TP C#13 : Network

Consignes de rendu

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

```
rendu-tp13-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- Rednit/
        |-- Rednit.sln
        |-- Rednit_Lite/
            |-- Everything except bin/ and obj/
        |-- Rednit_Server/
            |-- Everything except bin/ and obj/
    |-- Chat/
        |-- Chat.sln
        |-- Client/
            |-- Tout sauf bin/ et obj/
        |-- Server/
            |-- Tout sauf bin/ et obj/
    |-- Bonus/
        |-- Tout bonus complementaire fait
```

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) :

```
*␣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

Ce TP a pour but de vous faire comprendre ce qu'est le réseau. Vous devez au moins être capable de visualiser comment fonctionne une architecture réseau, ça vous aidera quand vous étudierez le sujet en profondeur par la suite. Un autre but du TP est que vous puissiez vous amuser. N'hésitez pas à ajouter des fonctionnalités (tant que vous en discutez au préalable avec vos ACDC), ça devrait être accessible à tout le monde.

La première partie du cours est très théorique. Elle explique les bases du réseau, vous devez donc vous assurer d'avoir bien compris cette partie avant de passer à la suivante.

La deuxième partie du cours décrit les outils C# qui pourront vous être utiles pendant le TP. Si vous ne comprenez ce qu'il faut faire dans un des exercices, référez-vous d'abord au cours théorique puis à cette partie pour trouver la ou les bonnes fonctions à utiliser.

Le premier exercice est très simple, c'est une application directe du cours. Si vous avez bien compris, ça ne vous posera aucun problème.

Le deuxième exercice est encore plus simple en termes de code. Cependant vous aurez besoin de comprendre le fonctionnement du programme. Le but de cet exercice est de vous montrer une architecture d'application en réseau différente et plus complète que l'exercice 1. D'autre part c'est un support pour aller plus loin, pour ceux qui veulent en savoir plus sur le sujet. En effet, un complément de TP, qui ne sera pas évalué, est à votre disposition sur l'intranet. Il vous permettra de comprendre et ainsi pouvoir modifier l'application a votre guise.

2 Cours

2.1 C'est quoi l'internet ?

Commençons par vous expliquer comment fonctionne les communications entre les machines. Il faut comprendre qu'il existe de très nombreuses manières de faire ça. Pour notre part, nous allons nous concentrer sur la suite de protocoles TCP/IP qui est à la base d'Internet.

Un protocole est une règle, une convention qui permet aux utilisateurs de communiquer sans avoir de problèmes de compatibilité. Il y a des protocoles à tous les niveaux d'une communication réseau : le support physique, l'architecture du réseau, la manière de communiquer, le format des messages, etc... Si vous voulez plus d'informations sur tous les protocoles réseau, Google est votre ami.

Vous êtes déjà perdu ? Ce n'est pas grave, on va recommencer avec les bases.

2.2 Il faut qu'on parle.

Un réseau c'est un certain nombre de machines (2 ou plus) qui communiquent à travers des protocoles communs.

Une machine peut être un ordinateur, un portable, une machine virtuelle ou plus globalement tout équipement doté d'une carte réseau.

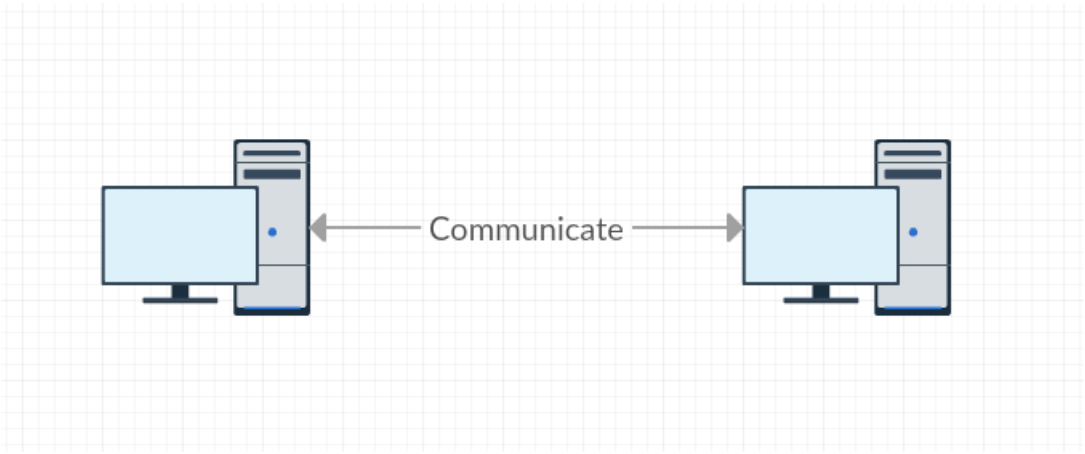

FIGURE 1 – Communication

2.3 Les socquettes

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.

Plus simplement, on peut visualiser une socket comme un fichier spécial. Pour récupérer le message que l'autre machine a envoyé, il suffit de lire celui-ci. Pour envoyer un message, il faut lui fournir la donnée à envoyer et la socket va s'occuper de toutes les procédures d'envoi elle même.

Une socket est donc une sorte d'outil magique qui va s'occuper de récupérer et d'envoyer les messages pour vous. Il faut cependant bien la configurer pour qu'elle se comporte de la manière que l'on souhaite. Nous verrons plus tard dans ce cours comment utiliser la classe Socket en C#.

2.4 Les adresses / ports / bateaux

Maintenant que vous voyez à peu près comment deux machines communiquent, vous devez vous dire que c'est extrêmement simple. En réalité ça l'est, cependant pas à ce point là, il y a d'autres facteurs qui vont complexifier l'ensemble.

L'exemple au-dessus schématise une connexion entre deux utilisateurs uniquement, ce qui rend le schéma incroyablement simple. Que se passe-t-il si l'on veut connecter 3 utilisateurs ? Et pour mille ? Et 8 milliards ? Comment savoir à qui nous sommes en train de parler ?

Regardez le schéma ci-dessous pour visualiser le problème. Considérez le nuage internet comme un conglomérat de routeurs, serveurs, machines d'utilisateurs, etc..., le tout interconnecté. Et considérez les quatre branches comme des morceaux de ce nuage qu'on a isolés.

Un routeur, pour faire simple c'est une boite qui permet à un certain nombre de machines et/ou routeurs de se connecter. Par exemple cinq machines se connectent à un même routeur, elles auront le possibilité de communiquer entre elles.

Donc, dans l'exemple ci-dessous comment reconnaître la machine visée si l'on veut contacter un utilisateur qui est sur une autre branche ? Comment est-ce que le routeur sait sur quel câble envoyer la donnée ? Il ne peut pas, donc il faut un système supplémentaire pour pouvoir faire ça.

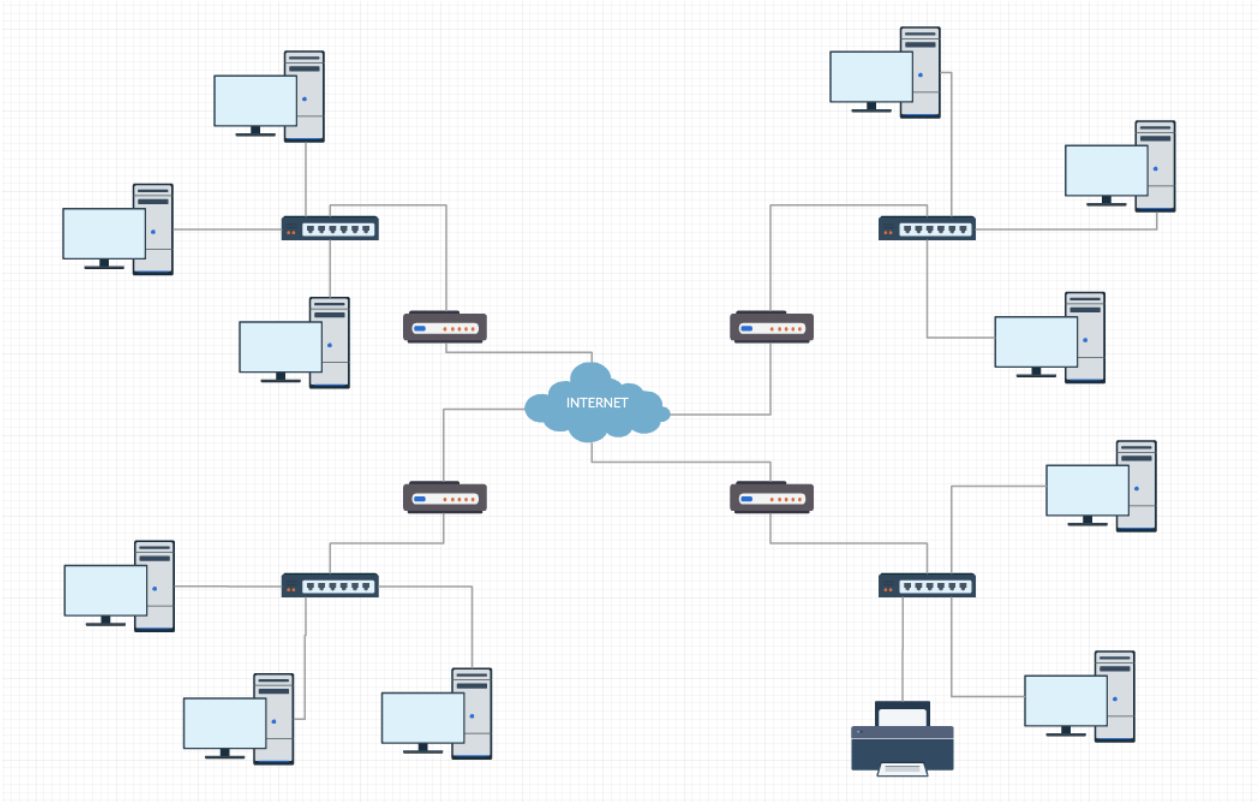

FIGURE 2 – Internet

Pour régler ce problème, un système d'adresses à été ajouté. Chaque machine se voit attribuer une adresse. Ainsi, pour communiquer, il suffit d'indiquer l'adresse de la machine visée. Essayez de taper dans votre navigateur favori (ouvrir Internet Explorer est passible de confloose) les 3 adresses présentes dans le schéma, vous comprendrez peut-être un peu mieux.

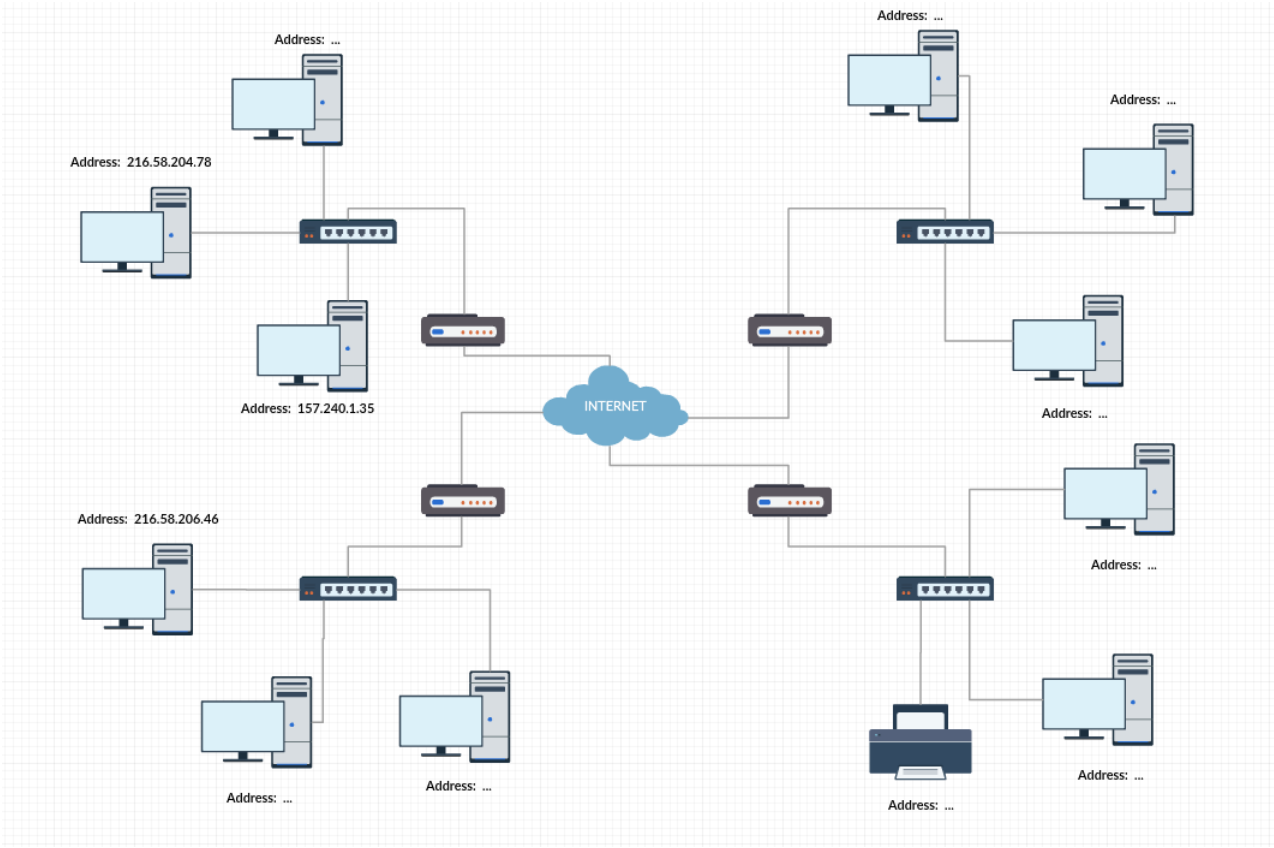

FIGURE 3 – Adresses

Un deuxième problème qui apparaît est la limite du nombre de connexions par machine. Si notre machine veut communiquer avec deux autres machines en simultané, il lui faudra (de préférence) 2 sockets. Cependant, comment faire pour les distinguer ? Comment savoir quelle socket permet de communiquer avec quelle machine ?

Prenons un exemple concret : Vous êtes sur votre ordinateur et vous voulez ouvrir l'intranet, mais aussi discord pour **discuter** du TP et un jeu en ligne. Donc vous avez trois applications qui vont chacune ouvrir au moins une socket. Comment est-ce qu'un serveur sait pour quel usage a été ouvert une socket ?

Un système de port a donc été mis en place. Un port est un entier non signé sur 16 bits. Les ports de 0 à 1024 sont réservés. Ainsi, une socket doit se lier à un port pour pouvoir communiquer avec l'extérieur.

Les notions d'adresses et de port paraissent simples expliquées comme ça, mais techniquement, leurs implémentations sont loin de l'être et il existe de nombreuses subtilités. On ne s'intéressera qu'aux bases dans ce cours, si vous voulez aller plus loin, comme toujours Google est votre ami.

Les pages wikipedia suivantes vous seront utile, lisez les. Elles ne sont pas trop longues et vous pouvez passer sur les versions françaises.

```
https://en.wikipedia.org/wiki/Internet
https://en.wikipedia.org/wiki/Internet_Protocol
https://en.wikipedia.org/wiki/Port_(computer_networking)
https://en.wikipedia.org/wiki/IP_address
```

Si vous voulez en apprendre un peu plus, regardez celles-ci. Elles traitent soit de sujets vu dans le TP, soit de technologies que vous utilisez dans la vie de tout les jours.

```
https://en.wikipedia.org/wiki/Transmission_Control_Protocol
https://en.wikipedia.org/wiki/User_Datagram_Protocol
https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
https://en.wikipedia.org/wiki/HTTPS
https://en.wikipedia.org/wiki/Secure_Shell
```

2.5 Un peu de pratique

Il est temps de voir en pratique comment faire communiquer deux machines. On va différencier deux rôles, les serveurs et les clients. Une machine peut jouer (et en pratique joue) les deux rôles en même temps. Lorsque vous vous connectez à une page internet, vous êtes un client qui contacte un serveur. Si vous lancez un site web sur votre machine et que des personnes extérieures peuvent s'y connecter, alors vous êtes aussi un serveur.

En général, les serveurs ne sont pas lancés sur des machines de particuliers. Ils sont lancés sur des machines hébergées dans des grandes salles serveur que le créateur peut administrer à distance sur son ordinateur personnel.

2.5.1 Serveur

Donc, un serveur, qu'est-ce que ça fait exactement ? A quoi ça sert ?
Son rôle est d'accepter des demandes de connections et des requêtes de clients puis de faire ce qui lui est demandé. Par exemple, on veut créer une application qui permet à un certain nombre de personnes de se connecter. Puis, on veut que chaque utilisateur puisse envoyer des messages et qu'ils soient reçus par tous les autres utilisateurs. Les rôles du serveur seront donc :

- Accepter les connexions des utilisateurs.
- Recevoir tous les messages envoyés par les utilisateurs.
- Envoyer les messages reçus à tous les autres utilisateurs (tout le monde sauf l'expéditeur)

Il y a 4 étapes minimum pour configurer la socket en mode serveur :

- Créer une socket et la configurer selon ses besoins.
- Lier la socket à un port (Bind). La socket écoutera désormais sur le port lié.
- Écouter le port (Listen). Créer une file d'attente de connexions et chaque client qui tente de se connecter est mis dans cette file.
- Accepter des connections (Accept). Accepter une connexion dans la file d'attente de connexions.

Avec ça, vous avez des clients qui peuvent se connecter et communiquer avec votre serveur.

2.5.2 Client

Maintenant, au tour du client, à quoi ça sert ?
Le rôle du client est de permettre à l'utilisateur d'utiliser l'application et les fonctionnalités du serveur. Un client pour le serveur en exemple au dessus aurait au moins les fonctionnalités suivantes :

— Se connecter à un serveur.
— Écrire et envoyer un message au serveur.
— Afficher les messages envoyés par le serveur.

Il pourrait avoir d'autres fonctionnalités, mais juste celles-là suffisent à avoir une application utilisable par n'importe qui.
Il y a 2 étapes minimum pour configurer la socket en mode client :

— Créer une socket et la configurer selon ses besoins.
— Se connecter à un serveur (Connect). La socket devra avoir une adresse et un port valide.

Avec ça, votre client peut se connecter et communiquer avec un serveur.

Il est important de bien visualiser la différence pour la suite du TP, vu que c'est un des exercices que vous aurez à faire.

2.6 Apprendre à aimer le C#

Dans cette partie de cours, vous allez avoir quelques explications sur les outils dont vous aurez besoin pendant le TP.

2.6.1 Socket

La classe **Socket** en C# vous donne accès à une socket et toutes les fonctions pour l'utiliser.

Constructeur

Vous devrez utiliser les constructeur suivant :

```
1  Socket(AddressFamily, SocketType, ProtocoleType);
```

Les trois paramètres vous seront indiqué pendant les exercices.

Send

La fonction **Send** permet d'envoyer un tableau d'octets à la socket associée (celle du destinataire).

```
1  public int Send(byte[] buffer, int size, SocketFlags socketFlags);
```

Les paramètres de la fonction :

— **buffer** est le tableau d'octets à envoyer.
— **size** est la taille de la donnée à envoyer, par exemple si on veut envoyer uniquement 16 octets et que le buffer fait 1024, il faut mettre une size de 16.
— **socketFlags**, il faut le mettre à None (SocketFlags.None).

La valeur de retour est le nombre d'octets envoyés.

Receive

La fonction **Receive** permet de recevoir un tableau d'octets du destinateur.

```
1  public int Receive(byte[] buffer, SocketFlags socketFlags);
```

Les paramètres de la fonction :

- **buffer** est le tableau d'octets où mettre la donnée reçue.
- **socketFlags** il faut le mettre à None (SocketFlags.None).

La valeur de retour est le nombre d'octets reçus.

Connect

La fonction **Connect** est utilisée uniquement par le client. Elle permet de demander au serveur d'accepter une nouvelle connexion. Une fois que la connexion est acceptée, il est possible d'utiliser les fonctions **Send** et **Receive** pour communiquer avec le serveur.

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

Les paramètres de la fonction :

- **address** l'adresse du serveur.
- **port** le port du serveur.

Bind

La fonction **Bind** est utilisée uniquement par le serveur. Elle permet de lier une socket à un port. Ainsi la socket 'écoutera' ce port pour accepter des connexions et communications. Une fois que la socket est liée, il est possible d'utiliser les fonctions **Listen** et **Accept** pour accepter des connections clients. Ces fonctions sont décrites plus bas.

```
1  public void Bind(EndPoint localEP);
```

Les paramètres de la fonction :

- **localEP** L'adresse et le port du serveur sous forme d'un objet IPEndPoint décrit plus bas.

Listen

La fonction **Listen** est utilisée uniquement par le serveur. Elle crée une file d'attente de connexions. Après avoir appelé cette fonction, la socket pourra recevoir des demandes de connexion de la part des clients. Une fois que la socket est en état "d'écoute", il est possible d'utiliser les fonctions **Accept** pour accepter les connections stockées dans la file d'attente.

```
1  public void Listen(int size);
```

Les paramètres de la fonction :

- **size** La taille maximale de la file d'attente, qui correspond au nombre maximum de demandes de connexions n'ayant pas encore été acceptées.

Accept

La fonction **Accept** est utilisée uniquement par le serveur. Elle accepte les demandes de connexion dans la file d'attente. Un appel à cette fonction doit être fait par client, sinon lui et le serveur ne pourront pas communiquer.

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

La valeur de retour de la fonction est une socket. Tout communication avec le client accepté se fera grâce à cette socket.

2.6.2 TcpClient

La classe **TcpClient** en C# est un wrapper autour de la classe **Socket**. Cela signifie qu'elle enrobe la classe Socket avec d'autres fonctions plutôt pratiques. Elle permet aussi de configurer une socket en mode Tcp automatiquement.

Constructeur

Le constructeur de **TcpClient** configure la socket pour une communication TCP. De plus, il connecte directement la socket au serveur. Il n'y a donc pas besoin d'appeler la fonction **Connect**.

```
1  TcpClient(IPEndPoint remoteEP);
2  TcpClient(string address, int port);
```

Les paramètres des fonctions :

- **première fonction :** Elle prend en paramètre les coordonnées du serveur sous forme d'un objet **IPEndPoint**.
- **seconde fonction :** Elle prend en paramètre l'adresse du serveur sous forme de **string** et le port sous forme d'integer.

GetStream

La fonction **GetStream** permet de récupérer et d'envoyer la donnée de manière différente que le Send et Receive. Elle retourne un **NetworkStream**. Il s'agit d'un stream spécial qui récupère et envoie la donnée, il est donc accessible en lecture et écriture.

Socket

Pour avoir accès à la socket qui est dans un **TcpClient**, il faut utiliser le champs **Client**.

2.6.3 IPEndPoint

C'est une classe pratique puisque'elle permet de stocker l'adresse et le port ensemble tout en les rendant accessibles. De plus elle permet de convertir automatiquement une adresse à partir de plusieurs formats.

Cette classe donne aussi accès à quelque fonctions supplémentaires. Elle fait aussi des vérifications de validité sur l'adresse et le port.

Les deux constructeurs sont les suivants et sont plutôt clair.

```
1  public IPEndPoint(long address, int port);
2  public IPEndPoint(IPAddress address, int port);
```

2.6.4 JsonConvert

Nous avions pensé vous faire utiliser le parseur de JSON que vous aviez codé pendant le TP Brainfuck compte tenu du haut taux de réussite à celui-ci ; Cependant par soucis d'égalité pour les rares n'ayant pas perfectionné celui-ci, nous avons décidé d'utiliser un parseur tout fait.

Il y a deux fonctions utiles pour parser du Json, une qui transforme un objet en chaîne de caractères (c'est la sérialisation) et une qui transforme une chaîne de caractères (format Json) en objet (c'est la dé-sérialisation).

```
1  object obj = JsonConvert.DeserializeObject(str);
2  string str = JsonConvert.SerializeObject(obj);
```

2.6.5 Thread

Souvent, quand on fait du réseau, on a besoin de réaliser plusieurs tâches en simultané. Par exemple, on veut pouvoir recevoir et envoyer des messages en même temps. Ou alors, accepter des connexions tout en traitant les demandes des clients déjà connectés.

Une solution (c'est bien UNE solution pas LA solution) est d'utiliser des **Threads**. Ça permet de faire tourner plusieurs tâches simultanément. Par exemple en exécutant les fonctions **Send** et **Receive** dans deux threads différents, elles tourneront en même temps et il sera possible de recevoir et d'envoyer des messages indépendamment.

2.6.6 Windows Forms

Les windows forms sont un outil extrêmement pratique en C#. Elle permettent de créer facilement une interface graphique pour son application (avec du glisser-déposer, un rendu visuel et une configuration guidée). Elles sont utilisable sur Linux. Cependant, si vous voulez en créer il faudra installer Visual Studio sur Windows.

Lisez le premier lien pour plus d'informations sur les windows forms et le deuxième lien pour avoir un tutoriel basique dessus :
`https://en.wikipedia.org/wiki/Windows_Forms`
`http://csharp.net-informations.com/gui/cs_forms.htm`

Conseil

Si vous avez besoins de plus de précisions sur ces différentes fonctionnalités, demandez à Google : "C# <fonctionnalité>".

3 Exercices

3.1 Exercice 1 - Chat

Dans cet exercice, il va falloir implémenter un système de Chat. Son principe est simple, il y a un serveur et n clients connectés dessus. Un message envoyé par un client doit être reçu par tous les autres clients.

Attention

Il n'est pas demandé de gérer les cas particuliers et il ne s'agit pas non plus de faire un Chat soit qui optimisé. Le but est que vous compreniez ce qu'il faut faire.

Cependant, si ça vous intéresse d'aller plus loin vous pouvez toujours modifier et améliorer le Chat. Les modifications ne doivent pas changer le comportement des fonctions demandées.

Important

Vous êtes obligé d'utiliser la classe **Socket** et ses fonctions pour cet exercice. Il est interdit d'utiliser une autre classe qui vous mènerait au même résultat.

3.1.1 Le client

Le client est très simple, il y a trois fonctions à implémenter dans la classe **Client**. Vous n'avez pas besoin de lancer ni d'attraper d'exceptions.

Constructeur

Vous devez créer une nouvelle socket, la mettre dans **_sock**, puis la connecter au serveur. Les paramètres de la socket doivent être **InterNetwork**, **Stream** et **Tcp**. Les coordonnées du serveur sont passées en argument.

```
1  public Client(IPAddress address, int port)
2  {
3      //FIXME
4  }
```

Reception

Vous devez recevoir un message du serveur. Sa taille ne peut pas dépasser 1024 caractères. Vous devez afficher ce que vous avez reçu dans la **Console**.

Conseil

Pour transformer un tableau d'octets en chaîne de caractères, il est conseillé d'utiliser cette fonction.

```
1  var str = Encoding.ASCII.GetString(byte_array);
```

Utiliser cette fonction va mettre un grand nombre de caractères non imprimables dans votre chaîne de caractères, je vous laisse comprendre pourquoi et le résoudre vous même. Petit indice, utilisez la valeur de retour de **Receive()**.

```
1  public void ReceiveData()
2  {
3      //FIXME
4  }
```

Envoi

Vous devez envoyer un message au serveur. Sa taille ne peut pas dépasser 1024 caractères. Vous devez lire le message à envoyer depuis la **Console**.

Conseil

Pour transformer une chaîne de caractères en tableau d'octets, il est conseillé d'utiliser cette fonction.

```
1  var byte_array = Encoding.ASCII.GetBytes(str);
```

Utilisez **ReadLine()** pour lire dans la console.

```
1  public void SendData()
2  {
3      //FIXME
4  }
```

3.1.2 Le serveur

Maintenant au tour du serveur, vous devez implémenter quatre fonctions dans la classe **Server**.

Constructeur

Vous devez créer la socket et initialiser la liste de clients **__clients**. La socket doit avoir les mêmes paramètres que le client.

```
1  public Server(int port)
2  {
3      //FIXME
4  }
```

Initialiser

Vous devez initialiser le serveur dans cette fonction. Pour le bind, utilisez **IPAdress.Any** (qui va gérer l'adresse automatiquement) et le port stocké dans **__port**. La file d'attente de connexions n'a pas besoin d'être trop longue, une taille de 10 suffit. Enfin, on veut être informé que le serveur est lancé, il faut donc écrire dans la console : "Server Started".

```
1  public void Init()
2  {
3      //FIXME
4  }
```

Accept

Vous devez accepter une connexion client dans cette fonction. Vous devez récupérer la socket du client, l'ajouter à la liste de clients et retourner la socket.

```
1  public Socket Accept()
2  {
3      //FIXME
4  }
```

Reception

Vous devez recevoir un message du client passé en paramètre. Sa taille ne peut pas dépasser 1024 caractères. Il faut ensuite l'envoyer à tous les autres clients, utilisez **SendMessage()**.

Conseil

Inspirez vous de la fonction **ReceiveData()** du client.

```
1  public void ReceiveMessages(Socket client)
2  {
3      //FIXME
4  }
```

Envoi

Vous devez envoyer un message à tous les clients sauf l'envoyeur (**sender**) et les clients déconnectés.

Conseil

Inspirez vous de la fonction **SendData()** du client.

```
1  public void SendMessage(string message, Socket sender)
2  {
3      //FIXME
4  }
```

Conclusion

Bravo vous avez maintenant un serveur de messagerie fonctionnel. Vous pouvez vous amuser avec pour aller plus loin. Si vous souhaitez modifier le comportement des fonctions, contactez vos ACDC respectifs et tentez de négocier.

Cependant, une application bien plus divertissante et approfondie vous attend dans l'exercice 2.

3.2 Exercice 2 - Rednit

Rednit est une application pour se faire des amis. Elle est inspirée du fonctionnement de **Tinder**.

3.2.1 Les fonctionnalités

L'application est très complète, elle possède donc de nombreuses fonctionnalités. Vous devrez vous assurer que toute celles-ci marchent à la fin du TP.

Créer un compte

Vous pouvez dans un premier temps créer un compte. C'est la première étape quand vous ouvrez l'application pour la première fois. Vous pouvez créer un nombre illimité de comptes.

Votre login doit être unique, s'il est déjà utilisé vous recevrez une erreur et il doit aussi avoir une taille inférieure à 50 caractères. Le mot de passe quant à lui n'a que la contrainte de taille inférieure à 50 caractères. Il peut être vide ou constitué de n'importe quel caractère.

> Attention
>
> Ne mettez pas de mot de passe que vous utilisez réellement, privilégiez des mots de passe du type "abc", "azerty", "123". Toutes les informations sont stockées en clair et sont donc accessibles à n'importe qui. La sécurité de l'application a été artificiellement réduite pour les besoins du TP.

Se connecter

Une fois que vous avez créé un compte, vous pouvez vous connecter. La création de compte est instantanée, donc vous pouvez vous connecter immédiatement après. Après avoir cliqué sur le bouton connecté, soit vous aurez une erreur parce que ce que vous avez entré est faux, soit vous serez redirigé vers la fonctionnalité de **match**.

Paramétrer son profil

Vous devez configurer votre profil juste après vous être connecté pour la première fois. Si ce n'est pas fait, vous ne pourrez pas utiliser la fonctionnalité de **match**. Dans l'onglet profil, il y a un certain nombre d'informations pouvant être modifié :

- **Prénom** : taille maximum 50 caractères, vous pouvez mettre ce que vous voulez dedans.
- **Nom** : taille maximum 50 caractères, vous pouvez mettre ce que vous voulez dedans.
- **Age** : nombre entre 0 et 666.
- **Description** : un texte d'une taille maximale de 1000 caractères.
- **Photo** : vous pouvez en sélectionner une nouvelle en cliquant sur la photo.
- **Centres d'intérêt** : il faut cocher les cases correspondant à vos centres d'intérêt. Si vous n'en cochez aucune, vous n'aurez pas accès à la fonctionnalité de **match**.

Après avoir configuré le profil, il faut cliquer sur **Save** pour sauvegarder sur le serveur.

Match / Like / Dislike

La fonction de **match** va vous permettre de trouver des amis potentiels en fonction des centres d'intérêt. C'est simple, lorsque vous voyer un profil d'un autre utilisateur, vous avez accès à deux boutons "j'aime" (Like) ou "je n'aime pas" (Dislike). Un *Like* signifie que vous souhaitez devenir son ami (et plus si affinités ?), un *Dislike* signifie le contraire. Si vous avez *Like* une personne et que cette personne vous a *Like* aussi, ça match et vous devenez amis. Vous pourrez discuter à travers le Chat.

La manière de trouver un utilisateur qui vous correspond est la suivante :
Sélectionner toutes les personnes qui vous ont *Like* mais à qui vous n'avez encore répondu (vous n'avez pas *Like* ou *Dislike* leur profil).
Parmi cette sélection, un profil aléatoire est gardé.
S'il n'y a personne dans la sélection, alors toutes les personnes ayant au moins un intérêt en commun avec vous sont sélectionnées.
Parmi cette sélection, un profil aléatoire est gardé.

Chat

Le chat permet de discuter avec ses amis (*Like* dans les deux sens). Il faut sélectionner un ami à qui parler puis il sera possible d'envoyer des messages. Le Chat ne va pas actualiser automatiquement les messages reçus, il faut donc cliquer sur le bouton envoyer pour les afficher. L'envoi d'un message vide permet d'actualiser les messages reçus sans envoyer de message.

Hack

Une fonctionnalité d'exploitation des failles a été ajoutée. Sécuriser son application est très important, il faut s'assurer de ne jamais laisser de failles. Ceux qui iront jusqu'au bout de ce TP comprendront à quel point cette application est pleine de failles et les possibilités que ça leur laisse pour la casser. A noter que les injections SQL ont tout de même été bloquées car trop violentes.

3.2.2 Les buts

Dans cet exercice vous aller devoir coder la partie réseau du client. Il faudra donc modifier le projet **Rednit_Lite**.

Ce qui est à faire

Il y a 12 fonctions à coder pour faire fonctionner le réseau du client. Cependant vous aurez besoin de comprendre comment fonctionne l'application dans son ensemble.

Compléments

Le serveur qui est fourni dans l'archive n'est pas utile pour cet exercice. Il sert pour les compléments de TP, pour aller plus loin.

Outils

Pour pouvoir tester le client que vous êtes en train de coder, un serveur de l'application à été mis en ligne. Tous les clients (vous) peuvent ainsi ce connecter à ce même serveur. Ça signifie que chacun d'entre vous peut voir et interagir avec les autres élèves de la promo.

Important

Le serveur étant public et ouvert à la totalité de la promotion, tout ajout de contenu pouvant être considéré comme indécent ou inapproprié sera sanctionné d'un zéro à la note du TP et possiblement d'autres sanctions disciplinaires en fonction de la gravité.

En cas d'abus ou de dégénérescence le serveur pourra être fermé. Vous devrez donc, dans ce cas configurer et lancer le serveur vous même afin de pouvoir exécuter le client.

3.2.3 La description de l'application

L'application est découpée en trois parties :
- Le client qui est mis à disposition de l'utilisateur pour qu'il puisse utiliser l'application.
- Le serveur qui répond aux requêtes des clients et permet des interactions entre les utilisateurs.
- La base de données qui stocke toutes les données relatives aux utilisateurs et à l'application.

Donc l'utilisateur communique avec le serveur à travers le client.
Le serveur répond aux requêtes des clients et envoie la donnée devant être stockée à la base de données.
La base de donnée stocke, modifie et envoie sa donnée en fonctions des demandes du serveur.

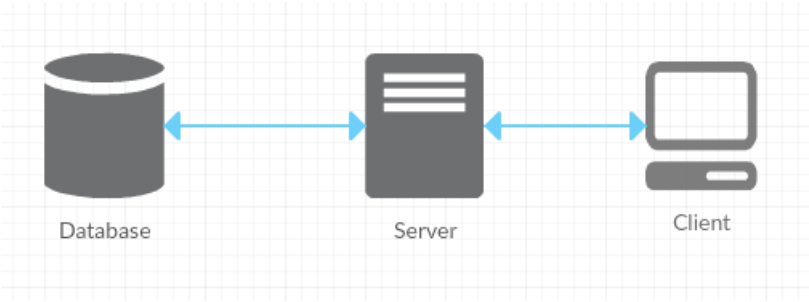

FIGURE 4 – L'architecture de l'application

Base de données

Un système de base de données permet de stocker de manière organisée des données. Ça permet de sauvegarder une quantité très importante d'informations sur une application.

Une base de données possède un certain nombre d'avantages qui la rendent pratique et facile à utiliser.

Une base de données est créée dans un serveur de base de données. Celui-ci est très facile à mettre en place et ne demande pas de connaissances en réseau. Ensuite une fois que le serveur est lancé, créer une base de données ne nécessite qu'une seule ligne de code.

L'utilisation d'une base de donnée est aussi très simple (tant qu'on n'y entre pas trop en profondeur). La première étape est de s'y connecter avec un login et un mot de passe. Ensuite, elle fonctionne avec un système de requêtes écrit dans un certain langage (très majoritairement du SQL). Avec ces requêtes, il est possible d'interroger, modifier et supprimer la base, selon les permissions de l'utilisateur connecté évidemment.

Un autre avantage majeur des bases de données est que toutes les requêtes sont gérées automatiquement. Il n'y a pas besoin de s'inquiéter quand une requête rate, quand votre connexion plante pendant l'envoi d'une requête ou quand deux utilisateurs modifient la base simultanément. Toutes les situations sont gérées de manière à ne pas détruire de la donnée ou corrompre la base en cas de problème.

Dans ce TP, on utilise le système de base données **PostgreSQL**. Vous n'avez même pas besoin de savoir que la base existe pour compléter cet exercice, mais ça vous sera utile dans le futur. Si vous voulez plus d'informations, cliquez sur ces liens :
`https://www.w3schools.com/sql/`
`https://en.wikipedia.org/wiki/PostgreSQL`
`https://docs.postgresql.fr/9.6/`
`https://www.postgresql.org/docs/9.6/static/index.html`

3.2.4 Les éléments importants du client

Le client est constitué d'un bon nombre de classes. Cette partie décrit celles qui sont importantes pour réaliser le TP.

Forms

Il y a six Windows Forms dans cette application, elles permettent d'accéder aux différentes fonctionnalités :

- Connexion / Inscription
- Match / Like / Dislike
- Profil
- Liste d'amis
- Chat
- Hack

Elles interagissent ensemble de la manière suivante :

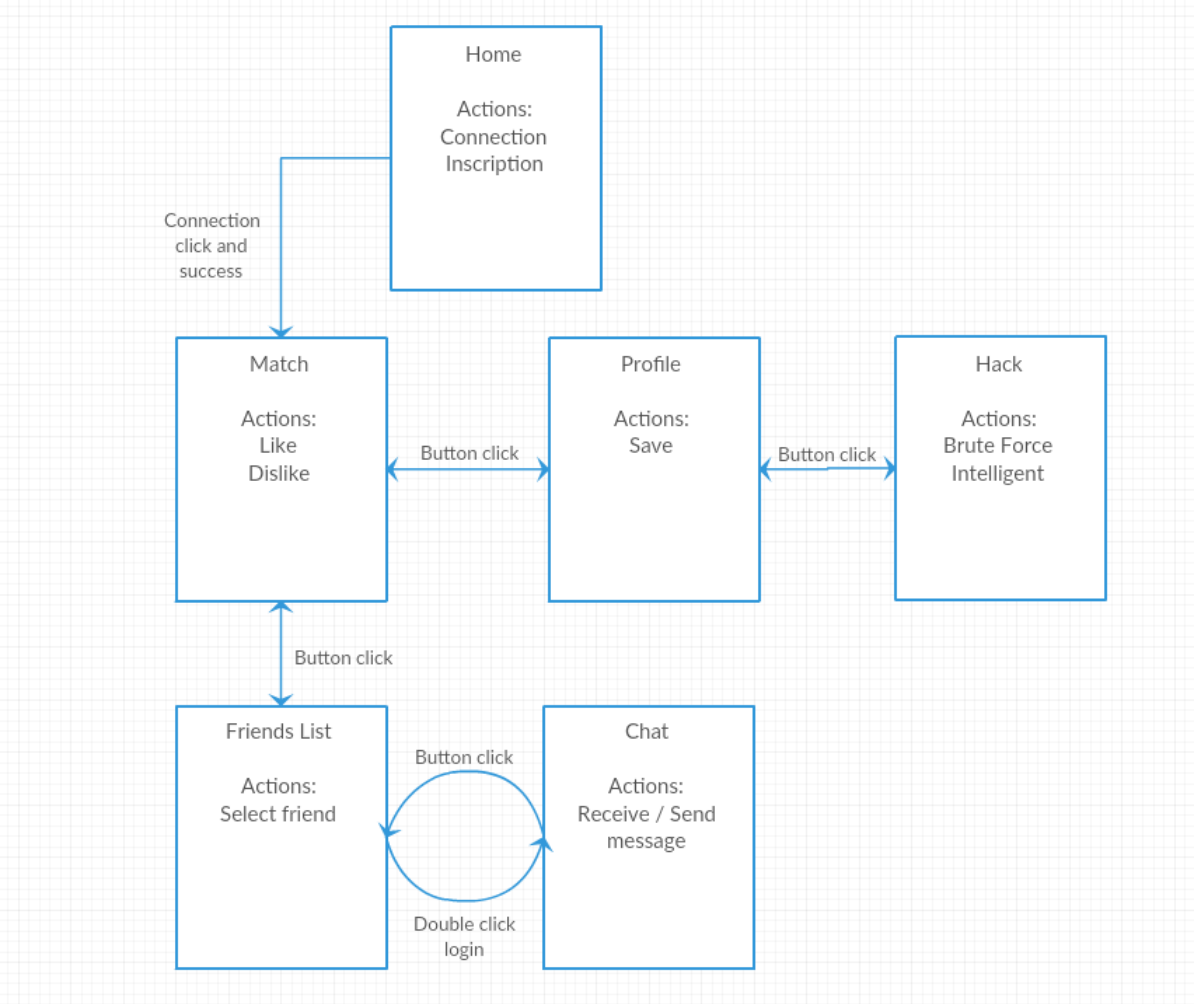

FIGURE 5 – L'architecture du client

Data

La classe **Data** stocke toute les informations nécessaires à l'exécution du client.

- Adress : l'adresse du serveur à contacter.
- Port : le port du serveur à contacter.
- Client : l'objet **TcpClient**.
- Forms : Toutes les windows forms utilisées par l'application.
- NextForm : La prochaine **form** à charger.
- Login : Login de l'utilisateur quand il est connecté.
- Friend : Ami sélectionné pour lui parler dans le Chat.
- Default : Profil de Dolores (pas de match).

La classe est **static** c'est-à-dire qu'elle n'est pas instanciable. Ainsi, la donnée de cette classe est accessible depuis n'importe où dans le code et est unique.

UserData

La classe **UserData** stocke un profil utilisateur. Elle contient toutes les informations nécessaires. Par exemple le profil de Dolores est un objet **UserData**.

- Login : le login du profil.
- Firstname : le prénom.
- Lastname : le nom.
- Age : l'age.
- Description : un texte de description.
- Picture : une photo de profil.
- AnimesSeries : Intérêt pour les animes ou les séries TV (vrai si oui sinon faux).
- Books : Intérêt pour livres (vrai si oui sinon faux).
- Games : Intérêt pour les jeux (vrai si oui sinon faux).
- MangasComics : Intérêt mangas et bandes dessinées (vrai si oui sinon faux).
- Movies : Intérêt pour les films (vrai si oui sinon faux).

Protocol

Contrairement à l'exercice 1, les communications avec le serveur ne consistent plus à envoyer de simple message. Nous avons besoin de données organisées. Pour cela nous n'envoyons plus une **string** transformée en tableau d'octets, mais un objet transformé en tableau d'octet. La classe utilisée s'appelle **Protocol**.
Le serveur à lui aussi cette classe et est capable de retransformer le tableau d'octet en objet **Protocol**.
La classe est composée de :

- Type : Le type de requête, allez voir le code pour connaître les différents types.
- Message : Un message, champ assez peu utilisé.
- Login : Un login, souvent utilisé dans les requête.
- Password : Un mot de passe, utilisé pour la connexion et l'inscription.
- User : Un objet de type **UserData**.

Chaque type de requête utilise entre zéro et tous les champs, selon leur besoin.

Formatter

Le **formatter** propose deux fonctions.
La première qui transforme un objet en tableau d'octet.
La seconde transforme un tableau d'octet en objet.
Cette classe utilisée pour sérialiser et déserialiser les objets **Protocol** pour pouvoir les envoyer au serveur.
Pour sérialiser les objets, ils sont d'abord transformé en une chaîne de caractères JSON puis

transformé en tableau d'octets.
Pour la déserialisation, il se passe exactement l'inverse.

Network

C'est la classe où vous devez coder l'exercice 2. Toutes les fonctions que vous avez à implémenter sont utilisée dans d'autres classes.

ReceiveMessage

Dans la classe **Network**, une fonction est déjà implémentée : **ReceiveMessage**. Il est conseillé de l'utiliser pour recevoir des messages du serveur. Vous pouvez cependant la recoder si vous le souhaitez.
Cette fonction est très lente, très peu optimisée, mais elle marche dans tous les cas, elle ne générera jamais de bug (normalement).

Pour envoyer un message au serveur, utiliser la fonction qui vous semble la plus adaptée et la plus simple. Il n'y a pas de contrainte à ce niveau. Il est important de se rappeler que pour chaque envoi d'un message au serveur, vous devez attendre une réponse de celui-ci.

Hacker Tools

01001110011011110010000001101001011011100110011001101111011100100110110101100
00101110100011010010110111101101110001000000111011101101001011011000110110000100000011000100110010100100000011001110110100101110111001100101011011100010000001100000101100011101111011101010111010000100000011101000110100001101001011100110010111000100000011010101011011100110010001100101011100100111001101110100001100001011011100100000100000011010010111010001110011001000000111010101110011011001010111001100100000011110010110111110111010101110010011100110110010101100011001100010111

3.2.5 Les fonctions à implémenter

Vous n'avez besoin de modifier que la classe **Network** pour les questions suivantes. Lorsque toutes ces fonctions seront codées, l'application marchera parfaitement.

ConnectSocket

Créer un objet **TcpClient** pour se connecter au serveur.
Utiliser les informations contenues dans la classe **Data**.
Il n'y a pas besoin de lancer ni d'attraper d'exception.
Il faut retourner le nouvel objet **TcpClient**.

```
1  public static TcpClient ConnectSocket()
2  {
3      //FIXME
4  }
```

CreateAccount

Demander au serveur de créer un nouveau compte avec les identifiants **login** et **password**.
Créer une nouvelle instance de **Protocol** avec les informations correctes.
Le type doit être **Create**, le **login** doit être mis dans le champs **Login** de l'instance et le **password** doit être mis dans le champs **Password**.
Utiliser la classe **Formatter** pour récupérer le tableau d'octets à envoyer.
Utiliser la méthode **Send** de la socket stockée dans l'objet **TcpClient** créé dans **ConnectSocket**

(il a été mis dans **Data.Client**).
Attendre une réponse du serveur (utiliser **ReceiveMessage**).
La valeur de retour doit être **vrai** si la réponse du serveur est de type **Response**, sinon **faux**.

```
1  public static bool CreateAccount(string login, string password)
2  {
3      //FIXME
4  }
```

Conseil

Toutes les fonctions suivantes ressemblent à **CreateAccount**, seul le format de l'objet **Protocol** et la valeur de retour changent. Seules ces étapes seront décrites dorénavant, mais il ne faudra pas oublier les autres.

ConnectAccount

Demander au serveur de se connecter à un compte avec les identifiants **login** et **password**.
Le type de **Protocol** doit être **Connect**, le **login** doit être mis dans le champs **Login** de l'instance et le **password** doit être mis dans le champs **Password**.
La valeur de retour doit être **vrai** si la réponse du serveur est de type **Response**, sinon **faux**.

```
1  public static bool ConnectAccount(string login, string password)
2  {
3      //FIXME
4  }
```

AskData

Demander au serveur le profil de l'utilisateur ayant le login donné en paramètre.
Le type de **Protocol** doit être **RequestData** et le **login** doit être mis dans le champs **Login** de l'instance. La valeur de retour doit être un objet de type **UserData** si la réponse du serveur est de type **Response**, sinon **null**.

```
1  public static UserData AskData(string login)
2  {
3      //FIXME
4  }
```

SendData

Mettre à jour son profil sur le serveur.
Le type de **Protocol** doit être **SendData**, le profil (user passé en paramètre) doit être mis dans le champ **User**. La valeur de retour doit être **vrai** si la réponse du serveur est de type **Response**, sinon **faux**.

```
1  public static bool SendData(UserData user)
2  {
3      //FIXME
4  }
```

SendLike

Envoyer au serveur si on *like* ou non un profil.
Le type de **Protocol** doit être **Like** ou **Dislike**, votre login doit aller dans le champ **Login** et le login de l'autre profil doit aller dans le champ **Message**. La valeur de retour doit être **vrai** si la réponse du serveur est de type **Response**, sinon **faux**.

```
1  public static bool SendLike(string login, bool like)
2  {
3      //FIXME
4  }
```

AskMatch

Demander au serveur un match, un profil pouvant devenir votre ami.
Le type de **Protocol** doit être **RequestMatch**, votre login doit aller dans le champ **Login**. La valeur de retour doit être un objet de type **UserData** si la réponse du serveur est de type **Response**, sinon **null**.

```
1  public static UserData AskMatch(string login)
2  {
3      //FIXME
4  }
```

AskFriends

Demander au serveur la liste de tous ses amis.
Le type de **Protocol** doit être **RequestFriends**, votre login doit aller dans le champ **Login**. La valeur de retour doit être un tableau de login (**string**) si la réponse du serveur est de type **Response**, sinon **null**. Le serveur, si tout se passe bien, donnera la liste d'ami dans le champ **Message**. Elle aura le forma suivant : "'login1' 'login2' 'login3'". Il faudra créer un tableau de **string** avec un login par **string** et enlever les guillemets simple.

```
1  public static string[] AskFriends()
2  {
3      //FIXME
4  }
```

MessageTo

Envoyer un message à un autre utilisateur en passant par le serveur.
Le type de **Protocol** doit être **MessageTo**, le login visé doit aller dans le champ **Login** et le message dans le champ **Message**. La valeur de retour doit être **vrai** si la réponse du serveur est de type **Response**, sinon **faux**.

```
1  public static bool MessageTo(string login, string message)
2  {
3      //FIXME
4  }
```

MessageFrom

Récupérer les message d'un autre utilisateur en passant par le serveur.
Le type de **Protocol** doit être **MessageFrom**, le login visé doit aller dans le champ **Login**. La valeur de retour doit être la string du champ **Message** si la réponse du serveur est de type **Response**, sinon **null**.

```
1  public static string MessageFrom(string login)
2  {
3      //FIXME
4  }
```

3.2.6 Les bonus

Hack me

Vous devez hacker les comptes des autre utilisateurs. Pour cela vous devez implémenter deux méthodes différentes : brute force et intelligente.

Pour la brute force, vous avez le choix, soit comprendre et utiliser la classe **HackerTools** qui vous permettra de coder la fonction très simplement, soit tout faire vous même. Vous devez compléter la fonction suivante dans la classe **Network** :

```
1  public static string HackBruteForce(string login)
2  {
3      //FIXME
4  }
```

Le login passé en paramètre est le login du compte à hacker.

Attention

La brute force est extrêmement lente, réellement extrêmement lente. Il est donc très fortement déconseillé de tester sur des mots de passe d'une longueur supérieure à 5 caractères. Des comptes avec des mots de passe facile à hacker seront mis à disposition. Leur login seront : hack0, hack1, hack2, hack3.

Pour le hack intelligent, on vous donne juste quelques indices :
- C'est simple.
- Il ne faut chercher très loin.
- C'est une faille qui a été ajoutée à la main juste pour le bonus.

Vous devez trouver comment faire, et utiliser la brute force ne fonctionnera pas. Bonne chance :

```
1  public static string HackIntelligent(string login)
2  {
3      //FIXME
4  }
```

Un beau Chat

Les Chats que vous avez créé sont moche, donc maintenant on veut qu'ils deviennent beaux et agréables à utiliser. Toutes les modifications que vous faites doivent être précisées dans le README. Vous pouvez faire ce bonus sur l'exercice 1 et/ou l'exercice 2. Il faudra l'expliciter dans votre README aussi.

Voici une liste non exhaustive de ce que vous pouvez faire :
- Mettre une couleur en fonction du locuteur.
- Envoyer la date dans le message pour pouvoir les trier par ordre d'envoi.
- Faire un historique des messages pour pouvoir les consulter (avoir un fichier qui stocke les messages et les charge au lancement de l'application).
- Afficher le login de l'envoyeur au dessus du message.

Si vous avez ça c'est déjà bien, mais si vous avez d'autres idées, n'hésitez pas.

3.2.7 Pour aller plus loin

Si le réseau vous intéresse ou vous intrigue, un second document est mis à votre disposition. Il n'est pas particulièrement difficile à comprendre, cependant il risque de consommer pas mal de temps vu qu'il aborde des notions nouvelles et un peu plus avancées. Il donne une utilité au serveur fourni et une réelle utilité à l'application **Rednit** tout entière.

Un complément de TP, qui ne sera pas évalué, est à votre disposition sur l'intranet.
Si vous souhaitez le montrer ou le faire évaluer (pour des points bonus éventuels) négociez avec vos ACDC respectifs.

These violent deadlines have violent ends.