

Université de Bourgogne



Compte Rendu du TP pour le cours Base de Données et Environnement

Distribué -M2 BDIA

**Implémentation du "Bag of Tasks" avec persistance des données dans  
une architecture RMI**

Sous la supervision de: Eric Leclercq & Anabelle Gillet

Réalisé par : Moussa TRAORE

Octobre 2023

# Table des matières

<b>Table des matières</b>	<b>i</b>
<b>Introduction</b>	<b>1</b>
<b>1 Les systèmes distribués avec RMI</b>	<b>2</b>
1.1 objectif de RMI . . . . .	2
1.2 Bag of Tasks . . . . .	3
1.3 L’annuaire . . . . .	3
1.3.1 Fonctionnement de l’annuaire . . . . .	3
1.4 Callback . . . . .	4
1.5 Persistence des Données sous Oracle . . . . .	4
<b>2 Analyse Conceptuelle du système</b>	<b>6</b>
2.1 Diagramme des cas d’utilisation . . . . .	6
2.2 Architecture globale du système . . . . .	9
2.3 Diagramme des classes . . . . .	10
2.3.1 Commentaires sur l’architecture . . . . .	11
2.4 Diagramme des Séquences . . . . .	13
2.4.1 Enregistrer une tâche dans le Bag Of Tasks . . . . .	13
2.4.2 Gestion des Workers et attribution des tâches . . . . .	14
2.4.3 Exécution des tâches . . . . .	15
<b>3 Implémentation et test de l’architecture</b>	<b>17</b>
3.1 Implémentation du Serveur . . . . .	17
3.2 Implémentation du client applicatif . . . . .	18
3.3 Worker . . . . .	20
3.4 Tests de l’architecture . . . . .	23
3.4.1 Compilation et lancement du serveur . . . . .	23

3.4.2	Compilation et lancement de la connexion à la base de données . . . . .	24
3.4.3	Compilation et lancement du client applicatif . . . . .	24
3.4.4	Lancement du Worker router . . . . .	24
3.4.5	Elements de réponse aux questions du TP . . . . .	27
3.4.6	Asynchronisme . . . . .	27
<b>Conclusion</b>		<b>30</b>
<b>Annexes</b>		<b>31</b>

# Introduction

**Un système distribué** est un ensemble de composants sur différents ordinateurs en réseau qui fonctionnent ensemble pour atteindre un objectif commun. Il est constitué d'un certain nombre de programmes informatiques qui s'exécutent sur plusieurs noeuds de calculs(aussi appelés ressources).L'objectif global visé est :

- Eliminer les points centraux de défaillance
- Eviter les goulots d'étranglements.
- Améliorer la tolérance aux interruptions de service,
- Améliorer la fiabilité l'accessibilité
- Améliorer la rapidité des traitements en faisant coopérer plusieurs noeuds

Ce rapport présente l'implémentation du pattern Bag of Tasks avec la persistance dans une ou plusieurs bases de données.L'architecture du système est composée principalement d'un ensemble de clients qui accèdent à des objets distants par le biais d'une référence fournie par un annuaire.Chaque objet distant comporte une ou plusieurs tâches qui interagissent avec une base de données.La tâche à exécuter est une requête vers une base de données.La durée d'exécution de la requête pouvant être très longue,un système de Callback permettra de récupérer les résultats d'exécution et ainsi assurer l'asynchronisme du système.Des noeuds chargés d'exécuter la tâche (workers)seront mis en place.

Nous étudierons des solutions permettant d'améliorer la performance de cette architecture en gérant notamment des problématiques liées à la gestion de la concurrence,l'asynchronisme , l'ordonnancement des tâches et l'attribution d'exécution des tâches aux workers,l'optimisation de la mémoire pour le stockage des objets distants et des tâches.

Ce rapport est structuré comme suit :

Le premier chapitre présente les technologies pour les systèmes distribués en Java :RMI,Object Factory,Bag of Tasks.Nous présentons ensuite l'étude conceptuelle du système :Diagramme des classes,Diagrammes des cas d'utilisation et Disagramme des séquences. Ensuite nous détaillerons l'implémentation du système avec le langage Java. Nous testerons le bon fonctionnement du système.

# Chapitre 1

## Les systèmes distribués avec RMI

### 1.1 objectif de RMI

Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'exécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outil `rmic` fourni avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

Les différentes étapes pour créer un objet distant et l'appeler avec RMI sont les suivantes : Le développement coté serveur se compose de :

La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance  
L'écriture d'une classe qui implémente cette interface  
L'écriture d'une classe quiinstanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de noms RMI (RMI Registry)  
Le développement côté client se compose de :

L'obtention d'une référence sur l'objet distant à partir de son nom  
L'appel à la méthode à partir de cette référence  
Enfin, il faut générer les classes stub et skeleton en exécutant le programme `rmic`

avec le fichier source de l'objet distant.

Le serveur peut déléguer la création et la gestion d'objets à un autre composant. Ce mécanisme est appelé **Object Factory**.

## 1.2 Bag of Tasks

Le pattern **Bag of Task** est un pattern utilisé dans l'architecture RMI pour la gestion des tâches qui seront distribués aux applications clientes. Il permet une gestion d'un ensemble de tâches qui peuvent être exécutées de façon parallèle par plusieurs clients.

Les tâches sont stockées dans une pile ou htable et sérialisées vers les clients distants dès qu'elles sont invoquées. C'est une sorte de médiateur distribuant les tâches à des clients pour être traitées en parallèle.

L'avantage clé du modèle "bag of tasks" réside dans sa capacité à exploiter efficacement les ressources distribuées. Chaque machine peut traiter indépendamment les tâches qui lui sont assignées, maximisant ainsi l'utilisation des capacités de calcul disponibles.

Dans le chapitre 3, nous détaillerons les mécanismes mis en place dans l'implémentation de la gestion du Bag of tasks.

## 1.3 L'annuaire

Dans Java RMI (Remote Method Invocation), l'annuaire joue un rôle crucial en tant que service de recherche distribué permettant aux clients de localiser des objets distants sur le réseau. L'annuaire RMI est souvent implémenté à l'aide du registre RMI (rmiregistry).

### 1.3.1 Fonctionnement de l'annuaire

1. Enregistrement d'objets distants : Lorsqu'un objet distant est créé, il doit être enregistré auprès de l'annuaire RMI. Cela se fait en utilisant la méthode bind du registre RMI ou une méthode similaire. L'objet distant est associé à un nom unique dans l'annuaire.

Pour enregistrer un objet dans l'annuaire il faut préciser un numéro de port, le nom de l'objet et un hostname.

2. Recherche d'objets distants : Lorsqu'un client souhaite utiliser un objet distant, il doit le localiser dans l'annuaire. Cela se fait en utilisant la méthode lookup du registre RMI. Le client fournit le même nom unique sous lequel l'objet distant a été enregistré.

3. Invocation de méthodes distantes : Une fois que le client a obtenu une référence à l'objet distant, il peut invoquer des méthodes à distance comme s'il s'agissait d'objets locaux.
4. Gestion des exceptions : Java RMI gère automatiquement la sérialisation des paramètres et des résultats lors des appels de méthode distante. En cas d'erreur, des exceptions spécifiques à RMI, telles que `RemoteException`, peuvent être gérées pour traiter les problèmes de communication ou d'exécution à distance. L'annuaire RMI simplifie la recherche et l'accès aux objets distants, facilitant ainsi le développement d'applications distribuées en Java. Il permet aux clients de trouver dynamiquement des services distants sans avoir à connaître l'emplacement physique des objets sur le réseau.

## 1.4 CallBack

## 1.5 Persistence des Données sous Oracle

La persistance des données dans une base de données fait référence à la capacité de stocker des données de manière permanente, de sorte qu'elles puissent être récupérées et utilisées même après la fermeture du programme, du système ou de la machine qui a créé les données. Cela garantit que les données sont disponibles et conservées au fil du temps, même en cas de redémarrage du système ou de fermeture de l'application.

Dans le contexte des bases de données, la persistance des données est essentielle pour plusieurs raisons :

1. **Conservation des données** : Les données stockées dans une base de données persistent même après la fin de l'exécution du programme ou du processus qui les a créées. Cela garantit que les informations ne sont pas perdues et restent disponibles pour une utilisation future.
2. **Partage des données** : La persistance permet de partager des données entre différents utilisateurs, sessions ou applications. Les données stockées peuvent être consultées et modifiées par différents utilisateurs à des moments différents.
3. **Reprise après incident** : En cas de panne du système, de défaillance matérielle ou de redémarrage, la persistance des données garantit que les informations ne sont pas perdues. Les bases de données sont conçues pour être robustes et résilientes.
4. **Intégrité des données** : Les bases de données offrent généralement des mécanismes pour maintenir l'intégrité des données, notamment des contraintes et des validations. Ces mécanismes persistent avec les données, assurant ainsi la qualité des informations stockées.

5. **Performances** : Les systèmes de gestion de bases de données (SGBD) sont optimisés pour la recherche, la récupération et la modification rapides des données. La persistance des données permet d'optimiser les performances d'accès aux informations.

En résumé, la persistance des données dans une base de données garantit la conservation, l'accessibilité et l'intégrité des informations stockées, ce qui est crucial pour de nombreuses applications et systèmes informatiques.



# Chapitre 2

## Analyse Conceptuelle du système

La conception d'un système est une étape cruciale dans la réalisation de tout logiciel. Nous avons utilisé le langage UML, un langage orienté objet qui permet de séparer les points de vues du système. Du point de vue statique, nous pouvons élaborer le diagramme de classes qui montre les différents composants du système avec les liens qui existent entre eux. Le diagramme des cas d'utilisation permet de lister les fonctionnalités offertes par le système et les différents acteurs qui interviennent pour la réalisation de ces fonctionnalités. Enfin nous présentons le diagramme de séquence qui est une vue dynamique détaillant les scénarios nominaux et alternatifs pour mettre en oeuvre une fonctionnalité sur le système.

### 2.1 Diagramme des cas d'utilisation

Le tableau 2.1 détaille les différents acteurs de l'architecture mise en place avec leurs fonctionnalités respectives.

Rôle des acteurs	
Acteur	Rôle
Serveur	stocker le bag of task, enregistre l'objet distant dans l'annuaire
Bag of tasks	gérer une liste des tâches, ajout et suppression des tâches, création de la tâche
Base de données	Stocker de manière persistante les données
CallBack	Récupérer les résultats d'exécution d'une tâche
Worker	exécuter les tâches
Client applicatif	Créer et envoyer les tâches au Bag Of Tasks
Routeur-worker	Coordonner la gestion et l'affectation des tâches aux différents workers

TABLE 2.1 – Rôles des composants du système

La figure 2.1 présente un exemple de diagramme de cas d'utilisation de notre système.

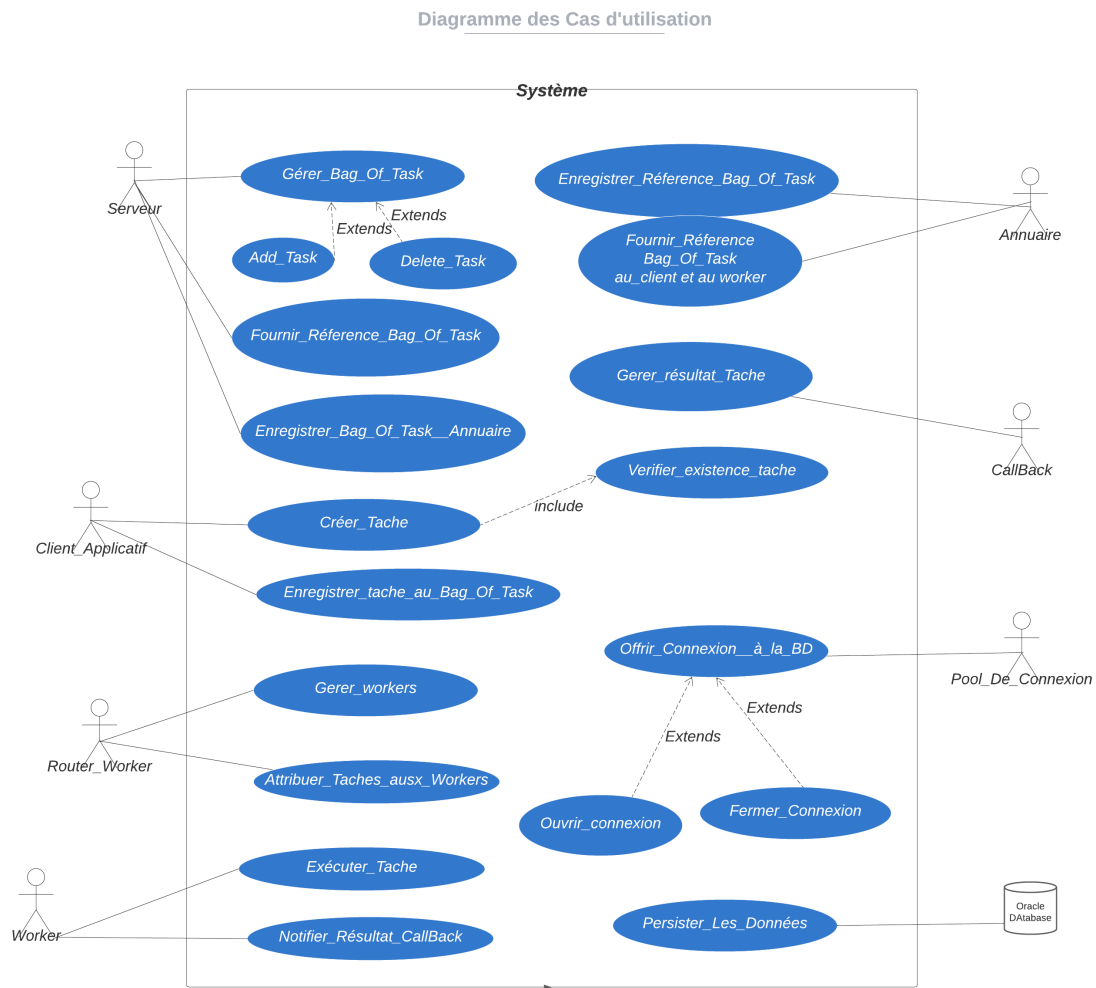


FIGURE 2.1 – Diagramme des cas d'utilisation

## 2.2 Architecture globale du système

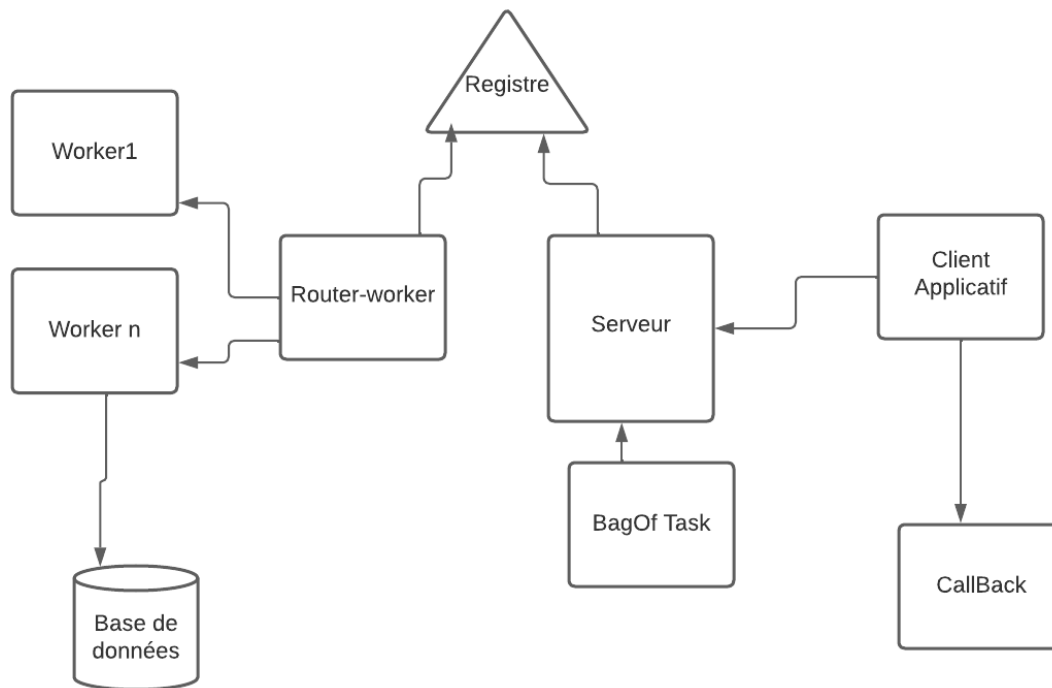


FIGURE 2.2 – Architecture globale

## 2.3 Diagramme des classes

Le diagramme de classe présentée par la figure 2.3, montre l'architecture globale de notre système et la relation entre les composants identifiées.

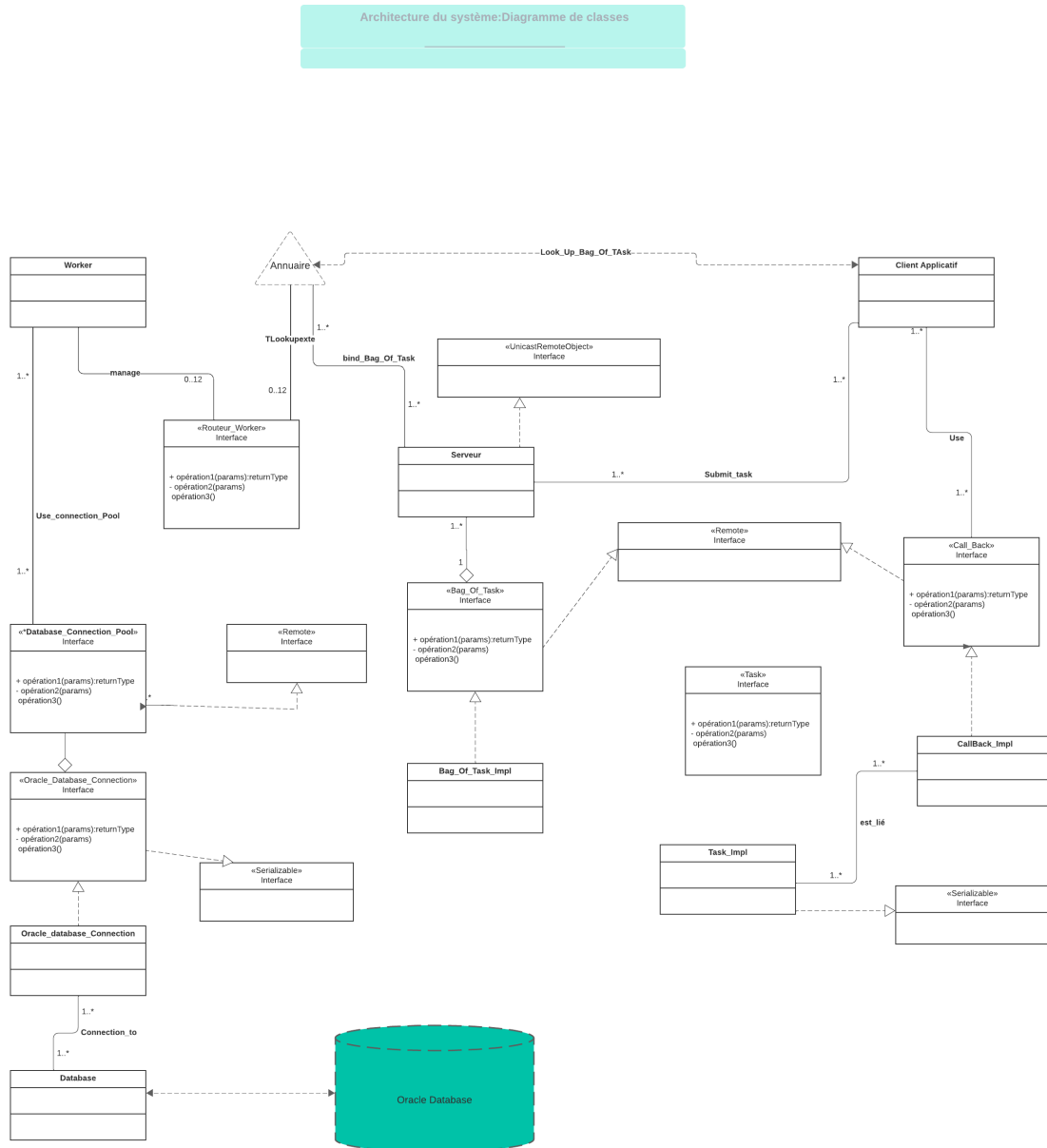


FIGURE 2.3 – Diagramme de classes

Pour éviter que des informations en rapport avec la base de données ne soient sérialisées, nous avons apporté une modification sur ce diagramme en créant un serveur qui offre une pool de connexions à la base de données. Un object factory va créer une liste de connexions. La pool de

connexion sera stockée dans l'annuaire et chaque worker s'adresse au serveur pour obtenir une connexion.

### 2.3.1 Commentaires sur l'architecture

Le diagramme de classe présentée à la figure 2.3 montre l'architecture statique du système.

Dans une architecture distribuée fonctionnant avec RMI, il s'agit d'un appel de méthodes à distance par un ou plusieurs clients.

Au cours de ce TP nous avons mis en place un ensemble d'interfaces et classes permettant à un client applicatif de créer une tâche qui fait appel à une base de données sous Oracle. La tâche peut être une simple **requête d'affichage du contenu d'une table**, une **transaction**, mise à jour des enregistrements ou même une **transaction ou procédure stockée**.

La liste des tâches en attente d'exécution sont placées dans une file d'attente gérée avec l'algorithme **FIFO :First In First Out**.

Afin d'identifier l'origine de chaque tâche, toute tâche est caractérisée par un Id unique et l'Id du client applicatif ayant émis la tâche. La création d'une tâche se fait en implément.

1. **Le client applicatif** interagit directement avec le serveur à travers un annuaire pour avoir la référence du Bag of Task stocké au niveau du serveur et vers lequel il peut envoyer la tâche.
2. **Le serveur** permet de découpler la partie métier et la partie fonctionnelle du système. Il est chargé de stocker le Bag of Task contenant une liste de tâches soumises par le client applicatif, envoyer la référence du Bag of task vers un annuaire qui écoute sur le port 1099 afin que les workers puissent y accéder et exécuter les différentes tâches en attente.
3. **Le Callback** permet de gérer l'asynchronisme dans l'exécution des tâches. Cela est utile dans les cas où le serveur a besoin de notifier le client de certains événements ou de lui envoyer des mises à jour de manière asynchrone. Dans notre cas, une fois la tâche exécutée par le worker, le client applicatif sera notifié du résultat obtenu et ce de façon asynchrone.
4. **Le Router-worker** a été mis en place pour gérer la création des workers (client RMI) et l'attribution des tâches. Dans notre architecture nous avons opté à la création de 3 worker. Chaque worker peut avoir deux états (état disponible : aucune tâche en cours d'exécution ou non disponible lorsque une tâche n'a pas encore fini son exécution). Le router worker communique avec le serveur à travers l'annuaire pour voir périodiquement si des tâches ont été chargées dans le Bag of Task dans lequel cas il récupère les tâches.
5. **une worker** représente des noeuds avec un processeur pour exécuter une liste de tâches. Pour avoir une haute disponibilité du système, nous avons mis ensemble une liste de workers qui

peuvent travailler en parallèle offrant ainsi une accélération quant à l'exécution des tâches en attente.

6. **La persistance des données se fait à travers un pool de connexion vers la base de données.** Comme nous l'avons vu plus haut, l'objectif visé par ce TP était l'exécution d'une tâche comportant une requête à une base de données. Pour cela nous avons créé un composant chargé de gérer l'accès à la base de données. Pour assurer l'intégrité des données nous utiliserons un système de verrous

## 2.4 Diagramme des Séquences

### 2.4.1 Enregistrer une tâche dans le Bag Of Tasks

Le diagramme de séquences présentée par la figure 2.4, donne les détails des étapes effectuées par le client applicatif pour enregistrer une tâche dans le Bag of Task.

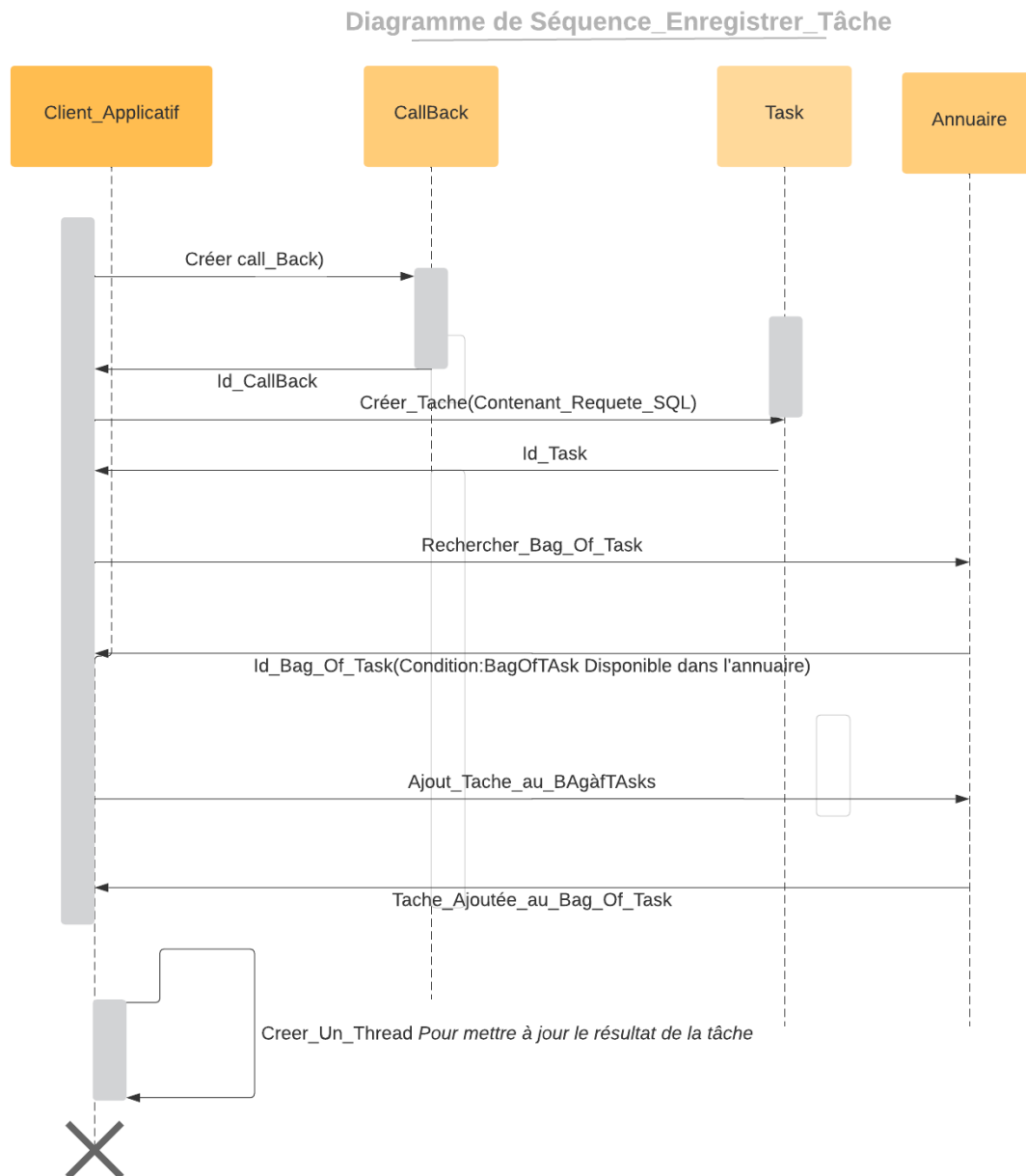


FIGURE 2.4 – Diagramme de séquence pour enregistrer une tâche



## 2.4.2 Gestion des Workers et attribution des tâches

Le diagramme de séquences présentée par la figure 2.5, montre les étapes pour attribuer les tâches aux différents workers.

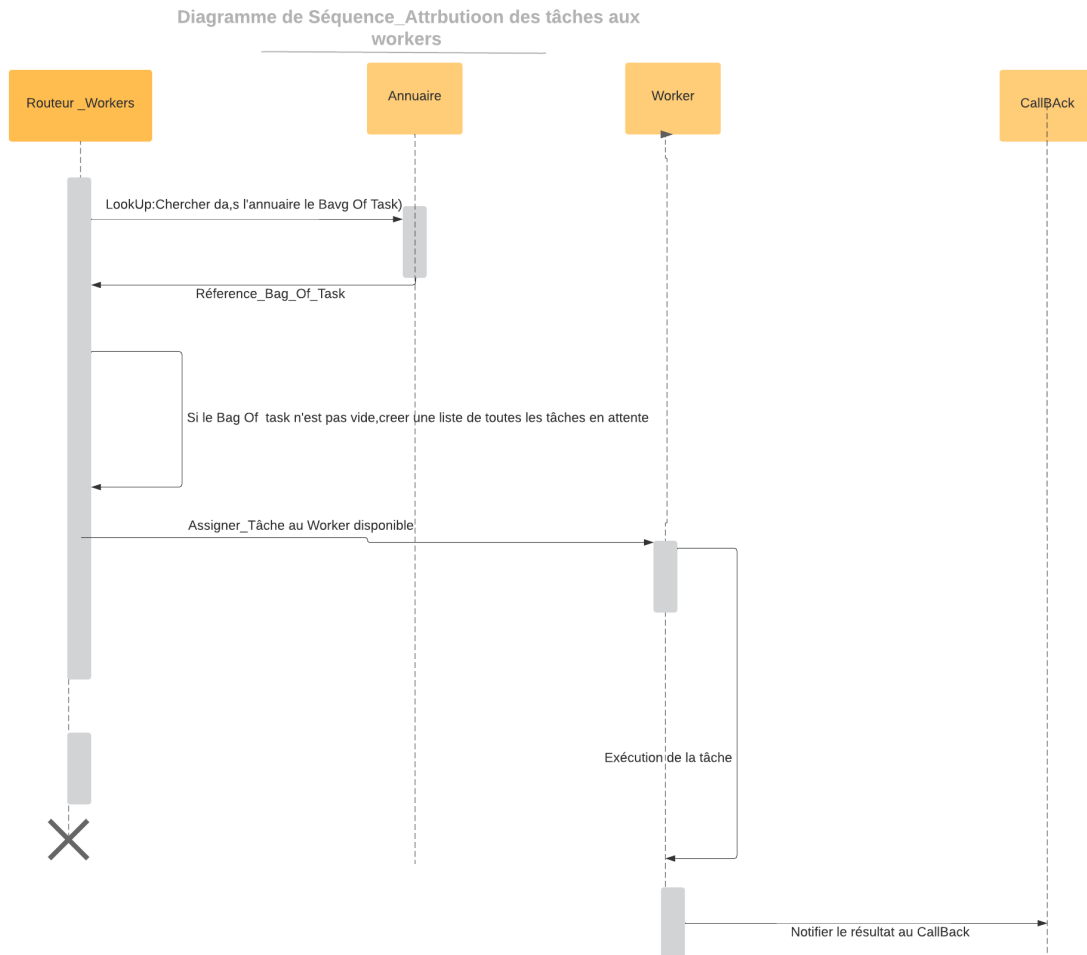


FIGURE 2.5 – Diagramme de séquence pour la gestion des workers

### 2.4.3 Exécution des tâches

Le diagramme de séquences présentée par la figure 2.6, montre comment le worker exécute les tâches qui lui sont attribuées.

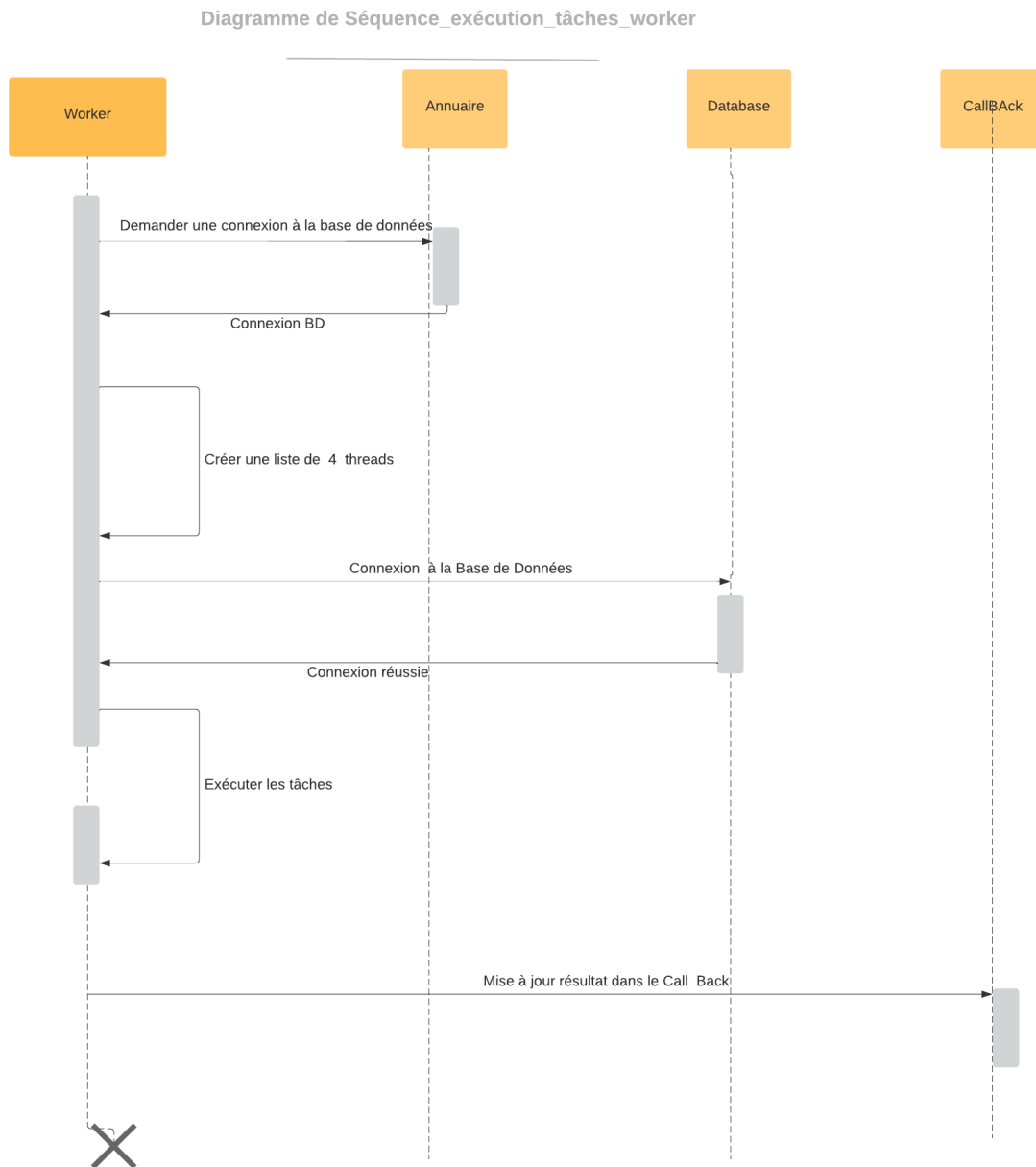


FIGURE 2.6 – Diagramme de séquence l'exécution des tâches par le worker

Pour la gestion de l'accès concurrent à une même table de la base de données , nous avons opté d'utiliser des **verrous optimistes** dans l'interface BD qui gère les interactions avec la base de données.

Une autre solution serait d'utiliser *les vues matérialisées*. Les verrous optimistes sont une approche intelligente pour gérer les opérations concurrentes sur les données.

Avec les verrous optimistes, nous permettons à plusieurs clients d'accéder aux mêmes données en même temps, sans les verrouiller. Au lieu de verrouiller les données pendant une mise à jour, chaque utilisateur vérifie d'abord si les données ont été modifiées par quelqu'un d'autre. Si oui, l'utilisateur peut gérer la situation, fusionner les modifications ou résoudre les conflits.

Cette approche offre des avantages significatifs. Elle améliore les performances, car elle permet à de nombreux utilisateurs d'accéder aux données en même temps. De plus, cela permet une interaction plus fluide et en temps réel avec la base de données.

**Les vues matérialisées** en revanche, sont des copies statiques des données d'origine, et elles peuvent causer des problèmes de performances et de synchronisation.

En utilisant des verrous optimistes, nous avons une gestion plus flexible des données et un meilleur support pour les utilisateurs concurrents. Cela permet de maintenir la performance du système tout en évitant les problèmes potentiels liés aux vues matérialisées.

# Chapitre 3

## Implémentation et test de l'architecture

### 3.1 Implémentation du Serveur

```
1
2 import java.rmi.RemoteException;
3 import java.rmi.registry.LocateRegistry;
4 import java.rmi.registry.Registry;
5 import java.rmi.server.UnicastRemoteObject;
6
7 public class Server extends UnicastRemoteObject{
8     protected Server() throws RemoteException {
9         super();
10    }
11    public static void main(String[] args) {
12        try {
13            //Cr er une instance de BagOfTasksImpl
14            BagOfTasksInterface bagOfTasks = new BagOfTasks();
15            int port=1099;
16            //Obtenez le registre RMI
17            Registry registry = LocateRegistry.createRegistry(port);
18            //Liez l'objet distant (le sac de t ches) au registre
19            registry.rebind("BagOfTasks", bagOfTasks);
20            System.out.println("Serveur pr t.");
21        } catch (Exception e) {
22            System.err.println("Erreur du serveur : " + e.toString());
23        }
24    }
25 }
```

Listing 3.1 – Code java pour le serveur

Le serveur instancie un nouveau **bag of task** et l'enregistre dans l'annuaire à l'aide de la méthode "rebind"

## 3.2 Implémentation du client applicatif

```
1
2
3 import java.rmi.registry.LocateRegistry;
4 import java.rmi.registry.Registry;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 public class Client {
9
10     private List<TaskInterface> taskList;
11     private List<CallbackInterface> callbackList;
12     private String Client_id;
13
14     public Client(List<TaskInterface> tasks, String client_id, List<
15 CallbackInterface> callbacks) {
16         taskList = tasks;
17         Client_id = client_id;
18         callbackList = callbacks;
19     }
20
21     public List<TaskInterface> getTaskList() {
22         return taskList;
23     }
24
25     public List<CallbackInterface> getCallbackList() {
26         return callbackList;
27     }
28
29     public static void main(String[] args) {
30         try {
31             List<CallbackInterface> callbacks = new ArrayList<CallbackInterface
32 >();
33             CallbackInterface clB = new Callback();
34             CallbackInterface clB1 = new Callback();
35             CallbackInterface clB2 = new Callback();
36             callbacks.add(clB);
37             callbacks.add(clB1);
```

```

36         callbacks.add(clB2);
37
38         TaskInterface task1 = new Task("UPDATE TRANSACTION SET MONTANT =
400.00 WHERE VERSION=0", "1", callbacks.get(0));
39         TaskInterface task2 = new Task("SELECT * FROM COMPTE", "2",
callbacks.get(1));
40         TaskInterface task3 = new Task("SELECT * FROM CLIENT", "1",
callbacks.get(1));
41         TaskInterface task4 = new Task("SELECT * FROM TRANSACTION", "1",
callbacks.get(0));
42         TaskInterface task5 = new Task("SELECT * FROM COMPTE", "2",
callbacks.get(1));
43         TaskInterface task6 = new Task("SELECT * FROM CLIENT", "1",
callbacks.get(1));
44
45
46         List<TaskInterface> tasks = new ArrayList<TaskInterface>();
47         tasks.add(task1);
48         tasks.add(task2);
49         tasks.add(task3);
50         tasks.add(task4);
51         tasks.add(task5);
52         tasks.add(task6);
53
54         final Client client = new Client(tasks, "1", callbacks); // Cr ez
une instance de Client
55         final Registry registry = LocateRegistry.getRegistry("localhost",
1099);
56         List<Thread> taskThreads = new ArrayList<Thread>();
57         final int numThreads = 2; // Limitez 2 threads
58         final int tasksPerThread = tasks.size() / numThreads; // Nombre de
t ches par thread
59         final int size=tasks.size();
60         for (int i = 0; i < size; i += tasksPerThread) {
61             final int startIndex = i;
62             Thread taskThread = new Thread(new Runnable() {
63                 public void run() {
64                     try {
65                         BagOfTasksInterface bagOfTasks = (
BagOfTasksInterface) registry.lookup("BagOfTasks");
66                         for (int j = startIndex; j < startIndex +
tasksPerThread && j < size; j++) {
67                             bagOfTasks.addTaskToInitial(client.getTaskList
().get(j));

```

```

68         System.out.println("Thread "+startIndex+"sent
task " + j + ": " + client.getTaskList().get(j).getTaskData() + "\n");
69     }
70     for (int j = startIndex; j < startIndex +
tasksPerThread && j < size; j++) {
71         while (client.getTaskList().get(j).getCallback
().getResultat() == null)
72             Thread.sleep(2000);
73         System.out.println("Thread "+startIndex+"
received result " + j + ": " + client.getTaskList().get(j).getCallback().
getResultat() + "\n");
74     }
75     System.out.println("-----\n");
76     } catch (Exception e) {
77         e.printStackTrace();
78     }
79 }
80 });
81
82     taskThreads.add(taskThread);
83 }
84
85     for (Thread thread : taskThreads) {
86         thread.start();
87     }
88
89     for (Thread thread : taskThreads) {
90         thread.join();
91     }
92 } catch (Exception e) {
93     e.printStackTrace();
94 }
95 }
96 }

```

Listing 3.2 – Code java pour le client applicatif

### 3.3 Worker

```

1
2 import java.rmi.RemoteException;
3 import java.rmi.registry.LocateRegistry;

```

```

4 import java.rmi.registry.Registry;
5 import java.rmi.NotBoundException;
6 import java.util.List;
7 import java.util.ArrayList;
8
9
10 public class Worker {
11     private List<TaskInterface> tasks;
12     private List<Thread> Liste_Threads;
13     DatabaseConnectionPoolService connectionPoolService;
14     private String Worker_Id;
15     public OracleDatabaseConnectionInterface connection;
16
17     public Worker(String id) throws java.rmi.RemoteException{
18         tasks = new ArrayList<TaskInterface>();
19         Liste_Threads=new ArrayList<Thread>();
20         this.Worker_Id=id;
21         this.connection=null;
22
23         int port=1098;
24         try {
25             Registry registry = LocateRegistry.getRegistry("localhost", port);
26             this.connectionPoolService = (DatabaseConnectionPoolService) registry.
lookup("DatabaseConnectionPoolService");
27         } catch (java.rmi.NotBoundException e) {
28             e.printStackTrace();
29         } catch (Exception e) {
30             e.printStackTrace();
31         }
32         Thread_IniZialing();
33     }
34
35     public void Thread_IniZialing() {
36         final int numThreads = 2; // Limitez 2 threads
37         final int tasksPerThread = 3; // Nombre de t ches par thread
38
39         for (int i = 0; i < numThreads; i++) {
40             final int startIndex = i * tasksPerThread;
41             Thread thread = new Thread(new Runnable() {
42                 public void run() {
43                     try {
44                         for (int j = startIndex; j < startIndex +
tasksPerThread && j<5; j++) {
45                             String result = null;

```



```

46         while (result == null) {
47             Thread.sleep(1000);
48             if (!tasks.isEmpty()) {
49                 TaskInterface task = tasks.remove(0);
50                 result = connectionPoolService.
getAvailableDatabase(task.execute());
51                 task.getCallback().setResultat(result);
52                 System.out.println(result);
53                 System.out.println(Liste_Threads.size()+1);
54             } else {
55                 Thread.sleep(1000);
56             }
57         }
58     }
59     } catch (Exception e) {
60         e.printStackTrace();
61     }
62 }
63 });
64 Liste_Threads.add(thread);
65 thread.start();
66 }
67 }
68
69
70
71
72 public void assignTasks(List<TaskInterface> newTasks) throws java.rmi.
RemoteException{
73     if (newTasks != null) {
74         tasks.addAll(newTasks);
75         executeTasks();
76     }
77 }
78
79 public boolean isAvailable() {
80     return tasks.isEmpty();
81 }
82
83
84
85
86 public void executeTasks() {
87     for (Thread thread : Liste_Threads) {

```

```

88         if (thread.getState() == Thread.State.NEW) {
89             thread.start();
90         }
91     }
92 }
93
94 }

```

Listing 3.3 – Code java pour le worker

## 3.4 Tests de l'architecture

Pour tester le système mise en place, nous compilons et exécutons les différentes classes dans l'ordre suivant :

### 3.4.1 Compilation et lancement du serveur

Se positionner dans le répertoire contenant les différentes classes java et taper les commandes

```

1 javac Server.java
2 java Server

```

Listing 3.4 – Commande pour lancer le serveur

La figure 3.1, montre comment compiler et démarer le serveur.

The screenshot shows a terminal window titled "TERMINAL". It contains two commands and their outputs:

```

PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> javac Server.java
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> java Server
Serveur prêt.

```

FIGURE 3.1 – Compilation et exécution du serveur

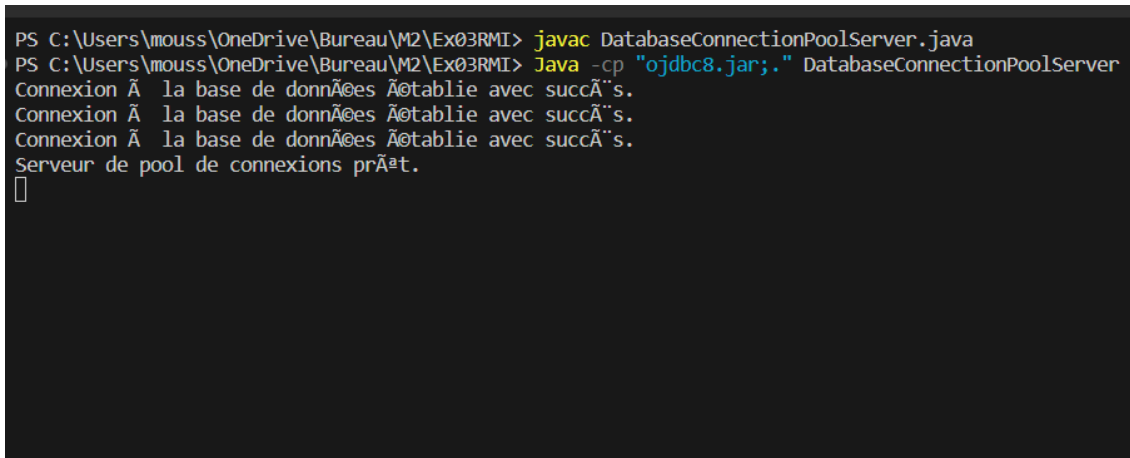
### 3.4.2 Compilation et lancement de la connexion à la base de données

Se positionner dans le répertoire contenant les différentes classes java et taper les commandes

```
1 javac DatabaseConnectionPoolServer.java
2 java -cp "ojdbc8.jar";." DatabaseConnectionPoolServer
```

Listing 3.5 – Commande pour lancer le serveur de pool de connexion

La figure 3.2, montre comment lancer la connexion à la base de données.



```
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> javac DatabaseConnectionPoolServer.java
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> Java -cp "ojdbc8.jar;." DatabaseConnectionPoolServer
Connexion Ã la base de donnÃes Ãtablie avec succÃs.
Connexion Ã la base de donnÃes Ãtablie avec succÃs.
Connexion Ã la base de donnÃes Ãtablie avec succÃs.
Serveur de pool de connexions prÃt.
[]
```

FIGURE 3.2 – Activation de la connexion à la base de données

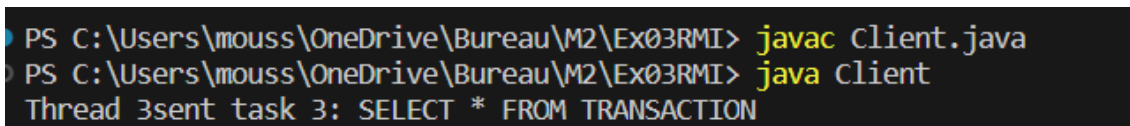
### 3.4.3 Compilation et lancement du client applicatif

Se positionner dans le répertoire contenant les différentes classes java et taper les commandes

```
1 javac Client.java
2 java Client
```

Listing 3.6 – Commande pour lancer le client applicatif

La figure 3.3, montre comment compiler et démarer le client applicatif.



```
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> javac Client.java
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> java Client
Thread 3sent task 3: SELECT * FROM TRANSACTION
```

FIGURE 3.3 – Lancement du client applicatif

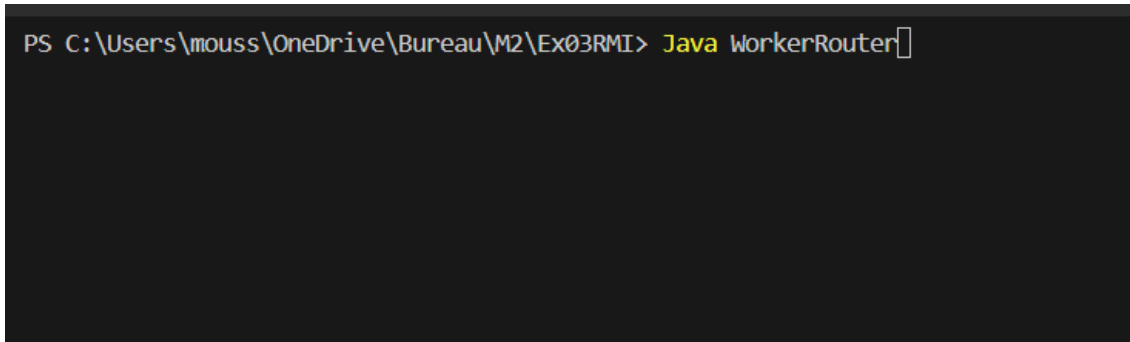
### 3.4.4 Lancement du Worker router

Se positionner dans le répertoire contenant les différentes classes java et taper les commandes

```
1 javac WorkerRouter.java
```

Listing 3.7 – Commande pour lancer le client applicatif

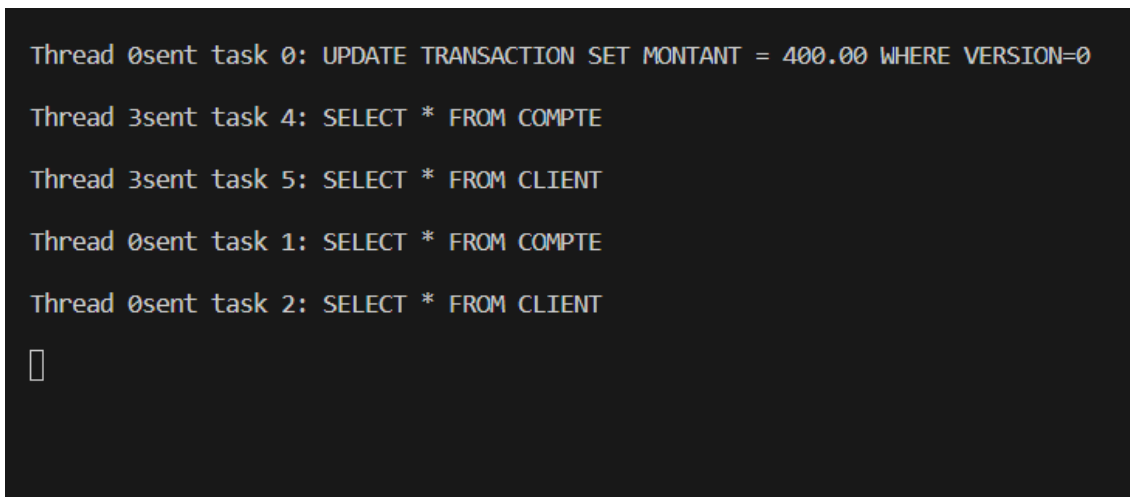
La figure3.4, montre comment compiler et démarer le worker-router.



```
PS C:\Users\mouss\OneDrive\Bureau\M2\Ex03RMI> Java WorkerRouter
```

FIGURE 3.4 – Lancement du worker-router qui coordonne l'exécution des tâches

La figure3.5, montre la liste des tâches qui ont été émises par le serveur.



```
Thread 0sent task 0: UPDATE TRANSACTION SET MONTANT = 400.00 WHERE VERSION=0
Thread 3sent task 4: SELECT * FROM COMPTE
Thread 3sent task 5: SELECT * FROM CLIENT
Thread 0sent task 1: SELECT * FROM COMPTE
Thread 0sent task 2: SELECT * FROM CLIENT

```

FIGURE 3.5 – Affichage de la liste des tâches envoyées par le client applicatif

La figure3.6, montre le résultat du côté du routeur-worker.

```
Type de compte: Courant

Solde: 9.61674077205082328621295069027681259195E03
Type de compte: Courant

Solde: 4.79914367416428860549610799092026224609E03
Type de compte: Courant

Solde: 1.94794256386311009027526153790946651448E03
Type de compte: Courant

3
operation Ã©chouÃ©Nom: TRAORE
PrÃ©nom: Emilien
Adresse: Boulevard Mansart,56
NumÃ©ro de tÃ©lÃ©phone: 7734141741

Nom: TRAORE
PrÃ©nom: ThÃ©rence
Adresse: Avenue Alain Savary,69
NumÃ©ro de tÃ©lÃ©phone: 8109543046

Nom: CISSE
PrÃ©nom: Moussa
Adresse: Rue des Rossoirs,3
NumÃ©ro de tÃ©lÃ©phone: 1482587924

Nom: BELLEGUEULLE
PrÃ©nom: Emilien
Adresse: Republique,28
NumÃ©ro de tÃ©lÃ©phone: 1766919133

Nom: TRAORE
PrÃ©nom: Adama
Adresse: Boulevard Mansart,56
NumÃ©ro de tÃ©lÃ©phone: 3921165383

Nom: BAGAYOKO
PrÃ©nom: Moussa
Adresse: Avenue Alain Savary,77
NumÃ©ro de tÃ©lÃ©phone: 5139512248
```

FIGURE 3.6 – Affichage des rÃ©sultats du cÃ´tÃ© du worker router

La figure3.7, montre le rÃ©sultat du cÃ´tÃ© du client applicatif.

```

Nom: GILLET
Prénom: Mathieu
Adresse: Avenue Alain Savary,50
Numéro de téléphone: 4995350105

Nom: SANOGO
Prénom: Mathieu
Adresse: Rue Nelson Mandela,57
Numéro de téléphone: 4890652261

Nom: Michaud
Prénom: Moussa
Adresse: Rue Nelson Mandela,34
Numéro de téléphone: 6963015345

Nom: CISSE
Prénom: Mathieu
Adresse: République,3
Numéro de téléphone: 1019666679

Nom: SANOGO
Prénom: Adama
Adresse: Rue Nelson Mandela,43
Numéro de téléphone: 2940678697

Nom: SANOGO
Prénom: Mathieu
Adresse: Rue des Rossoirs,57
Numéro de téléphone: 3401674200

Nom: DIARRA
Prénom: Adama
Adresse: Boulevard Mansart,89
Numéro de téléphone: 8315982356

Nom: TRAORE
Prénom: Thérèse
Adresse: Avenue Alain Savary,3
Numéro de téléphone: 2636940581

Nom: CISSE

```

FIGURE 3.7 – Affichage des résultats du côté client applicatif

Nous pouvons constater que dans le scenario nominal, le système fournit bien les fonctionnalités attendues à savoir l'exécution d'une liste de tâches faisant appel à une base de données.

### 3.4.5 Elements de réponse aux questions du TP

### 3.4.6 Asynchronisme

L'utilisation du Callback offre un certain niveau d'asynchronisme. En effet le client applicatif peut envoyer plusieurs tâches et le résultat sera notifié en passant par le callback. Certaines requêtes peuvent être longues et les résultats peuvent arriver dans un certain désordre. Pour contourner ce problème nous avons mis en place un système d'identification de chaque tâche avec le résultat correspondant.

## Les impacts au niveau de la concurrence ?

Dans le contexte précis de l'utilisation de RMI et de threads pour gérer les tâches et les connexions à la base de données Oracle, voici comment la concurrence s'applique :

1. Concurrence des Connexions : Plusieurs clients (travailleurs) peuvent simultanément demander des connexions à la base de données depuis le pool de connexions. Cela permet à plusieurs clients d'exécuter des tâches de manière concurrente, améliorant ainsi l'utilisation des ressources.
2. Concurrence des Tâches : Les clients applicatifs peuvent exécuter des tâches de manière concurrente en utilisant des threads. Chaque tâche est traitée de manière indépendante, réduisant ainsi le temps d'attente global et permettant de gérer plusieurs tâches en parallèle.
3. Gestion des Tâches Concurrentes : Les tâches sont gérées de manière asynchrone, ce qui signifie que les clients n'ont pas besoin d'attendre la fin de chaque tâche pour en lancer une autre. Cela permet une utilisation efficace des ressources du système.
4. Limites de la Concurrence : Bien que la concurrence soit bénéfique en termes de performances, elle peut également entraîner des problèmes potentiels, tels que les conditions de course et les conflits d'accès aux ressources partagées, comme les connexions de base de données. Une gestion appropriée de la concurrence est essentielle pour éviter de tels problèmes.
5. Complexité de la Programmation Concurrente : La programmation concurrente peut être complexe et nécessite une planification minutieuse. Les développeurs doivent mettre en œuvre des mécanismes de synchronisation, tels que les verrous, pour garantir un accès sûr et cohérent aux ressources partagées. Dans ce contexte, la concurrence est essentielle pour maximiser l'utilisation des ressources du système, permettant ainsi aux clients d'exécuter des tâches de manière concurrente tout en gérant l'accès aux connexions de base de données. Cependant, une gestion appropriée de la concurrence et de la synchronisation est nécessaire pour éviter les problèmes de concurrence potentiels.

## Gestion de plusieurs types de requêtes en même temps

Oui, dans cette approche, il est possible de gérer simultanément différents types de requêtes, tels qu'UPDATE, INSERT, SELECT et DELETE. Les clients peuvent soumettre différents types de tâches à exécuter sur la base de données, et le système distribué est capable de traiter ces tâches de manière concurrente. Cela signifie que le système est conçu pour prendre en charge des opérations variées sur la base de données en même temps. Par exemple, un client peut soumettre une requête SELECT pour récupérer des données tout en un autre client soumet une requête UPDATE pour

modifier des données, et ces opérations peuvent être gérées en parallèle. La gestion des connexions dans le pool, l'exécution asynchrone des tâches et la gestion des verrous optimistes contribuent à permettre la gestion simultanée de divers types de requêtes sans causer de conflits majeurs ou de blocages dans le système. Cependant, il est essentiel de mettre en œuvre des mécanismes appropriés pour éviter les conflits potentiels, notamment en utilisant la gestion des verrous optimistes comme nous l'avons fait pour les opérations de mise à jour



# Conclusion

Dans ce rapport nous avons détaillé la conception et la mise en place d'une architecture distribuée basée sur RMI avec persistance des données dans une base de données Oracle. Cette architecture offre de bonnes performances en terme de parallélisme dans l'exécution de plusieurs tâches.

Le découplage de la partie métier et la partie fonctionnelle est primordiale pour assurer la sécurité, l'évolutivité et la maintenance du système en cas de panne. Dans notre architecture un client applicatif se contente d'invoquer une tâche comme l'affichage des informations d'un compte donné, la mise à jour du solde d'un compte et la partie fonctionnelle se charge de récupérer la requête pour l'affecter à un ou plusieurs noeuds (workers).

Nous pourrions ajouter plusieurs clients applicatifs qui accèdent de façon concurrente à des workers. De la même façon, en fonction de la charge (nombre de requêtes à exécuter), un ensemble de workers peuvent être ajoutés sans perturber les autres couches de l'architecture. Ceci donne une bonne flexibilité et une bonne performance du système.

Néanmoins, plusieurs problèmes pourraient survenir. Si l'annuaire est indisponible, le système ne pourra plus référencer les objets distants, ce qui provoquera une interruption du service.

L'accès concurrent à une base de données par plusieurs applications soulève aussi des problèmes liés à l'intégrité et la cohérence des données. Pour parier à ce problème nous avons opté d'abstraire la connexion à la base de données en confiant ce service à une classe particulière (Object Factory) qui sera stockée dans l'annuaire. Cela évite de dévoiler les informations relatives au nom de la base de données, les paramètres de connexion, l'adresse du serveur. Seule une référence à la connexion sera accessible par les workers, assurant ainsi une certaine sécurité de la base de données.

Cette architecture pourrait être améliorée pour prendre en charge plusieurs connexions vers des bases de données gérées par des SGBD différents. L'interopérabilité entre les systèmes écrits dans des langages de programmation différents constitue aussi une perspective d'amélioration.

# Annexes

**Code SQL pour la création des tables et triggers :** Pour pouvoir peupler notre base de données, nous avons utilisé un mécanisme de génération des aléatoires d'enregistrements à partir de plusieurs variables.

```
1
2
3 -- Cr ation de la table Client
4 CREATE TABLE Client (
5     ClientID NUMBER PRIMARY KEY,
6     Nom VARCHAR2(50),
7     Prenom VARCHAR2(50),
8     Adresse VARCHAR2(100),
9     NumeroTelephone VARCHAR2(20)
10 );
11
12 -- Cr ation de la table Compte
13 CREATE TABLE Compte (
14     CompteID NUMBER PRIMARY KEY,
15     ClientID NUMBER,
16     Solde NUMBER,
17     TypeDeCompte VARCHAR2(20),
18     CONSTRAINT fk_ClientID FOREIGN KEY (ClientID) REFERENCES Client(ClientID)
19 );
20
21 -- Cr ation de la table Transaction
22 CREATE TABLE Transaction (
23     TransactionID NUMBER PRIMARY KEY,
24     CompteID NUMBER,
25     TypeDeTransaction VARCHAR2(20),
26     Montant NUMBER,
27     DateTransaction DATE,
28     CONSTRAINT fk_CompteID FOREIGN KEY (CompteID) REFERENCES Compte(CompteID)
29 );
```

```

30 CREATE OR REPLACE PROCEDURE RemplirBaseDeDonnees AS
31     v_nom VARCHAR2(50);
32     v_prenom VARCHAR2(50);
33     v_numero_tel VARCHAR2(20);
34     v_adresse VARCHAR2(1000);
35     v_numero_bat NUMBER;
36
37     -- Declare arrays for names and street names
38     Noms DBMS_SQL.VARCHAR2A;
39     Prenoms DBMS_SQL.VARCHAR2A;
40     NomsRue DBMS_SQL.VARCHAR2A;
41 BEGIN
42     -- Initialize names and street names
43     Noms(1) := 'TRAORE';
44     Noms(2) := 'NIBAREKE';
45     Noms(3) := 'GILLET';
46     Noms(4) := 'DIARRA';
47     Noms(5) := 'BELLEGUEULLE';
48     Noms(6) := 'Michaud';
49     Noms(7) := 'BAGAYOKO';
50     Noms(8) := 'CISSE';
51     Noms(9) := 'SANOGO';
52     Noms(10) := 'DEMBELE';
53
54     Prenoms(1) := 'Moussa';
55     Prenoms(2) := 'Th rence';
56     Prenoms(3) := 'Ibrahim';
57     Prenoms(4) := 'Christophe';
58     Prenoms(5) := 'Mathieu';
59     Prenoms(6) := 'Emilien';
60     Prenoms(7) := 'Adama';
61     Prenoms(8) := 'Alima';
62     Prenoms(9) := 'Ousmane';
63     Prenoms(10) := 'Mohamed';
64
65     NomsRue(1) := 'Boulevard Mansart';
66     NomsRue(2) := 'Avenue Alain Savary';
67     NomsRue(3) := 'Rue des Rossoirs';
68     NomsRue(4) := 'Republique';
69     NomsRue(5) := 'Rue Nelson Mandela';
70
71     FOR i IN 1..100 LOOP
72         -- Generation d'un nom et d'un pr nom
73         v_nom := Noms(TRUNC(DBMS_RANDOM.VALUE(1, 10)));

```

```

74      v_prenom := Prenoms(TRUNC(DBMS_RANDOM.VALUE(1, 10)));
75
76      -- G n ration al atoire de num ros de t l phone
77      v_numero_tel := TO_CHAR(TRUNC(DBMS_RANDOM.VALUE(1000000000, 9999999999)
78  ));
79
80      -- G n ration al atoire d'une adresse en effectuant des permutations
81      v_adresse :=NomsRue(DBMS_RANDOM.VALUE(1, 5)) || ',' || TO_CHAR(TRUNC(
82  DBMS_RANDOM.VALUE(1, 100)));
83      INSERT INTO Client(ClientID,Nom,Prenom,Adresse,NumeroTelephone)
84      VALUES(i,v_nom,v_prenom,v_adresse,v_numero_tel);
85      -- Chaque client a au moins un compte
86      INSERT INTO Compte(CompteID, ClientID, Solde, TypeDeCompte)
87      VALUES (i, i, DBMS_RANDOM.VALUE(1000, 10000), 'Courant');
88
89      -- G n rez quelques transactions pour chaque compte
90      FOR j IN 1..DBMS_RANDOM.VALUE(1, 5) LOOP
91          INSERT INTO Transaction(TransactionID, CompteID, TypeDeTransaction,
92  Montant, DateTransaction)
93          VALUES (i * 100 + j, i, CASE WHEN DBMS_RANDOM.VALUE < 0.5 THEN '
94  Depot' ELSE 'Retrait' END, DBMS_RANDOM.VALUE(10, 500), SYSDATE - DBMS_RANDOM
95  .VALUE(1, 365));
96      END LOOP;
97
98      END LOOP;
99
100     COMMIT;
101 END RemplirBaseDeDonnees;
102 /
103 CREATE OR REPLACE PROCEDURE RemplirBaseDeDonnees AS
104     v_nom VARCHAR2(50);
105     v_prenom VARCHAR2(50);
106     v_numero_tel VARCHAR2(20);
107     v_adresse VARCHAR2(100);
108     v_numero_bat NUMBER;
109
110     -- Declare arrays for names and street names
111     Noms DBMS_SQL.VARCHAR2A;
112     Prenoms DBMS_SQL.VARCHAR2A;
113     NomsRue DBMS_SQL.VARCHAR2A;
114 BEGIN
115     -- Initialize names and street names
116     Noms(1) := 'TRAORE';
117     Noms(2) := 'NIBAREKE';

```

```

113 Noms(3) := 'GILLET';
114 Noms(4) := 'DIARRA';
115 Noms(5) := 'BELLEGUEULLE';
116 Noms(6) := 'Michaud';
117 Noms(7) := 'BAGAYOKO';
118 Noms(8) := 'CISSE';
119 Noms(9) := 'SANOGO';
120 Noms(10) := 'DEMBELE';
121
122 Prenoms(1) := 'Moussa';
123 Prenoms(2) := 'Th rence';
124 Prenoms(3) := 'Ibrahim';
125 Prenoms(4) := 'Christophe';
126 Prenoms(5) := 'Mathieu';
127 Prenoms(6) := 'Emilien';
128 Prenoms(7) := 'Adama';
129 Prenoms(8) := 'Alima';
130 Prenoms(9) := 'Ousmane';
131 Prenoms(10) := 'Mohamed';
132
133 NomsRue(1) := 'Boulevard Mansart';
134 NomsRue(2) := 'Avenue Alain Savary';
135 NomsRue(3) := 'Rue des Rossoirs';
136 NomsRue(4) := 'Republique';
137 NomsRue(5) := 'Rue Nelson Mandela';
138
139 FOR i IN 1..100 LOOP
140     -- G n r a t i o n a l a t o i r e d ' u n n o m e t d ' u n p r n o m
141     v_nom := Noms(TRUNC(DBMS_RANDOM.VALUE(1, 10)));
142     v_prenom := Prenoms(TRUNC(DBMS_RANDOM.VALUE(1, 10)));
143
144     -- G n r a t i o n a l a t o i r e d e n u m r o s d e t l p h o n e
145     v_numero_tel := TO_CHAR(TRUNC(DBMS_RANDOM.VALUE(1000000000, 9999999999)
146 ));
147
148     -- G n r a t i o n a l a t o i r e d ' u n e a d r e s s e e n e f f e c t u a n t d e s p e r m u t a t i o n s
149     v_adresse := NomsRue(TRUNC(DBMS_RANDOM.VALUE(1, 5)) || ', ' || TO_CHAR(
150 TRUNC(DBMS_RANDOM.VALUE(1, 100))));
151
152     -- Rest of your code
153     -- ...
154
155 END LOOP;

```

```

155     COMMIT;
156 END RemplirBaseDeDonnees;
157 -- Cr ez un d clencheur BEFORE UPDATE sur la table Compte
158 CREATE OR REPLACE TRIGGER VerifierSoldeAvantRetrait
159 BEFORE UPDATE ON Compte
160 FOR EACH ROW
161 DECLARE
162     SoldeCompte NUMBER;
163 BEGIN
164     -- R cup rez le solde actuel du compte
165     SELECT Solde INTO SoldeCompte
166     FROM Compte
167     WHERE CompteID = :NEW.CompteID;
168
169     -- V rifiez si la nouvelle valeur du solde est inf rieur e au solde actuel
170     IF :NEW.Solde < SoldeCompte THEN
171         -- Emp chez la mise      jour du solde en g n rant une erreur
172         RAISE_APPLICATION_ERROR(-20002, 'Solde insuffisant pour effectuer le
retrait. ');
173     END IF;
174 END;
175 /
176
177
178 -- Cr ez un d clencheur AFTER UPDATE sur la table Compte
179 CREATE OR REPLACE TRIGGER EnregistrerModificationSolde
180 AFTER UPDATE ON Compte
181 FOR EACH ROW
182 DECLARE
183     MontantTransaction NUMBER;
184     TypeTransaction VARCHAR2(20);
185 BEGIN
186     -- Calculez le montant de la transaction
187     MontantTransaction := :NEW.Solde - :OLD.Solde;
188
189     -- D terminez le type de transaction en fonction du montant
190     IF MontantTransaction > 0 THEN
191         TypeTransaction := 'Depot';
192     ELSIF MontantTransaction < 0 THEN
193         TypeTransaction := 'Retrait';
194     ELSE
195         -- Aucune modification de solde, pas de transaction
196         RETURN;
197     END IF;

```

```
198
199      -- Enregistrez la transaction
200      INSERT INTO Transaction (TransactionID, CompteID, TypeDeTransaction,
Montant, DateTransaction)
201      VALUES (SequenceTransaction.NEXTVAL, :OLD.CompteID, TypeTransaction, ABS(
MontantTransaction), SYSDATE);
202 END;
203 /
```