TP C# 14: TFTP

1 Guidelines

At the end of this practical, you will submit an archive that respects the following architechture:

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- TFTP.sln*
    |-- TFTP*
        |-- Everything but bin/ and obj/
```

Of course, you must replace "`login_x`" by your own login. All the files marked with an asterisk are *mandatory*.

Verify the following before submitting your work:

- The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
- No `bin` and `obj` directories in your project.
- Remove *all the tests* from your code.
- **The code must compile!**

2 Introduction

2.1 Objectives

For this last practical, you will have to make good use of all the knowledge you have acquired this year in order to work on a "real" project. `fact` and `fibo`'s time is over ; the work you accomplish will be usable as a true program.

The subject chosen is the implementation of a program that respects the TFTP protocol. The final goal is the ability to tranfer files through the network, between a server and a certain number of clients. All the knowledge you need will be explained in the tutorial part of this practical.

This practical is freer than the previous ones. No class architechture is required, and it is time for you to show your ability to work on a real project. Don't hesitate to ask your assistants for help for your code structure, or to make sure you truly understand the protocol that we ask you to follow.

Finally, it also is an opportunity for you to have fun developing an application entirely by yourself, so give it everything you have, spend some time on it, and... have fun!

3 Advanced C#

3.1 Serialization

The *serialization* of data is the process of transforming a data structure (or an object) into a form that can be stored or sent (through the network for example). The inverse process, which is the making of the original data structure from its serialized form, is called.. the deserialization, of course.

A serialization function takes the object to serialize as argument and returns a byte sequence (in C# , it will be a `Byte` array). There is no specific serialization function to each data structure. What is important is to be certain that the deserialization function will always give the original object. (This means that `myObject.equals(deserialize(serialize(myObject)))` will always be verified).

3.2 Socket

In the first practical about networking, we manipulated the TCPClient class to send datas to a TCPListener. This approach is very useful because it hides what really happens inside the computer. The important underlying element in these classes are **sockets**.

The `Socket` class (in the `System.Net.Sockets` namespace) gives access to a *lower programming level*, such as `Send` and `Recieve`, for the connected mode (TCP). In this practical, sockets will be used in unconnected mode (UDP), and will be using other functions: `SendTo` and `RecieveFrom`. These functions don't manipulate streams, but rather byte arrays directly.

An example of usage of the `Socket` class will be given in the next part of this tutorial.

4 Network

The current practical work is based on the knowledge you've previously acquired, which is why the whole vocabulary is not fully redefined here. Moreover this is a summary representing the different notions to have in mind for accomplishing of this practical work. You should do personal research in order to clarify any fields you may be unconfortable with.

4.1 Network stack and IP protocol

The word network has such a large definition that we will not look at its full meaning but instead a subset of it. You will manipulate one protocol in one layer: the UDP protocol in the transfer layer. But first, what are these layers?

On one hand, the *ISO* (International Organization for Standardization) describes the *OSI* model ((Open System Interconnection) that represents the network in seven layers. On the other hand, gurus at *DARPA* offers the *TCP/IP* model that is much less theoretical and divides the network into four layers.

The four layers of the *TCP/IP* protocole are the link layer, the internet layer that holds the *IP* protocol, the transport layer that holds the *UDP* protocol and the application layer that holds the *TFTP* protocol. As you can see, each protocol are linked by an inclusion relation like an onion. This relation is clearly represented on the next figure.

The *IP* protocol (Internet Protocol) of the Internet (Interconnected Network) layer is a communication protocol responsible for the transport of paquets from one IP to another. The signification of transport in this context refer to addressing and routing paquets through the network. In order to address packet, an IP address is needed of course. The shape of these addresses is defined by the version of the IP protocol (mainly IPv4, or IPv6). The IP protocole gives a logical network layer over a physical one.

As the name suggests, *Internet* is not a single whole entity but rather an interconnexion of smaller networks. These networks are connected together by *routers* that route packets, that is, which transport packets from one network to another.

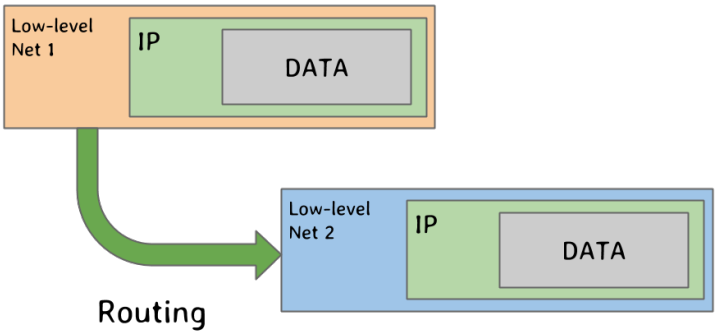

Figure 1: Routing

4.2 UDP

The protocol of interest today is the *UDP* (User Datagram Protocol) protocol. The other widely used one is *TCP* (Transfmission Control Protocol).

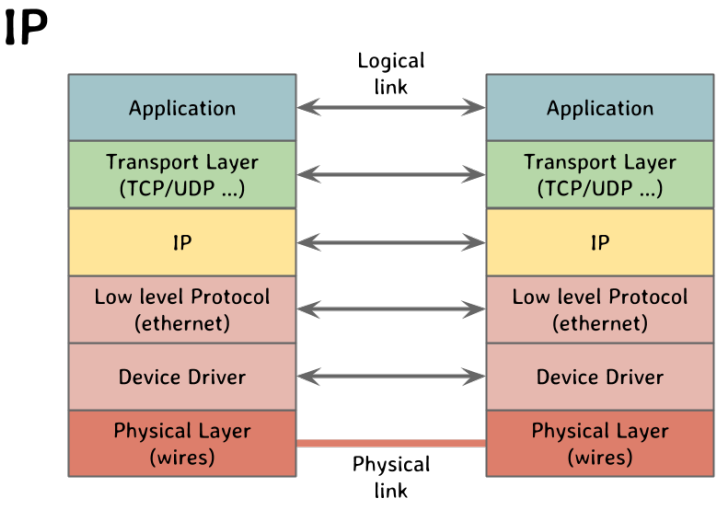

Figure 2: IP

The transport layer is on top of the IP layer and is responsible for the point-to-point communication between the applications. This is the layer that ensure the integrity of the data, but also multiplexing, and transport reliability. The UDP protocol does not guarantee that the

data will arrive someday, unlike TCP. UDP packets are called datagrams. We say that the UDP protocol works in connectionless mode, thus facilitating broadcast and multicast operations.

4.3 Client & Serveur

Most of the time, an online app can be split into two programs: a client and a server. (In the first practical about networks, you developed the client).

- The server is a program that is launched only once, and stays running permanently. As long as no client connects to it, the server keeps waiting. When a client comes, it sends a request to the (still waiting) server which analyses it and sends back a response.

- The client will perform one or more requests to the server and process the response (such as dispaying it).

For instance, a web browser is used to access the web. When you type http://epita.fr in the URL bar, the browser (the client) will send a request to the web server (apache, nginx, myhttpd...), which will interpret your request and send back the requested page.

4.4 C# API

As the *API* is huge, here are some tips that will help you in your research on MSDN for the method that fits your needs.

```
using System.Net.Sockets;

IPHostEntry hostEntry = Dns.GetHostEntry(Dns.GetHostName());
IPEndPoint endPoint = new IPEndPoint(hostEntry.AddressList[0], 69);

Socket sock = new Socket(
                endPoint.Address.AddressFamily,
                SocketType.Dgram,
                ProtocolType.Udp
                );
sock.ReceiveTimeout = 1000;
sock.SendTo(buffer, senderEP);
int len = sock.ReceiveFrom(buffer, ref senderEP);
```

- IPHostEntry: "The IPHostEntry class associates a Domain Name System (DNS) host name with an array of aliases and an array of matching IP addresses." [1]

- IPEndPoint: "The IPEndPoint class contains the host and local or remote port information needed by an application to connect to a service on a host. By combining the host's IP address and port number of a service, the IPEndPoint class forms a connection point to a service." [2]

[1] (https://msdn.microsoft.com/en-us/library/system.net.iphostentry(v=vs.110).aspx)
[2] (https://msdn.microsoft.com/en-us/library/system.net.ipendpoint(v=vs.110).aspx)

- Socket: "The Socket class provides a rich set of methods and properties for network communications. The Socket class allows you to perform both synchronous and asynchronous data transfer using any of the communication protocols listed in the ProtocolType enumeration." [3]

If you wonder why we need to specify the reciever on the `send` or `recieve` calls, that's only because we're working in connectionless mode. It is also possible however to call the `connect` method on a connectionless *socket*. This behaviour will not take into account the packets from the given `EndPoint`. Thus, it will be possible to use `send` instead of `sendTo` but it will also force you to create a new socket for each client because you won't be able to reuse it.

4.5 TFTP

TFTP stands for "Trivial File Transfer Protocol". Tt's a very simple file transfer protocol. It works on the following principles:

1. The client makes an initial request for reading or writing a file.
2. The server answers if an error has occured or not. One possible error, for example, is the request for reading a non existing resource.
3. The transfer now begins. It is composed by a sequence of datagrammes followed by an acknowledgment packet in a successive order.

Each protocol over the internet is described in an official document called **RFC** ("Request For Comment") associated with its number. In our case, TFTP is the *RFC1350*. It's a very small RFC, which is the main reason we chose it for this practical work. You must read it all, maybe more than once, and follow the behaviors described in this document. Please note that every question you may have on the TFTP protocol is answered there. You can find one version of this document here: http://www.ietf.org/rfc/rfc1350.txt

4.6 Tips

Although this practical will require some time, it is not that complicated. You do, however, need to read and gather a lot of information in order to be able to fully understand the whole subject.

You will implement a full application that can be launched as a client or a server, according to the options given to the command line following the RFC1350. We will give you some functions that you have to use as reference points in order to develop your implementation, but you are free to work as you see fit. Your grade will depend on the functionalities given in the RFC.

You will mainly have to work with byte arrays that will fill your datagrams, and thus to transmit your files on through the network.

[3] (https://msdn.microsoft.com/en-us/library/system.net.sockets.socket.aspx)

5 Exercise

As you already know, the only exercise in this practical is the development of a program that can be launched as a TFTP client or server. To help you begin your work more easily, here are some indications that will probably help you.

5.1 Parsing options, launching the application

First, you should parse options given in the command line in order to define how your application should behave. You can both launch it in server mode to handle client requests or in client mode to read or write a file into the server.

For example, here is the options handled by the reference:

```
Usage: myTFTP [-E [-e <key>]] <operation> <mode> <target>

mode:           Client or Server (-s)
operation:      Read (-R) or Write (-W)
file:           In Server mode this is the working directory,
                In Client mode IP:Path
```

As you will see, there are some options that are not handled by the RFC. These are bonuses that you can do if you have already finished implementing the full RFC. The extended mode activated by the -E option extends the features of the RFC and so transfers files heavier than 32MB and activates some options like the encryption of data between client and server (-e).

You may use a structure to store options given to the application. This will allow you quick access to these later.

5.2 Quick note on classes

5.2.1 IO

During the lifecycle of your application, it will mainly do two different things: read from a file on the disk and send it through the network in chunks of paquets of the same size (except for the last one) or write incoming file from the network to the disk. Therefore, two very similar classes...

It might be worth thinking about the bonuses from the beginning, at least preparing them. It may be a smart move to start processing the data stream while transfering it from the source to the destination with your IO module. For example, let's say that I receive 512 bytes and I want to increase by one the value of each of them. We could create a function that takes one byte array and returns the processed byte array. Think about the delegates that we have seen previously.

5.2.2 Datagrams

To be able to transmit information, you need to respect well defined rules described in the RFC, like a short header followed by 512 bytes maximum of data for a DATA datagram. It would be wise to have a class representing this packet, to be able to manipulate it as an object, change its type and its content, and finally, ask for its serialized form. As a result it becomes easier to create a packet, transform it into an array of bytes and then to send it using your UDP socket. This class is **mandatory** and also needs to override the `ToString()` method, which must return a representation following this format:

```
-----------------------------------------
Type:   RRQ
File:   testfile.ext
Mode:   OCTET
-----------------------------------------
Type:   DATA
Block:  1
Data:   content_of_the_file
-----------------------------------------
Type:   ACK
Block:  1
```

These are three datagrams, the first one being a request that could send a client to a server in order to fetch the 'testfile.ext' file. The second would be the response from the server, in the case that the file does exist and its content is at least 512 bytes (which is the case here). The last one would be the response from the client, after he recieved the second datagram. Don't forget to handle the error datagrams, it will help you to understand what is wrong during the implementation.

5.3 Misc

As you probably noticed, the TFTP protocol tends to be in a lock situation (for instance, a client sends a request and then closes the connexion). It is possible to avoid this situation using timeouts. You can't handle the server properly without using the timeout properties from your socket. To do so, as always, have a look at the MSDN page about the Socket class.

5.4 Bonii

The reference implements a symetric encryption option, which can be applied to the stream during the transmission, but you can imagine much more. For those who are looking for ideas, you might want to compress the data, the LZ77 compression technique would be a great choice. Also, you can use steganography on the unused bytes from the identifier of the packet.

Bullshit is strong with this one...