Rapport Systeme Distribué

Sujet 7 - Gestion des comptes bancaire

Assia Hammani

Groupe b TP3

13 janvier 2023

Table des matières

1 Introduction

1.1 Presentation du sujet

Le sujet consiste à développer un système de gestion de comptes bancaires à l'aide d'Akka. Un banquier est en charge de plusieurs comptes, et travaille dans une banque. Lorsqu'un client se présente a la banque pour déposer ou retirer de l'argent, la banque doit transmettre la demande du client au banquier responsable de son compte. Un mécanisme de persistance doit être mis en œuvre en liaison avec un SGBD, afin de pouvoir retrouver l'état des comptes des clients même en cas d'arrêt du système.

Le sujet vise à développer un système de gestion de comptes bancaires en utilisant Akka, un modèle d'acteur pour la programmation concurrente et distribuée.
On modelisera un système dans lequel le banquier sera reponsable de plusieurs comptes bancaires. (Chaque client est associé a un compte bancaire specifique)et le banquier est charge de gerer les operations financieres c'est-à-dire les dépôts et retraits pour ces comptes. Concernant la communication entre la banque et les clients, lorsqu'un client se presente a la banque pour effectuer une operation la banque doit acheminer la demande du client au bon banquier qui est responsable du compte concerne.
On mettra en place un mecanisme de persistance en liaison avec un système de gestion de bases de données(SGBD). Ce mécanisme permettra de stocker les données des comptes des clients de manière a les retrouver même en cas d'arrêt du système. Il assure donc la sauvegarde et la recuperation des données de manière durable, garantissant la cohérence des données malgré l'arrêt du système.

2 Analyse du sujet

Chaque banquier est responsable de plusieurs comptes bancaires, c'est une entité distincte et indépendante. Dans ce contexte de programmation en Akka chaque banquier serait généralement représenté par un acteur distinct. Chaque acteur dans Akka est une entité isolée et concurrente qui communique avec d'autres acteurs via des messages. Chaque banquier étant responsable de ses propres opérations sur le compte qu'il gère on modélisera chaque banquier n tant qu'acteur distinct. Cela permet d'assurer l'isolation et la gestion indépendante des opérations bancaires pour chaque compte. Ces acteurs pourraient recevoir des demandes de dépôt, de retrait, ce qui permettrait une gestion parallèle des opérations bancaires. De même pour les clients.

2.1 Structure de données et algorithmes

J'utilise principalement des classes pour encapsuler les données spécifiques et les références d'acteurs pour communiquer entre eux. Les méthodes et les logiques spécifiques à chaque acteur sont définies dans les méthodes createReceive. Les structures de données sont principalement constituées de variables membres et de classes internes pour représenter des messages spécifiques.
Un objet Scanner pour lire les entrées utilisateur est utilise ddanq App.java. Il est utilisé pour obtenir l'ID du client à partir de la console. Dans la calsse ClientActor.java on a la variable 'solde' utilise pour stocker le solde du client. Elle est mise à jour en fonction des messages reçus.
Enfin on a des classes internes DemandeDepot,DemandeRetrait,Transaction et SoldeMessage).

- DemandeDepot et DemandeRetrait contiennent les informations necessaires pour une demande de depot ou retrait.
- Transaction contient le type de transaction et le montant.
- SoldeMessage contient le solde du client et son Id.

Les classes internes permettent d'encapsuler les données nécessaires lors de la communication entre les acteurs.

CLIENT			
idClient	nom	solde	idBanquier
1	Dupont	10	1
2	Richard	15	2
3	Escoffier	3	3
4	Hammani	120	2
5	Boyard	1000	3
6	Ronaldo	50000	1

BANQUIER	
idBanquier	nom
1	Ducreux
2	Martin
3	Petit

FIGURE 1 – Table BDD

Un mécanisme de persistance a été mis en place, il permet d'enregistrer les données de l'application dans un emplacement sûr de manière a ce qu'on puisse les retrouver même en cas d'arrêt du système. On utilisera un SGBD pour stocker les données.Pour cela j ai cree dans la base de donnée 2 tableaux CLIENT et BANQUIER.

J'utilise le JDBC (Java Database Connectivity) pour établir une connexion avec une base de données Oracle et effectuer des opérations de base de données à partir de l'application Java. JDBC permet de fournir une interface Java standard pour interagir avec des bases de données relationnelles. Cela facilite l'intégration de la base de données Oracle dans notre application et offre un moyen robuste d'exécuter des requêtes SQL.
La connexion à la base de données Oracle est réalisée en utilisant une chaîne de connexion, un nom d'utilisateur et un mot de passe. La classe DriverManager est employée pour gérer la connexion, elle fournit ainsi un point d'entrée pour interagir avec la base de données.

```
String url = "jdbc:oracle:thin:@butor:1521:ENSB2023";
        String user = "ah975865";
        String mdp = "ah975865";
    }}
 Connection con = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            try {
                con = DriverManager.getConnection(url, user, mdp);
                Statement stmtExecute = con.createStatement();

} catch (ClassNotFoundException e) {
            System.err.println("Impossible de charger le pilote jdbc:odbc");
             e.printStackTrace();
            System.exit(99);
        }
```

Les requêtes SQL sont construites et exécutées à l'aide de l'interface PreparedStatement. Dans le code, une requête SQL est utilisée pour vérifier l'existence d'un ID spécifique dans la table "CLIENT". L'utilisation de PreparedStatement sécurise la requêtes en évitant les attaques par injection SQL.

```
    String query = "SELECT * FROM CLIENT WHERE idClient = ?";
        PreparedStatement stmt = con.prepareStatement(query);
        stmt.setInt(1, id);
        ResultSet rs = stmt.executeQuery();

Statement statement = con.createStatement();
ResultSet resultat = statement.executeQuery("SELECT * FROM CLIENT WHERE idClient = " + id);
```

J'utilse les requetes pour connaitre egalement le nom et le solde du client qui souhaite effectuer une transaction. En effet le client entre son id dans la console, grace a une requete sql on recupere les informations du client

dans la base de donnee.
Une gestion appropriée des exceptions JDBC a été mis en place notamment avec l'utilisation des blocs try-catch-finally pour traiter les erreurs de connexion, d'exécution de requête et autres exceptions liées à la base de données. Cela assure une expérience utilisateur robuste et informative en cas de problème.

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    try {
        con = DriverManager.getConnection(url, user, mdp);
        Statement stmtExecute = con.createStatement();
        } catch (ClassNotFoundException e) {
            System.err.println("Impossible de charger le pilote jdbc:odbc");
             e.printStackTrace();
            System.exit(99);
        }
```

Les ressources JDBC telles que les connexions, les statements et les result sets sont correctement fermées avec la methode close() pour éviter les fuites de mémoire et libérer les ressources du serveur de base de données.

```
    statement.close();
    resultat.close();

    con.close();
    System.exit(99);
```

2.2 Paradigme retenu (sa justification)

Le paradigme utilisé est le modèle d'acteur. C'est un paradigme de programmation concurrente qui permet de modéliser des systèmes distribues en termes d'acteurs.
Le modèle d'acteur est un paradigme permettant d'organiser des applications pour introduire le parallélisme des traitements et éviter la gestion de la concurrence, il n'y a pas d'état partagé mais des acteurs autonomes échangeant des messages. Le mode de communication est asynchrone. Ses entités centrales sont les acteurs qui sont spécialisés dans une tâche. Pour demander à un acteur d'exécuter une tâche, il faut lui envoyer un message qui lui servira à savoir quel traitement réaliser. Les acteurs sont isolés les uns des autres. Un acteur peut avoir un état qui le fera réagir différemment aux messages. Les acteurs ont des ressources affectes, chaque acteur a un accès exclusif a ses ressources. Les acteurs communiquent entre eux via des messages cela leur permet également de demander l'exécution de tâches.
Dans le cas du système de gestion de comptes bancaires, trois acteurs seront nécessaires :

- Un acteur Client
- Un acteur Banquier
- Un acteur Banque

2.3 Décomposition en sous-problèmes éventuels

Interaction avec la base de données :

- pouvoir etablir une connexion a la base de donnée oracle et ainsi verifier les informations dans la table correspondante.
- pouvoir mettre a jour le solde du client apres la transaction est un probleme non resolus. En effet il est recommandé d'éviter d'accéder directement à la base de données depuis un acteur. Cela peut introduire des problèmes de concurrence et ne suit pas le modèle d'acteurs où la communication se fait par messages.

De plus je ne peux pas utiliser le systeme d'envoi de message avec App.java car ce n'est pas un acteur.Le nouveau solde apres une transaction n'est donc pas mis a jour, on garde les valeurs initial a chaque nouvelle execution.

Communication entre acteur :

- mettre en place la communication entre les acteurs pour l'envoi de messages
- pouvoir assurer la synchronisation et la cohérence des transactions entre les acteurs

Terminaison :

- pouvoir gérer la terminaison propre du système d'acteurs et des ressources telles que les connexions à la base de données

Gestion des erreurs et exceptions :

- implémenter la gestion des erreurs pour des situations telles que des ID non trouvés, des transactions impossibles du a un solde insuffisant, etc...

3 Spécification des classes principales ou des fonctions

3.1 Classes principales

- **App.java** : Cette classe est le point d'entrée de l'application. Elle initialise le système d'acteurs, crée les acteurs nécessaires (banqueActor, banquierActor, clientActor), et gère le processus global d'exécution.
- **BanqueActor** :Cet acteur agit comme une interface entre le clientActor et le banquierActor. Il reçoit les demandes de dépôt et de retrait du clientActor, les transmet au banquierActor, et renvoie les résultats au clientActor.
- **BanquierActor** :Cet acteur est responsable de la gestion des transactions bancaires. Il reçoit les demandes de dépôt et de retrait du BanqueActor, effectue les calculs nécessaires sur le solde, et renvoie les résultats.
- **ClientActor** :Cet acteur représente un client de la banque. Il interagit avec le banqueActor pour effectuer des transactions, recevoir des mises à jour de solde.
- Les classes **DemandeDepot** et **DemandeRetrait** presentes dans BanqueActor sont utilisées comme messages pour transmettre les demandes de dépôt et de retrait entre les acteurs.
- **Transaction** :Cette classe presente dans BanquierActor représente une transaction bancaire avec un type (dépôt ou retrait) et un montant associé.
- **SoldeMessage** :Cette classe presente dans ClientActor est utilisée comme message pour transmettre le solde du client après une transaction.

3.2 Fonctions

- **createReceive()** :Dans BanqueActor, il definit le comportement de l'acteur BanqueActor en réagissant à différents types de messages, notamment les demandes de dépôt, les demandes de retrait, et les résultats des transactions.

  Dans BanquierActor, il definit le comportement de l'acteur BanquierActor en réagissant aux demandes de dépôt et de retrait, et en renvoyant les résultats au client.

  Dans ClientActor, il definit le comportement de l'acteur ClientActor en réagissant aux messages de solde et de transactions, et en mettant à jour son propre solde.
- **checkDatabaseForId(Connection con, int id)** :dans App.java, il vérifie si un ID donné existe dans la base de données en exécutant une requête SQL.

4 Architecture logicielle détaillées de l'application

4.1 Vue globale

On a un système d'Acteurs (Akka), on utilise le modèle d'acteurs pour la gestion concurrente des opérations et on a 3 acteurs principaux : ClientActor,BanqueActor et BanquierActor. On a egalement une Base de Données (JDBC) qui gere une connexion à une base de données Oracle pour stocker et récupérer les informations des clients. Enfin on a une interface Console (Scanner) qui gere l'interaction avec l'utilisateur pour entrer les informations et effectuer des transactions.

5 Evaluation / Comparaison des performances

Le programme s'execute sans soucis particulier, pour la premiere saisi il recherche dans la base de donnee et renvoie les informations du client. Ensuite il effectue les demandes souhaites.

5.1 Amelioration possibles

Des ameliorations sont possibles :

— creer plusieurs banquier et envoye la demande du client au banquier responsable du client pour qu'il traite sa demande.

— mettre a jour le nouveau solde de client a chaque transaction et la stocker dans la base de donnée.

— creer plusieurs banques

6 Documentation pour compiler et exécuter le code

Pour compiler :mvn clean compile
Pour executer : mvn exec :java -Dexec.mainClass="sd.akka.App"

7 Jeu de tests

7.1 Etape d'execution

1-L'utilisateur exécute l'application depuis la console (App.java). Le système d'acteurs est créé (Akka) avec les acteurs BanqueActor, BanquierActor, et ClientActor.
2-L'utilisateur entre son ID. App.java utilise JDBC pour vérifier l'existence de l'ID dans la base de données. Si l'ID existe, le solde du client est récupéré et affiché.
3-L'utilisateur choisit une action (dépôt ou retrait) et entre le montant.
4-Le message correspondant est envoyé à ClientActor.
5-ClientActor communique avec BanqueActor pour initier la transaction.
6-BanqueActor crée un BanquierActor pour gérer la transaction.
7-Le BanquierActor execute la transaction et renvoie le resultat a BanqueActor.
8-Banqueactor recoit le resultat et le transmet a ClientActor qui l'affiche dans la console.
9-Le système d'acteurs se termine.

7.2 Execution

L'image (figure 3) nous montre l'execution d'une demande de retrait, le solde initial etait de 120 euros, il a ete envoye au client, le client la envoye a la banque qui la envoyé au banquier. Le banquier a effectue le retrait avec le montant saisi qui lui a egalement ete envoye avec le solde initial puis a renvoye le resultat a la banque.

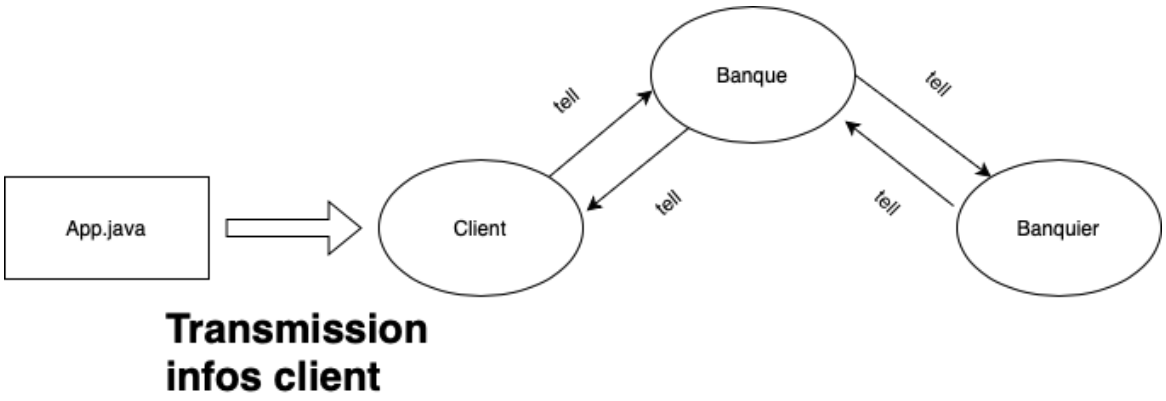

FIGURE 2 – Shema systeme

```
Veuillez entrer votre ID entre 1 et 6 :
4
Vous avez choisi l'id: 4
 Bonjour M.Hammani, votre solde actuel est de : 120 euros.
Choisissez une action :
1 - Effectuer un dépôt
2 - Effectuer un retrait
2
Veuillez entrer le montant :
30
Vous avez choisi d'effectuer une demande de retrait.
Votre solde est de : 90 euros.
```

FIGURE 3 – Execution d'une demande de retrait

La banque a envoye le resultat obtenu au client.Le nouveau solde est de 90 euros.
L'image (figure 4) nous montre l'execution d'une demande de depot, le solde initial etait de 3 euros il a ete envoye au client, le client la envoye a la banque qui la envoye au banquier. Le banquier a effectue le depot avec le montant saisi qui lui a egalement ete envoye avec le solde initial puis a renvoye le resultat a la banque. La banque a envoye le resultat obtenu au client.

Cas d'erreur :

- Si on entre un ID non valide : il nous informe que l'id entre n'a pas ete trouve (figure 5) et termine le processus d'execution.
- Si on entre un montant de retrait superieur au solde initial(figure 6) : dans ce cas le processus s'arrete sans donner le nouveau solde puisque le banquier n'a pas effectue le retrait car la condition si(montantretrait>solde) est vrai.
- Si on fait une demande de depot avec un montant negatif(figure 7) : le programme fait une demande de retrait et renvoie le resultat c'est a dire solde initial - montant.

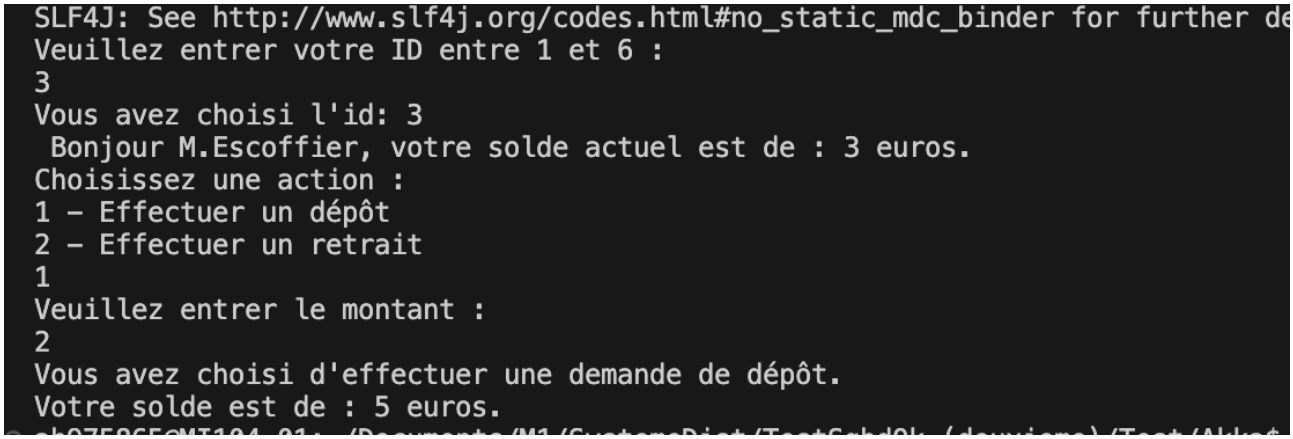

FIGURE 4 – Execution d'une demande de depot

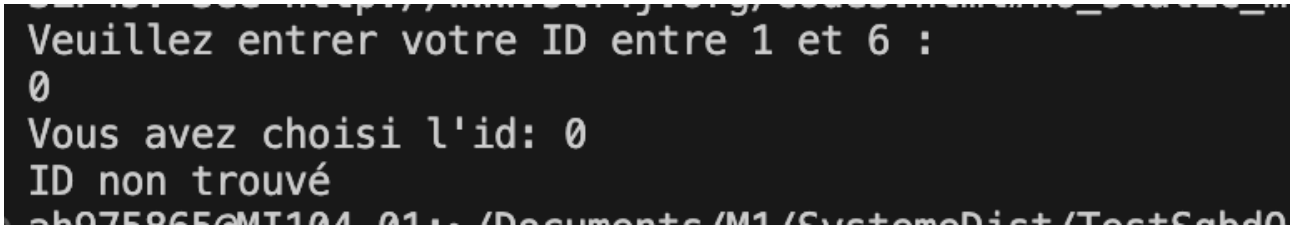

FIGURE 5 – Execution avec faux ID

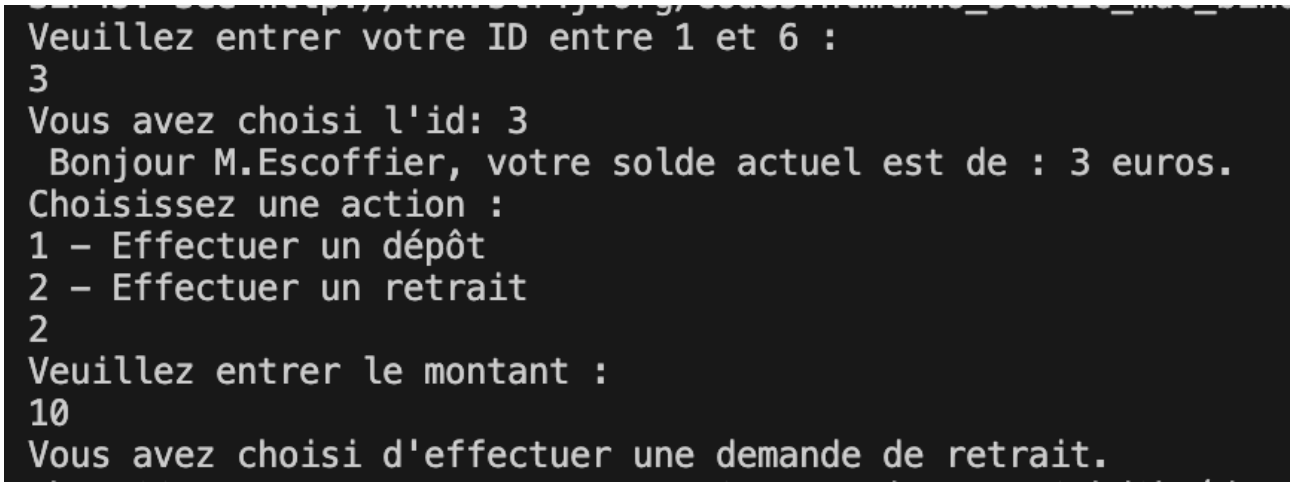

FIGURE 6 – Execution demande de retrait faux

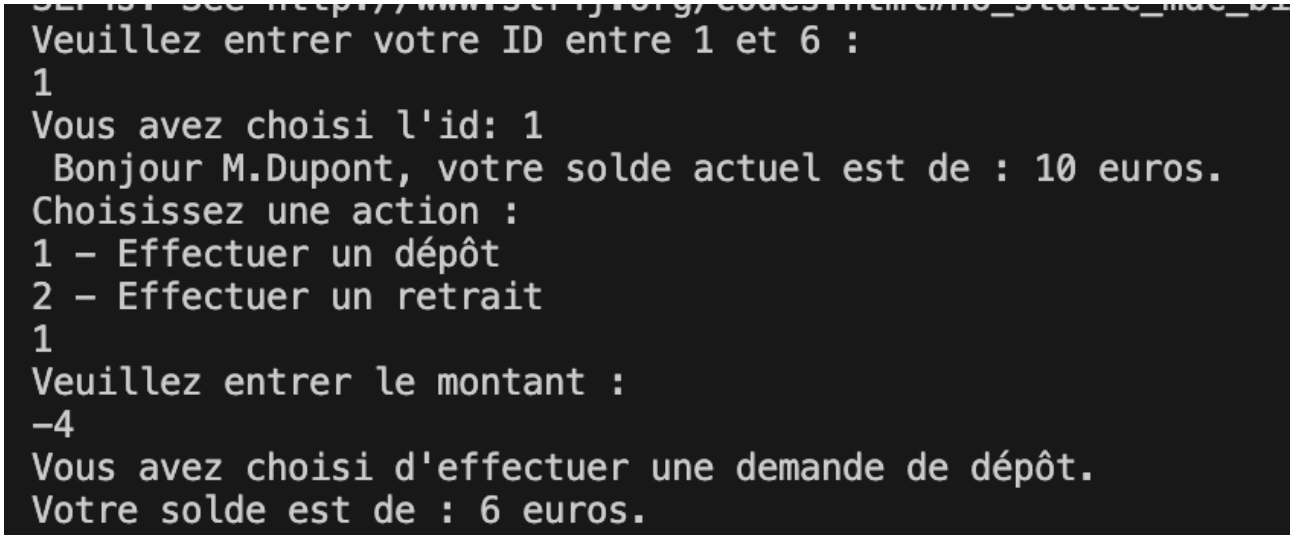

Figure 7 – Execution d'une demande de depot faux