Modèle client-serveur

Daniel Hagimont

IRIT / ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

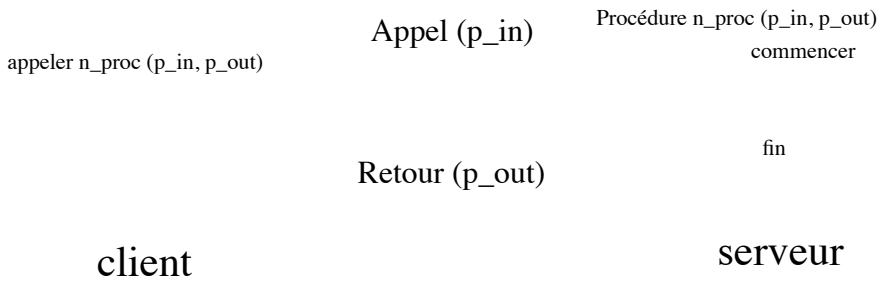
1

Cette conférence porte sur le modèle client- serveur. Il passe en revue les concepts et les illustre par son instanciation dans l'environnement Java, avec Remote Invocation de méthode (RMI).

Page 2

Modèle client-serveur basé sur la transmission du message

- Deux messages échangés (au moins)
  - ➢ Le premier message correspond à la demande. Il inclut les paramètres de la demande.
  - ➢ Le deuxième message correspond à la réponse. Il inclut les paramètres de résultat de la réponse.



Les interactions client-serveur peuvent être implémentées avec le passage de messages (en utilisant prises).

Vous avez alors au moins 2 messages échangés pour une telle interaction.

Le premier message correspond à la demande, y compris les paramètres, et le

le deuxième message correspond à la réponse, y compris les paramètres de résultat. L'exécution du client est suspendue après l'envoi de la demande, jusqu'à réception de la réponse.

On peut observer qu'une telle interaction ressemble à un appel de procédure, sauf que l'appelant (client) et l'appelé (serveur) sont situés sur des machines différentes.

Page 3

Appel de procédure à distance (RPC)
Des principes

- Génération de la majeure partie du code
  - ➢ Emission et réception de messages
  - ➢ Détection et réémission des messages perdus
- Objectifs: le développeur doit pouvoir programmer l'application sans avoir à gérer les messages

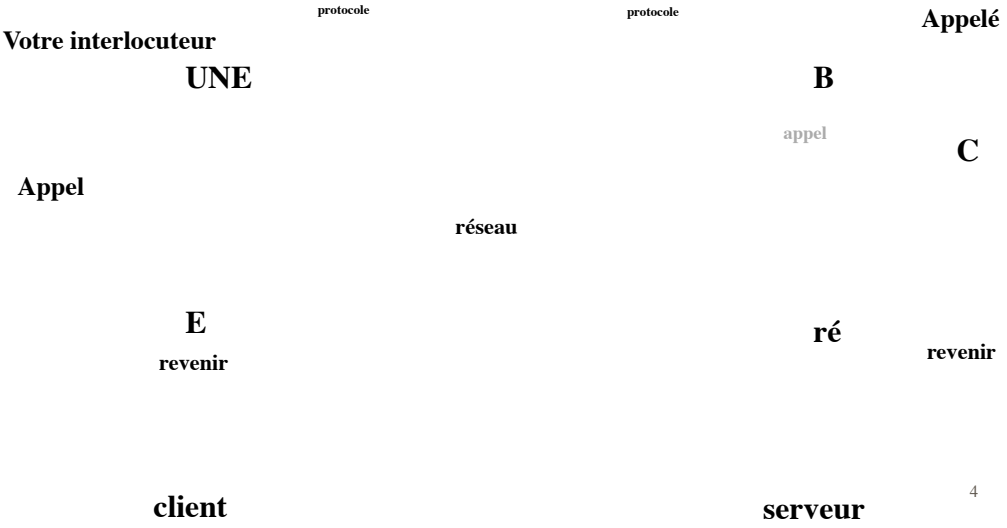
3

Nous appelons RPC (Remote Procedure Call) un outil qui simplifie la développement d'applications reposant sur de telles interactions client-serveur, en générer le code qui implémente les échanges de messages (requêtes et réponses). L'idée est que tout ce code peut être généré à partir d'une description de l'interface de la procédure (qui peut être appelée sur le serveur à partir d'un client distant).

L'objectif est de permettre au développeur de programmer et de tester son application en s'il était centralisé (exécuté sur une machine) sans la charge de traiter avec des échanges de messages. Le code permettant de distribuer l'application peuvent être générés et le code de l'application reste simple.

Page 4

RPC [Birrel et Nelson 84]
Principe de mise en œuvre



C'est le principe général des outils RPC.

En bleu, vous avez 2 segments de code, l'appelant dans le client qui appelle (appeler) un service et l'appelé dans le serveur qui fournit le service.

Dans un environnement centralisé, l'appel serait un simple appel de procédure entre l'appelant et l'appelé.

Le principe d'un outil RPC est de générer 2 segments de code (en marron) appelés le stub client (à gauche) et le stub de serveur (à droite).

Le stub client représente le service sur l'ordinateur client et donne le illusion que le service est local (peut être appelé localement avec un simple appel de procédure). Le stub client implémente la même procédure que le serveur afin de donner cette illusion. Un appel à la procédure dans le stub client crée et envoie un message de demande au serveur. Le stub de serveur reçoit et transforme les messages de demande en appels de procédure locale sur la machine serveur.

Page 5

RPC (point A)
Principe de mise en œuvre

- Du côté de l'appelant
  - ➢ Le client fait un appel procédural au stub client
    - Les paramètres de la procédure sont transmis au stub
  - ➢ Au point A
    - Le stub collecte les paramètres et assemble un message comprenant les paramètres (paramètre triage)
    - Un identifiant est généré pour l'appel RPC et inclus dans le message
    - Une minuterie de surveillance est initialisée
    - Problème: comment obtenir l'adresse du serveur (un service de dénomination registres procédures / serveurs)
    - Le stub transmet le message au protocole de transport pour émission sur le réseau

5

Du côté de l'appelant, le client effectue un appel de procédure (appelant le service) comme si le service était local sur l'ordinateur client. Notez que le stub client implémente la même procédure que le serveur, mais l'implémentation de celle-ci la procédure est différente.

Au point A, le stub client est appelé et reçoit les paramètres du appel de procédure. Il assemble un message de requête qui inclut ces paramètres (cette étape est appelée marshalling des paramètres). Un identifiant pour cet appel RPC est généré et inclus dans le message de demande. Cet identifiant permet de détecter côté serveur la réception de 2 requêtes pour le même appel (si le message est censé être perdu et réémis).

Un minuteur de surveillance est initialisé. Il se réveille après un temps donné. Si nous ne le faisons pas recevoir une réponse avant le réveil, nous considérons que la demande a été perdue et la demande est réémise.

Un problème ici est d'obtenir l'adresse (IP / port) du processus serveur pour l'envoi demandes. Généralement, un service de nommage permet d'enregistrer les procédures disponibles et leurs adresses.

Le stub peut alors envoyer le message de demande avec le protocole de communication (généralement UDP car les données à transmettre ne sont pas volumineuses).

Le client est alors suspendu, attendant le message de réponse.

Page 6

RPC (points B et C)
Principe de mise en œuvre

- Du côté de l'appelé
  - ➢ Le protocole de transport délivre le message au RPC service (stub de serveur)
  - ➢ Au point B
    - Le stub du serveur désassemble les paramètres (paramètre unmarshalling)
    - L'identifiant RPC est enregistré
  - ➢ L'appel est ensuite transmis à la procédure distante qui est exécuté (point C)
  - ➢ Le retour de la procédure revient au stub de serveur qui reçoit les paramètres de résultat (point D)

6

Du côté de l'appelé, le protocole de communication délivre le message de demande à le stub du serveur. Au point B, le stub de serveur désassemble les paramètres du call (cette étape est appelée unmarshalling de paramètre). L'identifiant RPC est enregistré pour détecter les demandes redondantes pour le même appel.

L'appel est alors reproduit, c'est-à-dire que la procédure à appeler dans l'appelé est effectivement appelé (point C). Il s'agit d'un appel de procédure normal. Au retour, le procédure retourne (point D) au stub du serveur (avec un résultat paramètres).

Page 7

RPC (point D)
Principe de mise en œuvre

- Du côté de l'appelé
  - ➢ Au point D
    - Les paramètres de résultat sont assemblés dans un message
    - Un autre chronomètre de chien de garde est initialisé
    - Le stub du serveur transmet le message au transport protocole d'émission sur le réseau

sept

Au point D, le stub de serveur assemble les paramètres de résultat dans une réponse message.

Un autre minuteur de surveillance est initialisé. Il se réveille après un temps donné. Si nous ne le faisons pas recevoir un accusé de réception du client (que la réponse a été reçue)

avant le réveil, on considère que la réponse a été perdue et la réponse est re
émis.

Le stub de serveur peut alors envoyer la réponse avec le protocole de communication.

Page 8

RPC (point E)
Principe de mise en œuvre

- Du côté de l'appelant
  - ➢ Le protocole de transport délivre le message de réponse à le service RPC (stub client)
  - ➢ Au point E
    - Le stub client désassemble les paramètres de résultat (paramètre unmarshalling)
    - Le minuteur de surveillance créé au point A est désactivé
    - Un message d'acquittement avec l'identifiant RPC est envoyé au stub du serveur (le minuteur de surveillance créé à le point D peut être désactivé)
    - Les paramètres de résultat sont transmis à l'appelant avec un retour de procédure

8

Du côté de l'appelant, le protocole de communication délivre le message de réponse à le stub client.

Au point E, le stub client désassemble les paramètres de résultat (paramètre unmarshalling).

La minuterie de surveillance créée au point A peut être désactivée.

Un message d'accusé de réception avec l'identifiant RPC est envoyé au stub du serveur (la minuterie de surveillance créée au point D peut être désactivée).

Les paramètres de résultat sont transmis à l'appelant avec un retour de procédure.

Page 9

RPC
Rôle des talons

Client stub Stub de serveur

- C'est la procédure qui interfaces avec le client
  - ➢ Reçoit l'appel localement
  - ➢ Le transforme en un appel à distance avec un envoyé message
  - ➢ Reçoit les résultats dans un message
  - ➢ Renvoie les résultats avec un retour de procédure normale

- C'est la procédure sur le nœud de serveur
  - ➢ Reçoit l'appel comme un message
  - ➢ Effectue la procédure appel sur le nœud serveur
  - ➢ Reçoit les résultats du appeler localement
  - ➢ Transmet les résultats à distance sous forme de message

9

Nous résumons ici le rôle du stub client et du stub serveur.

Page 10

RPC
Perte de m essage

- Côté client
  - ➢ Si le chien de garde expire
    - Réémission du message (avec le même identifiant RPC)
    - Abandonner après N tentatives
- Côté serveur
  - ➢ Si le chien de garde expire
  - ➢ Ou si nous recevons un message avec un identifiant RPC connu
    - Réémission du message de réponse
    - Abandonner après N tentatives
- Côté client
  - ➢ Si nous recevons un message avec un identifiant RPC connu
    - Réémission du message d'acquittement

dix

Nous fournissons ici une vue globale de la gestion de la perte de message.

Côté client, nous avons créé un chien de garde avant d'envoyer la requête. Si cela watchdog expire, nous pouvons supposer que la requête a été perdue et nous renvoyer le request avec le même identifiant RPC (nous réinitialisons le chien de garde avant Envoi en cours). Nous abandonnons après N tentatives, en supposant que le réseau est en panne.

Côté serveur, nous créons un chien de garde avant d'envoyer la réponse. Comme auparavant, nous renvoyons la réponse si le chien de garde expire. Un autre cas sur le côté serveur, c'est lorsque nous recevons une requête avec un identifiant RPC connu (requêtes sont enregistrés). Cela signifie que nous avons déjà reçu cette demande et la procédure a été appelé et la réponse envoyée, mais la réponse a été perdue. Ensuite, nous renvoyons le réponse. Comme précédemment, on abandonne après N tentatives.

Enfin, côté client, si nous recevons une réponse avec un identifiant RPC connu (les réponses sont enregistrées), c'est-à-dire une réponse que nous avons déjà reçue, cela signifie que le l'accusé de réception envoyé au serveur a été perdu et nous le renvoyons.

Page 11

RPC
Problèmes

- Traitement des pannes
  - ➢ Réseau ou serveur congestion
    - La réponse arrive trop tard (systèmes critiques)
  - ➢ Le client plante pendant le traitement des demandes sur le serveur
  - ➢ Le serveur plante pendant la manipulation du demande
  - ➢ Échec du système de communication
  - ➢ Quelles garanties?
- Problèmes de sécurité
  - ➢ Authentification du client
  - ➢ Authentification du serveur
  - ➢ Confidentialité des échanges
- Performance
- Désignation
- Aspects pratiques
  - ➢ Adaptation à conditions d'hétérogénéité (protocoles, langues, Matériel)

11

De nombreux autres problèmes peuvent être traités par les systèmes RPC.

La gestion des pannes couvre de nombreux types de pannes:

- gérer la congestion du réseau ou du serveur. Les messages peuvent être réémis, mais les messages redondants doivent être gérés. Dans un système en temps réel, l'exécution l'heure d'une procédure est spécifiée et la procédure doit renvoyer une erreur si le le délai n'est pas respecté.

- gérer le crash du client ou du serveur lors de la gestion du demande, ou la défaillance du système de communication. Le système doit fournir des garanties (par exemple, comportement transactionnel).

Un outil RPC peut également intégrer des fonctionnalités de sécurité, telles que l'authentification et cryptage des échanges.

De nombreux autres aspects ont également été pris en compte:

- les performances du RPC, notamment l'optimisation lorsque le client et le serveur les processus sont sur la même machine ou sur le même LAN.

- désignation: un schéma de désignation différent peut être fourni, pour identifier la cible (processus) de l'appel.

- hétérogénéité: beaucoup de travail a été fait pour permettre l'hétérogénéité (de langues, OS…) entre l'appelant et l'appelé (voir CORBA).

Page 12

RPC
IDL: spécification d'interface

- Utilisation d'un langage de description d'interface (IDL)
  - ➢ Spécification commune au client et au serveur
  - ➢ Définition des types et natures de paramètres (IN, OUT, IN-OUT)
- Utilisation de la description IDL pour générer:
  - ➢ Le stub client (également appelé proxy ou stub)
  - ➢ Le stub du serveur (également appelé squelette)

12

Généralement, un outil RPC génère des stubs à partir de la spécification de l'interface de la procédure qui peut être appelée à distance.

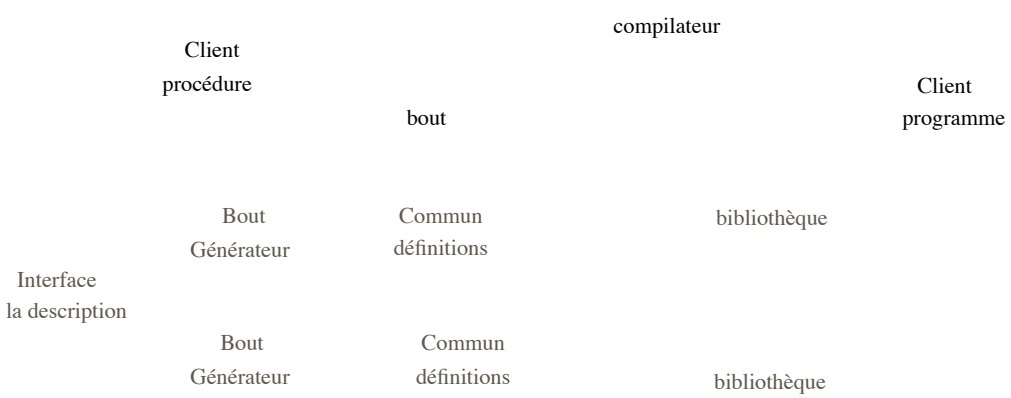
Un IDL (Interface Description Language) est un langage simple pour décrire l'interface d'une procédure qui peut être appelée via un système RPC. Il

permet simplement de décrire la signature de la procédure, y compris le type de les paramètres (structures de données).

Une telle spécification permet de générer le stub client (parfois appelé proxy ou simplement stub) et le stub du serveur (souvent appelé squelette).

Page 13

RPC
Mode fonctionnel (rpcgen)



Serveur procédure squelette compilateur Serveur programme

13

rpcgen est l'un des premiers outils RPC disponible sous Unix / C environnement.

A partir de la description de l'interface (exprimée avec l'IDL), un générateur de stub génère à la fois le sous et le squelette.

Côté client, la procédure client (appelant) est compilée avec le stub dans afin d'obtenir un exécutable binaire (programme client).

Côté serveur, la procédure serveur (appelé) est compilée avec le squelette afin d'obtenir un exécutable binaire (programme serveur).

Ces 2 binaires peuvent être installés sur différentes machines et exécutés.

Page 14

Appel de méthode à distance Java RMI

- Un RPC basé sur des objets intégré à Java: interaction entre des objets situés dans différents espaces d'adressage (*machines virtuelles Java* - JVM) sur machines distantes
- Facile à utiliser: un objet distant est appelé comme s'il était local

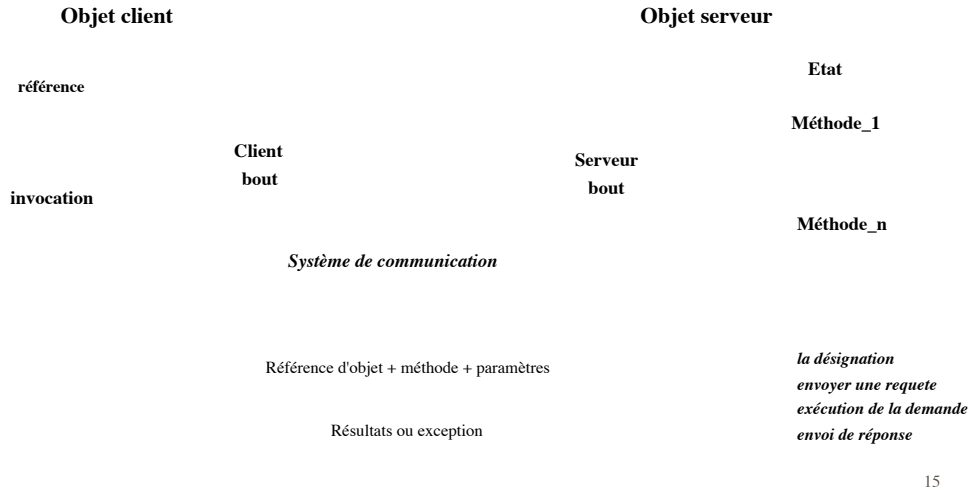
14

Java RMI (Remote Method Invocation) est un exemple d'implémentation d'un Outil RPC intégré dans un environnement de langage (ici Java).

Il permet l'invocation de méthodes sur des instances situées sur des machines distantes (dans une machine virtuelle Java distante). Un tel appel de méthode à distance est programmé comme si le l'objet cible était local sur la machine virtuelle Java actuelle.

Page 15

Java RMI
Principe



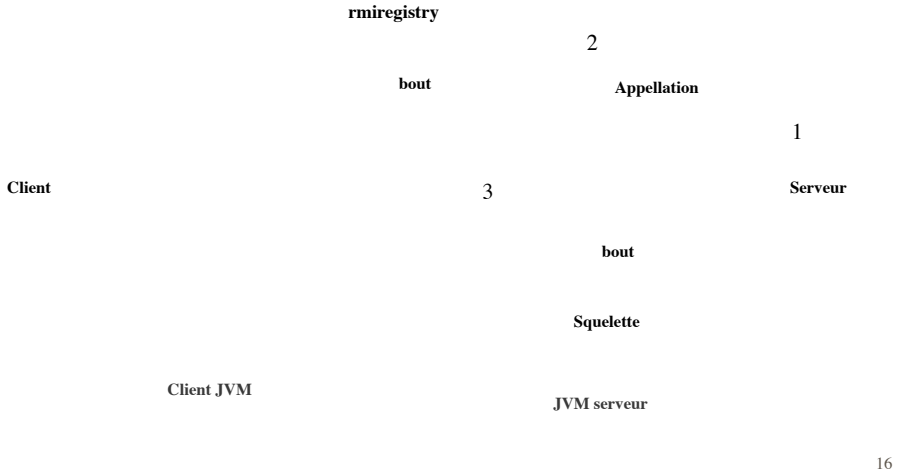
Le principe général de Java RMI est illustré dans cette figure.

Un objet client dans une machine virtuelle Java (à gauche) comprend une référence à un objet serveur (distant) dans une autre JVM (à droite).

Cette référence est en fait une référence à un objet stub local (stub client). Ce stub transforme un appel de méthode en un message de requête (qui comprend un référence d'objet pour identifier l'objet dans le serveur JVM, un identifiant de méthode et les paramètres de l'appel). Ce message de demande est reçu par un squelette objet (stub de serveur) qui exécute l'appel de méthode réel sur le serveur objet.

Page 16

Java RMI
Du côté serveur



Nous décrivons le fonctionnement général du RMI avant de décrire son modèle de programmation.

Nous supposons qu'une classe Server a été programmée suivant le RMI modèle de programmation.

Côté serveur, lorsque la classe Server est instanciée, stub et squelette les objets sont instanciés L'objet squelette est associé à un port local de la machine pour recevoir les demandes.

Afin de rendre l'objet Serveur accessible depuis les clients, il doit être enregistré dans un service de dénomination appelé rmiregistry (le rmiregistry s'exécute dans une autre JVM). Cette inscription est possible grâce à la classe Naming qui fournit une méthode de liaison (qui enregistre l'association entre un nom ("toto") et l'objet Serveur).

Cette inscription dans la rmiregistry fait une copie du stub dans la rmiregistry

(et enregistre son association avec "foo"). La rmiregistry est prête à livrer copies du talon aux clients.

Page 17

Java RMI
Du côté serveur

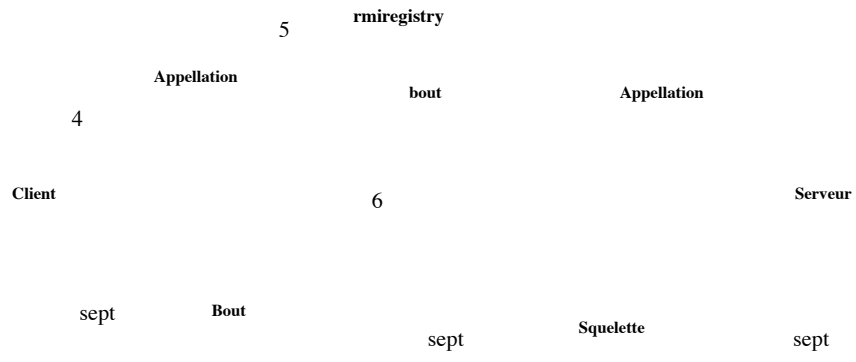
- 0 - Au moment de la création de l'objet, un *stub* et un *squelette* (avec un port de communication) sont créés sur le serveur
- 1 - Le serveur enregistre son instance auprès d'un service de dénomination (*rmiregistry*) en utilisant le *Naming* class (méthode de *liaison*)
- 2 - Le service de dénomination (*rmiregistry*) enregistre les *bout*
- 3 - Le service de dénomination est prêt à donner le *stub* aux clients

17

Nous rappelons ici l'étape principale de création d'un objet Serveur et d'inscription dans le rmiregistry.

Page 18

Java RMI
Côté client



18

Côté client, le client peut récupérer une référence à l'objet Serveur à partir de la rmiregistry. Ceci est possible grâce à la classe Naming qui fournit un méthode de recherche (qui interroge l'objet enregistré avec un nom ("foo")).

La requête sur la rmiregistry renvoie une copie du stub (associé à "toto"). Ce stub implémente la même interface que l'objet Serveur. Ça peut être utilisé par le client pour appeler une méthode. Le stub crée et envoie une demande message au squelette qui effectue l'appel réel sur l'objet Serveur.

Page 19

Java RMI
Côté client

- 4 - Le client appelle le service de dénomination (*rmiregistry*) en utilisant la classe *Naming* pour obtenir un copie du stub de l'objet serveur (*recherche* méthode)
- 5 - Le service de dénomination fournit une copie du *stub*
- 6 - Le *stub* est installé dans le client et son Java la référence est renvoyée au client
- 7 - Le client exécute un appel à distance par appeler une méthode sur le *stub*

19

Nous rappelons ici l'étape principale d'interrogation de la rmiregistry et d'effectuer une invocation de méthode.

Page 20

Java RMI
Utilisation

- Codage
  - ➢ Ecriture de l'interface serveur
  - ➢ Ecriture de la classe serveur qui implémente l'interface
  - ➢ Ecriture du client qui appelle le serveur distant objet
- Compilation
  - ➢ Compilation des sources Java (javac)
  - ➢ Génération de *stubs* et *squelettes* (rmic)
    - *(plus nécessaire, génération dynamique)*
- Exécution
  - ➢ Lancement du service de nommage (*rmiregistry*)
  - ➢ Lancement du serveur
  - ➢ Lancement du client

20

Voici les principales étapes d'utilisation de RMI.

Concernant le codage:

- vous devez définir l'interface Java du serveur. Cette interface est utilisée à la fois par le serveur et le client.

- la classe Server implémente l'interface précédente. Le serveur est instancié et l'instance est enregistrée dans la rmiregistry.

- le Client peut déclarer une variable dont le type est l'interface précédente. le Le client obtient une copie du stub du rmiregistry. Le stub implémente L'interface. Le client peut appeler une méthode sur ce stub.

Concernant la compilation:

- l'application est compilée avec javac comme d'habitude

- les classes stub et squelette peuvent être générées avec rmic (un générateur de stub). Ce n'est plus nécessaire sur les versions récentes de Java, les stubs étant généré dynamiquement en cas de besoin.

Concernant l'exécution:

- vous devez lancer la rmiregistry

- alors vous pouvez lancer le serveur puis le client

Page 21

Java RMI
Programmation

- Programmation d'une interface distante
  - ➢ interface publique
  - ➢ interface: étend java.rmi.Remote
  - ➢ méthodes: lance java.rmi.RemoteException
  - ➢ Paramètres sérialisables: implémente Serializable
  - ➢ paramètres de référence: implémente Remote
- Programmation d'une classe distante
  - ➢ implémente l'interface précédente
  - ➢ étend java.rmi.server.UnicastRemoteObject
  - ➢ mêmes règles pour les méthodes

21

La programmation des applications RMI s'accompagne de contraintes de programmation.

Pour l'interface du serveur:

- l'interface doit être publique

- l'interface doit implémenter l'interface distante

- toutes les méthodes doivent lancer RemoteException

- les paramètres des méthodes distantes peuvent être de type intégré (int, char) ou Java référence. Dans ce dernier cas, leur type doit être une interface qui est soit Sérialisable ou à distance (ceci est détaillé plus tard).

Pour la classe Serveur:

- il doit implémenter l'interface précédente

- il doit étendre la classe UnicastRemoteObject

- mêmes règles pour les méthodes (comme dans l'interface précédente)

Page 22

Java RMI
Exemple: interface

fichier Hello.java

```
interface publique Hello étend java.rmi.Remote {
   public void sayHello ()
            lance java.rmi.RemoteException;
}
```

La description du interface

22

Nous passons en revue un exemple très simple.

Voici la définition de l'interface.

L'interface Hello implémente Remote et lance RemoteException.

Page 23

Java RMI
Exemple: serveur

fichier HelloImpl.java

```
import java.rmi. *;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl étend UnicastRemoteObject
                                        implémente Hello {
    Message de chaîne;

   // Implémentation du constructeur
   public HelloImpl (String msg) lève java.rmi.RemoteException {
            message = msg;
   }
  // Implémentation de la méthode distante
   public void sayHello () lance java.rmi.RemoteException {
             System.out.println (message);
   }
```

la mise en oeuvre
du
classe de serveur

...

23

Voici le code de la classe serveur.

La classe HelloImpl étend UnicastRemoteObject et implémente l'interface Bonjour.

Vos constructeurs doivent lancer RemoteException.

La méthode distante sayHello () lève RemoteException.

Page 24

Java RMI
Exemple: serveur

fichier HelloImpl.java

la mise en oeuvre du classe de serveur

```
...
public static void main (String args []) {
        essayez {
                // Créer une instance de l'objet serveur
                Hello obj = new HelloImpl ("bonjour");
                // Enregistrer l'objet auprès du service de nommage
                Naming.rebind ("// ma_machine / mon_serveur", obj);
                System.out.println ("HelloImpl" + "lié dans le registre");
        } catch (Exception exc) {…}
    }
}
```

AVIS: dans cet exemple, le service de nommage (rmiregistry) doit avoir été lancé avant l'exécution du serveur

24

Le reste du code du serveur.

La méthode main crée une instance de la classe serveur (HelloImpl) et l'enregistre dans la rmiregistry, grâce à la classe Naming.

L'URL passée dans la méthode rebind () est // <machine-name>: <port> / <name>

- nom-machine est le nom de la machine qui exécute la rmiregistry

- port est le port utilisé par la rmiregistry (le port par défaut est 1099)

- nom est le nom identifiant l'objet enregistré dans la rmiregistry

Dans son implémentation en Java, la rmiregistry doit être colocalisée (sur le même machine) avec la JVM qui exécute l'objet serveur. Une solution consiste à implémenter une autre rmiregistry (permettant les enregistrements à distance).

Notez qu'après l'enregistrement, c'est la fin de la méthode principale et le JVM quitterait. Ce n'est pas le cas, car lorsque nous avons instancié le serveur objet, un squelette a été instancié avec création d'une prise de communication et d'un thread en attente de requêtes entrantes. À cause de ce fil, le JVM ne se ferme pas.

Ici, nous supposons que la rmiregistry a été lancée (rmiregistry est une exécutable) sur la même machine que l'objet serveur, avec la commande:

rmiregistry <port> (la valeur par défaut est 1099)

Page 25

Java RMI

exécution de rmiregistry dans la machine virtuelle Java du serveur

fichier HelloImpl.java

```
public static void main (String args []) {
    int port; URL de chaîne;

    essayez {
      Integer I = new Integer (args [0]); port = I.intValue ();
    } catch (Exception ex) {
      System.out.println ("Veuillez saisir: java HelloImpl <port>"); revenir;
    }

    essayez {
       // Lancement du service de nommage - rmiregistry - dans la JVM
       Registre de registre = LocateRegistry.createRegistry (port);

       // Créer une instance de l'objet serveur
       Bonjour obj = new HelloImpl ();

       // calcule l'URL du serveur
       URL = "//"+InetAddress.getLocalHost().getHostName()+":"+
                                          port + "/ mon_serveur";
       Naming.rebind (URL, obj);
    } catch (Exception exc) {...}
}
```

25

Dans cette autre version, nous lançons une rmiregistry dans la même JVM que celle l'hébergement de l'objet serveur.

La méthode createRegistry lance une rmiregistry dans la JVM locale sur le port spécifié.

L'intérêt de le faire est que lorsque vous démarrez l'application, une rmiregistry est automatiquement lancé et lorsque vous tuez la JVM, la rmiregistry est également tué. Ceci est très pratique lors du débogage.

Piste 26

Java RMI
Exemple: client

fichier HelloClient.java

```
import java.rmi. *;

public class HelloClient {
  public static void main (String args []) {
    essayez {
      // récupère le stub de l'objet serveur de la rmiregistry
      Bonjour obj = (Bonjour) Naming.lookup ("// ma_machine / mon_serveur");
      // Invocation d'une méthode sur l'objet distant
      obj.sayHello ();
    } catch (Exception exc) {…}
  }
}
```

la mise en oeuvre du classe de client

26

Voici le code côté client.

Il demande d'abord une référence à l'objet cible de la rmiregistry, en utilisant le lookup de la classe Naming (l'URL utilisée est la même qu'avant.

Notez qu'ici le client peut s'exécuter sur une machine différente.

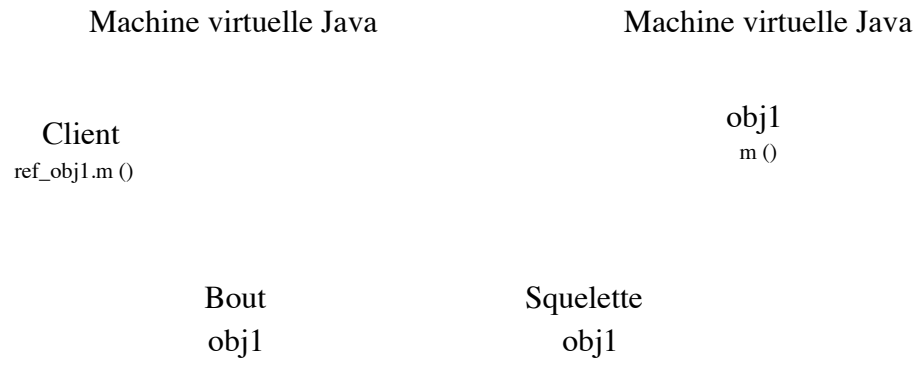
La rmiregistry renvoie une instance de stub. Cette instance de stub implémente le même interface que l'objet serveur (ici Hello). Ainsi nous pouvons lancer le obtenu référence avec l'interface Hello.

Ensuite, l'appel d'une méthode sur l'objet distant est programmé comme si l'objet était local.

Page 27

Java RMI

Principe de l'invocation de méthode à distance



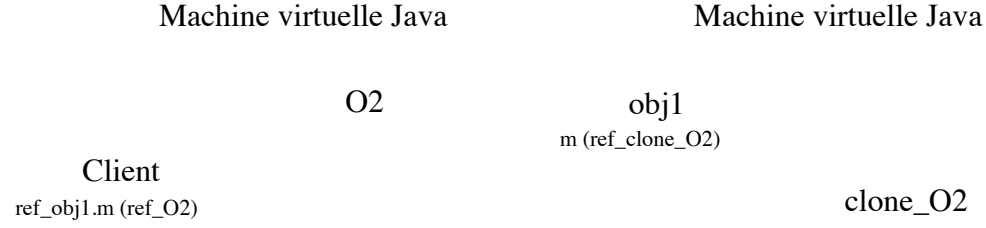
Pour résumer le fonctionnement de Java RMI, un client qui a obtenu (du rmiregistry) une référence distante (ref_obj1) à un objet distant (obj1) a en fait une référence à un objet stub local (Stub obj1). Le client peut invoquer un méthode m () sur l'objet distant. Il invoquera cette méthode sur le stub,

qui enverra le message de demande. Ce message est reçu par le skeleton (Skeleton obj1) qui effectue l'appel réel sur le serveur objet.

Page 28

Java RMI

Passage de paramètre d'objet sérialisable



Bout
obj1

Squelette
obj1

28

Les paramètres passés dans une méthode distante peuvent être de types intégrés (int char…). Ensuite, les paramètres sont simplement copiés (transférés) dans le serveur distant.

Si un paramètre est une référence Java à un objet, le type du paramètre dans le la signature de méthode doit être une interface. Ensuite, il y a 2 possibilités:

- Sérialisable. Si l'interface est sérialisable (hérite de Serializable), alors l'objet passé est copié sur le serveur (l'objet est cloné).

- À distance. Si l'interface est Remote (hérite de Remote), alors la télécommande la référence (c'est-à-dire le stub) est passé au serveur, ce qui signifie que le stub est copié dans le serveur. Par conséquent, l'objet passé devient accessible à distance dans le serveur.

- si l'interface n'est ni sérialisable ni distante, c'est une erreur (elle doit pas compiler).

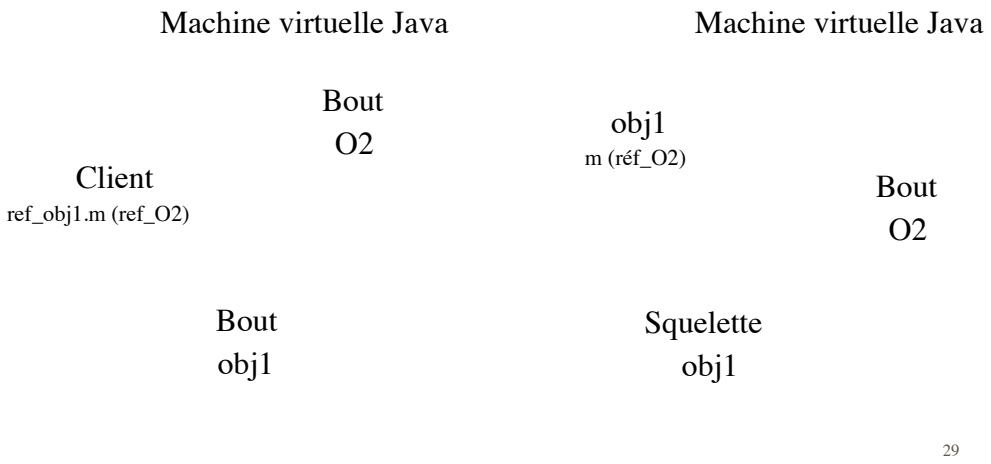
Cette figure illustre le cas sérialisable. Le client passe comme paramètre a référence à l'objet O2 qui est local dans le client. Ensuite, O2 est copié dans le serveur et la méthode invoquée (m) reçoit une référence à un clone d'objet O2 dans le serveur.

Page 29

Java RMI

Passage de paramètres d'objet distant

O2



Cette figure illustre le cas Remote. Le client passe comme paramètre a référence à l'objet O2 qui est distant (dans une autre JVM). Cela signifie que le la référence à O2 dans le client est une référence locale à un sous de O2. Ensuite, le talon d'O2 est copié sur le serveur appelé et la méthode invoquée (m) reçoit un référence à une copie du stub d'O2 dans le serveur. Par conséquent, m () reçoit un référence à distance à O2.

Piste 30

Java RMI
Compiler

- Compilation de l'interface, du serveur et du client
  - ➢ javac Hello.java HelloImpl.java HelloClient.java
- Génération de stubs (*plus nécessaire*)
  - ➢ rmic HelloImpl
    - *squelette* dans HelloImpl_Skel.class
    - *stub* dans HelloImpl_Stub.class

30

Pour exécuter l'application, vous devez d'abord compiler l'interface et le serveur et classes de clients.

Comme mentionné précédemment, la génération de stubs et de squelettes n'est pas nécessaire plus, mais vous pouvez toujours le faire.

Piste 31

Java RMI
Déploiement

- Lancement du service de dénomination
  - ➢ rmiregistry &
- lancement du serveur
  - ➢ java HelloImpl
  - ➢ java -Djava.rmi.server.codebase = http: // ma_machine /…
    - URL d'un serveur Web à partir duquel la JVM cliente pourra pour télécharger les classes manquantes
    - Exemple: sérialisation
- Lancement du client
  - ➢ java HelloClient

31

Ici, nous lançons explicitement la rmiregistry dans un shell.

Ensuite, nous pouvons lancer le serveur puis le client.

Un problème délicat est la disponibilité des classes. Supposons que le client appelle un méthode m (Data d) sur le serveur, Data étant une interface sérialisable.
Le client et le serveur connaissent les données d'interface (il était nécessaire d'utiliser la méthode m et pour compiler le code). Ensuite, le client peut invoquer m passant une instance de la classe ClientData (qui implémente Data). Mais le serveur qui reçoit une copie de l'objet n'a pas la classe ClientData (et les clients peuvent avoir différentes implémentations de l'interface de données).

Plus généralement, une JVM peut transférer des copies d'objets (avec sérialisation) vers autres JVM. Comment la première JVM peut-elle mettre ces classes à la disposition d'autres JVM?

La solution est de spécifier, lors du lancement d'une JVM, un site web à partir duquelles classes peuvent être téléchargées. Lorsque des classes sont manquantes pour l'utilisation d'un objet sérialisé, le les classes sont téléchargées et installées dynamiquement.

java -Djava.rmi.server.codebase = URL <a class>

Lors du lancement d'une JVM de cette façon, nous spécifions que si des instances sérialisées sont données à d'autres JVM, les classes manquantes peuvent être trouvées sur le site web défini par URL.

Piste 32

Java RMI: conclusion

- Très bon exemple de RPC
  - ➢ Facile à utiliser
  - ➢ Bien intégré à Java
  - ➢ Passage des paramètres de référence Java: sérialisation ou référence à distance
  - ➢ Déploiement: chargement dynamique de classes sérialisables
  - ➢ Désignation avec URL

De nombreux tutoriels sur la programmation RMI sur le Web…

Exemple: https://www.tutorialspoint.com/java_rmi/java_rmi_application.htm

32

Pour conclure cette conférence, Java RMI est un exemple de RPC intégré dans un Java.

De nombreux tutoriels sur Java RMI peuvent être trouvés sur le net.