TP C#9 : Connect4

Consignes de rendu

A la fin de ce TP, votre rendu devra respecter l'architecture suivante :

```
|-- csharp-tp9-{login}/
    |-- AUTHORS
    |-- README
    |-- Connect4
        |-- Connect4.sln
        |-- Program.cs
        |-- Network.cs
        |-- Connect4.cs
        |-- 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`.
- 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) :

```
*␣prenom.nom$
```

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 "* prenom.nom" > 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).

1 Introduction

Le but de ce tp est de vous donner un aperçu de la façon dont le réseau fonctionne au sein d'un programme. Pour cela nous allons implémenter le jeu du puissance 4 (Connect 4 en Anglais) et faire en sorte de pouvoir jouer en ligne, depuis 2 PC distants. À la fin de ce tp, vous devez avoir un jeu fonctionnel avec lequel vous pouvez lancer un partie contre n'importe qui et depuis n'importe où !

2 Cours

2.1 Les Sockets

Le réseau est géré par ce qu'on appelle les '**sockets**'. Un socket est simplement une interface de communication entre processus. Les sockets vont vous permettre d'envoyer et de recevoir des informations sous forme de paquets. Nous vous invitons très fortement à lire la documentation MSDN des classes **Socket**, **IPAddress** et **IpEndPoint**.

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

Dans un réseau, une connexion se fait généralement entre un client et un serveur. Le client connaît l'adresse ip du serveur. Il lui suffit d'envoyer une requête au serveur pour l'informer qu'il veut se connecter.

```
1 //Côté client
2
3 /**
4 ** Essaye de se connecter au port 11000 du serveur
5 ** qui a pour adresse "42.420.69.3"
6 **/
7 socket.Connect("42.420.69.3", 11000);
```

De son côté, le serveur doit attendre qu'un client se connecte pour établir la connexion.

```
1 //Côté serveur
2 IPEndPoint remoteEP = new IPEndPoint("42.420.69.3", 11000);
3
4 socket.Bind(remoteEP);
5 socket.Listen(1);
6 Socket client = s.Accept();
```

Le serveur relie le **Socket** à ce qu'on appelle un **endpoint** grâce à la méthode **Bind**. Après avoir lié correctement, il faut **Listen**, ce qui va permettre d'écouter l'**endpoint** et de rajouter dans une file les connexions entrantes. Enfin, il faut **Accept**. Cette méthode est bloquante, c'est à dire qu'elle va arrêter le fil d'exécution tant que la file de connexion entrante est vide. Elle va ensuite pop une connexion, et renvoyer le **Socket** qui permet la communication avec cette connexion.

2.2 Send and Receive

Une fois la connexion établie, les méthodes **Send** et **Receive** vous servent à envoyer ou recevoir des informations. Elles prennent en argument des tableaux de bytes. Les méthodes de conversion peuvent vous servir.

```
1 //Envoyer des données
2 byte[] buf = System.Text.Encoding.UTF8.GetBytes("What door?");
3 socket.Send(buf);
```

```
1 //Recevoir des données
2 byte[] buf = new byte[256];
3 socket.Receive(buf);
4 string receive = System.Text.Encoding.UTF8.GetString(buf);
```

Note : Pour vérifier que la connexion est toujours établie, vous pouvez utiliser l'attribut **Connected** de la classe **Socket**.

3 Exercice : Connect4

3.1 Network

3.1.1 My IP

```
1  public static IPAddress My_ip();
```

La fonction **My_ip** doit renvoyer l'adresse ip (ipV4) de l'utilisateur. Pour la récupérer facilement, nous vous conseillons très fortement de regarder la documentation de la classe Dns et des méthodes **GetHostName** et **GetHostEntry** associées.

3.1.2 Connect

```
1  public static Socket Connect(string ip);
```

La fonction **Connect** doit créer un **Socket**, le connecter à "ip" sur le port 11000 et le renvoyer. Si une erreur intervient, la fonction doit renvoyer **null**. Vous devez gérer ce cas en utilisant un **try-catch**.

3.1.3 Wait Connection

```
1  public static Socket Wait_connection();
```

Puisque nous n'avons pas de serveur à vous proposer. Votre programme final doit pouvoir "simuler" un serveur en acceptant la connexion d'un client. La fonction **Wait_connection** doit simplement attendre une connexion sur le port 11000 et renvoyer le socket correspondant retourné par la méthode **Accept**.

3.1.4 Send Action

```
1  public static bool Send_action(Socket s, byte action);
```

Pour un puissance 4, les informations envoyées d'un joueur à l'autre sont très limitées. En réalité, nous n'avons besoin d'envoyer que la position à laquelle le joueur joue. Pour mettre tout le monde d'accord, et qu'il n'y ait pas de conflit entre vos différents programmes, les données envoyées correspondront toujours à la colonne sur laquelle le joueur veut (et peut) jouer numérotée de 0 à 6 et de gauche à droite. **Send_action** doit envoyer un byte (et seulement 1 !) via le Socket s. Si le socket est déconnecté, la fonction ne doit pas envoyer le byte et renvoyer **false**. Sinon la fonction renvoie **true**.

3.1.5 Receive Action

```
1  public static byte Receive_action(Socket s);
```

Receive_action doit récupérer un byte (et toujours 1) du Socket **s** puis le retourner.

3.2 Connect4

Maintenant que nous avons quelques fonctions basique pour gérer notre réseau, nous allons nous occuper du jeu en lui même. Nous reviendrons sur le réseau à la fin du tp. Tout d'abord un petit rappel sur les règles du puissance 4 :

Le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille comptant 6 rangées et 7 colonnes. Chaque joueur dispose de 21 pions d'une couleur (par convention, en général jaune ou rouge). Tour à tour les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans la dite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle. <fr.wikipedia.org/wiki/Puissance_4>

3.2.1 Constructeur

```
1  public Connect4();
```

Le constructeur de la classe **Connect4** doit simplement initialiser une grille de puissance 4 en remplissant **grid** de **Empty**.

3.2.2 Display

```
1  public void Display();
```

La méthode **Display** doit afficher la grille de puissance 4 dans la console. Vous êtes libre sur l'affichage, mais cela doit rester lisible et compréhensible !

3.2.3 Play

```
1  public bool Play(state player, int column);
```

La méthode **Play** doit simuler le coup d'un joueur. Pour cela elle doit rajouter un pion de la couleur de **player** dans la première case vide de la colonne **column** (toujours numérotée de 0 à 6 et de gauche à droite). La fonction retourne **true** si le coup est possible et modifie la grille de jeu. Si le coup n'est pas possible (colonne pleine ou paramètre(s) incorrecte(s)), **Play** doit retourner **false**.

La gestion des erreurs est primordiale en réseau. Tout ce qui est susceptible de mal tourner tournera nécessairement mal. C'est pourquoi il ne faut jamais faire confiance aux données que l'autre joueur vous envoie et systématiquement vérifier qu'il ne vous demande pas quelque chose d'impossible, ou même simplement de tricher.

3.2.4 Check Win

```
1  public bool Check_Win(state player);
```

Cette fonction doit vérifier si **player** a gagné. Elle renvoie **true** si elle trouve au moins 4 pions alignés de la couleur du **player**. Les pions peuvent être alignés verticalement, horizontalement ou en diagonale.

3.3 Program.cs

Maintenant que toutes vos fonctions sont terminées, vous allez pouvoir les assemblées pour créer le jeu de vos rêves.

3.3.1 Main

Votre **Main** va vous permettre d'initialiser votre partie. Avant toute chose, vous devez afficher votre adresse IP dans la console de cette façon :

```
Votre adresse ip : <my ip>
```

Ensuite, votre programme doit vous demander si vous voulez héberger la partie ou rejoindre une partie.

```
Voulez vous héberger la partie ? (O/N) :
```

Si le joueur choisit d'héberger la partie, votre programme jouera alors le rôle du serveur. Vous devez donc attendre que quelqu'un se connecte à l'aide de votre fonction **Wait_connection**.

Si le joueur choisit de rejoindre une partie, votre programme doit alors demander à l'utilisateur de rentrer l'adresse ip à laquelle il veut se connecter, puis se connecter à celle-ci. Si une erreur intervient, votre programme doit redemander une nouvelle adresse jusqu'à ce qu'elle soit valide.

```
Entrez l'adresse de votre partie :
```

Une fois le socket créé (à l'aide de **Wait_connection** ou **Connect**) votre **Main** doit appeler la fonction **game**. Encore une fois, pour mettre tout le monde d'accord et éviter les problèmes, le joueur qui héberge la partie sera le joueur "Bleu" et sera le premier à jouer.

3.4 Game

```
1  static void game(Connect4.state player, Socket s);
```

Une fois cette fonction terminée, votre jeu devrait être fonctionnel. game doit avoir une boucle principale qui continue tant que ces trois conditions sont respectées :

— Le socket est toujours connecté.
— Aucun des 2 joueurs n'a gagné.
— La grille n'est pas remplie.

Pour vous aider à implémenter le jeu, voici les étapes à suivre dans la boucle principale sous forme d'un pseudo-code :

```
Afficher la grille
Si c'est à votre tour de jouer :
|
| Tant que la colonne est invalide :
| | Demander à l'utilisateur de choisir sa colonne
| Jouer dans cette colonne
| Envoyer le numéro de la colonne au deuxième joueur
| Passer au tour suivant
|
Sinon
|
| Attendre que le deuxième joueur joue et récupérer ce coup
| Si son coup est incorrect :
| | Quitter la partie immédiatement et afficher un message d'erreur
| Appliquer son coup à la grille
| Passer au tour suivant
|
Fin si
```

3.5 Bonus

3.5.1 Tableau des scores

Pour réaliser ce bonus, vous pouvez sauvegarder dans un fichier toutes les ip contre lesquelles vous avez joué, puis de leur associer un ratio de victoires/défaites afin d'établir un classement des meilleurs joueurs de puissance 4 de votre promos.

3.5.2 IA

En bonus, vous pouvez implémenter une IA qui jouera à votre place (sans même que le joueur adverse ne le sache!). Pour cela, vous pouvez jeter un coup d'œil à l'algo "min-max" qui vous permettra d'avoir une IA très forte sans trop de problème. En résumé, le principe de l'algo est assez simple : Il vous suffit de créer l'arbre de tous les coups possibles jusqu'à une certaine profondeur. Puis, en remontant, de prendre successivement le meilleur coup que vous pouvez faire et le meilleur coup que l'adversaire peut faire (donc le pire coup qu'il pourrait vous arriver). Une fois arrivé à la racine, vous avez donc un chemin des coups optimaux à jouer sur les X prochains coups.

3.5.3 Autre

C'est votre dernier tp de C#, impressionnez nous! Si vous avez d'autres idées de bonus, réalisez les, tant que ceux-ci ne perturbe pas votre jeu ni le réseau. Pensez simplement à en parler dans votre README.

If you knew time as well as I do, you wouldn't talk about wasting it