Projet Long
Intergiciels et Systèmes Concurrents et Communiquants
Un service de partage d'objets dupliqués en JAVA

Yoann ZIMÉRO
Thibaut ETIENNE
Groupes E et F

1er mai 2009

Résumé

Ce projet consiste en la réalisation en Java un service de partage d'objet par duplication reposant sur la cohérence à l'entrée ou encore entry consistency.
Les Applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux (ils s'effectuent sur les copies (réplicas) locaux des objets). Durant l'exécution, le sevice est mis en œuvre par un ensemble d'objets Java répartis qui communiquent au moyen de Java / RMI pour implanter le protocole de gestion de la cohérence.

Table des matières

1 Présentation générale du service

Dans ce service, les objets sont représentés par des descripteurs (instances de la classe SharedObject d'interface SharedObject_itf) qui possèdent un champ obj qui pointe sur l'instance (ou une grappe d'objets) Java partagée. Toute référence à une instance partagée doit passer par une telle indirection (comme avec les stubs dans le système Javanaise).
Dans une première étape, cette indirection est visible pour le programmeur qui doit adapter son mode de programmation. Dans une seconde étape, on implantera des stubs qui masquent cette indirection.

On organisera le service de la manière suivante :

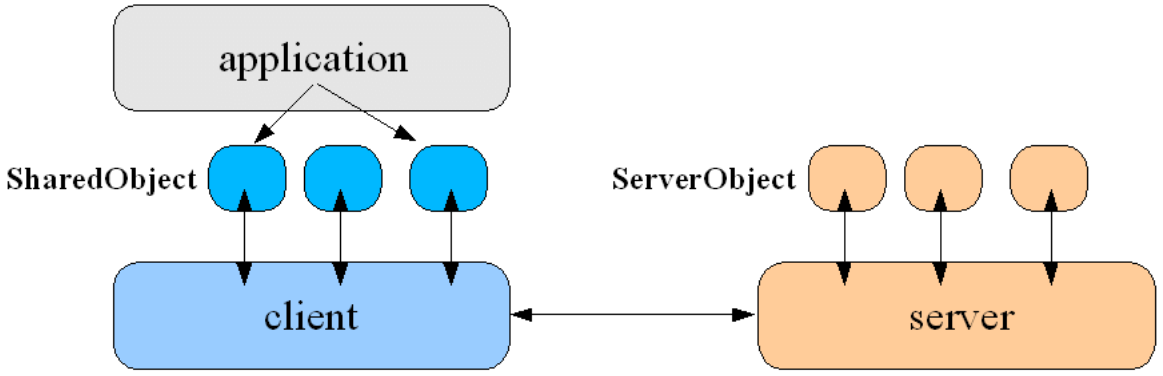


Fig. 1 – *Architecture du service de partage d'objets répartis et dupliqués*

L'utilisation d'un objet partagé se fait toujours avec une indirection à travers un objet de classe SharedObject. Cette classe fournit notamment des méthodes lock_read(), lock_write() et unlock() qui permettent de mettre en oeuvre la cohérence à l'entrée depuis l'application. Notons que dans le cadre de ce service, nous ne gèrerons pas les prises de verrou imbriquées : 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.
Un SharedObject contient notamment un entier id qui est un identifiant unique alloué par le système (ici ce sera le serveur centralisé) à la création de l'objet, ainsi qu'une référence obj à l'objet lorsqu'il est cohérent.

La couche (classe) appelée Client fournit les services pour créer ou retrouver des objets dans un serveur de noms (comme le Registry RMI).

Un exemple d'application (Irc.ava) utilisant un objet partagé de classe Sentence nous est fournie. On utilisera alors cette application pour tester notre projet, mais nous aurons non seulement à modifier cette classe pour pousser les tests plus loin, mais également implanter d'autres jeux de tests.

1.1 Les SharedObject

Les *SharedObject*, qui sont utilisés par les programmes pour tous les accès aux objets, contiennent des informations sur l'état des objets, du point de vue de la cohérence. On y trouvera notamment une variable entière *lock* qui signifie :

- NL (No local Lock) : 0
  Il n'y a dans ce cas là aucun verrou sur l'objet

- RLC (Read Lock Cached) : 1
  L'objet est en cache et était auparavant verrouillé en lecture

- WLC (Write Lock Cached) : 2
  L'objet est en cache et était auparavant verrouillé en écriture

- RLT (Read lock Taken) : 3
  L'objet est verrouillé en lecture

- WLT (Write Lock Taken) : 4
  L'objet est verrouillé en écriture

- RLT_WLC (Read Lock Taken and Write Lock Cached) : 5
  L'application a pris le verrou en écriture puis en lecture. Il s'agit d'un état de verrouillage intermédiaire entre WLC et RLT

Un verrou est dit *taken* s'il est pris par l'application. Il sera libéré lors du prochain *unlock*() et passera alors à l'état *cached*.

Le verrou est dit *cached* s'il réside sur le site sans être pris par l'application. Un verrou *cached* peut alors être pris par l'application sans communication avec le serveur.

L'état hybride RLT_WLC correspond à une prise de verrou en lecture par l'application alors que le *SharedObject* possédait un WLC.

Ainsi, en fonction de l'état de l'objet sur le site client, la demande d'un verrou nécessitera (ou pas) de propager un appel au serveur.

Afin de mettre en oeuvre la cohérence au sein de ce système, les méthodes lock_read() et lock_write() de la classe *SharedObject* ont besoin d'appeler des méthodes du serveur qui génèrent la cohérence. Ces appels passent par la couche cliente (classe *Client*).

On mettra également en oeuvre des méthodes permettant au serveur d'invalider un *SharedObject* (en passant par la couche client qui ne fera que rediriger les appels vers les bons SharedObject).

On ajoutera alors les méthodes suivantes :

- Object reduce_lock()
- void invalidate_reader()
- Object invalidate_writer()

Les appels de méthode d'un client au serveur et du serveur à un client sont représentés sur la figure suivante :

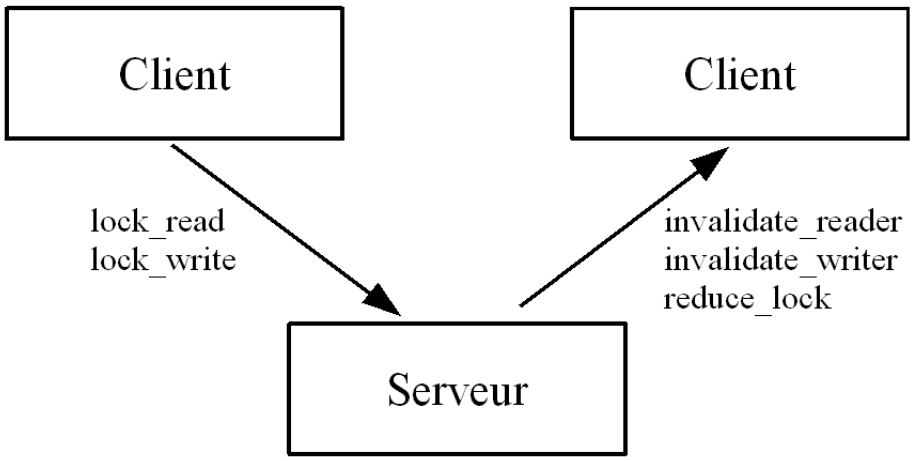


Fig. 2 – *Appels de méthode Client1 ⟶ Serveur ⟶ Client2*

ATTENTION : On fera attention au fait que les requêtes peuvent se croiser !

1.2 Le Client

La classe *Client* fournira donc les méthodes suivantes :
- **static Object lock_read (int id) :** demande d'un verrou en lecture au serveur, en lui passant l'identifiant unique de l'objet (le *SharedObject* devant être en No local Lock). Cette méthode retourne donc l'état de l'objet.
- **static Object lock_write(int id) :** demande d'un verrou en écriture au serveur, en lui passant l'identifiant unique de l'objet (le *SharedObject* devait être en No local Lock ou en Read Lock Cached). Cette méthode retourne donc l'état de l'objet si le *SharedObject* était en No local Lock.

Ces méthodes statiques de la classe *Client* ne font que propager ces requêtes au serveur, en ajoutant aux paramètres de la requête une référence au client (instance) lorsque cela est nécessaire. On ajoutera également des méthodes qui permettront au serveur d'effectuer des invalidations sur des écrivains ou des lecteurs ou de "passer le verrou de l'écriture à la lecture".

Ces méthodes sont alors les suivantes :
- **Object reduce_lock (int id) :** permet au serveur de réclamer le passage d'un verrou de l'écriture à la lecture
- **void invalidate_reader(int id) :** permet au serveur de réclamer l'invalidation d'un lecteur
- **Object invalidate_writer(int id) :** permet au serveur de réclamer l'invalidation d'un écrivain

1.3 Le Serveur

L'interface du serveur (distante) comprend donc les méthodes suivantes :
- Object lock_read(int id, Client_itf client)
- Object lock_write(int id Client_itf client)

Ces méthodes incluent une référence au client afin de pouvoir le rappeler ultérieureument. Sur le site serveur, la gestion de la cohérence d'un objet est attribuée à une instance de la classe *ServerObject*. On ajoutera alors d'autres méthodes, notamment pour implanter le service de nommage évoqué précédemment)

1.4 Les ServerObject

La classe *ServerObject* indique l'état de la cohérence de l'objet du point de vue serveur, avec notamment le client écrivain si l'objet est en écriture ou la liste des clients lecteurs si l'objet est en lecture (afin d'être en mesure de propager des invalidations). Les appels au serveur sont transférés au *ServerObject* concerné, ce *ServerObject* implantant une interface similaire :
- Object lock_read(Client_itf client)
- Object lock_write(Client_itf client)

La référence au client reçue par le serveur lui permet de réclamer un verrou et une copie de l'objet au client qui la possède.

1.5 Représentation interne des verrous

L'ensemble des méthodes de verrouillage et d'invalidation décrites précédemment nous conduisent alors à réaliser le "diagramme d'état" suivant :

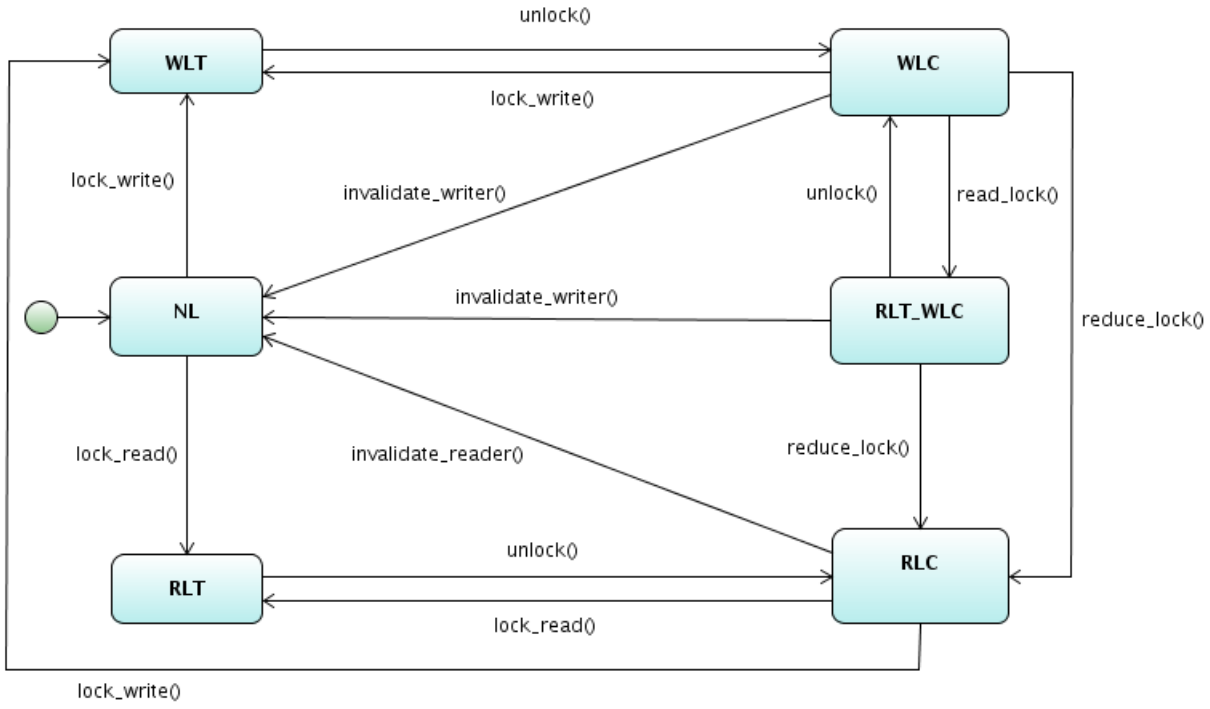


Fig. 3 – *Représentation de l'état du verrou sur un SharedObject*

2 Organisation du travail

2.1 Première étape

Nous implanterons le service de gestion d'objets partagés répartis. Dans cette première version, les *SharedObject* sont utilisés explicitement par les applications.
Plusieurs applications peuvent accéder de façon concurrente au même objet, ce qui nécessite de mettre en oeuvre un schéma de synchronisation globalement cohérent pour le service que nous implantons.

On suppose que chaque application voulant utiliser un objet en récupère une référence (à un *SharedObject*) en utilisant le serveur de nom. On ne gère pas le stockage de référence à des objets partagés dans les objets partagés.

2.2 Deuxième étape

On désire soulager le programmeur de l'utilisation des *SharedObject*. On doit donc implanter un générateur de "stubs".

Nous prendrons alors les hypothèses suivantes :

- un objet partagé est une classe sérialisable. Il s'agit de la classe métier de l'objet partagé.
- l'objet partagé est utilisable à partir de variables de type une interface (par convention, les méthodes métier de Sentence auquel on ajoute les méthodes de verrouillage. Sentence_itf hérite de SharedObject_itf mais Sentence n'implémente pas Sentence_itf, car la classe métier ne définit pas les méthodes de verrouillage (c'est le "stub" qui le fait).
- un "stub" est généré, appelé Sentence_stub. Ce stub hérite de SharedObject (donc des méthodes de verrouillage) et il implémente l'interface Sentence_itf.

2.3 Troisième étape

On désire prendre en compte le stockage de références (à des objets partagés) dans des objets partagés. Le problème qui se pose est la copie d'un objet partagé O1, qui inclut une référence à un objet O2, entre deux machines M1 et M2. O1 inclut une référence au stub de O2 sur la machine M1. Après la copie, il faut que la copie de O1 inclut la référence au stub de O2 sur la machine M2. Ceci nécessite d'adapter les primitives de sérialisation des stubs pour que, lorsqu'un stub est sérialisé sur M1, on ne copie pas l'objet référencé et lorsqu'il est désérialisé sur M2, le stub soit installé de façon cohérente sur M2 (sans installer plusieurs stubs pour un même objet sur la même machine).

Nous exploiterons (nous spécialiserons) la méthode Object readResolve().

3 Conception

Il s'agit alors de définir l'ensemble des intéractions entre les différents objets intervenant dans la réalisation du système. Comme nous l'avons vu dans la première partie, l'application manipulera des SharedObject et on fera le lien entre chaque application et le serveur via la couche (la classe) *Client*.

On peut alors se demander comment les intéractions entre ces différents objets peuvent se réaliser. En effet, même si la plupart des opérations de lectures et d'écritures doivent se faire localement chez les clients, de nombreuses intéractions entre chacun de ces clients et le serveur doivent être réalisées.

3.1 Création d'un SharedObject (méthode create)

Prenons l'exemple de la création d'un SharedObject par une Application. L'application passe alors par le client pour effectuer la demande de création d'un objet partagé. Le client retransmet cette demande au serveur. Ce dernier crée un ServerObject en lui attribuant un identifiant unique et retourne ensuite cet identifiant au client qui dispose de tous les éléments pour créer le ServerObject.

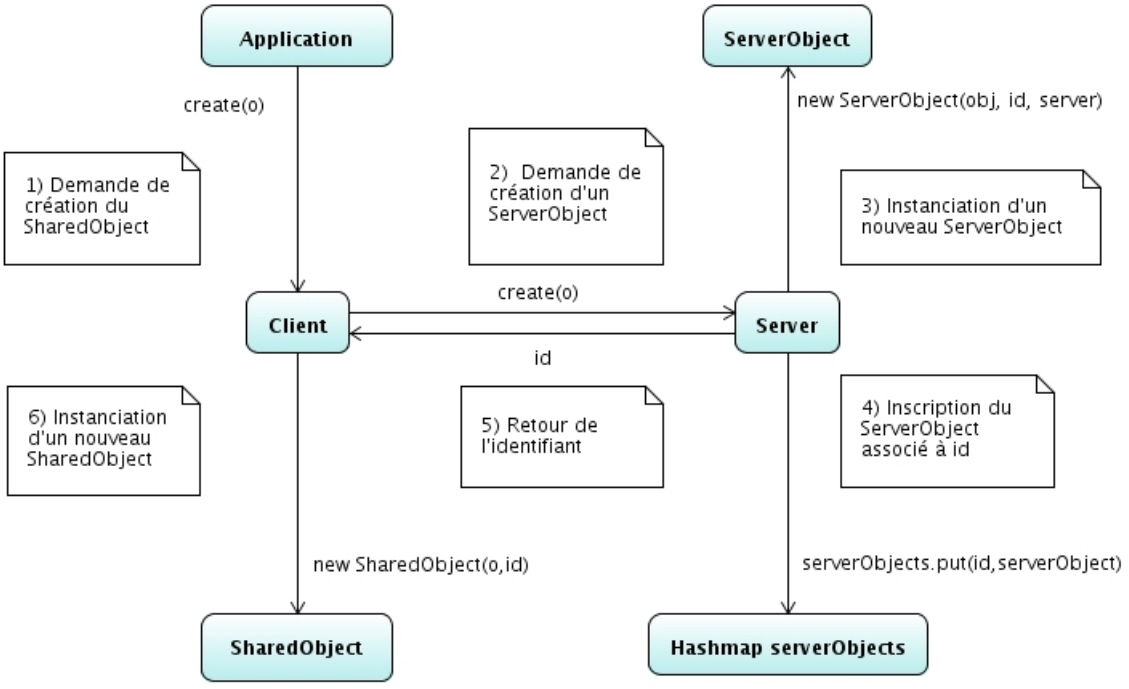


Fig. 4 – *Schéma de création d'un SharedObject*

3.2 Enregistrement de l'identifiant d'un objet dans la HashMap serverNames (méthode register)

De même, après la création d'un objet partagé, il faut l'enregistrer dans le serveur de nom. Le protocole d'enregistrement se fait alors en suivant le même parcours que précédemment.

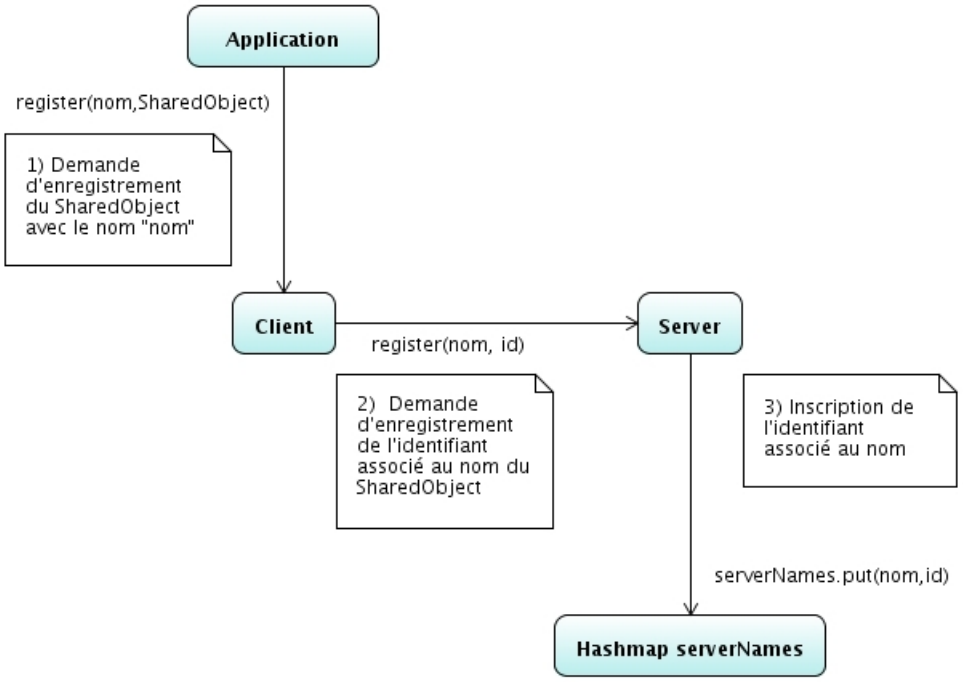


Fig. 5 – *Schéma d'enregistrement d'un SharedObject*

3.3 Recherche d'un objet (méthode lookup)

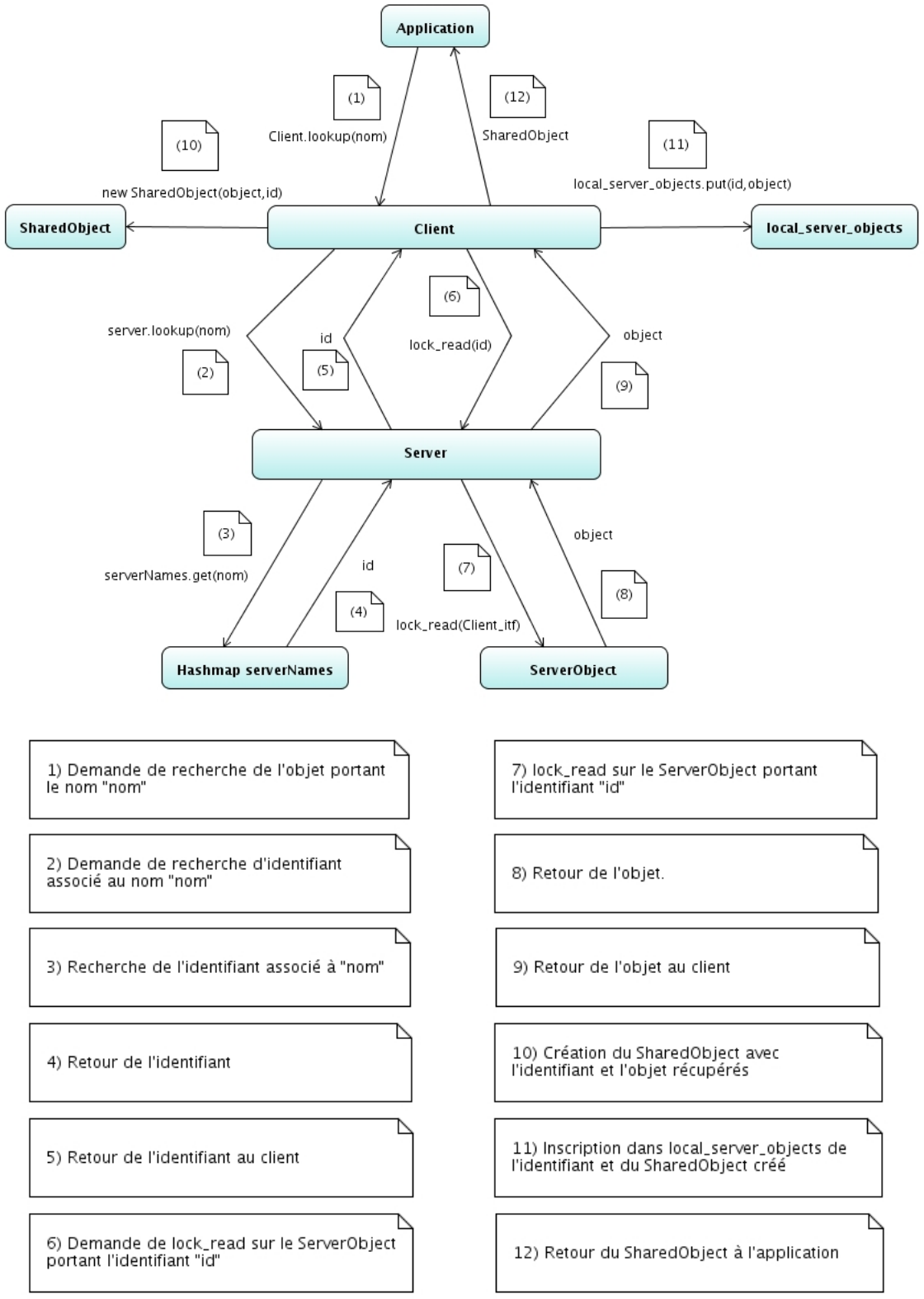


Fig. 6 – *Schéma de recherche d'un SharedObject*

3.4 Initialisation du client (méthode init())

Concernant l'initialisation du client, elle s'effectue en deux étapes :

- Recherche de l'enregistrement du serveur dans le serveur de nom grâce à la méthode Naming.lookup.
  On récupère ainsi toutes les informations nécessaires pour communiquer avec le serveur. Cependant chaque utilisateur doit connaître l'adresse exacte du serveur (ce qui semble assez logique... En effet comme dans le système mis en place par IRC, l'utilisateur doit rentrer lui-même l'URL (voir clients IRC tels que GunScript, mIRC, ...)

- Initialisation de HashMap local_shared_object contenant la clef id et l'objet SharedObject associé à cet identifiant
  Dans cette HashMap seront stockées en tant que clefs, l'identifiant unique de chaque objet, mais également l'ensemble des SharedObject associés. Cette HashMap nous sera alors utile lorsque le seveur appellera des méthodes d'invalidation et de réduction de verrou.

3.5 Verrouillage (méthodes lock_read/write, invalidate_reader/writer et reduce_lock)

Mais comment opère-t-on pour effectuer une demande de verrouillage en lecture ou en écriture auprès du serveur ? Tout d'abord, l'application effectue une demande de verrou en lecture ou en écriture auprès du SharedObject. Selon l'état de son verrou, le SharedObject demandera alors au client de transférer au Serveur sa demande de verrouillage. Le serveur aiguillera donc cette demande vers le ServerObject associé à l'identifiant placé en paramètre. Ce dernier (le ServerObject) effectuera alors, selon l'état de son propre verrou, des demandes d'invalidations ou de reduce_lock auprès du bon client. On n'oubliera pas bien sur de mettre à jour la liste des lecteurs et d'inscrire l'écrivain...

On obtient ainsi le schéma suivant :

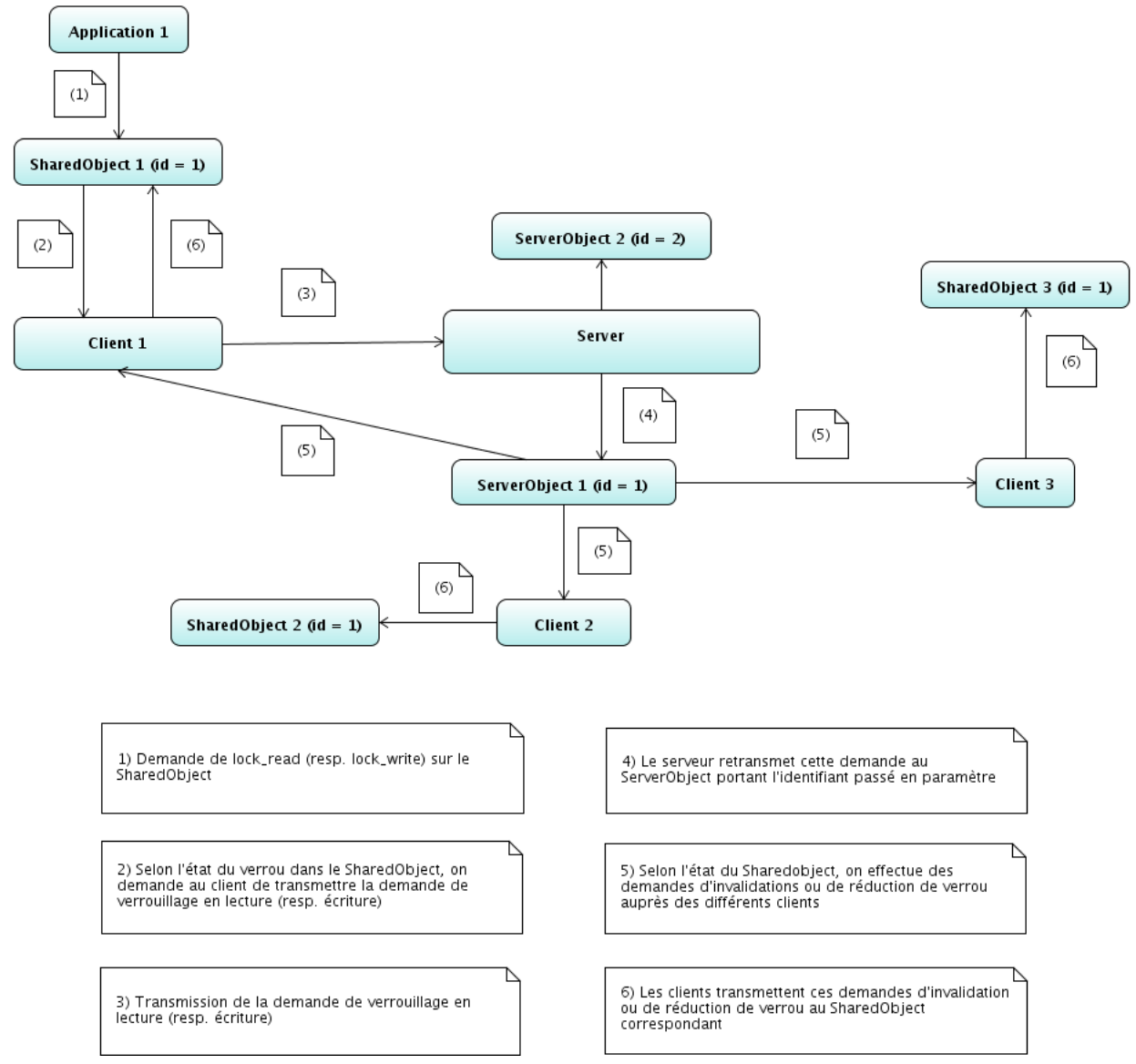


Fig. 7 – *Schéma de demande de verrouillage en lecture ou en écriture*

4 Codage

Dans cette partie, nous étudierons la manière de coder et les choix qui ont été faits. En effet, plusieurs choix ont été faits, plusieurs points méritent d'être éclaircis, notamment les choix de synchronisation pour éviter des dead_lock. De plus, en complément des schémas présentés ci-dessus, nous éclaircirons les points importants des différentes méthodes implémentées au cours de ce projet. Nous expliquerons alors les algorithmes mis en œuvre pour faire communiquer les clients et le serveur.
Tout d'abord, pourquoi l'utilisation des mots clés synchronized ? Quelle est l'utilité des méthodes wait() et notify() ?

4.1 L'utilité du mot clé "synchronized"

Le mot clé *synchronized* est utilisé pour mettre en œvre le concept de moniteur étudié précédemment en cours de systèmes concurrents. Le but est de maintenir un accès privilégié (pour ne pas dire unique dans notre cas) sur une *ressourcecritique*. Ainsi, l'utilisation des méthodes $wait()$ et $notify()$ se font dans le cadre de moniteurs réalisés en Java par des méthodes ou des bouts de code délimités par le mot clé synchronized.

La méthode $wait()$ implique donc que le processus qui veut accéder à la ressource critique attendra que cette *ressourcecritique* soit libérée de manière à pouvoir y accéder sans causer "d'incohérences" au sein du programme. Le thread rencontrant cette méthode $wait()$ se met alors en attente dans ce que l'on appelle le $wait - set$.

La méthode $notify()$ nous permettra alors de réveiller un des processus mis en attente dans le $wait - set$ et lui permettra ainsi d'accéder à la ressource. Cette méthode sera utilisée chez le client lorsqu'il souhaitera libérer le verrou en appelant la méthode $unlock()$. Nous aurions pu également utiliser $notifyAll()$ de manière à réveiller tous les threads mis en attente dans le $wait - set$...

Ainsi, ces méthodes relatives au concept de moniteur ont été utilisées notamment chez chacun des clients. Ces clients effectuant des lectures et des écritures sur chacun des objets partagés, l'utilisation de ce concept est nécessaire.

4.2 Les méthodes de la classe SharedObject

Le passage d'un SharedObject d'un état à un autre s'effectue au sein de la classe $SharedObject$ au cours des méthodes lock_read et lock_write. On peut alors se demander dans quel cas un objet atteint un état particulier. Pour répondre à cette question, on peut se reporter à la figure 3 présentée page 5.

Cependant, il n'est pas si simple de réaliser un tel passage. Plusieurs conditions doivent être remplies. C'est la raison pour laquelle ces méthodes sont importantes, non seulement car elles utilisent le principe de moniteur mis en évidence au paragraphe précédent, mais aussi parce qu'elles gèrent les différents états internes d'un SharedObject.

Nous verrons alors le code des méthodes mises en jeu dans le phénomène de verrouillage d'un objet partagé, de manière à voir comment ces transitions sont franchies, puis nous expliciterons nos choix quant-à la synchronisation.
Voyons maintenant comment coder ces méthodes.

4.2.1 La méthode lock_read()

```
public void lock_read() {

  boolean demanderLockRead = false;

  // Utilisation du mot-clef "synchronized" ici pour eviter
  // les problemes de synchronisation (dead lock notamment)
  synchronized(this) {

    // tant que l'attribut "attente" est vrai, on attend
    while(this.attente){
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Quand il devient faux, le SharedObject doit changer d'etat
    // Voir graphe des transitions entre les etats
    switch(this.lock){
      case NL :
        // Si on n'avait pas dans le cache l'objet valide,
        // il faudra en faire la demande via le client
        // par la methode lock_read(id)
        demanderLockRead = true;

        this.lock = RLT;          // NL => RLT
        break;

      case RLC :
        this.lock = RLT;          // RLC => RLT
        break;

      case WLC :
        this.lock = RLT_WLC;      // WLC => RLT_WLC
        break;

      default : break;
    }
  }

  // a la sortie si le SharedObject etait dans l'etat NL
  // on demande un lock_read au client en lui passant l'id
  if(demanderLockRead){
    this.obj = Client.lock_read(this.id);
  }
}
```

Fig. 8 – *Code de la méthode lock_read()*

Nous avons vu comment verrouiller un SharedObject en lecture. Qu'en est-il de l'écriture ?

4.2.2 La méthode lock_write()

```
public void lock_write() {

  boolean demanderLockWrite = false;

  // Utilisation du mot-clef "synchronized" ici pour eviter
  // les problemes de synchronisation (dead lock notamment)
  synchronized(this){

    // tant que l'attribut "attente" est vrai, on attend
    while(this.attente){
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Quand il devient faux, le SharedObject doit changer d'etat
    // Voir graphe des transitions entre les etats
    switch(this.lock){
      case NL :
        // Si on n'avait pas dans le cache l'objet valide,
        // il faudra en faire la demande via le client
        // par la methode lock_write(id)
        demanderLockWrite = true;
        this.lock = WLT;      // NL => WLT
        break;

      case RLC :
        demanderLockWrite = true;
        this.lock = WLT;      // RLC => WLT
        break;

      case WLC :
        this.lock = WLT;      // WLC => WLT
        break;

      default : break;
    }
  }
  // a la sortie si le SharedObject etait dans l'etat NL
  // on demande un lock_write au client en lui passant l'id
  if(demanderLockWrite){
    this.obj = Client.lock_write(this.id);
  }
}
```

Fig. 9 – *Code de la méthode lock_write()*

Ces deux méthodes constituent le coeur de la classe SharedObject. En effet, elles gèrent la politique de verrouillage d'un objet partagé : elles sont alors au centre de notre programme, faisant le lien entre l'application et le serveur.
Cependant, la mise en œvre d'une troisième méthode est nécessaire à cette politique de verrouillage d'un objet : il s'agit de la méthode *unlock*() qui permettra à l'application de relâcher le verrou sur l'objet partagé. Voyons alors comment serait codée cette méthode.

4.2.3 La méthode unlock()

```
public synchronized void unlock() {

  // Voir graphe des transitions entre les etats
  switch(this.lock){

    case RLT :
      this.lock = RLC;      // RLT => RLC
      break;

    case WLT :
      this.lock = WLC;      // WLT => WLC
      break;

    case RLT_WLC :          // RLT_WLC => WLC
      this.lock = WLC;
      break;

    default :
      break;
  }

  // on avertit que le verrou est relache
  try {
    notify();
  } catch(Exception e) {
    e.printStackTrace();
  }
}
```

Fig. 10 – *Code de la méthode unlock()*

Nous avons alors parlé de l'ensemble des méthodes qui permettent de gérer le verrou en lecture et en écriture. Cependant, le serveur (plus précisément les ServerObject) peuvent demander des invalidations en lecture, en écriture, mais également des réductions de verrou.

Il convient alors de coder ces méthodes dans la classe SharedObject. Voyons alors comment ces méthodes sont implémentées.

4.2.4 La méthode reduce_lock()

La méthode *reduce_lock*() permet de réduire les droits d'un client sur un objet partagé. En effet, si chez le client l'objet est en Write Lock Cached (WLC dans le code), alors on doit lui retirer la possibilité d'écrire, mais on lui laisse la possibilité de lire en le passant en RLT (s'il était en cours de lecture) ou en RLC (s'il n'utilisait pas l'objet partagé)

Voyons comment cette méthode sera implémentée.

```
public synchronized Object reduce_lock() {

  // on passe l'attribut attente a la valeur true
  // => pas de lock tant qu'il n'est pas passe a false
  this.attente = true;

  switch(this.lock) {
    // cas ou lock == WLT : tant que c'est le cas, on attend
    case WLT :
      while(this.lock == WLT) {
        try {
          wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
      // ATTENTION PAS DE BREAK !

    case WLC :           // WLC => RLC
      this.lock = RLC;
                 break;

    case RLT_WLC :       // RLT_WLC => RLT
      this.lock = RLT;
      break;

    default : break;
  }
  // on repasse l'attribut attente a la valeur false
  this.attente = false;

  // on avertit qu'on rend la main
  try {
    notify();
  } catch(Exception e) {
    e.printStackTrace();
  }

  // on retourne alors l'objet qui a ete modifie
  return obj;
}
```

Fig. 11 – *Code de la méthode reduce_lock()*

On remarquera alors que l'objet est retourné et sera renvoyé au serveur qui mettra à jour sa copie de l'objet. En effet, si le SharedObject se trouvait dans l'un des états WLC, WLT, RLT_WLC, on est certain que c'est ce client qui a la dernière "version" de l'objet partagé. Ainsi, en donnant le verrou en écriture à un autre client, on doit mettre à jour la version de l'objet dans le ServerObject.

4.2.5 Les méthodes invalidate_reader() et invalidate_writer()

Nous avons vu comment nous pouvions réduire les droits des utilisateurs de l'objet partagé, ou comment passer d'un état à un autre en utilisant les méthodes lock_read() et lock_write(). Cependant, lorsque le serveur doit invalider des lecteurs et des écrivains, comment réalise-t-il cette opération ?
Voyons ensemble comment les opérations d'invalidations sont effectuées en analysant le code de ces deux méthodes.

Commençons par la méthode invalidate_reader() :

```
public synchronized void invalidate_reader() {

  this.attente = true;

  switch(this.lock) {
    case RLT :          // tant que this.lock == RLT on attend
      while(this.lock == RLT) {
        try {
          wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }                 // ATTENTION : pas de break !!!

    case RLC :          // RLC => NL
        this.lock = NL;
        break;

    default : break;
  }

  // attente <- false et on avertit qu'on rend la main
  this.attente = false;
  try {
    notify();
  } catch(Exception e) {
    e.printStackTrace();
  }
}
```

Fig. 12 – *Code de la méthode invalidate_reader()*

Remarques :

– Il n'y a pas de break dans le premier cas du switch. En effet, lorsque l'on sort du cas RLT, on doit se retrouver en RLC. On applique donc à ce moment là le changement d'état pour le SharedObject.

– Nous ne renvoyons pas d'objet. En effet, le verrou était en lecture, donc on n'a pas la dernière version de l'objet partagé. Il est donc normal que cette méthode soit void.

Voyons maintenant comment nous avons réalisé la méthode invalidate_writer() :

```
public synchronized Object invalidate_writer() {

  this.attente = true;

  switch(this.lock) {
    case WLT :          // Tant que this.lock == WLT on attend
      while(this.lock == WLT) {
        try {
          wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }                 // ATTENTION : pas de break !!!

    case WLC :          // On peut atterir ici en venant du cas WLT
      this.lock = NL;   // WLC => NL
      break;
    case RLT_WLC:       // Tant que this.lock == RLT_WLC on attend
      while(lock == RLT_WLC) {
        try {
          wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
      this.lock = NL; // puis passage en NL
      break;
    default : break;
  }

  // attente <- false et on avertit qu'on rend la main
  this.attente = false;
  try {
    notify();
  } catch(Exception e) {
    e.printStackTrace();
  }
  return obj;
}
```

Fig. 13 – *Code de la méthode invalidate_writer()*

4.3 Les méthodes de la classe Client

Après avoir étudié les méthodes principales de la classe SharedObject, il nous faut alors voir les méthodes principales de la couche Client.
Tout d'abord, penchons nous un instant sur la méthode *init*() qui permettra d'initialiser le client au démarrage de l'application.

4.3.1 La méthode init() de la couche Client

La méthode *init*() permet d'initialiser le client au cours du démarrage de l'application. La question qui se pose est alors de savoir ce que doit contenir cette méthode. Nous nous sommes déjà penchés sur ce point au paragraphe 3.4. Résumons alors : récupérer les informations sur le serveur en utilisant la commande Naming.lookup() et initialiser la HashMap local_shared_objects.

Voyons alors les choix effectués pour l'implémentation de cette méthode :

```
public static void init() {

  System.out.println("Beginning Client Initialization");

  // si aucune instance de client n'a ete faite
  if (Client.instance == null) {
    // on cree une nouvelle instance de Client
    try {
      Client.instance = new Client();
    } catch (RemoteException e) {
      e.printStackTrace();
    }

    try {
      // on determine l'adresse URL du serveur
      Client.Server_URL = "//"
                + InetAddress.getLocalHost().getHostName()
                + ":" + Registry.REGISTRY_PORT
                + "/my_server";

      // Recherche du serveur dans le rmiregistry
      server = (Server_itf) Naming.lookup(Server_URL);

      // initialisation des HashMap
      local_shared_objects = new HashMap<Integer,SharedObject>();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println("Client Init. Successfully Completed.");
  }
  // s'il existe deja une instance de client on ne fait rien
}
```

Fig. 14 – *Code de la méthode init()*

4.3.2 Les méthodes create() et lookup() de la couche Client

Dans cette partie, nous traiterons le cas de la première étape uniquement, le générateur de stub devant être présenté avant la présentation des différentes modifications. En effet, au cours de la deuxième étape, nous aurons à modifier ces méthodes pour prendre en compte le stub généré par la classe *StubGenerator*. Voyons alors comment ces méthodes ont été réalisées dans la première étape :

```
public static SharedObject lookup(String name) {
  SharedObject soResult = null;
  try {
    // on recupere l'id de l'objet associe
    // au nom name chez le serveur
    int id = server.lookup(name);

    // si l'id n'est pas dans la table du cote
    // serveur alors on revoie null
    // sinon on renvoie le SharedObject associe
    if (id < 0) {
      soResult = null;
    } else {
      // Attention : on recupere l'objet en le
      // verrouillant en lecture
      Object o = lock_read(id);
      // Creation du SharedObject a partir de
      // l'objet recupere et de
      // l'id renvoye par server.lookup
      soResult = new SharedObject(id, o);
      soResult.setName(name);

      soResult.unlock(); // On n'oublie pas de deverrouiller
                         // l'objet...
      // On ajoute le SharedObject associe a id a
      // local_shared_objects pour pouvoir l'utiliser localement
      local_shared_objects.put(id, soResult);
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  return soResult;
}
```

Fig. 15 – *Code de la méthode lookup(String name)*

Remarquons que pour récupérer l'objet nous avons du faire appel à la méthode lock_read(), de manière à récupérer la dernière version valide de l'objet partagé. Il ne faudra cependant pas oublier de relâcher le verrou, sinon l'objet partagé restera verrouillé en lecture.
L'enregistrement auprès de *local_shared_objects* est également nécessaire pour que le client connaisse cette objet.
Cette méthode suit donc un déroulement logique de manière à récupérer la dernière version de l'objet partagé sans causer de problèmes concernant le déroulement "normal" de l'application.

Etudions les choix réalisés pour la conception de la méthode create() :

```
public static SharedObject create(Object o) {
  SharedObject so = null;
  try {
    // creation sur le serveur du ServerObject
    // et recuperation de l'id
    int id = server.create(o);
    // on cree le SharedObject a partir de l'objet
    // et de l'id recupere
    so = new SharedObject(id,o);
    // on ajoute la cle id et le SharedObject so
    //a la HashMap local_shared_objects
    local_shared_objects.put(so.getId(), so);
  } catch (RemoteException e) {
    e.printStackTrace();
  }
  return so;  // et bien sur on retourne le SharedObject
              //defini a partir de o
}
```

Fig. 16 – *Code de la méthode create(Object o)*

Nous verrons alors par la suite comment modifier cette méthode de manière à prendre en compte le stub généré par la classe StubGenerator.
Nous avons alors respecté le schéma établi précédemment au paragraphe 3.1 concernant la conception de la méthode create.

4.4 La couche Server

Cette couche réalise le “routage” des requêtes depuis le client vers le ServerObject portant le bon identifiant.
Pour réaliser ce “routage”, nous avons fait le choix de deux attributs statiques :

```
// nameServerObject => idServerObject
private static HashMap<String,Integer> serverNames;

// idServerObject => ServerObject
private static HashMap<Integer,ServerObject> serverObjects;
```

Fig. 17 – *Deux HasMap de la classe Server*

L’application cherchera à appeler la méthode lookup() de la couche Client. Or nous avons vu que cette méthode lookup() chez le Client faisait appel à la méthode lookup() de la classe Server, récupérant ainsi l’identifiant associé au nom de objet partagé.

En effet, voici la signature de notre méthode lookup() dans la classe Server :

```
public int lookup(String name) throws RemoteException ;
```

Fig. 18 – *Signature de la méthode lookup() dans la classe Server*

L’entier retourné est donc l’identifiant de l’objet partagé qui porte le nom *name*. Il est donc nécessaire que la méthode register associe ce nom à l’identifiant du ServerObject dans la HashMap *serverNames* lors de la création d’un ServerObject.

De même la méthode create() enregistrera le ServerObject dans la HashMap *serverObjects* avec pour clé son identifiant unique. La question qui se pose est alors de savoir comment créer un identifiant unique. Nous avons fait le choix d’un attribut *current_id* qui sera incrémenté à chaque appel de create() grâce à la méthode deliverUniqueId().

```
private static int current_id = 0;

private int deliverUniqueId() {
  return Server.current_id++;
}
```

Fig. 19 – *Attribut current_id et méthode deliverUniqueId() de la classe Server*

Venons en maintenant à l'initialisation du serveur. Le serveur doit être lancé et enregistré dans le rmi registry pour que les clients puissent interagir avec lui. En effet, chaque client devra connaître le serveur pour communiquer avec lui. On réalisera alors l'initialisation du serveur de la manière suivante :

```
public static void main(String[] args) throws UnknownHostException{
  System.out.println(">>>>>> Server Init. <<<<<<");
  try {
    // on choisit le port et l'url par defaut
    int port = Registry.REGISTRY_PORT;
    String url = InetAddress.getLocalHost().getHostName();

    LocateRegistry.createRegistry(port);
    Server server = new Server();
    String Server_URL = "//" + url + ":" + port +"/my_server";

    // definition du lien entre Server_URL et l'objet Remote server
    Naming.rebind(Server_URL, server);
  } catch (Exception e) {
    e.printStackTrace();
  }
  System.out.println(">>>>>> Server Ready <<<<<<");
}
```

Fig. 20 – *Méthode main de la classe Server*

4.5 La classe ServerObject

Le passage d'un ServerObject d'un état à un autre s'effectue au sein de la classe *ServerObject* grâce aux méthodes *lock_read*() et *lock_write*().

4.5.1 Les attributs de la classe ServerObject

Tout d'abord, nous devons déterminer comment faire passer un ServerObject d'un état à un autre. Il nous faudra pour cela définir un attribut que l'on appellera *lock* qu'on choisira de type entier.

```
private int lock; // L'etat du ServerObject
//  Definitions des variables globales
private final static int NL = 0;
private final static int RL = 1;
private final static int WL = 2;
```

Fig. 21 – *Attributs relatifs à l'état du verrou d'un ServerObject*

On considèrera que si un écrivain possède le verrou en écriture, aucun client ne peut accéder à l'objet partagé. En effet, on devra invalider tous les lecteurs chaque fois qu'un écrivain demandera le verrou en écriture. Nous aurons donc en attribut pour chaque ServerObject, un seul écrivain. Cependant, lorsque le verrou est pris en lecture, alors il peut y avoir plusieurs lecteurs simultanés. On utilisera alors un Vector<>.

```
// Il peut y avoir plusieurs lecteurs simultanes
private Vector<Client_itf> lecteurs;
// Attention : un seul ecrivain
private Client_itf ecrivain;
```

Fig. 22 – *Attributs lecteurs et écrivain de la classe ServerObject*

Nous devrons définir un identifiant propre à chaque ServerObject, de manière à pouvoir l'identifier.

```
private int id; // l'identifiant unique du ServerObject
```

Fig. 23 – *Attribut id : identifiant du ServerObject*

Et bien entendu nous n'oublierons pas l'attribut principal, l'objet lui-même :

```
private Object o;
```

Fig. 24 – *Attribut id : identifiant du ServerObject*

4.5.2 Le constructeur de la classe ServerObject

Voyons tout d'abord comment nous avons défini le constructeur de la classe *ServerObject*.

```
public ServerObject(Object obj, int id) {
  this.id = id;
  this.o = obj;
  // initialisation du verrou a NL
  this.lock = NL;
  // aucun ecrivain a la creation du ServerObject
  this.ecrivain = null;
  // aucun lecteur a la creation du ServerObject
  this.lecteurs = new Vector<Client_itf>();
}
```

Fig. 25 – *Attribut id : identifiant du ServerObject*

4.5.3 Les méthodes lock_read() et lock_write() de la classe ServerObject

Intéressons nous maintenant à la méthode *lock_read*(). Voici les choix que nous avons faits pour la réalisation de la méthode lock_read() :

```
public synchronized Object lock_read(Client_itf client) {
  try {
    switch (this.lock) {
      // si l'objet est en WL, on doit passer en RLC ou RLT
      // chez le client : on doit appliquer un reduce_lock
      case WL:
        // on recupere la derniere version valide de l'objet
        this.o = ecrivain.reduce_lock(id);
        // et on doit ajouter "l'ancien ecrivain" en tant que lecteur
        this.lecteurs.add(this.ecrivain);
        break;
      default:
        break;
    }
    // il ne peut plus y avoir d'ecrivain
    this.ecrivain = null;
    // On doit ajouter le lecteur a la liste des lecteurs
    this.lecteurs.add(client);
    this.lock = RL; // On passe le ServerObject dans l'etat RL
  } catch (RemoteException e) {
    e.printStackTrace();
  }
  return o; // on retourne la derniere version de l'objet
}
```

Fig. 26 – *Méthode lock_read() de la classe ServerObject*

Cependant, nous devons également nous intéresser à la méthode lock_write() de la classe ServerObject.

```
public synchronized Object lock_write(Client_itf client) {
  switch(this.lock){
    case RL :
      this.lecteurs.remove(client);
      // on va invalider tous les lecteurs
      for (Client_itf cli : this.lecteurs) {
        try {
          cli.invalidate_reader(this.id);
        } catch (Exception e) {
          System.out.println("Le client s'est deconnecte");
        }
      }
      break;

    // si this.lock == WL alors on invalide l'ecrivain
    // et on recupere le dernier objet valide
    case WL :
      try {
        this.o = this.ecrivain.invalidate_writer(this.id);
      } catch (NullPointerException e) {
        System.out.println("Pas d'ecrivain !");
      } catch (RemoteException ee) {
        System.out.println("Le client s'est deconnecte");
      }
      break;

    default :
      break;
  }
  // il ne peut plus y avoir de lecteurs
  this.lecteurs.clear();
  this.ecrivain = client;
  this.lock = WL;
  return o; // On retourne l'objet
}
```

Fig. 27 – *Méthode lock_write() de la classe ServerObject*

Nous avons alors défini l'ensemble des méthodes nécessaires à la classe ServerObject. Il s'agira alors d'effectuer des tests poussés, pour nous assurer du bon fonctionnement de ce code. Nous nous intéresserons alors à ces tests dans le chapitre 5 de ce rapport.

4.6 Réalisation du générateur de stubs (Partie 2)

Dans la deuxième partie de ce projet, nous avons réalisé une classe *StubGenerator* qui permettra de soulager le programmeur de l'utilisation des SharedObject.
Nous nous sommes alors basés du package java.lang.reflect pour réaliser cette classe. Vous pouvez étudier cette classe en annexe, cependant, analysons les points principaux de cette classe

```
DEBUT

  Creer un fichier nom_de_la_classe + "_stub.java"

  Ecrire dans ce fichier "public class " + nom_de_la_classe
                  + "_stub extends SharedObject implements "
                  + nom_de_l_interface
                  + ", java.io.Serializable {";

  //constructeur
  Ecrire dans ce fichier "public " + nom_de_la_classe
                  + "_stub (int id, Object o) {"
                  + "super(id,o); }"

  //methodes
  Pour chaque methode Faire
    Ecrire dans ce fichier "public " + type_methode
                  + " " + nom_methode + " ("
                  + types_et_parametres_methodes
                  + nom_de_la_classe + " o = ("
                  + nom_de_la_classe + ") obj ;"

    Si type_methode == "void" Alors
      Ecrire dans ce fichier "o." + nom_methode + "("
                  + parametres_methode + ") ;"
    Sinon
      Ecrire dans ce fichier "return o." + nom_methode
                  + parametres_methode + ") ;"
    Fin Si

  Fin Pour

  Ecrire dans ce fichier "}"

FIN
```

Fig. 28 – *Algorithme de génération de stubs*

4.7 Modification des méthodes create() et lookup() de la classe Client

4.7.1 L'intérêt de ces modifications

Pour éviter au programmeur de manipuler directement des SharedObjects, en nous servant du générateur de stubs précédent, nous modifierons les méthodes create() et lookup() de la classe *Client*. Le but est alors de permettre au programmeur de manipuler les objets tels que des Sentence directement par exemple.

Prenons l'exemple de l'IRC fourni et voyons l'impact de ces modifications.

Au cours de la première étape de ce projet, notre application IRC manipulait directement des SharedObject.

```
SharedObject s = Client.lookup("IRC");
if (s == null) {
  s = Client.create(new Sentence());
  Client.register("IRC", s);
}
                new Irc(s);
```

Fig. 29 – *Extrait de l'IRC fourni*

Le but de cette étape est de permettre au programmeur de manipuler uniquement des Sentence et Sentence_itf (dans le cas de notre exemple bien sur).

```
Sentence_itf s = (Sentence_itf) Client.lookup("IRC");
if (s == null) {
  s = (Sentence_itf) Client.create(new Sentence());
  Client.register("IRC", s);
}
new Irc(s);
```

Fig. 30 – *Extrait de l'IRC modifié*

4.7.2 Les modifications apportées à la classe Client

Pour réaliser cette abstraction et la propager au niveau de la couche Client, nous devons alors modifier les méthodes *create*() et *lookup*() qui sont à la base de la création ou de la récupération d'un objet partagé.

Voyons alors comment ces modifications ont été effectuées. Définissons tout d'abord la méthode createStub() :

```
public static SharedObject createStub(Object o, int id) {

  SharedObject so = null;
  try {
    // Creation de la classe x_stub ou x represente la classe veritable
    // de o
    Class<?> classe = Class.forName(o.getClass().getName() + "_stub");

    // Recuperation du constructeur
    java.lang.reflect.Constructor<?> constructeur = classe
            .getConstructor(new Class[] { int.class, Object.class });

    // on cree le stub a partir de l'objet et de l'id recupere
    so = (SharedObject) constructeur
           .newInstance(new Object[] { id, o });

    // on ajoute la cle id et le SharedObject so a la HashMap
    // local_shared_objects
    local_shared_objects.put(so.getId(), so);

  } catch (Exception e) {
    e.printStackTrace();
  }
  return so;
}
```

Fig. 31 – *Modification de la méthode create()*

Puis voyons les modifications apportées à la méthode create() :

```
public static SharedObject create(Object o) {
  SharedObject so = null;
  // creation sur le serveur du ServerObject et recuperation de l'id
  try {
    int id = server.create(o);
    so = createStub(o, id);
  } catch (RemoteException e) {
    e.printStackTrace();
  }

  return so;
}
```

Fig. 32 – *Modification de la méthode create()*

Puis la méthode lookup() :

```
public static Object lookup(String name) {
  SharedObject soResult = null;
  try {
    // on recupere l'id de l'objet associe au nom name chez le serveur
    int id = server.lookup(name);

    // si l'id n'est pas dans la table du cote server alors on revoie
    // null sinon on renvoie le stub associe
    if (id >= 0) {
      // Attention : on recupere l'objet en le VERROUILLANT en lecture
      Object o = lock_read(id);
      soResult  = createStub(o, id);
      soResult.unlock();
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  return soResult;
}
```

Fig. 33 – *Modification de la méthode lookup()*

4.8 Réalisation de l'étape 3

Le but de cette étape est de permettre au sharedObject de faire référence à d'autres sharedObject ou stub. La difficulté est que l'objet partagé référencé doit être dans un état cohérent. Autrement dit, il faut s'assurer qu'il s'agit bien de la dernière version.
Pour y parvenir, il est nécessaire d'intervenir lors de la désérialisation de l'objet qui contient la référence. En effet, c'est à ce moment là qu'il est possible de récupérer chaque instance référencée par le SharedObject "conteneur". Dès lors, il est facile de vérifier si le client les possède déjà, grâce à l'identifiant. Si tel est le cas, alors il suffit de remplacer l'objet référencé par celui que connaît le client. Dans le cas contraire, il est nécessaire d'avertir le client qu'il doit créer le stub associé.

La méthode java permettant cette opération a pour signature : Object readResolve(). Elle est appelée automatiquement à chaque désérialisation d'un objet. Il suffit de la réimplanter afin qu'elle fasse ce que l'on désire. Seulement comment distinguer la désérialisation Client/Serveur de celle Serveur/Client ? Cela se fait facilement grâce à l'ajout d'une classe statique très simple :

```
public class WIA {

  private static boolean isServer = false;

  public static boolean getIsServer() {
    return isServer;
  }

  public static void setServer(boolean isServer) {
    WIA.isServer = isServer;
  }

}
```

Fig. 34 – *Classe WIA permettant de distinguer désérialisation Client/Serveur et Client/Serveur*

Ainsi chaque JVM est caractérisée par un attribut indiquant s'il s'agit d'un client ou d'un serveur. Il est alors possible d'écrire complètement readResolve.

```
protected Object readResolve() {

  SharedObject so  = Client.lookId(id);

  if (so == null && !WIA.getIsServer()) {
    Client.createStub(obj, id);
  }

  return so;
}
```

Fig. 35 – *Méthode ReadResolve du SharedObject*

On a alors ajouté une méthode lookId() à la classe Client permettant de récupérer le SharedObject associé à l'identifiant id passé en paramètre :

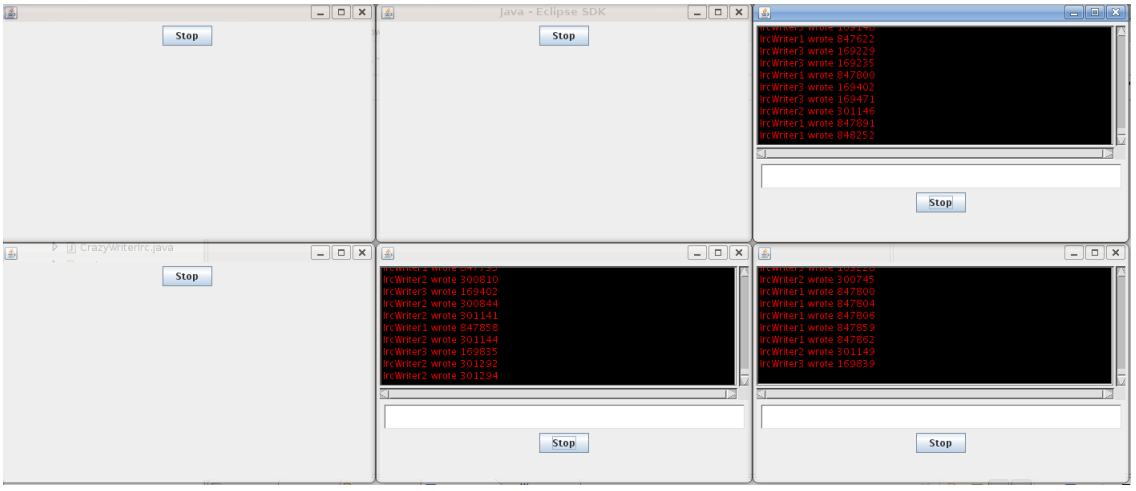
```
public static SharedObject lookId(int id) {
  return local_shared_objects.get(id);
}
```

Fig. 36 – *Méthode lookId de la couche Client*

5 Tests réalisés

5.1 Test avec Interface Graphique

Le premier test que nous avons réalisé consistait à lancer plusieurs lecteurs qui lisaient en permanence et plusieurs écrivains qui écrivaient en permanence. Aucun dead lock n'est remarqué



et les Sentence lues correspondent bien à celles écrites.

Cependant, l'affichage graphique consomme beaucoup de ressources...

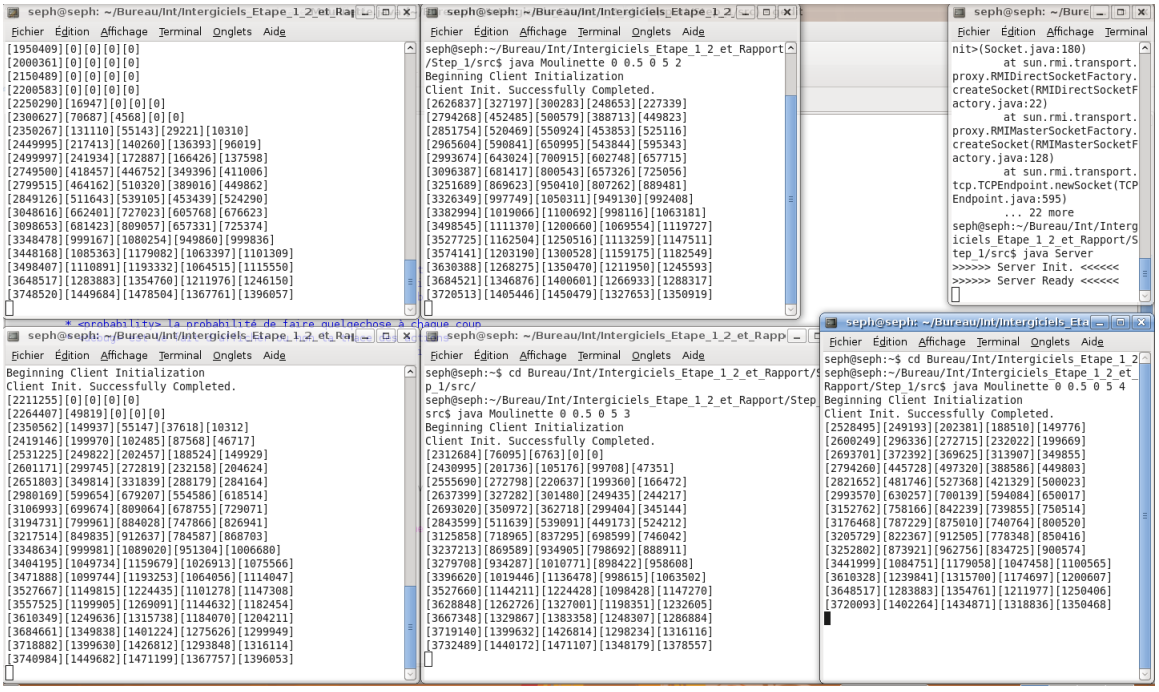
5.2 Test sans Interface Graphique

Nous avons alors réalisé un deuxième test affichant le nombre de lectures ou d'écritures réalisées toutes les 100000 lectures. Le nombre de lectures maximal sera de 1 million. Ce test permet alors de mettre en valeur la gestion de la synchronisation de notre système.
Nous suivons alors dans chaque terminal une ligne préformatée :

$$[nombre_1][nombre_2][nombre_3][nombre_4][nombre_5]$$

tels que $nombre_i$ correspond aux nombres de lectures et écritures réalisées par l'application i depuis son exécution.

On obtient alors le résultat suivant :



6 Conclusion

6.1 Améliorations possibles

Nous aurions pu simplifier le code, notamment dans le ServerObject où l'attribut *lock* fait "double emploi" avec le Vecteur de lecteurs et l'écrivain. En effet, pour savoir si le ServerObject est verrouillé en écriture, il suffirait de vérifier si l'attribut *ecrivain* est *null*.
D'autres simplifications seraient très certainement possibles mais par manque de temps, nous n'avons pas pu les réaliser.

6.2 Difficultés rencontrées

Tout d'abord, l'accumulation de projets en cette fin d'année ne nous a pas permis de nous pencher pleinement sur ce projet qui nécessitait un temps de réflexion assez important compte tenu des différentes notions à maîtriser : concurrence (méthodes synchronisées, wait, notify), RMI, Générateur de Stubs avec l'utilisation de java.lang.reflect que nous avions développé AVANT que l'on ne l'introduise en séance de projet encadré.

La synchronisation était une difficulté majeure dans l'élaboration du projet. Nous avons du alors nous pencher en détails sur ces problèmes de dead lock et trouver une solution adaptée.

De plus, les vacances ne nous ont pas permis de nous voir même si nous avons mis en place un SVN.

Cependant, même si les interfaces nous ont été fournies, le sujet restait peu clair concernant certains points (Générateur de "stubs", ces stubs ne faisant pas réellement allusion à ceux que nous avions étudié en cours).

6.3 Ce que ce projet nous a apporté

Tout d'abord, ce projet nous a permis d'approfondir différentes notions étudiées en cours : systèmes concurrents, RMI,etc...
De plus, ce projet nous a permis de travailler en équipe et d'utiliser différents outils (Eclipse, SVN, LaTeX).

Nous avons été "forcés" à nous organiser du fait de la relative complexité du travail attendu, mais également de la quantité de travail à fournir pour les autres matières (Spécifications Formelles, Analyse de Données, Traductions des Langages).