

TZ20 - Rapport

PINARD Maxime - LAZARE Lucas

A2015

Sommaire

I. Introduction	2
II. Description des objectifs et énoncés des problèmes	2
III. Solutions choisies	3
1. Client - server : communication	3
2. Server : Restriction d'accès	5
3. Client : Affichage	7
4. Server : Stockage des informations sur les fichiers	8
5. Client : upload / download récursif	9
6. Server : Configuration	10
7. Server, client : Message d'accueil	11
IV. Conclusion	13
1. Améliorations possibles	13
2. Connaissances acquises	14

I. Introduction

What if we programmed server and its client from scratch, instead of using already existing one? That's the question we asked ourselves when we decided to work on this project, because, well, if they are a lot of programs that already does that — such as Apache, for instance, we wanted to understand. How does this kind of program work? How does the client and the server communicates together? Even though we could simply read the source codes of Apache, we decided it was much funnier to program our own, and see what we could do. To do so, we needed to decide :

- How to organise the files in the server.
- What will do what (ie : will that be the server that will sort the files when the client wants to list them?).
- How will they communicate together.
- What should the client be able / not be able to do.

To answer these questions, we spend quite a few time, as these were important questions. For instance, the file's organisation sets a lot about the server's working way.

II. Description des objectifs et énoncés des problèmes

L'objectif du projet TSiD est de coder un serveur et un client fonctionnels en ligne de commande, cela en utilisant la bibliothèque réseau SFML pour communiquer par internet. Les fonctionnalités seront les suivantes :

- L'accès au serveur est restrain aux membres qui possèdent un compte
- Chaque membre a accès a un dossier publique ainsi qu'a un dossier privé. Seul la personne a qui appartient le dossier privé peut y accéder
- Lorsque le client liste les fichiers d'un dossier, la date de création ainsi que le nom du créateur de chaque fichier lui sont aussi indiqués
- Un utilisateur, après avoir uploadé un fichier dans l'espace publique peut le supprimer dans les 24 heures
- Le serveur enregistre des logs de connection
- Les utilisateurs peuvent changer de mot de passe
- Un utilisateur peut ajouter une dscription a un fichier qu'il a uploadé
- Un utilisateur peut créer un compte pour quelqu'un (c'est le seul moyen de créer un compte)
- Le serveur est configurable (ex : autoriser ou non aux utilisateurs l'accès a leur dossier privé)

Aussi, plus tard peut etre, le serveur ne devra pas enregistrer les mots de passe des utilisateur mais un hash de ceux-ci. Les deux programmes (client et serveur) doivent être disponibles sous les deux systèmes d'exploitation : Windows et Linux Pour implémenter ces fonctionnalités nous auront a résoudre les problèmes suivants :

- Comment communiqueront le client et le serveur ? Comment traiter les commandes envoyées par le client ?
- Comment traiter les accès aux dossier ainsi que les restrictions d'accès aux dossier ?
- Comment récupérer la liste des dossiers et fichiers présents dans un dossier ?
- Comment stocker des informations a propos des dossiers et fichiers uploadés ?
- Comment afficher des informations de façon ergonomique/lisible (avec des couleurs) ?
- Comment envoyer des mails via un programme ?
- Comment créer, gérer et utiliser la configuration du serveur ?
- Comment permettre au client de télécharger / uploader un dossier complet ?

Nous devons résoudre chaque problème sur les deux systèmes d'exploitation (Linux & Windows)

III. Solutions choisies

1. Client - server : communication

In order to allow the server and the client to communicate together, we had to implement a '*standardised grammar*'. For instance, it works as the following :

- The client sends '*I want to do this, right here*'
- The server will then try to fulfill the client's wish. If it can do it, it will reply either '*Alright, do so*', or directly give to the client the expected answer. Otherwise it will send to the client the appropriate answer (action prohibited, an error occurred, ...)
- If the client still has something to do (ie : upload a file), it will do it

For the communication at its low level, which is sending and receiving a variable, we will use **SFML/Network** library that provides the functions required to send a variable across the internet, these are : `sf::TcpSocket::send(sf::Packet& packet)` and `sf::TcpSocket::receive(sf::Packet& packet)`, where a packet is an iostream. Here is a pseudo-code example : client wants to upload a file

Pseudo-code, client :

```

1 function uploadAFile( remote_working_directory: string ,
  |   name_of_file_to_upload: string ): boolean
2 |
3 |   server_answer: integer
4 |   file: table of characters
5 |
6 |   sendToServer( upload ) //where 'upload' is associated to a
  |   command code (integer)
7 |   sendToServer( remote_working_directory )
8 |   receiveFromServer( server_answer )
9 |

```

```

10 |   if isPositive( server_answer ) = true then
11 |       |
12 |       |   file <- readfile( name_of_file_to_upload )
13 |       |   sendToServer( name_of_file_to_upload )
14 |       |   sendToServer( file )
15 |       |   return true
16 |       |
17 |   else
18 |       |   return false
19 |   endif
20 |
21 | done

```

Pseudo-code, server :

```

1 | procedure mainLoop()
2 | |
3 | |   directory: table of characters
4 | |   command: integer
5 | |
6 | |   while true do
7 | |       |
8 | |       |   receiveFromClient( command )
9 | |       |   receiveFromClient( directory )
10 | |       |
11 | |       |   switch command
12 | |       |       |
13 | |       |       |   .
14 | |       |       |   .
15 | |       |       |   .
16 | |       |       |   case upload: retrieveAFile( directory )
17 | |       |       |   .
18 | |       |       |   .
19 | |       |   end
20 | |   done
21 | done

```

```

1 | procedure retrieveAFile( directory: table of characters )
2 | |
3 | |   file: table of characters
4 | |   file_name: table of characters
5 | |
6 | |   if ClientIsAllowedToUploadThere( directory ) = true then
7 | |       |
8 | |       |   sendToClient( positive_answer ) //where 'positive_answer' is
9 | |       |   receiveFromClient( file_name )
10 | |       |   receiveFromClient( file )
11 | |       |   write( file_name, file )
12 | |   end

```

```

13 |     else
14 |         |     sendToClient( negative_answer ) //where 'negative_answer' is
           associated to an answer code (integer)
15 |     endif
16 done

```

2. Server : Restriction d'accès

note : './' is the current folder, '../' is the parent's folder.

What we basically want to do here is to avoid people to access files that are not in the server itself, but that might be, for example, at the root of the filesystem. We also want to avoid the access by a member of another member's private folder.

This is easily done by analysing the path sent by the client.

For the first side of the part, we will assume that the client used by the member is the client we programmed and not a custom one, or that the member's client will have a behaviour similar to our own in term of path sending. In particular, our client is never supposed to input a '../' or a './.'. As such, we just look for these sub-strings in the path, and, if we find them, we consider the client's tentative to be prohibited.

For the second part, we will use the server's folders architecture, wich is the following, the downloadable files (given foo and bar to be members) :

```

server execution folder/
    - Public/
        - SomeFolders/
        - SomeFiles.ext
    - Private/
        - foo/
            - SomePrivateFolders/
            - SomePrivateFiles.ext
        - bar/
            - SomePrivateFolders/
            - SomePrivateFiles.ext

```

File listing and client's pathing behaviour :

- The client starts at './', but this is interpreted by the server by './Public'
- If the client wants to list files & folders in './Public', he sees all stuff in './Public' plus an extra folder, wich is './Private/'.
- If the client wants to access './Public/Private', the server should reroot him to './Private/client_id' silently (the client's display will simply shows it is in './Private'; remember './Public' is './')

Given this, we will watch for the server's code, as there is not and should not be any restrictions checking from the client's side. We will also need to modify the main procedure given previously.

Pseudo-code, sever's restrictions (note that `client_id` is known) :

```

1 procedure main()
2 |
3 |     directory: table of characters
4 |     command: integer
5 |
6 |     while true do
7 | |
8 | |         recieveFromClient( command )
9 | |         receiveFromClient( directory )
10 | |
11 | |         if formatPath( directory , client_id ) = true then
12 | | |
13 | | |             switch command
14 | | | |
15 | | | |         .
16 | | | |         .
17 | | | |         .
18 | | |         end
19 | |
20 | |         else
21 | | |             sendToClient( prohibited ) //where 'prohibited' is
22 | | |             associated to an answer code (integer)
23 | |         endif
24 |     done
25 done

```

```

1 function formatPath( directory: table of characters , client_id:
  table_of_characters ): boolean
2 |
3 |     if find( directory , "/"../" ) = true then
4 | |         //if you can find "/"../" in directory
5 | |         return false
6 |     endif
7 |
8 |     if endsBy( directory , "/".." ) = true then
9 | |         //if directory ends by "/".."
10 | |         return false
11 |     endif
12 |
13 |     if startsBy( directory , "/"Private" ) = true then
14 | |         //if directory starts by "/"Private"
15 | |         insert( directory , 9, "/" + client_id ) //inserts /client_id
16 | |         right after /Private
17 |     else
18 | |         directory <- "/"Public" + directory

```

```

18 |     endif
19 |     directory <- "." + directory //The directory is not from the
    |     disk's root, but from the server execution folder
20 |     return true
21 |
22 | done

```

3. Client : Affichage

A readable display is required for the client so that the member may understand its output. For instance, if we make a really simple display like this :

```
Pictures/ sample.mp3 sample.mp4 Movies/ Documents/ Private/
```

This is, indeed, easy to implement. However, that's quite hard to read. Instead, we decided to :

- Display Files / Folders line by line
- Display firstly folders in blue, and then files in green, separated with a blank line
- Sort folders and files in alphabetical order
- Add a column with creation date and another with creator's id

the output displays looks like this :

Name	Creat. Date	Creator
Documents/		
Movies/		
Pictures/		
Private/		
sample.mp3	Fri 06/11/15	foo
sample.mp4	Sat 07/11/15	bar

We also decided to make a 'pacman style' percentage display for downloads/uploads. For this display, we have to adapt pacman's line length to the window's length. Rules are the following :

- if there is less than 3 characters available, '—' are displayed.
- if there is between 11 and 34 characters available, the name and the already transferred byte's number is displayed.
- if there is between 35 and 45 characters available, the name and the number of transferred bytes over the number of bytes transferred is displayed.
- if there is more than 45 characters available, the above is displayed followed by pacman style's percentage display, which takes at most 1/3 of the total available characters.

Color being system dependent, the function is programmed within guards

4. Server : Stockage des informations sur les fichiers

The objective is to store basic informations about uploaded files, such as the upload date and the creator. To do so, we will use the following architecture :

```
server execution folder/
- Public/
    - SomeFolders/
    - SomeFiles.ext
- Private/
    - foo/
        - SomePrivateFolders/
        - SomePrivateFiles.ext

- FilesData/
    - Public/
        - SomeFolders/
        - .SomeFolders
        - .SomeFiles.ext
    - Private/
        - foo/
            - SomePrivateFolders/
            - .SomePrivateFolders
            - SomePrivateFiles.ext
            - .SomePrivateFiles.ext
```

Where *.SomeFolders*, *.SomeFiles.ext*, ... contains informations about *SomeFolders/*, *SomeFiles.ext*, ... The reason why a dot is added is so that we can write description about a folder and the folder itself in the same directory. Note that the creation of directories starting by a dot should not be allowed.

So all we need to do when foo uploads '*file*' in '*directory/*' (where directory is already formatted as seen above) is :

- Write '*file*' in '*directory/*'
- Write the date and foo in *./FilesData/directory/.file*

Here is a pseudo-code example, modifying the retrieveAFile procedure : (No changes to the client)

Pseudo-code, server :

```
1 procedure retrieveAFile( directory: table of characters, client_id:
   table of characters )
2 |
3 |     file: table of characters
4 |     file_name: table of characters
5 |
6 |     if ClientIsAllowedToUploadThere( directory ) = true then
7 |         |
8 |         |     sendToClient( positivite_answer ) //where 'positive_answer' is
   associated to an answer code (integer)
```



```

9 |         receiveFromClient( file_name )
10 |         receiveFromClient( file )
11 |         write( file_name, file )
12 |         writeFileInformations( directory, file_name, client_id )
13 |
14 |     else
15 |         sendToClient( negative_answer ) //where 'negative_answer' is
associated to an answer code (integer)
16 |     endif
17 done

```

```

1 procedure writeFileInformations( directory: table of characters,
  file_name: table of characters, client_id: table of characters )
2 |
3 |     date: table of characters
4 |
5 |     date <- retrieveDate()
6 |     makeDirectory( directory )
7 |     write( date + NEWLINE + client_id, "./FilesData" + directory +
  "." + file_name ) //writes the file's info in
FilesData/directory/.filename
8 |
9 done

```

5. Client : upload / download récursif

In order to allow the client to download and upload a whole folder, we decided to use a recursive approach, working as the following :

- 1 Open the folder and see the first element it contains
- 2a If it is a folder, step back to 1 with this new folder, then step forward to 3
- 2b If it is a file, download/upload it, then step forward to 3
- 3 Step back to 2 with the next element of the folder

There is the recursive download pseudo-code (no changes to the server)

Pseudo-code, client :

```

1 function recursiveDownload( remote_working_directory: table of
  characters ): boolean
2 |
3 |     successful: boolean
4 |     server_answer: integer
5 |     i: integer
6 |     file_list: table of table of characters
7 |
8 |     sendToServer( listFiles ) //Where 'listFiles' is associated
to a command code

```

```

9 |   sendToServer( remote_working_directory )
10 |   receiveFromServer( server_answer )
11 |
12 |   if server_answer = negative_answer then //Where 'negative_answer
    is associated to some answer codes
13 |   |   return false
14 |   endif
15 |
16 |   successful <- true
17 |
18 |   receiveFromServer( file_list )           //file_list now contains a
    list of the files and folders that are within
    'remote_working_directory'
19 |
20 |   for i in [1..file_list.size()]           //from 1 to the number of
    files/folder within 'file_list'
21 |   |
22 |   |   if endsBy( file_list[i], "/" ) then //if 'file_list[i]' ends
    by '/', it is a folder
23 |   |   |   recursiveDownload( remote_working_directory + '/' +
    file_list[i] )
24 |   |   |
25 |   |   else
26 |   |   |   success <- download( remote_working_directory + '/' +
    file_list[i] ) and success // 'success' is true if the download
    was successful AND if it was true before
27 |   |   endif
28 |   done
29 |
30 |   return success
31 |
32 | done

```

6. Server : Configuration

We want to have a configuration for the server. There is a list of the configurable elements for now :

- Generate or not the server folders at next startup
- Create a new user or not at next startup
- Allow members to invite another member (basically : create a new account) or not
- Allow members to write in their private folders or not
- Allow members to download from their private folders or not

To do so, we decided to create an object Config, having the following private variables :

- `user_creation_allowed`
- `private_folder_writing_allowed`
- `private_folder_reading_allowed`

all booleans

This object will be in read-only.

The first two setting do not appear in the object, as they are used only at startup.

A pointer to this object will be passed to each client's thread.

7. Server, client : Message d'accueil

The objective is to allow the the server owner to write a user-adjustable welcome message for all users. In this way we wanted to implement some variables in the welcome message, so we used the syntax `'${variable_name}'`, the available variables are :

- **user** the user name
- **date** the date formatted `'dd/mm/yy'`
- **day** the day formatted `'Mon, Tue, Wed, Thu...'`
- **hour** the hour formatted `'hh :mm'`
- **color** to set the text color from the variable to the next color variable

where **color** can be blue, green, cyan, red, magenta, yellow, white. To put the '\$' symbol simply put '\$\$'

Pseudo-code, server :

```

1 procedure mainLoop()
2 |
3 |   formattedWelcomeMessage( message, client_id )
4 |   SendToClient( message )
5 |
6 |   while ClientConnected() do
7 |       .
8 |       .
9 |       .
10 |   done
11 done

```

```

1 Procedure formattedWelcomeMessage(message: table of characters ,
  client_id: table of characters)
2 |
3 |   read( "WelcomeMessage.txt" , message )
4 |
5 |   foreach ${command} in message do
6 |       |
7 |       switch command

```

```

8
9      case user: replace( $[command], client_id )
10
11     case date: replace( $[command], getDate() )
12                  //formatted 'dd/mm/yy'
13
14     case day: replace( $[command], getDay() )
15                  //formatted 'Mon, Tue, Wed, Thu...'
16
17     case hour: replace( $[command], getHour() )
18                  //formatted 'hh:mm'
19
20     default: //let the client interpret $$ and $[color]
21 end
22 done
23 done

```

Pseudo-code, client :

```

1 Procedure main()
2
3     connectToServer()
4     InterpretWelcomeMessage()
5
6     while ConnectedToServer() do
7         .
8         .
9         .
10    done
11 done

```

```

1 Procedure InterpretWelcomeMessage()
2
3     ReceiveFromServer(message)
4
5     while( not_the_end_of_the_message )
6
7         Print( GetTextUntilSymbol(message, '$') )
8
9         switch( GetNextChar(message) ) //switch char after $
10
11             case '$': Print('$') //there is $$ in the message, put $
12
13             case '[': //there is $[color] in the message, change the
color
14
15                 color <- GetTextUntilChar(']') //color take the value
in $[color]
16
17             if( IsAPrintableColor( color ) ) then

```

```
18 |  
19 |           SetTextColor( color )  
20 |  
21 |         endif  
22 |  
23 |       break  
24 |     end  
25 |   done  
26 | done
```

IV. Conclusion

This project was very instructive and pleasant to work on. The last version of the program for TZ20 is V1 Ranitomeya reticulata, it doesn't contain everything we wanted to implement but the most important functionalities were programmed. We want to continue to work on this program on the future, as a personal interest. The actual version can be used with people you trust (friends for exemple), but security was not tested at all so we don't recommand it for a wide scale use.

1. Améliorations possibles

Some fonctionnality deserve to be developped, but were not because we hadn't enough time, such as :

- **Deleting files**

Members can upload files and folders, but they can't delete them. We thought of a voting system where people could vote for or against the deletion of a file.

- **Passwords**

Passwords are saved in plain text in the server, which is quite unsecure. We'd better save a hash of the passwords.

- **Mails**

We wanted to have a mail system that would be used to :

- Invite guys
- Let a member know an unknown ip tried to access their account
- Inform a member if a vote has started to delete a file they uploaded

- **IP filter**

In order to help people in securing their account, we could add an ip checking. Any unauthorized ip would not be allowed to connect to the account. A mail could be send to allow the member to add the ip to the whitelist. A member should have the possibility to deactivate this service if he likes to.

- **Admin thread**

There is no way to delete a user, a file, or to change the server configuration easily, which is not nice. The problem could be solved by adding an admin console, were you could enter some commands

- **Limit private folder space usage**

The goal of this server is mainly to use the public folder, so it could be nice to set a limit to the private folder, let's say 10Gio, configurable via the server's configuration

2. Connaissances acquises

What did we learn while doing this project ?

- **Teamwork**

We learned to work as a team, in a different way we did in others UV with a team presentation, such as LE03

- **Work organization**

As we were working in autonomy, we had to organize us in order to meet the deadlines. We had to organize both the working order (what to do), and timing (when to do it). That may be very useful in our professional life.

- **GitHub**

GitHub is a powerful tool of version gesture that helps developpers to work together on a project. This could help us as well in our active life, as we may work as developpers, and our company will probably use a similar tool

- **Strings manipulation**

We learned a lot about strings [in C++], and that knowledge can easily be transfered most of others programming languages

- **Cmake**

Finally, we learned about CMake, which is a powerful tool to generate MakeFiles.