TZ20 - Rapport

PINARD Maxime - LAZARE Lucas

A2015

Sommaire

I. Introduction

Et si nous programmions nous même un serveur et son client, en partant de rien, plutôt que d'utiliser un programme préexistant ? C'est la question que nous nous sommes posés lorsque nous avons décidés de commencer ce projet. En effet, bien qu'il existe déjà de nombreux programmes pouvant faire cela — comme Apache, par exemple —, nous voulions comprendre la façon dont ces programmes fonctionnent. S'il est vrai que nous aurions pu simplement lire les sources d'Apache, nous avons pensés qu'il serait plus instructif de le refaire nous-même. Ce faisant, nous devions décider de :

- Comment organiser les fichiers sur le serveur.
- Quelles sont les tâches respectives du client et du serveur.
- Comment ceux-ci communiquerons-t-ils.
- Quelles sont les autorisations du client.

Nous avons longuement réfléchi, afin de répondre à ces importantes questions. Le fonctionnement du serveur est, par exemple, très dépendant de l'organisation des fichiers.

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, en utilisant la bibliothèque réseau SFML pour communiquer par internet. Les fonctionnalités seront les suivantes :

- L'accès au serveur est restreint aux membres qui possèdent un compte
- Chaque membre a accès à un dossier publique ainsi qu'à un dossier privé. Seule le propriétaire de ce dossier 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 téléversé un fichier dans l'espace publique, peut le supprimer dans les 24 heures
- Le serveur enregistre des logs de connexions
- Les utilisateurs peuvent changer de mot de passe
- Un utilisateur peut ajouter une description à un fichier qu'il a téléversé
- Un utilisateur peut créer un compte pour quelqu'un (il s'agit du seul moyen de créer un compte)
- Le serveur est configurable (ex : autoriser ou non aux utilisateurs l'accès à leur dossier privé)

On pourra également ajouter le fait que le serveur ne devrait pas enregistrer les mots de passe des utilisateurs mais un hash de ceux-ci. Les deux programmes (client et serveur) doivent être disponibles sous deux systèmes d'exploitation : Windows et Linux.

Pour implémenter ces fonctionnalités nous aurons à résoudre les problèmes suivants :

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

Nous devons résoudre chaque problèmes sur deux systèmes d'exploitation (Linux et Windows)

III. Solutions choisies

1. Client - serveur : communication

Afin de permettre la communication entre le client et le serveur, nous avons dû mettre en place une *'grammaire standard'*. Celle-ci fonctionne de la façon suivante :

- Le client envoie : *'Je veux faire ça, ici'*
- Le serveur essaiera ensuite de répondre à la requete du client. Si cela est possible, il répondra au client *'D'accord, fait le'*, ou renverra directement la réponse attendue. Sinon, il renverra la réponse appropriée (action non autorisée, une erreur est survenue, ...)
- Si le client doit effectuer une action supplémentaire (ie : téléverser un fichier), il le fera.

Pour la communication de bas niveau (envoyer et recevoir une variable), nous avons utilisé la librairie **SFML/Network**, qui fournit toutes les fonctions nécessaires à l'envoi et la réception de variables par internet. Nous utilisons notamment : **sf::TcpSocket::send(sf::Packet& packet)** et **sf::TcpSocket::receive(sf::Packet& packet)**, où **sf::Packet** est un flux d'entrée/sortie. Algorithme ci-après : le client téléverse un fichier

Pseudo-code, client :

```
function uploadAFile( remote_working_directory: string,
    name_of_file_to_upload: string ): boolean
|
|    server_answer: integer
|    file: table of characters
|
|    sendToServer( upload ) //where 'upload' is associated to a
    command code (integer)
|    sendToServer( remote_working_directory )
|    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
```

Algorithme, serveur :

```
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 |    |    |    case upload: retrieveAFile( directory )
16 |    |    |    .
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( posivite_answer ) //where 'positive_answer' is
       associated to an answer code (integer)
9  |    |    receiveFromClient( file_name )
10 |    |    receiveFromClient( file )
11 |    |    write( file_name, file )
```

```
12 |    |
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 : *'./'* désigne le dossier courant et *'../'* désigne le dossier parent.

L'objectif de cette fonctionnalité est d'empêcher le client d'accéder aux dossiers qui ne sont pas dans l'architecture du serveur même, ou qui sont dans des dossiers parallèles (tel que *'./UsersData'*). Il faut également restreindre l'accès aux dossiers personnels, pour empêcher un membre d'accéder au dossier privé d'un autre membre. Afin de satisfaire ces besoins, nous analyserons simplement le chemin d'accès envoyé par le client.

Pour le premier point, nous supposerons que le client utilisé est bien celui que nous avons programmé, et pas un autre fait par un tiers. En particulier, les chaînes *'/..'* et *'../'* ne sont jamais présentes dans le chemin d'accès fourni par le client. Par conséquent, si l'une d'entre elles est trouvée, l'action du client sera refusée, quelle qu'elle soit.

Pour la deuxième vérification, nous utiliseront l'architecture des dossiers du serveur (avec *foo* et *bar* des membres inscrits) :

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

                                  - bar/
                                        - SomePrivateFolders/
                                        - SomePrivateFiles.ext
```

Listage de fichiers et comportement du client pour le chemin d'accès :

- Le client considère qu'il accède par défaut dans *'/'*, mais cela est réinterprété par le serveur en *'./Public'*.
- Si le client demande à lister les fichiers et dossiers dans *'/'*, il verra les fichiers de *'./Public'*, ainsi qu'un dossier supplémentaire, *'Private/'*.
- Si le client demande l'accès à *'/Private'*, le serveur le redirigera vers *'./Private/client_id'* silencieusement (le client affichera être dans *'/Private'*).

Pour faire ces vérifications, il nous faut modifier le code du serveur. Aucune vérification d'accès ne doit être effectuée du côté du client.

Algorithme : restrictions d'accès (note : client_id est connu) :

```
procedure main()
|
|    directory: table of characters
|    command: integer
|
|    while true do
|    |
|    |    recieveFromClient( command )
|    |    receiveFromClient( directory )
|    |
|    |    if formatPath( directory, client_id ) = true then
|    |    |
|    |    |    switch command
|    |    |    |    .
|    |    |    |    .
|    |    |    |    .
|    |    |    |    .
|    |    |    end
|    |    |
|    |    else
|    |    |    sendToClient( prohibited ) //where 'prohibited' is
     associated to an answer code (integer)
|    |    endif
|    done
done
```

```
function formatPath( directory: table of characters, client_id:
    table_of_characters ): boolean
|
|    if find( directory, "/../" ) = true then
|    //if you can find "/../" in directory
|    |    return false
|    endif
|
|    if endsBy( directory, "/.." ) = true then
|    //if directory ends by "/.."
|    |    return false
|    endif
|
|    if startsBy( directory, "/Private") = true then
|    //if directory starts by "/Private"
|    |    insert( directory, 9, "/"+client_id ) //inserts /client_id
     right after /Private
|    else
|    |    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

Afin que le membre puisse comprendre l'affichage, il faut que celui-ci soit lisible. Il aurait été par exemple très facile de faire un affichage comme celui-ci :

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

Cependant, c'est assez difficile à lire et à comprendre. Nous avons donc opté pour un affichage obéissant aux règles suivantes :

- Afficher les éléments ligne par ligne.
- Afficher en premier les dossiers (en bleu), puis les fichiers (en vert), en les séparant d'une line vide.
- Trier les fichiers et dossiers par ordre alphabétique.
- Ajouter des colonnes indiquant la date de création et le pseudo du créateur.

Le résultat final donne donc :

```
Name          Creat. Date  Creator
Documents/
Movies/
Pictures/
Private/

sample.mp3    Fri 06/11/15 foo
sample.mp4    Sat 07/11/15 bar
```

Nous avons également décidé d'indiquer les pourcentages de téléchargement/téléversement avec un affichage en style *'pacman'*. Afin d'être toujours lisible, l'affichage doit respecter les règles suivants :

- Si moins de 10 caractères sont disponibles pour l'affichage, on n'affiche que des *'–'*.
- Si 11 à 34 caractères sont disponibles, le nom du fichier et le nombre d'octets transférés est affiché.
- Si 35 à 45 caractères sont disponibles, on affiche en plus le nombre total d'octets à transférer.
- Si plus de 45 caractères sont disponibles, on affiche également le pourcentage *'pacmanisé'*, qui prendra au plus 1/3 des caractères disponibles.

4. Server : Stockage des informations sur les fichiers

Il peut être intéressant de stocker des informations sur les fichiers ayant été uploadés, tels que la date de création et l'auteur. Pour ce faire, l'architecture suivante sera utilisée :

```
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
```

Où *.Somefolders*, *.Somefiles.ext*, ... contiennent les informations concernant *SomeFolders/*, *SomeFiles.ext*, ... Un point est ajouté au début du nom du fichier / dossier afin de permettre la création d'un dossier et de sa description au même endroit. Il est à noter que la création d'un dossier commençant par un point doit être interdite.

Ainsi, lorsque *foo* souhaite uploader le fichier *'file'* dans *'directory'*, il suffit de :

- Écrire *'file'* dans *'directory/'*
- Écrire la date et *'foo'* dans *'./Filesdata/directory/.file'*

Algorithme utilisé, en modifiant la procédure retrieveAFile (pas de changement dans le client) :

Algorithme, serveur :

```
procedure retrieveAFile( directory: table of characters, client_id:
    table of characters )
|
|   file: table of characters
|   file_name: table of characters
|
|   if ClientIsAllowedToUploadThere( directory ) = true then
|   |
|   |   sendToClient( posivite_answer ) //where 'positive_answer' is
    associated to an answer code (integer)
|   |   receiveFromClient( file_name )
```

```
|   |   receiveFromClient( file )
|   |   write( file_name, file )
|   |   writeFileInformations( directory, file_name, client_id )
|   |
|   else
|   |   sendToClient( negative_answer ) //where 'negative_answer' is associated to an answer code (integer)
|   endif
done
```

```
procedure writeFileInformations( directory: table of characters, file_name: table of characters, client_id: table of characters )
|
|   date: table of characters
|
|   date <- retrieveDate()
|   makeDirectory( directory )
|   write( date + NEWLINE + client_id, "./FilesData" + directory + "." + file_name ) //writes the file's info in FilesData/directory/.filname
|
done
```

5. Client : upload / download récursif

Afin de permettre au client de télécharger et de téléverser un dossier, nous avons choisi une approche récursive, fonctionnant de la façon suivante :

1 Ouvrir le dossier et lire le premier élément qu'il contient et aller à l'étape suivante.

2a Si c'est un dossier, recommencer l'étape 1 avec ce nouveau dossier, puis aller à l'étape 3.

2b Sinon, le télécharger/verser, puis aller à l'étape suivante.

3 Retourner à l'étape 2 avec l'élément suivant.

Algorithme du téléchargement récursif (pas de changements au serveur) :

Algorithme, client :

```
function recursiveDownload( remote_working_directory: table of characters ): boolean
|
|   successful: boolean
|   server_answer: integer
|   i: integer
|   file_list: table of table of characters
|
|   sendToServer( listFiles )        //Where 'listFiles' is associated to a command code
|   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. Serveur : Configuration

Nous avons également voulu permettre aux utilisateurs de configurer facilement le serveur. Il est pour le moment possible de régler les éléments suivants :

- Générer ou non l'architecture des dossiers au démarrage.
- Créer ou non un nouvel utilisateur au démarrage.
- Autoriser ou non les membres à inviter d'autres membres (et domc créer un nouveau compte).
- Permettre ou non aux membres de téléverser des fichiers dans leur dossier personnel.
- Autoriser ou non les membres de télécharger depuis leur dossier personnel.

Pour ce faire, nous avons ajouté l'objet Config à notre serveur, celui-ci contenant les variables suivantes :

- user_creation_allowed
- private_folder_writing_allowed
- private_folder_reading_allowed

Toutes étant booléennes, et en lecture seule.
Les deux premières configurations n'apparaissent pas dans l'objet, car elles ne sont utilisées qu'au démarrage.
Un pointeur vers cet objet est passé à chaque thread de client.

7. Serveur, client : Message d'accueil

L'objectif est de permettre au propriétaire du serveur d'envoyer un message personnel a chaque utilisateur, lors de leur connexion. Dans cette optique, nous avons implémenté des variable pouvant être utilisé dans le fichier contenant le message, celles-ci ayant la syntaxe *'$[nom_de_variable]'*.
Variables utilisables :

- **user** Nom de l'utilisateur
- **date** Date du jour, au format *'dd/mm/yy'*
- **day** Jour, au format *'Mon,Tue,Wed,Thu,...'*
- **hour** Heure actuelle, au format *'hh :mm'*
- **color** Change la couleur du texte pour que celle-ci corresponde à la variable indiqée

color pouvant être : *'can be blue, green, cyan, red, magenta, yellow, white'*. Pour écrire le symbole *'$'*, il suffit d'écrire *'$$'*.

Algorithme, serveur :

```
procedure mainLoop()
|
|    formatedWelcomeMessage( message, client_id )
|    SendToClient( message )
|
|    while ClientConnected() do
|    |    .
|    |    .
|    |    .
|    done
done
```

```
Procedure formatedWelcomeMessage(message: table of characters,
   client_id: table of characters)
|
|    read( "WelcomeMessage.txt", message )
|
|    foreach $[command] in message do
```

```
6  |   |
7  |   |   switch command
8  |   |   |
9  |   |   |   case user: remplace( $[command], client_id )
10 |   |   |
11 |   |   |   case date: remplace( $[command], getDate() )
12 |   |   |              //formated 'dd/mm/yy'
13 |   |   |
14 |   |   |   case day: remplace( $[command], getDay() )
15 |   |   |              //formated 'Mon, Tue, Wed, Thu...'
16 |   |   |
17 |   |   |   case hour: remplace( $[command], getHour() )
18 |   |   |              //formated 'hh:mm'
19 |   |   |
20 |   |   |   default: //let the client interpret $$ and $[color]
21 |   |   end
22 |   done
23 done
```

Algorithme, 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

Ce projet fut très instructif et intéressant. La dernière version du programme pour l'UV de TZ20 et la V1 *Ranitomeya reticulata*, elle ne contient pas tout ce que nous aurions voulu implémenter mais les fonctionnalités les plus importantes l'ont été. Nous avons la volonté de continuer à travailler sur ce programme. La version actuelle peut être utilisée par des personnes de confiance, mais la sécurité n'ayant pas été testée nous ne recommandons pas son utilisation à plus grande échelle.

1. Améliorations possibles

Certaines fonctionnalités mériteraient d'être plus développées mais ne l'ont pas été par manque de temps, par exemple :

- **Suppression de fichiers**
  Les utilisateurs peuvent téléverser des fichiers et dossier mais ils ne peuvent pas les supprimer. Nous avions pensé à un système de vote ou les utilisateurs auraient pu voter pour ou contre la suppression d'un fichier.

- **Mots de passe**
  Les mots de passe sont enregistrés en clair sur le serveur, ce qui n'est pas assez sécurisé. Il serait préférable de sauvegarder des hash de ces derniers.

- **Mails**
  Nous aurions voulu implémenter un système de mails qui aurait pu être utilisé pour :

  - Inviter des nouveaux utilisateurs
  - Prévenir les membres lorsqu'une adresse IP inconnue essaye d'accéder à leur compte
  - Informer un membres lorsqu'un vote a été lancé pour la suppression d'un fichier qu'il a téléversé

- **Filtre IP**
  Dans le but d'aider les utilisateurs à sécuriser leur compte, nous voudrions ajouter une vérification de l'adresse IP à la connexion. Toute IP non autorisée se verra ainsi refuser la connexion au compte. Un mail pourra être envoyé à l'utilisateur pour ajouter l'IP à la liste des adresses autorisées. L'utilisateur aurait la possibilité de désactiver ce service.

- **Thread Administrateur**
  Il n'y a actuellement aucun moyen de supprimer un compte, un fichier, ou de changer la configuration du serveur directement depuis le serveur. Ce problème pourrait être résolu en ajoutant une console administratrice, où il serait possible d'entrer des commandes.

- **Limiter la taille des dossiers privés**
  Il est impossible pour l'instant de limiter la taille des dossiers privés, ce qui pourrait être intéressant afin d'empêcher les abus. La taille maximum serait également configurable.

2. Connaissances acquises

Qu'avons-nous appris en faisant ce projet ?

- **Travail d'équipe**
  Nous avons appris à travailler en équipe, d'une manière différente que ce que nous avons déjà fait dans d'autres matières avec des présentations de groupe.

- **Organiser son travail**
  Comme nous avons travaillé en autonomie, nous avons dû nous organiser afin de pouvoir travailler efficacement, ce qui nous sera utile pour notre vie professionnelle.

- **GitHub**
  GitHub est un outil de gestion de version puissant, qui aide les programmeurs à travailler de concert sur un projet. Ceci pourra également nous être utile dans notre vie active, où nous serons vraisemblablement amenés à travailler avec un outil similaire.

- **Manipulation de chaînes de caractères**
  Nous avons appris beaucoup concernant la manipulation de chaînes de caractères [en C++]. Cette connaissance est facilement extrapolable à d'autres langages de programation.