transmettre le même message successivement aux différents destinataires, ce qui nécessite une simple boucle.
- Envoyer le message en multicast (cette solution sera étudiée plus loin).
- Envoyer le message en broadcast.

Pour envoyer un message en broadcast, il suffit de modifier ainsi la procédure d'envoi :

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
    byte[] messageBytes;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    sock.SetSocketOption(SocketOptionLevel.Socket,
                         SocketOptionName.Broadcast, true);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur = new IPEndPoint(IPAddress.Broadcast, 33000);
    MessageReseau leMessage = new MessageReseau(  adrIpLocale,
                                                  tb_message.Text);
    messageBytes = leMessage.GetInfos();
    sock.SendTo(messageBytes, epRecepteur);
    sock.Close();
}
```

Le message sera reçu par tous les hôtes en état de réception sur le port UDP 33000.

2. Définir un protocole

Il s'agit de fixer les règles de la communication entre les différentes applications. Imaginons l'exemple d'un pauvre chat :

- Nous avons un serveur et plusieurs clients.
- Lorsqu'un client se connecte, il en informe le serveur.
- Le serveur conserve la liste des clients connectés.
- Lorsqu'un client envoie un message au serveur, celui-ci le transmet à l'ensemble des clients.
- Lorsqu'un client se déconnecte, il en informe le serveur.
- Les clients sont repérés par leur adresse IP, l'unicité des pseudos n'est pas assurée.

Nous avons besoin de définir plusieurs types de messages :

- Connexion (type C)
  Emetteur : un client qui se connecte
  Récepteur : le serveur
  Contenu : L'adresse IP et le pseudo du client
  Réaction du serveur : mémorisation du pseudo

- Envoi (type E)
  Emetteur : un client qui envoi un texte sur le chat
  Récepteur : le serveur
  Contenu : l'adresse IP du client et le texte envoyé
  Réaction du serveur : réémission du texte à tous les clients

- Réémission (type R)
  Emetteur : le serveur
  Récepteur : les clients
  Contenu : l'adresse IP du serveur, le pseudo de l'auteur du texte et le texte lui-même
  Réaction des clients : affichage du pseudo et du texte

- Déconnexion (type D)
  Emetteur : un client qui se déconnecte, c'est-à-dire qui quitte l'application
  Récepteur : le serveur
  Contenu : l'adresse IP du client et son pseudo
  Réaction du serveur : mise à jour de la liste des clients connectés

F/ Le pauvre chat (Chat)

1. La classe MessageChat

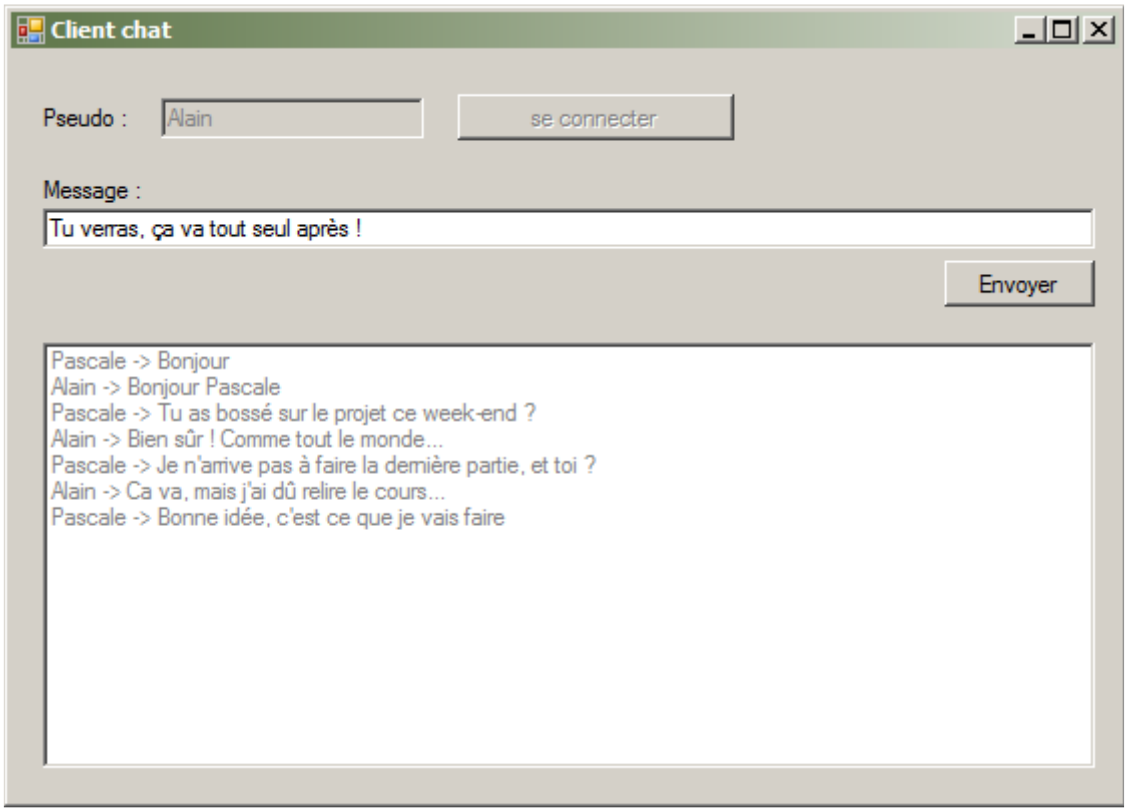
```
class MessageChat
{
    private static char separateur = '#';
    private char typeMessage;
    public char TypeMessage
    {
      get { return typeMessage; }
    }
    private IPAddress ipEmetteur;
    public IPAddress IpEmetteur
    {
      get { return ipEmetteur; }
    }
    private string texte;
    public string Texte
    {
      get { return texte; }
    }
    public MessageChat(  char p_typeMessage, IPAddress p_ipEmetteur,
                         string p_texte)
    {
      typeMessage = p_typeMessage;
      ipEmetteur = p_ipEmetteur;
      texte = p_texte;
    }
    public byte[] GetInfos()
    {
      string infos =   typeMessage.ToString()+ separateur
                     + ipEmetteur.ToString() + separateur.ToString()
                     + texte + separateur.ToString();
      return Encoding.Unicode.GetBytes(infos);
    }

    public MessageChat(byte[] p_infos)
    {
      string infos = Encoding.Unicode.GetString(p_infos);
      string[] tabInfos = infos.Split(new char[] { separateur });
      typeMessage = tabInfos[0][0];
      ipEmetteur = IPAddress.Parse(tabInfos[1]);
      texte = tabInfos[2];
    }
}
```

Remarque :

Cette classe, la fonction *GetAdrIpV4* ainsi que les différentes constantes peuvent facilement être regroupées dans une DLL utilisée par les deux applications. Le dossier *ExempleCommun* illustre cette possibilité en reprenant l'exemple 4.

- Le projet *Commun* contient la classe *MessageReseau* et la classe *UtilIP*. Il s'agit d'un projet de type « Librairie de classes ». Le résultat de sa génération est donc une DLL.
- Les deux projets *Client* et *Serveur* utilisent *Commun.dll*. Il suffit pour cela d'ajouter la référence à cette DLL dans ces projets (« Explorateur de solution », clic droit sur le dossier « Références », « Ajouter une référence », onglet « Parcourir », désigner « Commun.dll »).
- Ajouter « using Commun » dans le source de *Fm_client* et *Fm_serveur*.

2. Le client



Généralités :

- L'adresse IP du serveur est supposée connue.
- Il faut également déterminer les ports utilisés par le serveur et par les clients.

```
public partial class Fm_client : Form
{
    public Fm_client()
    {
      InitializeComponent();
      adrIpLocale = getAdrIpLocaleV4();
    }
    private IPAddress adrIpLocale;
    private static IPAddress ipServeur = IPAddress.Parse("192.168.3.3");
    private static int portServeur = 33000;
    private static int portClient = 33001;
    private static int lgMessage = 1000;
    private Socket sockReception;
    private IPEndPoint epRecepteur;
    byte[] messageBytes;
    private IPAddress getAdrIpLocaleV4()
    {
      // idem…
    }
```

L'envoi d'un message ne pose pas de problème particulier.

```
private void envoyer(MessageChat leMessage)
{
    byte[] messageBytes;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur = new IPEndPoint(ipServeur, portServeur);
    messageBytes = leMessage.GetInfos();
    sock.SendTo(messageBytes, epRecepteur);
    sock.Close();
    tb_message.Clear();
    tb_message.Focus();
}
```

Quand l'utilisateur clique sur « se connecter », le message de connexion est envoyé au serveur et le client se met en état de réception. Il devient ensuite possible de poster un texte sur le chat.

```
private void bt_connecter_Click(object sender, EventArgs e)
{
    if (tb_pseudo.Text != "")
    {
      bt_connecter.Enabled = false;
      tb_pseudo.Enabled = false;
      MessageChat leMessage = new MessageChat(    'C', adrIpLocale,
                                                  tb_pseudo.Text);
      envoyer(leMessage);
      bt_envoyer.Enabled = true;
      tb_message.Enabled = true;
      initReception();
      tb_message.Focus();
    }
}
```

```
private void initReception()
{
   messageBytes = new byte[lgMessage];
   sockReception = new Socket( AddressFamily.InterNetwork,
                               SocketType.Dgram, ProtocolType.Udp);
   epRecepteur = new IPEndPoint(adrIpLocale, portClient);
   sockReception.Bind(epRecepteur);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.BeginReceiveFrom(   messageBytes, 0, lgMessage,
                                     SocketFlags.None, ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
```

A la réception d'un message, le client met à jour son interface à l'aide d'un *BackgroundWorker*. La méthode *worker_DoWork* se contente de transmettre son argument.

```
private void recevoir(IAsyncResult AR)
{
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.EndReceiveFrom(AR, ref epTemp);
   MessageChat leMessage = new MessageChat(messageBytes);
   BackgroundWorker worker = new BackgroundWorker();
   worker.DoWork += new DoWorkEventHandler(worker_DoWork);
   worker.RunWorkerCompleted +=
        new RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
   worker.RunWorkerAsync(leMessage);
   Array.Clear(messageBytes, 0, messageBytes.Length);
   sockReception.BeginReceiveFrom(   messageBytes, 0, lgMessage,
                                     SocketFlags.None, ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
void worker_DoWork(object sender, DoWorkEventArgs e)
{
   e.Result = e.Argument;
}

void worker_RunWorkerCompleted( object sender,
                                RunWorkerCompletedEventArgs e)
{
   MessageChat leMessage = (MessageChat)e.Result;
   switch (leMessage.TypeMessage)
   {
     case 'R':
        lb_messages.Items.Add(leMessage.Texte);
        break;
   }
}
```

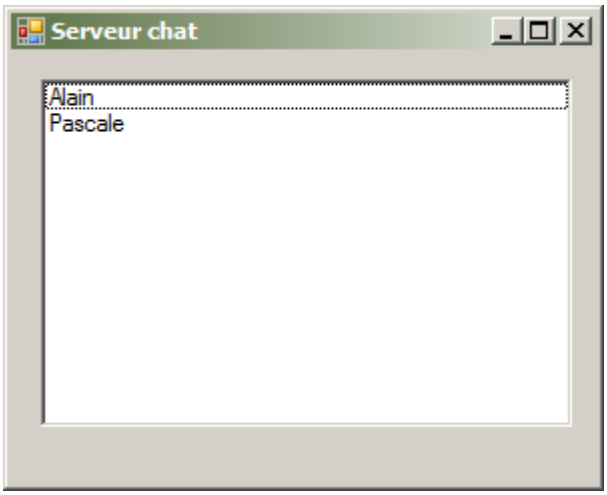
Quand l'utilisateur clique sur le bouton « envoyer », le texte saisi est envoyé au serveur.

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
   MessageChat leMessage = new MessageChat('E', adrIpLocale,
                                                tb_message.Text);
   envoyer(leMessage);
}
```

La déconnexion est gérée par un message envoyé à la fermeture du formulaire.

```
private void Fm_client_FormClosing(   object sender,
                                      FormClosingEventArgs e)
{
    MessageChat leMessage = new MessageChat('D', adrIpLocale,
                                                 tb_pseudo.Text);
    envoyer(leMessage);
}
```

3. Le serveur



Outre les constantes nécessaires, le serveur déclare un dictionnaire qui lui permettra de gérer la liste des clients connectés. Dès son lancement, le serveur se met en état de réception sur son port.

```
public partial class Fm_serveur : Form
{
    public Fm_serveur()
    {
        InitializeComponent();
        initReception();
    }
    private static int lgMessage = 1000;
    private static int portServeur = 33000;
    private static int portClient = 33001;
    private IPAddress adrIpLocale;
    private Socket sockReception;
    private IPEndPoint epRecepteur;
    byte[] messageBytes;
    Dictionary<IPAddress, string> clients;
    private IPAddress getAdrIpLocaleV4()
    {
      // idem…
    }
```

```
private void initReception()
{
   clients = new Dictionary<IPAddress, string>();
   messageBytes = new byte[lgMessage];
   adrIpLocale = getAdrIpLocaleV4();
   sockReception = new Socket( AddressFamily.InterNetwork,
                               SocketType.Dgram, ProtocolType.Udp);
   epRecepteur = new IPEndPoint(adrIpLocale, portServeur);
   sockReception.Bind(epRecepteur);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.BeginReceiveFrom(   messageBytes, 0, lgMessage,
                                     SocketFlags.None, ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
```

Une méthode *envoyerBroadcast* permet l'envoi d'un message sur l'ensemble du réseau.

```
private void envoyerBroadcast(MessageChat leMessage)
{
   byte[] messageBroadcast;
   Socket sockEmission = new Socket( AddressFamily.InterNetwork,
                                     SocketType.Dgram,
                                     ProtocolType.Udp);
   sockEmission.SetSocketOption(  SocketOptionLevel.Socket,
                                  SocketOptionName.Broadcast, true);
   IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
   sockEmission.Bind(epEmetteur);
   IPEndPoint epRecepteur = new IPEndPoint(IPAddress.Broadcast,
                                                   portClient);
   messageBroadcast = leMessage.GetInfos();
   sockEmission.SendTo(messageBroadcast, epRecepteur);
   sockEmission.Close();
}
```

Les messages reçus sont traités par un *BackgroundWorker* en fonction de leur type.

```
private void recevoir(IAsyncResult AR)
{
   BackgroundWorker worker = new BackgroundWorker();
   worker.DoWork += new DoWorkEventHandler(worker_DoWork);
   worker.RunWorkerCompleted +=
        new RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.EndReceiveFrom(AR, ref epTemp);
   MessageChat leMessage = new MessageChat(messageBytes);
   worker.RunWorkerAsync(leMessage);
   Array.Clear(messageBytes, 0, messageBytes.Length);
   sockReception.BeginReceiveFrom(   messageBytes, 0, lgMessage,
                                     SocketFlags.None, ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
```

```
void worker_DoWork(object sender, DoWorkEventArgs e)
{
    MessageChat leMessage = (MessageChat)e.Argument;
    switch (leMessage.TypeMessage)
    {
      case 'C':
        clients.Add(leMessage.IpEmetteur, leMessage.Texte);
        break;
      case 'E':
        string pseudo=clients[leMessage.IpEmetteur];
        MessageChat messageRetransmis =
               new MessageChat( 'R', adrIpLocale, pseudo + " -> "
                                + leMessage.Texte);
        envoyerBroadcast(messageRetransmis);
        break;
      case 'D':
        clients.Remove(leMessage.IpEmetteur);
        break;
    }
    e.Result = e.Argument;
}

void worker_RunWorkerCompleted( object sender,
                                RunWorkerCompletedEventArgs e)
{
    MessageChat leMessage = (MessageChat)e.Result;
    switch (leMessage.TypeMessage)
    {
      case 'C':
        lb_clients.Items.Add(leMessage.Texte);
        break;
      case 'D':
        lb_clients.Items.Remove(leMessage.Texte);
        break;
    }
}
```

4. Bilan

Finalement, il a du chien notre pauvre chat !

Il reste à l'améliorer :

- Faire en sorte de mieux gérer : le problème de l'unicité des pseudos, le problème des pseudos ou messages contenant un #, le problème de la fermeture du serveur (peut-être faudrait-il prévenir les clients), etc…
- Afficher la liste des clients connectés sur l'interface des clients.
- Afficher les messages sur l'interface du serveur pour permettre une modération du chat par un « superviseur ».
- Permettre aux clients de choisir une couleur d'écriture.
- Imaginer la possibilité de dialogues privés entre deux clients.
- Gérer plusieurs salons.
- Sérialiser les messages (voir plus loin).
- Envoyer les messages en multicast et non en broadcast (voir plus loin).

En clair, faites ce qui vous tente !

Un simple conseil : avant de vous lancer, réorganisez votre application pour utiliser une DLL commune au client et au serveur pour éviter la duplication du code.
Ah oui, encore une chose… Le protocole de communication doit être défini correctement avant toute étape de codage.

G/ Compléments

1. Sérialisation binaire (serialisation)

La sérialisation binaire consiste à transformer un objet en une suite de bytes. L'opération inverse, la désérialisation binaire permet de construire un objet à partir d'une suite de bytes.

Les applications « client » et « serveur » du dossier « Serialisation » reprennent l'exemple 4 et utilisent ces possibilités pour échanger des messages non balisés par un séparateur.

La classe MessageReseau

L'attribut *Serializable* indique que cette classe est sérialisable.

```
[Serializable]  // La classe est sérialisable
class MessageReseau
{
    private IPAddress ipEmetteur;
    private string texte;
    public MessageReseau(IPAddress p_ipEmetteur, string p_texte)
    {
        ipEmetteur = p_ipEmetteur;
        texte = p_texte;
    }
    public IPAddress GetIpEmetteur()
    {
        return ipEmetteur;
    }
    public string GetTexte()
    {
        return texte;
    }
```

Cette méthode statique produit un tableau de bytes à partir d'un objet MessageReseau.

```
public static byte[] Serialiser(MessageReseau p_message)
{
    MemoryStream flux = new MemoryStream();
    BinaryFormatter formateur = new BinaryFormatter();
    formateur.Serialize(flux, p_message);
    byte[] buffer = flux.GetBuffer();
    flux.Close();
    return buffer;
}
```

Cette méthode statique construit un objet MessageReseau à partir d'un tableau de bytes.

```
public static MessageReseau Deserialiser(byte[] p_buffer)
{
    MemoryStream flux = new MemoryStream(p_buffer);
    BinaryFormatter formateur = new BinaryFormatter();
    formateur.Binder = new TypeDeserialisation();
    object obj = formateur.Deserialize(flux);
    flux.Close();
    return (MessageReseau)obj;
}
```

Cette classe est utilisée pour fournir un *Binder* au formateur de la méthode précédente. En fait, les bytes contenant l'objet sérialisé contiennent également des informations concernant sa classe. La sérialisation est effectuée par l'application « Client » et la classe de l'objet est donc *Client.MessageReseau*. La désérialisation effectuée par l'application « Serveur » doit produire un objet de la classe *Serveur.MessageReseau*. Ce *Binder* et cette classe *TypeDeserialisation* seraient inutile si les deux opérations étaient effectuées par la même application, ou encore en cas d'utilisation d'une classe définie dans une dll commune (voir les applications du dossier *SerialisationCommun* pour un exemple).

```
class TypeDeserialisation : SerializationBinder
{
    // Nécessaire pour indiquer le type d'objet
    //retourné par la désérialisation
    public override Type BindToType(  string assemblyName,
                                      string typeName)
     {
         if (typeName == "Client.MessageReseau")
         {
             typeName = "Serveur.MessageReseau";
         }
         string infosType = typeName + ","
                          + Assembly.GetExecutingAssembly().FullName;
         return Type.GetType(infosType);
     }
}
```

Le client

Seule la constitution du tableau de bytes à envoyer est modifiée.

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
    byte[] messageBytes;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur =
         new IPEndPoint(IPAddress.Parse(tb_ipDestinataire.Text), 33000);
    MessageReseau leMessage = new MessageReseau(  adrIpLocale,
                                                  tb_message.Text);
    // Sérialisation de l'objet MessageReseau
    // pour le transformer en tableau de bytes
    messageBytes = MessageReseau.Serialiser(leMessage);
    sock.SendTo(messageBytes, epRecepteur);
    sock.Close();
}
```

Le serveur

Seule la création du Message lors de la réception du tableau de bytes est modifiée.

```
private void recevoir(IAsyncResult AR)
{
    BackgroundWorker worker = new BackgroundWorker();
    worker.DoWork += new DoWorkEventHandler(worker_DoWork);
    worker.RunWorkerCompleted +=
         new RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.EndReceiveFrom(AR, ref epTemp);
    // Créer un nouveau message par désérialisation des bytes reçus
    MessageReseau leMessage = MessageReseau.Deserialiser(messageBytes);
    worker.RunWorkerAsync(leMessage);
    Array.Clear(messageBytes, 0, messageBytes.Length);
    sock.BeginReceiveFrom(  messageBytes, 0, lgMessage,
                            SocketFlags.None, ref epTemp,
                            new AsyncCallback(recevoir), null);
}
```

2. Envoi de messages en multicast (ChatMulticast)

Il s'agit cette fois pour le serveur de n'envoyer les messages qu'aux hôtes connectés au chat et non à l'ensemble du réseau local.

Pour cela, il faut utiliser une adresse de multicast IP (plage 224.0.0.1 -> 239.255.255.254). Une adresse multicast identifie un groupe de machines.

- Un hôte peut s'ajouter au groupe de réception multicast.
- Un hôte peut envoyer un message à un groupe de réception multicast (il n'a pas besoin de faire partie du groupe lui-même).

Modification du client chat

Seule la méthode *initReception* est modifiée, le client s'ajoute au groupe de réception multicast.

```
private void initReception()
{
    messageBytes = new byte[lgMessage];
    sockReception = new Socket(  AddressFamily.InterNetwork,
                                 SocketType.Dgram, ProtocolType.Udp);
    epRecepteur = new IPEndPoint(adrIpLocale, portClient);
    sockReception.Bind(epRecepteur);
    IPAddress IpMulticast = IPAddress.Parse("224.168.100.2");
    MulticastOption optionMulticast =
                        new MulticastOption(IpMulticast, adrIpLocale);
    sockReception.SetSocketOption( SocketOptionLevel.IP,
                                   SocketOptionName.AddMembership,
                                   optionMulticast);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sockReception.BeginReceiveFrom(    messageBytes, 0, lgMessage,
                                       SocketFlags.None, ref epTemp,
                                       new AsyncCallback(recevoir),
                                       null);
}
```

Modification du serveur chat

La méthode *envoyerBroadcast* est remplacée par *envoyerMulticast*. Celle-ci envoie un message à destination de l'adresse multicast au lieu de l'envoyer en broadcast.

```
private void envoyerMulticast(MessageChat leMessage)
{
   byte[] messageMulticast;
   Socket sockEmission = new Socket( AddressFamily.InterNetwork,
                                     SocketType.Dgram,
                                     ProtocolType.Udp);
   IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
   sockEmission.Bind(epEmetteur);
   IPAddress IpMulticast = IPAddress.Parse("224.168.100.2");
   IPEndPoint epRecepteur = new IPEndPoint(IpMulticast, portClient);
   messageMulticast = leMessage.GetInfos();
   sockEmission.SendTo(messageMulticast, epRecepteur);
   sockEmission.Close();
}
```