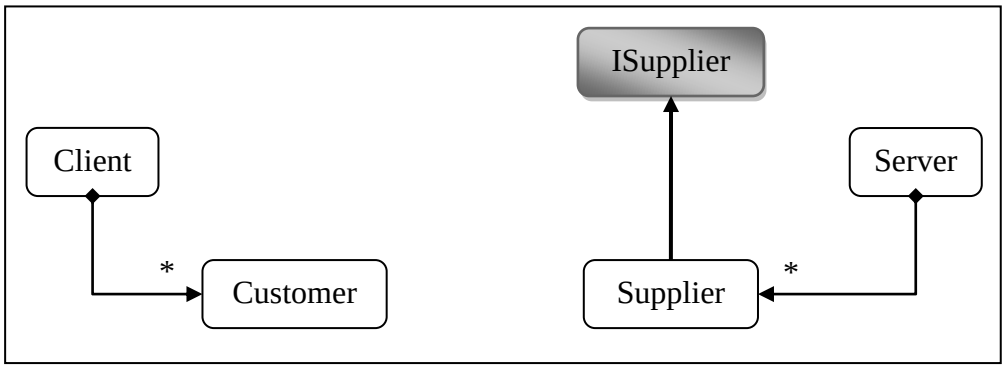
TD Application Répartie

P.Morat & F.Boyer

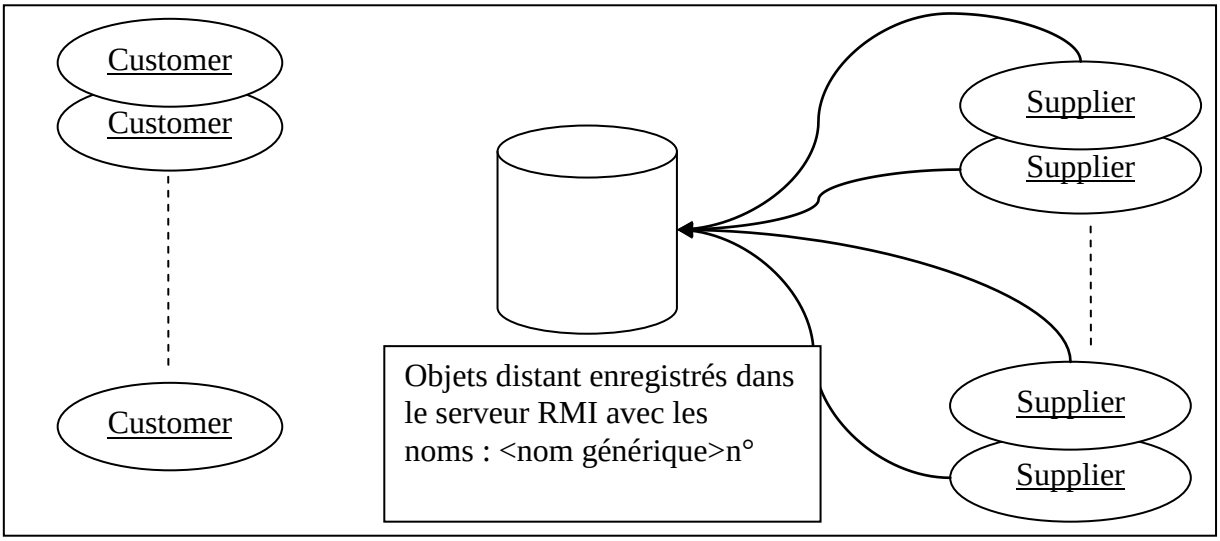
Pour vous familiariser avec la mise en œuvre du modèle RMI, on vous propose de constituer une application permettant à un client distant d'interroger un serveur pour obtenir les valeurs de propriétés de l'environnement d'exécution.
L'architecture générale de l'application sera constituée de 4 classes et d'une interface comme le montre le schéma ci-dessous.



La classe Client constitue le programme principal de lancement des clients qui sont représentés par la classe Customer. La classe Server constitue le programme principal de lancement des objets distants qui sont représentés par la classe Supplier.

Vous utiliserez l'environnement Eclipse pour la suite des opérations à effectuer. Pour cela constituer un projet comprenant trois répertoires sources : un pour la partie client, un pour la partie server et le troisième pour la partie commune.

L'application mettra en œuvre plusieurs Customer et plusieurs Supplier pour être au plus près de la réalité.



1) Consulter l'interface ISupplier.
2) Consulter et compléter la classe Server. L'application « Server » admet 3 options qui sont : le nom générique des objets distants, le port du rmiregistry et le nombre d'objets distants accessibles. Le nom d'un objet distant est constitué du nom générique suivi du rang de l'objet.
3) Consulter et compléter la classe Client. L'application « Client » admet 4 options qui sont : la désignation de la machine distante à interroger, le nom générique des objets distants, le nombre d'objets distants accessibles et le nombre d'objet clients à construire.
4) Consulter et compléter la classe Customer.
5) Réaliser la classe **Supplier**. Chaque Supplier est identifié par un numéro unique qui lui est fourni lors de sa construction. La méthode toString de cette classe aura la forme suivante : public String toString(){ return "supplier"+num ;} où num est l'attribut contenant le numéro d'identité. Vous ferez tracer le passage dans chaque méthode par un affichage sur la console.
6) A cette étape nous disposons de tous les éléments nécessaires au fonctionnement de l'application. Dans Run Dialog insérer un nouveau launcher pour exécuter le serveur. Dans la partie Arguments insérer les options de l'application, dans la partie VM arguments insérer les options suivantes en les complétant :

   -Djava.security.policy="…" -Djava.security.manager -Djava.rmi.server.codebase="…"

   Dans un premier temps vous utiliserez le protocole file pour désigner le codebase : soit file://<le chemin du directory terminé par / >, Pour cela créer un répertoire CodeBase dans votre projet. Vous fixerez dans la rubrique classpath que seul le fichier jar créé est utilisable.

   Faites de même pour exécuter le client. Dans la partie Arguments insérer les options de l'application, dans la partie VM arguments insérer les options suivantes en les complétant :

   -Djava.security.policy="…" -Djava.security.manager
7) Après avoir compilé l'ensemble des classes, constituer un fichier jar nommé Client.jar contenant les entités Client, Customer et ISupplier et ayant comme main la classe Client. Faites de même pour le fichier Server.jar contenant les entités Server, Supplier et ISupplier avec la classe Server comme main class.
8) Refaire l'étape 6) en utilisant cette fois ci seulement le fichier jar destiné au serveur (resp. au client).
9) Insérer dans votre projet le fichier ant.xml, utilisez la vue ant pour démarrer un rmiregistry en réalisant des ajustements si nécessaire.
10) Lancer le serveur via le run Dialog, si des erreurs d'exécution apparaissent, modifier l'environnement d'exécution en conséquence.
11) Lancer le client de la même manière.
12) Pourquoi utilise-t-on la méthode name pour identifier l'objet distant ? remplacer cet appel par celui de obj.toString() et constater le comportement.

13) On souhaite ajouter le service suivant dans la classe Supplier et sa spécification dans l'interface ISupplier :

```
/**
 * met à jour le paramètre s en le complétant par =<valeur de la
propriété correspondante>
 * @param s
 */
public void question(Appendable s) throws RemoteException{
      System.out.println("calling question(Appendable)");
      s.append(question(s.toString());
}
```

Ajouter dans la classe Customer le traitement au cas correspondant :

```
}case 2: { // cas de l'utilisation de question(StringBuffer){
      Appendable d = new StringBuffer(select());
      System.err.print(this+"->"+obj.name() + ".question("+d+") = ");
      obj.question(d);
      System.out.println(d);
      break;
```

Reconstituer l'application et tester de nouveau. Le comportement est-il celui-attendu. Pourquoi l'utilisation de la troisième forme de « question » est-elle insatisfaisante ?

- Proposez le principe d'une solution au problème, on développera la solution à la question 17).

14) On souhaite ajouter le service suivant dans la classe Supplier et sa spécification dans l'interface ISupplier :

```
public IProperty question(StringBuilder s) throws RemoteException{
      System.out.println("calling question(StringBuilder)");
      return new Property(System.getProperty(s.toString()));
}
```

Ajouter dans la classe Customer le traitement au cas correspondant :

```
}case 3 :{ // cas de l'utilisation de question(StringBuilder){
      StringBuilder d = new StringBuilder(select());
      System.out.println(this+"->"+obj.name()+".question("+d+") =
"+obj.question(d).value());
      break;
```

Soit la classe `Property` qui implémente l'interface IProperty qui doit être réalisée.

Reconstituer l'application et tester de nouveau. Modifier l'environnement d'exécution pour que le système fonctionne correctement.

15) Vider le fichier de policy de tout droit, et par essais successifs constituer le système de droits minimal nécessaire. Il est préférable de différencier les droits du serveur de ceux du client.

16) Utiliser un serveur http pour mettre en place le codebase avec un protocole http. Construire un répertoire nommé « rmi » dans l'espace du serveur http choisi et y placer les éléments se trouvant dans le répertoire CodeBase.

17) On revient sur le problème mis en exergue à la question 13), proposez une solution au problème (attention requiert plus d'attention !!!).

ANNEXES :

Le codebase

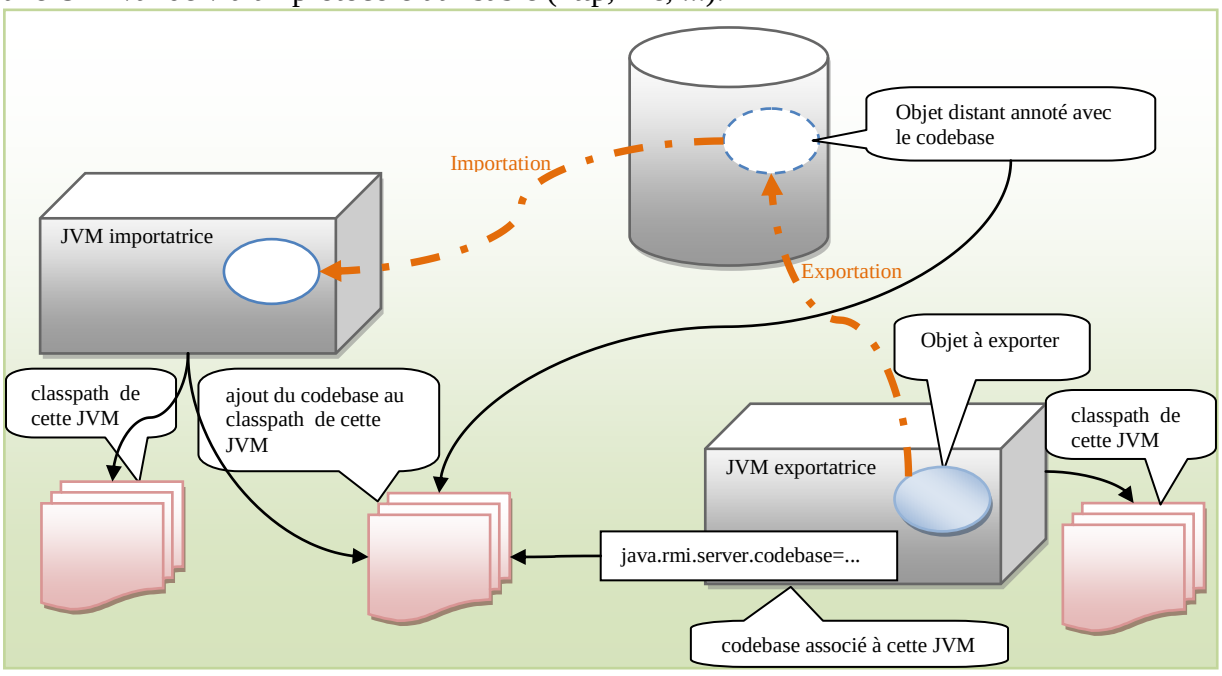
L'utilisation distante d'un objet nécessite sur la JVM importatrice de charger au minimum la classe du "stub" de l'objet distant, il est évident que celle-ci est a priori inconnue du client. Il faut donc que la JVM importatrice puisse accéder lors de son exécution à cette ressource via un classLoader. Pour cela le codebase fixe le ou les endroits supplémentaires où la JVM importatrice pourra faire ses recherches. Plusieurs façons pour transmettre cette information sont envisageables :

1. Le principe initial consiste à annoter l'objet distant stocké dans l'annuaire avec l'information du codebase. Lors du chargement de celui-ci sur la machine importatrice elle donc est en mesure de récupérer cette information. Depuis la version 1.7xxx, pour que cette approche fonctionne, il faut positionner la variable d'environnement **`java.rmi.server.useCodebaseOnly`** à false[1].

   > If this value is `true`, automatic loading of classes is prohibited *except* from the local CLASSPATH and from the `java.rmi.server.codebase` property set on this VM. Use of this property prevents client VMs from dynamically downloading bytecodes from other codebases.

   Dans le cas contraire (à true) il faut appliquer le cas n°4.
2. L'annuaire d'objets distants (RmiRegistry) est démarré de façon indépendante avec un classpath qui correspond au codebase, on se retrouve dans la même situation que précédemment.
3. L'annuaire d'objets distants (RmiRegistry) est démarré au sein de la JVM exportatrice, le classpath contient le codebase, on se retrouve dans la même situation que précédemment.
4. Les JVMs exportatrice et importatrice ont le même codebase.

La troisième est la plus simple à mettre en œuvre.
Il faut dans tous les cas que les ressources soient accessibles par la machine importatrice par une URI valide via un protocole utilisable (http, file, ...).



[1] Personnellement, ceci n'a jamais fonctionné !

Génération à la volée

Depuis la version 1.5 il est possible de ne pas pré-générer les classes de Stub, donc de ne plus faire l'étape "rmic". A la création d'un objet distribuable, si la classe Stub correspondante n'est pas trouvée (ou si la génération dynamique est forcée) une classe dynamique est engendrée à partir de la classe Proxy. Ceci ne supprime pas la nécessité d'un codebase pour fournir l'ensemble des autres ressources nécessaires à la distribution (classes annexes).