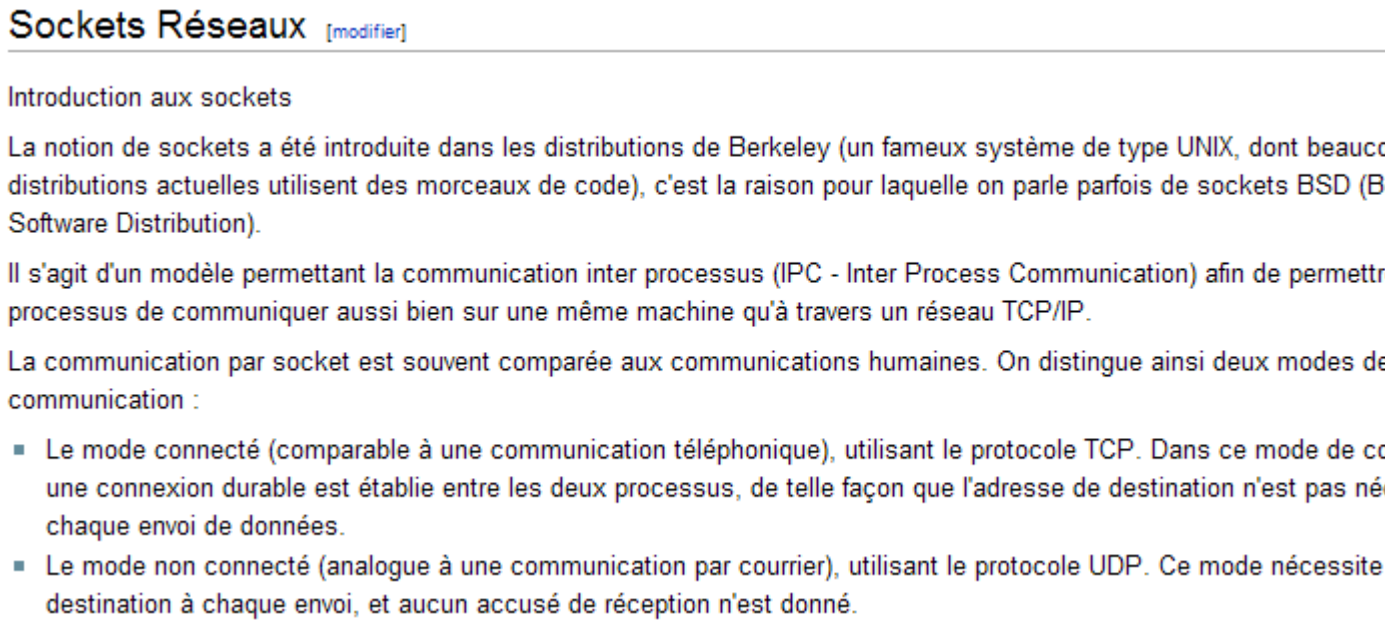
Un chat en C#

Un chat utilise un mécanisme de communication entre deux applications distantes. Il existe plusieurs solutions pour faire communiquer deux applications ; nous allons, ici, utiliser le protocole UDP. Ce protocole (couche transport) est utilisé sur internet pour des échanges en temps réel, peu sensibles, ne nécessitant pas le maintien de la connexion.

Le protocole est basé sur la notion de socket.



Sockets Réseaux [modifier]

Introduction aux sockets

La notion de sockets a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beauco distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (B Software Distribution).

Il s'agit d'un modèle permettant la communication inter processus (IPC - Inter Process Communication) afin de permettr processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket est souvent comparée aux communications humaines. On distingue ainsi deux modes de communication :

- Le mode connecté (comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de co une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas né chaque envoi de données.
- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite destination à chaque envoi, et aucun accusé de réception n'est donné.

Extrait de Wikipédia

Dotnet propose plusieurs classes pour mettre en oeuvre cette communication, de plus ou moins haut niveau (encapsulant plus ou moins de services). La classe de plus haut niveau est la classe **UdpClient**. Elle fournit des services de connexion, emission et réception de messages.

La classe UdpClient

L'utilisation de cette classe nécessite la déclaration deux namespaces :
using System.Net.Sockets;
using System.Net;

Cette classe dispose de plusieurs constructeurs surchargés, dont certains peuvent réaliser une connexion (par défaut) à un hôte distant ainsi que différentes méthodes dont :

- **Connect**, pour se connecter si ce n'est pas fait par le consructeur
- **Send**, pour envoyer des données (sous la forme d'un tableau d'octets)
- **Receive**, pour recevoir des données

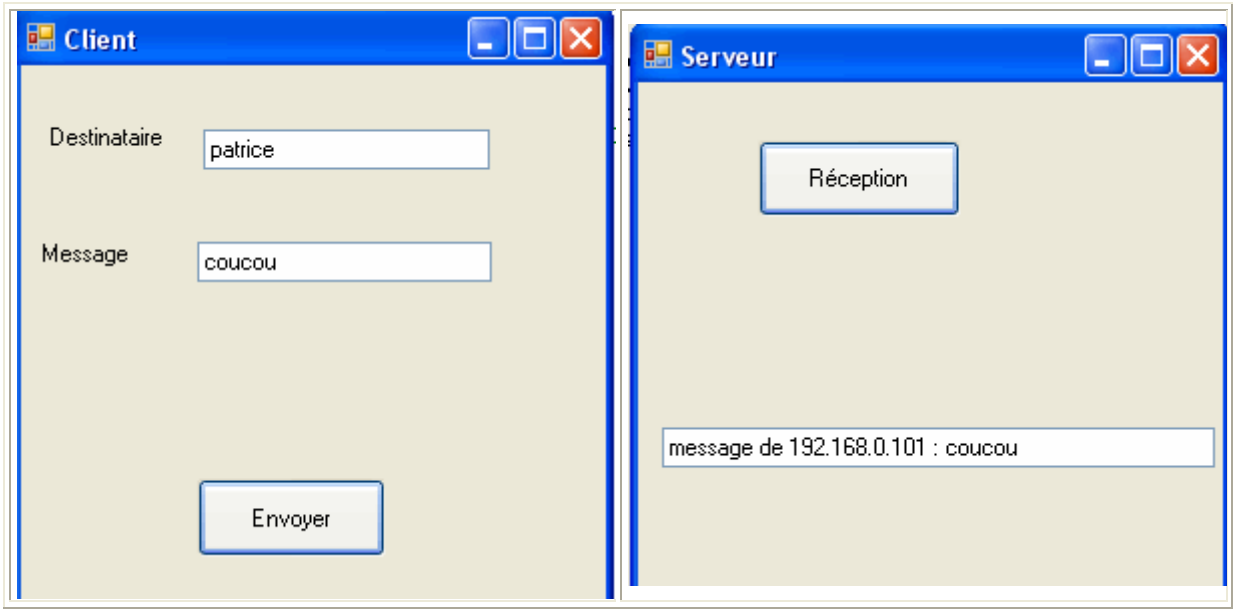
Ceci est suffisant pour envoyer un message d'une application cliente vers une application serveur

La classe IPEndPoint

Cette classe décrit un point de connexion ; couple IP/port. Elle est souvent utilisée par un objet UdpClient pour signifier le destinataire des connexions et/ou des messages.

1) Une application simpliste de message client/serveur.

Il s'agit simplement, à partir d'une application client, d'envoyer un message sur une application serveur :



1.1 L'application cliente

Le code (ici dans l'événement click) est réduit :

```
/* Ligne 1 */      UdpClient udp = new UdpClient();
/* Ligne 2 */      udp.Connect(txtDestinataire.Text, 1500);
/* Ligne 3 */      message = Encoding.Unicode.GetBytes(txtMessage.Text);
/* Ligne 4 */      udp.Send(message, message.Length);
/* Ligne 5 */      udp.Close();
```

Commentaires
Ligne 1 : création d'un objet. On pouvait également ne pas se connecter explicitement en fournissant les arguments de Connect dans la construction.
Ligne 2 : on peut utiliser l'adresse IP ou (ici) le nom DNS et le port d'écoute (1500).
Ligne 3 : le message doit être converti en tableau d'octets
Ligne 4 : tout pouvait se faire ici !! *udp.Send(message, message.Length,txtDestinataire.Text, 1500);*

1.2 L'application serveur

C'est elle qui "écoute", le code (ici dans l'événement click) est aussi réduit:

```
/* Ligne 1 */      UdpClient udp = new UdpClient(1500);
/* Ligne 2 */      IPEndPoint EmetteurIpEndPoint = new IPEndPoint(IPAddress.Any, 1500);
/* Ligne 3 */      Byte[] donneesRecues = udp.Receive(ref EmetteurIpEndPoint);
/* Ligne 4 */      string message = Encoding.Unicode.GetString(donneesRecues);
/* Ligne 5 */      string nom = EmetteurIpEndPoint.Address.ToString();
/* Ligne 6 */      txtMessageRecu.Text=("message de " + nom + " : " + message );
/* Ligne 7 */     udp.Close();
```

Commentaires
Ligne 1 : la définition du port par défaut est nécéssaire ici
Ligne 2 : création d'un point terminal de connexion ; toutes les connexions (*IPAddress.Any*) sur le port 1500 sont écoutées.
Ligne 3 : récupération des données reçues (dans un tableau de caractères) grâce à la méthode Receive. L'argument de type *IPEndPoint* est renseigné à ce moment -noter le passage par **ref**-. C'est lui qu'il faudra interroger pour avoir les informations sur l'émetteur (ligne 5)
Ligne 4 : conversion du tableau de caractères en string
Ligne 5 : interrogation du *IPEndPoint* pour obtenir l'adresse de l'émetteur

Travail à faire.
Développer les deux applications. Tester
Questions : Comment réagit l'application serveur lorsqu'elle attend un message d'un client ? que se passe t-il pour l'application serveur si on envoie plusieurs messages d'un client? Expliquez pourquoi.

1.3 Evolution : un serveur et plusieurs clients

Dans le scénario précédent l'application serveur est bloquée dans l'attente d'un message ; par aillleurs après la réception d'un message il faut reconnecter le serveur pour recevoir un nouveau message. Le fonctionnement est en mode **synchrone** (bloquant dans l'attente d'un message). Il est parfois nécessaire de mettre en oeuvre des mécanismes **asynchrones** afin de *simuler* des fonctionnement parallèles. En développement ce mécanisme est basé sur une programmation multithread. Un thread est une mini-tâche d'une application ; la programmation multithread consiste à simuler des mini-tâches parallèles afin qu'aucune ne soit bloquante pour les autres. La plupart des langages fournissent des ressources (classes ou méthodes) permettant de mettre en oeuvre ce mécanisme.

Dotnet expose des méthodes asynchrones ; elles sont préfixées **Begin** et **End** de l'équivalent des méthodes synchrones. Ainsi, *Receive* permettait de récupérer les données en mode synchrone : *BeginReceive* et *EndReceive* le feront en mode asynchrone.

Mise en oeuvre.
Ajouter une zone de liste dans l'application serveur afin de récupérer les messages.

L'application cliente est inchangée

Dans l'application serveur :

1.3.a Dans l'événement click.

```
/* Ligne 1 */  UdpClient udp = new UdpClient(1500);
/* Ligne 2 */  AsyncCallback appelAsynchrone = new
```

AsyncCallback(receptionMessage);
/ Ligne 3 */* **udp.BeginReceive(appelAsynchrone, udp);**

Commentaires :
Ligne 1 : création d'un objet d'écoute
Ligne 2 : création d'un objet pour l'appel asynchrone. L'argument du constructeur est une méthode : celle qui sera appelée à chaque réception de message
Ligne 3 : appel de la méthode asynchrone de réception. Le deuxième argument est de type Objet et laissé à la liberté du développeur ; il sera utilisé comme argument dans la méthode traiteMessage.

1.3.b la méthode receptionMessage

```
    private void receptionMessage(IAsyncResult ar)
    {
/* Ligne 1 */          UdpClient e = (UdpClient)(ar.AsyncState);
/* Ligne 2 */          IPEndPoint EmetteurIpEndPoint = new IPEndPoint(IPAddress.Any,
1500);
/* Ligne 3 */          Byte[] tabBytes = e.EndReceive(ar, ref EmetteurIpEndPoint);
/* Ligne 4 */          string message = Encoding.Unicode.GetString(tabBytes);
/* Ligne 5 */          lstMessages.Items.Add(message);
/* Ligne 6 */          AsyncCallback appelAsynchrone = new
AsyncCallback(traiteMessage);
/* Ligne 7 */          e.BeginReceive(appelAsynchrone, e);
     }
```

Ligne 1 : récupération de l'objet d'écoute de type UdpClient (cf ligne 3 plus haut)
Ligne 3 : appel d'une méthode de fin de réception qui récupère les données et valorise le point de connexion (comme pour la méthode synchrone)
Lignes 6 et 7 appel réentrant de la méthode traiteMessage à chaque nouvelle réception.

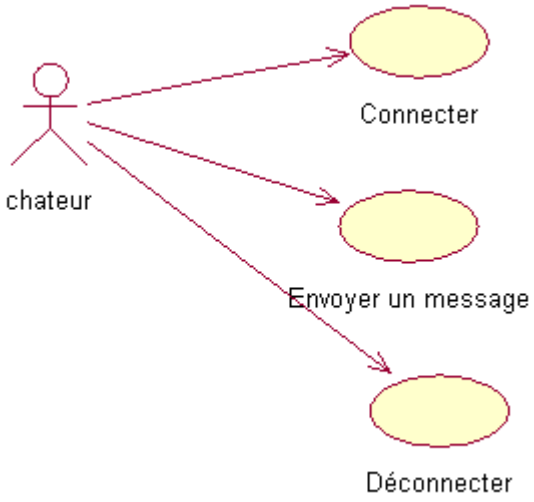
Travail à faire.
Modifier l'application en conséquence. Dans un premier temps tester en lançant sans débogage (CTRL + F5) ; envoyez plusieurs messages à partir d'un ou plusieurs clients. Dans un second temps tester l'application en mode débogage (F5) ; que constatez-vous. Suivez le conseil de débogage, effectuez la modification évoquée dans la partie *Remarque*.

2) Une application un peu plus consistante.

Nous sommes prêts maintenant à développer une application plus réaliste. Une application qui va permettre de faire converser plusieurs utilisateurs à partir d'un serveur unique.

2.1 Analyse

2.1.a 3 cas d'utilisation :



Cas d'utilisation *connecter* :
1. L'utilisateur fournit l'IP du serveur de chat et demande la connexion
2. Le système -le serveur- enregistre ce nouvel inscrit

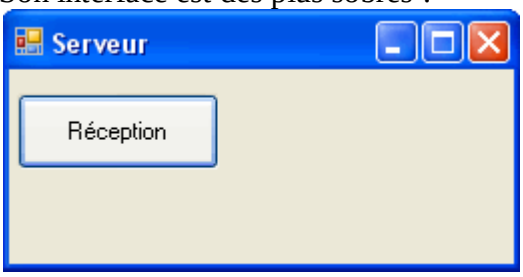
Cas d'utilisation *envoyer un message* :
1. L'utilisateur rédige et envoie un message
2. Le système -le serveur- reçoit le message et le retourne à tous les incrits en indiquant le nom de l'émetteur

Cas d'utilisation *déconnecter* :
1. L'utilisateur demande à se déconnecter ou ferme son appliction
2. Le système -le serveur- retire cet inscrit à sa liste

Deux applicationt sont nécessaires :

2.1.b L'application serveur

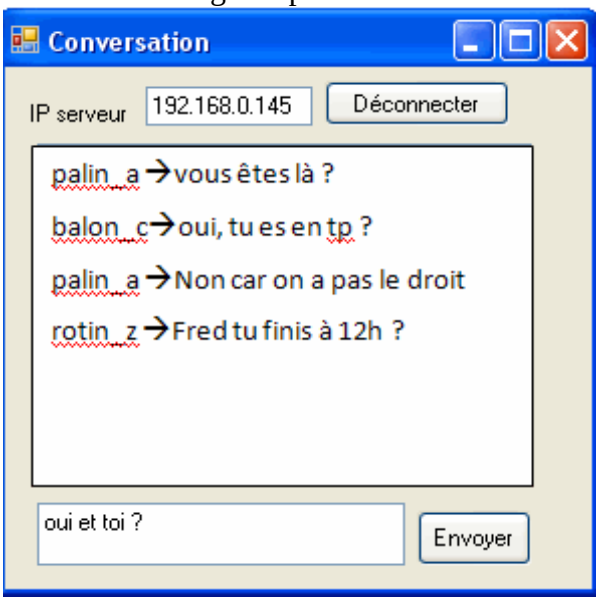
Son interface est des plus sobres !



Responsabilités :
- Enregistrer un nouvel insrit
- Dispacher chaque message reçu vers tous les inscrits
- Supprimer un inscrit, à sa demande

2.1.c L'application cliente

L'interface n'est guère plus riche



Responsabilités :
- Envoyer son userName au serveur à la connexion
- Envoyer un message au serveur
- Informer le serveur de sa déconnexion

2.2 Mise en oeuvre

2.2.a Gestion des messages.

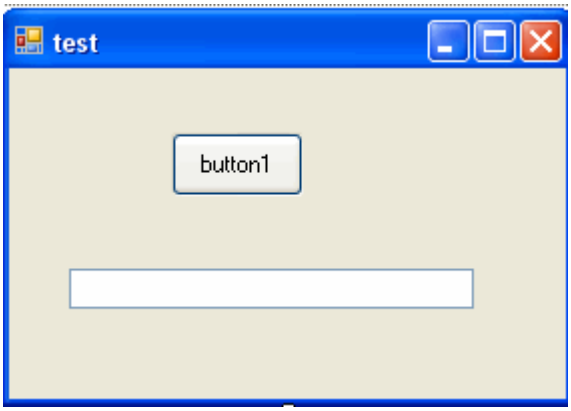
Les deux applications vont communiquer en s'échangeant des messages, de 3 natures différentes ; un message de connexion (sans contenu), un message avec un texte (le chat) et un message de déconnexion (sans contenu). La technique choisie ici est de créer une classe Message ainsi qu'un mécanisme de sérialisation/désérialisation.

La classe *Message* contient 3 champs privés :
- un champ *emetteur* (string) qui contiendra le userName
- un champ *texte* (string) correspondant au contenu envoyé
- un champ *action* (char) qui précise le type d'action : connexion ('c'), deconnexion('d'), message ('m').

Deux constructeurs :
- un avec deux arguments, *emetteur* et l'*action*
- un autre avec deux arguments, *emetteur* et *texte*.

Les 3 accesseurs sur les champs privés.

Travail à faire.
Créer un nouveau projet : le client. Créer un formulaire correspondant à celui présenté plus haut.
Ajouter un nouveau formulaire (test) de test qui ne contiendra qu'un bouton et une zone de texte et qui servira pour les tests.



Modifier dans le fichier Program.cs le formulaire lancé :
Application.Run(new test());
Tester.
Ajouter la nouvelle classe *Message* au projet ; tester dans le formulaire *test*.

Le mécanisme de sérialisation.

Une classe SerializeMessage se chargera, grâce à deux méthodes statiques de transformer un tableau de bytes en objet Message et une autre qui fait le travail inverse. On vous fournit une partie du code.

```
class SerializeMessage
{
        public static Byte[] toBytes(Message m)
       { /* code à écrire*/


        }
        public static Message getMessage(byte[] bytes)
        {
            MemoryStream flux = new MemoryStream(bytes);
            BinaryFormatter bf = new BinaryFormatter();
             bf.Binder = new DeserializeBinder();
             object o = bf.Deserialize(flux);
             flux.Close();
              return (Message)o;
          }
public class DeserializeBinder : SerializationBinder
{
          public override Type BindToType(string assemblyName,string typeName)
           {
                  return typeof(Message);
            }
}
```

Commentaire
La classe *DeserializeBinder* est nécessaire pour la situation où l'application qui sérialise n'est pas celle qui désérialise. Elle n'est pas à utiliser dans le code que vous avez à écrire.

Travail à faire.
Ajouter une classe au projet : SerializeMessage. Remplcer le code généré par celui fourni juste au dessus. Ecrire le code de la méthode *toBytes*. Tester avec le code suivant :

```
private void button1_Click(object sender, EventArgs e)
{
        Message m = new Message("toto", "bonjour");
         Byte[] bytes = SerializeMessage.toBytes(m);
        Message m1;
         m1 = SerializeMessage.getMessage(bytes);
       string s = "emetteur : " + m1.getEmetteur() + " action : " + m1.getAction().ToString() +
" texte : " + m1.getTexte();
       MessageBox.Show(s);
}
```

2.2.b L'application cliente

Des attributs privés peuvent être déclarés :

```
private IPAddress IPServeur;
private int portEmission = 1500;
private int portReception = 1501;
private UdpClient udpReception;
private UdpClient udpEmission;
```

Trois points d'intervention :
a) Sur l'événement click de connexion
- Construction d'un objet IPAdress à partir de l'adresse IP fournie
- Récupération du userName de l'utilisateur ; demander le service au DNS
- Création d'un objet UdpClient et connexion à partir de lIPAdress du serveur
- Création et envoie au serveur d'un message de connexion
- Mise en oeuvre du processus d'écoute vue dans la première partie ; utiliser un autre objet UdpClient et une autre valeur du port

b) Une méthode privée d'envoie de message
private void envoyer(Message m)

c) La méthode d'écoute
private void receptionMessage(IAsyncResult ar)
qui va charger les messages (sérialisés) reçus dans la liste déroulante

Attention à la gestion des objets de connexion *udpEmission* et *udpReception* a utiliser
La deconnexion peut être gérée sur l'évement click de connexion également qui passe à l'état déconnexion après une connexion. Elle peut être en plus faite à la fermeture du formulaire.

Travail à faire.
Développer le formulaire

2.2.c L'application serveur

Des attributs privés peuvent être déclarés :

```
private int portEmission = 1501;
```

```
private int portReception = 1500;
private UdpClient udpReception;
private UdpClient udpEmission;
private ArrayList lesConnectes;
```

Trois points d'intervention :

a) Sur l'événement click de connexion
- Création du UdpClient de réception
- Création de l'ArrayList
- Mise en oeuvre du processus d'écoute vue dans la première partie

b) La méthode d'écoute
private void receptionMessage(IAsyncResult ar)
c) Elle appellera une méthode privée :
private void traiteMessage(Byte[] bytes)
qui récupèrera le message (sérialisation), et selon la valeur du champ action :
- cas 'c' : ajout dans la liste des inscrits
- cas 'd' : suppression de la liste des inscrits
- cas 'm' : envoie du message à tous les inscrits

Travail à faire.
Développer le formulaire