TP C#8 : MyNetcat

1 Consignes de rendu

L'architecture de rendu est la suivante :

```
-- rendu-tp-login_x.zip
   |-- login_x/
       |-- AUTHORS
       |-- BONUS (facultatif)
       |-- MyNetcat/
           |-- Solution du projet
       |-- Crackme/ (facultatif)
           |-- flags.txt
           |-- Crackme.cs
```

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
- pas de dossiers `bin` ou `obj` dans le projet
- et surtout, **le code doit compiler**

Si vous faites des bonus, vous **devez** les lister dans le fichier `BONUS`. Cette consigne n'est pas là pour vous embêter mais pour éviter que nous passions à côté de vos bonus lors de la correction.

2 Introduction

Dans ce TP, nous allons aborder la programmation réseau en C#. Par soucis de simplicité (le sujet étant assez vaste), nous ne ferons pas de véritable introduction aux réseaux. En revanche, nous verrons tout ce qui vous sera nécessaire à la réalisation du TP, durant lequel vous écrirez une version simplifiée de Netcat, un utilitaire permettant assez simplement d'échanger des données par le réseau.

3 Cours

3.1 Les sockets

Une **socket** est une interface permettant la communication d'une ou plusieurs applications tournant soit sur la même machine, soit sur une ou plusieurs machines connectées à un même réseau.

Cette communication peut se faire dans les deux sens : sur une même socket, on peut à la fois envoyer et recevoir des données. Il existe d'autres interfaces (par exemple, les pipes) qui ne permettent de communiquer que dans un seul sens : on peut soit envoyer, soit recevoir des données, mais pas les deux à la fois.

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, vous l'avez deviné, 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.

Une 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 la socket (ce numéro de port est important car on veut que plusieurs applications puissent utiliser des sockets en même temps, ce qui ne serait

pas possible avec seulement l'adresse IP, qui ne permet pas de départager).

Pour parler en métaphore : imaginons que la machine soit un immeuble résidentiel et que ses applications soient des habitants de cet immeuble.
Pour localiser l'immeuble, on utilise son adresse physique : numéro, nom de rue, code postal, ville, etc. Pour localiser la machine, on utilise son adresse IP.
Pour contacter un des habitants de l'immeuble, on laisse un message dans sa boîte aux lettres, dont on connaît le numéro. Pour communiquer avec une des applications de la machine, on envoie des données sur son port, dont on connaît le numéro.

On représente souvent les adresses de socket de la manière suivante : `<adresse IP>:<numéro de port>`. Par exemple, `13.37.13.37:4242`, où `13.37.13.37` est l'adresse IP et `4242` 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 vidéo 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).

3.2 La classe `Socket`

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

3.2.1 Créer une socket

Pour créer une socket, il faut l'instancier. On utilisera le constructeur suivant :

```
public Socket(
    AddressFamily addressFamily,
    SocketType socketType,
    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`.

En résumé, pour créer une socket :

```
Socket socket = new Socket(
    AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp
);
```

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

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

3.2.2 Côté client

Puisque nous utilisons TCP, il faut d'abord établir une connexion avec la socket distante avant de pouvoir commencer à échanger des données.

Pour cela, on utilise la méthode `Connect` de l'objet `Socket` :

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

Par exemple :

```
IPAddress address = IPAddress.Parse("13.37.13.37");

try {
    socket.Connect(address, 4242);
} catch (Exception e) {
    Console.WriteLine("Erreur lors de la connexion: " + e.Message);
}
```

3.2.3 Côté serveur

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

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

La méthode `Bind` associe la 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 de recevoir des demandes de connexion 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 la socket écoute, elle ne peut plus servir à autre chose (elle ne peut donc pas envoyer et recevoir de données). Cependant, quand elle recevra des demandes de connexion de client, elle pourra les **accepter** et créer de nouvelles sockets dédiées à la communication avec ces clients. Pour cela, on utilise la méthode `Accept` :

```
public Socket Accept()
```

Cette méthode retourne une socket connectée au client, dont on pourra se servir pour communiquer avec lui.

Un exemple complet :

```
IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 4242);

try {
    serverSocket.Bind(endPoint);
} catch (Exception e) {
    Console.WriteLine("Erreur lors du binding: " + e.Message);
}

serverSocket.Listen();

// La methode Accept va bloquer l'execution du programme jusqu'a ce qu'un
// client se connecte
Socket clientSocket = serverSocket.Accept();
```

On peut maintenant utiliser `clientSocket` pour communiquer avec le client.

3.2.4 Envoyer et recevoir des données

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

```
public int Send(byte[] buffer)
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 de données de type `byte`, mais des `int`s ou des `string`s. Pour pouvoir utiliser la méthode `Send`, il nous faut donc les convertir en tableaux de `byte`s.

Heureusement, c'est très 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` :

```
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);
```

Comme vous pouvez le voir, les différentes variantes de la méthode `GetBytes` font le café.

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

```
// La chaine a envoyer
string message = "What hath God wrought?";

// On envoie d'abord la longueur de la chaine, pour que le destinataire
// puisse savoir combien d'octets il doit lire.
// Note: puisque qu'un caractere tient sur un octet, message.Length est egal
// au nombre d'octets de la chaine
byte[] messageLength = BitConverter.GetBytes(message.Length);
socket.Send(messageLength);

// Puis on envoie la chaine elle-meme
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`, ce qui n'est bien évidemment pas très pratique). Elle renvoie elle aussi le nombre d'octets reçus, ce qui permet de vérifier que tout est bien arrivé.

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) representant le nombre
// d'octets a lire
byte[] messageLengthData = new byte[4];
socket.Receive(messageLengthData);
int messageLength = BitConverter.ToInt32(messageLengthData, 0);

// On lit la chaine de messageLength octets
byte[] messageData = new byte[messageLength];
socket.Receive(messageData);
string message = System.Text.Encoding.UTF8.GetString(messageData);

Console.WriteLine(message);
// "What hath God wrought?"
```

3.3 Les protocoles

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**, ou protocole pour faire simple.

Dans l'exemple d'envoi et de réception d'une chaîne de caractères ci-dessus, on a, même si vous ne vous en êtes pas rendus compte, 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 "TOTO" pour indiquer leur fin.

L'expéditeur aurait alors envoyé son message, puis la chaîne "TOTO" pour indiquer "fin du message". Le destinataire aurait ensuite lu des données jusqu'à tomber sur la chaîne "TOTO".

Évidemment, cela signifie qu'il n'est pas possible d'envoyer un message contenant "TOTO", sous peine de le voir coupé en deux, mais c'est un protocole tout à fait valide, même si assez peu pratique.

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

- Le protocole "TOTO" est un protocole **textuel** : on n'envoie que du texte, avec une signification particulière (ici, le "TOTO" 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.

3.4 Gestion du Ctrl-C

En mode console, on peut stopper les programmes en tapant 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 ouvertes).

Pour remédier à ce problème, nous allons intercepter l'évènement produit par Ctrl-C pour faire notre nettoyage avant de quitter comme demandé. Nous vous fournissons un exemple de code. Par soucis de simplicité, nous ne détaillerons pas le concept des évènements.

```
public class MyClass {
    private static bool _keepRunning = true;

    public static void Main() {
        Console.CancelKeyPress +=
            delegate(object sender, ConsoleCancelEventArgs e) {
                e.Cancel = true;      // On annule l'arret du programme
                _keepRunning = false; // On demande a la boucle de s'arreter
            };

        Socket socket = ...;
        while (_keepRunning) {
            // ...
        }
        socket.Close();
  }
}
```

4 Exercices

4.1 MyNetcat

Comme nous l'avons expliqué précédemment, nous allons maintenant écrire une version simplifiée de Netcat. Nous nous contenterons d'écrire un transfert de fichier, même si Netcat est en réalité capable de faire beaucoup plus que cela [1].

4.1.1 Palier 1 : gestion des arguments en ligne de commande

Nous allons commencer par gérer les arguments en ligne de commande.

Nous nous servirons du même programme pour gérer le client et le serveur (autrement dit, le client et le serveur seront contenus dans le même exécutable). Pour distinguer les deux modes, nous nous servirons des arguments passés au programme :

Client Le client se lance simplement en précisant successivement l'IP du serveur auquel on veut se connecter, son port, et optionnellement le nom du fichier dont on veut transférer le contenu. En l'absence de ce nom de fichier, le client lit les données à envoyer sur l'entrée standard, ligne par ligne.

Par exemple :

```
C:\tp> mynetcat 127.0.0.1 1337 file.txt
C:\tp> mynetcat 127.0.0.1 1337
Je tape cette ligne directement dans la console, et elle sera envoyee au
serveur des que j'appuierai sur Entree.
```

Serveur Dans le cas du serveur, on donne d'abord l'option `-l` (L minuscule, comme *listen*) suivie du numéro du port sur lequel on va écouter, puis le nom du fichier dans lequel on va écrire le contenu qu'on reçoit. De la même manière que pour le client, si on ne précise pas de nom de fichier, on écrit les données reçues sur la sortie standard.

Par exemple :

```
C:\tp> mynetcat -l 1337
Ceci est le contenu de file.txt envoye dans l'exemple ci-dessus
^C
C:\tp> mynetcat -l 1337 output.txt
C:\tp> type output.txt
Je tape cette ligne directement dans la console, et elle sera envoyee au
serveur des que j'appuierai sur Entree.
```

Note : le serveur doit continuer de tourner même lorsque le client s'est déconnecté. C'est pour cela que, pour le relancer en rajoutant `output.txt` en paramètre, on l'arrête avec Ctrl-C à la troisième ligne de l'exemple.

Vous noterez que les modes d'entrée/sortie pour le client et le serveur sont indépendants : ce n'est pas parce qu'on lit un fichier côté client qu'on doit écrire dans un fichier côté serveur, et ce n'est parce qu'on lit sur l'entrée standard côté client qu'on doit écrire sur la sortie standard côté serveur.

En cas d'erreur dans le traitement des arguments en ligne de commande, vous afficherez une *usage string* : un message court indiquant comment se servir du programme. Optionnellement, vous afficherez également un message détaillant l'erreur survenue, le tout au format de votre choix.

1. Pour vous en convaincre, nous vous invitons à vous référer à la page de manuel Linux de Netcat : `http://linux.die.net/man/1/nc`

Par exemple :

```
C:\tp> mynetcat -m 1337
mynetcat: unknown option: -m
Usage: mynetcat [-l port|ip port] [file]

C:\tp> mynetcat 333.333.333.333 25154
mynetcat: invalid IP address: 333.333.333.333
Usage: mynetcat [-l port|ip port] [file]

C:\tp> mynetcat -l sdfsfdsdf
mynetcat: invalid port number: sdfsfdsdf
Usage: mynetcat [-l port|ip port] [file]
```

4.1.2 Palier 2 : serveur jetable

En mode serveur, écrivez une gestion de client simple : acceptez une connexion, affichez un message (format libre, toujours) dans la console, libérez les ressources puis quittez.

N'oubliez pas de gérer les erreurs ! Pour cela, référez-vous à la documentation pour obtenir la liste des exceptions lancées par les différentes méthodes de la classe `Socket`. On ne vous demande que des gérer les exceptions directement liées au réseau (ex. `SocketException`).

Par exemple :

```
C:\tp> mynetcat -l 80
mynetcat: could not bind to address 127.0.0.1:80: Address already in use
C:\tp> mynetcat -l 1337
New connection from 127.0.0.1:3029
C:\tp>
```

4.1.3 Palier 3 : client de ping

Écrivez maintenant une connexion simple au serveur. Là-encore, affichez simplement un message dans la console puis quittez. N'oubliez pas la gestion des erreurs.

```
C:\tp> mynetcat 127.0.0.1 7331
mynetcat: could not connect to 127.0.0.1:7331, is the server running?
C:\tp> mynetcat 127.0.0.1 1337
Connected to 127.0.0.1:1337
C:\tp>
```

4.1.4 Palier 4 : serveur recyclable

Modifiez le serveur pour qu'il ne s'arrête plus après la déconnexion d'un client. Autrement dit, on vous demande de gérer plusieurs clients (pas simultanément, mais les uns après les autres).

```
C:\tp> mynetcat -l 1337
Listening on port 1337...
New connection from 127.0.0.1:3029... Disconnected.
New connection from 127.0.0.1:3030... Disconnected.
New connection from 127.0.0.1:3031... Disconnected.
^C
C:\tp>
```

4.1.5 Palier 5 : échange de données

C'est ici que notre programme commence à ressembler à un vrai Netcat.

Modifiez le client pour envoyer le texte lu sur l'entrée standard, ligne par ligne, et le serveur pour l'afficher sur la sortie standard.

À ce stade, vous pouvez enlever les messages que nous affichions à la connexion ou à la déconnexion d'un client.

```
C:\tp> mynetcat 127.0.0.1 1337
Coucou
J'envoie ce texte
ligne
par
ligne
^Z
C:\tp> mynetcat 127.0.0.1 1337
Et quand y en a plus, y en a encore!
^Z
C:\tp>
```

```
C:\tp> mynetcat -l 1337
Coucou
J'envoie ce texte
ligne
par
ligne
Et quand y en a plus, y en a encore!
```

Note : Ctrl-Z envoie le caractère spécial `EOF` (End Of File), qui signifie ici qu'il n'y a plus rien à lire sur l'entrée standard.

4.1.6 Palier 6 : échange de fichiers

Modifiez :

- **le client** pour envoyer le contenu d'un fichier si l'argument en ligne de commande est présent (n'oubliez pas de gérer les erreurs : existence du fichier, etc.), auquel cas rien n'est lu sur la sortie standard, et le programme quitte automatiquement lorsqu'il a envoyé l'intégralité du fichier
- **le serveur** pour écrire les données reçues dans le fichier précisé en argument, auquel cas rien n'est affiché sur la sortie standard.
  Si votre protocole ne le permet pas déjà, faites en sorte que le serveur se ferme tout seul lorsque tout le fichier a été reçu.

```
C:\tp> type file.txt
What hath God wrought?
C:\tp> mynetcat 127.0.0.1 1337 file.txt
C:\tp>
```

```
C:\tp> mynetcat -l 1337 output.txt
C:\tp> type output.txt
What hath God wrought?
C:\tp>
```

4.2 Bonus : Crackme

Trouvez les deux flags (chaînes de caractères ayant une signification particulière) cachés dans `Crackme.exe` (disponible sur l'intra). Les flags sont indépendants, mais l'un est plus facile à trouver que l'autre.

Vous écrirez les flags que vous avez trouvé, un par ligne, dans le fichier `flags.txt`, situé dans le dossier `Crackme`, et vous fournirez le code que vous avez utilisé pour résoudre le problème dans `Crackme.cs` (pas de code = pas de points bonus).

Quelques (petits) indices : `abcerprqrapr` et `fvkglsbheovgvag`.

5 Conclusion

In ACDC you trust.