 

**Année scolaire 2022 – 2023**

**Projet de fin de semestre**

D-SYNC

Application de synchronisation de répertoires

**Nom :** DOUMIT Geryes, PROUX Marc

**Formation :** 1A – IR – ENSISA

Table of Contents

[Introduction 2](#_Toc134379096)

# Introduction

Dans le cadre de ce projet Java de fin de semestre, nous avons dû réaliser une Application de synchronisation de répertoires (local et LAN) avec son interface graphique.

Notre objectif était de créer une application grâce à laquelle les utilisateurs pourront synchroniser des répertoires tant sur un seul ordinateur que sur deux connectés au même réseau.

Le cahier des charges était le suivant :

* Cloner un dossier D et un dossier D’
* Synchroniser les deux dossier (si un élément du premier change, les mêmes modifications s’appliquent au second)
* Intégrer une fonctionnalité supplémentaire pouvant rendre l’application plus attractive
* La synchronisation peut se faire en réseau
* L’application doit disposer d’une interface graphique

Vous pouvez retrouver le code source, sa documentation ainsi que ce rapport sur notre dépôt [GitHub](https://github.com/Marc-Proux/d-sync).

# Interface graphique

Geryes s’est principalement occupé de l’interface graphique. Voici ses commentaires :

Il n’y a pas grand-chose à dire vis-à-vis de l’interface graphique, sachant qu’après avoir compris le fonctionnement de Swing, c’était juste une question d’esthétique et d’implémentation des fonctions de Dsync.java et Network.java (on en parle après dans ce rapport).

Cependant, j’ai quand même quelques points à noter :

## Threading :

Pour avoir une interface graphique et un programme qui tourne en même temps, il faut absolument utiliser les fonctions Thread de java. En effet, si on ne met pas la synchronisation dans une thread, l’interface graphique se bloque jusqu’à ce que la synchronisation s’arrête, ce que n’arrive jamais parce qu’on est sensés l’arrêter avec cette interface graphique.

C’est pour cela que les classes Dsync.java et Network.java héritent de java.lang.Thread et redéfinissent la méthode run() de Thread. On appelle ensuite la méthode start() de Thread pour lancer leurs méthodes run() dans une nouvelle thread.

## Thèmes de l’application :

Notre fonctionnalité bonus ! En effet, nous avons mis 4 thèmes par défaut pour l’application, tous basés sur la couleur de l’arrière-plan de cette dernière. De plus, vous pouvez choisir un thème personnalisé avec la couleur de votre choix !

N’hésitez pas à vous amuser avec le sélectionneur de thème, c’est assez cool de voir quelles couleurs rendent bien et lesquelles rendent l’application immonde !

## Sauvegarde des données utilisateur :

Dans le répertoire d’exécution de l’application, je crée un fichier texte dans lequel je sauvegarde certaines données utilisateurs.

A chaque appui du bouton « Start Syncing » je sauvegarde :

* Les chemins rentrés
* L’adresse IP et le numéro de port rentrés
* La couleur du thème de l’application

Je charge ces données à la réouverture de l’application pour éviter de devoir retaper les mêmes informations ou de rechanger le thème à chaque fois.

# Synchronisation locale

Geryes s’est occupé de la synchronisation en local. Voici ses commentaires :

Pour la synchronisation en local, j’utilise principalement les chemins des fichiers. En effet, il y a plusieurs fonctions dans Dsync.java, DateAndName.java et Directory.java pour aider à la synchronisation, comme nous pouvons le voir dans le diagramme des classes suivant :

Voici la manière dont je raisonne :

## Première synchronisation :

Au début de la synchronisation, je souhaite éviter de perdre des éléments. La fonction firstSync de Dsync.java (appelée deux fois parce qu’elle donne priorité à l’un des deux dossiers) fait en sorte que les deux dossiers se retrouvent avec les mêmes éléments.

Par exemple, si le dossier A contient a.txt et b.txt, et que le dossier B contient b.txt et c.txt, à la fin de la première synchronisation, les deux dossiers contiendront a.txt, b.txt et c.txt, avec la version la plus récente de b.txt.

## Synchronisation normale :

Ensuite, nous devons gérer la suppression d’un fichier dans un des dossiers : nous devons le supprimer dans le second. Pour cela, J’utilise la fonction syncAndDelete qui va donner priorité à un dossier, et donc supprimer un fichier de l’autre s’il n’est pas présent dans le dossier prioritaire, ou le copier s’il est présent dans le dossier prioritaire.

Par exemple, si le dossier A contient a.txt et b.txt, et que le dossier B contient b.txt et c.txt, si on donne priorité au dossier B, les deux dossiers synchronisés contiendront b.txt et c.txt, avec la version la plus récente de b.txt.

Enfin, nous devons savoir quelles fonctions appeler et à qui donner la priorité. C’est là que DateAndName.java intervient ainsi que la variable lastState.

La variable lastState est une liste d’objets DateAndName, qui ne sont autre que des objets qui contiennent le nom ainsi que la date d’un fichier. Je l’utilise pour la simple et bonne raison que je ne souhaite que comparer le nom et la date des fichiers pour savoir s’ils diffèrent ou non.

Je donne donc priorité au dossier qui a été modifié par rapport au dernier état synchronisé (lastState), et tout ça se passe dans ma fonction syncLastState de Dsync.java.

Si les deux dossiers ont été modifiés entre deux synchronisations, on refait un fisrtSync pour éviter de perdre des données.

## Les fonctions de Directory.java :

Directory.java a un but très simple, c’est d’effectuer des opérations sur les dossiers. Il existe trois méthodes dans cette class, tous en statique :

* **copyDirectory(source, destination)** qui copie un dossier et tout ce qu’il contient dans le chemin de destination.
* **deleteDirectory(chemin)** qui supprime un dossier et tous les éléments qu’il contient.
* **lastModifiedDate(fichier)** qui renvoie la date du fichier le plus récent présent dans un dossier, pour savoir quel dossier est plus récent.

## Interactions de l’utilisateur :

Je vérifie avant chaque synchronisation si les dossiers à synchroniser existent encore, sinon l’utilisateur est notifié et la synchronisation s’arrête jusqu’à ce que le dossier soit remis, ou bien jusqu’à ce que l’utilisateur réinitialise le tout.

Il existe d’autres options pour l’utilisateur à travers le GUI, notamment mettre la synchronisation en pause.

Voici un diagramme de séquence qui indique l’interaction entre l’utilisateur, l’interface graphique, et l’objet qui synchronise les fichiers :

# Synchronisation réseau

La synchronisation en réseau de notre application à pour but de connecter deux ordinateurs entre eux et d’échanger les fichiers présents dans leur répertoire de travail afin de les synchroniser et ce, tant que l’application est lancée.

Pour ce faire, j’ai dû créer 3 classes :

* Une classe abstraite : Network
* Deux classes qui héritent de Network : Client et Server

## Network :

La classe Network est la classe « mère » de Client et Server. C’est elle qui contiendra les fonctions nécessaires au bon fonctionnement de la partie réseau.