TP C#13: Network

Submission instructions

You must submit a zip file with the following architecture:

```
rendu-tp5-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- Rednit/
        |-- Rednit.sln
        |-- Rednit_Lite/
            |-- Everything except bin/ and obj/
        |-- Rednit_Server/
            |-- Everything except bin/ and obj/
    |-- Chat/
        |-- Chat.sln
        |-- Client/
            |-- Everything except bin/ and obj/
        |-- Server/
            |-- Everything except bin/ and obj/
    |-- Bonus/
        |-- Every bonus done
```

You shall pay attention to check the following rules:

- Replace `firstname.lastname` with your own login and do not forget the `AUTHORS` file.
- `AUTHORS` and `README` files are mandatory.
- No `bin` or `obj` folder.
- Follow scrupulously the given prototypes.
- Remove all the tests from your code.
- **The code MUST compile !**

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where $ represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is `AUTHORS` with no extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

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

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

During this project you will learn about networking. You should at least be able to visualize how a network architecture works, which will be useful when you will study this subject in depth later.
This project also aims at being fun. Feel free to add some features, it should be accessible to everyone.

The first part of this lesson is very theoretical. It explains the basics of networking, and you must understand this part before moving on to the next one.

The second part describes the C# tools that will help you during this week. If you do not understand what to do in one of the exercises, please refer to the theoretical part and then to this part to find the right function(s) to use.

The first exercise is very simple, it is a direct application of the course. If you understood the theory, it shouldn't be a problem.

The second exercise is even simpler in terms of code. However, you will need to understand how the program works. The purpose of this exercise is to show you a different and more complete network application architecture than the exercise 1. Moreover, it is an interesting base for those who want to go further. Indeed, a bonus project, which will not be evaluated, is available on the intranet. It will help you to understand and, thus, be able to modify the application at your leisure.

2 Lesson

2.1 Internet ? Never heard of it.

There are a lot of ways to build communication between machines. We will focus on TCP/IP protocol, which is the basis of the Internet.

A protocol is a rule, a convention that allows users to communicate without compatibility issues. There are protocols at all network communication levels: physical media, network architecture, message format, etc. If you want more information about network protocols, Google is your friend.

Now that I've lost you, we'll start over with the basics.

2.2 We need to talk.

A network is a number of machines (2 or more) that communicate through common protocols.

A machine can be a computer, a laptop, a virtual machine, or any device with a network card.

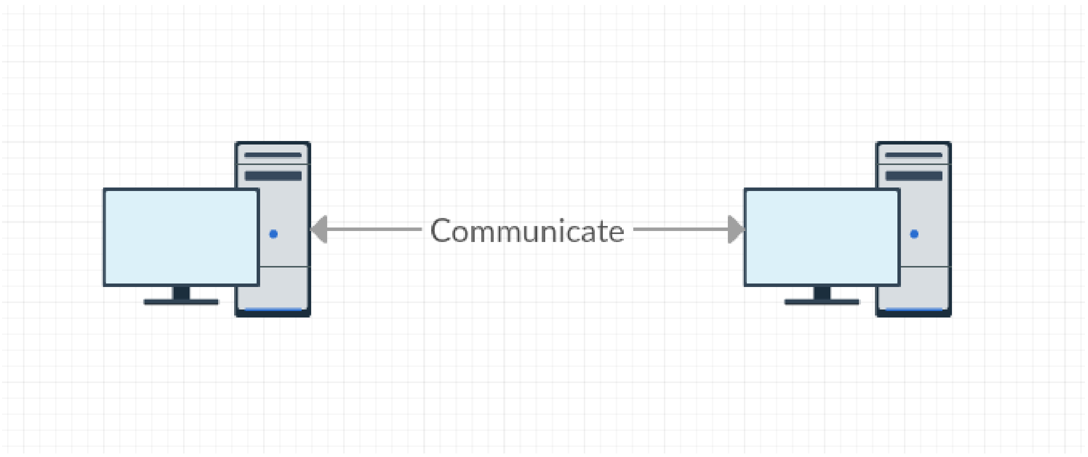

Figure 1: Communication

2.3 Sockets

A socket is a connection interface that allows local communication between processes on a machine or between machines connected on a network. Communication is possible in both directions on the same socket. Therefore, we can send and receive data at the same time.

In more simple terms, you can view a socket as a special file. To retrieve the message sent by the other machine, just read this one. To send a message, you must provide the data to send and the socket will take care of all the sending procedures itself.

So a socket is a kind of magic tool that will take care of retrieving and sending messages for you. However, it must be configured so that it behaves in the way you want. We will see later in this lesson how to use the Socket class in C#.

2.4 Adresses / ports / ships

Now that you understand approximately how two machines communicate, you might think that it is extremely simple. It is simple, but not that much. There are other factors that will complicate the whole thing.

The example above represents a connection between two users only. This is a simple case, but what happens if we want to connect 3 users or 5 or 10 or a thousand or 8 billion? How do we know who we are talking to?

Look at the schema below to visualize the set up. Consider the Internet cloud as a conglomeration of routers, servers, user machines, etc., all interconnected. And consider the four branches as pieces of this cloud that has been isolated.

A router is a box that allows a number of machines and / or routers to connect. For example, five machines connected to the same router will be able to communicate with each other.

So, in the example below, how can one recognize the machine of a user who is on another branch if we want to contact him ? How does the router know which cable to send the data to? It can not, so you need an extra system to do that.

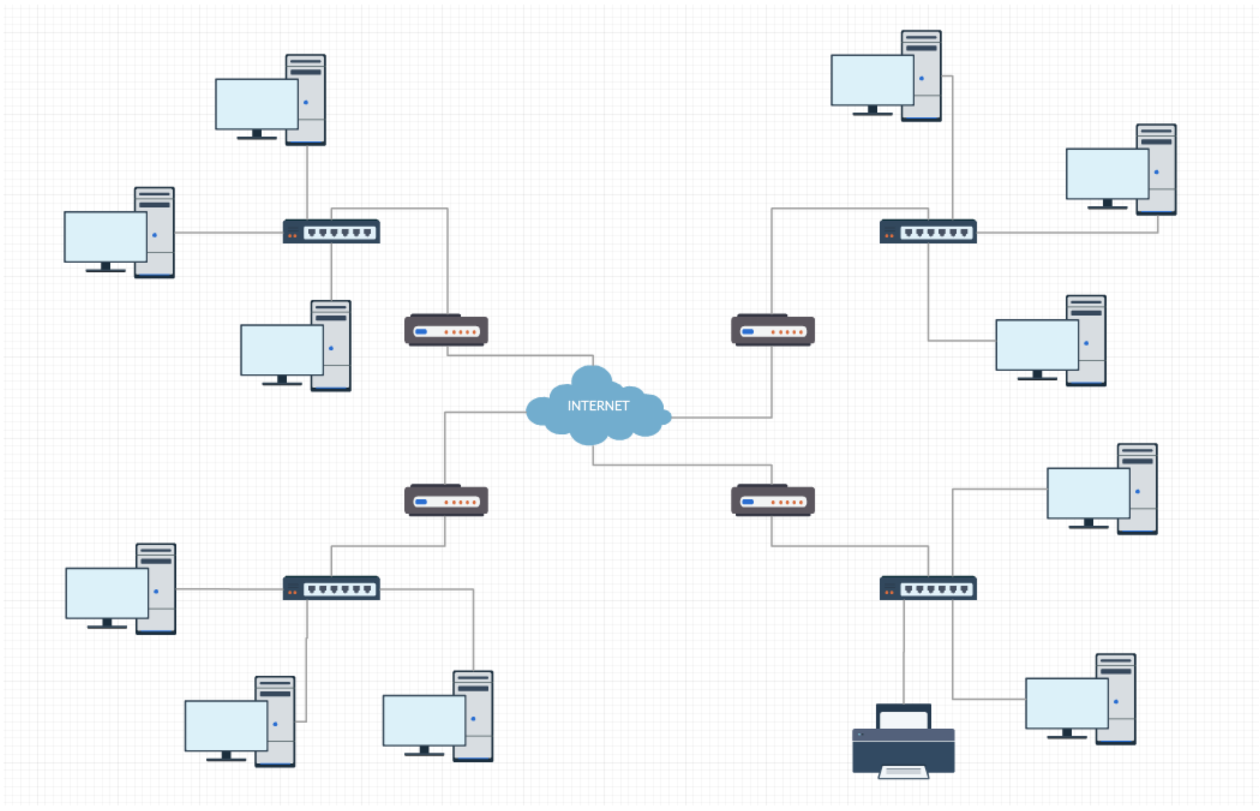

Figure 2: Internet

To fix this problem, an address system has been added. An address is assigned to each machine. Thus, to communicate, it is sufficient to indicate the address of the targeted machine. Try to type the 3 addresses in the diagram, then you may understand a little better.

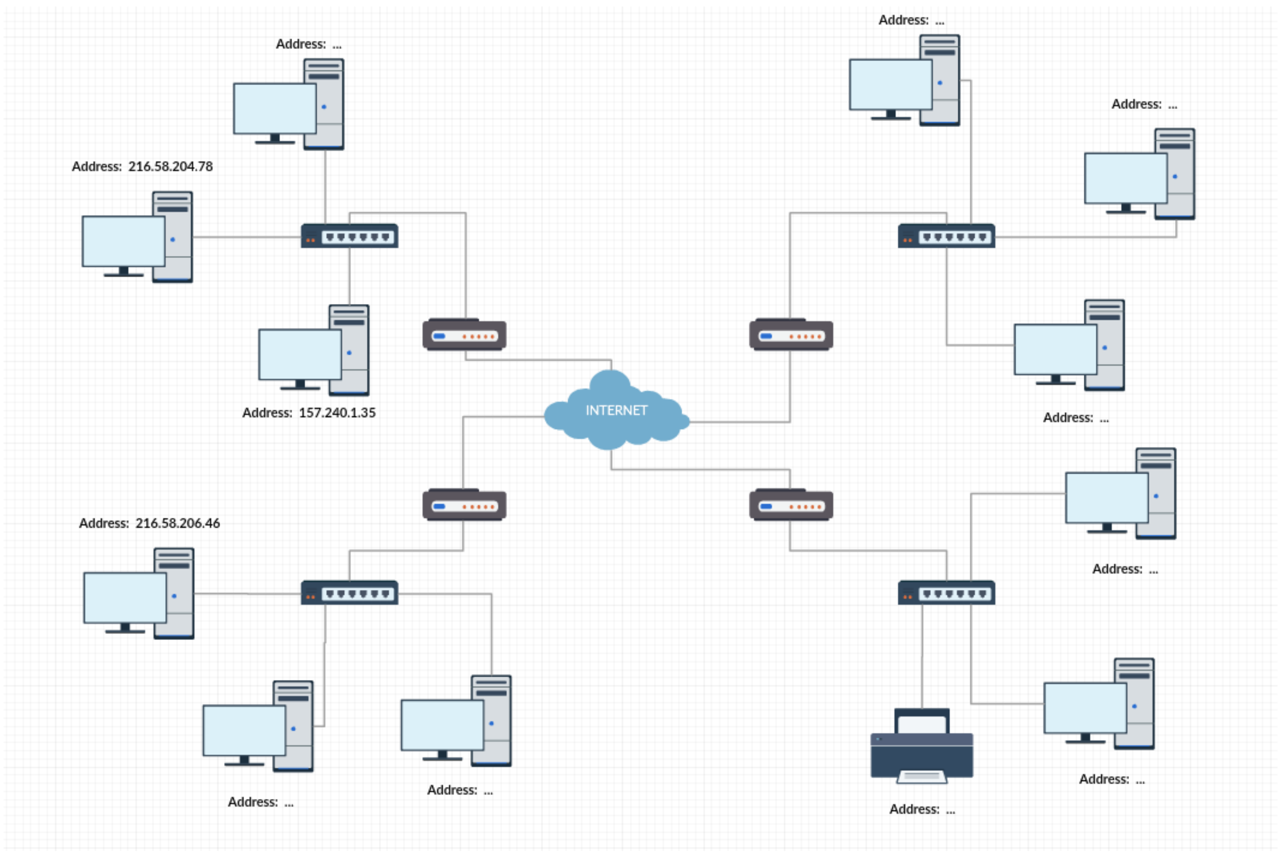

Figure 3: Addresses

A second problem is the maximum number of connections per machine. Our machine will need (preferably) 2 sockets to communicate with two other machines simultaneously. However, how do you distinguish them, how do you know which socket can communicate with a specific machine?

For example, you are on your computer and you want to open the intranet, but also Discord to **discuss** about your the project and an online game. So you have three applications. Each one will open at least one socket. How does a server know which socket has been opened for which use?

A port system has been established. A port is a 16-bit unsigned integer. Ports 0 through 1024 are reserved. Thus, a socket must be linked to a port in order to communicate with the outside.

The following wikipedia pages will help you, so you should read them.
`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`

To go further, you can read these pages. They deal with topics seen in this project, or technologies that you use in your everyday life.
`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 A little bit of practice

It is time to see concretely how two machines communicate. We will differentiate two roles: servers and customers. A machine can play both roles at the same time. When you connect to a web page, you are a customer who contacts a server. If you launch a website on your machine and external people can connect to it, then you are also a server.

Usually, servers are not launched on private machines. They are launched on machines hosted in large server rooms that the creator can administer remotely on his personal computer.

2.5.1 Server

What is a server exactly ? What is its purpose ?
A server accepts connection requests, customer requests and does what is asked of it. For example, we want to create an application that allows people to connect. Then, we want each user to be able to send messages. Those messages have to be received by all other users. The tasks of the server will be:

- Accept user connections.
- Receive all messages sent by users.
- Send received messages to all other users (everyone except the sender).

There are 4 steps to configure the socket in server mode:

- Create a socket and configure it according to your needs.
- Bind the socket to a port. The socket will now listen on the linked port.
- Listen to the port. Create a connection queue and enqueue each client that tries to connect.
- Accept connections. Accept a connection in the connection queue.

Thereby you have customers who can connect and communicate with your server.

2.5.2 Client

Now, what's the point of the customer?
The client role is to allow the user to use the application and the server functionalities. For example, a client for the server above would have at least the following features:

- Connect to a server.
- Write and send a message to the server.
- Display messages sent by the server.

It could have other features, but those are enough to have an application that anyone can use.
There are at least 2 steps needed to configure the socket in client mode:

- Create a socket and configure it according to your needs.
- Connect to a server. The socket must have valid address and port.

This way, your client can connect and communicate with a server.
It is important to visualize the difference for the rest of the TP, since this is one of the exercises you will have to do.

2.6 Why you should love C#

In this part of the course, you will have some explanations on the tools you will need during the project.

2.6.1 Socket

The **Socket** class in C# gives you access to a socket and all the functions to use it.

Constructor

You will have to use the following constructor:

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

The three parameters will be indicated during the exercises.

Send

The **Send** function sends an array of bytes to the associated socket (the receiver's socket).

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

The parameters are:

- **buffer** is the byte array to send.
- **size** is the size of the data to send. For example if you want to send only 16 bytes and the buffer is 1024, you must choose a size of 16.
- **socketFlags**, it must be set to None (SocketFlags.None).

The return value is the number of bytes sent.

Receive

The **Receive** function allows the user to receive a byte array from the sender.

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

The parameters are:

- **buffer** is the byte array to store the received data.
- **socketFlags** it must be set to None (SocketFlags.None).

The return value is the number of bytes received.

Connect

The **Connect** function is used only by the client. It allows the server to accept a new connection. Once the connection is accepted, it is possible to use the **Send** and **Receive** to communicate with the server.

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

The parameters are:

- **address** the server address.
- **port** the server port.

Bind

The **Bind** function is used only by the server. It allows the server to link a socket to a port. So the socket will 'listen' to this port to accept connections and communications. Once the socket is linked, it is possible to use the **Listen** and **Accept** to accept client connections. These functions are described below.

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

The parameters are:

- **localEP** The server address and port as an IPEndPoint object described below.

Listen

The **Listen** function is used only by the server. It creates a connection queue. After calling this function, the socket may receive connection requests from clients. Once the socket is in "listening" state, it is possible to use the **Accept** functions to accept connections stored in the queue.

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

The parameters are:

- **size** The maximum size of the queue (the maximum number of connection requests not yet accepted)

Accept

The **Accept** function is used only by the server. It accepts connection requests in the queue. The client must call this function, otherwise he and the server will not be able to communicate.

```
public Socket Accept()
```

The return value of the function is a socket. Any communication with the accepted client will be done through this socket.

2.6.2 TcpClient

The **TcpClient** class in C# is a wrapper of the **Socket** class. It wraps the Socket class with other useful functions. It also allows you to configure a socket in Tcp mode automatically.

Constructor

The **TcpClient** constructor configures the socket for TCP communication. Moreover, it connects the socket directly to the server. Therefore, there is no need to call the **Connect** function.

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

The parameters are:

- **first function** It takes as parameter the server coordinates as an **IPEndPoint** object.
- **second function** It takes as parameter the server address as a **string** and the port as an integer.

GetStream

The **GetStream** function allows you to retrieve and send data differently than **Send** and **Receive**. It returns a **NetworkStream**. It is a special stream that retrieves and sends the data, so it is accessible in reading and writing.

Socket

To access the socket that is in a **TcpClient**, use the **Client** field.

2.6.3 IPEndPoint

This is a convenient class since it allows one to store the address and the port together while making them accessible. In addition it allows one to automatically convert an address from several formats.

This class also gives access to some additional functions. It also does validity checks on the address and the port.

The two constructors are rather simple.

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

2.6.4 JsonConvert

We thought you could use the JSON parser that you coded during the Brainfuck project considering how well students did on this; however, for the sake of equality for the few who have not perfected this one, we decided to use a parser already done.

There are two useful functions to parse Json, one that transforms an object into a character string (this is called serialization) and one that transforms a string of characters (Json format) into an object (this is called de-serialization). .

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

2.6.5 Thread

When you work on a network, you often need to perform several tasks simultaneously. For example, we want to be able to receive and send messages at the same time, or accept connections while processing requests from already connected customers.

One solution (it's ONE solution not THE solution) is to use **Threads**. It allows the user to run several tasks simultaneously. For example, by executing the **Send** and **Receive** functions in two different threads, they will run at the same time and it will be possible to receive and send messages independently.

2.6.6 Windows Forms

Windows Forms are an extremely useful tool in C#. It allows the user to easily create a graphical interface for an application (with drag-and-drop, visual rendering and guided configuration). They can be used on Linux. However, if you want to create it you will have to install Visual Studio on Windows.

Read the first link for more information on Windows Forms and the second link to have a basic tutorial on it:
`https://en.wikipedia.org/wiki/Windows_Forms`
`http://csharp.net-informations.com/gui/cs_forms.htm`

Conseil

If you need more details on these features, ask Google: "C# <functionality>".

3 Exercices

3.1 Exercise 1 - Chat

In this exercise, you will have to implement a Chat system. It is simple, there is a server and n clients connected to it. A message sent by a customer must be received by all other customers.

Warning

It is not mandatory to handle special cases and we do not ask you to make an optimized Chat. The goal is that you understand what to do.

However, if you are interested to go further, you can always edit and improve the Chat. Modifications should not change the behavior of the requested functions.

Important

You have to use the **Socket** class and its function for this exercise. It is forbidden to use another class that would lead you to the same result.

3.1.1 Client

The client is very simple, there are three functions to implement in the **Client** class. You do not need to run or catch exceptions.

Constructor

You must create a new socket, put it in **_sock**, and connect it to the server. The socket parameters must be **InterNetwork**, **Stream** and **Tcp**. The server coordinates are passed as argument.

```
public Client(IPAddress address, int port)
{
    //FIXME
}
```

Reception

You must receive a message from the server. Its size can not exceed 1024 characters. You must display what you received in the **Console**.

Advice

To transform an array of bytes into a string, we advise you to use this function.

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

Using this function will put a lot of non-printable characters in your character string, I'll let you understand why and solve it yourself.
Hint: use the return value of **Receive()**.

```
public void ReceiveData()
{
    //FIXME
}
```

Sending

You must send a message to the server. Its size can not exceed 1024 characters. You must read the message to send from the **Console**.

Advice

To transform a string into an array of bytes, we advise you to use this function.

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

Use **ReadLine()** to read from the console.

```
public void SendData()
{
    //FIXME
}
```

3.1.2 Server

For the server, you have to implement four functions in the **Server class**.

Constructor

You must create the socket and initialize the **_clients** list. The socket must have the same settings as the client.

```
public Server(int port)
{
    //FIXME
}
```

Initializing

You must initialize the server in this function. For the bind, use **IPAdress.Any** (which will handle the address automatically) and the port stored in **_port**. The connection queue does not have to be too long, a size of 10 is enough. Finally, we want to be informed that the server is started, so you have to write in the console: "Server Started"

```
public void Init()
{
    //FIXME
}
```

Accept

You have to accept a client connection in this function. You must get the client socket, add it to the client list, and return the socket.

```csharp
public Socket Accept()
{
    //FIXME
}
```

Receiving

You must receive a message from the client given as a parameter. Its size can not exceed 1024 characters. It must then be sent to all other clients, using **SendMessage ()**.

Advice

You can take a leaf from the **ReceiveData()** function from the client.

```csharp
public void ReceiveMessages(Socket client)
{
    //FIXME
}
```

Sending

You must send a message to all clients except the sender and disconnected clients.

Advice

You can take a leaf from the **SendData()** function from the client.

```csharp
public void SendMessage(string message, Socket sender)
{
    //FIXME
}
```

Conclusion Congratulations you now have a functional mailing server. You can have fun with it to go further. If you want to change the behavior of the features, contact your respective ACDCs and try to negotiate.

However, a much more entertaining and thorough application awaits you in Exercise 2.

3.2 Exercise 2 - Rednit

Rednit is an application to make friends. It's inspired by the **Tinder** application.

3.2.1 Functionalities

The application is very complete, and has many features. You will have to make sure that all of them work at the end of the TP.

Create an account

You can begin by creating an account. This is the first step when you open the app for the first time. You can create an unlimited number of accounts.

Your login must be unique, if it is already being used, you will receive an error; and it must also be smaller than 50 characters. The password must be smaller than 50 characters, but it can be empty or made up of any character.

Warning

Do not put a password that you actually use, prefer passwords of the type "abc", "azerty", "123". All information is stored in clear and is, therefore, accessible to anyone. The security of the application has been artificially reduced for the purposes of the TP.

How to connect

Once you have created an account, you can sign in. Account creation is instant, so you can sign in immediately. After clicking the connected button, either you will get an error because what you entered is wrong, or you will be redirected to the **match** feature.

Set up your profile

You must set up your profile right after logging in for the first time. If this is not done, you will not be able to use the **match** feature. In the profile tab, there is some information that can be modified:

- **Firstname** : maximum size 50 characters, you can write anything.
- **Name** : maximum size 50 characters, you can write anything.
- **Age** : a number between 0 and 666.
- **Description** : a text with a maximum size of 1000 characters.
- **Photo** : you can select a new one by clicking on the picture.
- **Hobbies** : you must tick the boxes corresponding to your interests. If you do not tick any, you will not have access to the **match** feature.

After setting up your profile, you must click **Save** to save it on the server.

Match / Like / Dislike

The **match** feature will allow you to find potential friends based on interests. When you see a profile of another user, you have access to a "Like" and a "I do not like" (Dislike) buttons. A *Like* means that you want to become his friend, a *Dislike* means the opposite. If you *Like* a person and that person *likes* you too, it matches and you become friends. You can talk through the chat.

The way to find a user who suits you is:
Select all the people who *Liked* you but to whom you have not yet answered (you have not *Liked* or *Disliked* their profile). Among this selection, a random profile is kept.
If there is no one in the selection, then all people with at least one interest in common with you are selected. From this selection, a random profile is kept.

Chat

The Chat allows you to chat with your friends. You have to select a friend to talk to, then it will be possible to send messages. The Chat will not automatically update the received messages, so you have to click on the send button to display them. Sending an empty message updates the received messages without sending a message.

Hack

An exploit feature has been added. Securing your application is very important. You must make sure you never leave any breach. Those who will reach the end of this TP will understand how this application is full of flaws and the possibilities to break it. Note that the SQL injections were still blocked as they are too violent.

3.2.2 Purpose

In this exercise you will have to code the network part of the client. It will, therefore, be necessary to modify the **Rednit_Lite** project.

What to do

There are 12 functions to code to complete the customer's network. However, you will need to understand how the application works as a whole.

Complements

The server provided in the archive is not useful for this exercise. It is used for TP supplements, to go further.

Tools

To be able to test the client that you are coding, an application server is available online. All clients (you) can thus connect to this server. This means that each of you can see and interact with other students in the graduating class.

Important

Since the server is public and open to the entire graduating class, any addition of content that may be considered indecent or inappropriate will be penalized with a zero on the TP score and possibly other disciplinary sanctions depending on the severity.

In case of abuse, the server may be closed. In this case, you will need to configure and launch the server yourself in order to run the client.

3.2.3 Description of the application

The application is divided into three parts:

- The client available to the user so that he can use the application.
- The server that responds to client requests and allows interactions between users.
- The database that stores all user and application data.

So the user communicates with the server through the client.
The server responds to client requests and sends the data to be stored in the database.
The database stores, modifies and sends its data according to the requests from the server.

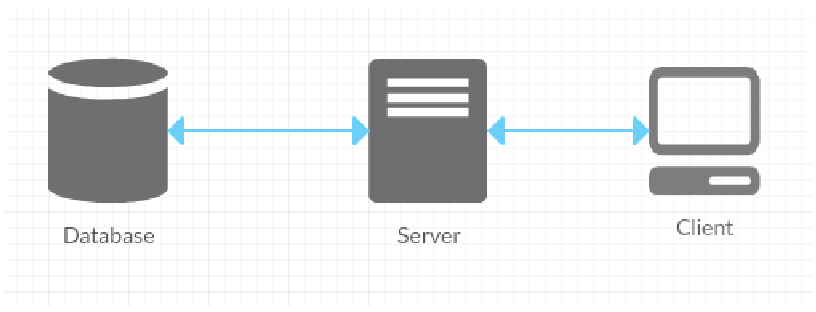

Figure 4: Architecture of the application

Database

A database system allows one to store data in an organized manner. It saves a very large amount of information on an application.

A database has several advantages that make it convenient and easy to use.

A database is created in a database server. It is very easy to set up and does not require network knowledge. Then once the server is started, creating a database requires only one line of code.

The use of a database is also very simple (as long as you do not get too deep in it). The first step is to connect to it with a login and a password. Then it works with a query system written in a certain language (mostly SQL). With these queries, it is possible to query, modify and delete the database, depending on the permissions of the connected user, of course.

Another major advantage of databases is that all queries are handled automatically. There is no need to worry when a request fails, when your connection crashes while sending a request, or when two users modify the database simultaneously. All situations are managed to avoid destroying data or corrupting the database in case of problems.

In this project, we use the **PostgreSQL** database system. You do not even need to know that the base exists to complete this exercise, but it will be useful in the future. If you want more information, click on these links:
`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 Important elements of the client

The client contains many classes. This part describes those that are important to complete the project.

Forms

There are six Windows Forms in this application, which provide access to the different features:

- Connection / Inscription
- Match / Like / Dislike
- Profile
- Friends list
- Chat
- Hack

They interact together as follows:

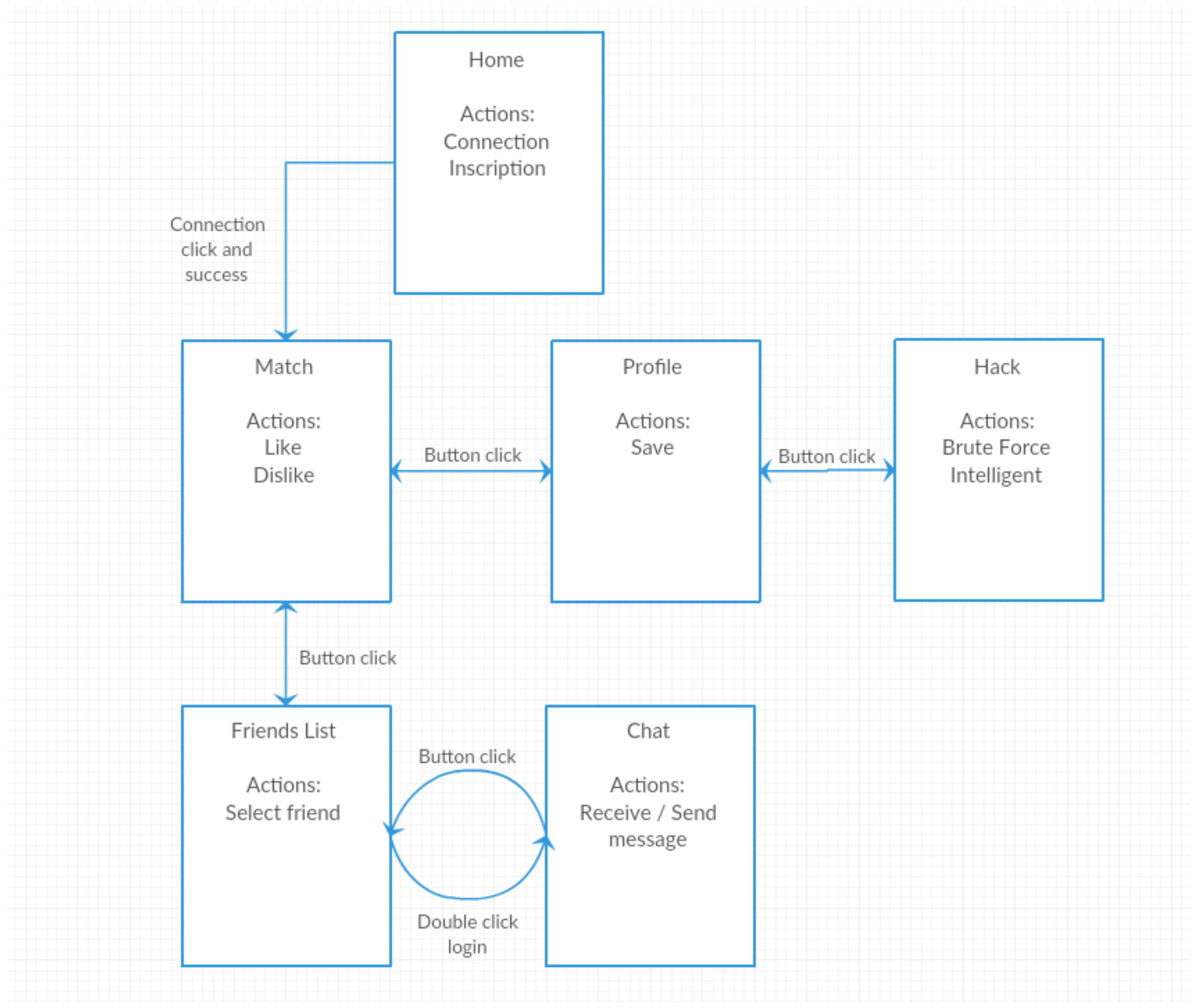

Figure 5: Architecture of the client

Data

The **Data** class stores all the information needed to execute the client.

- Adress: the address of the server to contact.
- Port: the port of the server to contact.

- Client: the **TcpClient** object.
- Forms: All windows forms used by the application.
- NextForm: The next **form** to load.
- Login: User login when he is logged in.
- Friend: Friend selected to talk to in the Chat.
- Default: Dolores's Profile (no match).

The class is **static**, it is not instantiable. Thus, the data of this class is accessible from anywhere in the code and is unique.

UserData

The **UserData** class stores a user profile. It contains all the necessary information. For example the profile of Dolores is a **UserData** object.

- Login: the profile login.
- Firstname: the firstname.
- Lastname: the lastname.
- Age: the age.
- Description: a description text.
- Picture: a profile picture.
- AnimesSeries: Interest for anime or TV series (true if yes, else false).
- Books: Interest for books (true if yes, else false).
- Games: Interest for the games (true if yes, else false).
- MangasComics: Interest mangas and comics (true if yes, else false).
- Movies: Interest for films (true if yes, else false).

Protocol

Unlike Exercise 1, communication with the server no longer consists of sending a simple message. We need organized data. To achieve this we do not send a string transformed into an array of bytes anymore, but an object transformed into an array of bytes. The class used is called **Protocol**.
The server also has this class and is able to retransform the byte array into a **Protocol** object.
The class is composed of:

- Type: The type of request, the different types are written in comments in the code.
- Message: A message, not often used.
- Login: A login, often used in the requests.
- Password: A password, used for login and registration.
- User: A **UserData** object.

Each type of query uses between zero and all fields, according to their need.

Formatter

The formatter offers two functions.
The first that transforms an object into a byte array.
The second transforms a byte array into an object.
This class is used to serialize and deserialize **Protocol** objects so that they can be sent to the server.
To serialize the objects, they are first transformed into a JSON string and then transformed into an array of bytes.
For deserialization, the opposite happens.

Network

This is the class where you must code exercise 2. All the functions you have to implement are used in other classes.

ReceiveMessage

In the **Network** class, a function is already implemented: **ReceiveMessage**. It is recommended to use it to receive messages from the server. You can, however, recode it if you wish.
This function is very slow, very little optimized, but it works in all cases, and will never generate a bug.

To send a message to the server, use the function that seems the most suitable and the simplest. There is no constraint at this level. It is important to remember that for each message sent to the server, you must wait for a response from it.

Hacker Tools

0100111001101111001000000110100101101110011001100110111101110010011011010101100
0010111010001101001011011110110111000100000011101110110100101101100011011000010
0000011000100110010100100000011001110110100101110110011001010110111000100000011
0000101100010011011110111010101110100001000000111010001101000011010010111001100
1011100010000000101010101101110011001000110010101110010011100110111010001100010
1101110011001000010000000110100101110100011100110010000000111010101110011011001
01
0111001100100000011110010110111101110101011100100111001101100101011011000110011
000101110

3.2.5 The functions to implement

You only need to change the **Network** class for the following questions. When all these functions are coded, the application will work perfectly.

ConnectSocket

Create a **TcpClient** object to connect to the server.
Use the information from the **Data** class.
You do not need to throw or catch any exception.
You have to return the new **TcpClient** object.

```
public static TcpClient ConnectSocket()
{
    //FIXME
}
```

CreateAccount

Ask the server to create a new account with the identifiers **login** and **password**.
Create a new instance of **Protocol** with the correct information.
The type must be **Create**, the **login** must be put in the **Login** field of the instance and the **password** must be put in the **Password** field.
Use the **Formatter** class to retrieve the array of bytes to send.
Use the **Send** method of the socket stored in the **TcpClient** object created in **ConnectSocket** (it was put in **Data.Client**).
Wait for a response from the server (use **ReceiveMessage**).
The return value must be true if the server response is **Response**, false otherwise.

```csharp
public static bool CreateAccount(string login, string password)
{
    //FIXME
}
```

Advice

All of the following functions look like **CreateAccount**, only the format of the **Protocol** object and the return value change. Only these steps will be described from now on, but it will not be necessary to forget the others.

ConnectAccount

Ask the server to connect on an account with the right **login** and **password**.
The **Protocol** type must be **Connect**, the **login** must be put in the **Login** field of the instance and the **password** must be put in the field **Password**
The return value must be true if the server response is a **Response** type, false otherwise.

```csharp
public static bool ConnectAccount(string login, string password)
{
    //FIXME
}
```

AskData

Ask the server the profile of the user with the login given as parameter.
The **Protocol** type must be **RequestData** and the **login** must be in the **Login** field of the instance. The return value must be an object of type **UserData** if the response of the server is of type **Response**, null otherwise.

```csharp
public static UserData AskData(string login)
{
    //FIXME
}
```

SendData

Update one's profile on the server.
The **Protocol** type must be **SendData**, the profile (user given as parameter) must be put in the **User** field. The return value must be true if the server response is **Response**, false otherwise.

```
public static bool SendData(UserData user)
{
    //FIXME
}
```

SendLike

Send to the server if we like or dislike a profile.
The **Protocol** type should be **Like** or **Dislike**, your login should go to the **Login** field and the other profile's login should go to the **Message** field. The return value must be true if the server response is a **Response** type, false otherwise.

```
public static bool SendLike(string login, bool like)
{
    //FIXME
}
```

AskMatch

Ask the server for a match, a profile that can become your friend.
The **Protocol** type must be **RequestMatch**, your login must go to the **Login** field. The return value must be an object of **UserData** type if the response of the server is of type **Response**, null otherwise.

```
public static UserData AskMatch(string login)
{
    //FIXME
}
```

AskFriends

Ask the server for a list of all his friends.
The **Protocol** type must be **RequestFriends**, your login must be in the **Login** field. The return value must be a login array if the server response is of type **Response**, null otherwise. The server, if all goes well, will give the friend list in the **Message** field. It will have the following form: "'login1' 'login2' 'login3'". It will be necessary to create an array of strings with one login by string and to remove the single quotation marks.

```
public static string[] AskFriends()
{
    //FIXME
}
```

MessageTo

Send a message to another user through the server.
The **Protocol** type must be **MessageTo**, the intended login must be in the **Login** field and the message in the **Message** field. The return value must be true if the server response is **Response**, false otherwise.

```
public static bool MessageTo(string login, string message)
{
    //FIXME
}
```

MessageFrom

Retrieve the messages of another user through the server. The **Protocol** type must be **MessageFrom**, the login must be in the **Login** field. The return value must be the string of the **Message** field if the server response is of type **Response**, null otherwise.

```
public static string MessageFrom(string login)
{
    //FIXME
}
```

3.2.6 The bonus

Hack me

You must hack the accounts of other users. For this you need to implement two different methods: brute force and intelligent.

For the brute force, you have a choice: either understand and use the class **HackerTools** which will allow you to code the function very simply, or do everything yourself. You must complete the following function in the **Network** class:

```
public static string HackBruteForce(string login)
{
    //FIXME
}
```

The login passed as parameter is the login of the account to hack.

Warning

Brute force is extremely slow, really extremely slow. It is, therefore, strongly inadvisable to test on passwords longer than 5 characters. Accounts with easy hacker passwords will be made available. Their login will be: hack0, hack1, hack2, hack3.

For the smart hack, we just give you some clues:

- It's simple.
- Do not search for something too complicated.
- It's a breach that has been added just for the bonus.

You have to find out how to do it, and using the brute force will not work. Good luck:

```
public static string HackIntelligent(string login)
{
    //FIXME
}
```

A pretty chat

The chats you created are ugly, so now we want them to become beautiful and enjoyable to use. Any changes you make must be specified in the README. You can do this bonus on exercise 1 and / or exercise 2. It will be necessary to explain it in your README too.
Here is a non-exhaustive list of what you can do:

- Put a color according to the speaker.

- Send the date in the message to sort them by sending order.
- Make a history file of the messages to be able to consult them (a file which stores the messages and loads them at the launching of the application).
- View the sender login above the message.

If you have that, it's good enough, but if you have other ideas, do not hesitate.

3.2.7 To go further

If you are interested in the network, a second document is at your disposal. It is not particularly difficult to understand. However, it will likely consume a lot of your time as it deals with new concepts and a little bit more advanced. It gives utility to the server provided and real utility to the entire Rednit application.

A complement of the project, which will not be evaluated, is available on the intranet.
If you want to show it or have it evaluated (for potential bonus points) negotiate with your respective ACDCs.

These violent deadlines have violent ends.