TP C#11 : MyTinyIrc

1 Consignes de rendu

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

```
rendu-tp-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- MyTinyIrc/
        |-- MyTinyIrc.sln
        |-- Client/
            |-- Tout sauf bin/ et obj/
        |-- Server/
            |-- Tout sauf bin/ et obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
* firstname.lastname$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* firstname.lastname" > AUTHORS
```

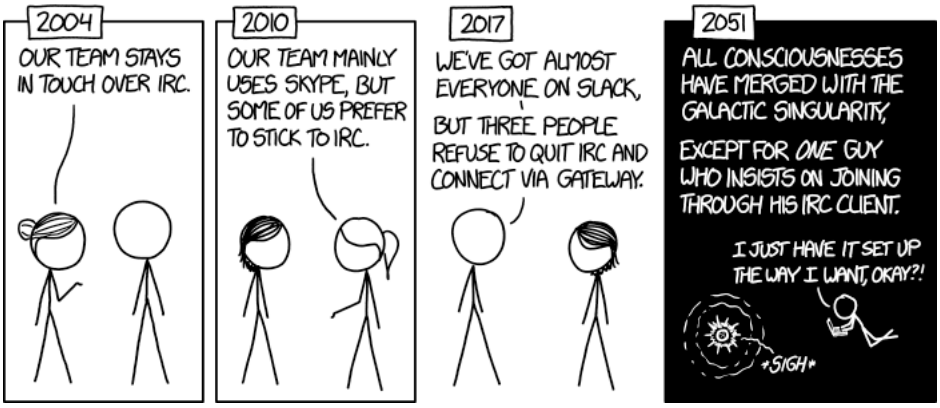
README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

2 Introduction

2.1 Objectifs

Beaucoup d'entre vous connaissent Messenger, Skype, Discord ou Slack. Mais les vrais connaissent l'existence du précurseur du chat sur Internet, le protocole IRC. IRC, ou Internet Relay Chat pour les intimes, a été conçu en 1988 par Jarkko 'WiZ' Oikarinen. Les messages sont transmis d'un point A vers un point B en étant relayé par un serveur IRC. EPITA a son propre réseau IRC 'irc.rezosup.org' sur lequel vous pouvez vous connecter à l'aide d'un client comme X-Chat ou WeChat.



3 Cours

3.1 It's dangerous to go alone, take this

Ce TP est réputé l'un des plus difficiles de l'année en raison des thèmes sensibles abordés. Nous avons tenté de le simplifier au maximum et de le rendre le plus clair possible. Néanmoins, pour le faire, nous vous encourageons à bien lire la documentation et à ne pas hésiter à utiliser Internet pour trouver la réponse à vos questions. Ce TP peut être rendu assez facile si vous apprenez à lire la documentation.

A lesson without pain is meaningless – Edward Elric, Full Metal Alchemist

3.2 Les chaussettes

Pour échanger des informations entre deux machines, nous utilisons le protocole IP. Le protocole IP est à la base même de l'Internet et vous permet de consulter des pages web, d'envoyer vos mails ou de jouer en réseau. Cependant, la gestion brute du protocole IP est compliquée pour les pauvres développeurs que nous sommes, par conséquent les interfaces sockets ont été créées pour simplifier notre travail.

Une socket est une interface de connexion qui permet la communication entre des processus en local sur une machine ou entre plusieurs machines reliées sur un réseau. La communication est possible dans les deux sens sur une même socket, on peut donc envoyer et recevoir des données en même temps.

Afin de vous servir des sockets Internet que nous utiliserons dans ce TP, il est nécessaire de vous expliquer le fonctionnement basique du protocole IP. Sur le réseau internet, chaque machine doit pouvoir être identifiée pour pouvoir communiquer avec les autres, un

peu à la manière des adresses postales. L'adresse IP identifie la machine en question tandis que le numéro de port identifie l'application qui utilise le socket. Histoire de faire une analogie, l'adresse IP est un peu comme l'adresse postale d'un immeuble tandis que le port permet d'identifier le numéro d'appartement d'un habitant. La représentation est souvent sous la forme `<ip_address>:<port_number>`, par exemple `127.0.0.1:4242` ou `192.168.0.1:1337`. L'adresse IP 127.0.0.1 est spéciale, il s'agit de l'adresse qui représente votre machine. Utilisez-la pour effectuer des tests en local.

Lorsqu'un développeur veut envoyer une information à une autre machine, il va créer une socket avec trois grandes caractéristiques.

L'adresse locale de la machine qui envoie l'information ainsi que le numéro de port de l'application qui utilise la socket.

L'adresse distante de la machine à qui il faut envoyer l'information ainsi que son numéro de port.

Le mode de transport qui influe sur l'intégrité et la vitesse des données transmises. Il existe deux principaux modes de transport de données.

— **TCP** (*Transmission Control Protocol*) qui privilégie l'intégrité des données en garantissant que les données seront bien livrées au destinataire, dans le bon ordre grâce à un système "d'accusés réception".

— **UDP** (*User Datagram Protocol*) qui privilégie la vitesse de transmission des données au détriment de leur intégrité. Ce mode est particulièrement utilisé dans le domaine du jeu vidéo où il est acceptable de perdre quelques données au profit d'une meilleure vitesse, ce qui est à l'origine des lags.

3.3 La classe `Socket`

Le framework `.NET` vous pré-mache le travail et vous met à disposition tout un panel de classes pour gérer le réseau. Mais celle qui nous intéresse particulièrement est la classe `System.Net.Sockets` [1].

Vous allez d'abord instancier la classe Socket avec le constructeur suivant. La déclaration d'une socket est nécessaire à la fois pour le client et pour le serveur. N'oubliez pas de fermer la socket une fois que vous avez fini de l'utiliser.

```
Socket (AddressFamily, SocketType, ProtocolType);
```

— **AddressFamily** qui représente l'adresse IPv4 de notre machine (il existe aussi l'IPv6, version 6, que nous n'utiliserons pas ici). Pour l'obtenir, utilisez `AddressFamily.InterNetwork`.
— **SocketType** qui est le type de socket utilisé. Nous utiliserons `SocketType.Stream`.
— **ProtocolType** qui est le type de protocole utilisé. Nous utiliserons `ProtocolType.Tcp`.

3.3.1 Socket du Client

Une fois votre socket déclarée, vous avez besoin de pouvoir vous connecter au serveur. Vous pouvez faire cela via la méthode `Connect` de `Socket`. Une fois notre socket connectée, il ne reste plus qu'à envoyer les données au serveur.

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

1. `https://msdn.microsoft.com/fr-fr/library/system.net.sockets.socket.aspx`

3.3.2 Socket du Serveur

Du côté du serveur, nous avons besoin de pouvoir accepter les connections d'un client. Pour cela, nous pouvons utiliser les méthodes `Bind`, `Listen` et `Accept`.

```
void Bind (IPEndPoint endPoint);
void Listen (int maxWaitList);
Socket Accept();
```

`Bind` va permet d'associer votre socket que vous avez déclarée à un point d'entrée sur votre machine, ou `endpoint`. Nous vous laissons le soin de consulter la documentation pour comprendre le fonctionnement de `Bind` et `IPEndPoint`.

`Listen` va permettre d'écouter l'`endpoint` de notre socket pour attendre des connections entrantes. L'argument de cette méthode est le nombre de connexions que l'on peut mettre en attente pendant que l'on traite l'une d'elles. Lorsqu'un client tentera de se connecter sur le serveur, il sera mis dans la file d'attente de Listen.

`Accept` va permettre d'accepter une connexion en suspens dans la file d'attente de `Listen` afin de la traiter. Elle retourne une socket directement connectée au client, la connexion est établie, il ne reste plus qu'à recevoir les données du client.

3.3.3 Envoyer et recevoir des données

Pour envoyer et recevoir des données à partir de nos sockets, une fois la connexion établie, nous pouvons utiliser les méthodes `Send` et `Receive` de votre `Socket`.

```
int Send(byte[] buffer);
int Receive(byte[] buffer);
```

`Send` prend en paramètre un tableau d'octets (ce qui se traduit par byte en anglais...) qui contient les données à envoyer et retourne le nombre d'octets qui ont pu être envoyé.

`Receive` prend en paramètre un tableau d'octets où les données reçues seront écrites. Il est donc indispensable que ce tableau soit de taille suffisante pour contenir les données à recevoir. Elle renvoie le nombre d'octets qui ont été reçus.

3.3.4 Les tableaux d'octets

Bien entendu, vous n'êtes pas habitués à l'utilisation du type `byte`. En effet, vous savez utiliser `char`, `int` et `long` mais pour pouvoir les envoyer avec `Send` ou lire les données reçues avec `Receive` il faut pouvoir les convertir ces `byte` dans les deux sens.

Pour les `string` vous pouvez utiliser la méthode `GetBytes`et `GetString` dans `System.Text`.

```
byte[] bt = System.Text.Encoding.UTF8.GetBytes("Shepard, un ACDC en platine");
string str = System.Text.Encoding.UTF8.GetString(bt);
```

Pour les autres type, vous pouvez utiliser `BitConverter`.

```
byte[] btInt = BitConverter.GetBytes(1337);
byte[] btFloat = BitConverter.GetBytes(13.37);
byte[] btChar = BitConverter.GetBytes('a');
byte[] btBool = BitConverter.GetBytes(true);

int c = BitConverter.ToInt32(btInt, 0);
float a = BitConverter.ToSingle(btFloat, 0);
char f = BitConverter.ToChar(btChar, 0);
bool e = BitConverter.ToBoolean(btBool, 0);
```

3.3.5 Matroska

Dans MonoDevelop, nous pouvons créer plusieurs projets par solution. Lorsque vous créez une nouvelle solution, MonoDevelop créé une solution contenant un projet. Pour ajouter un nouveau projet à votre solution, faites un clic droit sur votre solution dans l'arborescence de la solution, `Add`, `Add New Project`.

Cependant, vu qu'il y a deux projets dans une même solution, MonoDevelop compilera celui qui est actif, représenté en gras dans l'arborescence de la solution. Pour changer de projet actif, faites un clic droit sur le projet à définir en actif, puis `Set As Startup Project`.

Bien entendu, votre rendu devra contenir une solution `MyTinyIrc` avec les deux projets, `Client` et `Server`.

3.3.6 Keep Calm and RTFM

Nous insistons particulièrement sur le fait que vous *devez* lire la documentation avant de poser vos questions. N'hésitez pas à utilisez les `try` et `catch` pour vos opérations réseau.

> *Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.* – Thomas A. Edisson

4 Exercices

4.1 MyTinyIrc

Nous allons maintenant écrire notre propre protocole IRC permettant de réaliser un client capable d'envoyer des messages ou un fichier à un serveur. Regardez bien les exemples pour vous faire une idée.

I promise I will tell you when it's time to panic – John Reese, Person of Interest

```
> ./Client.exe 127.0.0.1 4242 data.txt
Connected to 127.0.0.1:4242
File sent successfully !
> ./Client.exe 127.0.0.1 4242
Connected to 127.0.0.1:4242
>> I am mad scientist !
>> It's so cool !
```

```
> ./Server.exe 1337 received.txt
Incoming connection from 127.0.0.1:1337
File received successfully !
> ./Server.exe 4242
Incoming connection from 127.0.0.1:4242
From Client : I am mad scientist !
From Client : It's so cool !
```

4.1.1 Palier 0 : Arguments

Dans ce palier, vous devez inclure deux arguments à votre programme pour le client et le serveur. Vous pouvez *customiser* vos propres messages d'erreurs et de log tant que vous restez dans les bornes du sujet.

Le client doit pouvoir accepter deux arguments, à savoir l'adresse IP et le numéro de port de la machine à laquelle on veut se connecter et éventuellement un troisième argument représentant le nom d'un fichier à transmettre. Si jamais les arguments ne sont pas valides ou que la connexion échoue, vous devez afficher un message d'erreur. Cela implique que vous devez vérifier la validité de l'adresse IP, du port et du fichier passés en argument.

```
> ./Client.exe 127.3.0.1 4242
/*Envoie les messages écrits par l'utilisateur au serveur*/
> ./Client.exe 37.187.3.40 4242 data_send.txt
/*Envoie le contenu du fichier data_send.txt au serveur*/
> ./Client.exe
/*Mauvaise utilisation du binaire*/
Usage : Client.exe ip_address port_number [file]
```

Le serveur doit pouvoir accepter un premier argument qui est le numéro de port à écouter, et éventuellement un second argument représentant le fichier dans lequel le serveur écrira les données reçues. Même chose qu'au dessus, vous devez vérifier la validité des arguments et afficher un message d'erreur en cas de problème.

```
> ./Server.exe 4242
/*En attente de données à recevoir*/
> ./Server.exe 4242 data_received.txt
/*En attente de données à recevoir*/
> ./Server.exe
/*Mauvais usage du binaire*/
Usage : Server.exe port_number [file]
```

4.1.2 Palier 1 : Server.exe

Dans ce palier, vous devez coder un début de serveur qui sera capable d'instancier une socket, de *binder* dessus les adresses et le port à écouter ainsi que d'attendre une connexion entrante et la gérer. N'oubliez pas d'afficher un message d'erreur en cas de problème [2] et un autre message dès qu'un client réussi à se connecter.

```
> ./Server.exe 80
Could not bind 127.0.0.1:80 : Address already in use
> ./Server.exe 4242
Incoming connection from 127.0.0.1
```

Nous vous conseillons d'implémenter les méthodes suivantes :

```
Server (int port, string filename = null);
/*Initialise les attributs et créer la socket*/
Run ();
/*Bind le socket et attends une connexion*/
```

4.1.3 Palier 2 : Client.exe

Nous allons maintenant coder un client qui va juste établir une connexion avec le serveur et afficher un message en cas de réussite ou d'erreur.

```
> ./Client.exe 127.0.0.1 666
Could not connect to 127.0.0.1:666 : Is the server running ?
> ./Client.exe 127.0.0.1 4242
Connected to 127.0.0.1:4242
```

Nous vous conseillons d'implémenter les méthodes suivantes :

```
Client (IPAddress address, int port, string filename = null);
/*Initialise les attributs et créer la socket*/
Run ();
/*Connecte la socket au serveur*/
```

4.1.4 Palier 3 : Échange

Dans ce palier, vous devez être capable d'envoyer et recevoir des données. Lorsqu'aucun fichier n'est spécifié en argument du client, celui-ci demande le message à envoyer au serveur puis l'envoie lorsque l'utilisateur appuie sur `Enter`. Côté serveur, lorsqu'aucun fichier n'est spécifié en argument le serveur affiche les données reçues sur la sortie standard. Dans le cas contraire vous écrivez les données reçues dans le fichier spécifié (le créer si il n'existe pas).

2. Protip : `try { ... } catch { ... }`

```
> ./Server.exe 1337
Incoming connection from 127.0.0.1:1337
From Client : Chaussette
```

```
> ./Client.exe 127.0.0.1 1337
Connected to 127.0.0.1:1337
>> Chaussette
```

4.1.5 Threshold 4 : File

Dans ce palier, vous allez devoir envoyer au serveur un fichier passé en paramètre du client. Le serveur doit être capable de l'afficher sur la sortie standard ou de l'écrire dans un fichier selon les paramètres qui lui ont été passés.

```
> ./Server.exe 1337 output.txt
Incoming connection from 127.0.0.1
File received successfully !
> cat output.txt
El Psy Congroo
```

```
> cat file.txt
El Psy Congroo
> ./Client.exe 127.0.0.1 1337 file.txt
Connected to 127.0.0.1:1337
File sent successfully !
```

4.1.6 Threshold 5 : Protocole Fantôme

Dans ce palier, vous devrez pouvoir transmettre au serveur un horodatage des messages ainsi que le pseudo de l'émetteur. Nous vous laissons la liberté d'inclure une option pseudo (avec `-p` en argument de votre client) afin de préciser le pseudo du client. Si ce pseudo n'est pas précisé, vous utiliserez un pseudo par défaut `root`.

Bien que nous vous laissions la liberté d'implémenter cette partie à votre guise, nous vous conseillons de mettre en place un mini-protocole entre votre serveur et client afin de pouvoir envoyer au serveur le pseudo et l'horodatage de vos messages.

4.1.7 Threshold 6 : Bonus

Vous pouvez faire des bonus qui seront comptés dans la notation. N'hésitez à demander conseil à vos ACDC pour des idées de bonus ! Histoire de vous donner quelques idées, voici une liste non-exhaustive...

— Des options comme un vrai IRC pour changer le pseudo et les couleurs.
— Un serveur capable de "recycler" les connections lorsqu'un client se déconnecte
— Un serveur capable de se connecter à plusieurs clients en même temps.
— Un chat en P2P