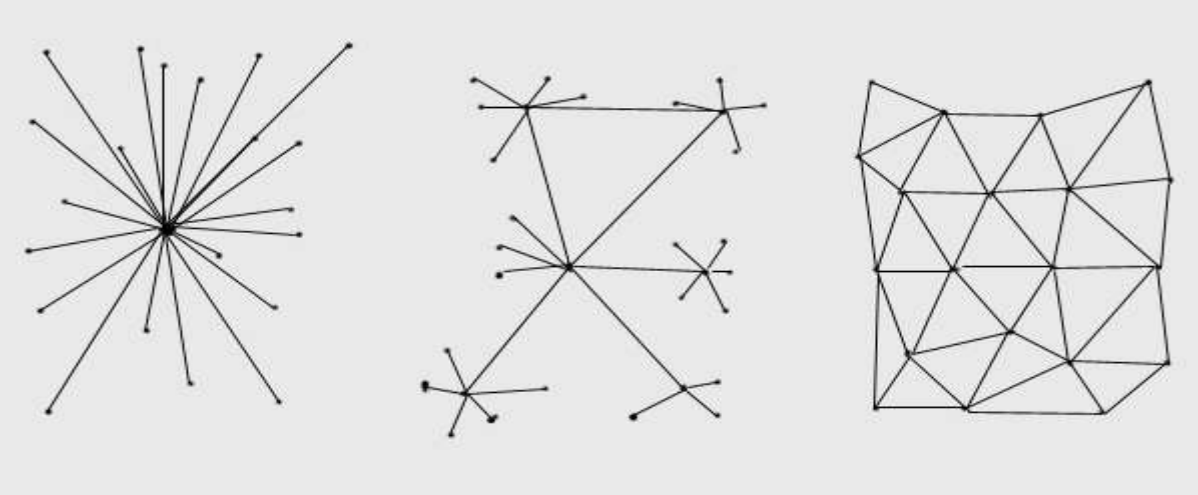
TP Applications Réparties



Programmation avec java.net & java.rmi

Sommaire

Table des matières

1. Objectif

L'objectif de ce TP est une introduction aux systèmes à agents mobiles, il complète aussi la mise en place d'applications réparties en utilisant la communication basique par TCP et le modèle RMI d'appel distant. Il permet aussi d'aborder succinctement l'aspect méta-modèle qu'une telle programmation peut nécessiter. Il consiste à programmer une application d'agents mobiles élémentaire[1].

Lorsqu'on répartit une application, on diminue la charge que l'on fait porter sur chaque nœud accueillant une partie de celle-ci. En contrepartie on augmente la charge liée aux interactions entre les différentes parties de l'application qui sont distribuées sur le réseau. Dans ce modèle les unités de calcul sont présentes sur les nœuds de l'application et elles sont sollicitées via des « requêtes ». Ceci est efficace quand les unités de calcul sont d'une faible variabilité. L'interrogation d'une base donnée est un exemple classique qui fonctionne très bien. Dans un certain nombre de cas le calcul à mettre en œuvre est plus spécifique et ne peut être généralisé aussi simplement.

Dans d'autres situations le surcoût de la communication peut devenir rédhibitoire ou, pour le moins, couteux alors une approche alternative doit être mis en place.

La solution consiste à déplacer, contrairement au modèle classique, le calcul afin qu'il s'exécute au plus proche des éléments dont il dépend. Le principe général se résume à un **bus à agents mobiles** (dénommé BAM dans la suite de ce document) : une architecture qui permet de déplacer des agents mobiles (dénommé agent dans la suite de ce document) sur des sites où ils s'exécuteront. Cette architecture peut être plus ou moins sophistiquée et offrir des services plus ou moins raffinés. Dans notre cas nous resterons suffisamment raisonnables afin de pouvoir mener à terme le développement.

De façon schématique ce BAM est constitué de serveurs dédiés à recevoir des agents et leur faire exécuter les tâches qu'ils ont à y effectuer. Dans notre cas, cette architecture est ouverte et pourrait évoluer dynamiquement. Les différents serveurs ne se connaissent pas et ont tous les mêmes droits (symétrie), il n'existe pas de serveur doté de privilèges supérieurs.

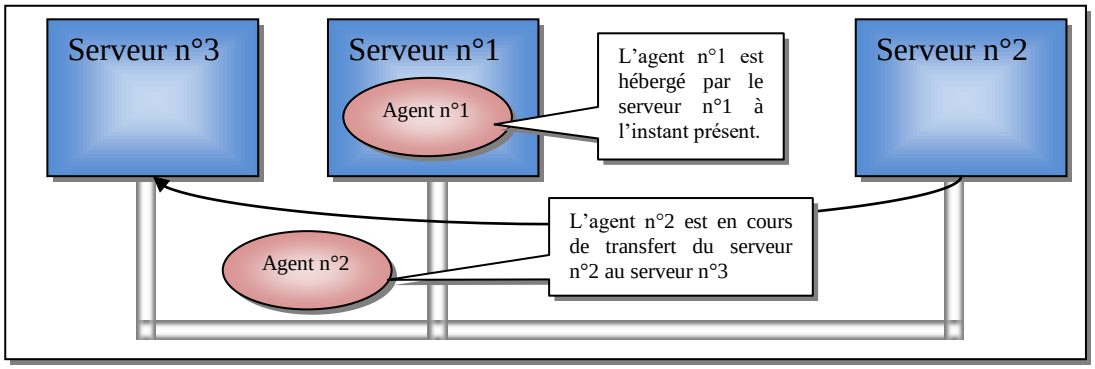


Schéma 1 : Schéma de principe

Une application simple et démonstrative de ce type de fonctionnement est la recherche d'informations. L'agent se déplace sur différents sites choisis pour y chercher l'information souhaitée. Pour compléter notre étude nous mettrons en place une

[1] Dixit Sherlock Holmes

solution classique et une à base d'agents mobiles puis nous étudierons les performances de chacune dans un scénario qui montre les différences de coût. On programmera cette application dans un premier temps à l'aide du modèle Client/serveur en utilisant la technologie RMI, puis avec le BAM Mobilagent. On établira des scénarios afin de comparer les performances de chacune des solutions.

Plus précisément l'objectif de ce TP est la spécification et la réalisation d'un BAM et d'une application test permettant de le faire fonctionner. Nous avons choisi une application classique consistant à faire une jointure entre des bases d'informations. D'un coté on pourra obtenir une liste de noms d'hôtels se trouvant dans une certaine localisation et d'un autre coté obtenir des numéros de téléphone, in fine le client obtiendra la liste des numéros de téléphone des hôtels correspondant à son critère de sélection (ici la localisation).

2. *Spécifications*

Un **Agent** est fortement autonome sachant quelle feuille de route il doit suivre. Il est initialement déployé sur un serveur et il terminera son parcours sur ce même serveur. Pour des raisons d'uniformisation un agent exécutera en premier une action vide sur son serveur de départ.

La feuille de route d'un agent décrit la succession de serveurs qu'il doit parcourir et les actions qu'il doit entreprendre sur chacun d'eux. Pour cela nous allons définir la notion de **Route** qui sera composée d'**Etape**s. Le comportement d'un agent pourrait être vu comme un statechart où les transitions correspondent aux transferts d'un serveur vers un autre et les états aux passages sur les serveurs où l'agent exécute ses actions. Une route sera représentée par une séquence d'étapes car dans un premier temps on ne souhaite pas avoir un modèle comportemental plus complexe.

Une étape sera caractérisée par la désignation du serveur utilisé et par l'**Action** à entreprendre. L'agent sera le seul détenteur des actions qu'il aura à exécuter sur les différents serveurs, ce qui lui confère cette autonomie le rendant peu dépendant des serveurs d'agents. Cette approche a pour corollaire que le BAM n'est pas une structure préétablie et figée. Les Serveurs d'agents n'ont pas vocation à se connaître et être interconnectés. Chaque agent, via sa feuille de route, établit son BAM. Il s'ensuit qu'un agent peut essayer de parcourir un BAM non opérationnel lorsqu'au moins un des serveurs souhaités n'est pas disponible. La sémantique des agents devra tenir compte de cette caractéristique.

Pour que ce modèle soit utilisable, il est préférable qu'un agent soit d'une taille raisonnable. Cette taille peut être déterminée par la dimension du code nécessaire à son exécution et celle des informations (attributs) qui le caractérisent. Cette dernière est entièrement spécifique au type de l'agent. D'autre part, il serait préférable qu'un agent ait des droits restreints sur l'environnement qui

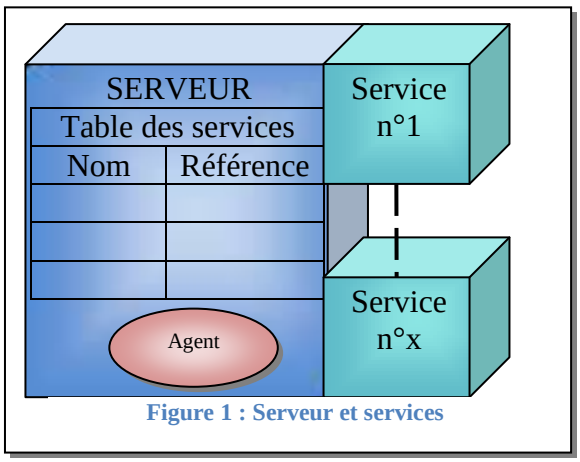


Figure 1 : Serveur et services

l'accueille lors de son passage sur un serveur et ce pour éviter qu'il n'y exécute des actions regrettables. Pour répondre à ces 2 attentes, on peut concevoir que des **Service**s seront accessibles aux agents. La probité de ceux-ci sera garantie par le(s) serveur(s) qui les héberge(nt). Ils pourront fournir des fonctionnalités aussi sophistiquées que nécessaire (Base de données par exemple) et de ce fait constituer une part importante du code de l'agent.

Pour qu'un agent ne puisse pas faire n'importe quoi, les codes exécutables qui lui seront accessibles doivent être réduits. Les mécanismes de protection fournis par Java permettent d'assurer ceci de plusieurs façons. Nous allons nous intéresser à celle que le principe de chargement des classes (ClassLoader) nous propose.

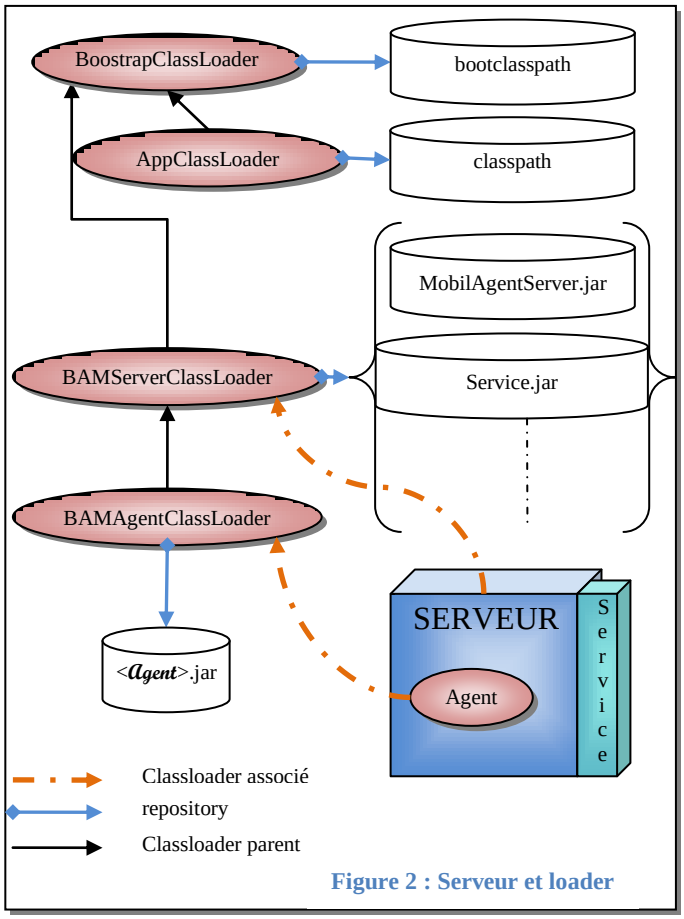


Figure 2 : Serveur et loader

Lorsqu'on a besoin de charger une classe, ceci est fait soit par le ClassLoader de la classe dont l'exécution entraine ce besoin, soit de façon programmée en utilisant un ClassLoader dédié. Le modèle Objet indique précisément qu'un objet est la réunion d'une structure de données qui caractérise son état (attributs) et des opérations (méthodes) pouvant les manipuler (modèle de fermeture). Classiquement dans une JVM un seul exemplaire des méthodes est nécessaire. Quand on fait migrer un objet, il est évident que la structure de données qui le caractérise individuellement doit être déplacée. Il en est de même de ses méthodes, sauf si ce code est déjà présent sur la destination. Cette dernière option est en contradiction avec nos objectifs où les serveurs ne fixent pas les fonctionnalités à priori. En conséquence, où qu'il aille, l'agent doit emporter avec lui le code (ensemble de classes) qui est nécessaire à son bon

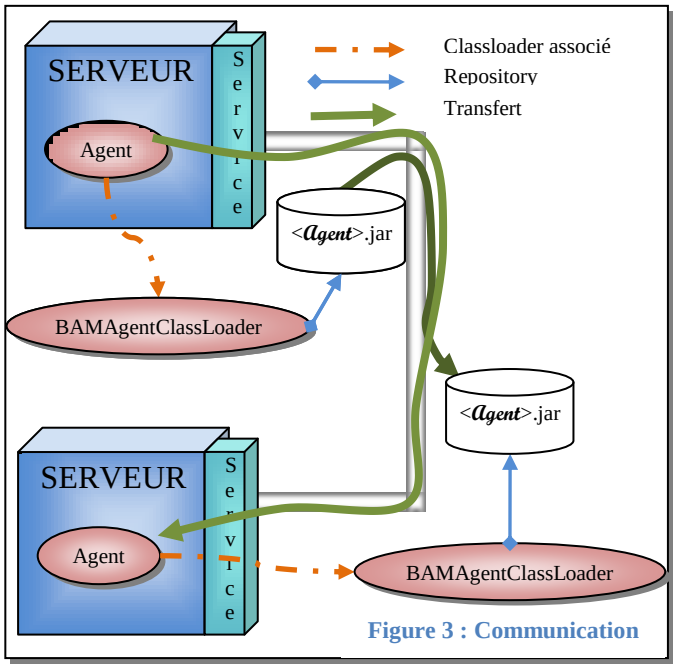


Figure 3 : Communication

fonctionnement. Chaque agent sera donc chargé par un **BAMAgentClassLoader** qui est un ClassLoader spécialisé dans notre système pour répondre à nos exigences. Afin de simplifier la manipulation, ce code sera rassemblé dans un fichier jar qui sera la seule source du repository.

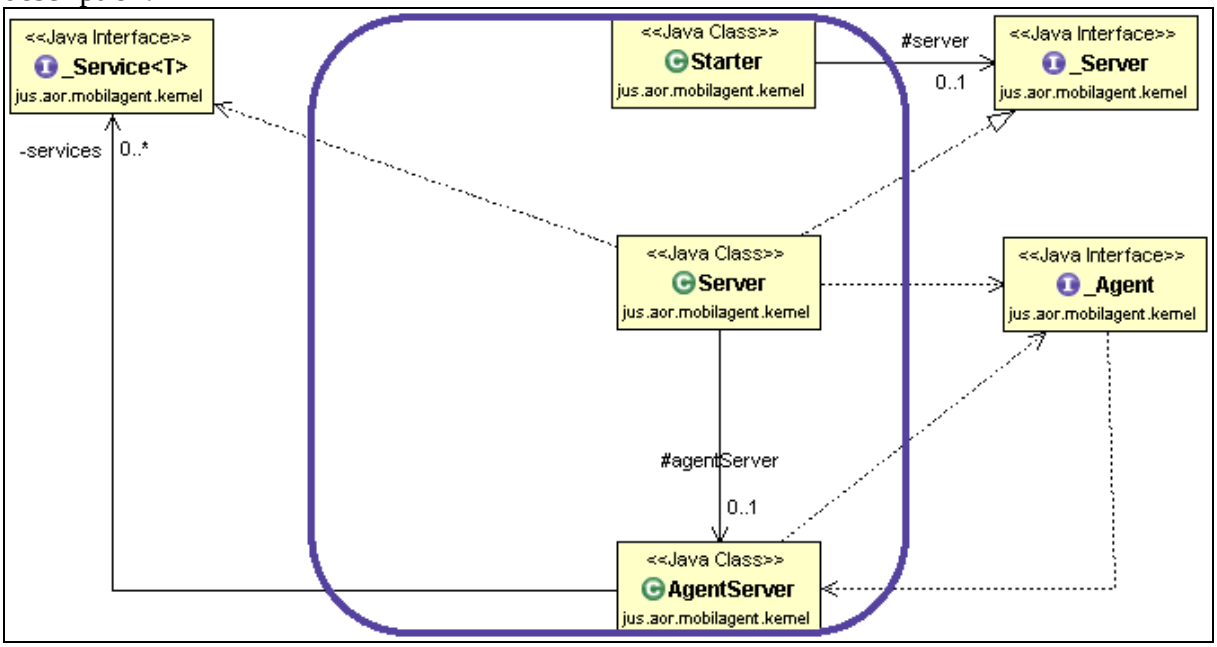
Pour limiter, autant que faire ce peut, les classes accessibles à un agent sans le restreindre de trop, il faut lui permettre d'accéder aux classes des services qu'il est en droit d'utiliser sur un serveur. Pour cela un serveur sera chargé avec un ClassLoader spécifique (**BAMServerClassLoader**) ne donnant accès qu'aux classes du noyau et des services ajoutés à ce serveur. Il dérivera d'un URLClassLoader permettant de disposer de toutes les fonctionnalités offertes par ce niveau et, entre autres, de pouvoir ajouter de nouveaux repositories (AddURL). On pourra étendre le repository lorsqu'un nouveau service sera installé sur ce serveur. On n'envisagera pas l'opération inverse.

L'ensemble des communications se fera par le biais du niveau Socket. On utilisera le mécanisme de transfert d'objet via leur sérialisation. Cette opération utilise des ObjectInputStream et ObjectOutputStream qui permettent de lire ou d'écrire des objets. Lors de la lecture d'un objet, la classe de celui-ci doit être chargeable.

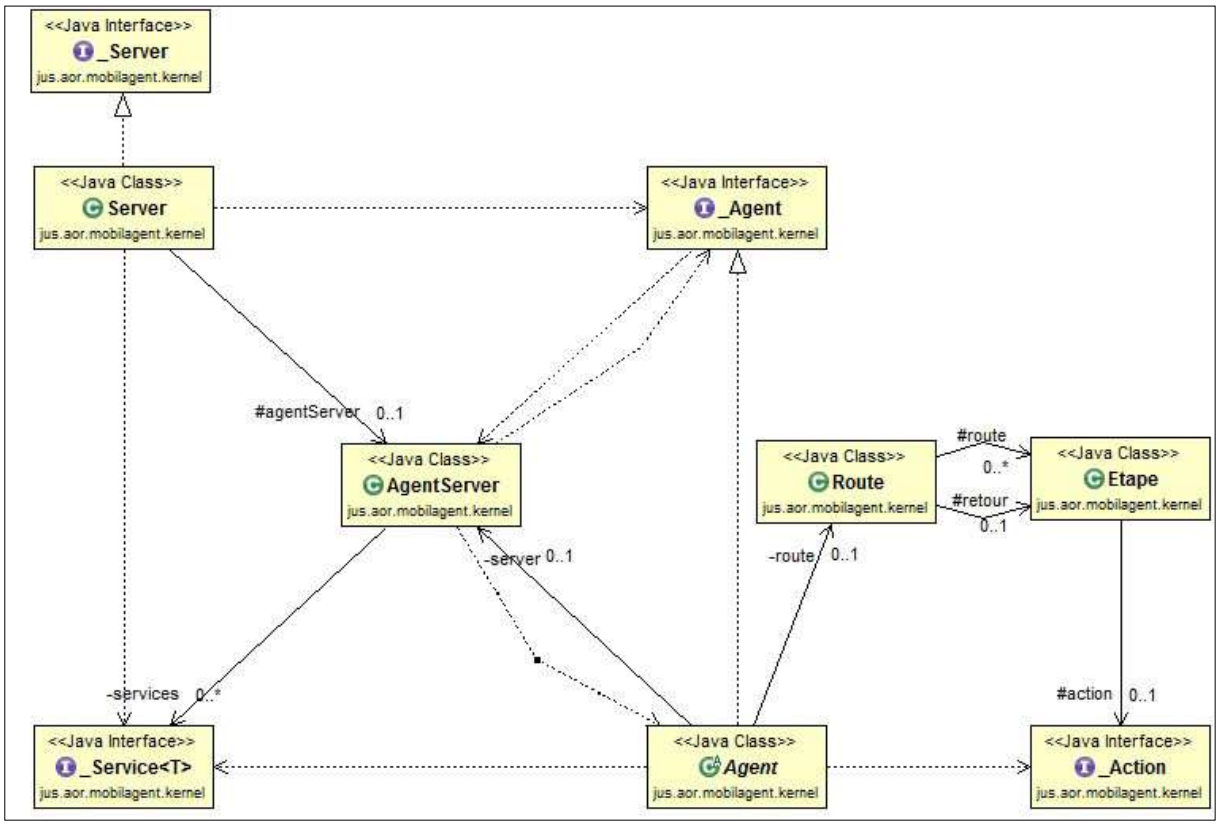
3. *Structure générale de l'application*

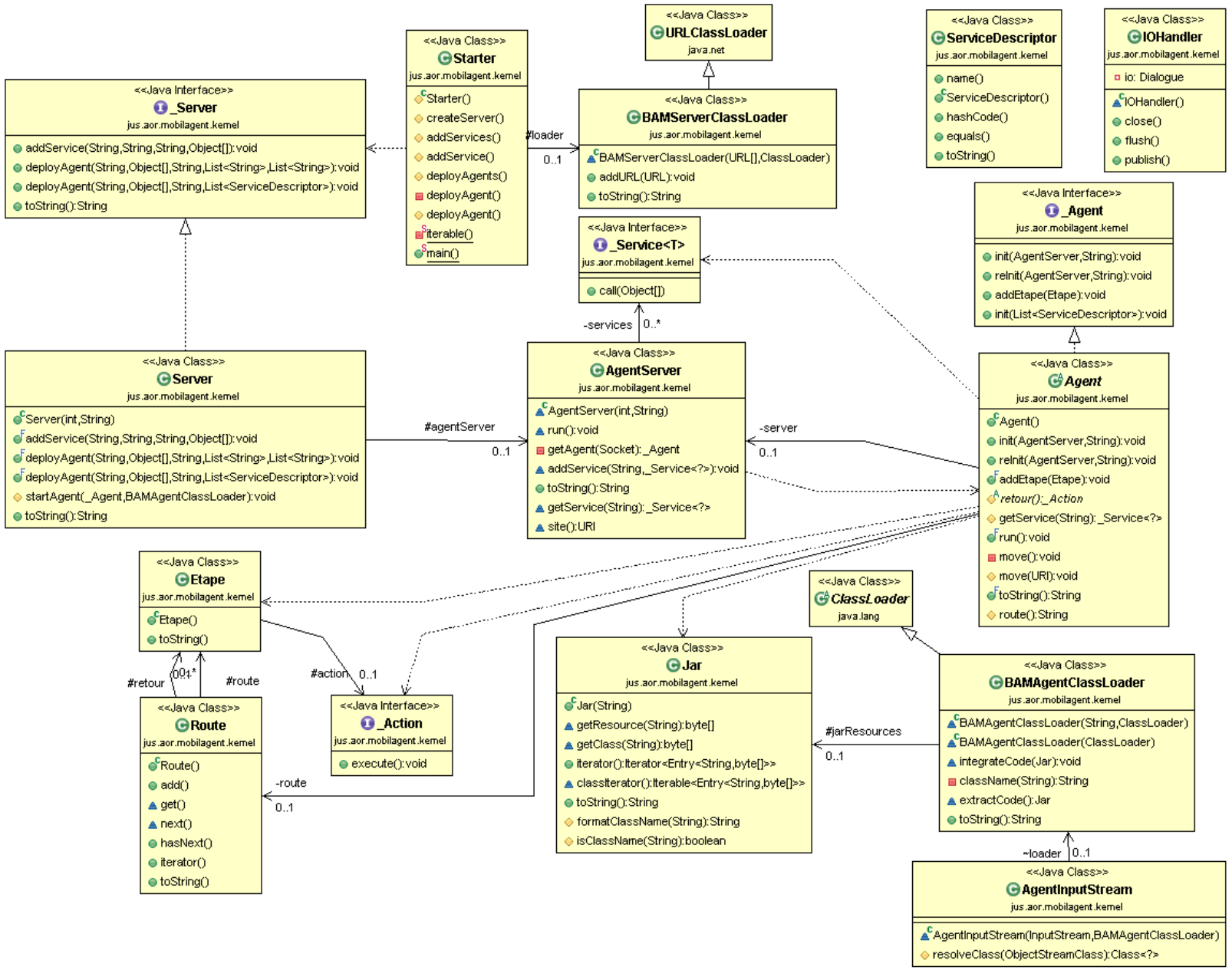
Les schémas ci-après montrent l'ensemble des concepts que vous aurez à mettre en place.
Un serveur du BAM sera réalisé par 3 classes :

- La classe AgentServer décrit la partie opérationnelle du fonctionnement de ce serveur. Elle contient la boucle (méthode run) de réception des agents mobiles.
- La classe Server est une classe qui protège la précédente, permet de gérer l'AgentServer et fournit les primitives d'ajout d'un service et de déploiement d'un agent.
- La classe Starter qui chapote la précédente et rend le système plus simple à l'usage en permettant de configurer les caractéristiques d'un serveur via un fichier xml de description.



Nous disposerons de 2 interfaces pour modéliser d'une part les services pouvant être installés sur les serveurs (_Service<T>) et d'autre part les agents qui pourront venir s'y exécuter (_Agent). La liaison entre le Starter et le Server n'apparaît pas sur le schéma car il se fait via un loader, le Server et l'AgentServer étant chargés avec le loader spécifique.
Chaque agent pourra exécuter des actions appropriées dont la modélisation est assurée par l'interface _Action. Chaque agent possède une feuille de route, nous la représenterons à l'aide des classes Route et Etape.




<<Java Interface>>
_Server
jus.aor.mobilagent.kernel
addService(String,String,String,Object[]):void
deployAgent(String,Object[],String,List<String>,List<String>):void
deployAgent(String,Object[],String,List<ServiceDescriptor>):void
toString():String
<<Java Class>>
Starter
jus.aor.mobilagent.kernel
Starter()
createServer()
addServices()
addService()
deployAgents()
deployAgent()
deployAgent()
Iterable()
main()
#loader
0..1
<<Java Class>>
URLClassLoader
java.net
<<Java Class>>
BAMServerClassLoader
jus.aor.mobilagent.kernel
BAMServerClassLoader(URL[],ClassLoader)
addURL(URL):void
toString():String
<<Java Class>>
ServiceDescriptor
jus.aor.mobilagent.kernel
name()
ServiceDescriptor()
hashCode()
equals()
toString()
<<Java Class>>
IOHandler
jus.aor.mobilagent.kernel
io: Dialogue
IOHandler()
close()
flush()
publish()
<<Java Interface>>
_Service<T>
jus.aor.mobilagent.kernel
call(Object[])
-services
0..*
<<Java Interface>>
_Agent
jus.aor.mobilagent.kernel
init(AgentServer,String):void
reInit(AgentServer,String):void
addEtape(Etape):void
init(List<ServiceDescriptor>):void
<<Java Class>>
Server
jus.aor.mobilagent.kernel
Server(int,String)
addService(String,String,String,Object[]):void
deployAgent(String,Object[],String,List<String>,List<String>):void
deployAgent(String,Object[],String,List<ServiceDescriptor>):void
startAgent(_Agent,BAMAgentClassLoader):void
toString():String
#agentServer
0..1
<<Java Class>>
AgentServer
jus.aor.mobilagent.kernel
AgentServer(int,String)
run():void
getAgent(Socket):_Agent
addService(String,_Service<?>):void
toString():String
getService(String):_Service<?>
site():URI
-server
0..1
<<Java Class>>
Agent
jus.aor.mobilagent.kernel
Agent()
init(AgentServer,String):void
reInit(AgentServer,String):void
addEtape(Etape):void
retour():_Action
getService(String):_Service<?>
run():void
move():void
move(URI):void
toString():String
route():String
<<Java Class>>
Etape
jus.aor.mobilagent.kernel
Etape()
toString()
#retour
0..1
0..*
#route
#action
0..1
<<Java Class>>
Route
jus.aor.mobilagent.kernel
Route()
add()
get()
next()
hasNext()
iterator()
toString()
<<Java Interface>>
_Action
jus.aor.mobilagent.kernel
execute():void
-route
0..1
<<Java Class>>
Jar
jus.aor.mobilagent.kernel
Jar(String)
getResource(String):byte[]
getClass(String):byte[]
iterator():Iterator<Entry<String,byte[]>>
classIterator():Iterable<Entry<String,byte[]>>
toString():String
formatClassName(String):String
isClassName(String):boolean
<<Java Class>>
ClassLoader
java.lang
#jarResources
0..1
<<Java Class>>
BAMAgentClassLoader
jus.aor.mobilagent.kernel
BAMAgentClassLoader(String,ClassLoader)
BAMAgentClassLoader(ClassLoader)
integrateCode(Jar):void
className(String):String
extractCode():Jar
toString():String
~loader
0..1
<<Java Class>>
AgentInputStream
jus.aor.mobilagent.kernel
AgentInputStream(InputStream,BAMAgentClassLoader)
resolveClass(ObjectStreamClass):Class<?>

4. Travail à réaliser

Notre environnement sera doté de serveurs hébergeant des services. Le service Chaine représente une chaîne d'hôtels (par exemple les plus prisés : TonCarl, Letifos, …) capable de fournir l'ensemble des hôtels qu'elle possède dans une localisation géographique donnée. Le service Annuaire quant à lui permet, étant donnée une identification (un simple nom dans notre cas), de délivrer son numéro de téléphone. Ceci sera modélisé par les interfaces _Chaine et _Annuaire ainsi que les classes Chaine, Annuaire, Hotel et Numero. Enfin un client souhaitant effectuer une interrogation sera modélisé par la classe LookForHotel.

Il est à noter que l'Objectif n°1 (usage de RMI) est indépendant des objectifs n°2,3&4 (BAM). Ils peuvent donc être traités simultanément via une répartition judicieuse des tâches tenant compte du fait que la partie BAM nécessite certainement plus d'efforts pour atteindre l'équivalent de l'objectif n°1.

Objectif n°1. Réalisation de LookForHotel avec le modèle RMI

Le principe de la version RMI consiste à interroger successivement les différents serveurs de chaines d'hôtels permettant d'obtenir l'ensemble des hôtels se trouvant dans la localisation demandée, puis à interroger le serveur d'annuaire pour obtenir les numéros de téléphones de ces différents hôtels. Réalisez la classe LookForHotel qui assure cette recherche. Mettez en place un mécanisme pour évaluer le coût de cette approche.

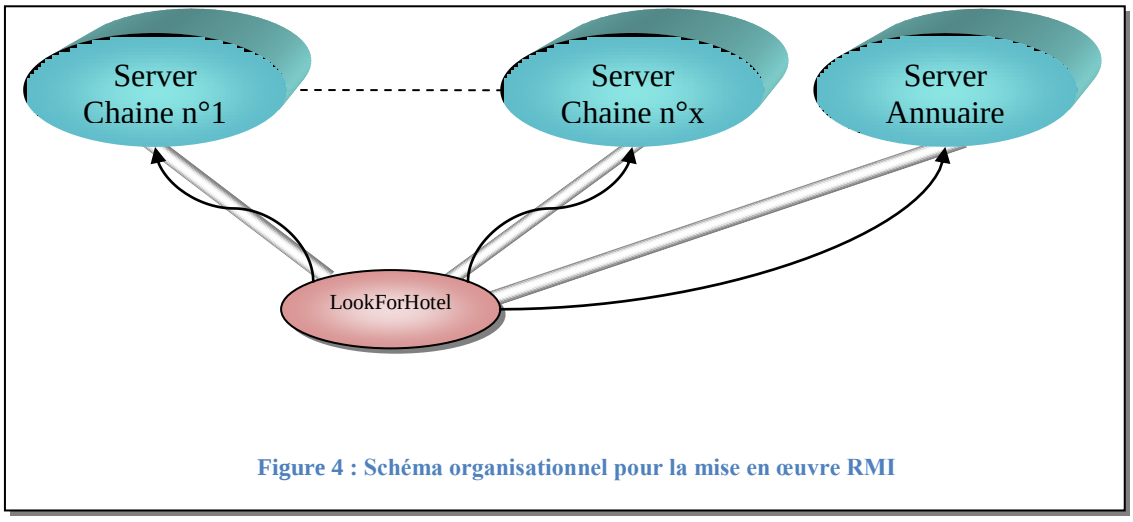


Figure 4 : Schéma organisationnel pour la mise en œuvre RMI

Attention il ne s'agit pas de l'objectif principal, il sera repris dans Objectif n°4 avec le modèle BAM. Il est indépendant des Objectif n°2, 3 & 4. Il peut donc être réalisé n'importe quand avant l'Objectif n°5 dont il sert de base. Dans l'architecture de votre projet vous prendrez soin d'isoler correctement les différentes parties indépendantes.

Objectif n°2. Etude de la hiérarchie de classes

Analysez le diagramme de classes qui représente les concepts que vous avez mis en évidence à la lecture de la présentation ci-dessus.

Objectif n°3. Réalisation du BAM

Complétez les classes qui vous sont fournies en fonction de leurs spécifications afin de réaliser l'infrastructure BAM.

Faites un essai de votre infrastructure avec l'agent Hello dont on vous fourni les prémisses et qui permet à un agent de suivre une feuille de route où sa seule action se résume à imprimer et noter son passage sur un serveur. In fine il imprimera, lors de son retour, la totalité de cette information.

- Dans votre application combien d'objets ont-ils permis de représenter l'agent Hello ?
- Comment montrer cette diversité lors de l'exécution ?
- Comment quantifier le coût de transfert d'un agent ?

Objectif n°4. Réalisation de LookForHotel avec le modèle BAM

On veut mettre en place un système de recherche d'informations similaire à celui développé dans l'Objectif n°1. Reprenez l'exemple de l'accès à des chaînes d'hôtels. Vous réaliserez un agent LookForHotel qui aura la mission de collecter auprès des serveurs de chaines d'hôtels l'ensemble des hôtels se trouvant dans la localisation choisie et d'obtenir auprès du serveur d'annuaire les numéros des différents hôtels sélectionnées.

Pour mettre en place cet agent on prendra l'option d'avoir des services spécialisés, très similaires à ceux que vous aurez fait dans la version de l'Objectif n°1. Un service Chaine pour les serveurs des chaines d'hôtels et un service Annuaire pour le serveur d'annuaire.

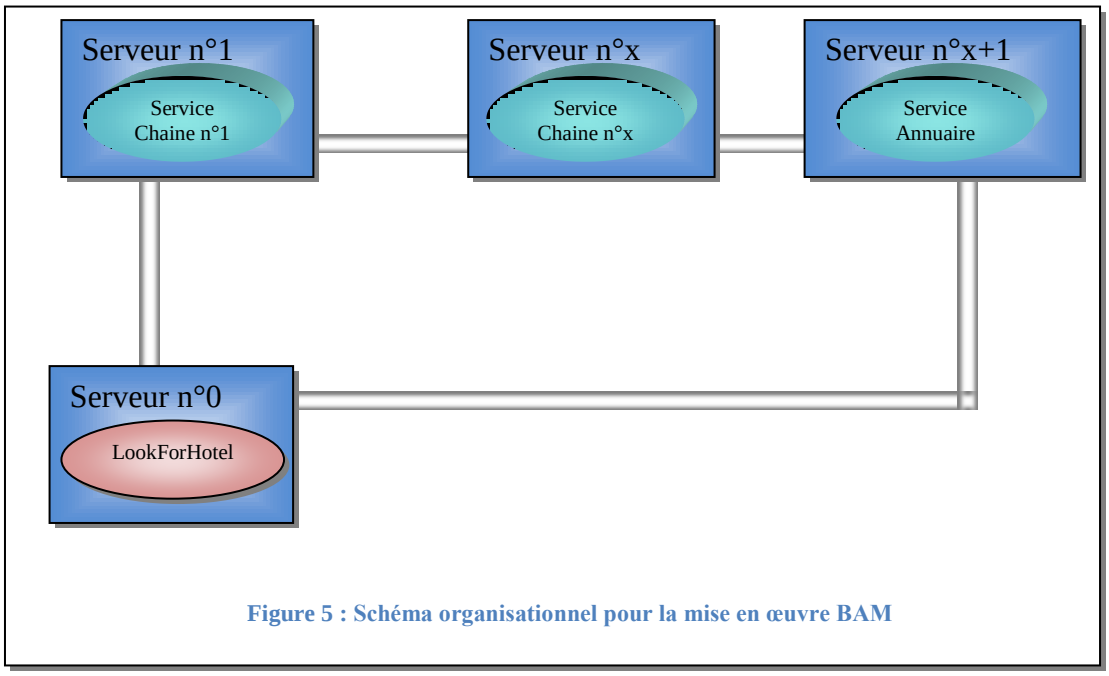


Figure 5 : Schéma organisationnel pour la mise en œuvre BAM

Vous construirez les jars nécessaires pour chaque service et pour l'agent ainsi que pour le démarrage de l'application. Ceci afin que le minimum de code soit mis en jeu lors des constructions et transferts dans l'application.

Lors du démarrage d'une JVM pour un serveur d'agent il peut être nécessaire de valuer la propriété suivante : **-Dsun.lang.ClassLoader.allowArraySyntax=true**. Si vous testez sur une machine unique (situation la plus probable) vous devez construire des jars représentant les ressources accessibles aux différents composants de votre application et construire les lanceurs en conséquence.

Objectif n°5. Comparaison des 2 réalisations

Donnez les définitions formelles des coûts des 2 solutions qui permettent d'avoir une bonne estimation du rapport de la solution utilisant RMI et de celle utilisant le BAM. Validez votre proposition par l'expérimentation, vous pourrez utiliser un tableur pour présenter les résultats obtenus.

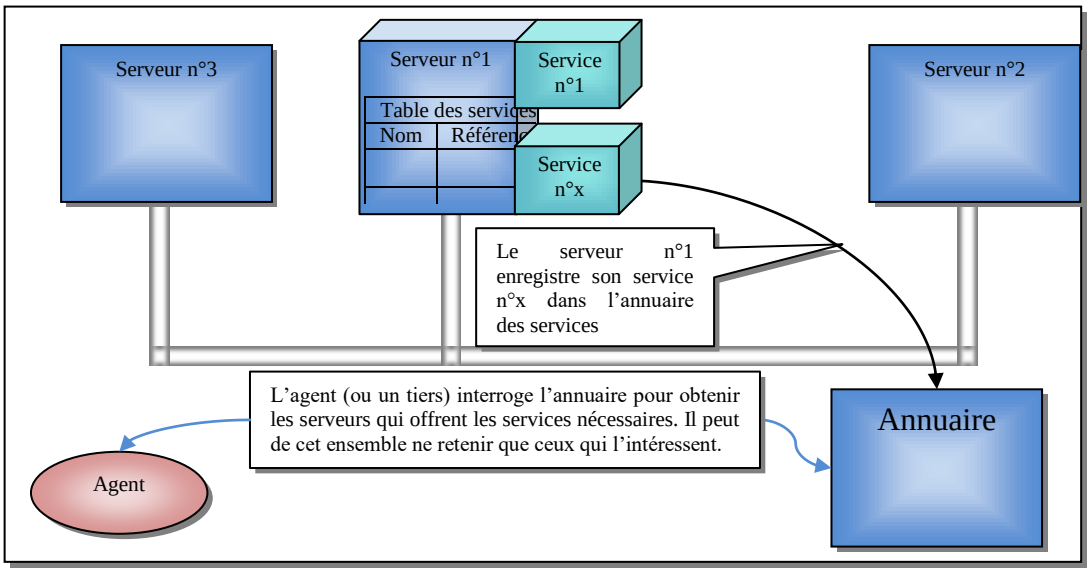
Objectif n°6. Ajout d'un annuaire de courtage.

Afin de facilité la constitution de la feuille de route d'un agent, on souhaite mettre en œuvre un système d'annuaire et d'interrogation de la disponibilité de services.

L'annuaire de services offrira la possibilité aux serveurs du BAM de s'y inscrire et de pouvoir ainsi être consultés. Il offrira aussi la possibilité d'être interrogé sur la disponibilité d'un service.

Dans le cas de l'interrogation, on peut estimer que certains services sont systématiquement rendus disponibles par tout serveur BAM et que d'autres doivent être explicitement déclarés comme utilisables sur un serveur BAM. On peut aussi prévoir que les serveurs BAM soient caractérisés par des propriétés permettant l'expression de sélections.

Adaptez la classe serveur pour fournir les nouvelles fonctionnalités requises et réalisez le service d'annuaire centralisé. Pour cela vous utiliserez la technologie RMI.



Annexes

1. Principe du ClassLoader

Pour chaque application tournant sur une JVM, il existe une arborescence de classloaders qui permet d'accéder aux différentes ressources requises par l'application. Lorsqu'une d'entre elles vient à manquer, une exception est émise (par exemple l'exception ClassNotFoundException). Le schéma ci-dessous montre l'organisation initiale au lancement d'une application. On dispose d'un classloader qui donne accès aux ressources (classes) de la librairie standard de Java, un second qui donne accès aux extensions contenu dans les fichiers jar stockés dans le directory d'extension et enfin un troisième qui donne accès aux ressources désignées par le classpath. Toute application peut compléter cette arborescence et faire usage de classloaders spécifiques. Lorsqu'un nouveau classloader est créé, il est attaché à un classloader existant. Cette structure reste donc strictement arborescente. En l'absence d'indication de classloader parent, c'est la racine qui est choisie.

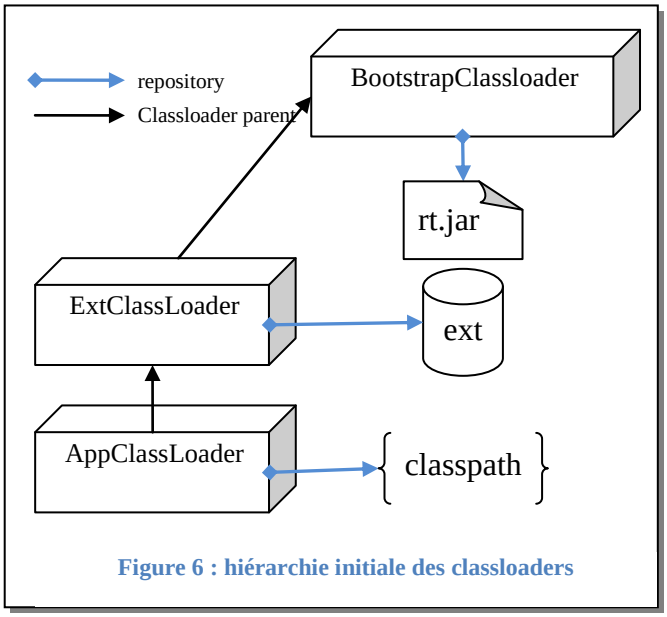


Figure 6 : hiérarchie initiale des classloaders

2. ClassLoader et transmission via des sockets

Lors de la transmission d'un objet via le mécanisme de sérialisation, il est nécessaire que la JVM réceptrice dispose de la classe typant cet objet afin de le restaurer correctement lors de l'opération « unmarshalling ». Il est bien sûr nécessaire que la classe de l'objet transmis soit accessible dès que l'objet lui-même est transmis pour que cette fonction d'unmarshalling puisse reconstruire l'objet dans sa forme standard.
Dans le modèle RMI soit celle-ci fait partie du classpath de la JVM réceptrice, soit du codebase attaché en annotation à l'objet transmis via RMI. Le codebase peut être considéré comme une extension du classpath dans ce cas particulier.
Dans le cas d'une transmission par un moyen plus élémentaire comme le niveau socket, cette nécessité n'est plus nécessairement garantie par l'environnement, il faut donc que le concepteur s'en assure ou fournisse le moyen d'y répondre.
La classe ObjectInputStream qui est classiquement utilisée pour lire des objets via un support quelconque offre entre-autres une fonctionnalité permettant de trouver une classe compatible à la description du type fourni :

```
protected  Class<?> resolveClass(ObjectStreamClass desc)
```
Load the local class equivalent of the specified stream class description.

C'est un moyen qui a l'avantage de se combiner aisément avec le mécanisme de classLoader dédié présenté précédemment. Cette méthode peut faire appel à un classLoader spécifique pour retrouver la classe de l'objet que l'on veut « désérialiser ». On vous fournit la classe AgentInputStream qui assure ce comportement.

3. Principe du Logging

Un logger offre un mécanisme de trace permettant d'observer (conserver, …) le comportement d'une application. Il est une alternative plus sérieuse au sempiternel « System.out.println » que l'on utilise régulièrement pour la mise au point des programmes. Au-delà de ce simple usage on parle de journalisation consistant à conserver les traces, sur un support quelconque, des évènements survenus dans une application.
Un logger offre des méthodes permettant de réceptionner une demande de journalisation, certaines d'entre-elles sont spécialisées pour correspondre à des situations récurrentes bien identifiées (par exemple l'entrée ou la sortie d'une méthode, …). Il permet d'associer à chaque demande une sévérité (sur une échelle établie) dont l'usage à postériori peut être multiple (par exemple le filtrage, …). Un logger est identifié par un nom au sein d'un pool unique de loggers. Le nom doit être choisi de façon pertinente pour potentiellement ne pas entrer en conflit avec d'autres. La classe Logger offre le moyen de récupérer (ou créer) un logger (getLogger) dans le pool. Un logger fait usage d'un ou plusieurs Handler qui sont des outils de prise en compte de la demande de journalisation permettant d'associer un traitement particulier à cette demande (par exemple : impression sur la console, enregistrement sur un fichier, …). Certains types d'Handler sont prédéfinis pour un usage immédiat. Chacun peut à sa guise réaliser des Handlers spécifiques. Pour compléter le traitement applicable à une demande de journalisation le Logger dispose aussi de Filters qui permettent, comme leurs noms l'indiquent, de réaliser des filtrages. Un Logger peut avoir un Logger parent dont il peut hériter des caractéristiques et à qui il peut retransmettre une demande de journalisation. Ceci permet de mettre en place des systèmes en couches de l'observation.

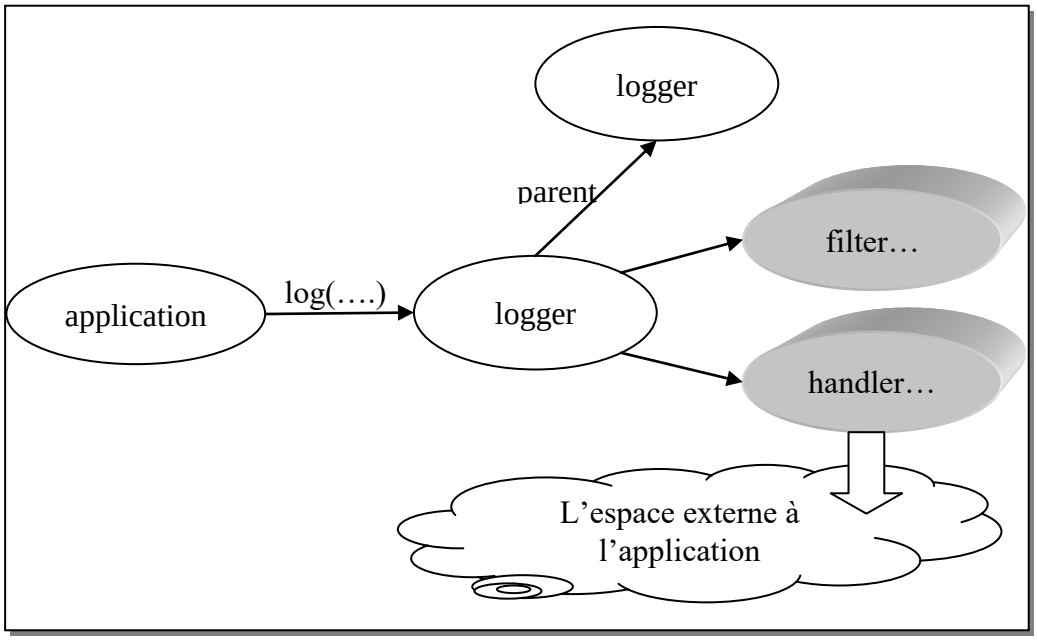


Figure 7 : schéma des intervenants d'un logging

Une description plus détaillée est accessible à l'adresse :
http://docs.oracle.com/javase/6/docs/technotes/guides/logging/overview.html.

4. Les fichiers de configuration

Chaque serveur doit être configuré lors de son démarrage pour correspondre à la situation souhaitée en termes de services présents sur celui-ci et d'agents à déployer lors de son démarrage. Pour éviter de passer cet ensemble d'informations dans la commande de lancement, nous utilisons des fichiers de configurations contenant ces dites informations. Pour être lisible et aisément éditable, nous avons décidé d'utiliser le format XML qui s'y prête bien. Le format choisi est structuré comme suit :