Alberto Blanc
02 99 12 24 25
alberto.blanc@imt-atlantique.fr

Christophe Couturier
02 99 12 24 01
christophe.couturier@imt-atlantique.fr

Jean-Pierre Le Narzul
02 99 12 70 41
jp.lenarzul@imt-atlantique.fr

Gwendal Simon
02 99 12 70 48
gwendal.simon@imt-atlantique.fr

UV MIN RES - RES 209 : Projet réseaux
Travaux pratiques

Printemps 2017

Responsables du module: A.Blanc, J.P.Le Narzul
Département SRCD

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Contents

RES 209 TP-1 Twisted/UDP Programming
Spring 2017

Git Tutorial

You are going to use the git version control system throughout RES 209. It is essential for each student to be able to use git to share the Python code that he/she writes. In this introductory exercise, you are going to learn the basics about git and version control. For some obscure reasons (related to the version of NFS being used when mounting the `/homes` directories), the behavior of Git is a bit weird when working in `/homes/yourlogin`. Instead of using `/homes/yourlogin`, you will use `/sansauvegarde/homes/yourlogin` to host your work. The fact that this filesystem is not backuped is not a problem as you will use the Git version control system to remotely store your work.

Exercise 1 Create a Directory to Host your Work

You first have to create a directory in the `sanssauvegarde` file system.

To Do

Open a terminal window and type the following commands:

```
cd /sanssauvegarde/homes/yourlogin
mkdir network
cd network
```

Exercise 2 Create a Git Working Copy and Configure Git

The first step is to clone a repository, that is to create a local copy of the code that the teaching staff has prepared and that you are going to use as a starting point.

You will use the git versioning/revision system for the project and the programming assignments. For the project, each group has four students. We will denote the group number by `X`. For the first two programming assignments, you must work in pairs. It is up to you to split each group into two pairs (subgroups). We will denote by `Y` the subgroup number (`Y={1,2}`). In each subgroup, there are two users: USER A and USER B.

Who is logged in?

YOU ARE LOGGED AS USER A

To Do

You first have to create a local copy of the files that are stored on the git server. Change your working directory to `network` (`cd network`) and type the following command in the Terminal (Recall that `X` is your group number):

```
git clone -o redmine-tb https://redmine-df.telecom-bretagne.eu/git/r328-s17-gX r328
```

At this point a new folder called `r328`, contains all the files for RES 209, both for the project and for the first two programming assignments. For this programming assignments, you are especially interested in:

- The folder `sgY/gitsandbox`, which contains the files for the Git tutorial.
- The folder `sgY/sibyl/scripts`, which contains the scripts to run the code that you will implement. Do not modify these files.
- The folder `sgY/sibyl/protocol`, which contains the Python files for your code. These are the files you will edit during the first two programming assignments.

To Do

Configure Git using the following commands in a terminal. **please use your name and your school email address rather than "John Doe" and "johndoe@imt-atlantique.fr"**:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@imt-atlantique.fr
git config --global core.editor gedit
git config --global merge.tool meld
git config --global diff.tool meld
git config --global color.ui true
git config --global push.default simple
git config --global credential.helper cache
git config --global credential.helper 'cache --timeout=3600'
```

Later, if you use your personal machine for working on the project, you will have to configure Git in the same way (at least, for user.name and user.email).

Each command (in order) does the following:

- Sets the user name that will appear in the commit logs. This is needed in order to know who was the author of a commit.
- Sets the mail address that appears in the logs. This is useful as different people can have the same first and last name but they will have a different email address.
- Sets the editor that git will open by default.
- Sets the program that git will launch when using the `git difftool` command. You can use this to see the differences between files, including the version in the working directory and any previously committed version.
- Sets the program that git will launch when using the command `git mergetool`. You can use this command to resolve conflicts. (More on this shortly.)
- Makes git use a pretty color output by default.
- Makes git push the current branch to the branch with the same name
- (the last two lines) Caches the password for one hour (timeout=3600 seconds).

Exercise 3

You will now configure the environment for the other member of the `Y` subgroup.

To Do

Logout the user that was logged in during the previous exercise. Login with the account of the other member of the pair and repeat the directory creation, cloning and configuration steps of Exercises 1 and 2.

Who is logged in?

YOU ARE NOW LOGGED AS USER B

Exercise 4

In the `sgY/gitsandbox` directory, you can find a file named `lines_count.py`. This is a simple Python script that counts the number of lines in a file.

To Do

Modify the description string of the `ArgumentParser` at line 16 to make it clearer.

Use the command `git status` to verify that git has detected your changes.

It is always an excellent idea to use the `git status` command whenever you want to know what the status of your working copy is. If you read carefully the output of this command, you will often find useful hints on what you can/should do. (At the very least you can use some of the terms as a starting point for a search with your favorite search engine.)

To Do

Whenever you want git to track a new file, you must explicitly ask git to track its changes. To test this functionality, use the following command (in a terminal) to download another python script:

```
wget https://formations.telecom-bretagne.eu/r328/gitsandbox/lower.py
```

To ask git to add a new file, use the following command:

```
git add lower.py
```

At this point `git status` should show that a new file has been added to the "the changes to be committed," in other words the file has been "staged." Note that "staged" changes are not yet saved to the git version control. As it previously did, git is also letting you know that there are changes not yet staged for commit (the `lines_count.py` file).

To Do

To "stage" this file (i.e., to tell git that you want the current state of that file to be saved for a commit) use the following command:

```
git add lines_count.py
```

To Do

Now that you have added all the files to the index, you can use the following command to actually save these files to the revision history. Don't forget to add a comment using the `-m` option (replace "your comment" with something more descriptive):

```
git commit -m 'your comment'
```

To Do

If you now use the `git log` command, you should see your last commit at the top of the revision history. Finally, push your modifications to the remote git server:

```
git push
```

Exercise 5

In this exercise you will learn how to resolve a modification conflict. A conflict happens when two users modify the same section of a given file.

You will first generate a conflict and the resolve it. **Make sure that you follow the instructions below very carefully. Do not deviate from them at all!**

> **To Do**
> First of all, logout the user that was loged in during the previous exercise. Login with the account of the other member of the pair (the one who did Exercise 2).

Who is logged in?

YOU ARE NOW LOGGED AS USER A

> **To Do**
> Modify again the description string of the `ArgumentParser` at line 16 to make it clearer, but in a different way.

> **To Do**
> Under this session, add another file as well. For that, download a third python script:
>
> ```
> wget https://formations.telecom-bretagne.eu/r328/gitsandbox/upper.py
> ```
>
> Add your modifications to the index and commit it to the local repository:
>
> ```
> git add lines_count.py upper.py
> git commit -m 'Another set of modifications'
> ```

> **To Do**
> Now try to upload your changes to the server:
>
> ```
> git push
> ```

Your push is rejected because your colleague has pushed some work on the remote branch and you did not integrate this work yet.

> **To Do**
> Now try to pull
>
> ```
> git pull
> ```

Congratulations! You produced your first conflict :-)

> **To Do**
> Open `lines_count.py` with a text editor to look at how git amended the file to show you where the conflict happened.
>
> At this stage, you can directly edit the source file to resolve the conflict. But you can also take advantage of a more user-friendly graphical conflict resolution software. To launch it type:

```
git mergetool
```

Use meld to resolve your conflict (i.e., decide which is the correct version). On startup, meld displays a windows with three editors. The editor on the left is the local version of the conflicting file. The editor on the right is the remote version of the conflicting file. The editor at the middle is the common ancestor of the file (before conflict occurs). Use the arrows to decide which version of the modification is the right one.

Then push your modifications using the `git add`, `git commit` and `git push` commands.

Twisted Framework Introduction

The goal of the following exercises is to familiarize yourself with Python and the Twisted framework. After successfully completing this programming assignment you are expected to be able to:

- use the Python Twisted framework to write simple client and server programs using the UDP protocol and timers;
- make your code interact with an existing User Interface;
- describe what asynchronous programming is and how a simple Twisted program works.

Important

These exercises have many points in common with the c2w application. Make sure you carefully read the whole text. You should refer back to these exercises whenever you have doubts during the implementation of the c2w protocol.

The most important elements of these exercises are summarized at the very end of this text. Make sure you read that part and that you fully understand each point, as you will run into exactly the same concepts when implementing the c2w protocol.

Configure your Editor

Python uses indentation in the code to delimit functions, loops, etc. Do not mix tab characters and spaces in the same source file. Make sure your editor is configured to insert four spaces when you press the tab key.

For *gedit* users: go to the Edit menu and then to the Preferences menu. Then select the "Editor" tab, write 4 next to "Tab Width" and make sure that the "Insert spaces instead of tabs" check-box is checked.

Sibyl: A Simple Client-Server Application

Congratulations! You are hired in a 2500-years-old company named Sibyl, Inc. It is a strategy consulting firm with an extensive expertise in giving ambiguous ("sibylline") advice to Fortune-500 companies. Sibyl is now an online service where clients send questions to the Sibyl server, which always replies with weird responses. You are the network guy. Other workers at Sibyl, Inc. focus on designing outstanding User Interface and creating new cryptic advices for the clients. The Sibyl application is written in Python, and it makes extensive use of the Twisted framework. The client and the server can communicate using UDP or TCP.

Your mission is to provide the network modules for the client and the server. In both cases, your module must interact with the other modules of the application. These modules are installed in `~stockrsm`. The documentation of the Application Programming Interface (API) of these modules is available at https://formations.telecom-bretagne.eu/r328/doc/.

To execute the application on your computer, you have to open two terminals, one to execute the server and the other for the client. You can also test the server of another (sub)group by using the `-m` command line option of your client to specify the address of the server. Please refer to Appendix A for information on how to execute a Python program.

Exercise 6

The Sibyl protocol is a simple protocol. You can find its specification right after these exercises. Your first task is to implement the text version of the protocol using UDP. (Hint: you might want to check the `time` Python library (see https://docs.python.org/3.4/library/time.html) to obtain the number of seconds since January 1st 1970. You can search for the "epoch" keyword.)

To Do

Implement the text version of the UDP protocol in `sibyl_client_udp_text_protocol.py` and `sibyl_server_udp_text_protocol.py` (in the `~/r328/sgY/sibyl/protocol` directory). Execute the programs `sibyl_udp_text_server.py` and `sibyl_udp_text_client.py` to start the corresponding server and client (in the `~/r328/sgY/sibyl/scripts` directory).

Once you have completed the implementation you can *commit* your changes and propagate them to the git server by typing the following commands inside the `~/r328/` directory:

```
git commit -am 'exercise 1 done' git push redmine-tb master
```

Exercise 7

Your next task is to implement the binary version of the protocol. Again, refer to the protocol specification for the details. (Hint: you can find some pointers on how to handle binary data with Python in Appendix B.)

To Do

Implement the binary version of the UDP protocol in the `sibyl_client_udp_bin_protocol.py` and `sibyl_server_udp_bin_protocol.py`. Execute the programs `sibyl_udp_bin_server.py` and `sibyl_udp_bin_client.py` to start the corresponding server and client. Commit your changes.

Exercise 8

So far the Sibyl server responds immediately to each query. Modify the server protocol so that it waits a certain time before replying. More precisely, the Sibyl waits for x seconds before responding to a request, where

$$x = \lceil \ln(l) \rceil$$

where l is the number of characters in the request and $\lceil z \rceil$ is the smallest integer greater than or equal to z (i.e., the ceiling of z).

To Do

Copy your version of the UDP binary protocol with the command (inside the folder `~/r328/sgY/sibyl/protocol`). Type the following command on a single line:

```
cp sibyl_server_udp_bin_protocol.py sibyl_server_timer_udp_bin_protocol.py
```

Modify the `sibyl_server_timer_udp_bin_protocol.py` file so that the response message is not sent immediately. Rename the protocol class `SibylServerTimerUdpBinProtocol`.

Hint: Twisted has a function that you can use to solve this problem easily. You will need to access the "reactor", which can be done by adding the following line at the beginning of the file:

```
from twisted.internet import reactor
```

Execute the program `sibyl_timer_udp_bin_server.py` to start the corresponding server. Once you're done, you must add the new file to git before committing. Type the following command (inside the folder `~/r328/sgY/sibyl/protocol`):

```
git add sibyl_server_timer_udp_bin_protocol.py
git commit -am 'completed the server_time_udp'
git push redmine-tb master
```

Exercise 9 (Bonus Question)

Suppose now your fellows at Sibyl, Inc. failed in implementing an efficient Sibyl brain module. Typically, the `generateResponse` methods takes up to one minute to return the awaited advice. How does such degraded Sibyl brain affect the behavior of the server?

To Do

Find a solution for this problem. Discuss it with one of the teachers before you try to implement it.

Most Important Concepts/Elements

At the end of these exercises you should fully understand (and remember) that **Twisted is an event-based framework**, which allows the implementation of client and server programs as well as functions that will run at a given time in the future.

Appendix A How to Execute a Python Program (Script)

There are two ways to run a python program (or script, both terms are equivalent in the context of RES 209):

1. You can use the command `python3.4 script.py` where script.py is the name of the script file.
2. You can put the following in the *first* line of your script: `#!/bin/env python3.4` and then mark the file as executable by using the command `chmod u+x script.py` then you can launch the script by typing `./script.py` in a shell.

Note that, in both cases, *it is essential to use* `python3.4` *and not just* `python`. By default, `python` corresponds to version 2.7 of the Python interpreter.

Appendix B Handling Binary Strings in Python

Starting with version 3, Python has an explicit type for binary data: `bytes`. Like `strings`, `bytes` are an immutable type: in other words, you can only initialize a variable of type `bytes`, you cannot modify it after.

The `struct` module provides functions that can be used to handle binary data. You can find the complete documentation at https://docs.python.org/3.4/library/struct.html. You can use the `pack` or the `pack_into` functions to build binary data. The former (`pack`) returns an object of type `bytes` containing the result, while the latter (`pack_into`) takes as an input argument a reference to a buffer where to store the result. You can use objects of type `bytearray` to build such buffers.

Say that you want to build a binary string with seven bytes, where the first byte has the value 3, the second 5, the third and the fourth 456, and the fifth, sixth and seventh contain the ASCII code for the letter 'a' (97), 'b' (98) and 'c' (99) respectively.

As mentioned above, one option is to use the `pack` function:

```
import struct

buf = struct.pack('BBH3s', 3, 5, 456, 'abc'.encode('utf-8'))
```

The first parameter of the `pack` function is the "format string" that describes the values that are to be packed into the buffer. For example, the '`B`' value corresponds to an unsigned integer coded on one byte, the '`H`' value corresponds to an unsigned char coded on two bytes and the '`s`' value corresponds to a string. The '3' before the '`s`' is to indicate the length of the string. Please read the documentation of the `struct` module for all the details about the format string. Note that, in Python 3, you have to convert a string (`'abc'`) to a sequence of bytes by using the `encode` method.[1] The last four parameters give the values that must be packed into the buffer.

The other option is to use `pack_into`:

```
import struct

# create a 7-byte buffer
buf = bytearray(7)
# put data into the buffer
struct.pack_into('BBH3s', buf, 0, 3, 5, 456, 'abc'.encode('utf-8'))
```

In this case, you first have to create a buffer large enough to contain the data. This is accomplished by calling the constructor of the `bytearray` class with the desired length (in bytes) as its only parameter, resulting in a buffer of 7 zeros. This is needed because `pack_into` needs a mutable buffer that can contain all the parameters to be packed.

The first parameter of the `pack_into` function is the "format string," which is the same as in `pack`. The second parameter (`buf`) is the name of the buffer where to pack the values. The third parameter (0) is the offset with respect to the beginning of the buffer. This can be used to build a buffer with multiple calls to `pack_into`. The last four parameters give the values that must be packed into the buffer.

After the call to `struct.pack_into`, `buf` contains the following bytes (in hexadecimal notation): `0x0305c801616263`. There is nothing special about the first two bytes, but you should be surprised by the third and fourth, whose value is `0xc801`, corresponding to 51201 in decimal notation. This is because, by default, Python uses the so-called little-endian format, where the least significant byte is placed last. You can change the format to big-endian by using the `'>BBH3s'` format string. Again, please read the documentation of the `struct` module for all the details about byte ordering.

In both examples, `buf` contains the same values. The difference is that, with the first method (`pack`), `buf` is immutable, i.e., it cannot be modified after the call to `pack`. While, with the second method ((pack_into)), it is possible to modify `buf` with other calls to `pack_into`.

If you want to build a binary sequence by calling more than once `pack`, you need to concatenate the different pieces by using the add operator (`+`). If you want to use `pack_into`, you can create a big enough buffer at the beginning and then use the offset parameter of `pack_into`.

[1] In this case we have used the utf-8 encoding, which is a superset of the ASCII encoding, i.e., all ASCII printable characters have the same encoding in utf-8 and ASCII. The advantage of utf-8 is that it supports multiple languages natively. If in doubt, use utf-8.

A. Blanc
Telecom Bretagne
June 2014

The Text and Binary Sibyl Protocol

Abstract

The Sibyl protocol satisfies all the communication requirements of the Sibyl application. Both the text and binary version of the protocol have only one message, which is used by the client to send a request and by the server to send the response.

Table of Contents

1. Introduction

The Sibyl application is a simple client-server application: the client sends a request (question) to the server that responds with a reply. The format of the question is the same as the reply. As clients always send requests and the server always sends response, it is always possible to determine whether a message is a requests or a response.

For the sake of simplicity, the message format is the same for the text and binary version.

Clients can use either UDP or TCP to exchange messages with a server. When sending a reply, the server MUST use the same layer four protocol used by the client for the corresponding request.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Message Format

All messages have the same format:

TIME: <SPACE> TEXT < CRLF >

Where SPACE is a single white space (ASCII decimal code 32) CR represents the ASCII character "Carriage Return" (ASCII decimal code 13) and LF represents the ASCII character "Line Feed" (ASCII decimal code 10).

TIME

For requests (i.e., messages sent by a client), this is the time at which the request was sent, expressed as the number of seconds since January 1st, 1970. (This is often referred to as the beginning of the "epoch" in Unix-like systems.) For responses, (i.e., a message from the server to the client), TIME field MUST be the same as the TIME field of the corresponding request.

Text

The text of the question, for requests, and the text of the reply for responses.

The server MUST always reply to a request. Clients MUST send at most one request every second.

In every message the TIME value MUST be followed immediately by a single colon, followed a single white space.

2.1. Example for the Text Protocol

At 3:35 pm on March 3rd, 2010, a client sends the following question to the server: "A year from now, is the price of oil going to be higher or lower than today?" (recall that CR represents the ASCII character "Carriage Return" (13) and LF represents the ASCII character "Line Feed" (10)):

1267626900: A year from now, is the price of oil going to be higher or lower than today?CRLF

The server replies with "Yes:"

1267626900: YesCRLF

3. Binary Message Format

The binary version has a single message as well, shown in figure Figure 1.

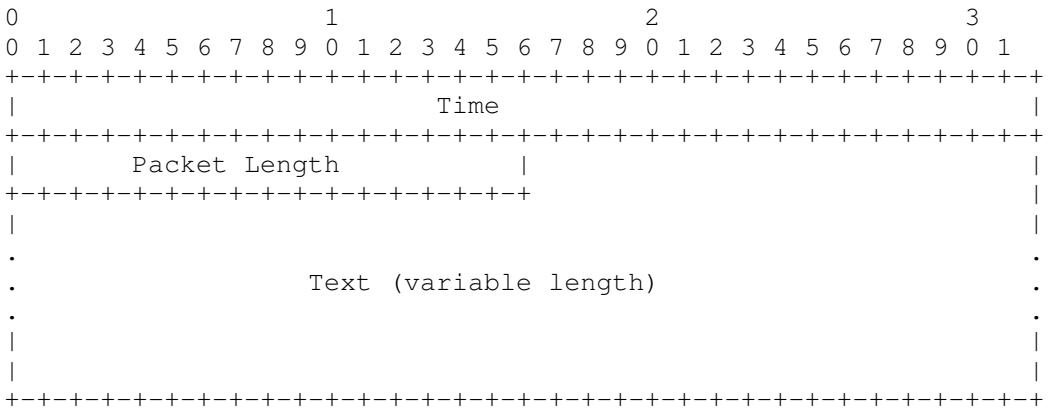


Figure 1

Time (32 bits)

For requests (i.e., messages sent by a client), this is the time at which the request was sent, expressed as the number of seconds since January 1st, 1970. (This is often referred to as the beginning of the "epoch" in Unix-like systems.) For responses, (i.e., a message from the server to the client), Time field MUST be the same as the Time field of the corresponding request.

Packet Length (16 bits)

The lengths, in bytes, of the whole message (i.e., header and text of the question/response).

Message Text (variable length)

The text of the question/response, encoded in ASCII.

As in the case of the text version of the protocol, The server MUST always reply to a request. Clients MUST send at most one request every second

3.1. Example for the Binary Protocol

The example is the same as the one for the text version. At 3:35 pm on March 3rd, 2010, a client sends the following question to the server: "A year from now, is the price of oil going to be higher or lower than today?" The values of each field are hexadecimal, with the decimal notation in parenthesis fir the Time and Packet Length.

```
Time = 4b 8e 73 94 (1267626900)
Packet Length = 52 (82)
Message Text = 41 20 79 65 61 72 20 66 72 6f 6d 20 6e 6f
               77 2c 20 69 73 20 74 68 65 20 70 72 69 63
               65 20 6f 66 20 6f 69 6c 20 67 6f 69 6e 67
               20 74 6f 20 62 65 20 68 69 67 68 65 72 20
               6f 72 20 6c 6f 77 65 72 20 74 68 61 6e 20
               74 6f 64 61 79 3f
```

The server replies with "Yes:"

```
Time = 4b 8e 73 94 (1267626900)
Packet Length = 09 (9)
Message Text = 59 65 73
```

4. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

Author's Address

Alberto Blanc
Telecom Bretagne
Cesson Sevigne
France

Phone: +33 02 99 12 24 25
Email: alberto.blanc@imt-atlantique.fr

RES 209 TP-2&3 Twisted/TCP Programming

Spring 2017

Introduction

The goal of these exercises is to familiarize yourself with the way Twisted deals with TCP connections, message framing and the testing framework that you will use for the c2w project. After successfully completing this programming assignment you are expected to be able to:

- use the Python Twisted framework to write simple client and server programs using the TCP protocol;
- implement a message framing algorithm;
- use the `trial` testing framework.

Preamble

Please re-compose the same pairs (also known as subgroup `sgY`) as the ones you composed last week.

Let us assume that your names are Camille and Dominique respectively. Without loss of generality, Camille was the one who logged in last week during the first assignment, that is the first assignment was done on Camille's workspace. Now, you should be able to identify who is Camille and who is Dominique.

Today, for this assignment, we would like Dominique to log in and work under its seesion, *i.e.*, clone the git repository, modify the code and commit the modifications.

Assignment Description

The Chief Technology Officer (CTO) at Sibyl, Inc. has become a little bit suspicious after inspecting your achievements during the first assignment. After all, you did not fully complete the assignment, did you? The CTO decided to go a bit deeper into network programming and quickly discovered the UDP vs. TCP debate. For some reason, the CTO is convinced that TCP is the best choice.

Exercise 1

We are afraid you have no other choice than implementing the binary Sibyl protocol over TCP. You will still use the Twisted framework, specifically the `Protocol` class. The APIs of the Sibyl user interface and Sibyl brain are unchanged.

Fortunately, since the Sibyl protocol remains the same, most of the code you wrote during the first assignment can be re-used. At this point, we cannot avoid suggesting that you extract from your code the lines that are used multiple times, typically the code for building (and decoding) messages. You should create you own Python file (i.e., a "module") with these shared functions (and/or classes). Then you can just use the appropriate `import` statement in the files with the protocol implementation.

To Do

Implement the binary version of the TCP protocol in `sibyl_client_tcp_bin_protocol.py` and `sibyl_server_tcp_bin_protocol.py` (in the `~/r328/sgY/sibyl/protocol` directory). Execute the programs `sibyl_tcp_bin_server.py` and `sibyl_tcp_bin_client.py` to start the corresponding server and client (in the `~/r328/sgY/sibyl/scripts` directory).

Exercise 2

Beginners often forget that TCP offers a reliable byte-stream service, delivering all the bytes in the correct order, but it does not guarantee that *all* the bytes sent with a single "write" call are going to be received with a single "receive" call. In other words, TCP does neither preserve message boundaries nor it offers any facility that can be used to signal message boundaries.

In the case of the Sibyl binary protocol, it is however possible to use the message length field in order to distinguish different messages. We refer to the process of identifying messages as "*message framing*" (framing for short).

To Do

Implement the framing directly into `sibyl_client_tcp_bin_protocol.py` and `sibyl_server_tcp_bin_protocol.py` (in the

`~/r328/sgY/sibyl/protocol` directory).

Execute the programs `sibyl_tcp_bin_server.py` and `sibyl_tcp_bin_client.py` to start the corresponding server and client (in the `~/r328/sgY/sibyl/scripts` directory).

Commit.

Exercise 3

If you are reading these lines, it means you think that you have successfully completed the previous exercise. How can you be so sure?

You are at a stage of the implementation where you should understand that it is not enough to just open the User Interface and play with it. Indeed, you implemented the framing to cope with scenarios where message transmission over TCP is not as simple as a pair of send and receive calls. Unfortunately, since you executed the server and the client on the same machine or, at most, on two computer connected by a Local Area Network (LAN), you executed your programs under the best possible circumstances so that, almost certainly, both send and receive calls perfectly match message boundaries.

So how can you be sure that your framing actually works?

To answer this question, developers rely on *testing*. The main idea is that an independent program is designed to interact with your code and see how it behaves when pushed to its limits. In our case, the independent program is typically a fake client that sends messages to your server.

During this assignment and then throughout this module, you will use the `trial` library, which is a unit testing framework designed to test programs that use Twisted. The `trial` framework contains a lot of facilities to ease the development of tests. You do not need to understand how `trial` works, except that it makes use of *scenario files*. For this assignment, you will use existing scenario files so that you just have to execute test.

If everything is fine, the last line displayed by the execution of `trial` is: **PASSED**. If (at least) one test fails, the last line displayed is: **FAILED**. In the latter case, it is your responsibility to detect what is wrong with your implementation. Please read carefully the output of the execution of trial to identify which test function failed and consequently which modification you should bring to your code. You might also want to add a few print statements to your code in order to better understand what is going on (for example by printing the data you receive each time).

To Do

In the `~/r328/sgY/sibyl/scripts` directory.

Execute:

`trial_sibyl_client_no_framing.py`

to test your client. Revise your code if necessary.

Execute:

`trial_sibyl_server_no_framing.py`

to test your server. Revise your code if necessary.

These tests only verify that your client (respectively server) generate messages that conform to the specification. This is done by comparing the messages sent by your code with those stored in the scenario file.

To test the framing implementation you need to execute another set of tests. Note that the following command execute more than one test. Each test deals with a specific special case, it is up to you to figure which one.

Execute:

`trial_sibyl_server_framing.py`

to test the framing implementation in the server. Revise your code if necessary.

Commit.

RES 209 Project Overview
Spring 2017

Context and Motivations

Computer networks are complex systems: even though each single element can be fairly simple, the interactions of many different elements lead to a significantly complex system. These interactions are typically controlled by *communication protocols*. These protocols play a fundamental role in computer networks, as they specify the format of the messages exchanged between different nodes and the rules governing these exchanges. Without them computer networks cannot function.

The goal for the RES 209 project is to let the students experience first-hand what a protocol is, how it can be defined and how it can be implemented. Protocols can be extremely complicated and require up to hundreds of pages to describe them. In order to have a protocol which can be easily described and implemented, the RES 209 project is concerned with a simple chat application, called c2w or "chat while you watch." The idea being that users can connect to a server offering several video streams, select a single video stream and chat with the other users watching the same stream.

Project Overview

Students, working in groups, must first submit a document specifying a protocol capable of satisfying all the requirements of the application (see the "Functional Requirements" document). Each student, using the hotcrp platform, will have to review and evaluate two protocol specifications prepared by another group. Each group must prepare a short presentation of its protocol specification but only the groups with the best rated documents will actually present their work. Based on these presentations, the teaching staff and the students will either select one of the proposed specifications or combine multiple specifications to produce a single protocol. The teaching staff will write the final version of the protocol specification and post it on Moodle. Each group can either implement this common specification or its own, subject to approval by the teaching staff.

Students will use Python and the Twisted framework, which have been introduced in the first two "TP," to implement the selected protocol specification. In order to simplify the implementation phase, students are expected to implement only the communication protocol: they will receive the code for the Graphical User Interface (GUI) of the client as well as a skeleton for the server program.

In order to monitor the progress of each group and in order to ensure that all groups work on the project throughout the time allocated to it, each group will have to periodically test its own code, using automated tests prepared by the teaching staff. Each group must also periodically submit its code, as detailed in the next section.

Deliverables/Assignments, Deadlines and Tools

During the first phase of the project, each student has to answer a few questions about the sample specifications (FindIPAddress). Then each group must prepare its own protocol specification. During RES 209 PC-1 and PC-2, students will work in groups and start preparing their own protocol specification. Each group *must* write its own protocol specification document in xml format and use the `xml2rfc` tool to generate a human-readable document. This tool is available on all the computers at Telecom Bretagne. It is also freely available for download on the IETF webpage.

In the second phase of the project, each group must implement the selected protocol specification. The version prepared by the teaching staff will be posted on Moodle the day of TP-4 (Apr. 25).

Deliverables and Assignments

The following sections contain a description of all the deliverables that each group must prepare for the RES 209 project. You can submit documents either in French or in English.

Assignment: FindIPAddress Protocol Moodle Quiz

In order to familiarize yourself with protocol specifications, you must carefully read the sample (and simple) FindIPAddress protocol specifications, both in binary and text-based form as well as the corresponding functional requirements document. The goal of this protocol is, like DNS, to find the IP address corresponding to a name. This protocol is extremely simple and is meant only as a trivial example. It is an application layer protocol, like the c2w protocol, therefore it can be either a binary or a text-based protocol.

Each student must answer all the questions about the two versions of the FindIPAddress protocol before April 4 at 9 am.

The quiz is graded automatically and you can take the quiz as many times as you want before the deadline.

Assignment: c2w Protocol First Message Format Draft

Based on the functional requirement document, each group must decide the format of each message, i.e., how to organize the information exchanged (number of fields, their lengths and content).

By the end of PC 2, each group must commit on the Git server a first draft of the protocol specification, in one of the following format (and only one):

- a text file;
- an xml file that can be compiled with `xml2rfc`;
- a pdf file;
- a Jpeg file containing the picture of hand-written notes.

This first draft must *at least* contain the format of most messages, the general overview of the reliability mechanism, and at least one or two detailed examples, including the values of all the fields in each message. This being a draft, the final version can be different.

The filename must be `draft-spec-r328-s17-gX` and it must be in the `spec` directory.

Protocol Specification

This document describes in detail the proposed protocol specification of each group. It must clearly describe the format and the content of all the messages exchanged between the client and the server and when these messages are to be used. It must also contain at least one example where a client connects to the server, asks for the user and movie lists, receives them, joins a movie room and sends a chat message. This example must specify the values of *all* the fields of every message used.

Remember that the idea is that one could give this document to two different groups, one in charge of implementing the server and another one in charge of the client, and these two groups, only by following this document, must be able to produce a server and a client that can work correctly. All this without being able to ask questions to the other group or to the authors of the document. In other words this document must be clear and contain all the information needed.

Students are expected to use the `xml2rfc` program to prepare this document. This is because it is much easier to review and compare the proposals of different groups if they are formatted in the same way. See section 2.3.1 below for more details. Students are strongly encouraged to use the `protocol-specification-template.xml` file as a starting point. This file is on Moodle in the directory with the other material for the "PCs" and "TPs."

In order to facilitate printing, all the lines in the xml file must have at most 72 characters. If this is not the case `xml2rfc` does print a warning message.

Each group must commit the final version of the xml file on the Git server by April 7 at 9:00 am. The file must be in the `spec` folder and must be named `spec-r328-s17-gX`, where `X` is the group number.

Each group must also upload in the same folder a pdf file containing the protocol specification on the HotCRP platform before April 7 at 9:00 am. (See section 2.3.3 below.)

Assignment: Protocol Specification Review/Evaluation

Each student must review and evaluate two protocol specifications written by another group. Each student must complete the reviews using the HotCRP platform (see section 2.3.3 below) before 9:00 am on April 11.

Protocol Specification Presentation

Each group must prepare a few slides illustrating its proposal. As all the students and the teachers are already familiar with the RES 209 project, these slides must present only the proposal of each group, without an introduction to the c2w application: 4 or 5 slides are more than enough (without counting a title slide). No presentation can have more than 8 slides. You can use whatever program you want to prepare the slides (Power-Point, Libre Office, Latex, etc.).

Each group must commit a pdf file (no other format is accepted) on the Git server before April 11 at 9:00 am. The filename must be `spec-pres-r328-s17-gX` and must be in the `spec` directory of the Git repository.

Python Code

Each group must periodically submit the Python code implementing the common protocol specification. While writing efficient and optimized Python code is not the objective of the RES 209 project, it is important that your code is as clear as possible. You should always include comments explaining what each function is doing. This is also going to be useful during the final evaluation of the project, when the teaching staff can ask any member of the group to find, explain and comment the code that implements a given functionality (e.g., "Show the function that, in the server, handles forwarding a chat message to all the other clients located in the same chat room").

At the beginning of TP-5 and TP-6, each group will be asked to show that the code committed to the Git server can pass a series of tests. These tests are meant to verify some of the functionalities of the server and of the client. Passing these tests is not a sufficient condition for a successful project. Theses tests are meant to make sure that all the groups work on the implementation throughout the semester and not just shortly before the final evaluation. These tests are also meant to help you verifying that certain portions of your code are correct. You will receive more details about the tests at the beginning of TP-4.

Note that your goal is to write code that implements correctly the selected protocol specification and not to write code that can simply pass the tests.[1]

Deadlines

The following are all the deadlines for the RES 209 project. Note that these are hard deadlines and will not be modified.

FindIPAddress Protocol Moodle Quiz (April 4, 9 am): all students must complete the FindIPAddress Protocol Moodle Quiz before this time.

c2w Draft Protocol Specification (April 4, end of class): each group must commit to the Git server a file with a first draft of the protocol specification. The filename must be draft-spec-r328-s17-gX and it must be in the `spec` directory (see section 2.3.2 below).

Protocol Specification (April 7, 9:00 am) (Git): each group must commit the xml file to the corresponding Git repository (see section 2.3.2 below). The name of the xml file must be `spec-r328-s17-gX` and it must be in the `spec` directory.

Protocol Specification (April 7, 9:00 am) (HotCRP): each group must submit a pdf file containing the proposed protocol specification on HotCRP (see section 2.3.3 below).

Protocol Specification Reviews on HotCRP (April 11, 9:00 am): each student must submit the reviews assigned on HotCRP.

Protocol Specification Presentation (April 11, 9:00 am): each group must commit a short presentation (slides) *in pdf format* to the Git repository in the `spec` folder. The filename must be `spec-pres-r328-s17-gX`. Note that only pdf files will be accepted for the presentation.

Intermediate Versions of the Python Code: (TP-5 (May 2, 9 am) and TP-6 (May 16, 9 am)) each group must commit the latest version of the code by 9 am the day of TP-5 and TP-6. This code will be used at the beginning of the TP to verify whether it passes the available tests. More details about these tests will be given during TP-4.

Final Version of the Python Code (May 30, 9 am): each group must commit the final version of the Python code. This version will be used for the project evaluation.

Tools

xml2rfc

The `xml2rfc` program converts an input xml file into a text file or into an html file. To convert an xml file into a text file you can type `xml2rfc filename.xml` in a shell (terminal), if there are no errors `xml2rfc` will create a text file called `filename.txt` in the same directory. To create an html file you can use the command `xml2rfc filename.xml filename.html`. Obviously, in both cases, you have to replace `filename` with the name of an existing xml file.

You can find a sample file (`sample-protocol-specification.xml`) on Moodle, in the `td-tp` directory in the supporting material section for RES 209. You are strongly encouraged to verify that you can indeed convert this xml file into a text (and/or html file). Then you should make a local copy of this file and you should start modifying it. You should compile the file fairly often as `xml2rfc` can be extremely

[1] As the tests basically compare the messages generated by your server (and client) to the "correct" messages in a few specific cases, it is possible to "cheat" by writing code that will only generate the few messages that are used in the tests. Obviously this is not what you are supposed to do.

picky about the format of the xml file and the error messages are not always easy to understand. If you compile often, you will know that the error was introduced in the lines that were added/modified since you last compiled.

This tool is used to produce the official IETF documents, which are always in English and need to use only printable ASCII characters. Because of this `xml2rfc` does not support accented letters and other diacritical marks when generating text files. In this case, it simply removes the diacritical marks, for example all accents are removed. It does, however, preserves these marks when generating html output. Therefore, if you are writing your document in French, you can generate the html version as well. Please note that you must commit *both* the text and the html version in this case and not just the html version.

Redmine and Git Repositories

Redmine is a web-based project management and bug-tracking tool. The user interface is fairly simple and intuitive. Redmine allows users to submit, monitor and modify bug reports (also called "issues"). It also offers a Wiki and file and document hosting/sharing. It is up to each group to decide whether to use these features or not. More information on Redmine is available by clicking on the "Help" link on the top right-hand corner of all the Redmine pages or at `https://redmine-df.telecom-bretagne.eu`.

You must use the Redmine instance installed `https://redmine-df.telecom-bretagne.eu` for the RES 209 project. You can login into Redmine using your Telecom Bretagne user-name and password. Once you login you should see a page with a link to the project corresponding to your group.

Redmine also offers a web-interface to view the Git repository associated with each project. You can access this interface by clicking on the corresponding tab. You can use this interface to see the differences between revisions.

Each group has a dedicated Git repository, which is hosted on the same machine. You can use the url `https://redmine-df.telecom-bretagne.eu/git/r328-s17-gX` to access this repository, where, as usual, `X` is the group number. For example, if you are using one of the Linux machines at Telecom Bretagne, you can use the following command to checkout the repository corresponding to your group (type this command on a *single* line):

```
git clone -o redmine-tb https://redmine-df.telecom-bretagne.eu/git/r328-s17-gX r328
```

HotCRP

HotCRP is a system used for peer reviews for scientific conferences and journals. Such a system can also be used in a context where different people need to review/evaluate documents anonymously, in the sense that the the authors of the document will see the reviews but will not know who wrote them. (Note that the teaching staff *does* know who wrote each review.)

You will receive an email with the HotCRP account information (username, password, url), after the groups for the project will have been finalized. You can change the password used for HotCRP but it is extremely important that you do not use the same password used on other Telecom Bretagne computers. Unfortunately HotCRP uses clear-text password in certain urls, therefore you should not use on HotCRP the password you use any other computer.

Each group must submit a pdf file with the protocol specification on the HotCRP site. Only pdf files are accepted. You can use the following command to convert the text file generated by `xml2rfc` to a pdf file:

```
a2ps -R --columns=1 --borders=0 -B -P pdf spec-r328-s17-gX.txt
```

where `spec-r328-s17-gX.txt` is the file generated by `xml2rfc`.

Note that you can submit the pdf file multiple times before the deadline. Each submission replaces the previous one.

After the protocol specification submission deadline each student will receive another email with the links to the documents to review.

Common Code

In order to simplify and limit the scope of the implementation part of the RES 209 project, you are going to use existing code for the Graphical User Interface (GUI) of the client and for the server aw well. When you launch the scripts in the Git repository, this code is loaded automatically.

You can find the corresponding documentation at https://formations.telecom-bretagne.eu/r328/doc/.

Rules: Class Attendance, Evaluations, Plagiarism, Exams

You must carefully read and follow the rules explained below, no exceptions will be made. Failure to comply with any of these can result in a loss of points for the project grade and/or the final grade.

Class Attendance

As with other projects at Telecom Bretagne, given that certain sessions are fundamental to the correct execution of the project, all students must be present for the following activities:

Apr. 04	whole day	RES 209 PC-1 and PC-2	protocol specification
Apr. 11	afternoon	RES 209 C3	specification presentation, discussion and selection
Apr. 25	afternoon	TP-4	implementation (1/3)
May 2	afternoon	TP-5	implementation (2/3)
May 16	afternoon	TP-6	implementation (3/3)
May 30	whole day	TP-7	project evaluation

If you are absent for any of these activities, you must contact the teacher in charge of RES 209 and you must provide to the school administration ("scolarité") a doctor's certificate, otherwise you absence will not be considered as excused (justified). Only the school administration can excuse your absences, even though you must always contact the teacher in charge of RES 209 as well.

Furthermore:

- For each unexcused absence to one of the activities listed above your grade will be reduced by 5%. Note that PC i ($i = 1, 2$) is considered as a single activity. For example, if you are absent for PC-1 and PC-2 you have been absent for *two* activities even though they take place on the same day. Each TP counts as a single activity.
- Starting from three unexcused absences the final grade for the project will be 0.
- Starting from four excused absences (illness, etc.) the grade of the project will be ignored. Therefore the grade for the whole UV will be based only on the final exam and the "contrôles continus" of the other modules in the UV.

Evaluations

There are two main evaluations for the RES 209 project, one concerning the protocol specifications and one concerning the final version of the Python code. Even though the RES 209 project is a group project, the grades are given individually to each student.

In each case there are going to be several individual questions, which play a crucial role in determining the grade of each student. It is therefore possible that people in the same group will have significantly different grades based on how they answer questions.

Protocol Specification and Presentation

The grade for the protocol specification part is based on several elements:

- the protocol specification document itself (completeness, clarity);
- the presentation of the protocol specification;
- the answers to the questions following the presentation.

Only a few groups will be asked to present their protocol specification. The teaching staff will communicate the identity of these groups only at the beginning of Course 3. Therefore all groups must submit a presentation and be ready to present.

Each group will have between 10 and 15 minutes to present their proposal (the exact time depends on how many groups there will be in each classroom). Note that the teaching staff will choose the student(s) who is (are) going to present the specification document. Following the presentation other students and the teachers will ask questions. For each question the teacher will decide who is going to answer. Therefore all the members of a group must be capable of presenting the document and answer any question concerning it.

Final Project Evaluation

TP-7 will be dedicated to the final evaluation of the project, using the code committed to the Git repository.

Each group will have between 20 and 30 minutes to present their code and to answer questions. Teachers will also ask each group to show that their implementation works as expected in a few simple cases, for example:

- a user connects to the server and enters the main room;
- multiple users connect to the server;
- users exchange messages in the main room;
- users join a movie room;
- users exchange messages in the movie room;
- users leave the movie room and then the main room.

In each case the clients must display an updated user-list (based on whether they are located in the main room or in a movie room) as well as all the chat messages addressed to the room where they are currently located. Note that this list is not exhaustive and other use cases can be considered as well.

Following this demo, the teachers will ask a few specific questions about the code and the implementation in general. As in the case of the protocol specification, the teacher will select who is going to answer each question: all students must be able to answer any question about the code and the answer "I didn't write this part" is not acceptable. That being said, students are not expected to learn the code by heart. There will be some time to look at the code and corresponding comments (this is why it is a good idea to comment the code!).

These are just some of the possible questions, the teacher can ask any question related to the project, including, but not limited to, questions about the tests, the protocol specification, the difference between TCP and UDP implementation.

Plagiarism

Obviously each group must submit only what has been written/prepared by the members of the group. Under no circumstances you can use in your document passages coming from other sources. You do not need to and you must not consult any external source in order to write the protocol specification. All what you need is in the functional requirements document. If you have any question you must ask the teaching staff and not other students.

The teaching staff will compare all the submitted proposals using sophisticated programs that can detect different forms of plagiarism. Whenever a group is suspected of having copied parts of their document, if its members cannot show that this was not the case, their grade will be reduced (eventually to 0 for the most egregious cases).

While preparing the protocol specification the members of each group MUST NOT discuss, in any way, the protocol specification with other groups.

During the implementation phase groups can communicate with each other but copying and pasting code is always strictly prohibited. In this case as well, the teaching staff will use programs that detect plagiarism in the Python code.

Final Exam

The final exam will cover all the modules of the UV RES 201, RES 202 and RES 203 . If, for certain students, the difference between the grade of the RES 209 project and the grade of the final exam is large, they can be asked to pass an oral exam in order to determine the final grade of the UV.

The FindIpAddress system is a simple client-server system: clients can send queries to the server in order to find the IP address corresponding to a name. (Obviously this an extremely simplified version of the Domain Name System.) Clients know the IP address and the port number of one server instance; this information can be specified either in a configuration file or as an option on the command line. Clients send requests to the server specifying the name they are looking for. The server MUST respond with a message specifying either the IP address corresponding to the name requested by the client or that the name does not exist in the system,

If a server does not know the IP address of a certain name, it MAY contact one of more servers, forwarding their responses to the client. Clients can use either TCP or UDP to contact the server.

A. Blanc
Telecom Bretagne
June 2014

Sample Protocol Specification: FindIpAddress (Binary Version) FindIpAddress-bin

Abstract

Sample protocol specification for the binary version of the FineIpAddress protocol. The goal of this protocol is to allow a client to query a server to find the IP address corresponding to a given name. This is an overly-simplified protocol which is only meant as a simple example.

Table of Contents

1. Introduction

The goal of the FindIpAddress protocol is to allow clients to find the IP address that corresponds to a given name. Each client MUST know the IP address and corresponding port number of at least one instance of a FindIpAddress server.

For the sake of simplicity, the message format is the same for all the messages.

Clients can use either UDP or TCP to exchange messages with a server. When sending a reply, the server MUST use the same layer four protocol used by the client for the corresponding request.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Message Format

All messages have the same format, shown in figure Figure 1.

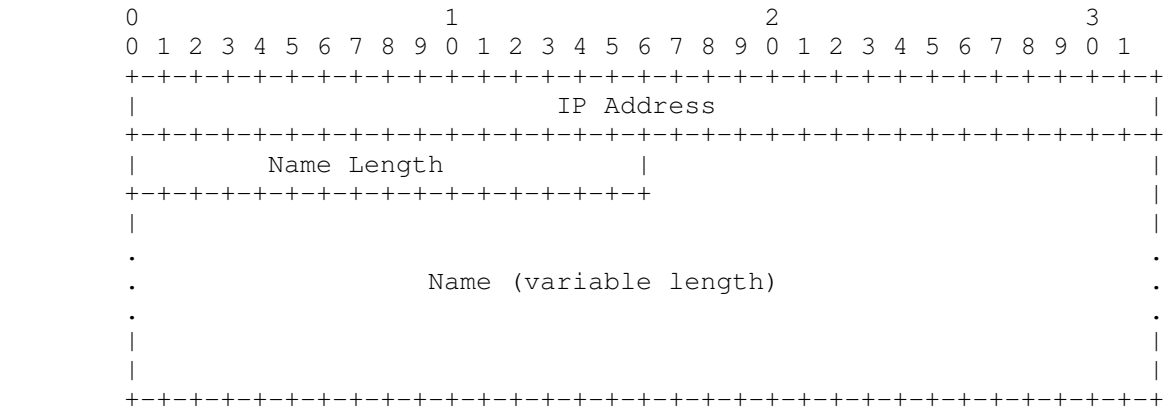


Figure 1

IP Address field (32 bits)

In a (positive) response message, this field contains the IP address corresponding to the name contained in the Name field. In a (negative) response, i.e., when the requested name does not exist, the server MUST set this field to 0. In a request message the client MUST set this field to 0

Name Length field (16 bits)

This field specifies the total length in bytes of the Name field. The sender MUST ensure that this field contains the correct value.

Name field (variable length)

This field contains the name that is the object of the request, encoded in ASCII. Both the client and the server MUST always specify the name in each message.

The server MUST always reply to a request, either with a positive response (in this case the IP Address field contains the IP address corresponding to the name requested by the client) or with a negative response (in this case the IP Address field MUST be set to 0).

3. Reliability

When using UDP, a client MAY resend a request for which it has not yet received a response after a certain time. In order not to overload servers, clients SHOULD wait at least 2 seconds before resending a request.

4. Server Configuration and Multiple Servers

Each server MUST maintain a local data base mapping names to IP addresses. If a server does not find a certain name in its local data base, instead of immediately sending a negative response, the server MAY contact one or more other servers, asking if they know the IP address corresponding to the name requested by the client. Upon receiving a response the server should forward it to the corresponding client. Note that in this case it is up to the server to resend the request to another server, if it has not received a response after a certain time.

5. Examples

Suppose a client would like to know the IP address corresponding to the computer1.example.com name. It should send the following message to the server (in hexadecimal notation):

```
IP Address = 00 00 00 00
Name Length = 15
Name = 63 6f 6d 70 75 74 65 72 31 2e 65 78 61 6d 70 6c 65 2e 63 6f 6d
```

If the IP address of the computer1.example.com name is 34.23.44.1, the server MUST respond with the following message:

```
IP Address = 20 17 2c 01
Name Length = 15
Name = 63 6f 6d 70 75 74 65 72 31 2e 65 78 61 6d 70 6c 65 2e 63 6f 6d
```

If, instead, the requested name does not exist, the server MUST respond with the following message:

```
IP Address = 00 00 00 00
Name Length = 15
Name = 63 6f 6d 70 75 74 65 72 31 2e 65 78 61 6d 70 6c 65 2e 63 6f 6d
```

6. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

Author's Address

Alberto Blanc
Telecom Bretagne
Cesson Sevigne
France

Phone: +33 02 99 12 24 25
Email: alberto.blanc@telecom-bretagne.eu

A. Blanc
Telecom Bretagne
June 2014

Sample Protocol Specification: FindIpAddress (Text Version) FindIpAddress-text

Abstract

Sample protocol specification for the text-based version of the FineIpAddress protocol. The goal of this protocol is to allow a client to query a server to find the IP address corresponding to a given name. This is an overly-simplified protocol which is only meant as a simple example.

Table of Contents

1. Introduction

The goal of the FindIpAddress protocol is to allow clients to find the IP address that corresponds to a given name. Each client MUST know the IP address and corresponding port number of at least one instance of a FindIpAddress server.

For the sake of simplicity, the message format is the same for all the messages.

Clients can use either UDP or TCP to exchange messages with a server. When sending a reply, the server MUST use the same layer four protocol used by the client for the corresponding request.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Message Format

All messages have the same format:

```
<IP Address> CRLF
<Name> CRLF
```

Where CR represents the ASCII character "Carriage Return" (13) and LF represents the ASCII character "Line Feed" (10). <Name> is the name for which the client would like to find the corresponding IP address. In a request <IP Address> MUST be '0' (without the quotation marks). In a positive response <IP Address> MUST contain the dotted decimal representation of the IP address (e.g., 135.34.231.3). In a negative response <IP Address> MUST contain '0' (again without the quotation marks).

The server MUST always reply to a request, either with a positive response (in this case the IP Address field contains the IP address corresponding to the name requested by the client) or with a negative response (in this case the IP Address field MUST be set to 0).

3. Reliability

When using UDP, a client MAY resend a request for which it has not yet received a response after a certain time. In order not to overload servers, clients SHOULD wait at least 2 seconds before resending a request.

4. Server Configuration and Multiple Servers

Each server MUST maintain a local data base mapping names to IP addresses. If a server does not find a certain name in its local data base, instead of immediately sending a negative response, the server MAY contact one or more other servers, asking if they know the IP address corresponding to the name requested by the client. Upon receiving a response the server should forward it to the corresponding client. Note that in this case it is up to the server to resend the request to another server, if it has not received a response after a certain time.

5. Examples

Suppose a client would like to know the IP address corresponding to the computer1.example.com name. It should send the following message to the server (recall that CR represents the ASCII character "Carriage Return" (13) and LF represents the ASCII character "Line Feed" (10)):

```
    0CRLF
    computer1.example.comCRLF
```

If the IP address of the computer1.example.com name is 34.23.44.1, the server MUST respond with the following message:

```
    34.23.44.1CRLF
    computer1.example.comCRLF
```

If, instead, the requested name does not exist, the server MUST respond with the following message:

```
    0CRLF
    computer1.example.comCRLF
```

6. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

Author's Address

Alberto Blanc
Telecom Bretagne
Cesson Sevigne
France

Phone: +33 02 99 12 24 25
Email: alberto.blanc@imt-atlantique.fr

The first step of the RES 209 project is to write a (detailed) specification for a session protocol capable of supporting the application Chat While Watching (c2w) described below. The protocol must be able to use either TCP or UDP. The protocol must be capable of supporting all the communication requirements of the application, that is, it must define all the messages needed to achieve the steps described below.

The c2w Application

The c2w application works as follows:

1. It shows a login window (Figure 1) where the user enters the name (or IP address) and port number of the server as well as his/her user-name.

2. If the login is successful, it shows a new window with the "main room" (Figure 2(a)). This window shows the list of all users in the system (specifying whether they are available "A" (i.e., they are in the main room) or whether they are in a Movie room "M"). This window includes also a list of all the available movies as well as a chat area and a text-input box, which can be used to send messages to the other users in the "main room."

3. When the user decides to join one of the movies, the application shows a third window with the list of all the users in that specific room, the corresponding video, a chat area and a text-input box, where the user can type messages to other users watching the same movie (Figure 2(b)).

4. The movie and the main room window have a leave button that the user can press. If the user is in a movie room, he/she goes back to the main room. If the user is in the main room, he/she leaves the system (going back to the login window).

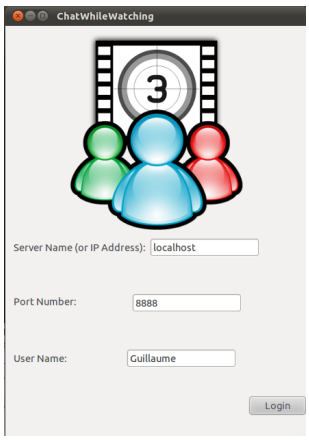
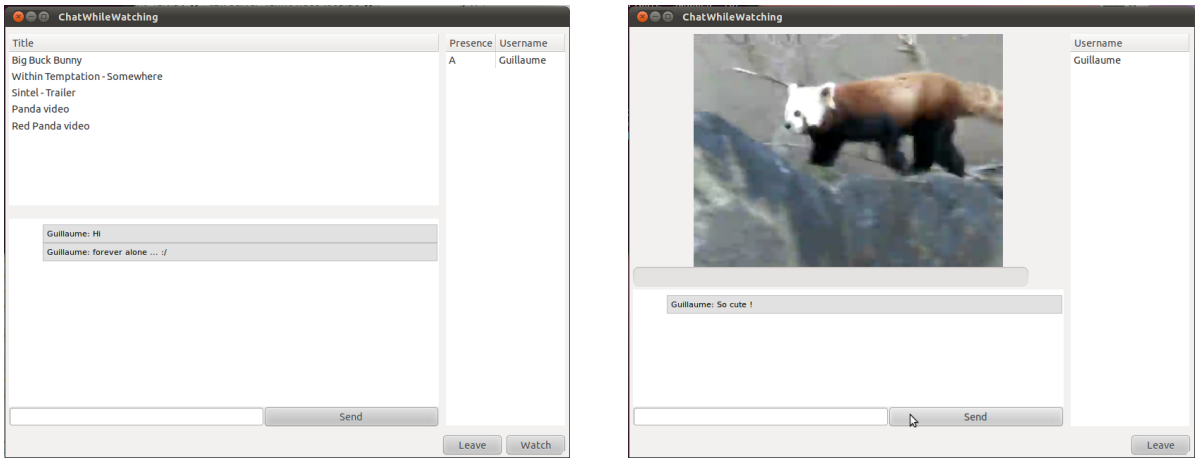


Figure 1: Login Window

Note that the application shows either the login window, the main room window or the movie room window at any given time.

The list of available movies is a configuration parameter of the server. (In other words you can assume that the server "knows" this list.) Every movie is associated with an IP address and a port number that are used to send (and receive) the video flow. The server must include this information when sending the movie list to the clients. For the sake of simplicity, you can assume that the movie list (and associated IP address and port number) does not change after the server has started.

The c2w application uses the Real Time Transfer (RTP) to send the video flow from the server to the clients. Your protocol must not deal with sending and receiving video. The only elements handled by your protocol and related to the video are the aforementioned IP address and port number associated with each movie.



(a) Main Room Window (b) Movie Room Window

Figure 2: The c2w user interface

Protocol Specification

Each group has to write his own document specifying the proposed protocol, in an IETF formatted document (see the template on moodle[1]). Once you have edited your specification, you can generate a .txt file using the `xml2rfc` command, which is available on all the Linux computers at the school. You can use as an example any of the RFC published by the IETF, such as `http://www.rfc-editor.org/rfc/rfc6275.txt`. The complete xml formatting language can be found at `http://tools.ietf.org/rfc/rfc2629.txt`.

This document must contain an abstract, an introduction, and a detailed description of all the messages, including their format (headers, length, payload (if any), etc.). The protocol must contain messages to handle a few simple errors (e.g., malformed message, unknown user, etc.) The document must also contain at least one sample exchange between a client and a server (from the time the client first connects to when it joins a movie and it sends one chat message). The document may also contain any additional scheme/material useful to better understand the proposal.

Your specification can be either in French or in English. If you write the proposal in French, you will notice that `xml2rfc` suppresses all accents in the text output (this is because it is used for the official Internet standards, which must use only printable ASCII characters). In this case you can use `xml2rfc` to generate an HTML file as well: just type `xml2rfc --html inputfile.xml` (obviously you should replace `inputfile` with the name you have used for the xml file).

The name of the xml file must be `spec-r328-s17-gX.xml` where `X` is the number of your group.

You must be ready to present your protocol on Apr. 11, during the PC5-PC6. The presentation must be 10-12 minutes long, excluding questions and further discussion, which will take place after each presentation. Any member of the group can be asked to make the presentation and/or answer questions. Please prepare a short presentation (between 4 and 6 slides, no more than 8). Do not include any introduction in your presentation about the context (e.g., the project, the c2w application, etc.), you should start immediately with your protocol proposal.

[1] See the file `protocol-specification-template.xml` in `td-tp` directory on Moodle.

A Few Guidelines

If you are ever in doubt about what you should put in your document consider the following example: Ideally you should be able to give your protocol specification to two different groups: one implementing the server and the other implementing the client. These two groups cannot talk with each other, nor with you. When they have finished the implementation, the client and the server must be able to work correctly, that is, they must exchange the messages specified in your document and they must interpret them correctly.

This is possible only if you include all the needed details in your specification document. For example, you must specify all the fields, their length (in the case of fixed-size fields), and their content. You must do this for all the different messages. You should also clearly explain the role of each message and if there are restrictions on its usage, for example a certain message could only be used following another one.

By reading your document, it must be possible to decode any protocol message: given an input string it must be possible to uniquely determine the corresponding message and the values of all the fields. This is extremely important, because the process receiving the message must be able to determine its content.

You can make reasonable assumptions about the client and the server implementation. For example, you can assume that the client and the server can store lists of users and associated information (e.g., the current chatroom).

Note that the messages of your protocol will be carried either by UDP or TCP (as clearly pointed out the the requirements). In both cases the implementation will use the Twisted framework that you have used in the Python programming exercises.

As you are writing a protocol specification, you must *not* specify how it will be implemented. You should *not* explain which function should send a certain message. You should *not* specify/assume using specific data structures (e.g., lists, hash tables).

Note that this is exactly the role that specifications of standard protocols have in the real world: different people (groups/companies) can write independent implementations of a protocol but, as long as they all respect the specification, the different implementations will be able to work together (i.e., "interoperate").

Submission Checklist

Before you submit the final version of the document, you should make sure all the following conditions are satisfied:

1. Your names as well as the group number are on the first page.
2. The files are in the `spec` folder of your Git repository and are named `spec-r328-s17-gX.xml`, `spec-r328-s17-gX.txt` (and `spec-r328-s17-gX.html` if your document is in French), where `X` is the number of your group.
3. All the lines in your document are at most 72 characters long (`xml2rfc` gives you a warning if this is not the case).
4. You have specified the format of all the packets in your protocol:
   (a) The size of each field.
   (b) How fields are organized in each message (unless this is not needed, for example because each field contains its name).
   (c) The format/content of each field.
   (d) The size/content of the payload (if any).
5. The protocol you have proposed works on an unreliable transport (e.g., UDP).

6. There is at least one complete example in your document (from the time the client first connects to when it joins a movie and it sends one chat message).

RES 209 Optional Programming Exercises: Python Socket Programming

Spring 2017

The goal of these first exercises is to familiarize yourself with Python, with the socket interface and with the fork system call. The second exercise asks you to think about the constraints imposed by the socket interface and the multi-process server model. The third exercise asks you to overcome these constraints by using the `select` system call.

The most important elements of these exercises are summarized at the very end of this text. Make sure you read that part and that you fully understand each point.

Recall that there are two ways to run a python program (or script, the two terms are equivalent in the context of RES 209):

1. You can use the command `python script.py` where script.py is the name of the script file.
2. You can put the following in the *first* line of your script: `#!/bin/env/python` and then mark the file as executable by using the command `chmod u+x script.py` then you can launch the script by typing `./script.py` in a shell. Note that the working directory of the shell, sometimes called the "current directory" has to be the one where the script is, hence the `./` "dot-slash" before the file name, indicating that the script is locate in "this" directory (in Unix the dot represents the current directory).

Exercise 1

As a first example of a simple program using the socket interface, you will implement a client-server pair, which we will call, for a lack of a better name, `echoserver` and `echoclient`. The server will simply print on the screen all the messages received from the client. Therefore, even though sockets can be used for bi-directional communication, in this first exercise, we will use them only in one direction. (As it will become clear later on, this greatly simplifies the programs.)

(a) **Single Process Server** First, consider the simplest possible case: a server capable of handling only one client at a time. In this case the server can use only one process. As a starting point, you can use the skeleton files available on Moodle in the supporting material folder for RES 209 (`td-tp/res-328-tp-1.tar.bz2`). To extract the files you can use the command `tar xjf res-328-tp-1.tar.bz2`. The files `echo_client_socket_skeleton.py` and `echo_server_socket_skeleton.py` contain the code for the client and the server, respectively.

The server should wait for the client to open a connection (using the port number specified on the command line). Then it will simply display whatever it receives from the client.

The client should open a socket connecting it with the server, using the address and the port number given on the command line. Then it should wait for the user to input a string and send it to the server.

Action Item 1 *First, make a copy of the* `echo_client_socket_skeleton.py` *file called* `echo_client_socket.py` *and a copy of the* `echo_server_socket_skeleton.py` *file called* `echo_server_socket.py`.[1] *Then, modify the files in order to obtain the behavior described above.*

(b) **Multi-Process Server** Using the `fork` system call (from the `os` module), one can write a server capable of handling multiple clients. In this case, the server will start by listening for incoming connections on the port specified on the command line. Whenever a new connection request arrives, it will spawn a new child process, which will

[1] This way, if you ever want to start from scratch again you still have a copy of the original skeleton files.

handle the connection (i.e., it will display what it receives from the client). Clearly you can use the same client as before.

Action Item 2 *First, make a copy of the* `echo_server_socket.py` *file called* `echo_server_socket_multi_clients.py`*. You can accomplish this by typing* `cp echo_server_socket.py echo_server_socket_multi_clients.py` *in the directory containing the echo_server_socket.py. Then, modify the newly created file in order to obtain the behavior described above.*

i. Make sure you test your server with multiple client in parallel

ii. What happens if you change the size of the receive buffer (that is the integer passed to the recv function) and you send fairly long messages from the client?

iii. Using the command `netstat -atn` you can list all the network connection on a machine. Find the ports associated to each connection between the server and the clients.

iv. As you might have noticed, in the sample code you used, the (server) socket was bound to an empty address (`host=''`).

What would happen if we use `host='192.168.100.10'` instead? Why? (You can try to make this change and run the program to see what happens.)

Exercise 2

In the previous exercise, we used all the connections in only one direction: the client(s) were writing and the server was reading. In this exercise we will use each connection in both directions. The goal is to make a first (baby) step toward a chat server. Now the second client to connect will display what is sent by the first client to connect. The server will simply forward the messages from the first client to the second.

The server starts by listening for the first connection on the port number given on the

command line. After the first client makes a connection, the server sends a string with the number 1 to the client, so that it will know that it is the first client. Then the server waits for the second client to connect, when it does, the server will send a string with the number 2. At this point the server is ready to start forwarding messages between the two clients: it will wait for the incoming messages from the first client and forward them to the second one.

The client starts by connecting to the server, using the address and port number given on the command line. Once the connection is established, it receives (reads) one single character from the server. If the character is '1', the client asks the user to input the messages that will be sent to the server (and, therefore, to the other client). Basically, in this case, the client has the same behavior as in the previous exercise. If the character received from the server is '2', instead, the client will simply wait for messages from the server and it will display them.

Action Item 3 *Do the following:*

- *Make copy of the* `echo_server_socket.py` *file called* `forwarding_server_socket.py`*, again, you can do this by typing*
  `cp echo_server_socket.py forwarding_server_socket.py` *in the directory containing* `echo_server_socket.py`*.*
- *Make a copy of the* `echo_client_socket.py` *file called* `forwarding_client_socket.py`*. As usual, you can do this by typing*
  `cp echo_client_socket.py forwarding_client_socket.py`*.*
- *Modify both newly created files in order to obtain the behavior described above.*

Do you think that it would be possible, using only the system calls you have utilized so far, to modify the server so that a third client can connect to the server and receive the messages from the first client as well? (Hint: think about blocking calls and what they imply.)

Exercise 3

Using the `select` system call, which is available in Python, it is possible to overcome the limitations of the solution presented in the previous exercise. `select()` allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., **read**(2)) without blocking. In our case, the server could handle multiple clients, forwarding the messages sent by the first client to all the others.

Action Item 4 *Make a copy of the* `forwarding_server_socket.py` *file called* `forwarding_server_socket_select.py`*. Modify the code of the server by using* `select`*, so that it will handle more than two clients.*

The following is an excerpt from the Python documentation:

`select` — **Waiting for I/O completion**

This module provides access to the `select()` and `poll()` functions available in most operating systems, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

[...]

`select(`*rlist, wlist, xlist, [timeout]*`)`

This is a straightforward interface to the Unix `select()` system call. The first three arguments are sequences of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned. Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

Most Important Concepts/Elements

At the end of these exercises you should fully understand (and remember) the following points:

- The `socket` interface allows processes to communicate using a TCP (or UDP) connection:
- In the case of TCP connections:
  - The data sent with a single `write` (or `send`) function can be received with multiple `read` (or `recv`) calls (you have seen this in the first exercise (1(b)ii) when you have changed the size of the receiver buffer).
  - A TCP connection offers a reliable "byte-stream" service: the bytes transmitted by the sender are reliably delivered in the correct order to the receiver.
- Some of the functions of the `socket` interface are *blocking* (e.g., `recv`): when a program calls one of these functions, its execution is suspended until a certain event "unblocks" the call. For example, in the case of the `recv` function, the execution is blocked until it receives data on the connection.

- A process can be blocked only on one function on any given time. This implies that a single process can wait for only one event at any given time.
- The `fork` system call creates a clone of a process. Each clone (child process) can then block on a different event (e.g., receiving data from a specific client).
- Different processes do not share the same address space, therefore they cannot share data easily.[2]
- A process can use the `select` system call to wait for multiple events. This way a single server process can wait for data (and connection requests) from multiple clients.

[2] As mentioned in class, this can be done, but it is outside the scope of RES 209.

Introduction to IP Networks

RES 201

SRCD Department
Spring 2017

"Mineure Réseaux"

- UV1:
  - RES 201: Réseaux IP (Internet Protocol)
  - RES 202: QoS et ingénierie des réseaux
  - RES 203: Réseaux mobiles et réseaux sans fils
  - RES 209: Programmation Réseaux (Project)
- RES 201 and RES 209 tighly coupled
- RES 209 associated with RES 119 (FIP)

Course Presentation

- Main goal: an *introduction* to IP networks
  - top-down approach (starting from the Application layer)
  - lectures will concentrate mainly on the Application, Transport,Network and MAC layers
  - you will have to design and implement a simple "session" protocol
- a large part of the course dedicated to the project
- show the importance of standards (interoperability)
- changes motivated (among other factors) by the comments of the students

Course Overview

- overview of the TCP/IP protocol suite
- network programming (Python)
- the IP protocol
  - addressing
  - routing
  - Network Address Translation
- Local Area Networks (LAN)
- Application Layer Protocols

The Project (I)

- Context: "Chat While You Watch" application
  - chat with other users watching the same video (movie)
    - streaming video server
    - users choose a video to watch
    - one chatroom for each video (users join the chatroom automatically)
  - client and server need to use a "session" protocol:
    - basically a "signaling" protocol (join, leave, get movie list, join movie, etc.)
    - you will have to specify and implement your own
  - we have already developed the application
  - you have to implement only the protocol-specific parts

The Project (II)

- Goal: write the protocol specification and implement it
  - "hands-on" approach
  - similar to what is done at the Internet Engineering Task Force
- groups of 4 people
- you will receive an email with the list of all the groups and their members

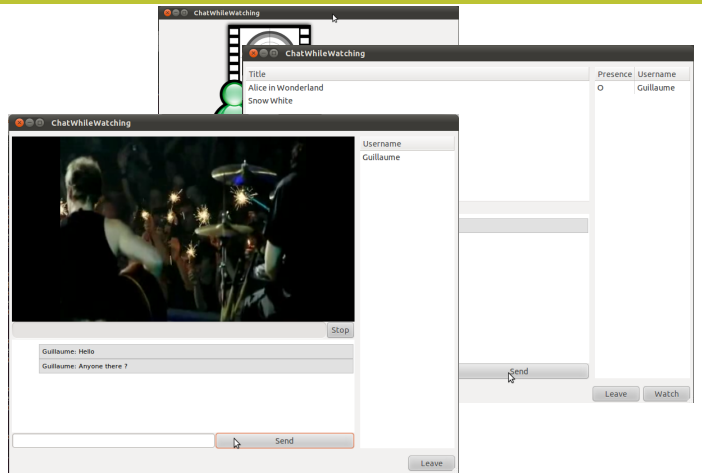
Project Goals and Motivations

- Goals:
  - design and implement a simple communication protocol
  - understand the role (and importance) of standards
  - introduction to network programming
- Motivations:
  - networks exist to connect end-nodes (which are more numerous than intermediate systems) ⇒ end-nodes are extremely important
  - more and more communicating devices (objects) ⇒ understanding their interactions with the network is useful
  - layered model ⇒ same principles apply to lower level protocols (each protocol uses the services offered by the lower layers)
  - network programming is far from simple ⇒ it can be a useful and profitable skill

The Future is Software Defined

- "Software is eating the world" (M. Andreessen, cofounder of Netscape)
  - Software Defined Networks
  - Network Function Virtualization
  - Software Defined Radio
  - Software Defined Storage
- virtualization:
  - use software to share physical resources
  - essential to know the hardware (electronics)
- engineers will have to deal with software
  - useful to know something about programming/software

c2w



Students' Contributions (I): Protocol Specification (I)

- *Each group* has to write a *different* protocol specification
  - do not even glance at the specifications of another group
  - any form of copying (even minor) will be severely sanctioned
- the functional requirement document is available on Moodle:
  - your protocol *must* satisfy all the requirements:
    - what messages are exchanged (e.g., login request, user list request, etc.)
    - the format of each message
  - you are expected to use `xml2rfc` (see sample file on Moodle)
    - IETF tool
    - do compile very often (detect errors as soon as possible)

Students' Contributions (I): Protocol Specification (II)

- Why do we need standards (i.e., protocol specification)?
  - different and independent groups can write inter-operable implementations
  - in real life this means different vendors can offer compatible products (e.g., routers, web-browser/servers, e-mail clients/servers etc.)
- ⇒ goal of your protocol specification document:
  - two other groups should be able, by only reading your document, to implement a client and a server which can work correctly together

Do Not Panic!

You are not expected to write a *perfect* protocol specification (you will be graded mainly on your effort). Just make sure your specification contains all the messages needed and their format.

Students' Contributions (II)

- Some "guidance" on protocol specifications:
  - Sample protocol specification and functional requirements (FindIpAddress) (extremely simple)
  - Moodle Quiz about the sample specification
    - each *student* must complete it before 9 am on the day of RES 209 PC-1 and PC-2 (April 4)
  - two-step review process

Students' Contributions (III)

- Each student must submit the review of two protocol specifications using HotCRP (9:00 am, Apr. 11) (the teaching staff will match students and documents to be reviewed)
  - the reviews will be "reviewed" and will be considered when assigning grades for the first part of the project
  - only the "best" documents will be presented during RES 209 C3(Apr. 11)
  - the reviews will be a starting point for discussion
- during the presentations (Apr. 11) we will:
  - either select one proposed protocol specification
  - or (most likely) merge/modify some of the proposed ones to have the "final" protocol specification (for the implementation)
- each group can implement either the "common" specification or its own (with the consent of the instructor)

Students' Contributions (IV) and Grading

- implementing a protocol specification is essential to "prove" the specification is complete and reduces the risk of major errors (the IETF requires at least two independent implementation of a proposed protocol before this can become a standard).
- your implementation can cover only a subset of the requirements
- you will be responsible only for the protocol implementation (re-use existing code)
- individual grading all along the project

Deadlines

date	time	
Apr. 4	9 am (CET)	complete the FindIpAddress quiz on Moodle
Apr. 4	end of class	push the draft protocol specification to the Git server
Apr. 7	9:00 am (CET)	push the final protocol specification (Git and HotCRP)
Apr. 11	9:00 am (CET)	complete the reviews using HotCRP
Apr. 11	9:00 am (CET)	push the specification presentation on Git
May 30	8:55 am (CET)	push final version of the code to the Git server
May 30	all day	final project evaluation and interoperability tests
TP: 4, 5, 6	end of class	push the code to the Git server for testing

Code Testing

- why? to make sure each group is working *throughout* the project
  - to force you to test your code often (there is no point in writing 100 (or more) lines of code without trying to execute them!)
- what? tests covering specific goals (e.g., sending a well-formed packet)
- when? each group MUST push their changes to the Git server by 9 am of the day of each lab programming session ("TP")
- pass/no-pass grades used mainly for progress/effort monitoring

Git

You are expected to read a tutorial about Git. For example the one at `https://git-scm.com/book`, you should read the first two chapters (Getting Started and Git Basics).

TCP *and* UDP Implementation

- TP-1 to TP-3 (more closely related to the project)
- you must implement the TCP *and* UDP version of the protocol for the project
  - ⇒ must correctly implement reliability mechanisms
- code testing framework (TP-2&3 and project)
- optional TP about the socket interface (still available on Moodle)

No Solutions Provided

- PC-0 is the only available solution
- don't bother asking for others: the answer is NO!
- we can give you feedback on *your* solution:
  - please try to be clear
  - allow a few days for a response (if no response after 5 working days ⇒ send again)
  - please specify the date of the exam if it is less than 1 week away
- why?
  - no such thing as "the" solution (many are possible)
  - encourage active participation in the PCs/TPs and beyond (contact teachers)
  - you must learn/get used to taking notes
  - no more "school-like" mentality
  - cannot ask your boss to give you "the solution" (when/if working)

PC Groups

- students can ask questions either in English or French in all groups
- you can submit protocol specifications either in English or in French

Introduction to TCP

- offers a "byte stream" service
- *Bytes* will be delivered in order and reliably
- no guarantees about message boundaries
- *reliable* service

The rest will be on the board

Introduction to Python

- interpreted language (i.e., no compilation/linking needed)
- dynamic typing (no variable declaration)
  - type checking done only at run-time
  - ⇒ often errors not detected until run-time
- high level language
- many existing libraries (the "standard" one is fairly large)
- object-oriented (optionally)
- useful for scripting (much easier than shell or Perl!)
- good tutorial at https://docs.python.org/3.4/tutorial/

Homework

You are expected to read the following chapters of the tutorial by the end of next week: 1, 2, 3, 4, 5 and 9. You are strongly encouraged to read the other chapters as well.

Built-in Types and Data Structures

- Python has the usual built-in types:
  - integer, real, (complex), string, etc.
- Several built-in data structures as well:
  - lists (ordered *mutable* collections, `l = [1,2,3]`)
  - tuples (ordered *immutable* collections, `t = (1,2,3)`)
  - and a few more

Caveats

- indentation is important:
  - blocks of code (e.g., a loop) identified by the indentation
  - use 4 spaces to indent code
  - do not use tabs! (make sure that your editor uses spaces when you press the tab key)
  - mixing tabs and spaces is a recipe for disaster (cryptic error messages)
  - Eclipse with the PyDev plugin takes care of this (you are encouraged to use it)
- parameter passing:
  - by value for immutable objects (i.e., numbers, strings, tuples, etc.)
  - by reference for mutable objects (i.e., lists, dictionaries, etc.)
- different versions of Python:
  - we will be using 3.4
  - (somewhat) significant differences between 2.x and 3.x

A First Python Script

```
#!/usr/bin/env python3.4

import sys

print("This is a simple program which adds two numbers ")
message = 'Type two numbers (put a space between them and press enter)
while True:
    inputString = input(message)
    fields = inputString.split()
    a = int(fields[0])
    b = int(fields[1])
    print("the answer is: {0}".format(a + b))
```

Comments

- the first line (`'#!/usr/bin/env python3.4'`) tells the shell to use the `'python3.4'` command to execute the script (the same as typing `'python3.4 add.py'`)
- the `import` statement loads the module sys (like `'#include'` in C)
- can use simple ' or double " quotation marks for strings
- `int(string)` converts a string to an integer (can throw and exception)
- `input` is a *blocking* method (i.e., function):
  - it returns (i.e., terminates) only after a certain event happens (in this case the user presses the enter key)

A Second Python Script

```
class AddApp(Gtk.Application):
    def __init__(self):
        Gtk.Application.__init__(self, application_id="apps.test.add")
        self.connect("activate", self.on_activate)
    def on_activate(self, data=None):
        self.window = Gtk.Window(type=Gtk.WindowType.TOPLEVEL)
        self.window.set_title("Add Example (GUI Version")
        grid = Gtk.Grid(column_spacing=2, row_spacing=10)
        self.window.add(grid)
        self.addButton = Gtk.Button(label="Add")
        self.quitButton = Gtk.Button(label="Quit")
        self.bEntry = Gtk.Entry()
        grid.attach(self.aEntry, 0, 0, 1, 1)
        grid.attach_next_to(labelPlus, self.aEntry, Gtk.PositionType.RIGHT, 1, 1)
        grid.attach_next_to(self.bEntry, labelPlus, Gtk.PositionType.RIGHT, 1, 1)
        self.addButton.connect("clicked", self.on_add_clicked)
        self.quitButton.connect("clicked", self.on_quit_clicked)
        self.window.show_all()
        self.add_window(self.window)
    def on_add_clicked(self, widget):
        a = int(self.aEntry.get_text())
        b = int(self.bEntry.get_text())
        result = a + b
        self.resultEntry.set_text(str(result))
    def on_quit_clicked(self, widget):
        sys.exit()

if __name__ == "__main__":
    app = AddApp()
    app.run(None)
```


Comments

- no blocking calls (other than `app.run()`)
- example of *event based programming* or *asynchronous programming*:
  - (almost) each function is called when an event takes place
  - exception: functions called at the beginning to "build" and "initialize"
  - the "*event loop*" is always running and takes care of calling the right function at the right time
    - e.g., it calls `on_add_clicked` when the user clicks on the add button
  - cannot use blocking calls anywhere except to start the event loop!

Operating System Processes

- from Wikipedia: "a *process* is an instance of a computer program that is being executed"
  - each time you execute a Python script the OS creates a new process
  - each process has its own "context," including variables
  - multiple processes can be instances of the same program (e.g., `add.py`)
- new processes created with the `fork` system call:
  - it *clones* a process (the two processes are *exactly* the same, except for the return value of the `fork` system call)
  - it is up to the programmer to handle the `fork` correctly
  - after the `fork` the two processes do not share memory (i.e., variables)
  - after the `fork` the two processes do share file descriptors (i.e., files and sockets)
- processes can "talk" to each other (this is beyond the scope of RES 201)
- process $\neq$ thread (this is beyond the scope of RES 201)

Programmation réseaux

RES 119 / RES 209

Notes de cours – Printemps 2017

Christophe COUTURIER
Département SRCD
christophe.couturier@imt-atlantique.fr

50

Introduction

- Organisation
- Déroulement du projet

Objectifs

- **Cours: Bases des réseaux => RES 112 / RES 201**
  - Modèles de référence (OSI, TCP/IP)
  - Importance de la normalisation
  - Adressage IP
  - Réseaux locaux
- **Apprendre par la pratique => RES 119 / RES 209**
  - Conception d'un protocole
  - Implémentation en Python (TP / projet)
  - Mode projet

Projet : Objectifs

- **Spécifier et implémenter un protocole**
  - Concevoir un protocole (simple) de niveau applicatif
  - Acquérir des notions de programmation réseau
- **Travail autonome**
  - En groupe de 4
  - Se confronter aux difficultés pour apprendre par la pratique
- **Mode projet**
  - Mise en situation d'ingénieur
  - Outils collaboratifs (GIT, Redmine, HotCRP…)
  - Méthodes (organisation, revues, travail en équipe…)

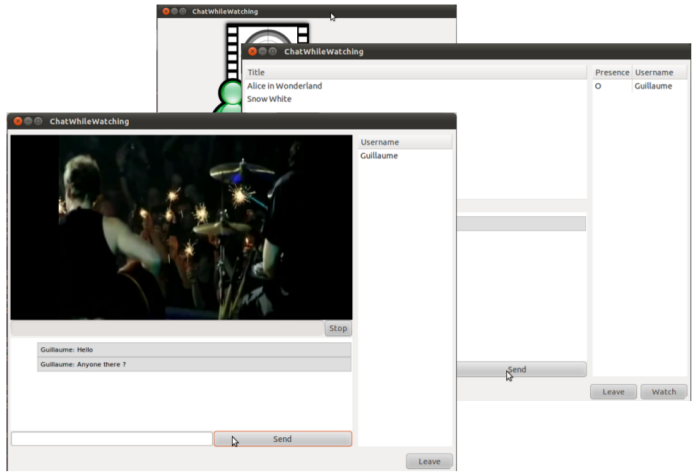
Projet: Organisation

- **Travail en groupe**
  - Quadrinômes (binômes pour TP1,2,3)
  - Mixité FIP-FIG
- **Évaluation**
  - Individuelle
  - Contrôle continu portant sur le projet
    - Évaluation permanente au cours du projet (GIT !!!)
    - Spécification
    - Démonstration de fin de projet

Projet : Présentation

- **Application C2W: « Chat While Watching »**
  - Chat entre plusieurs utilisateurs d'un serveur de vidéo
  - Application déjà existante
    - Seul le protocole est à implémenter
    - Respecter les API déjà en place
  - Programmée en Pyhton 3.4 + Twisted
- **Pour se faire la main**
  - 3 TP sur « sibyl »
  - Même environnement (Python + Twisted)
  - Appli plus simple que C2W
- **Documentation sur MOODLE**

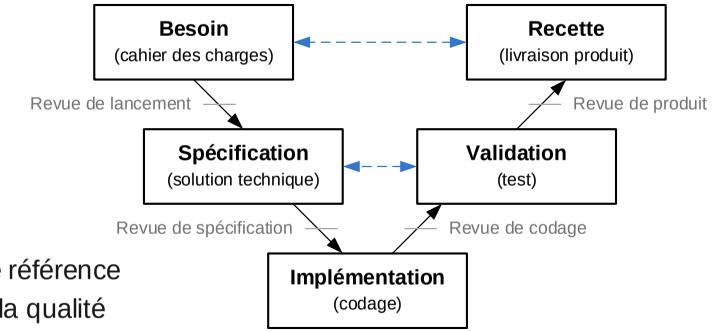
Projet : C2W



Projet : C2W

- **Fonctionnalités**
  - Serveur de vidéos (streaming)
  - Une « salle » par vidéo
  - Les spectateurs d'une même salle échangent des messages (chat)
- **Protocole de niveau « session » en client/serveur**
  - Les utilisateurs s'enregistrent
  - Téléchargent des listes de films et d'utilisateurs
  - Envoient / reçoivent des messages (chat)

Projet : Méthode

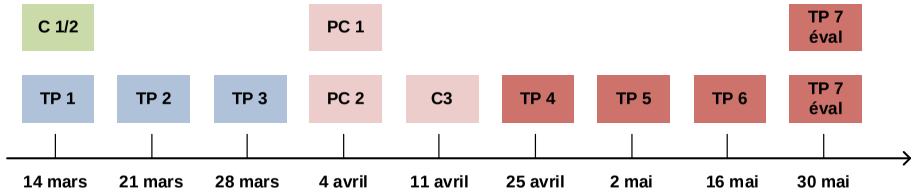


- **Cycle en V:**
  - Méthode de référence
  - Support de la qualité
  - Importance de la documentation
  - Outil de collaboration

Écrire – Faire – Tester

Projet: Déroulement

- **3 séances de TP (Sibyl)**
- **Élaboration de la spécification**
  - 1 journée en groupe (PC 1/2)
  - Choix de la spec commune en classe entière (C3)
- **Implémentation du protocole**
  - 3 TP + Evaluation (TP7)
- **Possibilité de travailler en anglais**



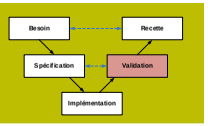
Projet : Directives

- **Faire par soi même**
  - Détection du plagiat et **sanction**
  - Aucun intérêt et discrédite l'ensemble du groupe
- **Lire les documents avant les séances !!!**
- **Travailler à la maison**
- **Publier très régulièrement son travail sur GIT**
- **Sources de documentation**
  - Document de TP/PC + cours sur MOODLE (énoncé, exemple de protocole…)
  - Documentation de l'API de C2W
  - Forum de RES 302
  - Documentation officielles de Python et Twisted sur internet
  - Tutoriaux, exemples de code sur internet
  - Livres (cf. références dans poly)…
- **Collaborer**

Projet : La spécification

- **Solution technique**
  - Chaque groupe doit avoir sa propre solution
    plusieurs solutions existent pour un même problème.
  - Ne pas discuter avec les autres groupes!
- **Unique document pour implémenter => exhaustivité**
  - Spécificateur != développeur
  - Pas d'interprétation possible (par ex. client et serveur doivent pouvoir être implémentés par des équipes différentes)
  - Détail du format des messages
  - Exemples pour illustrer
- **Formatage via xml2rfc**
- **Cela reste un exercice !**
  - Elaboration d'une spec commune en C3

Projet : Tests



- **Garantie de qualité**
  - Vérifier son interprétation de la spec
  - Se confronter à un regard externe
- **Jalonnement du développement**
  - Permet de savoir ce qui marche / ne marche pas
  - Non régression (a-t-on cassé ce qui marchait avant?)
- **Tests fournis par les enseignants**
  - Livraison au fur et à mesure via GIT
  - Simulation de cas difficiles à tester (ex. pertes de paquets)
- **Un test OK ne signifie pas une implémentation correcte**
  - À compléter par des tests fonctionnels
  - Mais test KO => erreur
- **Les TP Sibyl intègrent déjà des tests**

Jalons

Ces dates sont fermes, pas de dérogation

(comptent dans la note finale)

14 mars	Fin TP1	Premier commit sur GIT
4 avril	9:00	Quizz FindIpAddress sur MOODLE
4 avril	Fin PC 1/2	Spécification en l'état (txt, doc, pdf, xls, jpg…) sur GIT
7 avril	9:00	Spécification définitive sur GIT et HotCRP
11 avril	9:00	Revue des spécification assignées sur HotCRP
11 avril	9:00	Présentation de la spécification sur GIT
30 mai	9:00	Code définitif du projet sur GIT

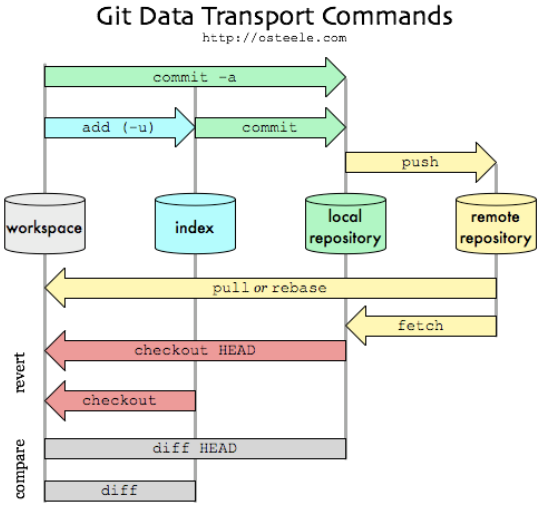
+ publications régulières sur GIT

(TP, spécification, projet…)

GIT: Gestion de version

- **Historisation**
  - Préparation (index) : `git add`
  - Mémorisation de l'état des fichiers à l'instant t: `git commit`
- **Partage des fichiers**
  - Remontée vers un serveur (dépôt): `git push`
  - Récupération des mises à jour d'un dépôt: `git pull`
- **Initialisation**
  - Création d'un dépôt local: `git init`
  - Récupération d'un dépôt: `git clone`

GIT: Gestion de version



GIT: Quelques outils

- **Visualisation graphique**
  - `gitk`
  - Redmine: https://redmine-df.telecom-bretagne.eu
- **Résolution de conflits**
  - Comparaison graphique: `meld`
- **Récupération d'une version**
  - Sélection: `git checkout --theirs / --ours`
- **Suivi**
  - `git status`
  - `git log`

GIT: Documentation

- **Doc en ligne**
  - https://git-scm.com/book/en/v2
  - Lien sur Moodle
  - Lire les 3 premiers chapitres

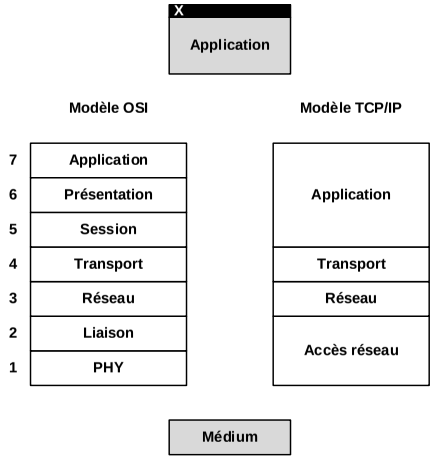
Rappels

- Modèles en couches
- Notion de protocole

Modèle en couches

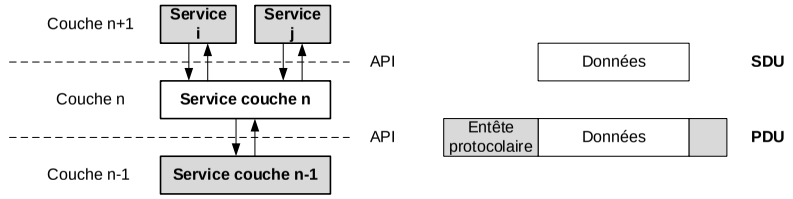
- **Structuration: de la machine vers l'utilisateur**
- **Intérêt:**
  - Universalité
    - Tout le mode se comprend; facilite l'interconnexion
  - Interopérabilité
    - Un même protocole peut tourner sur des machines différentes (Cisco/Alcatel ou PC/Linux/MAC)
  - Adaptabilité
    - Je peux utiliser IP sur du LTE, mais aussi sur Ethernet ou sur Wifi
    - Mon développement devient donc générique
  - Décomposition des problèmes et du travail
    - Equipes de développement différentes (parallélisassions)
    - Décomposition HW/SW
  - Évolutivité
    - Un protocole peut évoluer indépendamment des autres car les interfaces entre les couches ne changent pas

Exemples des modèles OSI et TCP/IP



- **Modèle OSI**
  - Open System Interconnection
  - Modèle de référence
- **Modèle TCP/IP**
  - Modèle de fait d'Internet
  - Fusion des couches 5 à 7
  - Fusion de 1 et 2 (ex. Ethernet)
- **Important**
  - Les applicatifs et les media sont EXTERIEURS aux modèles
  - Certains protocoles sont difficiles à situer (par ex ARP)

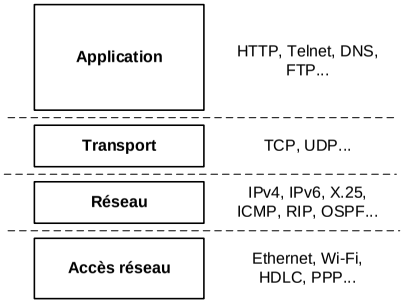
Modèle en couches: définitions



- **Chaque couche ajoute un ou plusieurs services à la précédente**
  - Interactions via des primitives (API: Application Programming Interface)
  - Services identifiés par leur SAP (Service Access Point)
- **SDU (Service Data Unit)**
  - Données utiles qui entrent (et sortent) par le haut de la couche
- **PDU (Protocol Data Unit)**
  - Données enrichies des informations protocolaires
  - Transitent par le bas de la couche

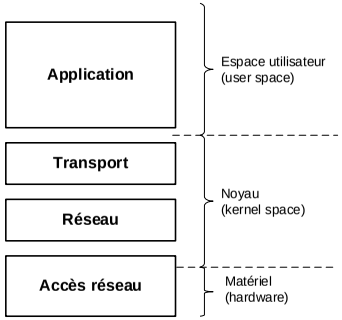
Exemples de protocoles

- **Exemples de protocoles répandus**
- **Certains sont difficiles à placer**
  - Ex: ARP qui assure la correspondance @MAC/@IP
- **Approche cross-layer**
  - Collaboration entre les différentes couches
  - Par ex: router en fonction de la qualité du signal radio



Implémentation sur les machines

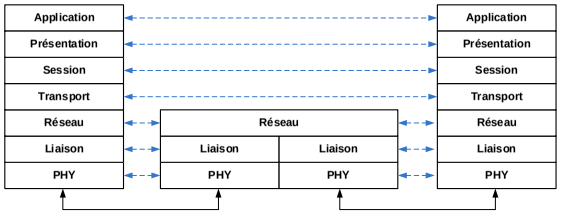
- **Espace utilisateur**
  - Applications et Protocoles applicatifs
  - Modification facile (recompilation programme)
- **Noyau**
  - Protocoles de transport et réseau
  - Couches haute de l'Accès (« drivers »)
  - Modification => recompilation du noyau
- **Matériel**
  - Couches basses de l'accès
  - Couche Physique
  - Modification => constructeur (microcode ou HW)



- **Modèle « général », variantes possibles**
  - Par ex. accélération matérielle (short path)

Modèle en couches: Interconnexion

- **Point à point aux niveaux 1, 2 ou 3**
- **Communication de bout en bout à partir du niveau 4**



Niveau	Unité de données	Ex. d'équipements
4. Transport	Segment / datagramme	
3. Réseau	Paquet	Routeur
2. Liaison	Trame	Commutateur (switch) Modem, carte réseau
1. Physique	Signal	Hub, connecteur

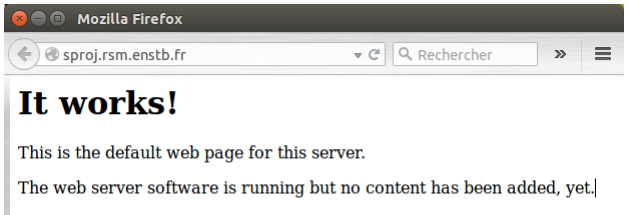
Passage à l'échelle

- **« Scalabilité » : propriété fondamentale des protocoles/architectures**
  - Généralisation à l'échelle de l'internet
  - Dimensionnement (par ex. taille des champs d'adresse)
  - Mode de fonctionnement (centralisé, réparti, distribué…)
  - Puissance de calcul des équipements
  - Capacité mémoire des équipements
  - Tolérance aux pannes
  - …
- **Mais pas indispensable**
  - Ex.: PPP n'assure que des liaisons point à point
  - => savoir quel est le besoin

Exemples de protocoles applicatifs

- HTTP, DNS
- Protocole texte, protocole binaire

Exemple de connexion HTTP




```
$ telnet sproj.rsm.enstb.fr 80
Trying 192.108.119.145...
Connected to rmtest.rsm.enstb.fr.
Escape character is '^]'.
GET /index.html
<html><body><h1>It works!</h1>
<p>This is the default web page for this
server.</p>
<p>The web server software is running but no
content has been added, yet.</p>
</body></html>
Connection closed by foreign host.
```

Exemple de connexion HTTP

- **Contexte**
  - Les couches inférieures permettent d'envoyer et de recevoir des données de façon fiable
  - HTTP: transmet à un client des pages web (HTML) situées sur un serveur
  - L'application (navigateur web) réalise l'affichage sur le client
- **Que voit-on dans cet exemple?**
  - Phase d'ouverture de connexion (TCP, avec résolution d'@IP)
  - HTML: représentation des données (HyperText Markup Language)
    - Format de représentation des données (en mode texte)
    - Séparation des champs par des balises HTML
  - HTTP: transfert des données (HyperText Transfer Protocol)
    - Commandes en mode texte (GET…), Séparateur: <CR><LF>
    - n° de port: 80
    - ressource par défaut: index.html

Exemple en HTTP 1.1

```
$ telnet sproj.rsm.enstb.fr 80
Trying 192.108.119.145...
Connected to rmtest.rsm.enstb.fr.
Escape character is '^]'.
GET /index.html HTTP/1.1
Host: sproj.rsm.enstb.fr

HTTP/1.1 200 OK
Date: Tue, 25 Aug 2015 09:52:15 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Thu, 08 May 2014 14:21:22 GMT
ETag: "b1-4f8e43200fc80"
Accept-Ranges: bytes
Content-Length: 177
Content-Type: text/html

<html><body><h1>It works!</h>
</body></html>
Connection closed by foreign host.
```


Exemple en HTTP 1.1

- **En précisant la version, on change le protocole**
  - Il est nécessaire de préciser le Host dans la requête
  - La réponse contient désormais un entête
    - Délimitation par « : » et <CR> <LF>
    - Type de réponse (200=>OK): permet au récepteur de connaitre le résultat sans analyser le code HTML (le contenu de la page d'erreur peut changer
    - Longueur (177): a-t-on reçu l'intégralité du message?
    - Type de contenu…
  - La connexion reste ouverte (pas longtemps) à la suite de la première requête
    - Permet de demander du contenu additionnel sans ré-ouvrir une connexion
  - Le même serveur gère les deux versions
    - Compatibilité ascendante

HTTP: Protocole client-serveur

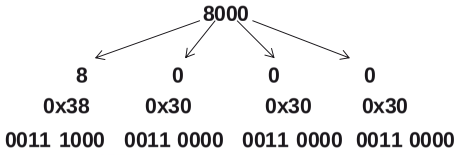
- **Le serveur est toujours allumé et attend les connexions**
  - Un serveur doit traiter en // les requêtes de plusieurs clients
  - Besoin d'un contexte pour chacun des clients connectés
- **En production: grosse machines**
  - Ou plutôt machines virtuelles sur ces serveurs
  - Administrées à distance
- **Ou plus légères**
  - PC classique si charge faible ou en phase de développement
  - Objet, capteur ….
- **En TP: serveur et client tourneront sur la même machine**
  - Mais ce sera bien 2 programmes distincts
  - En revanche, une partie du code source du programme pourra être partagé entre ces 2 programmes (par ex. format des messages)
- **Un serveur peut être lui-même client d'un autre serveur**
  - Ex: un serveur WEB peut être client d'une base de données

Mode texte / Mode binaire

- Mode de transmission (et de stockage) des données

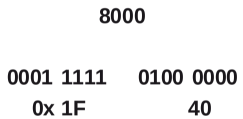
MODE TEXTE

- Informations sont sous forme de caractères lisibles
- Evolutivité mais complexité de décodage (parsing)
- Codage en ASCII, UTF-8, Unicode…
  - 1 caractère = 1 octet (ASCII)
    ex. 'A': 65 (0x41); '0': 48 (0x30)



MODE BINAIRE

- Écriture sous forme de nombre binaire
- Le texte reste représenté sous forme de code (ASCII, UTF…)
- Plus compact que le mode texte

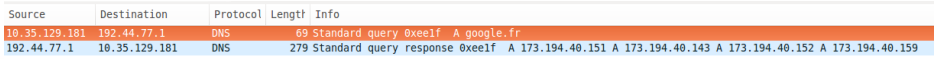


DNS: exemple de protocole binaire

- DNS: Domain Name System
  - Donne la correspondante entre nom de domaine et adresse IP
  - Protocole client – serveur en binaire
  - Format des messages dans RFC 1035
- Capture à l'aide d'un analyseur de protocole (wireshark)

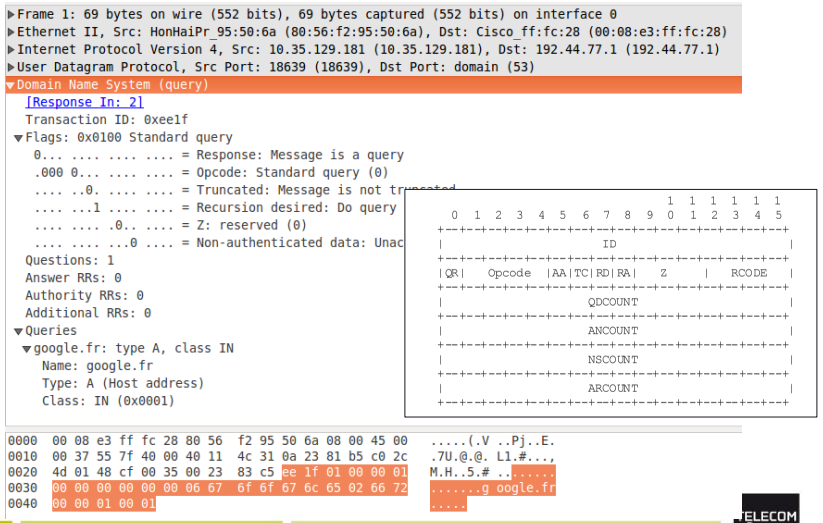
```
$ nslookup google.fr
Server:         127.0.1.1
Address: 127.0.1.1#53

Non-authoritative answer:
Name:   google.fr
Address: 173.194.40.151
Name:   google.fr
Address: 173.194.40.143
Name:   google.fr
Address: 173.194.40.152
Name:   google.fr
Address: 173.194.40.159
```

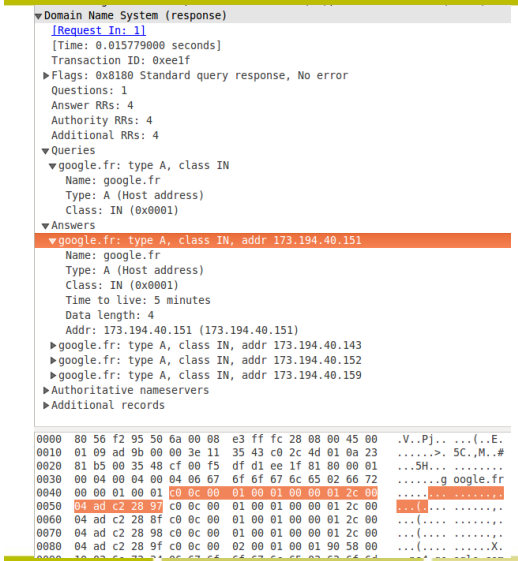


DNS: exemple de protocole binaire

- Requête



DNS: exemple de protocole binaire

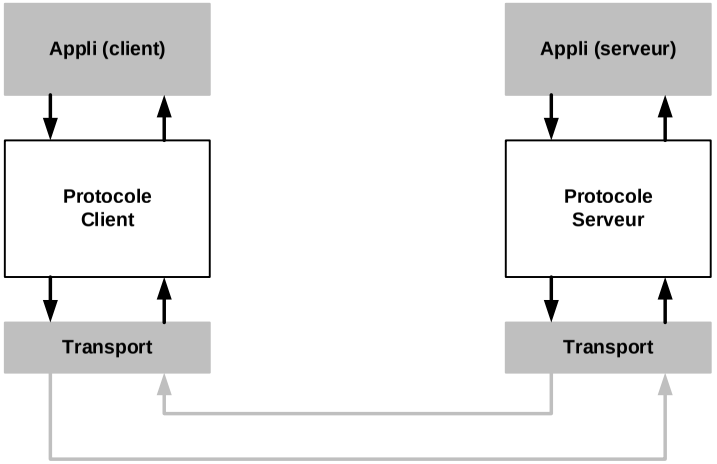


- Réponse
  - Chaine de caractères (string) en ascii
  - le reste est en binaire (hexa)
  - Ex: 173.194.40.151
    ad c2 28 97
- Décodage du début vers la fin
  - en interprétant les champs un à un
  - Format des messages dans RFC1035
  - La réponse utilise le même format que la requête
  - Dans sa réponse, le serveur reprend l'ID du client
    -> utile au client pour formuler plusieurs demandes en //

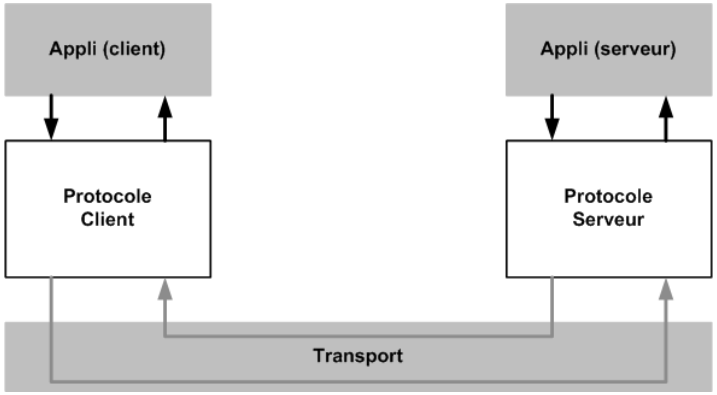
Le protocole du projet

- Interface avec l'application
- Interface avec la couche transport

Environnement du protocole (1/2)

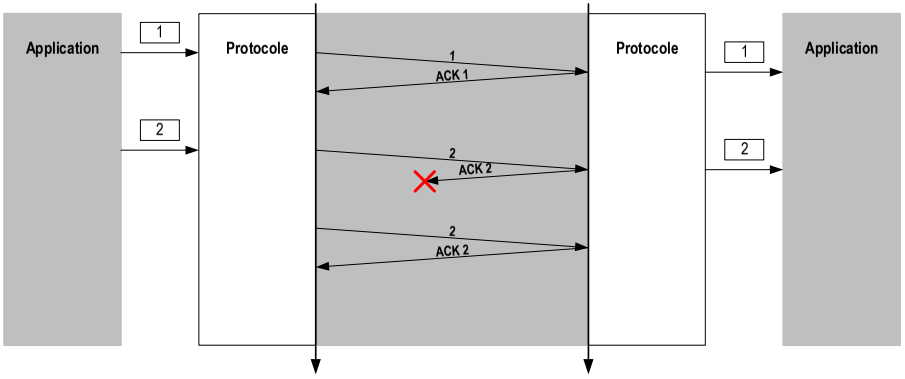


Environnement du protocole (2/2)



Le fonctionnement de la couche inférieure est transparent pour celle d'au dessus

Interfaces



Ce qui est ajouté par une couche à l'émission est retiré par la même couche en réception

09

Protocoles de transport

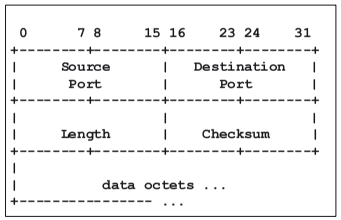
- UDP
- TCP
- Vue utilisateur:
  mode datagramme VS mode flux

Protocoles de transport: introduction

- **Il existe plusieurs protocoles de transport**
  - On utilisera TCP et UDP
  - Mais il y en a d'autres:
    SCTP (multi flux), RTP (temps réel), RSVP (Réservation)…
  - Même niveau dans la pile, mais offrent des services différents
- **Implémentation**
  - Transport implémenté dans le noyau de l'OS
  - => Implémentation logicielle mais modification difficile (recompilation noyau)
  - Interface de programmation standard (API): les sockets (en projet: Twisted est une surcouche au dessus des sockets)

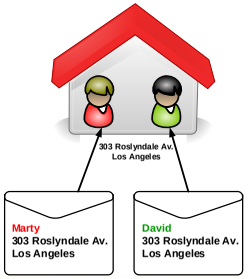
UDP: User Datagram Protocol

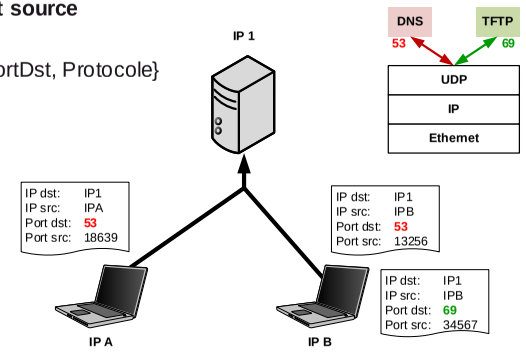
- **UDP: User Datagram Protocol (RFC 768)**
  - Un protocole très simple
- **Mode datagramme**
  - Un envoi pour chaque message
  - Si message trop gros => le décomposer
  - Comme une lettre à la poste:
    si ma lettre dépasse 50g, je la décompose en 2
- **Notion de port source et port destination**
  - Destinataire à l'adresse donnée
  - Couplage fort à IP pour l'adressage
- **Propriétés**
  - PAS besoin d'ouvrir une connexion => rapidité, simplicité
  - PAS de garantie d'acheminement (les messages peuvent de perdre)
  - PAS de garantie de séquence (les messages peuvent arriver dans le désordre)
- **Applications**
  - Temps réel tolérant aux pertes: VoIP, diffusion vidéo temps réel….
  - Protocoles simples: DHCP, TFTP, SNMP…
  - Diffusion à plusieurs destinataires (broadcast, multicast…)



Notion de port

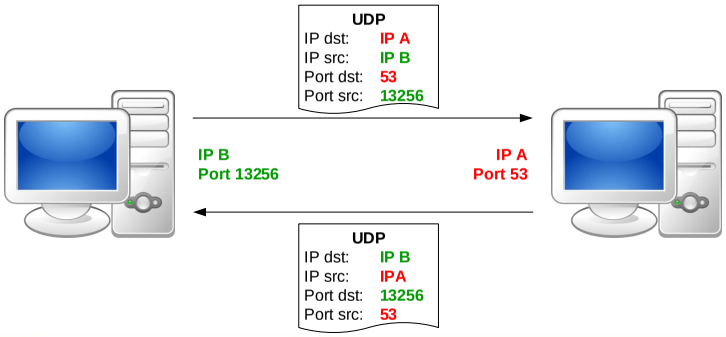
- **Valable pour UDP comme TCP**
- **N° de port = n° d'émetteur / destinataire sur la machine**
  - Plusieurs programmes peuvent utiliser UDP sur la même machine
  - Adresse service: couple (@IP, n°port) => couplage fort à IP
- **Un port destination ET un port source**
- **Identification d'une connexion**
  - {@IPsrc, @IPdst, PortSrc, PortDst, Protocole}



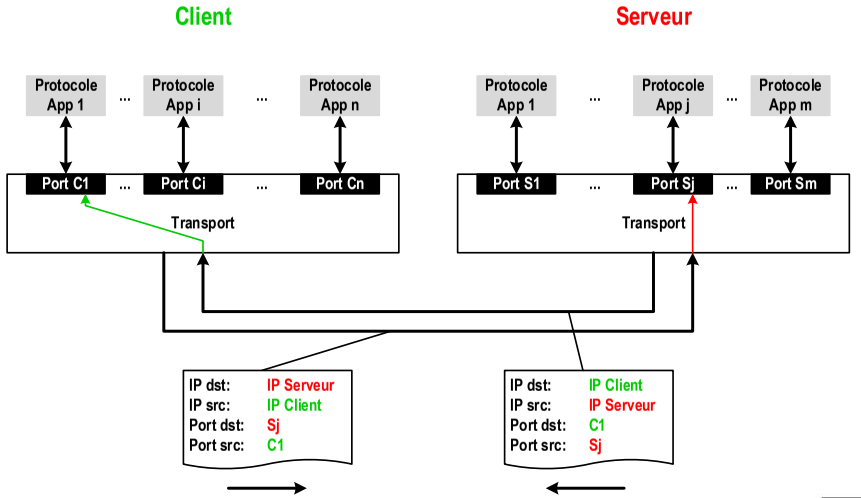


Identification d'une connexion

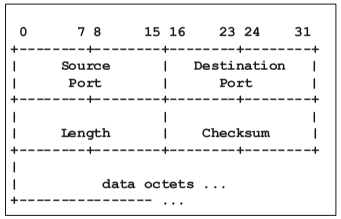
- **Identification par quintuplet**
  - @IP source, @IP destination
  - N° port source, N° port destination
  - Type de protocole de transport (UDP, TCP…)



Interface socket

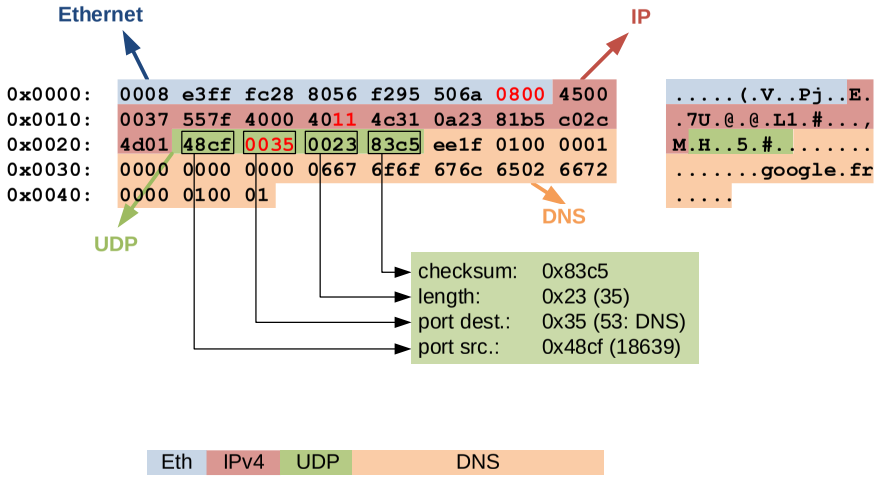


UDP: Détail des champs



- **Port source: 16 bits**
  - Alloué par l'OS de machine source
  - Si client: tiré au hasard parmi les n° dispos
  - Si serveur: N° du service
- **Port destination: 16 bits**
  - Convenu à l'avance
    (ou lors d'un premier échange pour la voie retour)
- **Assignation des n° de ports par l'IANA (RFC 6335)**
  - 0-1023 / 1024-49151: System Ports / User Ports
  - 49152-65535: Dynamic and/or Private Ports
- **Length: 16 bits**
  - Longueur en octets du datagramme complet (entête + données)
- **Checksum: 16 bits**
  - Somme de contrôle portant sur le datagramme et un condensé de l'entête IP
  - => Dépendance à IP

UDP: Désassemblage



TCP: Transmission Control Protocol

Principales propriétés:

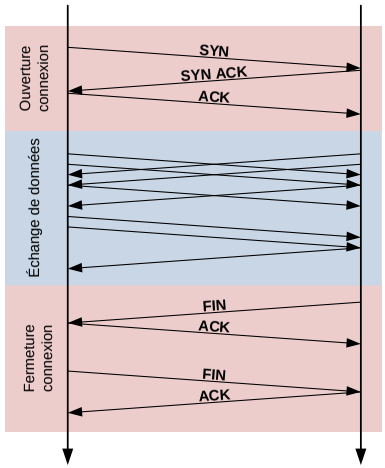
- **Mode connecté**
  - On envoie/reçoit des flux de données
  - Transparence de la taille des paquets
- **Fiabilité**
  - Retransmission en cas d'erreur
- **Garantie de séquencement**
  - Les données arrivent dans l'ordre où elles on été émises
- **Contrôle de flux**
  - Evite de saturer le récepteur
- **Contrôle de congestion**
  - Recherche de l'utilisation « optimale » des ressources disponibles
  - Partage équitable des ressources entre différents flux

TCP = Tuyau point à point

Le système d'exploitation implémente ces propriétés de façon transparente pour les applications

TCP: Mode connecté

- **Connexion = Tuyau point à point**
- **Etablissement de la connexion**
  - Echange de signalisation en 3 étapes (Three-way handshake)
  - SYN / SYN-ACK / ACK
  - connexion / échange de paramètres
  - **Géré par le système d'exploitation**
- **Echange de données**
  - Mode flux => pas de notion de paquet
  - On pousse les données dans le tuyau
  - Données regroupées en « segments »
  - Besoin de repères en réception (framing)
- **Fermeture de la connexion**
  - Gracefull stop
  - 2x 2 étapes: FIN / ACK / FIN ACK
  - **Géré par le système d'exploitation**

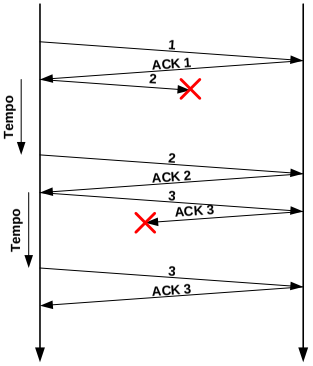


TCP: Fiabilisation

- **Détection d'erreurs**
  - Somme de contrôle
- **Acquittement des segments reçus**
- **Retransmission si erreur ou perte**
- **Go back N**
  - Plusieurs segments émis simultanément
  - Récepteur indique sa position dans la séquence reçue

Fiabilisation: Send & wait

- **Emission d'un paquet**
- **Attente de l'acquittement (ACK)**
- **Si OK**
  - Emission du paquet suivant
- **Si pas d'ACK après un délais donné**
  - Réémission de la donnée
- **Besoin de dimensionner la temporisation de réémission**
  - En TCP: Algo dynamique
  - Basé sur mesure du RTT (Round Trip Time)

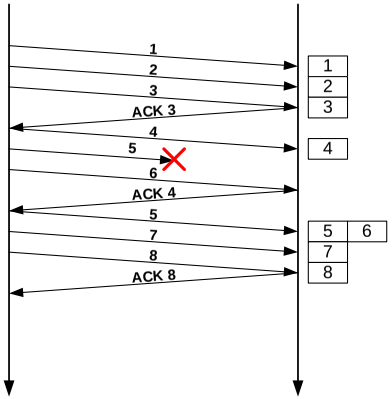


- **TCP utilise un AUTRE mode (Go Back N)**

Exercice: Faut-il numéroter les messages? Et les ACK?

Fiabilisation: Go back N

- **Emission consécutive de plusieurs segments**
  - Meilleure efficacité
  - Nombre max = « taille de fenêtre »
- **Récepteur indique le plus grand n° de segment consécutif reçu**
  - Possibilité de « sauter » des ACK
  - Rq.: TCP indique un N° de séquence et non un n° de segment
- **Réémission à partir du dernier ACK**
- **Le récepteur « bloque » les segments hors séquence**



- **Besoin de dimensionner la fenêtre**
  - -> Contrôle de congestion

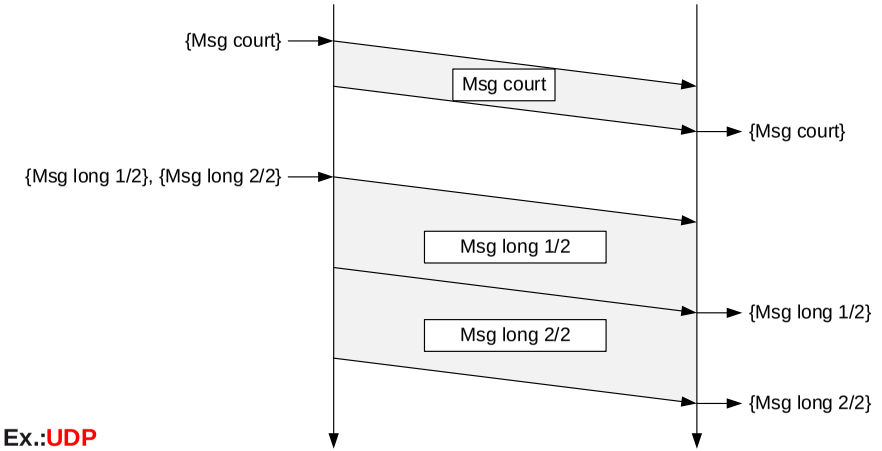
Contrôle de congestion

- **Congestion = saturation du réseau**
- **Différentes causes**
  - Capacité faible / débit nécessaire
  - Panne / perturbations
  - Changement de route…
- **Contrôle de congestion**
  - Adapter le trafic à la capacité du réseau
  - Fluctue en temps réel
- **Fonctionnement en TCP**
  - Hypothèse: En filaire perte de paquet ~= congestion
  - Si pas de perte => augmentation de la taille de fenêtre
  - Si perte => diminution de la fenêtre
  - Algo complexe qui dépasse le cadre de RES302

Contrôle de flux

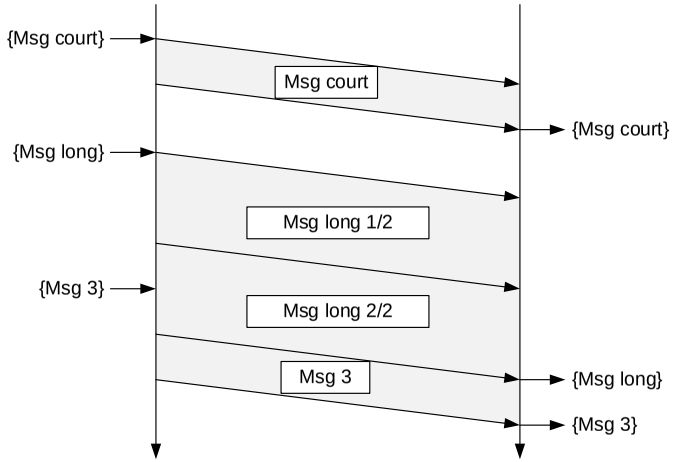
- **Eviter de saturer le récepteur**
  - Capacité de traitement limitée sur le récepteur
  - Capacité mémoire
  - Par ex: petite machine, ou soumise à forte charge
- **Le récepteur indique la taille de données qu'il peut recevoir**
  - L'émetteur ajuste son débit en conséquence

Mode datagramme



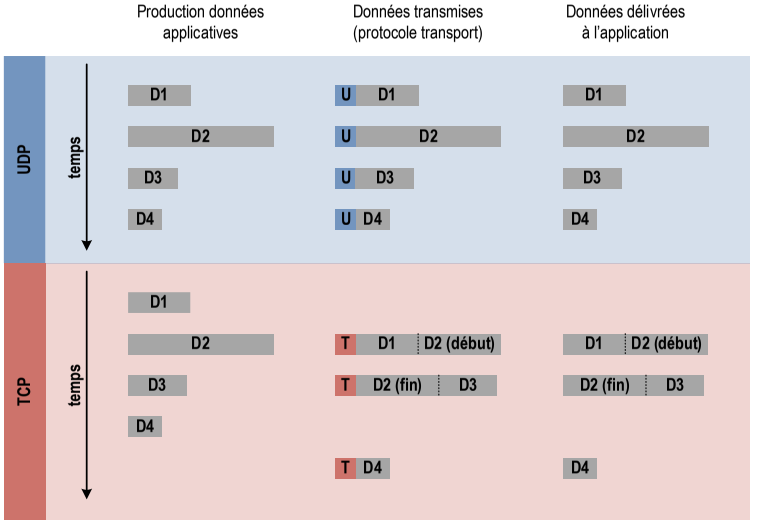
- **Ex.:UDP**
- **1 msg = 1 datagramme**
  - => segmentation par l'application

Mode flux



- **Ex.: TCP**
- **Segmentation par la couche transport**
  - => besoin de délimiter les messages

Comparaison datagramme / flux

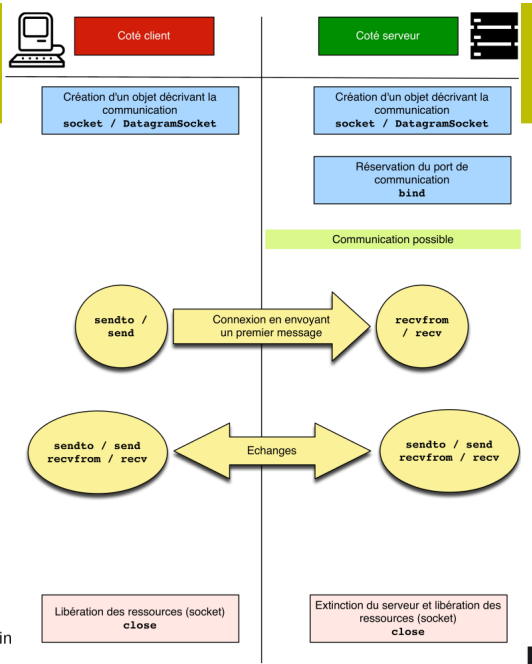


TELECOM Bretagne

Implémentation

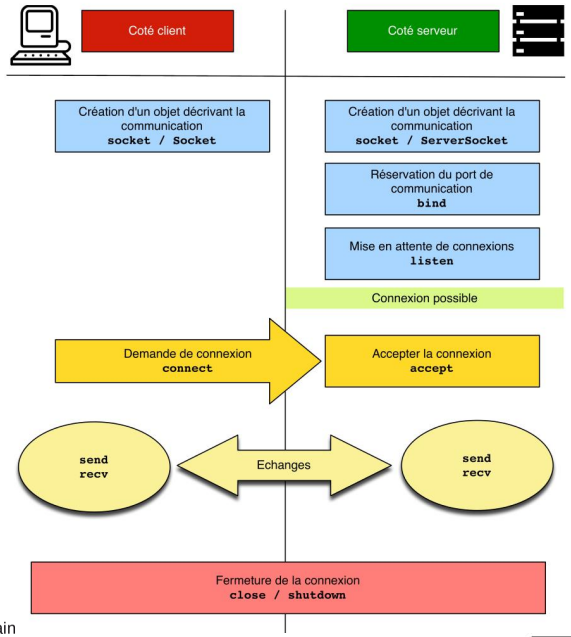
- UDP
- TCP
- Python

Socket UDP



Source:
MOOC Principe des Réseaux de données
C.Chaudet , O.Paul, P.Rolin, G.Texier, L.Toutain
Institut Mines Télécom

Socket TCP



Source:
MOOC Principe des Réseaux de données
C.Chaudet , O.Paul, P.Rolin, G.Texier, L.Toutain
Institut Mines Télécom

Programmation Python

- **Cf. support de cours en anglais (planches 22 à 40)**
  - Bases de python
  - Programmation séquentielle VS évènementielle
  - Notion de process
  - Notion de fork
  - Sockets

TELECOM Bretagne

Représentation des données

- Binaire, Hexadécimal, ASCII, UTF-8…

Représentation des données

Faire la différence entre:

- **Valeur**
  - Quantification d'une grandeur
- **Représentation**
  - Comment écrire cette valeur
- **Codage**
  - Comment la donnée est-elle stockée en machine?

Nombre entier

Valeur	1234	
Représentation	1234	décimal
	0100 1101 0010	binaire
	4 D 2	hexa
Codage	0000 0100 | 1101 0010	

Codage du texte

- **Les machine ne traitent que des nombres!**
- **Un code est associé à chaque caractère**
  - Ex: `A` → 65 (0x41)
    `:` → 58 (0x3A)
- **Codage ASCII**
  - 8 bits (256 combinaisons):
    - 128 fixes + 128 dépendantes de la table de code
- **Codage UTF-8**
  - 1 à 4 octets
    - Identique à ASCII pour les 128 premiers caractères
    - Puis >1octet pour caractères suivants

Chaines de caractères en Python

Valeur	Hello world	
Représentation	'hello world'	string
Codage	b'hello world'	string buffer

Le « string buffer » est bien adapté pour construire un message à envoyer sur le réseau

Chaines de caractères en Python

```
>>> s='spam egg €'
>>> b=s.encode('utf-8')

>>> s
'spam egg €'

>>> b
b'spam egg \xe2\x82\xac'


>>> from binascii import hexlify
>>> hexlify(b)
b'7370616d2065676720e282ac'
```

Chaines de caractères en Python

```
>>> len(s)
10
>>> len(b)
12


>>> s.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't
encode character '\u20ac' in position 9:
ordinal not in range(128)
```

www.telecom-bretagne.eu

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 3
France
Tél. : + 33 (0)2 29 00 11 11
Fax : + 33 (0)2 29 00 10 00

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
France
Tél. : + 33 (0)2 99 12 70 00
Fax : + 33 (0)2 99 12 70 19

Campus de Toulouse
10, avenue Edouard Belin
BP 44004
31028 Toulouse Cedex 04
France
Tél. : +33 (0)5 61 33 83 65
Fax : +33 (0)5 61 33 83 75

TELECOM
Bretagne