TP C#11 : NetCat !

Consignes de rendu

À la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
– login_x.zip
    |– login_x/
        |– AUTHORS
        |– README
        |– MyNetCat/
            |– MyNetCat.sln
            |– Client/
                |– Tout sauf bin/ et obj/
            |– ClientCLI/
                |– Tout sauf bin/ et obj/
            |– LibNetCat/
                |– Tout sauf bin/ et obj/
            |– Server/
                |– Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login_x* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre :

- Le fichier AUTHORS doit être au format habituel : ** login_x$* où le caractère '$' représente un retour à la ligne.
- Le fichier README doit contenir les difficultés que vous avez rencontrées sur ce TP, et indiquera les bonus que vous avez réalisés.
- Pas de dossiers bin ou obj dans le projet.
- **Le code DOIT compiler !**

1 Cours

1.1 Socket

Les **sockets** sont des interfaces de connexion permettant la communication entre processus : que ce soit sur la même machine ou sur une ou plusieurs machines connectées à un même réseau.

Cette communication peut se faire dans les deux sens : sur un même socket, on peut à la fois envoyer et recevoir des données. Il existe d'autres interfaces (par exemple le tube/pipe) qui ne permettent de communiquer que dans un seul sens : on peut soit envoyer, soit recevoir des données, mais pas les deux avec le même outils.
Les sockets que nous allons utiliser sont des sockets Internet, c'est à dire qu'elles se basent sur l'*Internet Protocol*, ou **IP**. La grande majorité des sockets sont des sockets Internet, qui sont utilisées pour communiquer par le réseau. Il existe d'autres types de sockets, parmi lesquels on trouve les sockets Unix, qui servent à communiquer localement sur une même machine, mais nous ne les verrons pas dans ce TP.
Un socket Internet posséde les caractéristiques suivantes : **une adresse** locale (*local socket address*) constituée d'une **adresse IP** identifiant la machine et d'un **numéro de port** identifiant l'application qui utilise le socket (ce numéro de port est important car on veut que plusieurs applications puissent utiliser des sockets en même temps, ce qui serait impossible avec seulement l'adresse IP).
Les adresses des sockets sont représentées de la manière suivante : <IP> :<port>.
Par exemple l'adresse 92.29.12.12 :4242, où 92.29.12.12 est l'adresse IP et 4242 est le numéro de port.

Une adresse distante (*remote socket address*) constituée, comme l'adresse locale, d'une adresse IP et d'un numéro de port, mais qui permet d'identifier la machine distante (celle à laquelle on est connecté) plutôt que la machine locale. **Un protocole de transport** qui détermine, entre autres, le format des paquets de données échangés. Les deux principaux protocoles de transport sont les suivants :

— **TCP** (*Transmission Control Protocol*) garantit que les paquets seront livrés et qu'ils le seront dans le bon ordre en les vérifiant, les corrigeant, en les renvoyant s'ils sont perdus et en envoyant des "accusés de réception". Tout ce traitement supplémentaire peut ralentir la communication mais garantit sa fiabilité. C'est ce protocole que nous utiliserons aujourd'hui.

— **UDP** (*User Datagram Protocol*) est plus rapide que TCP puisqu'il effectue très peu de vérifications et de corrections sur les paquets : il ne garantit ni qu'ils arriveront bien à destination ni qu'ils arriveront dans le bon ordre. Il est souvent utilisé dans les applications sensibles au facteur temps (par exemple, les jeux en ligne) car il est préférable de perdre occasionnellement des paquets (ce qui cause le fameux phénomène du lag) que d'attendre qu'ils arrivent (ce qui augmenterait la latence des joueurs).

1.2 La classe Socket

Le framework .NET fournit une classe Socket, située dans le namespace *System.Net.Sockets*, permettant de manipuler une socket. Nous vous invitons d'ores et déjà à ouvrir la documentation de cette classe, histoire de l'avoir à portée de main.

Créer un socket

Pour créer un socket, il faut l'instancier de la manière suivante :

```
1 public Socket(
2   AddressFamily addressFamily,
3   SocketType socketType,
4   ProtocolType protocolType)
```

Les paramètres de ce constructeur sont des énumérations, dépendantes les unes des autres :

— *addressFamily* : la famille d'adresses de la socket. Nous utiliserons *AddressFamily.InterNetwork* pour avoir une addresse IP.

— *socketType* : le type de la socket. Nous utiliserons *SocketType.Stream* pour avoir un flux de données fiable à deux sens.

— *protocolType* : le protocole de la socket. Nous utiliserons *ProtocolType.Tcp*.

Pour résumer, pour créer un socket :

```
1 Socket socket = new Socket(
2   AdressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

Une fois qu'on a fini d'utiliser un socket, il faut le fermer (pour terminer la connexion et libérer les ressources utilisées par le socket). Pour cela, on utilise tout bêtement la méthode *Close* :

```
1 socket.Close();
```

Côté Client

Puisque nous utilisons TCP, il faut d'abord établir une connexion avec la socket distant avant de pouvoir commencer à échanger des données.
Pour cela, on utilise la méthode *Connect* de l'objet Socket :

```
1 public void Connect(IPAddress address, int port)
```

Par exemple :

```
1 IPAddress address = IPAddress.Parse("92.29.12.12");
2 try
3   socket.Connect(address, 4242);
4 catch(Exception e)
5   Console.WriteLine("Erreur de connexion :" + e.Message);
```

Côté Serveur

Pour pouvoir accepter des connexions de sockets client, le socket serveur doit d'abord "écouter" sur un port. On utilise pour cela les méthodes *Bind* et *Listen* :

```
1 public void Bind(IPEndPoint ipEndPoint)
2 public void Listen(int backlog)
```

La méthode *Bind* associe le socket à un endpoint (une adresse et un port), représenté par la classe *IPEndPoint*.

La méthode *Listen* fait "**écouter**" la socket sur l'endpoint défini précédemment. Autrement dit, elle attend des connexions entrantes de clients. L'entier que la méthode prend en argument indique le nombre de connexions qu'on peut mettre dans la file d'attente d'acceptation. Dans notre cas, cela n'a pas beaucoup d'importance, on utilisera donc une valeur arbitraire ($>= 1$).

Une fois que le socket écoute, il ne peut plus servir à autre chose (il ne peut donc pas envoyer ni recevoir de données). Cependant, quand il recevra des demandes de connexion de client, il pourra les **accepter** et créer d'autres sockets dédiés à la communication avec ces clients. Pour cela, on utilise la méthode *Accept* :

```
1 public Socket Accept()
```

Cette méthode retourne un socket connecté au client, dont on pourra se servir pour communiquer avec ce dernier. Un exemple concret :

```
1 IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 4242);
2 try
3   serverSocket.Bind(endPoint);
4 catch
5   Console.WriteLine("Erreur de liaison: " + e.Message);
6 serverSocket.Listen();
7 // Accept bloque le programme jusqu'à la connexion d'un client
8 Socket clientSocket = serverSocket.Accpet();
```

Gestion des données

Pour envoyer et recevoir des données, on utilise respectivement les méthodes *Send* et *Receive* de la classe Socket :

```
1 public int Send(byte[] buffer)
2 public int Receive(byte[] buffer)
```

La méthode *Send* prend en paramètre un tableau de **bytes** (octets, qui sont des données binaires) contenant les données à envoyer, et retourne un entier indiquant le nombre d'octets qu'elle a pu envoyer.

Évidemment, en temps normal, nous ne manipulons pas les données par *byte*, mais en types connus (*int*, *string*...). Pour pouvoir utiliser la méthode *Send*, il nous faut donc les convertir en tableaux de *bytes*. Cela reste simple en C# :

— Pour les chaînes de caractères, qui sont un cas un peu spécial, nous utiliserons la méthode *Encoding.UTF8.GetBytes*, située dans le namespace *System.Text* :

```
1 byte[] data = System.Text.Encoding.UTF8.GetBytes("Hello World!");
```

— Pour les autres types de données, nous utiliserons les méthodes de la classe *BitConverter*. Nous vous invitons fortement à jeter un œil à sa documentation pour voir la liste des conversions disponibles. Au cas où, nous vous fournissons quelques exemples :

```
byte[]  intData = BitConverter.GetBytes(4242);
byte[]  floatData = BitConverter.GetBytes(42.0);
byte[]  charData = BitConverter.GetBytes('a');
byte[]  boolData = BitConverter.GetBytes(true);
```

Un exemple complet pour vous montrer comment envoyer une chaîne de caractères par le réseau :

```
// La  chaîne à envoyer
string  message = "This message is toasted.";

// On envoie la longueur du message pour qu'il soit reçu correctement
// Note: un caractère = un octet
byte[]  messageLength = BitConverter.GetBytes(message.Length);
socket.Send(messageLength);

// Puis on envoie la chaîne
byte[]  messageData = System.Text.Encoding.UTF8.GetBytes(message);
socket.Send(messageData);
```

La méthode *Receive* prend également en paramètre un tableau de *bytes*, mais elle y écrira les données qu'elle reçoit. Il faut donc prévoir un tableau de taille suffisante pour pouvoir recevoir tout ce qui a été envoyé (sinon les données pour lesquelles il n'y avait pas de place seront mises de côté jusqu'au prochain appel de **Receive**, dénaturant ainsi le message). Elle renvoie elle aussi le nombre d'octets reçus, ce qui permet de vérifier que la donnée est conforme.

Là encore, il nous faut convertir des données mais dans l'autre sens, pour retrouver les types de données auxquels nous sommes habitués :

— Pour les chaînes de caractères, on utilisera *Encoding.UTF8.GetString* :

```
byte[] receivedData;
...
string s = System.Text.Encoding.UTF8.GetString(receivedData);
```

— Pour les autres types, on utilisera à nouveau *BitConverter* :

```
int n = BitConverter.ToInt32(intData , 0);
float f = BitConverter.ToSingle(floatData , 0);
char c = BitConverter.ToChar(charData , 0);
bool b = BitConverter.ToBoolean(boolData , 0);
```

Un exemple complet pour vous montrer comment recevoir la chaîne de caractères envoyées plus haut :

```
// On veut lire un entier de 32 bits (4 octets)
// représentant le nombre d'octet à lire
byte[]  messageLengthData = new  byte [4];
socket.Receive(messageLengthData );
int  messageLength = BitConverter.ToInt32(messageLengthData , 0);
// On lit la  chaîne  de  messageLength  octets
byte[]  messageData = new  byte[messageLength ];
socket.Receive(messageData );
string  message = System.Text.Encoding.UTF8.GetString(messageData );
Console.WriteLine(message );
// "This message is toasted."
```

1.3 Protocole

Quand des machines d'échangent des données par le réseau, il faut qu'elles soient d'accord sur le format de ces données, pour savoir comment les lire et les interpréter : c'est ce qu'on appelle un **protocole de communication**.

Dans l'exemple d'envoi et de réception d'une chaîne de caractères ci-dessus, on a utilisé un protocole. En effet, à l'envoi, on a d'abord écrit la longueur du message, puis le message lui-même. À la réception, on a d'abord lu la longueur du message, puis le message lui-même. L'expéditeur et le destinataire du message étaient d'accord sur le format des données échangées.

À titre d'exemple, on aurait pu utiliser un protocole dans lequel, plutôt que d'être précédés par leur longueur, tous les messages doivent se terminer par la chaîne "EOD" (ou tout autre message) pour indiquer leur fin. L'expéditeur aurait alors envoyé son message, puis la chaîne "EOD" pour indiquer la fin du message. Le destinataire aurait ensuite lu des données jusqu'à tomber sur la chaîne "EOD". Évidemment, cela signifie qu'il n'est pas possible d'envoyer un message contenant "EOD", sous peine de le voir coupé en deux, mais c'est un protocole tout à fait valide, même si peu pratique.

On peut distinguer parmi ces deux exemples deux types de protocole :

— Le protocole "EOD" est un **protocole textuel** : on n'envoie que du texte, avec une signification particulière (ici, le "EOD" signifie "fin des données"). Les protocoles textuels sont généralement plus faciles à écrire et à débugger, mais la conversion en texte de certaines données peut s'avérer coûteuse.

— À l'inverse, le protocole consistant à envoyer la longueur du message avant le message est un **protocole binaire** : on envoie des données binaires, parfois d'une taille déterminée (par exemple, les 4 octets de la longueur du message), dans un ordre précis (longueur avant texte). Les protocoles binaires sont très adaptés aux échanges de données numériques et bien sûr binaires, même s'ils sont plus longs à écrire et moins faciles à débugger.

1.4 Interruption de programme

En console, on peut interrompre les programmes avec **Ctrl-C**. Malheureusement, par défaut, le programme se termine immédiatement, ce qui ne permet pas de libérer les ressources (par exemple, en fermant les sockets ouverts). Pour remédier à ce problème, nous allons intercepter l'évènement produit par **Ctrl-C** pour faire notre propre nettoyage avant de terminer. Nous vous fournissons un exemple de code. Par soucis de simplicité, nous ne détaillerons pas le concept d'événement.

```
1  public class MyClass
2  {
3    private static bool _running = true;
4
5    public static void Main()
6    {
7      // On attend l'événement d'intérruption
8      Console.CancelKeyPress +=
9              delegate(object sender, ConsoleCancelEventArgs e)
10     {
11       e.Cancel = true; //On annule l'interruption
12       _running = false; //On arrête la boucle
13     };
14     Socket s = ...;
15     while (_running)
16     {
17        ...
18     }
19     s.Close();
20   }
21 }
```

2 Exercices

L'objectif de ce TP est de coder un système de chat basique basé sur une architecture client/serveur. Les différentes commandes du protocole sont définies dans le fichier "*Instructions.cs*" de *LibNetCat*. Tous les différents types de messages transmis entre les clients et le serveur ont la même structure : un premier entier non signé de 16 **bits** (**UInt16**) correspondant à la structure de commande ci-dessous est envoyé puis selon le type de commande, le programme réagi différamment.

```
1  public enum Instructions : ushort
2  {
3      Ping,
4      BasicMessage,
5      FileTransfert,
6      Pong,
7      Disconnect
8  }
```

- *Ping*
  Cette commande est une requette qui se suffit à elle même, celui qui la reçoit doit juste renvoyer "*Instructions.Pong*".
- BasicMessage
  Cette commande est suivie d'un **Int64** contenant la taille du message (en **octet**) suivant.
- *FileTransfert*
  Cette commande est un peu particulière et sera expliquée en détail plus tard.
- *Pong*
- *Disconnect*
  Cette commande qui n'est suivie de rien indique que le socket va être fermé.

Comme beaucoup de code est en commun entre les clients et le serveur, votre TP est découpé en 4 sous-projets :

- *Client* : qui est une interface graphique du client.
- *ClientCLI* : qui est une interface en ligne de commande du client (pour que vous puissiez faire des tests)
- *Server* : Qui est le serveur
- *LibNetCat* : Qui va contenir la majorité de votre code et qui sera partagé entre toutes les autres applications.

La réception et l'envoi de données se fait exclusivement dans le *Socket* contenu dans *Client.cs*.

2.1 Créer le socket

Cette partie est la plus simple, il vous est demandé de coder la partie *Ping* du protocole. Vous devez créer un socket en écoute sur un port dans le serveur et qui va attendre un *Ping* pour répondre *Pong* (vous devez donc compléter la méthode *Ping()* dans *Client.cs*). Quand nous recevons un *Pong*, nous devons le signaler à l'application, c'est là que les évènements entrent en jeu, la majorité du code est déjà faite pour vous vous devez donc juste comprendre comment cela fonctionne.

```
1  public event EventHandler MonEvent; // Ceci est votre Handler d'events
2  public void Receiver(object sender, EventArg e)
3  {
4          Console.WriteLine("Wooo");
5  }
6  public void Main()
7  {
8      // Ici nous abonnons la méthode Receiver à MonEvent
9      MonEvent += Receiver;
10
11     if (MonEvent != null) {// Nous vérifions que MonEvent à des abonnés
12     // Nous appelons toutes les méthodes abonnées
13             MonEvent(this, EventArgs.Empty);
14         /**
15         ** Quand nous aurons des Handlers plus complexes,
16         ** nous remplacerons EventArgs.Empty par un objet
17         ** contenant des données et héritant de EventArgs.
18         **/
19     }
20 }
```

Vous devez donc lancer l'évènement *PongEvent* si vous recevez un message de type *Instructions.Pong*.

2.2 Gérer les déconnexions

Vous devez gérer le *Disconnect* côté serveur, un client doit pouvoir se connecter, se déconnecter puis se reconnecter sans avoir à relancer le serveur (vous devez donc compléter la méthode *Disconnect()* dans *Client.cs*). Vous devez aussi envoyer le signal *Disconnected* quand il faut.

2.3 Plus de clients...

Cette partie est très importante, vous allez découvrir les méthodes asynchrones ce qui va vous permettre de gérer plusieurs clients en même temps.
Les méthodes asynchrones sont implémentées sous la forme de deux méthodes nomées *BeginAction* et *EndAction*. Nous appelons *BeginAction* avec une méthode "*CallBack*" pour démarer *Action* sans stopper le fonctionnement du programme. La méthode "*CallBack*" est alors appelée automatiquement à la fin de l'*Action* où, elle appellera enfin *EndAction* pour finaliser l'*Action*. Bref un exemple est plus parlant.

```
1  public void Toto()
2  {
3    socket.BeginReceive(buffer, 0, buffer.Length,
4                        SocketFlags.None, CallBackMth, socket);
5  }
6
7  private void CallBackMth(IAsyncResult ar)
8  {
9    if (!ar.IsCompleted)
10     return;
11   socket.EndReceive(ar);
12   /**
13   ** Nous ne savons pas quand nous arriverons ici
14   ** Mais le buffer est maintenant rempli.
15   */
16 }
```

Maintenant vous devez attendre un message avec *BeginReceive* dans le constructeur de *Client* (qui va "appeler" *ReceiveDatas*). Votre serveur doit aussi attendre les clients avec *BeginAccept*.

2.4 BasicMessage

Nous allons enfin pouvoir envoyer et recevoir des vrais messages. Vous savez comment faire (les messages sont codés en **UTF8**). La méthodologie est de récupérer la taille du message (avec *Socket.Receive*), de créer un nouveau buffer *_content* de cette taille, puis de récupérer le reste du message avec *Socket.BeginReceive* qui appelera *ReceiveBasicMessage* qui va envoyer un *MessageEvent*.

```
1 if (MessageReceived != null)
2   MessageReceived(this, new MessageEvent("Un message"));
```

La Méthode SendMessage est aussi à implémenter.

2.5 FileTransfert

Cette partie est un peu plus compliquée, il ne vous est donc pas demandé de la gérer de manière asynchrone. La communication s'organise de la façon suivante :

- Un **Int16** est envoyé contenant la taille du nom du fichier.
- Un buffer contenant le nom du fichier est envoyé (celui qui réceptionne le fichier doit ouvrir un *BinaryWriter* dans le répertoire courant).
- Un objet de type *TransfertEvent* est créé (et un premier event *TransfertProgress* est envoyé)
- Un **Int64** est envoyé contenant la taille du fichier.
- Le fichier est alors envoyé découpé en buffers de 128 **octets** (moins pour le dernier). A la réception de chaque buffer, l'objet *TransfertEvent* est mis à jour puis un TransfertProgress envoyé
- Enfin l'objet *TransfertEvent* est mis à jour puis envoyé pour signaler la fin du transfert.

2.6 Bonus

Soyez imaginatifs :). Sinon un socket est ouvert sur **xdbob.net :4242** à vous de trouver le flag (à mettre dans votre readme avec la procédure que vous avez effectué pour l'obtenir).

3 Conclusion

Si vous avez des questions concernant le "protocole" vous pouvez contacter **Antoine Damhet** à **antoine.damhet [at] epita.fr** avec l'objet "**[acdc][NetCat] votre question**".
Nous ne répondrons pas aux questions posées le dimanche du rendu !