Maîtrise Informatique
M5: Programmation Répartie et Architecture N 1/3

DS – Session de Juin 2002

Denis Caromel — Lionel Mestre

Professeur
Université de Nice - Sophia Antipolis, Département Informatique

---

Les documents et calculatrices sont autorisés.
Prenez le temps de bien lire et de bien comprendre les questions !
Les réponses ne sont pas longues, soyez *concis* et ... *lisible* !
Notez *clairement* sur votre feuille les numéros des questions (exemple: 3. b.).
Durée: 2H00

1 (3 points)

Questionnaire à choix multiples:
Pour chaque question, il y a 0, 1, ou plusieurs réponses correctes. Marquez sur votre feuille le numéro de la question, et le numéro des **réponse correctes**.

Par exemple:
Question 1:
A: 1, et 4
B: aucune
C: toutes

Question A:
Dans le cadre de la programmation Java RMI :

1. On se retrouve de facto avec deux systèmes de typage.
2. On bénéficie de la sérialisation automatique des objets qui ne sont pas accessibles à distance.
3. Un protocole standard est défini afin de pouvoir faire communiquer entre elles n'importe quelles machines virtuelles Java.
4. On peut également rechercher des objets distants par les interfaces implémentées, mais à condition que celles-ci soient `Remote`.
5. On bénéficie également de services de sécurité et de transaction.

Question B:
Dans le cadre de la programmation CORBA :

1. On bénéficie de l'interopérabilité au niveau des langages (C++, Cobol, Java).
2. A partir de CORBA 2, on bénéficie également de l'interopérabilité au niveau des implémentations avec en particulier IIOP.
3. Il y a beaucoup plus de services standards définis que dans Java RMI.
4. Il est également défini des mapping IDL spécifiques aux principaux langages.
5. On se retrouve avec plus de deux systèmes de typage.

Question C:
Sur les EJB Java:

1. EJB Home et EJB Objet ne sont en fait tous les deux que des objets RMI accessibles à distance.
2. Grâce au standard, un Bean "Statefull" va être appelé par plusieurs clients.
3. Un Bean "Entity" ne peut jamais être associé à un seul client.
4. La liaison entre les classes et les interfaces ne se fait pas par programmation, mais dans un descripteur.
5. Des outils vont permettre la composition hiérarchique des Beans.

2 (4 points)

a. Rappeler brièvement le role du **codebase** en Java RMI (1 point).
b. Proposer une implémentation : simplement les principes, mais en détaillant bien les points importants (2 points).
c. Commenter cette implémentation (performance en temps et en espace) et mentionner éventuellement quelques optimisations (1 point).

3 (4 points)

Dans le cadre du modèle de composant CORBA 3, on souhaite spécifier un ensemble de composants pour la modélisation d'une voiture (très très simplifiée!), ceci afin de programmer ultérieurement une simulation par exemple.

a. Dans un premier temps, spécifier un composant `Roue` qui possède un attribut `taille` et peut émettre l'événement `CreuvaisonEvt`. D'autre part, ce composant offre une "*Facet*" de nom `RoueVueParMoteur` qui ne possède que deux méthodes sans paramètres et retournant *void*: `unTourAvant()` et `unTourArriere()`.
b. Spécifier maintenant un composant `Moteur` qui:
- utilise 4 roues,
- peut émettre l'événement `SurchauffeEvt`,
- peut utiliser l'événement `CreuvaisonEvt`,
- offre une "*Facet*" de nom `moteurVueParConducteur` ayant deux méthodes sans paramètres et retournant *void*: `accelere()` et `freine()`.
c. Enfin spécifier un composant `Conducteur` qui utilise un moteur, et peut utiliser les événements `CreuvaisonEvt` et `SurchauffeEvt`.
d. Avec les conventions de représentation utilisées dans le cours, faites une représentation graphique de l'assemblage naturel de quatre instances du composant *Roue*, avec une instance de chacun des composants *Moteur* et *Conducteur*.
L'événement `CreuvaisonEvt` sera propagé au moteur et au conducteur.

4 (4 points)

Question de synthèse : en une demi-page à une page grand maximum, donner de façon synthétique les différences principales entre les plateformes J2EE et .NET

5 (7 points)

Dans cette partie le but est de comprendre le fonctionnement et le déploiement d'un programme RMI existant. Le programme est composé d'un serveur (la classe Server) et d'un client (la classe Client).

Le serveur est lancé sur la machine `hostServeur` :

> java Server
Server bound in registry at the url rmi://hostServeur/Server

Le client est lancé sur la machine `hostClient` :

> java Client rmi://hostServeur/Server

Le serveur est un objet Remote qui s'enregistre dans le RMIRegistry de sorte que le client puisse récupérer une référence sur lui. Le serveur propose l'interface Remote `TestServer` qui permet de demander un objet Test de type A, B ou C.

Test est une interface implémenté par la classe TestImpl et TestProxy. En fonction du type, A, B ou C demandé par le client, le serveur retourne une instance de TestImpl ou de TestProxy (voir le code de Server).

On ne s'intéresse pas ici aux problèmes de déploiement tel que le chargement de code : on suppose que le client comme le serveur et le RMIRegistry partagent le même CLASSPATH.

Voici la liste des classes après compilation :

```
TestServer.class Client.class Server.class Server_Stub.class Server_Skel.class
Test.class RemoteTest.class TestImpl.class TestProxy.class
TestProxy$RemoteTestImpl.class TestProxy$RemoteTestImpl_Skel.class
TestProxy$RemoteTestImpl_Stub.class
```

On notera que la classe `RemoteTestImp` est une classe interne de `TestProxy` se qui explique la notation avec $.

Voici le code Java des fichiers sources :

——————— **Server** ——————— :

```
public class Server extends java.rmi.server.UnicastRemoteObject implements TestServer {

  private final static String REGISTRY_ENTRY_ID = "Server";
  private Test myTest;

  public Server() throws java.rmi.RemoteException {
    myTest = new TestImpl();
  }

  public Test getTest(int type) throws java.rmi.RemoteException {
    switch (type) {
      case TYPE_A:
        return myTest;
      case TYPE_B:
        return new TestProxy(new TestImpl());
      case TYPE_C:
        return new TestProxy(myTest);
    }
    return null;
  }

  public static void main(String args[]) {
    try {
      Server server = new Server();
      // Bind this object instance to the name "HelloServer"
      java.rmi.Naming.rebind(REGISTRY_ENTRY_ID, server);
      System.out.println(" Server bound in registry at the url ");
      System.out.println("
        rmi://"+java.net.InetAddress.getLocalHost().getHostName()+"/"+REGISTRY_ENTRY_ID);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

```
}
```

——————— **client** ———————:

```
public class Client {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("You must pass the URL of the remote Server such
                        as rmi://clio.unice.fr/Server");
      System.exit(1);
    }
    TestServer server = null;
    // get a remote reference (a stub) on the remote server
    try {
      server = (TestServer) java.rmi.Naming.lookup(args[0]);
    } catch (Exception e) {
      e.printStackTrace();
    }
    // use the remote reference to call the method
    if (server == null) return;
    try {
      printTest(server.getTest(TestServer.TYPE_A));
      printTest(server.getTest(TestServer.TYPE_B));
      printTest(server.getTest(TestServer.TYPE_C));
      printTest(server.getTest(TestServer.TYPE_A));
    } catch (java.rmi.RemoteException e) {
      e.printStackTrace();
    }
  }

  public static void printTest(Test ct) {
    System.out.println("v="+ct.getValue()+" on "+ct.getCurrentHost());
  }
}
```

——————— **Test** ———————:

```
public interface Test {
  public int getValue();
  public String getCurrentHost();
}
```

——————— **RemoteTest** ———————:

```
public interface RemoteTest extends java.rmi.Remote {
  public int getValue() throws java.rmi.RemoteException;
  public String getCurrentHost() throws java.rmi.RemoteException;
}
```

——————— **TestImpl** ———————:

```
public class TestImpl implements Test, java.io.Serializable {

  protected int value = 0;
  protected String currentHost;

  public TestImpl() {
    this.currentHost = getCurrentHostInternal();
  }

  public synchronized int getValue() {
    System.out.println(" --> TestImpl.getValue");
```

```
    value = value + 1;
    return value;
  }

  public String getCurrentHost() {
    System.out.println(" --> TestImpl.getCurrentHost");
    return currentHost;
  }

  private String getCurrentHostInternal() {
    try {
      return java.net.InetAddress.getLocalHost().getHostName();
    } catch (java.net.UnknownHostException e) {
      return "UNKNOWN";
    }
  }

  private void writeObject(java.io.ObjectOutputStream out) throws
                                                 java.io.IOException {
    out.defaultWriteObject();
    System.out.println(" --> TestImpl.writeObject");
  }


  private void readObject(java.io.ObjectInputStream in) throws
                          java.io.IOException, ClassNotFoundException {
    in.defaultReadObject();
    currentHost = getCurrentHostInternal();
    System.out.println(" --> TestImpl.readObject");
  }
}
```

——————— **TestProxy** ———————:

```
public class TestProxy implements Test, java.io.Serializable {

  private RemoteTest remoteTest;

  public TestProxy(Test test) throws java.rmi.RemoteException {
    this.remoteTest = new RemoteTestImpl(test);
  }

  public int getValue() {
    System.out.println(" --> TestProxy.getValue");
    try {
      return remoteTest.getValue();
    } catch (java.rmi.RemoteException e) {
      return -1;
    }
  }

  public String getCurrentHost() {
    System.out.println(" --> TestProxy.getCurrentHost");
    try {
      return remoteTest.getCurrentHost();
    } catch (java.rmi.RemoteException e) {
      return null;
    }
```

```
  }

  private void writeObject(java.io.ObjectOutputStream out) throws
                                             java.io.IOException {
    out.defaultWriteObject();
    System.out.println(" --> TestProxy.writeObject");
  }

  private void readObject(java.io.ObjectInputStream in) throws
                           java.io.IOException, ClassNotFoundException {

    in.defaultReadObject();
    System.out.println(" --> TestProxy.readObject");
  }
```

——————— **RemoteTestImpl** ———————:

```
  private class RemoteTestImpl extends java.rmi.server.UnicastRemoteObject
                                                    implements RemoteTest {
    private Test test;

    public RemoteTestImpl(Test test) throws java.rmi.RemoteException {
      this.test = test;
    }

    public int getValue() throws java.rmi.RemoteException {
      System.out.println(" --> RemoteTestImpl.getValue");
      return test.getValue();
    }

    public String getCurrentHost() throws java.rmi.RemoteException {
      System.out.println(" --> RemoteTestImpl.getCurrentHost");
      return test.getCurrentHost();
    }
  }
}
```

On rappelle que les méthodes privées readObject et writeObject sont appelées automatique par le mécanisme de sérialisation de Java à chaque fois que l'objet que les implémente est desérialisé ou sérialisé respectivement.

Questions :

a. Vous devez donner ce qui s'affiche sur la console du client d'une part et sur celle du serveur d'autre part. On considère ici une exécution normale où aucune exception n'est jeté.
   Le serveur étant déjà lancé, partez du moment où le client est lancé en suivant le fil d'exécution de celui-ci et des appels distants qu'il réalise.
   Chaque instruction System.out.println produit une trace du coté client ou serveur que vous devez écrire dans une des deux colonnes de votre réponse.
   Numérotez chaque ligne d'affichage de sorte a donner un ordre total entre les affichages du client et du serveur.
b. Donnez une représentation schématique (dessin) des objets existants sur les deux machines hostServeur et hostClient juste avant que ce premier client termine son exécution.
c. Le même serveur que celui de la première exécution étant toujours lancé, on relance un client sur la même machine après que le premier ait terminé son exécution. Donnez la trace de ce client sur la console.
   On ne demande pas la trace du serveur cette fois-ci!