Module Info4B Principe des systèmes d'exploitation

Projet 2023 - 2024

Outil de sauvegarde automatique

Erwan Roussel IE4-I92

mars 2024

Table des matières

1 Introduction

Dans le cadre de ce projet, l'objectif est d'implémenter une application, permettant de mettre en évidences l'utilisation de différents concepts des systèmes d'exploitation vus dans ce module. Le projet est construit de façon modulaire et pourrait encore être amené a évolué, il a été développer en plusieurs étapes, amenant a chaque fois de nouvelles fonctionnalité progressive et en testant et en appliquant ces nouveaux concepts.

Le projet a été développer sur le plate-forme Java, et compilé et tester avec un JDK-21. Il est totalement compatible et prévu pour être utilisé sur un système GNU/Linux, mais est également compatible sur un environnement Windows.

Les fichiers source du projet sont disponible sur un répertoire git : *https ://github.com/R-Erwan/OutilSauvegardeAuto*.

1.1 Rappel et analyse du sujet

Sujet 1 : Outil de sauvegarde automatique.
L'objectif final est de permettre à des utilisateurs de garder une sauvegarde de leurs fichiers de façon automatique, et de mettre a disposition plusieurs fonctionnalités pour gérer leurs données.
L'utilisateur devra avoir installé le logiciel sur un poste et renseigner plusieurs informations permettant leur identification.
Les fichiers seront sauvegardé sur un serveur, dans un répertoire dédié.

1.2 Les fonctionnalité principale

Le programme client permet a l'utilisateur d'interagir avec le système au moyen d'un *shell* (interface système) en ligne de commande. Le programme effectue des sauvegarde en explorant les fichier utilisateurs selon un intervalle spécifié. Il est également capable de communiqué avec le serveur pour échanger des informations, authentifié un utilisateur et transférer des fichiers. L'utilisateur peut renseigné des dossiers et des fichiers que le programme doit garder a jour sur un serveur de façon automatique, et sans gêner les interactions entre l'utilisateur et le logiciel. Plusieurs paramètres sont configurables par l'utilisateur pour modifié le comportement du système.

Le programme serveur tourne en permanence, il permet la connexion et la communication entre plusieurs client. Il gère la sauvegarde de plusieurs fichiers dans des répertoires dédié, et leurs archivages. Il permet d'envoyer des données aux utilisateurs et de communiquer avec le programme client au moyen d'un protocole.

2 Analyse fonctionnelle

Avant de définir l'architecture final du projet, il a fallut déterminer comment le système doit fonctionné, anticipé au maximum les problèmes que l'on pourra rencontrer et trouvé les solutions adéquat.

2.1 Les étapes du projet

Le développement du projet c'est vu divisé en 3 grandes parties successives :

Le **programme client** qui permet dans un premier temps :
- D'enregistrer un utilisateur avec ces informations (nom, mot de passe, liste de fichier).
- D'implémenter une file d'attente de fichier a sauvegarder qui peut être récupéré si le programme s'arrête lorsqu'elle n'est pas vide.
- Un système qui permet d'explorer les fichiers du poste de l'utilisateur et de les ajouter si besoin à la file d'attente.
- Un système qui permet de copier/envoyer automatiquement les fichiers a sauvegarder.

Le **programme serveur** :
- Permet d'ouvrir une connexion pour chaque programme client qui souhaite se connecter.
- D'authentifier des utilisateurs et de créer pour chacun un répertoire dédié.
- Permet de sauvegarder les fichiers qu'on lui envoie, et de les archiver si besoins.
- Répondre aux requêtes utilisateur.
- Mise en place d'un protocole pour communiquer.

L' **interface système** :
- Permet a l'utilisateur de communiquer avec le programme client.
- Effectuer des actions sur le système.
- Envoie des requêtes au serveur.
- Permet a l'utilisateur d'observer le fonctionnement de l'application.

De toutes ces étapes, découlent certains problème principaux qu'il faudra prendre en compte pour le développement :
L'utilisation d'**objets sérialisable** pour sauvegarder l'état du programme et les informations utilisateurs.
L'utilisation de **threads** pour permettre une gestion automatique de la sauvegarde des fichiers, et permettre a l'utilisateur d'interagir avec le programme sans gêner les autres opérations.
L'utilisation de **socket** pour communiquer avec le serveur.
Des problèmes de concurrence lors de l'accès a plusieurs ressources (la file d'attente, la connexion avec le serveur).

En plus de toutes ces étapes qui décrivent les fonctionnalité de chaque aspect du projet, une attention sera porté sur la modularité de l'application, la possibilité d'observer le fonctionnement interne du système, et sur la gestion d'erreur lors de l'exécution du programme.

2.2 Architecture de l'application :

Pour mettre en évidence le choix des solutions retenues, observons un schémas global de l'architecture finale de l'application :

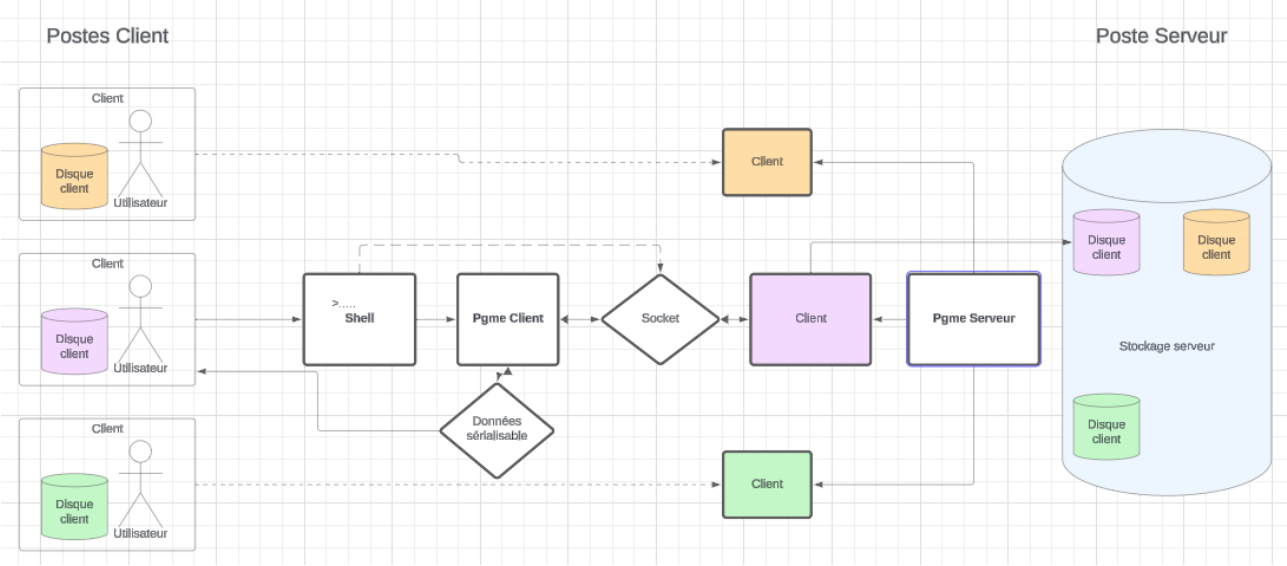

FIGURE 1 – Schema architecture global

Ce schémas illustre globalement l'architecture de l'application, un **utilisateur** communique avec le **programme client** au moyen d'une interface système **shell**. A partir du quel il est capable de rentrer des commandes pour exécuter des actions.
Le programme client gère les **donnée utilisateurs**, le **mécanisme de sauvegarde automatique**, et les **communication** avec le serveur au moyen du **socket**. Le programme client utilise différents **threads** avec des utilités différentes qui seront détaillé juste après.
Pour permettre plusieurs connexion simultané le **serveur** s'emploie a mettre a disposition certaines ressources par client, l'utilisation de **threads** sera également nécessaire.
Les structures de donnée utilisé seront également détaillé par la suite.

Architecture propre au programme client :

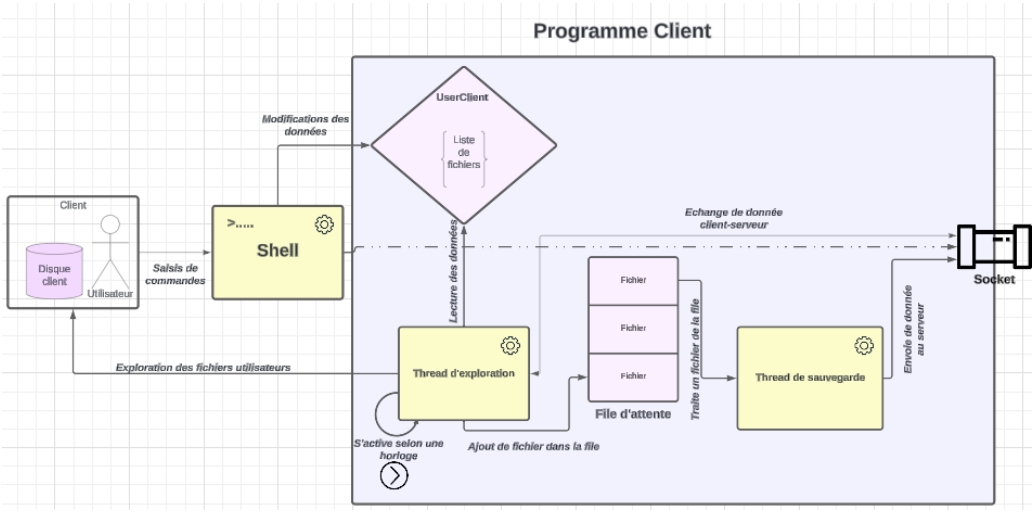

FIGURE 2 – Schema architecture client

La partie cliente de l'application utilise plusieurs threads et plusieurs ressources concurrentes. En comptant l'interface système, trois types de threads sont utilisé.
Le **Shell** de l'application écoute les saisis utilisateurs et effectue des actions sur les composants du programme.
Le **Thread d'exploration explore** les fichiers utilisateurs, soit selon un intervalle de temps qui est configurable par l'utilisateur, soit forcé par l'utilisateur depuis le shell. Fait des **requêtes** au serveur pour comparer les dates des fichiers et déterminé si ils doivent être sauvegarder. Il **ajoute** ainsi des fichier dans la **file d'attente.**
Le **Thread de sauvegarde** est en attente sur la **file** et envoie les fichiers au serveur en continu tant que la file de fichier n'est pas vide.

2.3 L'organisation des threads et des communication entre les ressources :

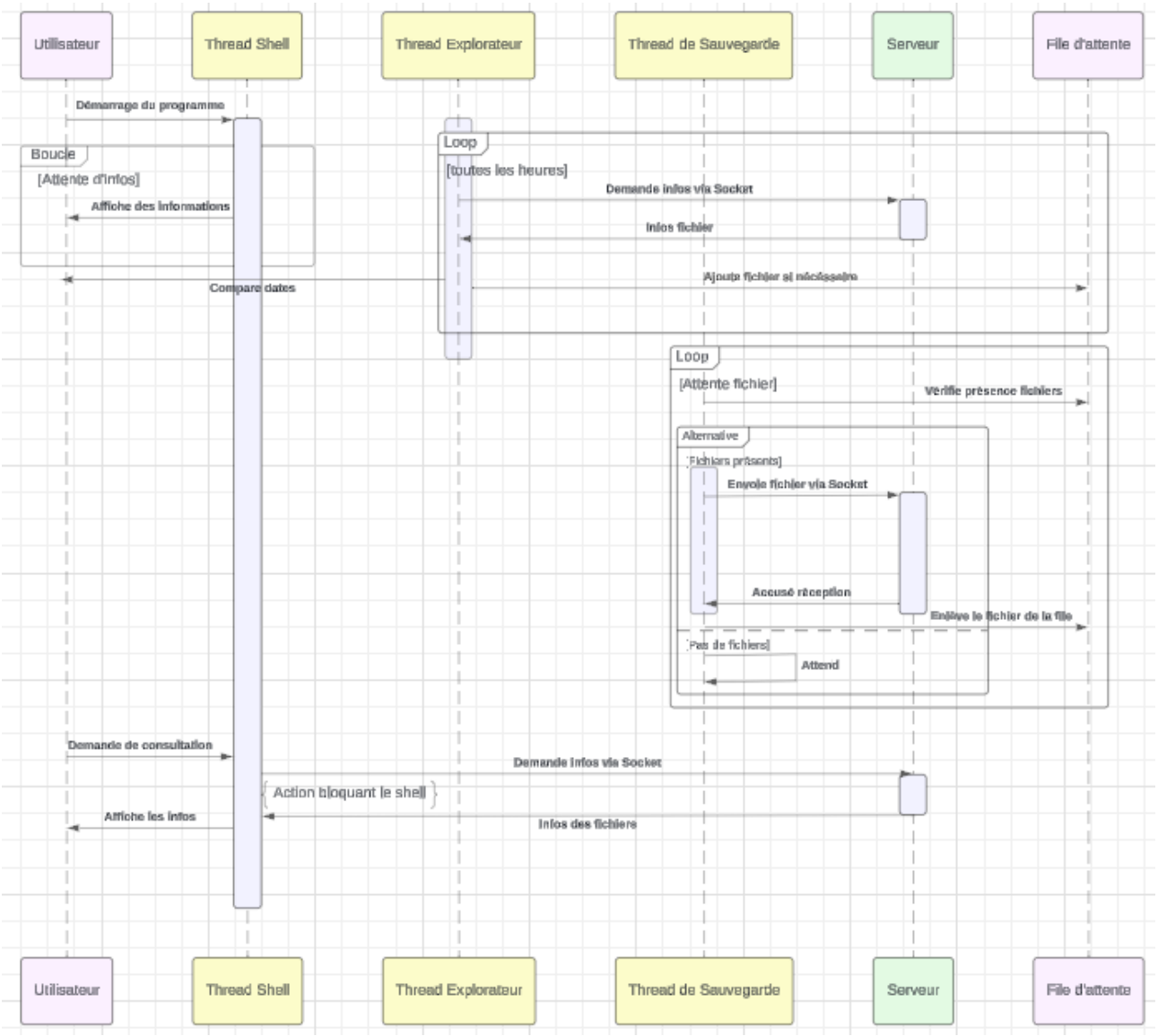

FIGURE 3 – Diagramme de séquence du processus de sauvegarde et de communication

Gestion des threads : L'utilisation et l'ordonnancement des threads est primordiale pour gérer les différentes tâches concurrentes de l'application, telles que l'exploration des fichiers, la sauvegarde des fichiers et la communication avec le serveur. Cela démontre une utilisation du principe de multithreading pour améliorer l'efficacité et la réactivité de l'application.

Communication entre les processus : La communication entre les processus est un aspect essentiel des systèmes d'exploitation modernes, permettant à des programmes distincts d'échanger des informations et de coordonner leurs activités. Cette communication peut prendre différentes formes, notamment les pipes, les sockets, les files d'attente, et les signaux (wait et notify). A partir de *Figure 3*, on déduit que plusieurs de ces mécanismes sont utilisé pour permettre la communication, les **files d'attente** permettent aux processus de partager des données de manière asynchrone, tandis que **les signaux** permettent à un processus d'envoyer des notifications à d'autres processus pour déclencher des actions spécifiques. Les **sockets** offrent une communication bidirectionnelle entre des processus sur des machines distantes via un réseau, c'est ce mécanisme que nous utiliserons pour communiqué avec le serveur.
En utilisant ces mécanismes de communication, les processus peuvent coopérer efficacement, partager des ressources et accomplir des tâches complexes de manière coordonnée

3 Implémentation :

Dans cette partie, sera détaillé plusieurs aspect du projet. Chaque class/module sera introduit par la problématique a laquelle il répond.

3.1 Organisation des packages et classes :

Pour avoir une vue d'ensemble du projet, voici une description de l'organisation des modules de l'application.
Le projet est divisé en deux programme, un pour le client et un pour le serveur.

3.1.1 Le programme Client :

Le programme client est divisé en 3 packages principaux :
- **model** qui représente des modèles de donnée, ce sont les ressources du programme :
  - *UserFiles*
  - *FileWaitingQueue*
  - *SocketConnextion*
- **operator** qui regroupe les classes qui sont des threads et qui effectue des actions sur les donnée :
  - *FileSaver*
  - *FileChecker*
- **interfaceUtilisateur** qui regroupe les classes qui gère l'interface utilisateur :
  - *ShellClient*
  - *CommandHandler*
  - *UserCommandHandler*
  - *ServerCommandHandler*
  - ...

Deux autres packages sont utilisé, **serverProtocol** qui contient la classe **Message** pour communiqué avec le serveur et le package **utils** qui regroupe toutes les classes utilitaires.

3.1.2 Le programme Serveur :

Le programme serveur est plus léger, pour l'instant il ne propose pas d'interface pour administré correctement le serveur.
Le package principale **server** contient les classes :
- *Server*
- *LogHandler*
- *User*
- *ClientConnexion*

Le package de protocole : **serverProtocol** :
- *Command*
- *Message*

3.2 Les données :

3.2.1 Les donnée utilisateurs : *(UserFiles)*

Comment enregistrer un utilisateur avec des informations de connections et une liste de fichier ?

Lorsqu'un utilisateur utilise le programme, il c'est d'abord identifié avec un *nom* et un *mot de passe* qui permettrons de l'authentifier ensuite sur le serveur pour lui donnée accès à ses fichiers personnels.
Du point de vue local, le système a également besoin de garder en mémoire la liste des fichiers que l'utilisateur souhaite sauvegarder. Nous utiliserons une liste dynamique permettant l'ajout et la suppression de données, et la *sérialisation* de toutes les informations. Ce mécanisme sera également utile pour permettre à un utilisateur d'utiliser l'application sur plusieurs machine et de retrouver ces données sur le serveur.

3.2.2 Les paramètres de l'application :

Comment permettre une personnalisation de l'application par l'utilisateur ?

Plusieurs paramètres seront réglable et le programme les utilise pour modifier sont comportement. Les configurations sont stockées dans un fichier *application.properties*, nous utiliserons le même mécanisme pour le programme coté serveur.

Beaucoup des classes du projet utilise ces propriétés, pour récupérer ces propriétés, on utilise deux méthodes utilitaires définie dans la classe **SystemUtils** du package ***utils*** :

```
1  public static Properties getProperties(String pFile) {
2          Properties prop = new Properties();
3          try (InputStream input =
               SystemUtils.class.getResourceAsStream("/" + pFile +
               ".properties")) {
4              if (input == null) {
5                  System.err.println("Le fichier de propriete " + pFile
                       + ".properties n'a pas ete trouve.");
6                  throw new RuntimeException();
7              }
8              prop.load(input);
9          } catch (IOException e) {
10             e.printStackTrace();
11             System.err.println("Erreur lors du chargement du fichier
                   de configuration.");
12             throw new RuntimeException();
13         }
14         return prop;
15     }
```

Les principales options possible sont par exemple :

- **la fréquence de sauvegarde** qui détermine tout les combiens de temps un fichier doit être de nouveaux sauvegardé sur le serveur, exprimé en nombres de jours.
- **La fréquence de rafraichissement**, détermine à quelle fréquence le programme effectue une exploration des fichiers. Initialement toutes les heures.
- D'autres paramètres comme les chemins de dossier systèmes, les informations de connexions au serveur, seront détaillés dans un annexe.

3.2.3 La file d'attente : *(FileWaitingQueue)*

Comment gérer une file d'attente pour gérer les problèmes de déconnexion ?

C'est un mécanisme très important de l'application client, la mémorisation de l'état du programme, et plus précisément la liste des fichiers non encore envoyés pour permettre a l'application de reprendre son travail lorsqu'il redémarre ou se reconnecte.
D'autres problèmes sont également à prévoir, le programme peut être interrompu pendant l'envoie d'un fichier, etc ... Pour implémenter ce mécanisme, nous utiliserons une structure de donnée particulière, basé sur une liste chainé **LinkedListWithBackup** du package **client.model** .
La file fonctionne comme cela :

1. Un élément est ajouté à la file principale
2. Un thread collecte le premier élément qui est alors déplacé dans la file de backup.
3. Si le thread à bien exécuté son traitement sur l'élément, il le supprime de la file de backup.
4. Sinon, c'est qu'il y a eu une erreur, alors l'élément n'est pas supprimé de la file de backup et on peut choisir le traitement à effectuer. Dans notre cas, l'élément sera repositionné en 1ère place dans la file d'attente principale lorsque la file sera dé-sérialisé, donc au redémarrage du programme.

LinkedListWithBackup

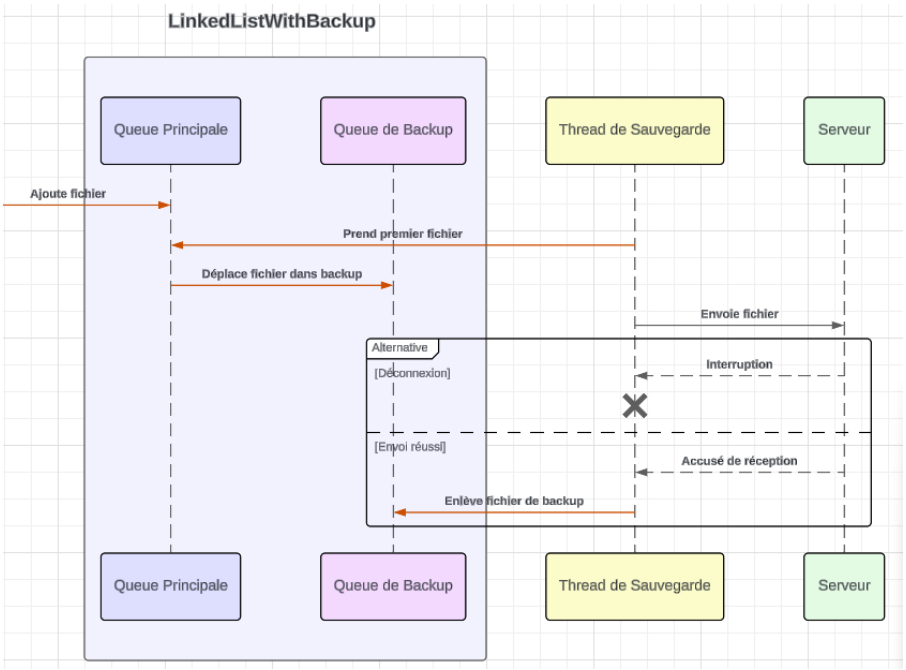

FIGURE 4 – Diagramme de séquence des interactions avec une linkedListWithBackup

La file d'attente est la ressource concurrente principale, toutes les méthodes d'opérations sur la file sont déclaré ***synchronized*** pour éviter les problèmes d'accès concurrent. Dans la première version de l'application, plusieurs thread de sauvegarde était utilisé pour permettre la sauvegarde de plusieurs fichiers simultanément, la ressources est totalement capables de se protéger de ce genre de situations si l'application était amené a ré-évolué dans ce sens.

Une méthode intéressante, la désérialisation de la liste :

```
public class LinkedListWithBackup<E> implements Serializable {
    @Serial
    private static final long serialVersionUID = 1175940438195421030L;
    private LinkedList<E> queue; //File principale
    private LinkedList<E> backUpQueue; //File de backup

    ...

    public synchronized void read(String source) throws IOException,
       ClassNotFoundException {
        try {
            FileInputStream fis = new FileInputStream(source +
               "fwq.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);

            // Deserialise d'abord la file de sauvegarde
            backUpQueue = (LinkedList<E>) ois.readObject();
            // Deserialise ensuite la file principale
            queue = (LinkedList<E>) ois.readObject();
            //On ajoute tout les elements au debut de la file
               principale
            queue.addAll(0,backUpQueue);
            //On vide la file de backup
            backUpQueue.clear();

            ois.close();
            fis.close();
        } catch (FileNotFoundException e){
            throw new FileNotFoundException();
        }
    }
}
```

Pour ce projet nous avons implémenter la classe générique **LinkedListWithBackup** qui implémente ce type de structure prenant n'importe quelle type d'objet en paramètre, et la classe **FileWaitingQueue** implémente cette classe avec des objet de type *java.io.File*.

3.3 Gestion des fichiers :

L'application manipule des fichiers locaux ainsi que des fichiers distants sur un serveur. Elle utilise des mécanismes pour explorer les fichiers locaux, vérifier leur état, et les sauvegarder sur le serveur de manière automatique. Cette gestion des fichiers met en œuvre des principes de manipulation de fichiers et de gestion des entrées/sorties.

Nous allons discuter dans cette parties de comment sont traité les fichiers localement, le cheminement qui permet, à partir d'un fichier présent sur le poste client, qu'il soit envoyé sur le serveur. Le cheminement peut être résumé en 3 étapes :

1. Le fichier est renseigné par l'utilisateur et **ajouter** a sa liste de fichier a garder en sauvegarde (dans ces données utilisateurs). VOIR 3.2.1
2. L'exploration des fichiers utilisateurs et leur ajout dans la file d'attente.
3. Le fichier est envoyé sur le serveur par le thread de sauvegarde.

3.3.1 L'exploration des fichiers : *(FileChecker)*

Comment exploré automatiquement les répertoires de l'utilisateur a des intervalles fixe ? Comment déterminer si un fichier doit être sauvegardé sur le serveur ?

Pour permettre l'exploration des fichiers , le programme utilise un thread qui parcours tout les fichiers renseignéS, regarde la date de dernière modification et demande au serveur si le fichier existe sur le serveur, et sa date de dernière modification, les compare, et si besoin l'ajoute dans la file d'attente.
C'est la classe **FileChecker** qui implements **ThreadFileOperator** du package ***client.operator*** qui s'occupe de cette fonctionnalité.

Règle de l'application :
La liste des fichiers de l'utilisateur peut contenir des dossier avec une arborescence complète de plusieurs sous-dossier et sous-fichier. Nous partons du principe qu'un dossier est considéré modifié dès qu'un de ces fragments l'est. Si un dossier contient plusieurs dossiers mais qu'un seul d'entre eux a été modifié, alors tout le dossier est considéré comme modifié et sera donc re-sauvegarder.

La méthode run du thread :

```
@Override
public void run(){
    long delay = (long) (this.refreshCheck * 3600000L);
    timer.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
            check();
        }
    },0,delay);
}
```

La méthode principale de la classe utilise un *Timer*, qui vas appeler la méthode **check** toutes les X heures.
La méthode check :

```
public void check(){
    ArrayList<File> userFilesList = this.userFiles.getListFile();
       //Liste des repertoires utilisateurs
    for (File file : userFilesList) {
        if(file.exists()){ //Si le fichier existe
            if(needSave(file)){ // Verifie si le fichier doit etre
               sauvegarde
                try {
                    this.fwq.putAndSerialize(file); //Ajoute a la file
                       d'attente
                } catch (IOException e){
                }
            }
        }
    }
}
```

La méthode ***needSave(file)*** appelé dans *check* vérifie si un fichier a besoin d'être sauvegardé Un fichier doit être sauvegardé dans 2 situations :

— Il n'existe pas encore sur le serveur, c'est la 1ère fois.

— Il a été modifié ET sa dernière sauvegarde remonte a au-moins plus que la **fréquence** configuré dans l'application.

La méthode needSave :

```
private boolean needSave(File clientFile) {
try{
    // Requete au serveur.
    long lastModifiedServer = sc.getFileInfo(clientFile.getName());

    // Indique que le fichier n'existe pas sur le serveur
    if(lastModifiedServer == -1) return true;

    //Si c'est un dossier, regarde chaque fichier
    List<File> files;
    if(clientFile.isDirectory()){
        //Liste tout les sous-fichier d'un repertoire
        files = FileUtils.listFiles(clientFile);
        for (File file : files){
            if( ( lastModifiedServer < file.lastModified()) &&
                    (Instant.now().toEpochMilli() - lastModifiedServer
                       > this.freq*3.6e+6)){
                return true;
            }
        }
        return false;
    }
    ...
}
}
```

3.3.2 La sauvegarde/ envoie des fichiers : *(FileSaver)*

Comment envoyer / sauvegarder automatiquement et en continu les fichiers ?

Pour sauvegarder les fichiers, nous utilisons un autre thread **FileSaver** qui implements **ThreadFileOperator** du package **client.operator**.

Le thread collecte un fichier dans la *file d'attente* dès que c'est possible, et l'envoie au serveur en passant par la *socket*.

Un des problèmes qu'il a fallut gérer dans cette algorithme est l'interruption de l'envoie du fichier au serveur. Si le programme interrompt le thread pendant l'envoie, ou qu'il y a eu un problème, on sauvegarde la file d'attente avec le fichier encore dedans. Ce qui permettra de recommencer l'envoie du fichier après un redémarrage.
Il y a également une fonctionnalité qui permet de mettre le thread en pause pour faire des simulations du programme.

La méthode run :

```
@Override
public void run() {
    loop : while (!stop) {

        if(this.pause != -1) this.pause();

        File fileToSave = null;
        try {
            //Collecte un fichier dans la file d'attente
            fileToSave = this.fwq.get();
        } catch (InterruptedException e) {
            //Interruption lorsque le thread est en 'wait' sur la file.
            if(this.pause !=-1){
                this.pause();
                continue loop;
            } else this.stop = true;
        }
        if (!stop) {
            try {
                //Envoie du fichier au serveur.
                sc.sendFile(fileToSave);

                //Supprime le fichier de la backupQueue
                this.fwq.remove(fileToSave);

                //Serialize la File d'attente
                this.fwq.write(getProperties("application").
                getProperty("app.fwqSerFile"));
            } catch (IOException e) {
                //Le fichier a mal ou pas ete envoye au serveur.
                System.err.println("Erreur lors de l'envoie du fichier
                    : "+fileToSave.getName());
                try {
                    //Serialize la file d'attente.
                    this.fwq.write(getProperties("application").
                    getProperty("app.fwqSerFile"));
                } catch (IOException ex) {
                    System.err.println("...");
                }
            }
        }
    }
}
```

3.3.3 La connexion avec le serveur : *(SocketConnection)*

Comment permettre aux module du programme client de communiquer avec le serveur ? Comment gérer des accès concurrent au socket ?

Pour centraliser au même endroit toutes les communications avec le serveur, la classe **SocketConnection** a été définie. Elle représente une *socket* entre le programme serveur et client. Elle implémente toutes les méthodes permettant la communication avec le serveur qui sont déclaré *synchronized* pour garantir un accès exclusif au socket par les threads. Le protocole de communication sera développer dans une prochaine partie mais nous présentons ici quelques méthodes intéressante.
Liste des fonctionnalité proposées par la classe :

- Envoyer un fichier.
- Demander des informations sur un fichier.
- Télécharger un fichier.
- Connecter et créer un utilisateur.
- Demander la liste des fichiers présent sur le serveur.

méthode sendFile :

```
public class SocketConnection{
    private final Socket socket;
    private final ObjectOutputStream soos; //Flux d'ecriture
    private final ObjectInputStream sisr; //Flux de lecture

    ...

     public synchronized void sendFile(File file) throws IOException {
        //Tableaux de couples : CheminRelatif - Fichier
        ArrayList<Object[]> list =  listerFichiers(file);

        for (Object[] pair : list) {
            String chemin = (String) pair[0];
            File fichier = (File) pair[1];

            // Envoie un message qui previens le serveur qu'on vas lui
               envoye un fichier.
            Message saveFileMessage = new
               Message(Command.SaveFile,chemin);
            soos.writeObject(saveFileMessage);

            try (FileInputStream fis = new FileInputStream(fichier)) {
                byte[] buffer = new byte[8192];
                int bytesRead;
                //Tant qu'on est pas a la fin du fichier, on ecrit le
                   buffer sur le flux
                while ((bytesRead = fis.read(buffer)) != -1) {
                    soos.write(buffer,0,bytesRead);
                }
                soos.flush();
            }
        }
    }
```

Lorsque la méthode *sendFile* est appelée par le *fileSaver*, on lui passe un dossier en paramètre. Mais l'envoie sur le serveur se fait fichier par fichier.
Un problème c'est posé pendant le développement, comment conserver l'arborescence du dossier ?.
Pour envoyer un fichier et conserver ses noeuds parents, on envoie un message au serveur avec en paramètre le chemin relatif au parent du fichier, ainsi le serveur sera ensuite capable de reconstruire le dossier avec tout les noeuds.

La méthode downloadFile :

```
1  public synchronized boolean downloadFile(String fileName, String dest)
       throws IOException {
2          //Dossier ou sera copie le fichier telecharge
3          File destFile = new File(dest);
4          if(!destFile.exists()) return false;
5          //Le fichier pour ecrire le fichier telecharge.
6          File copyFile = new File(dest+File.separator+
7          new File(fileName).getName() );
8          copyFile.createNewFile();
9          //Envoie un message pour demander au serveur de nous envoye un
               fichier
10         Message message = new Message(Command.GetFile,fileName);
11         soos.writeObject(message);
12         //Lecture des donnee sur le flux, et ecriture sur le fichier
               copyFile
13         try (FileOutputStream fos = new FileOutputStream(copyFile)) {
14             byte[] buffer = new byte[8192];
15             int bytesRead;
16             while ((bytesRead = sisr.read(buffer)) != -1){
17                 fos.write(buffer, 0, bytesRead);
18             }
19         }
20         //Lit la reponse du serveur apres le telechargement.
21         try {
22             Message response = (Message) sisr.readObject();
23             if(response.command == Command.Error) {
24                 System.out.println(Colors.RED+" SERVER :
                       "+Arrays.toString(response.params)+Colors.RESET);
25                 copyFile.delete();
26                 return false;
27             } else {
28                 return true;
29             }
30         } catch (ClassNotFoundException e) {
31             throw new RuntimeException(e);
32         }
33     }
```

La fonction prend en 1er paramètre le chemin relatif du fichier au dossier utilisateur sur le serveur, et en 2nd, le dossier de destination ou sera copié le fichier. Pour l'instant, il n'est possible de télécharger les fichier que 1 par 1.

Pour résumer, la partie client de l'application permet :

— Pour l'utilisateur de renseigner des **dossier / fichiers** sauvegardés dans une liste. Les dossier renseigné sont considérés comme un seul élément et doivent être maintenus a jour dès qu'un des ses sous-noeuds a été modifié.
— Un**thread** s'occupe en continue **d'explorer** cette liste, et d'ajouter les éléments nécessaire dans la file d'attente en fonction des données sur le serveur.
— Un **thread envoie** en continu les fichiers de la liste au serveur.
— Utilise une **file d'attente** qui permet de résoudre les problèmes de concurrences et de déconnexions.
— Utilise une **Connexion-Serveur** qui communique avec le serveur par une socket, qui gère les problèmes de concurrences. Chaque modules du programme client a un accès concurrent au socket.

3.4 Le serveur :

Pour la partie serveur, il a fallut mettre en place :

— La connexion cotée client pour communiquer. ***SocketConnection***
— L'architecture du système de fichier.
— Un protocole pour la communication client-serveur via un socket. ***Message et Command***
— Le programme serveur qui permet la connexion de plusieurs clients. ***Server***
— Un thread pour gérer les opérations du client coté serveur (authentification, sauvegarde de fichiers, demande d'informations). ***ClientConnexion***

Pour communiquer, le serveur et le client utilise un*socket TCP*, c'est une interface de communication pour la communication réseau. Il s'agit de l'accès à la couche de transport, ici TCP. Il représente un canal de communication bidirectionnel entre un client et un serveur.

3.4.1 Le protocole : (Message et Command)

Comment communique les deux programmes entre eux (client-serveur) ?

Pour communiquer entre sockets TCP on utilise des flots binaires de lectures et d'écritures. L'avantage du protocole TCP est qu'il intègre un mécanisme de retransmission des paquets, ce qui permet d'assurer la transmission fiable et ordonnée des données

Pour unifier les échanges et permettre une communication efficace et cohérente entre le client et le serveur, les communications se feront en utilisant un *protocole* unifié avec des règles lors de l'envoie et de la réception, "par dessus le protocole TCP". Pour ce faire on envoie des **Message** avec des **Command** prédéfinis.

Les classes implémentées pour ce mécanisme sont issue du package ***serverProtocol*** avec les classes **Message** et **Command** (enum). Le package est commun au deux programmes, en effet pour permettre la sérialisation des messages, les deux programmes doivent utilisés strictement la même classe.

Format des messages : *[Command] [...Datas]*
Tout les types de messages (requêtes, réponse, notifications) utilise ce format.

Actions supporté :

— *CREATE USER [pseudo motDePasse]* pour demander la création d'un utilisateur sur le serveur.
— *LOG USER [pseudo motDePasse]* pour connecter un utilisateur déjà existant.
— *SEND FILE [cheminRelatifFichier]* pour demander l'envoie d'un fichier.
— *GET DATE FILE [cheminFichier]* pour demander la date de modification d'un fichier.
— *GET LIST FILE* pour demander la liste des fichiers.
— *DOWNLOAD FILE [cheminFichier]* pour demander le téléchargement d'un fichier.
— *STATE [code d'état]* pour renvoyer une validation.
— *ERROR [erreur]* pour envoyer un message d'erreur.
— *END* pour fermer la connexion.

```java
package serverProtocol;

import java.io.Serial;
import java.io.Serializable;

public class Message implements Serializable {
    @Serial
    private static final long serialVersionUID = -1561957147213392842L;

    public Command command; // Action e effectuer
    public String[] params; //Parametre supplementaire

    /**
     * Constructeur de class
     *
     * @see Command
     * @param command Action a effectue
     * @param params Les parametres de la command
     */
    public Message(Command command, String ... params) {
        this.command = command;
        this.params = params;
    }
}
```

Flux de communication : Le client envoie une requête au serveur pour effectuer une action spécifique. Le serveur répond avec un message de réponse contenant les résultats de l'action demandée et/ou les données demandées.

3.4.2 L'architecture des fichiers du serveur :

Comment organiser les répertoires privés des utilisateurs sur le serveur ?

Pour organiser les données du serveurs et les données des utilisateurs, le serveur utilise une certaine organisation de fichiers.

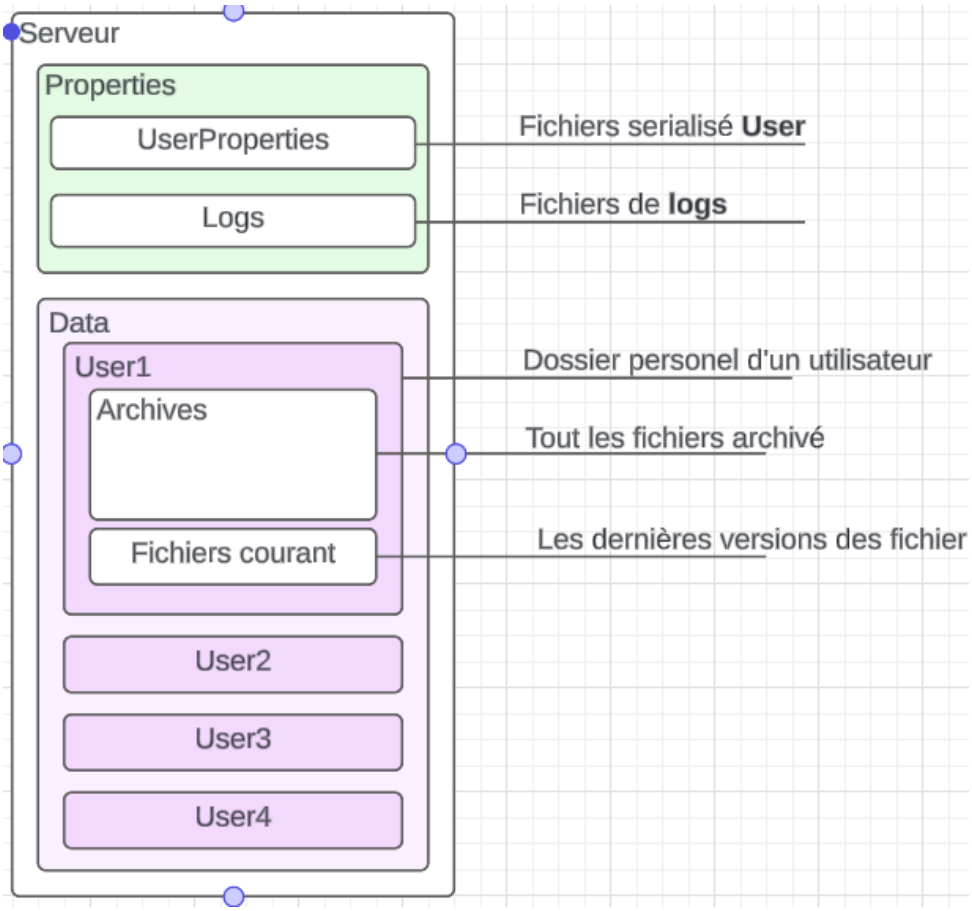

FIGURE 5 – Organisation des répertoires serveur

A la création d'un utilisateur sur le serveur, un dossier spécifique est créé et lui est attribué. Le serveur sauvegarde les informations utilisateurs (nom et mdp) dans des fichiers sérialisés. Le serveur implémente également un mécanisme de journal d'activité (logs).

3.4.3 Le serveur : (Server)

Comment écouter en permanence les connections entrantes ? Comment gérer chaque connections cliente et permettre plusieurs connections simultanée ?

Le programme principale coté serveur tourne en permanence et utilise un *ServerSocket* qui écoute sur une IP local et un port spécifié. A chaque tentative de connexion, on récupère une socket de service et on l'utilise pour connecté le client avec. Chaque connexion cliente est gérée par *1 thread dédié* **ClientConnexion**. Le serveur est configuré pour accepter un certains nombres de client simultanés, ce paramètre est réglables dans le fichier de configuration.

Le Server :

```java
public class Server {
    /** Le port d'ecoute du serveur */
    static int port;
    /** Liste des clients connectes au serveur */
    static ArrayList<ClientConnexion> clientConnexions;
    /** Nombre maximal de clients autorises a se connecter */
    static int maxClients;

    public void start() throws IOException {
        // Initialise la structure du systeme de fichiers du serveur
        this.initFileSystem();

        //Les configurations du serveur a partir du fichier properties
        Properties prop = getProperties("server");
        maxClients =
           Integer.parseInt(prop.getProperty("server.maxClient"));
        port = Integer.parseInt(prop.getProperty("server.port"));

        clientConnexions = new ArrayList<>(maxClients);

        // SocketServer pour ecouter les connexions entrantes
        ServerSocket serverS = new ServerSocket(port);
        System.out.println("Socket ecoute : "+ serverS);

        // Accepte les connexions des clients jusqu'a un nombre
           maximal autorise
        while (clientConnexions.size() < maxClients){
            // Attend une connexion entrante
            Socket socket = serverS.accept();

            // Cree un nouveau clientConnexion pour gerer la connexion.
            ClientConnexion clientConnexion = new
               ClientConnexion(socket, clientConnexions);

            // L'ajoute a la liste
            clientConnexions.add(clientConnexion);
            System.out.println("Nouvelle connection - Socket :" +
               socket);
            System.out.println("Nombres de clients : "+
               clientConnexions.size());

            //Demarre le thread
            clientConnexion.start();
        }
    }
```

3.4.4 Le thread Client : (Client) [la partie serveur de la communication]

Comment gérer les connexions clientes ? Comment authentifier l'utilisateur ? Comment recevoir et envoyer des fichiers ?

La classe **ClientConnection** gère la communication entre le serveur et le client, du coté serveur, en permettant au client d'effectuer diverses opérations comme l'authentification ou la manipulations de fichiers. A partir de la socket, on initialise deux flux en lecture et en écriture.

La méthode run :

```
@Override
public void run() {
    try {
        // Le client doit d'abord authentifier un utilisateur pour
           effectuer d'autres requetes
        if (authenticateUser()) {
            //Log la connection
            logHandler.logMessage(user.getName(), "CONNECTED");
            while (true) {
                //Attend un message
                Message message = (Message) sisr.readObject();

                //Log chaque message recus
                logHandler.logMessage(
                user.getName(),
                message.command +
            ""+Arrays.toString(message.params));

                //Appelle le gestionnaire de commande pour execute
                   l'operation
                handleCommand(message);

                //Commande de deconnexion
                if (message.command == Command.Stop) break;
            }
        }
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
    //Ferme toutes les ressources, socket, flux... .
        closeResources();
    }
}
```

Authentification :

1. Un utilisateur lance pour la première fois l'application coté client. Il doit à ce moment créer un utilisateur avec un nom et un mot de passe.
2. Le programme client essaie d'authentifié l'utilisateur, si un utilisateur avec ce nom existe, et que le mot de passe renseigné correspond, la connexion est validé. Sinon le serveur répond que la connexion a échoué (2 possibilité, l'utilisateur n'existe pas OU le mot de passe est incorrecte).
3. Si la connexion a échouée le programme client tente de créer un nouvel utilisateur avec

ces informations, si l'utilisateur avec ce nom n'existe pas, le serveur créer l'utilisateur sur le serveur et valide la connexion, sinon il répond que l'utilisateur existe déjà.

Le serveur ne permet qu'un seul utilisateur par nom unique, par contre il est possible de se connecter avec un même utilisateur sur plusieurs poste différents et donc d'accéder à ces fichiers sur plusieurs machines.
D'autres mécanisme pourrait être implémenter dans le futur, mais pour l'instant le mécanisme d'authentification est simple.
La méthode authenticateUser :

```
1  private boolean authenticateUser() throws IOException,
       ClassNotFoundException {
2      while (true) {
3          Message message = (Message) sisr.readObject(); //Attend un
               message
4          logHandler.logMessage("TENTATIVE", message.command + "
               "+Arrays.toString(message.params)); //Log le messag
5          System.out.println("Tentative de connection :
               "+message.command+ " "+Arrays.toString(message.params));
6          if (message.command == Command.Stop) {
7              return false;
8          }
9
10         boolean authenticated = switch (message.command) {
11             case CreateUser ->
                   signInUser(message.params[0],message.params[1]);
12             case ConnectUser ->
                   loginUser(message.params[0],message.params[1]);
13             default -> false;
14         };
15
16         if(authenticated){
17             //Envoie un message de validation
18             Message response = new Message(Command.State,"true",
                   message.command.texte);
19             sisw.writeObject(response);
20             sisw.flush();
21
22         } else {
23             //Envoie un message d'erreur.
24             String errorsMessage;
25             if (message.command == Command.CreateUser){
26                 errorsMessage = "L'utilisateur existe deja";
27             } else {
28                 errorsMessage = "Informations utilisateur incorrecte";
29             }
30             Message response = new
                   Message(Command.Error,"false",errorsMessage);
31             sisw.writeObject(response);
32             sisw.flush();
33         }
34         if (authenticated) return true;
35     }
```

La réceptions de fichiers :
Le client envoie un message **[SEND FILE, cheminRelatifFichier]** avec comme paramètre un certains chemin relatif depuis dernier parent que l'on veut conservé.
Par exemple, on souhaite envoyé le fichier chocolat.txt qui fait partie du dossier frigo/sucre/chocolat.txt. Si l'on souhaite conservé cette arborescence on envoie alors le chemin relatif jusqu'au dernier parent que l'on veut conservé, ici ***frigo/sucre/chocolat.txt****. Si on envoie uniquement* ***chocolat.txt****, le fichier sera bien envoyé, mais l'arborescence ne sera pas conservé sur le serveur.*
Puis il écrit les données du fichier sur le flux.
Le serveur :

1. Reçois le message **[SEND FILE, cheminRelatif]**.
2. Si le fichier n'existe pas on le créer avec touts ces dossiers parents dans le répertoire utilisateur.
3. Si le fichier existe déjà, on l'archive dans le répertoire d'archivage.
4. Enfin ont lit les donnée du fichier sur le flux et les écrit dans le fichier.

La méthode saveFile :

```
private void saveFile(String fileName) throws IOException {
        String repDeSauvegarde =
           getProperties("server").getProperty("server.usersData")+
        user.getName()+"/";

        File file = new File(repDeSauvegarde+fileName);
        //Si le fichier n'existe pas sur le serveur, on le cree
        if(!file.exists()){
            file.getParentFile().mkdirs();
            file.createNewFile();

        }
        //Sinon c'est que le fichier existe, donc on doit l'archive
        else {
            archiveFile(new File(fileName)); //On passe uniquement un
               chemin relatif au dossier parent
        }

        //ecriture du fichier et lecture sur le flux
        try (FileOutputStream fos = new
           FileOutputStream(repDeSauvegarde+fileName)) {
            byte[] buffer = new byte[8192];
            int bytesRead;
            while ((bytesRead = sisr.read(buffer)) != -1) {
                fos.write(buffer,0,bytesRead);
            }
        }
        System.out.println("finis de recevoir le fichier");
    }
```

L'archivage des fichiers : Une des fonctionnalité est de permettre l'archivage des fichiers dans un sous-répertoire portant la date de la sauvegarde. *La méthode archiveFile :*

```
 private void archiveFile(File file){
    //Dossier personnel utilisateur (Server/Data/USER)
    String userPath =
       getProperties("server").getProperty("server.usersData")+
            user.getName()+
            File.separator;
    // Dossier d'archive -> Archives/2024/March/24
    String archivePath = "Archives/" +
            LocalDate.now().getYear() + "/" +
            LocalDate.now().getMonth().toString() + "/" +
            LocalDate.now().getDayOfMonth() + "/";

    // newPath -> Server/Data/USER/Archives/2024/March/24/
    String newPath = userPath + archivePath;

    //Creer les dossiers
       Server/Data/USER/Archives/2024/March/24/DossierX/dossierY/...
    File newDir = new File(newPath + file.getParent()+"/");
    newDir.mkdirs();

    //Le fichier a deplacer Server/Data/USER/.../FILE
    File fileToRename = new
       File(getProperties("server").getProperty("server.usersData")+
            user.getName()+
            File.separator+
            file.getPath()
    );

    //Deplace le fichier
    if (fileToRename.renameTo(new
       File(newDir.getPath()+"/"+file.getName() ) ) ) {
        System.out.println(user.getName()+ " -> Fichier : " +
           file.getName() + " archive vers : " + archivePath);
    } else {
        System.err.println("ERREUR lors de l'archivage");
    }
}
```

Pour résumer : La partie serveur de l'application implémente plusieurs aspects cruciaux.

- Mise en place d'une *connexion* client-serveur via les **ServerSocket** TCP.
- Définition de l'architecture du **système de fichiers** et du **protocole de communication**.
- Utilisation d'un thread dédié pour gérer les opérations du client, telles que l'**authentification** et la **manipulation de fichiers**.
- Organisation des fichiers du serveur avec création de **dossiers spécifiques** pour chaque utilisateur et mécanisme de **journalisation** des activités.
- La **réception** et **l'envoi de fichiers** sont gérés de manière similaire, en assurant la conservation de la structure des répertoires.
- Les fichiers peuvent être **archivés** dans des sous-répertoires datés pour la sauvegarde.

3.5 L'interface système :

Pour permettre à l'utilisateur d'interagir avec l'application, on utilise une **interface système** en ligne de commandes. L'interface doit permettre a l'utilisateur :

- D'initialiser l'application.
- D'observer le comportement de l'application.
- D'effectuer des opérations (gérer ces fichiers, demander des informations, régler l'application)
- D'arrêter l'application.

Afin de rendre l'application fonctionnelle, les interactions de l'utilisateur ne doivent pas gêner le déroulement du programme. Le **shell** utilise donc un thread pour permettre à l'utilisateur d'exécuter des actions tout en laissant les autres threads manipulant les fichiers effectuer leurs actions.

3.5.1 Les fonctionnalités :

L'interface permet à l'utilisateur de saisir plusieurs commandes, qui traite différentes parties du logiciel. On utilise un ensemble de gestionnaires de commandes pour traiter les différentes opérations saisies par l'utilisateur. Chaque type correspond à un préfixe :

- **help** -> Afficher l'aide.
- **config** -> Les opérations en rapport avec le fichier de configuration.
- **app** -> Les opérations général sur l'application.
- **process** -> Les commandes relatives à la gestion des fichiers locaux.
- **server** -> Les commandes nécessitant des requêtes au serveur.
- **dev** -> Les commandes de test.

A chaque catégorie correspondra un gestionnaire de commandes, qui peut traiter différentes commandes relatives. La liste complète des opérations sera disponible en annexe.

3.5.2 Le fonctionnement global du shell :

1. Le **shell** de l'application est lancé par le programme client une fois que tout le reste est bien initialisé. Chaque gestionnaire de commande est initialisé au lancement.
2. Le thread tourne en permanence jusqu'à l'arrêt de l'application, il écoute les saisies de l'utilisateur en continue et appelle les différents gestionnaires en leur donnant les informations saisies.
3. Le gestionnaire analyse ce que l'utilisateur à renseigné et effectue les actions demandées si la commande et les paramètres sont corrects.
4. Une fois que le gestionnaire a fini son action, le shell recommence.

```
1  public class ShellClient extends Thread{
2      private final AppClient app;
3      private final Scanner scanner;
4      private final Map<String, CommandHandler> commandHandlers;
5
6      private void initializeCommandHandlers(){
7      commandHandlers.put("help",new HelpCommandHandler());
8      commandHandlers.put("config",new ConfigCommandHandler());
9      commandHandlers.put("app",new AppCommandHandler(this.app));
10     commandHandlers.put("process", new UserCommandHandler(this.app));
11     commandHandlers.put("server", new ServerCommandHandler(this.app));
12     commandHandlers.put("dev",new DevCommandHandler(this.app));
13 }
```

```java
 public void run(){
    System.out.println(Colors.YELLOW + "Bienvenue dans le shell de la
       l'application"+Colors.RESET);
    boolean running = true;
    while(running){
    //Lit la commande de l'utilisateur
        System.out.print("> ");
        String input = scanner.nextLine().trim();

    //Coupe la commandes
        String[] parts = input.split("\\s+");
        String command = parts[0].toLowerCase();

    //Appelle le gestionnaire correspondant
        CommandHandler handler = commandHandlers.get(command);
        if(handler != null){
            //Execute la commande
            boolean commandOut = handler.handleCommand(parts);

            //Si le gestionnnaire renvoie false affiche l'aide
            if(!commandOut){
                handler.displayHelp();
                System.out.println("Arguments incorrect. Tapez 'help'
                   pour afficher la liste des commandes disponible");
            }
        } else {
            System.out.println("Commande inconnue. Tapez 'help' pour
               afficher la liste des commandes disponibles");
        }

        //Pour arreter le thread
        if(input.equals("app stop")){
            running = false;
            System.out.println("Arret du shell correctement effectue");
        }
    }
}
```

4 Test et utilisation :

4.1 Installation de l'application :

Depuis le dépôt Git *https ://github.com/R-Erwan/OutilSauvegardeAuto*, cloner le répertoire *Jar* qui contient deux dossiers *ClientProjet* et *ServerProjet* avec les archives .jar.

Lancer premièrement un serveur :

```
java -jar ServerSauvegardeAuto.jar
```

Le programme serveur va automatiquement initialiser l'architecture de fichier dans un dossier *Server*.

```
Socket écoute : ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8080]
Nouvelle connection - Socket :Socket[addr=/127.0.0.1,port=33260,localport=8080]
Nombres de clients : 1
Tentative de connection : ConnectUser [Toto, 0000]
Tentative de connection : CreateUser [Toto, 0000]
New User create : Toto
```

Puis lancer un client :

```
java -jar ClientSauvegardeAuto.jar
```

De même le client va initialiser un système de fichier dans un dossier Config.
Le client invite ensuite a initialiser l'application en choisissant un pseudo et un mot de passe.

```
Configuration de l'application
Création d'un nouvel utilisateur
> nom : Toto
> mot de passe :0000
Valider vous ces informations : Toto -> 0000 ?
> O/N : o
Configurations :
> Fréquence de sauvegarde (jours): 0
> Valider ces paramètres ? (O/N):o
==========Initialisation de l'application==========
Informations utilisateurs récupéré
Bienvenue Toto
SERVER : réponse suite a l'authentification Informations utilisateur incorrecte
SERVER : réponse suite a l'authentification CREATE USER
Pas de file d'attente précédemment sauvegardé
Tout c'est bien passé, application prête
================================================
Lancement de l'app Client
FileSaver- : en attente sur la FWQ
FileChecker-Explore les fichiers utilisateur...
Bienvenue dans le shell de la l'application
>
```

Les fichiers sources sont également disponibles sur le github. *(Client et Serveur)*.

4.2 Simulations :

Pour effectuer les tests ont a créé un dossier de test avec des configurations de fichier spécifique, mais cela fonctionne avec n'importe quel chemin de dossier sur la machine.

```
mkdir -p Test/Split/Split1 && touch
   Test/Split/Split1/{t1.txt,t2.txt,t3.txt}
mkdir -p Test/Split/Split2 && touch
   Test/Split/Split2/{t1.txt,t2.txt,t3.txt}

dd if=/dev/zero of=F500.txt bs=100M count=5
dd if=/dev/zero of=F1000.txt bs=100M count=10
```

4.2.1 Envoie de fichier au serveur :

Client : On ajoute des fichiers à sauvegarder et on force une sauvegarde.

```
> help
Liste des commandes disponibles :
- help : Afficher l'aide.
- config : Opérations sur la configuration de l'application.
- process : Opérations sur les fichier à sauvegarder
- server : Opérations nécessitant des requires au serveur
- dev : Opérations de tests
- app stop : Arrête le programme et le shell
- app check : Lance un check des fichiers client

Pour utiliser une commande, saisissez son nom suivi des arguments nécessaire
s.
Exemple : command1 arg1 arg2.
> process addFile Test/Split
Le fichier : Split a été ajouter a la liste
> app check
FileChecker-Explore les fichiers utilisateur...
Split Ajouter a la FWQ
> FileSaver- s'occupe de copier le fichier : Test/Split
FileSaver- fini de copier Split
FileSaver- : en attente sur la FWQ
server list
[+] Toto
    [+] Archives
    [+] Split
        [+] Split1
                |-- t1.txt
                |-- t2.txt
                |-- t3.txt
        [+] Split2
                |-- t1.txt
                |-- t2.txt
                |-- t3.txt
```

On oberve que le dossier **Split** est ajouté à la file d'attente, et que le **FileSaver** a envoyé le fichier sur le serveur. On peut demander au serveur de nous renvoyer une liste de nos fichier (*server list*).

On modifie un des fichier :

```
nano Test/Split/Split1/t1.txt
```

On re-force une sauvegarde pour tester si les fichiers sont bien renvoyé au serveur et bien archivé.

```
> app check
FileChecker-Explore les fichiers utilisateur...
Split Ajouter a la FWQ
> FileSaver- s'occupe de copier le fichier : Test/Split
FileSaver- fini de copier Split
FileSaver- : en attente sur la FWQ
server list
[+] Toto
    [+] Archives
        [+] 2024
            [+] APRIL
                [+] 7
                    [+] Split
                        [+] Split1
                                |-- t1.txt
                                |-- t2.txt
                                |-- t3.txt
                        [+] Split2
                                |-- t1.txt
                                |-- t2.txt
                                |-- t3.txt
    [+] Split
        [+] Split1
                |-- t1.txt
                |-- t2.txt
                |-- t3.txt
        [+] Split2
                |-- t1.txt
                |-- t2.txt
                |-- t3.txt
```

On observe que les anciens fichiers ont été archivés dans un répertoire spécifique, et que les nouveaux fichiers sont venues remplacer les anciens dans le répertoire Toto. Si on vérifie le contenue des fichiers on vérifiera bien que le fichier *Archives/2024/APRIL/7/Split/Split1/t1.txt* est vide et que le fichier *Toto/Split/Split1/t1.txt* a les modifications.

4.2.2 Test sur la file d'attente en cas de déconnexion

```
> dev pauseSave 500
> FileSaver- en pause pendant : 500s
process addFile F500.txt
Le fichier : F500.txt a été ajouter a la liste
> process addFile F1000.txt
Le fichier : F1000.txt a été ajouter a la liste
> app check
FileChecker-Explore les fichiers utilisateur...
F500.txt Ajouter a la FWQ
F1000.txt Ajouter a la FWQ
> dev showFwq
FileWaitingQueue{queue=[F500.txt, F1000.txt], backUpQueue=[]}
> app stop
Arrêt de l'application
Arrêt du FileChecker
Arrêt du FileSaver FileSaver-
FileSaver- pause finis
File d'attente sauvegardé : FileWaitingQueue{queue=[F500.txt, F1000.txt], ba
ckUpQueue=[]}
Arrêt de la connexion au serveur
Arrêt de l'application Client...
Arrêt du shell correctement effectué
erwan@ErwanR:/mnt/c/Users/erwan/Documents/Jar/ClientProjet$ java -jar Client
SauvegardeAuto.jar
```

On met en pause le *thread de sauvegarde* pour pouvoir observer le mécanisme. Deux fichier sont ajoutés **F500** et **F1000**. Le *thread d'exploration* est forcé et ajoute ces deux fichiers à la *file d'attente*. Après on coupe le programme.

```
erwan@ErwanR:/mnt/c/Users/erwan/Documents/Jar/ClientProjet$ java -jar Client
SauvegardeAuto.jar
==========Initialisation de l'application==========
Informations utilisateurs récupéré
Bienvenue toto
SERVER : réponse suite a l'authentification LOG USER
File d'attente récupéré : FileWaitingQueue{queue=[F500.txt, F1000.txt], back
UpQueue=[]}
Tout c'est bien passé, application prête
==================================================
Lancement de l'app Client
FileSaver- s'occupe de copier le fichier : F500.txt
FileChecker-Explore les fichiers utilisateur...
Bienvenue dans le shell de la l'application
> dev showFwq
FileWaitingQueue{queue=[F1000.txt], backUpQueue=[F500.txt]}
> FileSaver- fini de copier F500.txt
```

Ici on peut observer que au redémarrage du programme, la file d'attente est récupérée et le programme reprend son exécution et le **FileSaver** commence à sauvegarder les fichiers. En parallèle le **FileChecker** effectue un *check* comme le programme viens de redémarrer, mais ne rajoute pas une 2nd fois les fichiers dans la file car ils y sont déjà.
On observe également le mécanisme de double file, lorsque le programme est entrain d'envoyer le fichier **F500**, celui ci est dans la *backupQueue* et le fichier **F1000** attend qu'on l'envoie dans la file *queue*.

5 Conclusion :

5.1 Que retenir ?

Le production finale propose une application cliente destiné à l'utilisateur qui après avoir installer l'application lui permet de sauvegarder automatiquement ces fichiers sur un serveur et d'intéragir avec l'application au moyen d'une interface système. Et une application serveur pour gérer ces fichiers.

En conclusion, ce projet pour le module "Principes des Systèmes d'Exploitations" a permis d'approfondir notre compréhension des concepts fondamentaux des systèmes d'exploitation à travers une implémentation concrète. En mettant en œuvre un outil de sauvegarde automatique, nous avons exploré divers aspects des systèmes d'exploitation, tels que la gestion des processus, la communication inter-processus, la gestion des threads, ainsi que la manipulation des fichiers et des entrées-sorties.

À travers les différentes étapes du projet, nous avons pu identifier et résoudre divers défis liés à la conception et à l'implémentation d'une application en JAVA. De la modularité de l'architecture à la gestion des erreurs et à la concurrence des ressources, chaque aspect du projet a été étudié pour garantir un fonctionnement le plus fluide et fiable possible.

L'analyse fonctionnelle du sujet a permis de définir les fonctionnalités principales de l'application, tandis que l'architecture détaillée nous a guidés dans la mise en place des solutions techniques appropriées. L'organisation des threads et la communication entre les processus ont été particulièrement cruciales pour assurer la réactivité et l'efficacité de notre application.

En fin de compte, ce projet nous a fourni une expérience précieuse dans le domaine de la programmation, en nous confrontant à des problématiques réels et en nous obligeant à trouver des solutions.

5.2 Les améliorations possibles :

Au cours du développement du projet, beaucoup d'idée n'ont pas pus être approfondie et implémenté, par exemple d'autres fonctionnalités concrètes, ou sur les questions d'optimisations des performances et des ressources, qui reflète bien les principe s vues dans ce module.

Sur ce dernier point on peut cité quelques idée d'évolutions :

- Pour optimiser la recherche des fichiers de l'utilisateur ont aurait pu implémenter une structure de donnée comme une *hashTable* ce qui augmenterais l'efficacité de l'application lorsque la quantité des dossiers utilisateurs augmente.
- Utiliser des flux séparé pour la communication réseau, ce qui permettrais a l'utilisateur d'intéragir avec le serveur sans gêner l'envoie des fichiers, actuellement lorsqu'un thread occupe la socket pour envoyé les données d'un fichier, les autres thread comme le **shell** ou le **FileChecker** doivent attendre avant de pouvoir faire des requêtes.
- Ajouter un mécanisme style ***watchDogs*** pour gérer les erreurs qui peuvent survenir si un des fichiers de l'utilisateur n'existe plus. Pour précisé, si un fichier est actuellement ajouté dans la file d'attente, mais que avant son envoie, l'utilisateur déplace ou supprime ce fichier, l'envoie ne pourra pas s'effectuer et le fichier sera remis en boucle dans la file d'attente à chaque redémarrage du programme. Un mécanisme de surveillance sur la file d'attente pourrais donc être implémenté.

Des idées de fonctionnalités :

— Ajouter une limite de stockage par utilisateur pour ne pas surchargé le serveur et archivé des fichiers inutilement, un mécanisme de destructions des fichiers pourrait être également implémenté.
— Permettre à l'utilisateur de spécifié un *Super Dossier*, où tout les dossier / fichier présent dedans serait automatiquement ajouter, sans que l'utilisateur est a les renseigner par l'interface.
— Ajouter une interface système pour gérer et surveillé le serveur.
— Permettre aux utilisateurs de partager leurs fichiers avec d'autres utilisateurs.

6 Bibliographie :

— *https ://www.jmdoudoux.fr/accueil.html*, site sur la programmation Java.
— *Progammer en Java*, livre de Claude Delannoy.
— *https ://github.com/EricLeclercq/Info4B*, le github des exemples du cours et des sujet de TP.
— Le cours lié a ce module.
— *https ://docs.oracle.com/javase/8/docs/api/*, La documentation java, javadoc sur le site d'oracle.
— *https ://anisfrikha.developpez.com/*, Pour des compléments sur le package *java.io* et sur la gestion d'erreur.
— *https ://www.lucidchart.com/blog/fr/types-de-diagrammes-UML*, pour faire les shémas, nottament les diagrammes UML.
— *https ://fr.overleaf.com/*, Pour rédiger ce rapport en laTeX.

7 Annexes :

7.1 Les commandes de l'interface système :

— *help* -> Afficher les commandes d'aides.

— *config info* -> Affiche les configurations de l'application.
— *config update key val* -> Modifie une configuration, nécéssite un redémarrage de l'application.

— *app stop* -> Arrête l'application.
— *app check* -> Lance un check forcé des fichiers a sauvegardé.

— *process listFile* -> Détails des fichiers répertorié a sauvegardé.
— *process clearFile* -> Rénitialise (vide) la liste de fichier.
— *process addFile filePath* -> Ajoute un fichier ou un répertoire a la liste de fichier.
— *process delFile filePath* -> Supprime un fichier ou un répertoire de la liste de fichier.
— *process showFile* -> Affiche des détails sur un répertoire en particuliéde la liste.

— *dev showFwq* -> Affiche l'état de la file d'attente de fichier.
— *dev pauseSave nbPause* -> Met en pause le thread de sauvegarde pendant 'nbPause' milisecondes

— *server listFile* -> Affiche la liste de tout les fichiers personnel présent sur le serveur.
— *server download filePath copyFolderPath* -> Télécharge un fichier sur le serveur et le copy dans un repertoire.

7.2 Les fichiers .properties

application.properties

```
app.firstLaunch=false
app.freq=0
app.fwqSerFile=Config/SerFiles/fwq.ser
app.host=localhost
app.port=8080
app.refresh=1
app.userName=toto
app.userSerFile=Config/SerFiles/
```

server.pr0operties

```
server.port=8080
server.maxClient=10
server.serUser=Server/Properties/UserProperties/
server.usersData=Server/Data/
server.logs=Server/Properties/Logs/
```