

Table des matières :

1	Introduction.....	2
2	Middleware	4
2.1	Fonctionnalités d'un Middleware	4
2.2	Architecture d'un middleware	5
3	Description des modules	7
3.1	Transport.....	7
3.1.1	Rôle	7
3.1.2	Architecture.....	8
3.1.3	Implémentation.....	9
3.1.4	Exercice 1 : UDP.....	11
3.2	Marshalling	12
3.2.1	Rôle	12
3.2.2	Architecture.....	12
3.2.3	Implémentation.....	12
3.2.4	Exercice 2 : Collections	19
3.3	Echange de messages	21
3.3.1	Rôle	21
3.3.2	Architecture.....	21
3.3.3	Implémentation.....	22
3.3.4	Extentions	29
3.4	RPC.....	31
3.4.1	Rôle	31
3.4.2	Architecture.....	31
3.4.3	Implémentation.....	32
3.4.4	Exercice 3:RPC.....	35
3.5	Discovery	36
3.5.1	Rôle	36
3.5.2	Architecture.....	36
3.5.3	Implémentation.....	37
3.5.4	Exercice 4: Discovery	39
4	Exercices supplémentaires	40
4.1	Exercice 0 : Application répartie	40
4.2	Exercice 5 : Publish/Subscribe.....	40
5	Conclusion	41

Annexe 0 : Application répartie

Annexe 1 : UDP

Annexe 2 : Collections

Annexe 3 : RPC

Annexe 4 : Discovery

Annexe 5 : Publish/Subscribe

1 Introduction

Aujourd'hui, le domaine des applications réparties ne cesse de prendre de l'ampleur dans le domaine de l'informatique. De plus en plus, les logiciels permettent à plusieurs utilisateurs travaillant sur des machines différentes de communiquer ou de partager des ressources. Cette émergence est accrue par l'augmentation du débit des lignes de communication qui diminue très fortement le temps de transfert des données d'une application à une autre.

Le bon fonctionnement d'une application répartie se base sur une couche intermédiaire qui permet d'offrir des méthodes de communication et de partage de ressources de manière transparente aux applications. Cette couche intermédiaire est appelée plus communément un middleware ou intergiciel en français. Le fonctionnement d'un middleware ainsi que les fonctionnalités qu'il offre est un des sujets du cours de Systèmes Répartis proposé par le LSR.

C'est afin d'offrir différents exercices pour le cours de Systèmes Répartis que ce projet a été proposé. Il contient deux buts principaux :

- ?? **Réaliser en Java un middleware simplifié constitué de différents modules substituables dont l'implémentation est invisible offrant chacun une fonctionnalité**
- ?? **Proposer un certain nombre d'exercices en rapport avec le middleware réalisé**

Les exercices proposés contiennent l'énoncé, le corrigé et un programme de test pour vérifier le fonctionnement de l'exercice. Les exercices peuvent être de deux types :

- ?? **Implémenter une application répartie utilisant une fonctionnalité du middleware**
- ?? **Implémenter une fonctionnalité du middleware**

La première partie du rapport décrit l'architecture générale d'un middleware, la seconde contient une description détaillée des différents modules implémentés dans ce projet ainsi qu'une description des exercices proposés pour chaque module. Deux exercices portant sur l'utilisation du Mini-Middleware sont ensuite décrits suivis d'une conclusion qui termine ce rapport. En annexe se trouvent les énoncés des exercices.

Les fichiers sources, ainsi que les fichiers correspondant aux exercices et leur corrigé ont été transmis électroniquement à l'assistant supervisant ce projet.

Pour la description des modules, les conventions d'écriture sont les suivantes :

Class : *public void maMethode()*

correspond à la définition d'une méthode du Mini-Middleware. Lorsque l'on fait référence à une méthode, une classe ou un package, ceux-ci seront en italique.

2 Middleware

Un Middleware, dans le cadre de l'informatique répartie est un logiciel de communication qui permet à plusieurs processus s'exécutant sur une ou plusieurs machines d'interagir à travers un réseau. On peut représenter le middleware comme une couche qui s'insère entre le système d'exploitation et les applications (Figure 1.1)

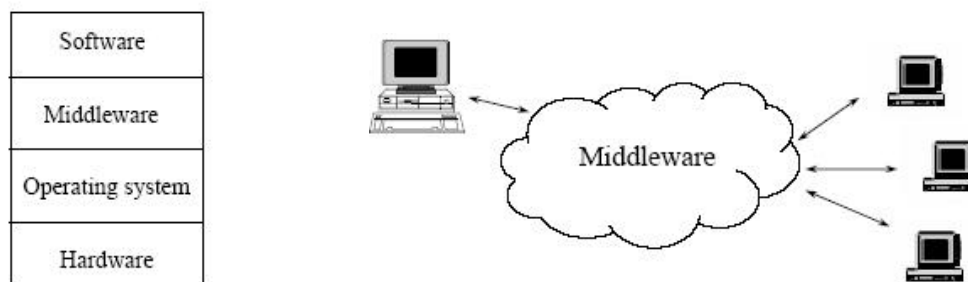


Figure 1.1

Le but d'un middleware est donc d'offrir des services de communication et de partage de ressources de manière transparente, c'est-à-dire que les applications vont utiliser les fonctionnalités de haut niveau offertes par le middleware sans se soucier de la manière dont les données sont traitées et transmises. Les langages orientés objets sont donc particulièrement adaptés à la réalisation d'un middleware grâce au mécanisme d'encapsulation qui les caractérisent.

2.1 Fonctionnalités d'un Middleware

Il existe de nombreux middleware sur le marché. Chacun d'eux est implémenté de manière différente et utilise des mécanismes de communication et de partage de ressources différents. Parmi tous ces middlewares les principales fonctionnalités offertes sont les suivantes :

?? **Echange de données**

Cette fonctionnalité permet la communication entre différentes applications. Elle peut être implémentée par l'échange de messages (Message oriented middleware) ou par une base de données commune à toutes les applications (Data oriented middleware). Cette communication peut s'effectuer en mode synchrone ou asynchrone.

?? **Formatage des données**

Cette fonctionnalité permet de transformer des données structurées en un format pouvant être transmis sur un réseau et de reconstituer les données structurées à

partir du format reçu sur le réseau. Cette fonctionnalité est très utile pour toutes les applications qui sont écrites dans un langage orienté objet et qui doivent transmettre un objet par le réseau.

?? Fonctions de sécurité

Cette fonctionnalité permet de transmettre des données chiffrées sur le réseau. Ces données ne pouvant être déchiffrées que par l'application à laquelle elles sont destinées.

?? Localisation des données

Cette fonctionnalité permet à une application de localiser les données dont elle a besoin, sans connaître à priori la machine qui héberge les données.

?? Appel de procédures à distance

Cette fonctionnalité permet l'appel de fonctions ou procédures sur des données qui se trouvent sur une autre machine. Pour les langages orientés objets, cette fonctionnalité permet d'invoquer une méthode sur un objet distant.

2.2 Architecture d'un middleware

Parmi les middleware du marché, on peut différencier trois types d'architectures pour la communication entre les différentes applications

?? Architecture client/serveur

Cette architecture distingue l'application client qui émet les requêtes de l'application serveur qui envoie des réponses aux requêtes du client. Le client communique directement avec le serveur (Figure 2.1)

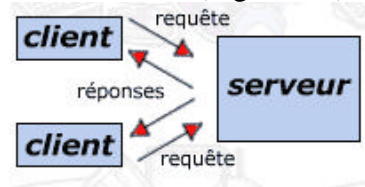


Figure 2.1

?? Architecture 3-tiers

Cette architecture contient une application centrale qui redirige toutes les requêtes des clients vers les différents serveurs (Figure 2.2)

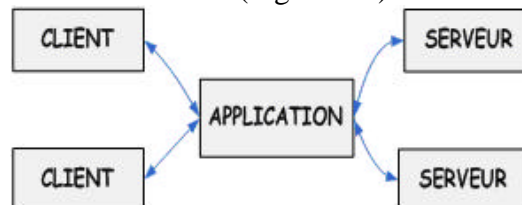


Figure 2.2

?? Architecture peer-to-peer

Dans cette architecture toutes les applications communiquent entre elles. On ne fait plus la différence entre client et serveur (Figure 2.3)

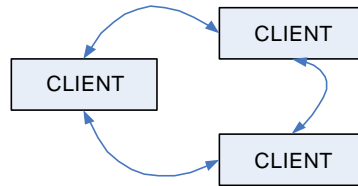


Figure 2.3

3 Description des modules

Le Mini-Middleware est un middleware basé sur l'architecture client/serveur décrite ci-dessus (Figure 2.1). Il implémente les fonctionnalités d'échange de données selon le principe de l'échange de messages (Message oriented middleware). La figure 3.1 décrit les différents modules qui composent le Mini-Middleware et la fonctionnalité qu'ils offrent.

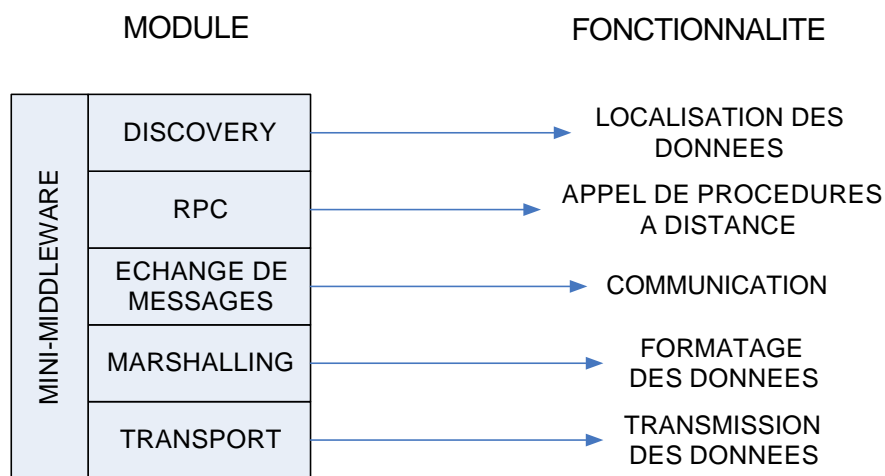


Figure 3.1

3.1 Transport

3.1.1 Rôle

Le rôle du module de transport est de permettre à une application (le client) d'envoyer des données sous forme d'un tableau de bytes et à une autre application (le serveur) de les réceptionner. Le client doit également transmettre toutes les informations nécessaires afin que le serveur puisse :

- ?? **reconstituer le message original**
- ?? **dispatcher correctement le message**
- ?? **envoyer une éventuelle réponse**

La figure 3.2 modélise le rôle du module de transport.

3.1.2 Architecture

Pour se reconnaître, le client et le serveur sont caractérisés par leur adresse. Le tableau de bytes est représenté par la classe *ByteMessage*. Afin que le serveur ait tout en main pour répondre à l'expéditeur, il doit recevoir un *ByteMessage* ainsi que l'adresse de l'expéditeur. C'est la classe *MessageFrom* qui permet de rassembler ces deux objets. Elle contient les méthodes permettant de retrouver le message et l'adresse de l'expéditeur.

La classe *ByteMessage* offre les méthodes suivantes :

ByteMessage : public byte[] toBytes()

Cette méthode permet de transformer les informations que contient un *ByteMessage* en un tableau de bytes.

ByteMessage : public ByteMessage(InputStream in)

Cette méthode permet de créer un *ByteMessage* à partir des informations qui se trouvent dans le Stream.

La classe *Address* offre la méthode suivante :

Address : public void send(ByteMessage bm)

pour l'envoi du message à l'adresse du serveur.

Pour recevoir le message, le serveur lance un *Thread* qui attend qu'un client envoie des données. C'est la classe abstraite *NetworkListener* qui s'en occupe. Elle offre les méthodes :

NetworkListener : public void run()

pour attendre les messages du réseau.

NetworkListener : public static Address getMyAddress(String transport)

pour retourner l'adresse à laquelle le Listener est atteignable pour une application distante

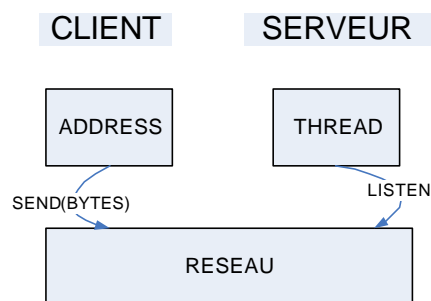


Figure 3.2

3.1.3 Implémentation

Il existe deux principaux protocoles pour la transmission de données à travers un réseaux : TCP (Transmission Control Protocol) et UDP (User Data Protocol). Dans ce projet l'utilisation des deux protocoles est implémentée et l'un est utilisé comme exercice. Dans la réalité, UDP est n'est pas toujours fiable, mais dans le cadre de ce projet, nous ferons l'hypothèse qu'il l'est.

La grande différence entre les deux protocoles est que TCP négocie l'ouverture d'une connexion avec le serveur avant l'envoi des données et utilise des *acknowledgements* après chaque envoi pour confirmer la réception des données, alors que UDP envoie les données sur le réseau sans se soucier de savoir si le serveur existe et s'il a reçu les données.

Selon l'implémentation du transport utilisé, la méthode d'envoi de la classe *Address* sera différente. Cette classe enregistre le moyen de transport utilisé ainsi que l'adresse de l'hôte. Cette adresse est représentée par une chaîne de caractères qui a la forme suivante : **host:port**.

La méthode d'envoi :

Address : public void send(ByteMessage bm)

appelle la classe permettant d'envoyer le message selon le moyen de transport désiré.

3.1.3.1 TCP

Lorsque l'on utilise le protocole TCP en Java, on travaille avec la notion de *socket* et de *serverSocket* (package *java.net*). Une *socket* représente une connexion entre le client et le serveur. La *serverSocket* permet au serveur d'attendre des connexions sur un certain port. Dès qu'une connexion arrive, une *socket* est créée. Pour pouvoir établir une connexion avec le serveur, le client doit créer une *socket* en lui passant en paramètre l'adresse et le port du serveur.

Ouvrir une connexion prend du temps. C'est pourquoi, lorsque plusieurs messages doivent être envoyés et reçus, il faut utiliser la même connexion, ou utiliser une connexion déjà existante. Si on remarque que cette connexion n'est plus utilisée depuis un certain temps, on peut la fermer.

C'est la classe *SocketManager* qui permet de gérer ces différents éléments. C'est un singleton, c'est-à-dire qu'elle n'est instanciée qu'une seule fois. De cette manière tous les messages qui sont envoyés et reçus passent par cette classe. Ceci lui permet de connaître toutes les connexions qui sont ouvertes et de les gérer. Les classes d'envoi et de réception de messages vont donc l'appeler pour chaque opération sur les *sockets*.

La classe *TCPListener* qui est une sous-classe de *NetworkListener* permet de récupérer les messages. Lors de sa création, elle va essayer de créer une *serverSocket* sur un des quatre ports suivants : 1234, 1235, 1236, 1237 (ceci permet d'exécuter trois fois cette application sur la même machine). Puis elle appelle la méthode *start()* qui lance le *Thread*. Dès que la *serverSocket* est ouverte correctement, une nouvelle adresse est créée correspondant à l'adresse à laquelle la *serverSocket* est atteignable.

Comme expliqué plus haut, c'est la classe *SocketManager* qui s'occupe d'envoyer et de recevoir les données sur les *sockets*. Elle utilise une table de hachage qui stocke d'un côté l'adresse (sous forme de chaîne de caractère) de l'hôte distant et de l'autre la *socket* qui les relie.

Lors de l'ouverture d'une *socket*, le client envoie sur celle-ci l'adresse (chaîne de caractère) à laquelle il est atteignable afin que le serveur puisse lui envoyer des données en retour même si la connexion est interrompue.

Cette classe offre les méthodes suivantes :

SocketManager : public synchronized void send(ByteMessage bm, TCPAddress a)

Lorsqu'un client veut envoyer un message, on vérifie dans la table de hachage si une *socket* existe déjà pour l'adresse de destination. Si c'est le cas, on envoie les données sur la *socket* correspondante. Sinon, on crée une nouvelle *socket*, l'enregistre dans la table de hachage et on y envoie les données.

SocketManager : public void listen(ServerSocket sSocket)

Pour accepter une nouvelle connexion, le serveur doit appeler la méthode bloquante *accept()* de la classe *ServerSocket*. Lorsque la connexion est établie, on enregistre la *socket* dans la table de hachage et on lance un nouveau *Thread* qui lit les données arrivant sur cette *socket*, afin que le premier *Thread* soit toujours disponible pour attendre de nouvelles connexions.

Pour permettre la fermeture d'une *socket* si celle-ci n'est plus utilisée depuis un certain temps, on utilise la classe *UseSocket* qui contient une *socket* et une variable de type *int* qui est incrémentée chaque fois qu'un message est reçu ou envoyé sur la *socket*. D'autre part, un *Thread* est lancé lors de l'instanciation du *SocketManager* qui, toutes les cent secondes, décrémente la variable de la classe *UseSocket* de chaque *socket*, ferme la *socket* et supprime l'entrée correspondante dans la table de hachage si cette variable vaut zéro.

3.1.4 Exercice 1 : UDP

Dans cet exercice, les étudiants doivent implémenter le transport des données selon le protocole UDP. Le but est qu'ils comprennent la manière dont deux applications peuvent communiquer à travers un réseau, et la notion de client/serveur.

Comme pour le protocole TCP, cette implémentation a besoin d'une classe permettant d'envoyer les données et une autre permettant d'écouter sur un port pour recevoir les données. Comme il n'y a pas de connexion entre les deux machines, l'envoi de messages consiste à créer un paquet contenant les données à envoyer et de l'envoyer à l'adresse spécifiée. La méthode *send()* de la classe *UDPSender* effectue cette opération. Elle est à compléter par les étudiants. La réception se passe de la même manière que pour TCP. A son initialisation, la classe *UDPListener* va écouter sur un port de la machine. Chaque fois que des données arrivent, un nouveau *Thread* de la classe *UDPReader* est créé pour s'occuper des nouvelles données tandis que le premier *Thread* continue à écouter le réseau. Les méthodes *run()* de ces deux classes sont également à compléter par les étudiants.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 1) et les fichiers permettant d'effectuer l'exercice compressés dans le fichier `Exercice1_UDP.zip`

Les classes de test vérifient uniquement que l'application cliente puisse envoyer un message sous forme de tableau de bytes au serveur et que celui puisse lui répondre.

3.2 Marshalling

3.2.1 Rôle

Nous avons vu que les seules données qui peuvent être transmises par le module de Transport sont des bytes. Le module du marshalling du Mini-Middleware permet à une application Java de transformer un objet en un tableau de bytes afin que l'objet puisse être transmis à une autre application par un réseau.

3.2.2 Architecture

Pour réaliser cette fonctionnalité, deux méthodes sont nécessaires. La première permet de transformer un tableau d'objets en un tableau de bytes, la seconde effectue l'opération inverse, c'est-à-dire de reconstituer un tableau d'objets à partir d'un tableau de bytes.

C'est la classe *Serializer* qui offre cette fonctionnalité. Les deux méthodes qui sont implémentées sont les suivantes :

```
public byte[] marshall(Object[] o)
```

permet de transformer un tableau d'objets en un tableau de bytes

```
public Object[] unmarshall(byte[] bytes)
```

permet de transformer un tableau de bytes en un tableau d'objets

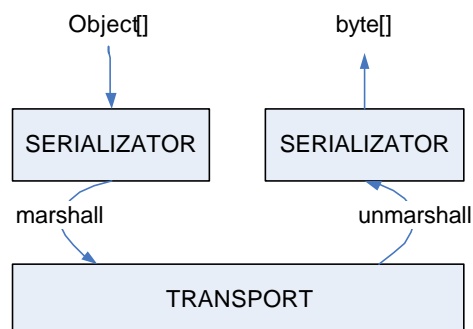


Figure 3.3

3.2.3 Implémentation

Pour transformer un objet en chaîne de bytes, il faut définir un protocole qui soit compris par le serveur afin que celui-ci puisse reconstituer l'objet à partir des bytes.

Une technique est implémentée dans le langage Java pour réaliser cette fonctionnalité. C'est la sérialisation. Une deuxième technique, basée sur le langage XML est implémentée dans ce projet.

C'est pourquoi une autre forme de sérialisation a été proposée dans ce projet. Celle-ci se base sur le langage XML facilement lisible pour un être humain. La classe *XMLDocument* permet de créer un document XML contenant les informations sur les objets et la classe *XMLParser* permet de reconstituer les objets à partir d'un document XML. Pour que le serveur puisse reconstituer l'objet, il faut qu'il reçoive les informations suivantes :

- ? nom de la classe de l'objet
- ? nom des champs de la classe
- ? valeur des champs de la classe
- ? si la classe est un tableau, valeur des éléments du tableau
- ? nom des éventuelles super-classes
- ? nom des champs des super-classes
- ? valeur des champs des super-classes

Les conventions suivantes ont été adoptées pour la constitution du document XML :

- ? l'élément Objects est la racine du document XML
- ? l'élément Object représente un objet
- ? l'élément Field représente un champ de la classe
- ? l'élément SuperClass représente la super-classe d'un objet
- ? l'élément ArrayValue représente un élément d'un tableau
- ? pour les éléments Object, SuperClass et Field, l'attribut type représente la classe de l'objet
- ? pour les éléments Object, Field et ArrayValue, l'attribut value représente la valeur de l'objet si elle peut être décrite sous forme de chaîne de caractères
- ? pour l'élément Field, l'attribut name représente le nom du champ de la classe

Pour qu'il puisse être décodé par le *XMLParser*, le fichier XML doit obéir à une certaine structure. Cette structure peut être décrite dans un fichier DTD qui a la forme :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Objects (Object*)>
<!ELEMENT Object (SuperClass?, (Field* | ArrayValue*))>
<!ELEMENT SuperClass (SuperClass?, Field*)>
<!ELEMENT Field (Object?)>
<!ELEMENT ArrayValue (Object?)>
<!ATTLIST Object type CDATA #REQUIRED
                value CDATA #IMPLIED>
<!ATTLIST SuperClass type CDATA #REQUIRED>
<!ATTLIST Field name CDATA #REQUIRED
                type CDATA #REQUIRED
                vaue CDATA #IMPLIED>

<!ATTLIST ArrayValue value CDATA #IMPLIED>
```

Voici un exemple du résultat de la sérialisation d'un objet de type *MonObjet* par le *XMLDocument* :

```
<?xml version="1.0" encoding="UTF-8"?>
<Objects>
  <Object type="ch.epfl.lsr.middleware.serialization.test.MonObjet">
    <SuperClass type="ch.epfl.lsr.middleware.serialization.test.MaSuperClasse">
      <SuperClass type="java.lang.Object"/>
      <Field name="objArray" type="[Ljava.lang.Object;">
        <Object type="[Ljava.lang.Object;">
          <ArrayValue>
            <Object type="java.lang.Class" value="java.lang.String"/>
          </ArrayValue>
          <ArrayValue>
            <Object type="java.lang.Character" value="@"/>
          </ArrayValue>
        </Object>
      </Field>
    </SuperClass>
    <Field name="intArray" type="[I">
      <Object type="[I">
        <ArrayValue value="5"/>
        <ArrayValue value="45"/>
        <ArrayValue value="79"/>
      </Object>
    </Field>
    <Field name="value" type="java.lang.String">
      <Object type="java.lang.String" value="Hello World"/>
    </Field>
    <Field name="obj" type="ch.epfl.lsr.middleware.serialization.test.Objet">
      <Object type="ch.epfl.lsr.middleware.serialization.test.Objet">
        <SuperClass type="java.lang.Object"/>
        <Field name="b" type="boolean" value="true"/>
      </Object>
    </Field>
  </Object>
</Objects>
```

Pour accéder ou recréer les informations décrites dans le document XML, on utilise les classes du package *java.lang* et *java.lang.reflect*.

Les méthodes suivantes permettent d'obtenir la classe et la super-classe de l'objet :

Object : `public Class getClass()`

Class : `public Class getSuperClass()`

Pour obtenir les champs de la classe, on utilise la méthode suivante :

Class : `public Field[] getDeclaredFields()`

Cette méthode déclenche une *IllegalAccessException* si le champ de la classe est privé ou protégé. Cela implique que cette méthode de sérialisation ne peut s'appliquer qu'à des objets dont les champs sont publics.

Il existe une méthode permettant de retourner l'objet contenu dans un champ pour un objet o, ou de fixer la valeur du champ pour l'objet o :

Field : public Object get(Object o)

Field : public void set(Object o, Object value)

Avant d'utiliser cette méthode, il faut s'assurer que le champ ne contienne pas une valeur de type primitif. Les types primitifs de Java sont les suivants : **int, byte, double, float, char, boolean, long** et **short**.

Dans ce cas, il faudrait appeler la méthode particulière au type primitif :

Field : int getInt(Object o), byte getByte(Object o) short getShort(Object o)

Field : void setInt(Object o, int value), void setShort(Object o, short value)

Pour obtenir le nom et le type du champ :

Field : public String getName(), public Class getType();

En ce qui concerne les tableaux, les méthodes suivantes permettent de vérifier si un objet est un tableau et le cas échéant d'obtenir la valeur de l'index i ou de fixer cette valeur :

Class : boolean isArray()

Array : public static Object get(Object o, int index)

Array : public static Object set(Object o, int index, Object value)

De la même manière que pour les champs, il se peut que les valeurs du tableau soient de type primitif. Après avoir repéré le type des données, on peut appeler la méthode correspondante :

Array : static int getInt(Object o, int index)...static short getShort(Object o, int index)

Array : static void setInt(Object o, int index, int value)

Finalement pour créer un objet dont on ne connaît que la classe, on utilise les méthodes suivantes :

Class : public Object newInstance()

si l'objet à créer n'est pas un tableau. Attention cette méthode déclenche une exception si la classe instanciée ne contient pas un constructeur public ne prenant aucun paramètre. C'est pourquoi, une autre contrainte à ce protocole de sérialisation est que les classes pouvant être sérialisées doivent posséder un constructeur public vide.

Array : public static Object newInstance(Class class, int length)

si l'objet à créer est un tableau. La classe à passer en argument correspond au type des objets du tableau.

Comme nous venons de le voir, les objets dont les champs ne sont pas publics et ne contenant pas un constructeur public vide ne peuvent être sérialisés. Ceci est le cas de la plupart des objets de Java. Il faut donc effectuer un traitement particulier pour chacune de ces classes. Dans ce projet, les types les plus importants ont été traités, mais on pourrait en ajouter bien d'autres. L'exercice portant sur ce module demande d'ailleurs aux étudiants de sérialiser des objets de type Collection.

Les types déjà traités dans ce projet sont les suivants:

- *java.lang.Integer, Byte, Double, Float, Character, Boolean, Long, Short*
- *java.lang.String, Class, java.lang.reflect.Array*

Toutes ces classes, à l'exception de *java.lang.reflect.Array* ont la caractéristique qu'on peut construire l'objet à partir d'une chaîne de caractères. De cette manière, le nom de la classe et un String suffisent à reconstruire l'objet.

Pour les objets de la classe *java.lang.reflect.Array*, il faut créer une structure supplémentaire permettant de reconstruire le tableau. C'est ce qui a été fait en ajoutant l'élément *ArrayValue*.

3.2.3.2.1 XMLDocument

Pour créer un document XML, la classe *XMLDocument* implémente les méthodes suivantes :

public byte[] marshall(Object[] o)

Cette méthode permet de sérialiser un tableau d'objets. Elle crée un document XML et appelle la méthode *marshallObject()* pour chacun des objets du tableau. Finalement le document XML est traduit en un tableau de bytes qui est retourné.

private Element marshallObject(Object o)

Cette méthode permet de créer un nœud qui représente un objet. Elle ajoute le type de l'objet au nœud puis vérifie si celui-ci ne fait pas partie d'un des types qui bénéficient d'un traitement particulier. Si c'est le cas, la valeur de l'objet est ajoutée au nœud et le nœud est retourné. Si le type est un tableau, c'est la méthode *marshallArray()* qui est appelée. Si l'objet n'est rien de tout ça, on doit accéder à ses champs et ceux de son éventuelle super-classe. Si l'objet possède une super-classe la méthode *marshallSuperClass()* est appelée. Ensuite pour chaque champ, la méthode *marshallField()* est appelée. Le nœud est finalement retourné.

private Element marshallSuperClass(Class superClass, Object o)

Cette méthode crée un nœud représentant la super-classe et ses champs. Elle se rappelle récursivement tant que la classe a une super-classe. La même méthode que plus haut est appelée pour sérialiser ses champs.

private Element marshallField(Field f, Object o)

Cette méthode crée un nœud représentant le champ d'un objet. On précise le nom du champ et son type. S'il s'agit d'un type primitif, on appelle la méthode correspondante (Field : *getInt()*, *getDouble()*...) pour obtenir sa valeur. Si le type du champ est un objet, la méthode Field : *get(Object o)* permet de le récupérer et la méthode *marshallObject()* retourne un nœud contenant les données de l'objet.

private void marshallArray(Object o, Element e)

Cette méthode permet de sérialiser le tableau o. Elle crée un élément ArrayValue pour chaque entrée du tableau. Cet élément deviendra un enfant du nœud e. Pour les tableaux, on distingue également les tableaux d'objets des tableaux contenant des éléments de type primitif.

3.2.3.2.2 XMLParser

Le rôle du parseur XML est de reconstituer le tableau d'objet à partir du tableau de bytes qu'a créé le XMLDocument.

Les méthodes implémentées sont les suivantes :

public Object[] unmarshall(byte[] b)

Cette méthode transforme le tableau de bytes b en un tableau d'objets. Le tableau de bytes est tout d'abord transformé en document XML. Pour chaque fils de la racine Objects du document, la méthode *unmarshallObject()* retourne un objet qui est inséré dans le tableau qui est finalement retourné.

private Object unmarshallObject(Element e)

Cette méthode reconstitue un objet à partir de l'élément e. L'attribut type de l'élément e permet de savoir si l'objet à créer fait partie des types de Java qui ont obtenu un traitement spécial. Si c'est le cas, la valeur est retrouvée, le constructeur appelé et l'objet retourné. Si le type correspond à un tableau c'est la méthode *unmarshallArray()* qui retourne l'objet. Dans les autres cas, la méthode *newInstance()* est appelée pour créer le nouvel objet. Si l'objet possède une super-classe et que celle-ci apparaît dans le document XML, alors, la méthode *unmarshallSuperClass()* est appelée pour fixer les champs de la super-classe. Ensuite, pour chacun des champs de la classe, on trouve le nœud lui correspondant dans le document XML et la méthode *unmarshallField()* permet de fixer les valeurs de ces champs. L'objet est ensuite retourné.

private void unmarshalSuperClass(Class super, Object o, Element e)

Cette méthode permet d'attribuer une valeur aux champs de la super-classe. Elle est appelée récursivement tant que la super-classe possède une super-classe. Enfin, elle attribue la valeur des champs de la super-classe grâce à la méthode `unmarshalField()`.

private void unmarshalField(Field f, Object o, Element e)

Cette méthode permet d'attribuer une valeur au champ `f`. Si le champ est d'un type primitif, le document XML devrait avoir stocké sa valeur et celle-ci est attribuée au champ par la méthode particulière du type (`Field : setInt()`...). Si le type n'est pas primitif, on attribue au champ la valeur retournée par la méthode `unmarshalObject()`.

private Object unmarshalArray(Element e)

Cette méthode retourne un objet de type tableau. Pour construire un tableau, on a besoin de connaître le type des éléments. En Java, le nom d'une classe correspondant à un tableau est représentée par le symbole suivant : [:

- ?? [B ↗ tableau de byte ↗ type des éléments = byte
- ?? [Z ↗ tableau de boolean ↗ type des éléments = boolean
- ?? [I ↗ tableau de int ...
- ?? [J ↗ tableau de long ...
- ?? [S ↗ tableau de short ...
- ?? [D ↗ tableau de double ...
- ?? [C ↗ tableau de char ...
- ?? [F ↗ tableau de float ...
- ?? [[B ↗ tableau de byte à deux dimensions ↗ type des éléments = [B
- ?? [L+nom_de_la_classe + ; ↗ tableau d'objets ...
- ?? ...

Cela signifie que si le type correspond à l'un des huit premiers cas, le tableau est créé avec le type des éléments correspondant. Si le type est un tableau à deux dimensions ou plus, il suffit de supprimer le premier [pour trouver le type qui servira à instancier le tableau. Enfin dans le dernier cas, on retrouve le type des éléments en supprimant le [L initial et le ; final.

Ceci étant fait, on parcourt tous les éléments `ArrayValue` et leur valeur est attribuée à un élément du tableau.

3.2.4 Exercice 2 : Collections

Dans cet exercice les étudiants vont ajouter au protocole défini ci-dessus la possibilité de sérialiser des objets de type collection. Le but de cet exercice est de leur faire comprendre le fonctionnement de la réflexion de Java, la création dynamique d'objets ainsi que le fonctionnement de la sérialisation.

Dans le corrigé de cet exercice, la sérialisation d'un type *Collection* est géré de la manière suivante :

On crée une nouvelle méthode permettant de sérialiser un objet de type *Collection* :

XMLDocument : private void marshalCollection(Object o, Element field)

Cette méthode parcourt tous les objets de la collection *o*. Pour chaque objet de la collection, un nœud *Object* est créé auquel la méthode *marshalObject()* ajoute la valeur de l'objet ou un nouveau nœud si cet objet ne peut pas être converti en valeur directement. Les *Collections* ont l'avantage sur les tableaux qu'ils ne peuvent contenir que des types *Object* et pas des types primitifs. Cela facilite grandement le traitement des éléments.

La méthode *marshalObject()*, après avoir testé si l'objet est d'un des types particuliers décrit plus haut, teste si l'objet est une instance de la classe abstraite *Collection*. Si c'est le cas, elle appelle la méthode *marshalCollection()* sur l'objet en question.

La désérialisation d'un type *Collection* s'effectue de la manière suivante :

On crée une nouvelle méthode permettant de désérialiser un objet de type *Collection* :

XMLParser : private void unmarshalCollection(Collection collec, Element e)

Cette méthode parcourt tous les enfants du nœud correspondant à la collection. Chaque nœud est un objet et la méthode *unmarshalObject()* est appelée pour reconstituer l'objet qui est ensuite inséré dans la collection.

Une des particularités des sous-classes de *Collection* est qu'ils ont un constructeur public ne prenant aucun paramètre. De cette manière la méthode *newInstance()* peut leur être invoquée sans provoquer d'exception. Ce n'est donc qu'après avoir instancié l'objet que la méthode *unmarshalObject()* vérifie s'il est de type *Collection*. Le cas échéant elle appelle la méthode décrite ci-dessus.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 2) , et les fichiers source compressés dans le fichier *Exercice2_Collection.zip*.

Pour vérifier le fonctionnement de la sérialisation d'un type *Collection*, le programme de test sérialise un *Vector* dans un tableau de bytes résultat, imprime ce résultat dans un fichier, désérialise le résultat et vérifie si le *Vector* obtenu contient les mêmes éléments que le *Vector* de départ.

3.3 Echange de messages

3.3.1 Rôle

L'échange de messages permet à deux applications de communiquer en s'envoyant différents types de messages de différentes façons. En effet, chaque application contient des données sous une forme qui lui est propre. Ce module permet d'envoyer ces différents formats de manière synchrone ou asynchrone pour le client et de les recevoir également de manière synchrone ou asynchrone chez le serveur

3.3.2 Architecture

Les différents types de messages sont représentés par des classes implémentant l'interface *Message*. Pour recevoir un message, le serveur utilise un *MessageDispatcher* correspondant au type de message qu'il désire recevoir. *MessageDispatcher* est une classe abstraite qui offre différentes méthodes de gestion de messages. Ses sous-classes correspondent aux types de messages et sont des singletons car s'il existait plusieurs instances d'une de ces classes, il y aurait un conflit d'intérêt à l'arrivée d'un message d'un certain type.

Pour traiter un message, on peut attacher un *MessageListener* au *MessageDispatcher*. Cette nouvelle interface offre une unique méthode qui doit être appelée lors de l'arrivée d'un message :

MessageListener : public void dispatch(MessageFrom mssgF)

Si un *MessageListener* a été attaché à la classe *MessageDispatcher*, cette méthode sera appelée à l'arrivée d'un message.

L'interface *Message* offre différentes méthodes pour l'envoi de messages :

Message : public void send(Address a)

Méthode permettant d'envoyer le message à l'adresse **a**.

Message : public void send(Address a, MessageListener mssgL)

Méthode permettant d'envoyer le message à l'adresse **a** et de fournir le *MessageListener* qui s'occupe de traiter la réponse si elle arrive.

Message : public MessageFrom blockSend(Address a)

Méthode bloquante permettant d'envoyer le message à l'adresse **a**, puis d'attendre la réponse au message.

Exemple d'échange de messages :

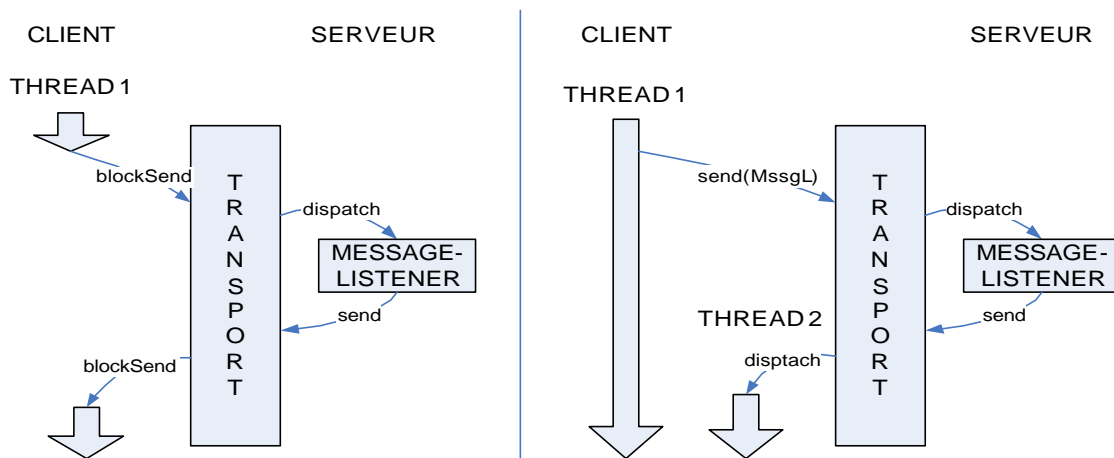


Figure 3.5

3.3.3 Implémentation

3.3.3.1 Messages

Le client crée un message d'un certain type et l'envoie au serveur. Afin de faciliter la reconnaissance des messages, chaque message est constitué d'un identificateur. Lorsque le serveur reçoit un message auquel il répond, il utilise le même identificateur que le message d'origine afin que chez le client, la réponse soit redirigée au bon endroit. Chaque message est également constitué d'une liste de propriétés. Cette liste permet de spécifier certains paramètres pour la transmission du message tels que le protocole utilisé, le type de transport ou le type de sérialisation des données. L'identificateur et la liste de propriétés sont envoyés avec le message.

Dans ce projet, trois types de messages ont été implémentés. Ces trois messages héritent des méthodes de l'interface Message vu qu'ils l'implémentent (Figure 3.6)

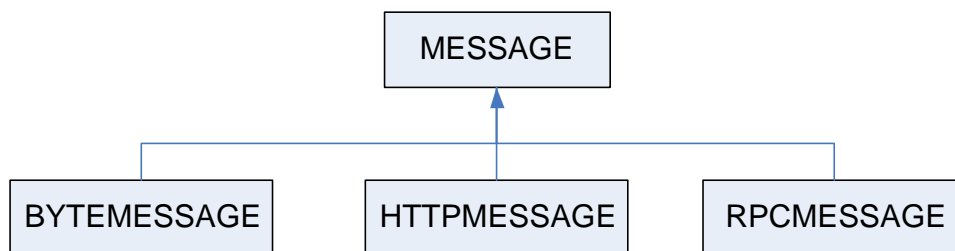


Figure 3.6

?? **ByteMessage**

Ce type de message correspond au message de plus bas niveau. En plus de son identificateur et la liste de propriétés, il contient un tableau de bytes correspondant au message à transmettre.

?? **HTTPMessage**

Ce message correspond à un message implémenté selon le protocole HTTP. Il contient les en-têtes HTTP suivantes : la méthode (GET, POST, OK,...), l'uri et le type du contenu. Le contenu est représenté par un tableau de bytes.

?? **RPCMessage**

Ce message contient les informations utiles pour effectuer des invocations à distance. Les paramètres de ce message seront détaillés lorsque l'on parlera du RPC.

Comme vu plus haut, la couche de transport n'est capable que d'envoyer un *ByteMessage*. Pourtant pour les couches supérieures, il est impossible de créer directement un tableau de bytes à partir des informations qu'elles possèdent. C'est la raison pour laquelle une hiérarchie de messages a été créée. Une des spécifications du projet est de permettre de modifier facilement cette hiérarchie de messages, afin d'insérer de nouvelles fonctionnalités. La classe *MessageTransformer* permet de transformer un type de message dans un autre type selon les propriétés transmises dans la liste de propriétés.

Elle offre les deux méthodes suivantes :

MessageTransformer : public void send(Message m)

Cette méthode permet d'envoyer un message de n'importe quel type. Elle va lire dans la liste des propriétés la hiérarchie à suivre pour transformer le message en *ByteMessage* puis envoyer le *ByteMessage*.

MessageTransformer : public void dispatch(ByteMessage m)

Cette méthode permet de transformer le *ByteMessage* dans son type final selon la hiérarchie fournie dans la liste de propriétés. Lorsque le message se trouve dans sa forme finale, il sera envoyé au *MessageDispatcher* correspondant à ce type de message.

Pour réussir ces différentes transformations, les messages doivent pouvoir être construits à partir d'autres messages. La hiérarchie proposée dans ce projet est la suivante :

ByteMessage ? HTTPMessage ? RPCMessage

De cette manière, le *ByteMessage* peut être construit à partir d'un *HTTPMessage*, le *HTTPMessage* à partir d'un *ByteMessage* et d'un *RPCMessage* et le *RPCMessage* à partir d'un *HTTPMessage*.

Dans la liste de propriétés le message HTTP correspond à la propriété :

PROTOCOLE=HTTP

et le message RPC à la propriété :

APPLICATION=RPC

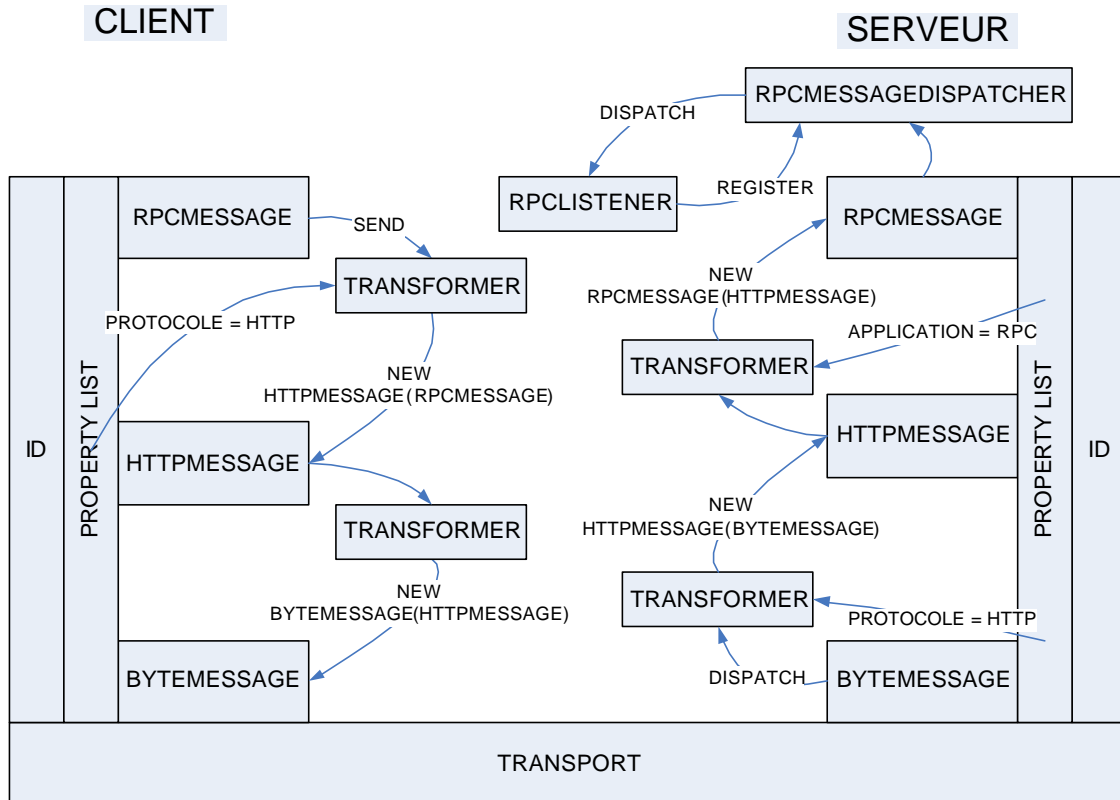


Figure 3.7

La figure 3.6 explique la transmission d'un *RPCMessage*. Lorsque la méthode `send` est invoquée sur cet objet, la méthode `send()` de la classe *MessageTransformer* est appelée. Cette méthode lit tout d'abord la propriété `protocole` qui est ici égale à `HTTP`. Elle va donc chercher le constructeur de la classe *HTTPMessage* qui prend en paramètre un *RPCMessage*. Le *HTTPMessage* étant un message de type `protocole`, ce message est directement traduit en un *ByteMessage* sans passer par la liste de propriétés. De l'autre côté, lorsque le module de transport a créé le *ByteMessage*, la méthode `dispatch()` de la classe *MessageDispatcher* est appelée. Elle vérifie le protocole utilisé puis appelle le constructeur de la classe *HTTPMessage*. Puis elle vérifie l'application à laquelle appartient le message et appelle son constructeur qui prend en paramètre le *HTTPMessage*. Lorsque le message est reconstruit dans son format original, il est envoyé au *MessageDispatcher* correspondant qui le dispatche selon l'identificateur du message.

3.3.3.2 Le protocole HTTP

Dans ce projet, un protocole a été utilisé pour différencier les différentes requêtes et les différentes réponses. Ce protocole s'inspire très fortement du protocole HTTP. Pour les requêtes, deux méthodes ont été implémentées : GET et POST. La première sert à obtenir des informations sur l'uri spécifié, la seconde permet d'effectuer des opérations sur l'uri. Pour les réponses, trois méthodes ont été implémentées : OK, NOT FOUND, BAD REQUEST. La première informe que la requête s'est effectuée correctement et envoie les informations demandées par la requête, la seconde informe que l'uri spécifié n'a pas été trouvé, et la troisième informe que la requête a été mal formulée.

Tous les messages contiennent le nom du protocole, le type des données et la longueur du contenu parmi les en-têtes. Les requêtes contiennent en plus l'uri. Pour créer un *ByteMessage* à partir du *HTTPMessage*, on répartit les informations ajoute une en-tête de la manière suivante :

Les en-têtes sont réparties sur les trois premières lignes. La première ligne contient la méthode, le protocole, et pour les requêtes on ajoute l'uri. La seconde ligne contient le type du contenu et la troisième ligne contient la longueur des données. Ensuite on laisse une ligne vide avant d'ajouter le contenu.

Pour créer un *HTTPMessage* à partir d'un *ByteMessage*, il suffit d'effectuer l'opération inverse.

3.3.3.3 MessageDispatcher

Pour gérer l'arrivée de nouveaux messages, la classe *MessageDispatcher* contient un vecteur et deux tables de hachage. Le vecteur stocke les identificateurs des messages pour lesquels un *Thread* ou un *MessageListener* enregistré est entrain d'attendre. La première table de hachage stocke d'une part les identificateurs des messages et de l'autre le *MessageListener* qui s'occupe du message contenant cet identificateur. La classe doit toujours vérifier que le même identificateur n'est pas attendu par deux objets à la fois. La seconde table de hachage stocke les messages reçus : d'un côté leur identificateur et de l'autre le message. On peut également ajouter un *MessageListener* par défaut qui sera appelé si le message n'est attendu par personne. Les différents *MessageDispatcher* sont décrits sur la figure 3.8.

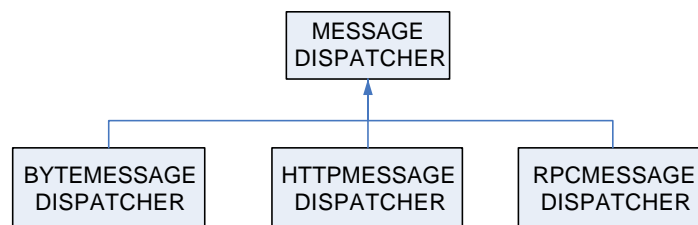


Figure 3.8

Les différentes méthodes de gestion de message offertes par la classe *MessageDispatcher* sont les suivantes :

MessageDispatcher : public MessageFrom getMessage(String id)

Cette méthode permet d'attendre un message particulier caractérisé par son identificateur. Cette méthode est bloquante (appel de la méthode *wait()*) jusqu'à ce que le message en question arrive.

MessageDispatcher : public MessageFrom getMessageImmediate(String id)

Cette méthode est identique à la méthode ci-dessus à la différence qu'elle n'est pas bloquante et que si le message n'est pas dans la liste, elle retourne une valeur nulle.

MessageDispatcher : public void register(MessageListener mssgL, String id)

Cette méthode permet d'attacher un *MessageListener* au *MessageDispatcher* pour un message particulier. Lorsque le message arrive, la méthode *dispatch()* du *MessageListener* est appelée.

MessageDispatcher : public void unregister(MessageListener mssgL, String id)

Méthode permettant de retirer le *MessageListener* de la liste pour un message particulier.

MessageDispatcher : public void register(MessageListener mssgL)

Méthode permettant de spécifier le *MessageListener* par défaut. Il ne peut y en avoir qu'un seul. Si cette méthode est appelée une deuxième fois, cette méthode lance une exception.

MessageDispatcher : public void unregister(MessageListener mssgL)

Cette méthode permet de retirer le *MessageListener* par défaut.

MessageDispatcher : public void addMessage(MessageFrom mssgF)

Cette méthode est appelée à l'arrivée d'un nouveau message. Elle vérifie si l'identificateur du message se trouve dans la liste des messages attendus par un *Thread*. Si c'est le cas, elle met le message dans la liste des messages arrivés et notifie tous les *Thread* de l'arrivée de ce nouveau message. S'il existe un *MessageListener* particulier pour ce message ou alors un *MessageListener* par défaut, la méthode crée un nouveau *Thread* qui appellera la méthode *dispatch()* du *MessageListener* en question. Sinon, le message arrivé est mis dans la liste.

Deux méthodes ont été ajoutées à cette classe :

MessageDispatcher : public int register(String id)

MessageDispatcher : public MessageFrom getMessage(int key)

Ces deux méthodes ont été ajoutées pour implémenter la méthode *blockSend()* de l'interface *Message*. En effet la méthode *blockSend()* envoie le message puis s'enregistre auprès du *MessageDispatcher* pour recevoir la réponse à ce message. Dans les tests effectués, il se produisait de temps en temps que la réponse au message arrivait avant que le *Thread* ait eu le temps de s'enregistrer auprès du *MessageDispatcher*, ce qui avait comme conséquence de générer un message inconnu pour le *MessageListener* qui récupérait les données et de bloquer le *Thread* qui s'enregistrait trop tard. Ce problème a été résolu en divisant la méthode *getMessage()* en deux. La première méthode permet d'ajouter l'identificateur dans la liste des messages attendus, et de générer une clef aléatoire qui est retournée. Cette clef doit être passée en paramètre de la méthode *getMessage()* afin de retrouver l'identificateur du message correspondant. Le message peut ensuite être envoyé. Si la réponse arrive avant que la méthode *getMessage()* soit invoquée le fait que l'identificateur ait été mis dans la liste des messages attendus évite que le message soit dispatché par le *MessageListener*, et lorsque la méthode *getMessage(int key)* est appelée elle retournera directement la réponse.

Toutes ces méthodes sont synchronisées car elles peuvent être accédées par différents *Thread* et utilisent des ressources communes (vecteurs, tables de hachage).

Pour éviter que les méthodes bloquantes *getMessage(String id)* et *getMessage(int key)* bloquent le client éternellement si le serveur ne répond pas, un timer de 50 secondes est lancé avant l'appel à la méthode *wait()*. Lorsque le timer arrive à 50 secondes, il interrompt le *Thread* bloqué ce qui a pour effet de lancer une exception. Si le *Thread* est réveillé avant que le timer arrive à 50 secondes, ce timer est annulé. La figure 3.9 montre un exemple de fonctionnement du *MessageDispatcher*.

Exemple du fonctionnement du MessageDispatcher :

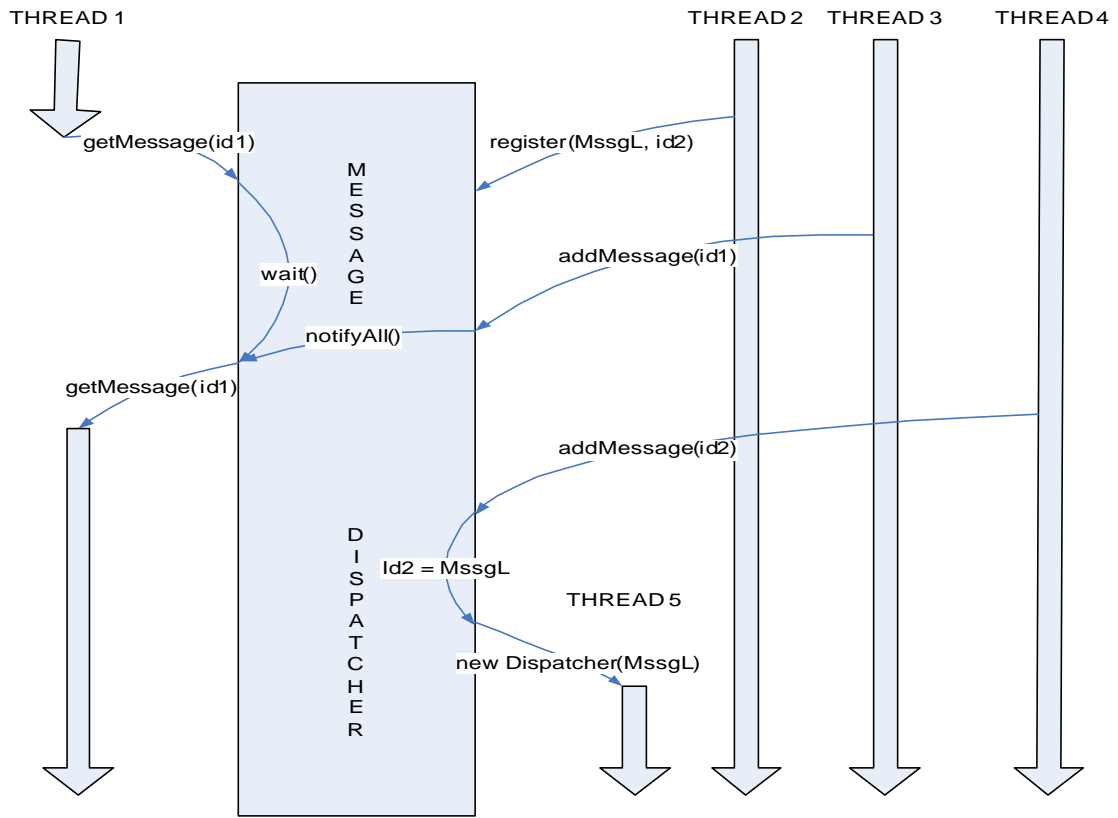


Figure 3.9

3.3.4 Extentions

Comme expliqué plus haut, la hiérarchie des messages n'est pas fixe. Cela signifie que l'on peut facilement créer un nouveau type de message ou leur ajouter de nouvelles fonctionnalités.

Un exemple serait de chiffrer les données du HTTPMessage :

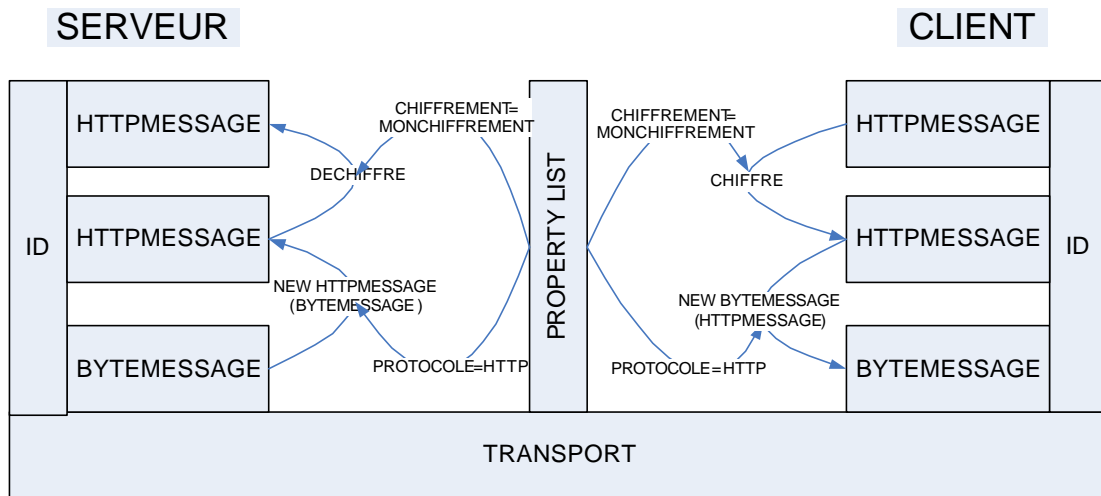


Figure 3.10

Ceci peut être fait de la manière suivante :

On crée les méthodes de chiffrement et de déchiffrement. Dans la liste des propriétés du message à envoyer, on ajoute par exemple la propriété :

CHIFFREMENT=MONCHIFFREMENT

Il suffit ensuite de modifier la classe MessageTransformer pour qu'elle vérifie la valeur de la propriété chiffrement après avoir créé le message HTTP. Si cette valeur n'est pas nulle, elle va appeler la méthode de chiffrement/déchiffrement et reconstruire un nouveau message HTTP dont le contenu est chiffré/déchiffré.

Un autre exemple serait d'inventer un nouveau protocole remplaçant HTTP :

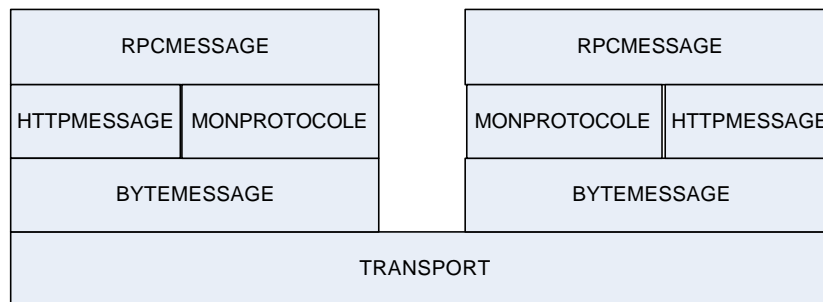


Figure 3.11

Ceci peut être réalisé de la manière suivante :

On crée la nouvelle classe représentant le message (MonProtocole) qui peut être construit à partir d'un RPCMessage ou d'un ByteMessage. On ajoute également la possibilité de construire un RPCMessage et un ByteMessage à partir de ce nouveau protocole. Dans la liste des propriétés du message à envoyer, on remplace la propriété

PROTOCOLE=HTTP par **PROTOCOLE=MONPROTOCOLE**

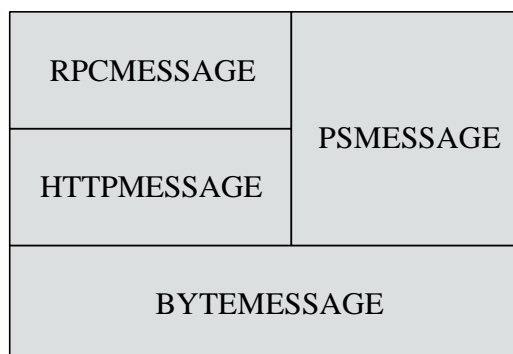
Il suffit d'ajouter la possibilité d'utiliser ce nouveau protocole dans la classe MessageTransformer pour que ce nouveau protocole puisse être utilisé.

Un autre exemple serait d'ajouter à cette hiérarchie les messages de l'exercice Publish/Subscribe qui se trouve en annexe (Annexe 6) et qu'ils correspondent à la propriété

APPLICATION=PUBLISH_SUBSCRIBE

Ainsi il faudrait que le PSMesssage implémente l'interface Message et créer une nouvelle classe PSMesssageDispatcher pour dispatcher les PSMesssage. Dans la classe MessageTransformer, on ajouterait simplement la transformation du PSMesssage en ByteMessage et inversement si la propriété APPLICATION vaut PUBLISH_SUBSCRIBE. La nouvelle hiérarchie est décrite sur la figure 3.17.

Figure 3.17



3.4 RPC

3.4.1 Rôle

Le module RPC (Remote Procedure Call) permet à l'application cliente d'invoquer des méthodes sur un objet distant, et au serveur de partager un objet qu'il a créé afin que d'autres applications puissent y accéder.

3.4.2 Architecture

Avant d'invoquer une méthode sur un objet distant, le client doit obtenir une référence sur cet objet. Le client doit donc connaître le type de l'objet recherché pour pouvoir ensuite invoquer des méthodes sur cet objet. Il faut également que le serveur et le client se mettent d'accord sur le nom de l'objet partagé.

La classe responsable de ces fonctionnalités est la classe *ObjectNaming*. Cette classe est un singleton afin que tous les objets partagés se trouvent au même endroit.

Cette classe offre les méthodes suivantes :

ObjectNaming : public RemoteInterface lookup(String objectName, String serverName)
pour la recherche de l'objet chez le serveur.

ObjectNaming : public bind(String objectName, Object o)
pour permettre au serveur de partager l'objet o. L'objet et son nom sont conservés dans une table de hachage.

Lorsque la référence sur l'objet distant est établie, le client possède une interface représentant l'objet. Un objet partagé doit donc posséder une interface. Cette interface doit elle-même être sous-classe de l'interface *RemoteInterface*, vu que c'est le type retourné par la méthode *lookup()*. La figure 3.13 décrit cette architecture.

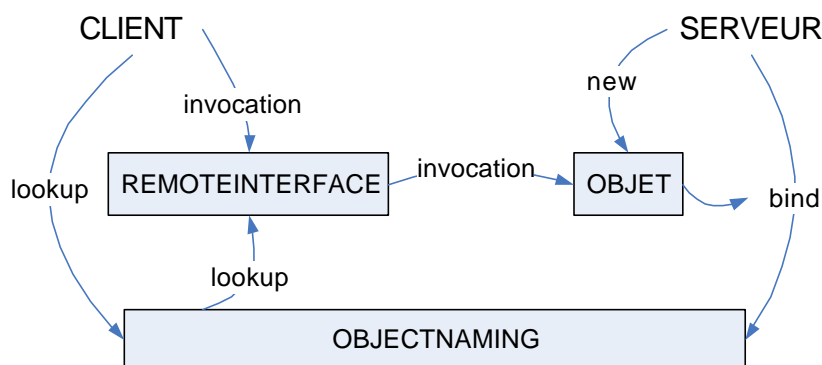


Figure 3.12

3.4.3 Implémentation

Lorsque le client appelle une méthode de l'objet distant, il faut que l'appel de la méthode soit modifié afin que l'on puisse transmettre l'appel au serveur. Ceci s'effectue grâce à la classe `Proxy` et l'interface `InvocationHandler` du package `java.lang.reflect` qui offrent les méthodes suivantes :

Proxy : `static Object newInstance (ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

InvocationHandler : `public Object invoke (Object proxy, Method method, Object[] args)`

La première méthode crée dynamiquement un objet de la classe loader qui implémente les interfaces passées en paramètre. Lorsqu'une méthode est invoquée sur le Proxy, la méthode `invoke` du handler est appelée.

Il faut donc créer une classe qui implémente l'interface `InvocationHandler` et redéfinir la méthode `invoke`. Dans ce projet, c'est la classe `RPCInvocationHandler` qui a cette tâche. Cette classe représente l'objet distant. C'est pourquoi le constructeur prend en paramètre le nom de l'objet distant, le nom de l'hôte possédant l'objet et la liste des propriétés concernant l'envoi. En plus de la méthode `invoke`, cette classe implémente la méthode :

RPCInvocationHandler : `public Class getRemoteClass()`

Cette méthode recherche chez le serveur l'interface de l'objet sur lequel on désire appeler des méthodes afin de pouvoir construire le Proxy (L'interface retournée par cette méthode sera passée en paramètre de la méthode `newInstance`).

La méthode

ObjectNaming : `public RemoteInterface lookup(String objectName, String serverName, Properties properties)`

crée un handler pour l'objet distant, cherche l'interface de l'objet chez le serveur puis crée le proxy en utilisant le loader de l'interface, l'interface et le handler lui-même comme paramètre de la méthode `newInstance`.

Pour que ces deux méthodes puissent envoyer des informations au serveur, elles ont besoin de créer des messages qu'elles envoient à celui-ci et attendre un autre message en retour. Pour transporter les informations de ces méthodes, on utilise les `RPCMessage`. Du côté du serveur, un `MessageListener` (`RPCListener`) attend les messages du client. Ce Listener est créé lorsque la classe `ObjectNaming` est instanciée et contient une référence sur la table de hachage contenant les objets partagés. Le `RPCListener` est ensuite enregistré chez le `RPCMessageDispatcher`. De cette manière tous les messages de type RPC lui seront envoyés.

Schéma du fonctionnement du RPC :

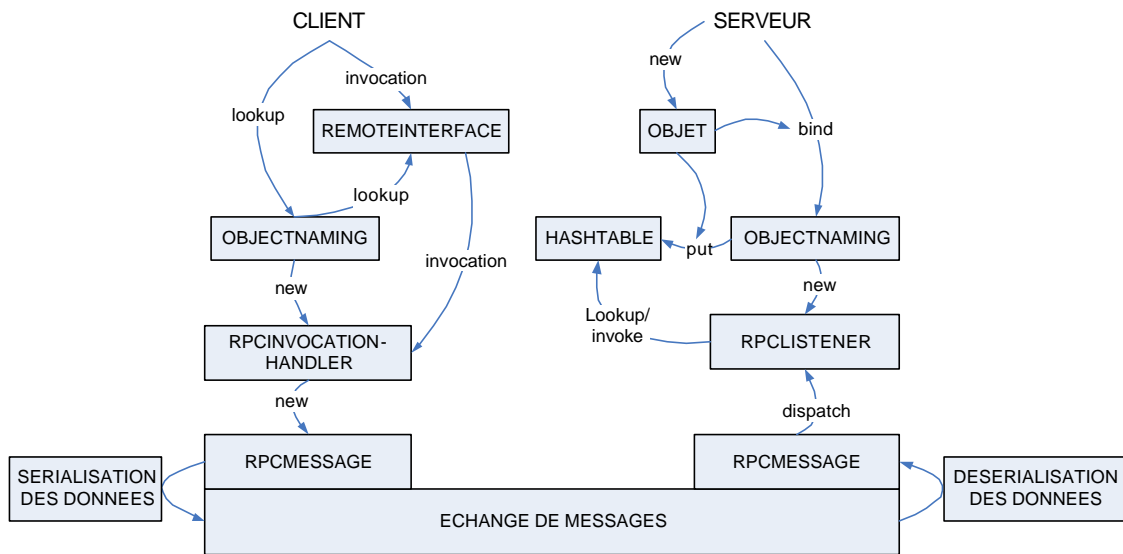


Figure 3.13

3.4.3.1 RPCMessage

Un *RPCMessage* est un message qui transporte les informations des méthodes du *RPCInvocationHandler*. Ce message doit donc contenir les informations suivantes :

- ?? le type de *RPCMessage* \approx *RPCtype*
- ?? le nom de l'objet distant \approx *target*
- ?? le nom de la méthode invoquée (si la méthode est *invoke*) \approx *method*
- ?? les propriétés pour l'envoi du message \approx *properties*
- ?? l'identificateur du message \approx *id*
- ?? un tableau d'objets correspondant aux arguments d'une méthode ou à l'objet retourné par une méthode \approx *obj*

Le type du message peut être :

- ?? **GET_REF** lorsque l'on désire connaître la classe de l'objet distant
- ?? **INVOKE** lorsque l'on veut appeler une méthode sur l'objet
- ?? **OK** si l'objet a été trouvé ou que l'invocation a réussi.
- ?? **NOT FOUND** si l'objet n'existe pas chez le serveur.
- ?? **BAD REQUEST** si le serveur n'a pas compris le message

Comme vu dans la partie d'échange de messages, un *RPCMessage* peut être traduit en un *HTTPMessage* et inversement. Cela se déroule de la manière suivante :

- ?? le RPCtype correspond à la méthode :
 - ?? GET_REF \approx GET
 - ?? INVOKE \approx POST
 - ?? OK , NOT FOUND, BAD REQUEST \approx idem
- ?? la méthode concaténée au target par le symbole @ deviennent l'uri :
 - ?? method@target \approx uri
- ?? la valeur de la propriété SERIALIZATOR devient le contentType
- ?? le tableau d'objet est transformé en tableau de bytes grâce à la méthode *marshall()* du *Serializator* correspondant.

3.4.3.2 RPCInvocationHandler

Comme expliqué plus haut, cette classe implémente deux méthodes :

RPCInvocationHandler : public Class getRemoteClass()

Pour rechercher l'interface de l'objet, la méthode *getRemoteClass* crée un *RPCMessage* de type GET_REF et le target correspond au nom de l'objet partagé. Ce message est ensuite envoyé par la méthode bloquante *blockSend* pour attendre la réponse du serveur avant de continuer. Si la réponse n'est pas de type OK, une exception est lancée. Sinon, on récupère l'objet qui est retourné et on vérifie qu'il s'agit bien d'une classe. Cet objet est ensuite retourné.

RPCInvocationHandler : public Object invoke (Object proxy, Method method, Object[] args)

Pour appeler une méthode à distance, la méthode *invoke* crée un *RPCMessage* de type INVOKE. Le target correspond au nom de l'objet sur lequel on veut appeler la méthode, le nom de la méthode est affecté au champ *method* et les arguments sont affectés au tableau d'objet *obj*. Le message est envoyé et la réponse attendue. On vérifie également que la réponse est de type OK et qu'un seul objet fait partie du tableau *obj* du *RPCMessage*. Le cas échéant cet objet est retourné.

3.4.3.3 RPCListener

Chez le serveur, le *RPCListener* attend des requêtes de la part du client. Cela signifie qu'il s'attend à recevoir des *RPCMessage* de type GET_REF ou INVOKE.

Si le message est de type GET_REF, on vérifie dans la table de hachage si la valeur du champ *target* s'y trouve. Si elle ne s'y trouve pas, un *RPCMessage* de type NOT FOUND est renvoyé à l'expéditeur. S'il s'y trouve, on y cherche l'objet correspondant. Le client attend du serveur qu'il lui envoie l'interface de l'objet afin qu'il puisse instancier le proxy. Mais il se peut que l'objet implémente plusieurs interfaces. C'est pourquoi, on retourne l'interface qui est une sous-classe de l'interface *RemoteInterface*. Un *RPCMessage* de type OK est créé et l'interface insérée dans le tableau d'objets *obj*. Le message est envoyé au client.

Si le message est de type INVOKE, on vérifie également dans la table de hachage si l'objet s'y trouve bien. Avant d'être appelée, la méthode doit être créée. Pour créer une méthode, on a besoin de sa classe, de son nom et d'un tableau de classes représentant les classes des arguments de la méthode, afin d'appeler la méthode suivante :

Class : public Method getMethod(String name, Class[] classes)

L'objet de la table de hachage nous permet de retrouver la classe de la méthode, le champ `method` du *RPCMessage* constitue le nom de la méthode et on crée un tableau de classes constitué de la classe de chaque objet du tableau `obj`. On peut ensuite faire l'invocation:

Method : public Object invoke(Object o, Object[] args)

L'objet `o` correspond à l'objet de la table de hachage et les `args` au tableau `obj` du *RPCMessage*. Le résultat est ensuite inséré dans un tableau d'objet et un *RPCMessage* de type OK est créé qui contient ce nouveau tableau d'objet. Finalement ce message est renvoyé au client.

3.4.4 Exercice 3:RPC

Dans cet exercice, les étudiants vont implémenter les trois méthodes suivantes qui ont été détaillées plus haut:

RPCInvocationHandler : public Class getRemoteClass()

RPCInvocationHandler : public Object invoke (Object proxy, Method method, Object[] args)

RPCListener: public void dispatch(MessageFrom mssgF)

A travers cet exercice, les étudiants vont découvrir le principe de la création d'objets "virtuels" sur lesquels on peut redéfinir l'invocation de méthodes. Ils utiliseront également le principe d'échange de messages du Mini-Middleware.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 3), et les fichiers sources compressés dans le fichier `Exercice3_RPC.zip`.

Le programme de test pour cet exercice correspond à l'exercice 0 que les étudiants auront implémentés pour découvrir le Mini-Middleware. Son principe est décrit dans l'annexe 0.

3.5 Discovery

3.5.1 Rôle

Le module Discovery doit permettre au client d'obtenir une référence sur un objet distant sans connaître l'hôte qui héberge cet objet à priori. Il doit également permettre au serveur de partager un objet qui peut être accédé par n'importe quel client.

3.5.2 Architecture

Pour réaliser cette fonctionnalité, on utilise un serveur de nom. Un serveur de nom permet de stocker le nom des objets partagés et les hôtes qui les hébergent. C'est la classe Discovery qui s'occupe de cette fonctionnalité. Elle offre les méthodes suivantes :

Discovery : public RemoteInterface lookup(String objectName)

pour obtenir une référence sur l'objet distant du nom objectName.

Discovery : public boolean bind(String objectName, Object o)

pour partager l'objet o. La méthode retourne faux si le nom est déjà utilisé et que l'objet n'a pas pu être lié.

Discovery : public boolean searchServer(String serverName)

pour rechercher le serveur à l'adresse serverName. C'est le seul hôte qui doit être connu de tous.

Discovery public static void main(String args[])

pour lancer le serveur de nom.

La figure 3.14 montre le schéma de fonctionnement du discovery.

Schéma :

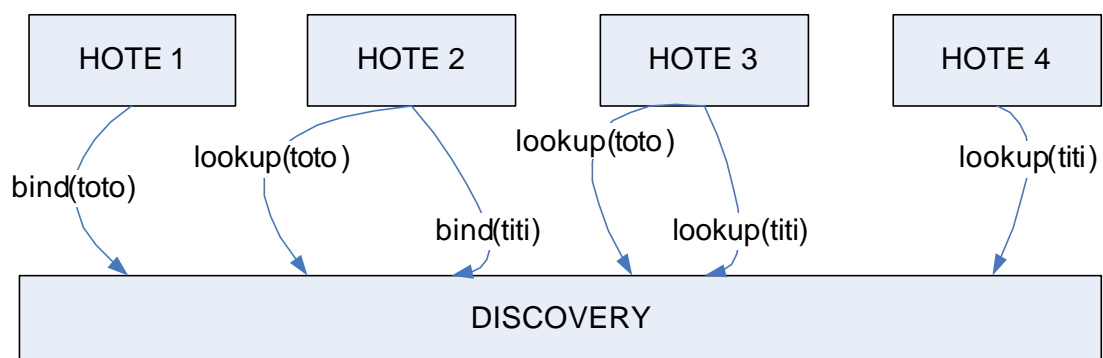


Figure 3.14

3.5.3 Implémentation

Le Discovery utilise l'architecture du RPC. On utilise un objet partagé NameService qui stocke le nom des objets partagés et l'adresse de l'hôte qui les héberge dans une table de hachage. Cet objet implémente l'interface NameServiceInterf. Les méthodes offertes par cette interface sont les suivantes :

NameServiceInterf : public Address lookup(String name)

Cette méthode permet d'obtenir l'adresse de l'hôte hébergeant l'objet name. Elle retourne null si le nom n'apparaît pas dans la table de hachage.

NameServiceInterf : public boolean addSharedObject(Address a, String objectName)

Cette méthode permet d'ajouter l'information que l'objet objectName se trouve à l'adresse a. Elle retourne faux si ce nom existe déjà dans la table de hachage.

NameServiceInterf : public void removeSharedObject(Address a, String objectName)

Cette méthode permet de retirer l'objet objectName des objets partagés.

La seule tâche du serveur de nom est donc de créer cet objet et de le partager.

Voici la description des méthodes offertes par le Discovery :

Discovery : public void searchServer(String serverAddress)

permet de trouver le serveur hébergeant l'objet NameService

Discovery : public RemoteInterface lookup(String objectName)

appelle la méthode lookup de l'objet NameService pour obtenir l'adresse de l'hôte partageant l'objet objectName. Connaissant cette adresse, la méthode lookup de ObjectNaming peut être invoquée pour obtenir la référence de l'objet désiré.

Discovery : public boolean bind(String objectName, Object o)

appelle la méthode bind de la classe ObjectNaming afin que les autres applications puissent accéder à l'objet, puis appelle la méthode addSharedObject de l'objet NameService en lui passant en paramètre sa propre adresse (qu'elle peut obtenir avec la méthode Address.getMyAddress) ainsi que le nom de l'objet.

La figure 3.15 décrit le fonctionnement de la méthode bind :

1. Le serveur de nom est créé par le serveur de Discovery
L'objet Toto est créé par l'hôte 2
2. Le serveur de Discovery partage le serveur de nom chez lui
3. L'hôte 2 partage l'objet Toto
 - 3.1. Il le partage chez lui
 - 3.2. Il recherche le serveur de nom chez le serveur de Discovery

3.3. Il appelle la méthode `addSharedObject` du serveur de nom qui indique qu'il partage l'objet Toto

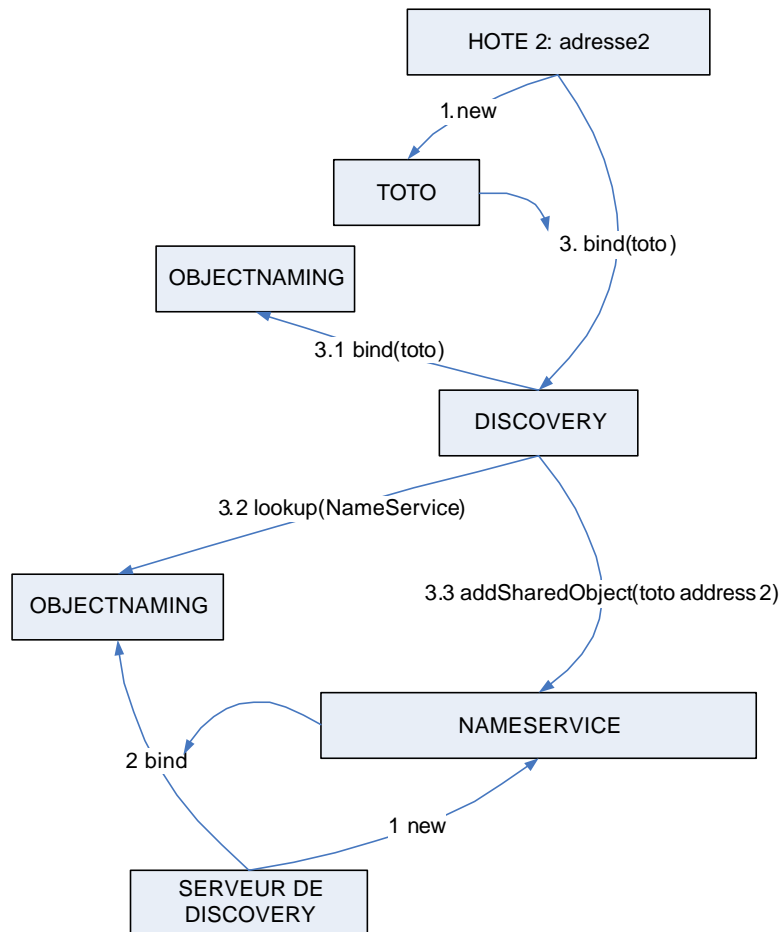


Figure 3.15

La figure 3.16 décrit le fonctionnement de la méthode `lookup` après que la méthode `bind` a été appelée :

1. L'hôte 1 appelle la méthode `lookup` du Discovery pour rechercher l'objet nommé Toto.
 - 1.1. Il recherche le serveur de nom chez le serveur de Discovery
 - 1.2. Il appelle la méthode `lookup` du serveur de nom pour obtenir l'adresse de l'hôte qui partage l'objet Toto
 - 1.3. Il appelle la méthode `lookup` chez l'hôte 2

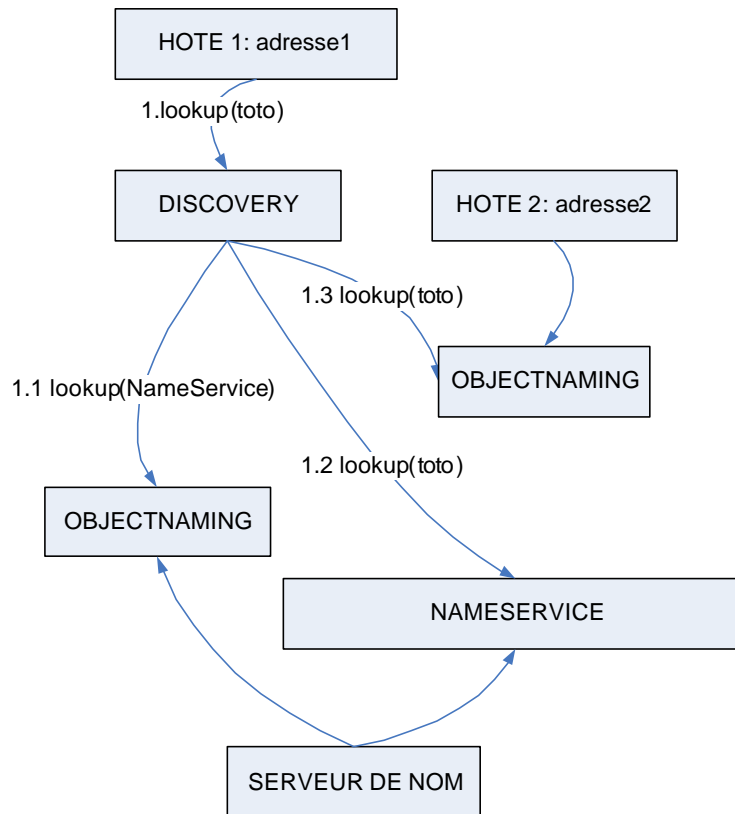


Figure 3.16

3.5.4 Exercice 4: Discovery

Dans cet exercice, les étudiants vont implémenter les méthodes de la classe Discovery décrites ci-dessus:

Discovery : public RemoteInterface lookup(String objectName)

Discovery : public boolean bind(String objectName, Object o)

Discovery : public boolean searchServer(String serverName)

A travers cet exercice, les étudiants vont utiliser les fonctionnalités du RPC afin d'offrir une nouvelle fonctionnalité.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 4), et les fichiers sources compressés dans le fichier Exercice4_Discovery.zip.

Les classes de test vérifient qu'une application puisse partager un objet et qu'une autre puisse invoquer une méthode sur cet objet sans connaître la localisation de cet objet, mais en ne connaissant que l'adresse du serveur contenant le service de nom.

4 Exercices supplémentaires

Deux exercices ne portant pas sur l'implémentation d'une partie d'un module particulier, mais sur l'utilisation de celles-ci sont proposés ici.

4.1 Exercice 0 : Application répartie

Cet exercice demande aux étudiants d'implémenter une application répartie basée sur le RPC. Le principe est que deux machines exécutant la même application puissent s'envoyer des messages. Cet exercice permet aux étudiants de se familiariser avec le Mini-Middleware et d'utiliser une de ses fonctionnalités.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 0) et les fichiers permettant d'effectuer l'exercice compressés dans le fichier Exercice0.zip

4.2 Exercice 5 : Publish/Subscribe

Cet exercice demande aux étudiants d'implémenter un module publish/subscribe basé sur le module de Discovery du Mini-Middleware. Cet exercice demande également des connaissances sur le principe d'échange de messages du Mini-Middleware ainsi que la sérialisation. C'est pourquoi il est proposé en dernier.

L'énoncé complet de cet exercice se trouve en annexe (Annexe 5) et les fichiers permettant d'effectuer l'exercice compressés dans le fichier Publish_Subscribe.zip

5 Conclusion

Pour réaliser un programme à but éducatif comme le Mini-Middleware, l'accent n'est pas mis sur les performances du programme, mais sur sa structure, sa facilité d'utilisation et sa simplicité, afin que son fonctionnement soit facilement compréhensible par un étudiant et donc mieux adapté à des exercices. Ce middleware a été divisé en différents modules offrant chacun une fonctionnalité particulière ce qui a permis de cibler chaque exercice sur un point particulier. Cette structure en modules lui permet également d'offrir des fonctionnalités de manière transparente vu que l'implémentation des différents modules est invisible à l'utilisateur. Ceci permet également d'utiliser d'autres implémentations pour les différents modules. Ceci a été notamment utilisé dans les modules de Transport et de Marshalling.

De plus, l'imbrication des différents modules entre eux offre une grande souplesse dans la hiérarchie des modules et permet ainsi l'insertion ou l'ajout facile de nouveaux modules.

L'élaboration de la structure du middleware a été une des grandes difficultés du projet dans la mesure où il fallait pouvoir estimer à l'avance les différents modes d'utilisation du middleware et les besoins qu'il devait satisfaire. En effet, chaque module offre une fonctionnalité particulière et pour définir cette fonctionnalité, il faut se demander dans quelles circonstances ces fonctionnalités seront utilisées. Je me suis inspirée des différents middleware existants pour définir les méthodes offertes par le Mini-Middleware, notamment RMI de Java pour effectuer le module RPC.

Ce middleware comporte tout de même certaines faiblesses, notamment le fait de devoir envoyer l'adresse du client lors de l'établissement d'une connexion ce qui le rend inutilisable si celui-ci est exécuté derrière un Firewall effectuant de la translation dynamique d'adresse, car l'adresse envoyée n'aura plus aucun sens pour le serveur. De plus, la partie marshalling comporte de grosses contraintes notamment le fait de ne pouvoir sérialiser que des objets dont les champs sont publics et comportant un constructeur vide. De plus, parmi les objets prédéfinis de Java, seule une toute petite partie a été traitée, ce qui réduit fortement son utilisation notamment par le module de RPC.

Les modes d'utilisation du Mini-Middleware sont décrits dans les énoncés des exercices proposés en annexe.