Rapport du projet de données réparties

Kawtar LYAMOUDI, Nabil SNIDI et Nihal BELMAKHFI

Département Sciences du Numérique - classe hpc et big data - deuxième année
UE Systèmes Concurrents et Communicants
2022-2023

Table des matières

1 Introduction

Dans le cadre de ce projet, nous allons mettre en pratique les concepts de programmation distribuée appris en cours en développant un système de partage d'objets basé sur la cohérence à l'entrée. Ce système permettra aux applications Java d'accéder de manière efficace aux objets partagés en utilisant des copies locales de ces objets. Il sera mis en œuvre par un ensemble d'objets Java distribués qui communiqueront entre eux en utilisant Java/RMI pour implémenter le protocole de gestion de la cohérence.

2 Architecture du Système :

Ce service représente les objets en utilisant des descripteurs (instances de la classe SharedObject) qui possèdent un champ "obj" qui pointe vers l'instance Java partagée ou un groupe d'objets. Toute référence à une instance partagée doit passer par cette indirection (similaire aux "stubs" dans le système Javanaise).

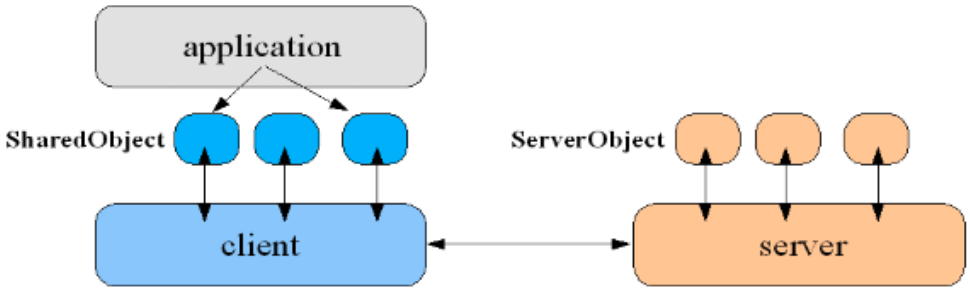


FIGURE 1 – Architecture du service de partage d'objets répartis et dupliqués

Au premier stade, cette indirection est visible pour le programmeur qui doit adapter sa façon de programmer. Au second stade, des "stubs" seront implémentés pour cacher cette indirection. Le service est organisé de la manière suivante : il utilise une classe appelée **SharedObject** pour accéder aux objets partagés. Cette classe fournit des méthodes telles que **lock_read()**, **lock_write()** et **unlock()** pour mettre en place la cohérence depuis l'application. Il est important de noter que dans ce service, les verrous imbriqués ne sont pas gérés : un **lock_read()** ou **lock_write()** sur un objet doit être suivi d'un **unlock()** pour pouvoir reprendre le même verrou sur le même objet à nouveau. Un SharedObject contient un entier "id" qui est un identifiant unique alloué par le système (ici, le serveur centralisé) lorsque l'objet est créé et une référence "obj" vers l'objet lorsqu'il est cohérent. La classe Client fournit des services pour créer ou trouver des objets dans un serveur de noms (comme le Registry RMI).
Un exemple d'application (Irc.ava) qui utilise un objet partagé de la classe >**Sentence** est fourni. Cette application sera utilisée pour tester le projet, mais la classe Sentence devra être modifiée et d'autres jeux de tests devront être implémentés pour pousser les tests plus loin.

2.1 SharedObject

Il s'aggit une implémentation en Java d'un objet partagé, qui permet à plusieurs programmes d'accéder à un objet partagé et de maintenir la cohérence entre eux. La classe possède plusieurs attributs tels que **obj** qui contient l'objet, **id** qui est l'identifiant unique de l'objet partagé, **attente** qui est un booléen utilisé pour attendre et **lock** qui est un entier représentant l'état du verrou de l'objet. L'état du verrou peut être l'un des suivants : NL (Pas de verrou local), RLC (Verrou de lecture mis en cache), WLC (Verrou d'écriture mis en cache), RLT (Verrou de lecture pris), WLT

(Verrou d'écriture pris) et RLT WLC (Verrou de lecture pris et verrou d'écriture mis en cache).

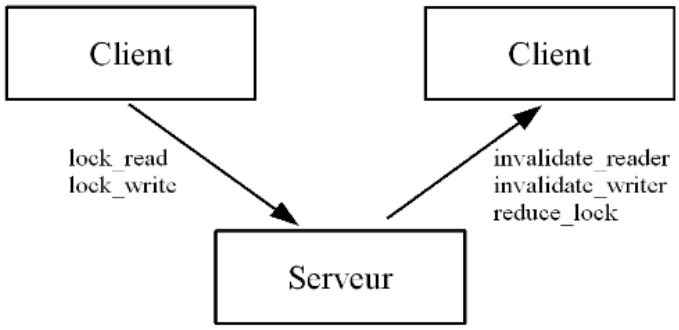


FIGURE 2 – Appel de méthods Client1 - Serveur - Client2

La classe possède également plusieurs variables globales pour l'état du verrou, par exemple NL signife "Pas de verrou" et RLC signifie "Verrou de lecture mis en cache". Il y a également des méthodes pour verrouiller l'objet en lecture ou en écriture, ainsi que des méthodes pour obtenir et définir l'identifiant et l'objet. Il y a également une méthode pour invalidate reader et invalidate writer qui permet au serveur d'invalider un objet partagé en passant par la couche cliente. Il y a une attention particulière pour gérer les demandes croisées entre les clients et le serveur.

2.2 Client

Cette classe est constituée des méthodes : init, lookup, register et create et des méthodes définies dans l'interface Client_itf : lock_read, lock_write, reduce_lock, invalidate_reader, invalidate_write.

-init : L'objectif de la fonction **init** est d'initialiser un client pour un système distribué. Il récupère le registre en utilisant le numéro de port et recherche l'objet serveur en utilisant l'URL construite, puis il instancie un nouveau client et crée un nouveau HashMap pour stocker les objets partagés.

-lookup : Dans la méthode **lookup** sur l'objet serveur "monServeur" en passant en paramètre une chaîne de caractères "name". Cette méthode renvoie un identifiant (id) associé à cette chaîne de caractères. Si l'identifiant est supérieur ou égal à zéro, cela signifie qu'il existe un objet partagé correspondant à ce nom. Le code crée alors un nouvel objet "SharedObject" avec cet identifiant, puis l'ajoute à la table "sharedObjects" en utilisant cet identifiant comme clé. Enfin, il affiche un message indiquant que la recherche de l'objet a été effectuée.

-register : La méthode **register** enregistre un objet partagé avec un nom donné sur le serveur. Si l'enregistrement échoue, elle récupère son nouvel identifiant. Puis, elle ajoute à nouveau l'objet partagé à la table des objets partagés du client.

-create : Par la méthode **create**, on crée un objet partagé et l'affecte à la variable "so", qui est initialement définie à null appelle la méthode "create" sur l'objet "Server" en passant en paramètre l'objet "o". La méthode renvoie un "id" qui est utilisé pour créer un nouvel objet partagé en utilisant l'id et l'objet "o". Le nouvel objet partagé est ensuite ajouté à une HashMap appelée "sharedObjects" avec l'id en tant que clé et l'objet partagé en tant que valeur.

-lock_read : La méthode **lock_read** permet de verrouiller en lecture un objet partagé sur le serveur en utilisant son identifiant, et retourne cet objet verrouillé.

-lock_write : La méthode **lock_write** permet de verrouiller en écriture un objet partagé sur le serveur en utilisant son identifiant, et retourne cet objet verrouillé.

-reduce_lock : Dans la méthode **reduce_lock()**,on applique un verrou de réduction sur un objet partagé localement qui porte un identifiant spécifique (id). La méthode utilise une fonction "reduce_lock()" pour verrouiller l'objet, puis elle retourne l'objet verrouillé.

-invalidate_reader : La **invalidate_reader** méthode récupère l'objet de la collection en utilisant l'identifiant "id" comme clé, puis appelle la méthode "invalidate_reader" sur cet objet. Cette méthode réclame l'invalidation d'un lecteur.

-invalidate_writer : La méthode **invalidate_writer** récupère un objet partagé dans la table "sharedObjects" en utilisant son identifiant, appelle la méthode "invalidate_writer" sur cet objet, et lève une exception RemoteException. Cette méthode réclame l'invalidation d'un ecrivain.

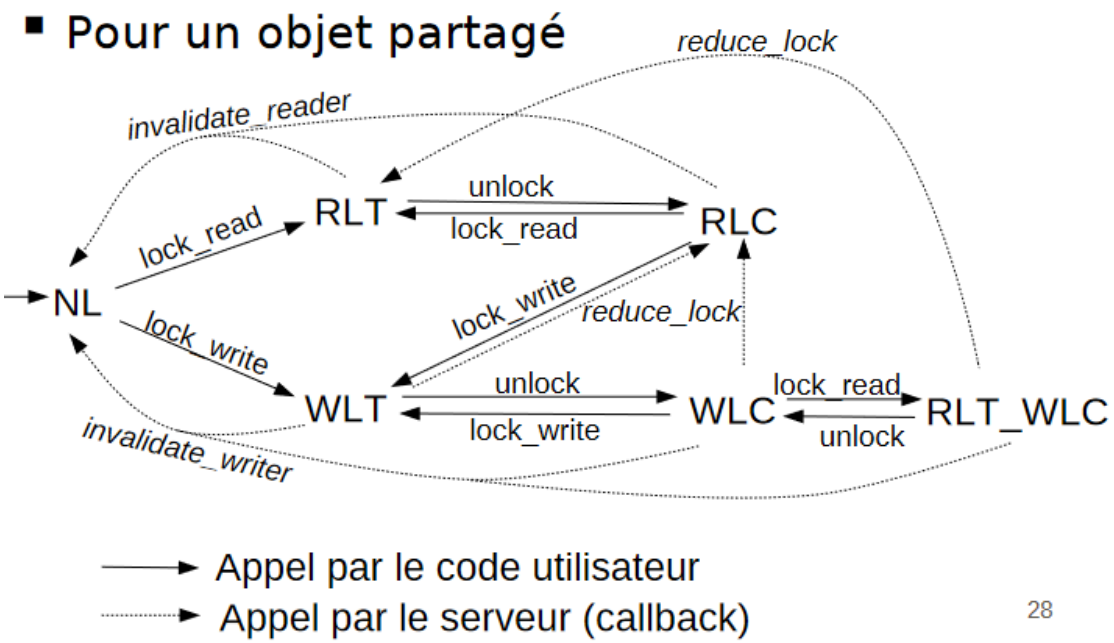


FIGURE 3 – point de vue client

2.3 Server

Cette classe est constituée des méthodes définies dans l'interface Server_itf : lookup, register,create deliveruniqueId(),lock_read, lock_write, et finalement le main pour lancer la classe Server.

-lookup : La méthode **lookup** permet de récupérer l'identifiant d'un objet partagé associé à un nom donné dans un HashMap

-register : La méthode **register** permet d'enregistrer un nom associé à un identifiant dans un HashMap.

-create : La méthode **create** crée un nouvel objet "ServerObject" en utilisant l'objet "o" passé en paramètre et en appelant la méthode "deliverUniqueId()" pour obtenir un identifiant unique pour l'objet. Ensuite, la méthode enregistre l'objet "ServerObject" dans un objet "serverObjects" (qui est une collection de type HashMap) en utilisant l'identifiant unique comme clé et l'objet "ServerObject" comme valeur.

-create : La méthode **create** crée un nouvel objet "ServerObject" en utilisant l'objet "o" passé en paramètre et en appelant la méthode "deliverUniqueId()" pour obtenir un identifiant unique pour l'objet. Ensuite, la méthode enregistre l'objet "ServerObject" dans un objet "serverObjects" (qui est une collection de type HashMap) en utilisant l'identifiant unique comme clé et l'objet "ServerObject" comme valeur.

-noubeauId : La méthode **noubeauId** permet de générer un identifiant unique.

-lock_read : La méthode **lock_read** permet à un client de verrouiller en lecture un objet partagé sur le serveur.

-**lock_write :** La méthode **lock_write** permet à un client de verrouiller en écriture un objet partagé sur le serveur.

-**Le main** permet de lancer un serveur RMI en définissant un port, en créant une instance de "Registry", en créant une instance de la classe "Server" et en associant cette instance à une URL spécifique. En cas d'exception, il affiche un message d'erreur et le serveur n'est pas lancé.

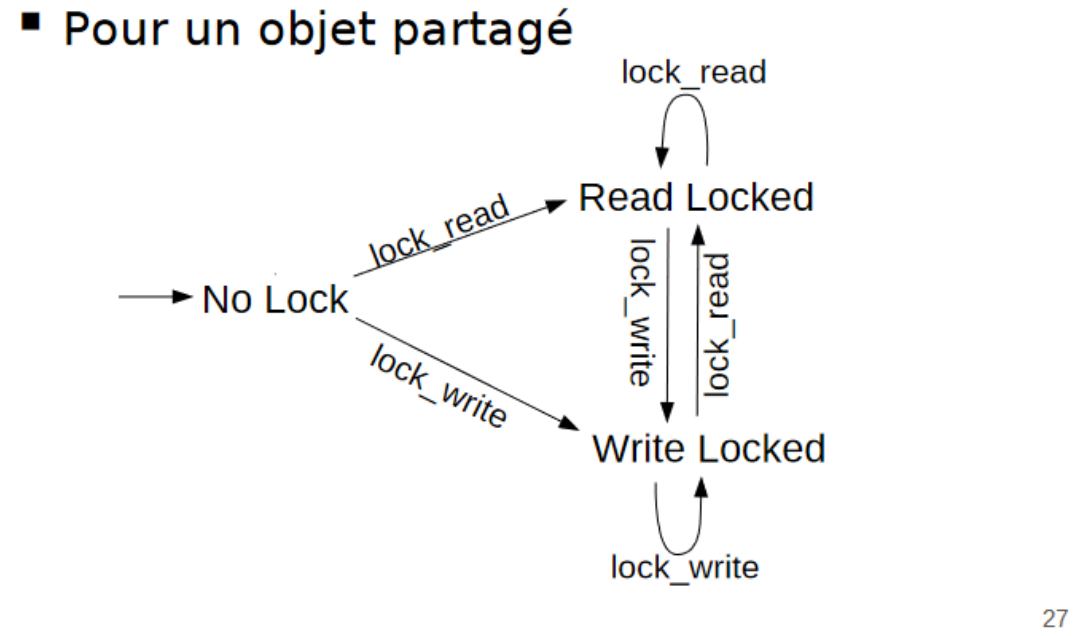


FIGURE 4 – point de vue serveur

2.4 ServerObject

La classe ServerObject est l'équivalent de la classe SharedObject du point de vue du Serveur. Ce-là veut dire que chaque SharedObject crée par le Client est mis sous le contrôle de cohérence d'un ServerObject crée par le Server. Il est définit par son id et son obj commun avec le SharedObject. Et deux autres attributs qui permettent de garder l'état de mémoire cohérent, notamment "monEcrivain" qui indique le client écrivain si l'objet est en écriture, et la liste de lecteurs "mesLecteurs" si l'objet est en lecture. On a défini également une variable enumeration :

public enum typeVerrou {NoLock, ReadLock, WriteLock} ;

qui permet d'indiquer le type de Verrou et appliquer les traitement ($invalidate_reader$,$invalidate_writer$, $reduce_lock$...) à partir de ça valeur. Les appels au Serveur sont transféré par la suite au ServerObject correspondant. On définit alors les méthodes suivantes :

-**lock_read :** Méthode pour que les clients peuvent demander un verrou en lecture d'un SharedObject. Le serveur est responsable de recevoir la demande $lock_read$ du client et appelle cette méthode du ServerObject avec une référence au client afin de pouvoir communiquer avec lui par la suite dans le cas du besoin.

-**lock_write :** Même chose que le $lock_read$ mais c'est pour demander un verrou en écriture.

3 Les étapes du travail

3.1 Etape 1

Dans la première étape du projet, nous allons implémenter un service de gestion d'objets partagés répartis. Pour cette première version, les applications utiliseront explicitement les SharedObject. Plusieurs applications pourront accéder de manière concurrente au même objet, ce qui nécessitera la mise en place d'un schéma de synchronisation globalement cohérent pour le service que nous implantons. Pour utiliser un objet, chaque application récupère une référence à un SharedObject en utilisant le serveur de nom. Nous ne gérons pas la gestion des références aux objets partagés dans les objets partagés eux-mêmes. Cela signifie que chaque application doit gérer ses propres références aux objets partagés et les mettre à jour en cas de modification de l'objet partagé. Cela peut entraîner des problèmes de synchronisation, qui seront résolus en implémentant un schéma de synchronisation globalement cohérent pour le service.

3.2 Etape 2

Dans cette deuxième étape, on vise à faciliter l'utilisation des SharedObjects pour les développeurs en implémentant un générateur de "stubs". Nous avons créé un "stub" appelé Sentence stub qui hérite des méthodes de verrouillage de SharedObject et qui implémente l'interface Sentence itf. Nous avons modifié la classe client pour ajouter un traitement au niveau des classes create et lookup qui permet de générer automatiquement les stubs. Nous avons également défini une classe StubGenerator qui permet de soulager les développeurs de l'utilisation des SharedObjects en générant automatiquement les stubs. En somme, l'objectif de la deuxième étape est bien la simplification de l'utilisation des SharedObjects pour les développeurs en créant des "stubs" automatiquement à l'aide d'une classe StubGenerator pour générer ces stubs de manière automatique. Cela facilitera l'accès aux objets partagés pour les applications tout en garantissant une synchronisation globale cohérente.

3.3 Etape 3

Pour cette troisième étape, on veut prendre en compte le stockage des références vers des objets partagés dans nos objets partagés. Cependant, lorsqu'on copie un objet partagé O1, qui contient une référence à un autre objet O2, entre deux machines M1 et M2, un problème se pose. O1 contient une référence à un stub de O2 sur la machine M1. Après la copie, il est nécessaire que la copie de O1 contienne également une référence au stub de O2 sur la machine M2. Pour cela, il faut adapter les primitives de sérialisation des stubs afin que lorsqu'un stub est sérialisé sur M1, l'objet référencé ne soit pas copié, et lorsqu'il est désérialisé sur M2, le stub soit installé de manière cohérente sur M2 (sans qu'il y ait plusieurs stubs pour un même objet sur la même machine).

4 Les algorithmes des opérations essentielles

4.1 Les méthodes de Client

4.1.1 L'iniatliastion d'un client

La méthode init 5

```
32  //=============================================//
33  //  Initialisation du Client par les applications  //
34  //=============================================//
35
36      // initialization of the client layer
37      public static void init() {
38          try {
39              int port = Registry.REGISTRY_PORT;
40              registry = LocateRegistry.getRegistry(port);
41              String localHost = InetAddress.getLocalHost().getHostName();
42              Client.URLDuServeur = "//" + localHost + ":" + port + "/monServeur";
43              monServeur = (Server_itf) registry.lookup(URLDuServeur);
44              monClient = new Client();
45              sharedObjects = new HashMap<Integer, SharedObject>();
46          } catch (RemoteException e) {
47              e.printStackTrace();
48          } catch (Exception e) {
49              e.printStackTrace();
50          }
51      }
52
```

FIGURE 5 – Algorithmes Essentielles : init Client

4.1.2 La création d'un objet partagé "SharedObject"

La méthode create 6

```
118 //=============================================================//
119 //  Méthode create: creation du so avec l'objet o              //
120 //=============================================================//
121
122     // creation of a shared object
123     public static SharedObject create(Object o) {
124         //on crée un objet partagé et l'affecte à la variable "so",
125         //qui est initialement définie à null
126         //appelle la méthode "create" sur l'objet "Server" en passant
127         //en paramètre l'objet "o". La méthode renvoie un "id" qui
128         //est utilisé pour créer un nouvel objet partagé en utilisant
129         //l'id et l'objet "o". Le nouvel objet partagé est ensuite ajouté
130         //à une HashMap appelée "sharedObjects" avec l'id en tant que clé et
131         SharedObject so = null;
132         try{
133             int id = monServeur.create(o);
134             // so = new SharedObject(o, id);
135             so = new SharedObject(id,o);      Lyamoudi Kawtar, 4 days ago • E
136             //on ajoute à la hashmap
137             sharedObjects.put(so.getId(), so);
138             System.out.println("Client: create effectué");
139         }
140         catch (RemoteException exp) {
141             System.out.println("Exception dans create");
142             exp.printStackTrace();
143     }
144     return so;
145 }
```

FIGURE 6 – Algorithmes Essentielles : create Client

4.1.3 L'enregistrement de l'identifiant d'un objet dans la HashMap

La méthode register 7

```java
//============================================================//
//  Méthode register: enregistrement du so crée sur le server //
//============================================================//

    // binding in the name server
    public static void register(String name, SharedObject_itf so) {
        // il y a un "casting" (conversion) de l'objet SharedObject_itf
        //en SharedObject,
        //Ensuite, l'objet partagé est enregistré dans le serveur de noms
        // en utilisant la méthode "register" avec les paramètres "name"
        //et l'ID de l'objet partagé.
        try {
            SharedObject soCast = (SharedObject) so;
            monServeur.register(name, soCast.getId());
            System.out.println("Client: register effectué");
        }
        catch (RemoteException exp) {
            //Dans le cas ou on catch une remote exception
            //C'est qu'on a crée un objet de trop
            //Alors il faut refaire le lookup et l'enlever le doublon
            //de notre table locale
            sharedObjects.remove(so.getId());
            int newid;
            try {
                newid = monServeur.lookup(name);
                SharedObject s = (SharedObject) so;
                sharedObjects.put(newid,s);
                so.setId(newid);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
    }
}
```

FIGURE 7 – Algorithmes Essentielles : register Client

4.1.4 La recherche d'un objet

La méthode lookup 8

```
53  //============================================================//
54  //  Méthode lookup:chercher si le so était déjà crée avant    //
55  //============================================================//
56
57      // lookup in the name server
58      public static SharedObject lookup(String name) {
59          SharedObject objetCree = null ;
60          // pour récupérer un objet partagé à partir d'un nom donné.
61          //on va utiliser lock_read" pour verrouiller l'objet correspondant,
62          //puis on va créer un nouvel objet "SharedObject" à partir de cet
63          //objet et de l'identifiant. on déverrouillera ensuite l'objet
64          //verrouillé et on ajoutera le nouvel objet "SharedObject" à une table
65          //locale appelée "sharedObjects"
66          try{
67          int id = monServeur.lookup(name); // id associé à name
68          //Si l'identifiant est valide (supérieur ou égal à zéro)
69          if (id>=0) {
70              // objetCree = new SharedObject(null, id);
71              //on ajoute l'objet creé à la table
72              sharedObjects.put(id, objetCree);
73          }
74          System.out.println("Client: lookup effectué");
75
76      }
77   catch (RemoteException exp) {
78      System.out.println("Exception dans SharedObject");
79  }
80  return objetCree;
81
82      }
83
```

FIGURE 8 – Algorithmes Essentielles : lookup Client

4.2 les méthodes de Serveur

4.2.1 La création d'un objet partagé "SharedObject"

La méthode create 9

```java
//============================================================//
//  Méthode create: crée un nouveau serverObject dans la table//
//============================================================//

    public int create(Object o) throws java.rmi.RemoteException{
    //La méthode crée un nouvel objet "ServerObject" en utilisant
    //l'objet "o" passé en paramètre et en appelant la méthode
    //"nouveauId()" pour obtenir un identifiant unique pour
    //l'objet. Ensuite, la méthode enregistre l'objet "ServerObject"
    //dans un objet "serverObjects" (qui est une collection de type HashMap)
    //en utilisant l'identifiant unique comme clé et l'objet "ServerObject"
    //comme valeur.
        try {
            createlock.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Server: create demand received");
        int id = nouveauId();
        ServerObject servobj = new ServerObject(o,id);
        serverObjects.put(id, servobj);
        createlock.release();
        return id;
    }
```

FIGURE 9 – Algorithmes Essentielles : create Server

4.2.2 L'enregistrement de l'identifiant d'un objet dans la HashMap

La méthode register 10

```java
//============================================================//
//  Méthode register: ajoute un ServerObject à la table   //
//============================================================//

    public void register(String name, int id) throws java.rmi.RemoteException{
        try {
            registerlock.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Server: register demand received");
        if (nomServeur.containsKey(name)){
            serverObjects.remove(id);
            registerlock.release();
            throw new RemoteException();
        }
        else {
            nomServeur.put(name, id);
            registerlock.release();
        }
    }
```

FIGURE 10 – Algorithmes Essentielles : register Server

4.2.3 La recherche d'un objet

La méthode lookup 11

```java
//=============================================================//
// Méthode lookup: cherche un nom de so sur la table des noms //
//=============================================================//

    public int lookup(String name) throws java.rmi.RemoteException{
        System.out.println("Server: lookup demand received");
        //on cherche l'objet partagé associé au nom
        //on trouve l'id de name
        Integer id = nomServeur.get(name);
        //on verifie si ça existe
        if (id != null) {
            int resultat = (int) id;
            return resultat;
        }
        else{
            throw new RemoteException();
        }
    }
```

FIGURE 11 – Algorithmes Essentielles : lookup Server

4.3 Les méthodes de ServerObject

4.3.1 Obtention de verrou en lecture

La méthode lock_read 12

```java
//=====================================================================//
// Méthode lock_read: demande un verrou en lecture pour un so d'identifiant id  //
//=====================================================================//

    @Override
    public Object lock_read(Client_itf client) {       You, 4 days ago • Projet TDL terminé …
            //le traitement dépends si on a un client qui écrit ou pas
            //On peut savoir ceci à partir de type de verrou
            if(this.Verrou == typeVerrou.WriteLock) {
                //On récupère l'écrivain
                Client_itf writer = monEcrivain;

                //Et puis on envoie une requete de reduce_lock(),
                //qui permet de passer vde mode écriture en lecture
                this.obj = writer.reduce_lock(id);

                //On ajoute notre ecrivain aux lecteurs
                mesLecteurs.add(writer);
            }

            //Maintenant on ajoute le client en tous cas au liste des lecteurs
            mesLecteurs.add(client);

            //Et on change le type du Verrou, en réinitialisant la valeur de l'écrivai à null
            this.monEcrivain = null;
            this.Verrou = typeVerrou.ReadLock;

            //Maintenant on réincrémente le sémaphore
            this.pasdEcrivain.release();
            this.MutexReadWrite.release();
        }
        catch(RemoteException exception){
            System.out.println("Server : lock_read error");
            exception.printStackTrace();
        }

        //On retourne la version mise-à-jour de notre objet
        System.out.println("Server : lock_read executed successfully");
        return obj;
    }
```

FIGURE 12 – Algorithmes Essentielles : lock_read ServerObject

4.3.2 Obtention de verrou en écriture

La méthode lock_write 13

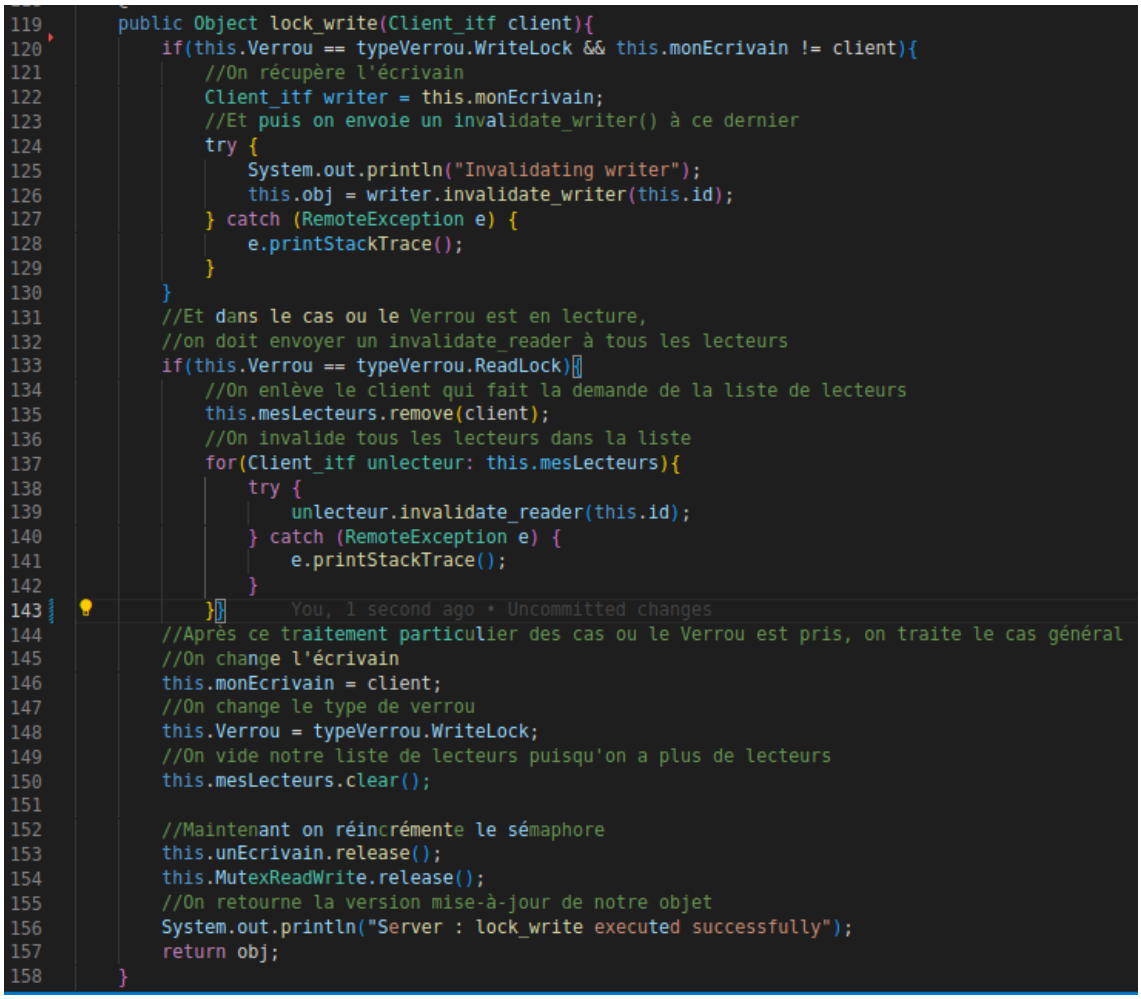


FIGURE 13 – Algorithmes Essentielles : lock_write ServerObject

4.4 Les méthodes de SharedObject

4.4.1 Obtention de verrou en lecture

La méthodes lock_read 14

```java
// invoked by the user program on the client node
public void lock_read() {
    boolean modifier = false;
    synchronized (this) {
        while (this.accesPossible) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        switch(lock) {
            case 0:
                modifier = true;
                lock = 3;
                break;
            case 1:
                lock = 3;
                break;
            case 2:
                lock = 5;
                break;
            default:
                break;
        }
        if (modifier) {
            obj = Client.lock_read(id);
        }
    }
}
```

FIGURE 14 – Algorithmes Essentielles : lock_read Shared Object

4.4.2 Obtention de verrou en écriture

La méthode lock_write 15

```
89      public void lock_write() {
90          boolean demander = false;
91          synchronized (this) {
92              while (this.accesPossible) {
93                  try {
94                      wait();
95                  } catch (InterruptedException e) {
96                      e.printStackTrace();
97                  }
98              }
99              switch (lock) {
100                 case 0:
101                     demander = true;
102                     lock = 4;
103                     break;
104                 case 1:
105                     demander = true;
106                     lock = 4;
107                     break;
108                 case 2:
109                     lock = 4;
110                     break;
111                 default:
112                     break;
113             }
114         }
115         if (demander) {
116             obj = Client.lock_write(id);
117         }
118     }
```

FIGURE 15 – Algorithmes Essentielles : lock_write Shared Object

4.4.3 Libération de verrou

La méthode unlock16

```
120  //============================================================//
121  //  Méthode libérer le verrou après un lock_read ou lock_write//
122  //============================================================//
123
124      // invoked by the user program on the client node
125      public synchronized void unlock() {
126          switch(lock) {
127              case 3:
128                  lock = 1;
129                  break;
130              case 4:
131                  lock = 2;
132                  break;
133              case 5:
134                  lock = 2;
135                  break;
136              default:
137                  break;
138          }
139          try {
140              notify();
141          } catch (Exception e) {
142              e.printStackTrace();
143          }
144      }
```

FIGURE 16 – Algorithmes Essentielles : unlock Shared Object

4.4.4 Passage d'écriture en lecture

La méthode reduce_lock 17

```
151     public synchronized Object reduce_lock() {
152         this.accesPossible = true;
153         switch (lock) {
154             case 4:
155             while (lock == 4) {
156                 try{
157                     wait();
158                 } catch (InterruptedException e) {
159                     e.printStackTrace();
160                 }
161             }
162             case 2:
163                 lock = 1;
164                 break;
165             case 5:
166                 lock = 3;
167                 break;
168             default:
169                 break;
170         }
171 
172         this.accesPossible = false;
173 
174         try {
175             notify();
176         } catch (Exception e) {
177             e.printStackTrace();
178         }
179         return obj;
180     }
```

FIGURE 17 – Algorithmes Essentielles : reduce_lock Shared Object

4.4.5 Invalidation d'un lecteur

La méthode invalidate_reader 18

```
187     public synchronized void invalidate_reader() {
188
189         this.accesPossible = true;
190         switch (lock) {
191             case 3:
192                 while (lock == 3) {
193                     try {
194                         wait();
195                     } catch (InterruptedException e) {
196                         e.printStackTrace();
197                     }
198                 }
199
200             case 1:
201                 lock = 0;
202                 break;
203             default:
204                 break;
205         }
206
207         this.accesPossible = false;
208         try {      You, 35 seconds ago • Uncommitted changes
209             notify();
210         } catch (Exception e) {
211             e.printStackTrace();
212         }
213     }
```

FIGURE 18 – Algorithmes Essentielles : invalidate_reader Shared Object

4.4.6 Invalidation d'un écrivain

La méthode invalidate_writer19

```
219     public synchronized Object invalidate_writer() {
220         this.accesPossible = true;
221         switch (lock) {
222             case 4:
223                 while (lock == 4) {
224                     try {
225                         wait();
226                     } catch (InterruptedException e) {
227                         e.printStackTrace();
228                     }
229                 }
230             case 2:
231                 lock = 0;
232                 break;
233             case 5:
234                 while (lock == 5) {
235                     try {
236                         wait();
237                     } catch (InterruptedException e) {
238                         e.printStackTrace();
239                     }
240                 }
241                 lock = 0;
242                 break;
243             default:
244                 break;
245         }
246 
247         this.accesPossible = false;
248         try {
249             notify();
250         } catch (Exception e) {
251             e.printStackTrace();
252         }
253         return obj;
254     }
```

FIGURE 19 – Algorithmes Essentielles : invalidateShared Object

5 Tests réalisés

Pour pousser les tests plus loin, on a définit 3 tests qui permettent de tester la synchronisation et la cohérence de l'état de mémoire pour différents cas d'erreurs :

5.1 Test d'écrivains

-**TestEcrivain :** Le principe de ce test est de lancer N fois des $lock_write$ sur un SharedObject s. A chaque fois qu'on obtient effectivement s en lock_write, on incrémente sa valeur. Comme ça a la fin, on peut savoir le nombre de requêtes qui n'était pas prises en compte. Par exemple si on choisi N = 1000, et on lance le test à partir de 10 terminaux, alors, on doit avoir une valeur final de s de 10000. Si on a que 9500 par exemple, alors 500 lock_write ont échoué et on a un problème.

```java
public class TestEcrivain {

    Run | Debug
    public static void main(String argv[]) {
        |       You, 1 hour ago • Etape1 100%
        if (argv.length != 1) {
            System.out.println("java TestEcrivain <nb d'écritures>");
            return;
        }
        int N = Integer.parseInt(argv[0]);

        // initialize the system
        Client.init();

        // look up the ecrivain_i object in the name server
        // if not found, create it, and register it in the name server
        SharedObject s = Client.lookup(name: "ecrivain_i");
        if (s == null) {
            Integer monint = 0;
            s = Client.create(monint);
            Client.register(name: "ecrivain_i", s);
        }

        //On lock notre shared object en ecriture, on écrit quelque chose dedans, puis on le unlock
        for(int i =0 ; i< N ; i++) {
            // lock the object in write mode
            s.lock_write();
            Integer ancienneVal = (Integer) s.obj;
            s.obj = ancienneVal + 1;

            //System.out.println("Le nombre d'écriture courant:" + s.obj);
            // unlock the object
            s.unlock();
        }
        System.out.println("Le nombre d'écriture effectué est:" + s.obj);
    }
}

```

FIGURE 20 – Tests réalisés : Test Écrivains

5.2 Test de lecteurs

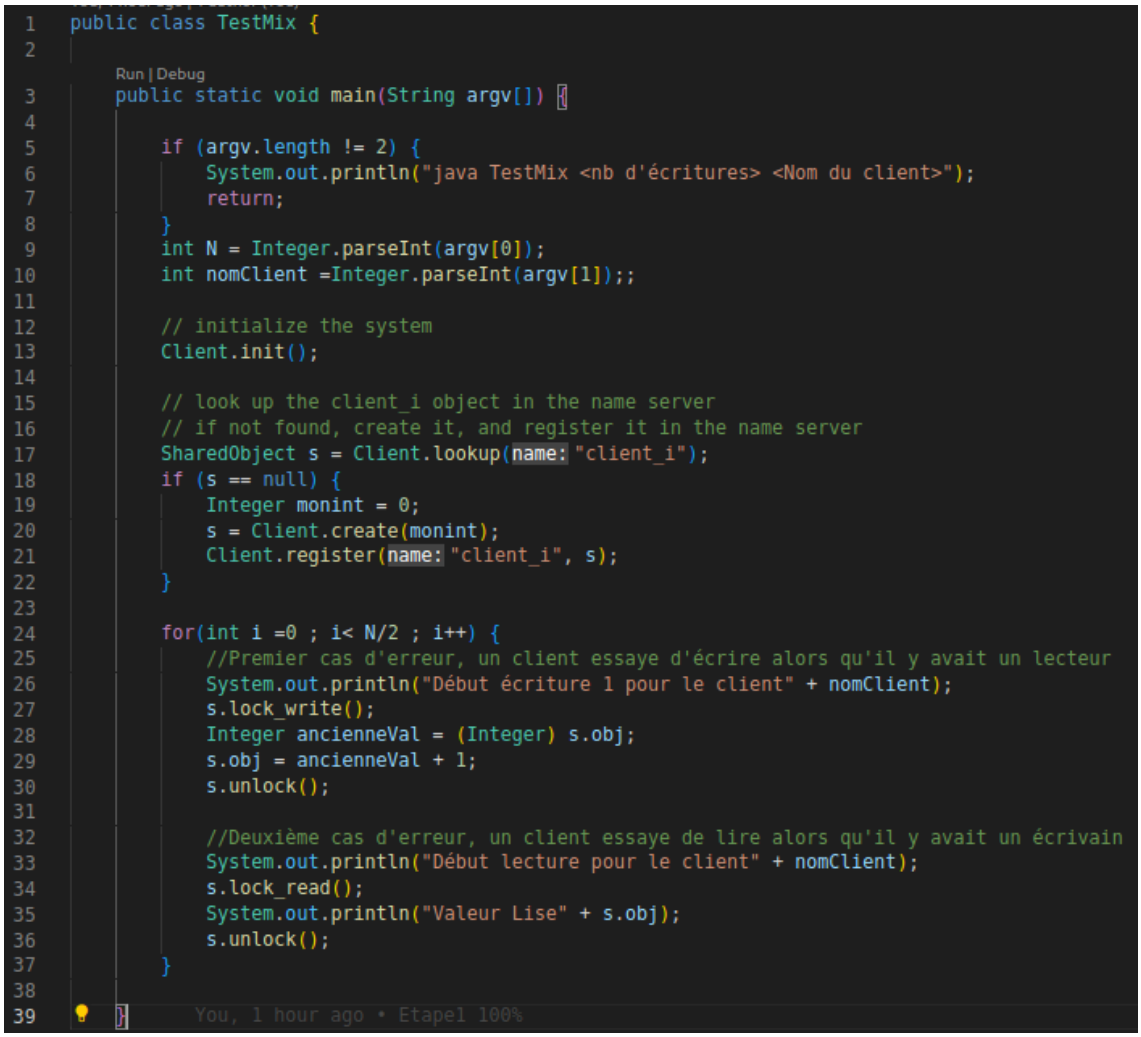
-**TestLecteurs :** Le principe de ce test est le même que le précédent, il suffit de lancer N fois des $lock_read$ sur un SharedObject s. On ajoute des sleep entre les lectures et on vérifie si la valeur lise reste la même. Comme ça a la fin, on peut savoir si les lock_read fonctionne et il n'y a pas de bug lors de l'utilisation ds sémaphores.

```java
public class TestLecteurs {
    Run | Debug
    public static void main(String argv[]) {
        if (argv.length != 1) {
            System.out.println("java TestLecteurs <nb d'écritures>");
            return;
        }
        int N = Integer.parseInt(argv[0]);
        // initialize the system
        Client.init();
        // look up the lecteur_i object in the name server
        // if not found, create it, and register it in the name server
        SharedObject s = Client.lookup(name: "lecteur_i");
        if (s == null) {
            Integer monint = 1;
            s = Client.create(monint);
            Client.register(name: "lecteur_i", s);
        }

        //On lock notre shared object en ecriture, on écrit quelque chose dedans, puis on le unlock
        for(int i =0 ; i<= N ; i++) {
            // lock the object in write mode
            s.lock_read();
            Integer ancienneVal1 = (Integer) s.obj;
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                System.out.println("Exception dans le système: Thread not working properly");
                e.printStackTrace();
            }
            Integer ancienneVal2 = (Integer) s.obj;

            //On compare les deux valeurs après une certaine période
            if (ancienneVal1 != ancienneVal2) {
                System.out.println("Il y a une erreur de lecture");
            }
            // unlock the object
            s.unlock();
        }
        System.out.println("Lecture bien effectué, valeur lise: " + s.obj);
```

FIGURE 21 – Tests réalisés : Test Lecteurs

5.3 Test avancé

-**TestMix :** Le test le plus avancé qu'on a implanté. Son principe est de lancer N/2 fois des écritures suivi par des lectures. Cela implique que lancer ce test à partir de plusieurs terminaux va nous permettre de tester tous les cas problématiques, puisqu'il y aura des lecteurs suivi d'écrivains suivis de lecteurs, et il y aura également des demandes simultanés d'écritures et des lectures.



```java
public class TestMix {

    Run | Debug
    public static void main(String argv[]) {

        if (argv.length != 2) {
            System.out.println("java TestMix <nb d'écritures> <Nom du client>");
            return;
        }
        int N = Integer.parseInt(argv[0]);
        int nomClient =Integer.parseInt(argv[1]);;

        // initialize the system
        Client.init();

        // look up the client_i object in the name server
        // if not found, create it, and register it in the name server
        SharedObject s = Client.lookup(name: "client_i");
        if (s == null) {
            Integer monint = 0;
            s = Client.create(monint);
            Client.register(name: "client_i", s);
        }

        for(int i =0 ; i< N/2 ; i++) {
            //Premier cas d'erreur, un client essaye d'écrire alors qu'il y avait un lecteur
            System.out.println("Début écriture 1 pour le client" + nomClient);
            s.lock_write();
            Integer ancienneVal = (Integer) s.obj;
            s.obj = ancienneVal + 1;
            s.unlock();

            //Deuxième cas d'erreur, un client essaye de lire alors qu'il y avait un écrivain
            System.out.println("Début lecture pour le client" + nomClient);
            s.lock_read();
            System.out.println("Valeur Lise" + s.obj);
            s.unlock();
        }

    }       You, 1 hour ago • Etape1 100%
```

FIGURE 22 – Tests réalisés : Test Lectures et écritures simultanés

Les test représente un seule client(une seule application) qui demande plusieurs fois des verrous en lecture ou en écriture du SharedObject s. Donc on a écrit des scripts bash (executer.sh, executerTest2.sh, executerTest3.sh) qui permettent de lancer chaque test (séparèment des autres) 10 fois. C'est une simulation de 10 clients en compétition pour un seule SharedObject s. Voici un exemple de script bash écrit pour lancer le test TestEcrivain 10 fois.

6 Point délicats

Pour l'étape 1, la mise en place correcte de la synchronisation s'avérer plus dur que ce qu'on avait pensé. Au début, on avait pas eu besoin de beaucoup de temps pour faire marcher le test basique IRC. Mais par la suite, on s'est rendus compte que le travail n'était pas terminé. En effet, après avoir mis en place le premier test TestEcrivain 20, on a remarqué que les applications crée à chaque fois des SharedObject indépendants pour lesquels l'accès était fait indépendamment. Il n'y avait pas d'accès concurrent ! On s'est vite rendu compte que le problème prévient de la méthode register() du client et du serveur, qui crée à chaque fois un autre ShredObject d'identifiant différent pour des SharedObjects du même nom. Alors on a changé cette méthode au niveau du Server pour vérifier si l'objet existait déjà sur la liste des noms, et lever une exception RemoteException s'il existe déjà. Cette exception est catched dans la méthode register du Client, et relance un lookup par la suite pour récupérer l'identifiant de l'objet crée précédemment.

Par la suite, on a aussi eu des problèmes de synchronisation après avoir testé l'étape 1, ce qui nous a poussé à faire plusieurs modifications dans les différentes classes afin de régler ces problèmes. Un exemple d'exception qu'on avait eu était au niveau du lock_read, qui s'exécuter correctement une seule fois et par la suite on pouvait plus accéder à l'objet en lecture. Finalement le problème survenait d'une mauvaise utilisation des sémaphores.

Pour assurer la cohérence des états des SharedObject, on a du ajouté des Mutex (Exclusion Mutuelle) au niveau de plusieurs fonctions. Par exemple pour empêcher les applications de faire des lock_write parallèles et changer l'état de l'objet au même instant, on a ajouté un sémaphore "unEcrivain" qui permet de limiter les accès au code de la fonction et fait de sorte qu'un seul écrivain peut changer la valeur de notre SharedObject à la fois. On a ajouté des Mutex également pour les fonctions du SharedObject et entre les fonction register et create du Server.

7 Conclusion

7.1 Résumé du Travail

En résumé, ce projet a mis en pratique les principes de programmation répartie en développant un service de partage d'objets en utilisant la technique de duplication et la cohérence à l'entrée sur Java. Ce service améliore l'accès aux objets répartis et partagés pour les applications Java. Il a été implémenté en utilisant des objets Java répartis et Java/RMI, et en utilisant des descripteurs pour représenter les objets partagés. Les aspects de l'indirection et de la masque ont été également abordés (implémentation de stubs).

7.2 Difficultés rencontrées

Pourtant on a eu quelques difficultés, notamment pour la gestion du temps surtout qu'on a eu ce projet en parallèle avec plusieurs projets et en plein partiels. Le travail sur ce projet a été rendu difficile en raison des limitations techniques rencontrées avec nos ordinateurs personnels aussi. Les tests prenaient beaucoup de temps à s'exécuter, ce qui a ralenti notre capacité à développer et à valider notre travail. Cette difficulté était accentuée par le fait que nous étions un groupe de trois personnes travaillant pendant les vacances de Noel, et que le VPN de l'école ne fonctionnait pas. Cela a rendu plus difficile pour nous de travailler ensemble et de partager les fichiers nécessaires au développement du projet. Malgré ces obstacles, nous avons réussi à poursuivre le développement du projet en utilisant des solutions alternatives pour partager les fichiers et en nous organisant efficacement pour maximiser notre temps de développement.

7.3 Compétences aquises

En ce qui concerne ce que ce projet nous a apporté, ce projet nous a permis de mettre en pratique les différentes notions étudiées en cours, telles que les systèmes concurrents, RMI, etc. Il nous a donné l'opportunité de plonger dans les détails de ces concepts et de les utiliser pour résoudre les défis liés à la programmation répartie. En outre, ce projet nous a permis de travailler en équipe, ce qui nous a aidés à développer nos compétences en matière de communication et de résolution de problèmes en groupe. Nous avons également eu l'occasion d'utiliser différents outils tels qu'Eclipse, VsCode, LATEX pour améliorer nos compétences en matière de développement et de rédaction de documents. En général, ce projet a été une expérience enrichissante qui nous a permis de développer nos compétences professionnelles dans de nombreux domaines.