USTL – Master mention Informatique – M1

Construction d'applications réparties

V. Java RMI

Lionel.Seinturier@lifl.fr

Plan

1. Caractéristiques

Java RMI

Solution Sun pour l'invocation à distance de méthodes Java

- inclus par défaut dans le JDK depuis 1.1
- nouveau modèle de souches dans JDK 1.2
- génération dynamique des souches dans JDK 1.5

- implantations alternatives (*open-source*)
  - NinjaRMI (Berkeley)
  - Jeremie (ObjectWeb)

- package `java.rmi`
- + outils
  - générateur de souches
  - serveur de noms
  - démon d'activation

1. Caractéristiques

Java RMI

Un mécanisme de RPC dédié aux objets Java

- *stubs* côté client
- *skeletons* côté serveur

⇨ encodages/désencodages données + communications rsx

• types simples (`int`, `float`, ...)	transmission **par copie**
• objet local	transmission **par copie** (sérialisation)
• objet RMI distant	transmission **par référence**

2. Modèle de programmation

Principes

Chaque classe d'objets serveur doit être associée à une interface

⇨ seules les méthodes de l'interface pourront être invoquées à distance

1. Ecriture d'une interface
2. Ecriture d'une classe implantant l'interface
3. Ecriture du programme serveur
4. Ecriture du programme client

≡

1. déclaration des services accessibles à distance
2. définition du code des services
3. instanciation et enregistrement de l'objet serveur
4. interactions avec le serveur

2. Modèle de programmation

Ecriture d'une interface

- interface Java normale
- doit étendre `java.rmi.Remote`
- toutes les méthodes doivent lever `java.rmi.RemoteException`

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface CompteInterf extends Remote {
    public String getTitulaire() throws RemoteException;
    public float solde() throws RemoteException;
    public void deposer( float montant ) throws RemoteException;
    public void retirer( float montant ) throws RemoteException;
    public List historique() throws RemoteException;
}
```


2. Modèle de programmation

Ecriture d'une classe implantant l'interface

- classe Java normale implantant l'interface
- doit étendre `java.rmi.server.UnicastRemoteObject`
- constructeurs doivent lever `java.rmi.RemoteException`
- si pas de constructeur, en déclarer un vide qui lève `java.rmi.RemoteException`

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

public class CompteImpl extends UnicastRemoteObject
                        implements CompteInterf {
    private String nom;
    private float solde;

    public CompteImpl(String nom) throws RemoteException {
        super();
        this.nom = nom;
    }
    public String getTitulaire() { return nom; }
    ... }
```


2. Modèle de programmation

Ecriture d'une classe implantant l'interface

1. Compilation de l'interface et de la classe avec `javac`
2. Génération des souches clientes et serveurs
   à partir du bytecode de la classe avec `rmic`

```
javac CompteInterf.java CompteImpl.java
rmic CompteImpl
```

Quelques options utiles de rmic

-d path	répertoire pour les fichiers générés
-keep	conserve le code source des souches générées
-v1.1	souches version JDK 1.1
-v1.2	souches version JDK 1.2
-vcompat	**par défaut**, souches pour JDK 1.1 et 1.2

2. Modèle de programmation

Ecriture d'une classe implantant l'interface

Fichiers générés pour chaque classe d'objet serveur RMI

```
rmic CompteImpl
```

`CompteImpl_Stub.java` : souche cliente
`CompteImpl_Skel.java` : souche serveur

2. Modèle de programmation

Ecriture du programme serveur

1. Instanciation de la classe serveur
2. Enregistrement de l'instance dans le serveur de noms RMI

```
public class Serveur {
    public static void main(String[] args) throws Exception {
        CompteInterf compte = new CompteImpl("Bob");
        Naming.bind("Bob",compte);
    }
}
```

- `compte` prêt à recevoir des invocations de méthodes
- programme "tourne" en permanence
  tant que `compte` reste enregistré dans le runtime RMI
  (attention : ≠ du service de nommage)
- désenregistrement : `UnicastRemoteObject.unexportObject(compte,false)`
  `false` : attente fin les requêtes en cours / `true` : immédiat

2. Modèle de programmation

Ecriture du programme client

1. Recherche de l'instance dans le serveur de noms
2. Invocation des méthodes

```
public class Client {
    public static void main(String[] args) throws Exception {
        CompteInterf compte = (CompteInterf) Naming.lookup("Bob");
        compte.deposer(10);
        System.out.println( compte.solde() );
    }
}
```


2. Modèle de programmation

Exécution des programmes

1. Lancer le serveurs de noms (`rmiregistry`)
   - une seule fois
   - doit avoir accès au *bytecode* de la souche cliente (→ CLASSPATH)

2. Lancer le programme serveur
3. Lancer le programme client

2. Modèle de programmation

Compléments

Passage de paramètres avec RMI

- types de base (`int`, `float`, ...) par copie
- objets implantant `java.io.Serializable` passés par copie (sérialisés)
- objets implantant `java.rmi.Remote` passés par référence
  → la référence RMI de l'objet est transmise

- dans les autres cas une exception `java.rmi.MarshalException` est levée

2. Modèle de programmation

Compléments

Ecriture d'une classe d'objet serveurs

- héritage `UnicastRemoteObject`
- pas toujours possible (si classe ∈ à une hiérarchie d'héritage)

→ méthode
`static UnicastRemoteObject.exportObject(Remote)`
enregistrement côté serveur

2. Modèle de programmation

Compléments

Objets serveur prévus pour fonctionner avec ≠ types de liaisons

1. objet distant joignable en point à point
2. objets distants répliqués
3. objets distants joignables par diffusion

En pratique seul 1. mis en oeuvre (classe `UnicastRemoteObject`)

Références d'objets distants RMI avec `UnicastRemoteObject`

- adresse IP
- n° port TCP
- identifiant local d'objet (entier)

2. Modèle de programmation

Compléments

Invocations concurrentes de méthodes RMI

Un objet serveur RMI est susceptible d'être accédé
par plusieurs clients simultanément

⇨ toujours concevoir des méthodes RMI *thread-safe*
i.e. exécutable concurrement de façon cohérente

⇨ la création de *thread* est faite automatiquement par le *runtime* RMI

2. Modèle de programmation

Compléments

Serveurs de noms

Problématique : publier la référence du serveur pour que les clients y accède
⇨ machine "connue de tous" sur le réseau (idem DNS)

Facilité, mais pas obligation

⇨ il "suffit" que le client connaisse le point d'accès
⇨ la souche cliente est sérialisable
⇨ sauvegarde dans un fichier
⇨ lecture par le client

Récupération de la souche cliente côté serveur

`java.rmi.server.RemoteObject.toStub(Remote)`

3. Services RMI

Services RMI

- service de nommage
- service d'activation d'objets
- ramasse-miettes réparti

3. Services RMI

Service de nommage

Permet d'enregistrer des liaisons
entre un objet serveur et un nom symbolique

- par défaut port 1099 (autre port : rmiregistry 12345)
- noms "plats" (pas de noms composés, pas de hiérarchies)

URL RMI `rmi://machine.com:1099/nomSymbolique`

`rmi://` et `:1099` facultatifs
`machine.com` par défaut `localhost`

Serveur de noms demarrable

- de façon autonome dans un *shell* avec l'outil `rmiregistry`
- dans un programme par appel de la méthode `static`
  `java.rmi.registry.LocateRegistry.createRegistry(int port)`

3. Services RMI

Service de nommage

Classe `java.rmi.Naming` pour l'accès local

Toutes les méthodes sont `static`

`1. void bind(String,Remote)`	enregistrement d'un objet
`2. void rebind(String,Remote)`	ré enregistrement d'un objet
`3. void unbind(String)`	désenregistrement d'un objet
`4. String[] list(String)`	liste des noms d'objets enregistrés
`5. Remote lookup(String)`	recherche d'un objet

Les paramètres `String` correspondent à une URL d'objet RMI
1 2 3 accessibles localement uniquement
1 lève une exception si le nom est déjà enregistré
4 URL du `rmiregistry`

3. Services RMI

Service de nommage pour l'accès distant

Classe `java.rmi.registry.LocateRegistry`
`static Registry getRegistry( String host, int port )`

`java.rmi.registry.Registry` : même méthodes que `Naming`

3. Services RMI

Service d'activation d'objets

Objets serveurs activables à la demande (depuis JDK 1.2)
en fonction demande clients

- démon `rmid`
- package `java.rmi.activation`

Avantages

- évite d'avoir des objets serveurs actifs en permanence
  (ie "tournant" dans une JVM)
  → trop coûteux si beaucoup d'objets serveurs
- permet d'avoir des références d'objets persistantes
  → en cas de *crash* d'objet serveur
  le démon peut le relancer avec la même référence
  → les clients continuent à utiliser la même référence

3. Services RMI

Service d'activation d'objets

```
public class Serveur {

    public static void main(String[] args) throws Exception {

ActivationGroupDesc gDesc = new ActivationGroupDesc(null,null);
ActivationGroupID gID =
       ActivationGroup.getSystem().registerGroup(gDesc);
ActivationGroup.createGroup(gID,gDesc,0);

MarshalledObject mo = new MarshalledObject("Bob");
ActivationDesc desc = new ActivationDesc("CompteImpl","",mo);
CompteInterf compte = (CompteInterf) Activatable.register(desc);

Naming.bind("Bob",compte);

    }
}
```

Programme client inchangé
Lancer `rmiregistry` et **démon d'activation** (`rmid`)

3. Services RMI

Extension du principe : *pool* d'objets

Avoir un ensemble d'objets prêts à traiter les requêtes

Même problématique que le *pool* de *threads*

⇨ gestion de la taille du *pool* (fixe, variable)
⇨ à programmer

3. Services RMI

Ramasse-miettes réparti

Récupérer les ressources (mémoire, ...)
occupées par un objet que personne ne référence
→ i.e. que l'on ne pourra plus jamais accéder

Difficulté : environnement distribué donc référencement à distance

Dans une JVM : mécanisme *mark-and-sweep*

1. parcours du graphe de référencement des objets
2. destruction des objets non visités

Avec RMI : double mécanisme géré par le Remote Reference Manager

- comptage de référence : # de clients référençant l'objet
- bail (*lease*) : mémoire "louée" à l'objet pour un temps fini

3. Services RMI

Ramasse-miettes réparti

Comptage

- chaque transmission de référence +1
- chaque fin de référencement -1

Bail

- par défaut 10 min (réglable par la propriété `java.rmi.dgc.leaseValue`)
- but : se prémunir
  - partitions de réseaux
  - pertes de message de déréférencement

Si le compteur tombe à 0 ou si le bail expire,
l'objet devient candidat pour le ramassage local (*mark-and-sweep*)

3. Services RMI

Ramasse-miettes réparti

Attention : un objet serveur "normal"
instancié par un programme serveur qui "tourne" en permanence
est toujours référencé par ce programme

⇨ il n'est pas ramassé (même au delà des 10 min)

⇨ le ramassage concerne des objets créés dont on "perd" la référence

Client

```
BarRemote b = foo();
...
b = null;
```

Serveur

```
BarRemote foo() {
    return new BarRemote();
}
```


3. Services RMI

Ramasse-miettes réparti

Interface de notification du ramassage réparti

```
interface Unreferenced {
       void unreferenced();
}
```

4. Fonctionnalités additionnelles

- RMI/IIOP
- Chargement dynamique de classes (`RMIClassLoader`)
- RMI et les *firewalls*
- personnalisation des communications

4. Fonctionnalités additionnelles

RMI/IIOP

2 possibilités pour acheminer les requêtes IIOP

- JRMP : protocole Sun (`UnicastRemoteObject`) utilisé par défaut
- IIOP : protocole OMG pour CORBA

RMI avec IIOP

⇨ permet intéropérabilité RMI - CORBA

4. Fonctionnalités additionnelles

RMI/IIOP

Modèle de programmation

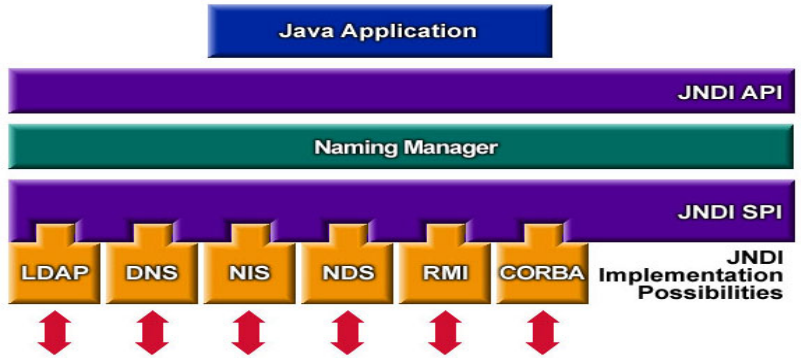
- les classes doivent étendre `javax.rmi.PortableRemoteObject`
- les souches doivent être générées avec `rmic -iiop`
  2 fichiers générés pour chaque classe d'objet serveur
  - `_nomClasse_Stub.java` : souche cliente
  - `_nomClasse_Tie.java` : souche serveur
- service de nommage JNDI (Java Naming and Directory Interface)
  - outil `tnameserv`
- les conversions de type sur les objets distants doivent utiliser la méthode `static PortableRemoteObject.narrow()`

4. Fonctionnalités additionnelles

RMI/IIOP

JNDI (Java Naming and Directory Interface)

- API
- unifie l'accès à ≠ serveurs de noms



4. Fonctionnalités additionnelles

RMI/IIOP

Exemple de code

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class CompteImpl extends PortableRemoteObject
                        implements CompteInterf {
...
}

public class Serveur {
  public static void main(String[] args) throws Exception {
    CompteInterf compte = new CompteImpl("Bob");
    Context ic = new InitialContext();
    ic.rebind( "Bob", compte );
```


4. Fonctionnalités additionnelles

RMI/IIOP

Cas où le serveur de noms n'est pas local

```
Hashtable env = new Hashtable();
env.put( "java.naming.factory.initial",
         "com.sun.jndi.cosnaming.CNCtxFactory" );
env.put( "java.naming.provider.url",
         "iiop://localhost:1704" );

Context ic = new InitialContext(env);
```


4. Fonctionnalités additionnelles

RMI/IIOP

Restrictions liées à RMI/IIOP

- les constantes définies dans les interfaces `Remote` doivent être de type primitif ou de type String et être évaluables à la compilation
- pas de surcharge pour les méthodes des interfaces `Remote` i.e. pas 2 méthodes avec le même nom et des profils ≠
- pas de mécanisme de ramasse-miettes réparti
- les fonctionnalités fournies par les interfaces suivantes ne sont pas utilisables `UnicastRemoteObject`, `RMISocketFactory`, `Unreferenced`, `java.rmi.dgc.*`

4. Fonctionnalités additionnelles

Chargement dynamique de classes

- Java charge les `.class` à la demande à partir du disque (CLASSPATH)
- RMI introduit en + un mécanisme de chargement des classes à distance par HTTP ou FTP

Avantage : classes déployées sur 1 seul site (+ rapide + simple à gérer)
Inconvénient : *single point of failure*

Utilisation

- propriété `java.server.rmi.codebase` : URL du serveur de téléchargement
- classe `RMISecurityManager`

4. Fonctionnalités additionnelles

RMI et les *firewalls*

But : utiliser RMI en passant au travers de *firewalls*

Pb : RMI ouvre des cx TCP directes sur des ports quelconques

Solution : encapsuler les requêtes RMI dans des requêtes HTTP POST
→ *tunneling* HTTP

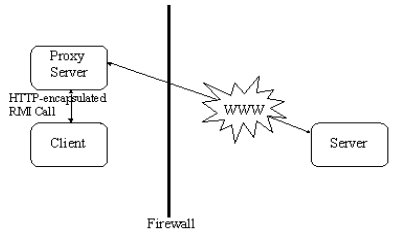
4. Fonctionnalités additionnelles

RMI et les *firewalls*

Fonctionnement

Le client essaie d'établir une cx directe → échec

Cas où le client est derrière un *firewall*



Le client essaie de passer par un proxy web local et
de le faire contacter le serveur à l'@ `http://hostname:port`
- on suppose que le *firewall* bloque le traffic RMI mais laisse passer HTTP
- le client doit positionner la propriété `http.proxyHost`

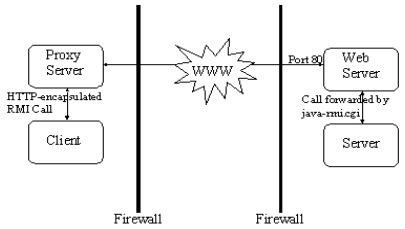
4. Fonctionnalités additionnelles

RMI et les *firewalls*

Fonctionnement

Si de nouveau échec

Cas où le client est derrière un *firewall*



1 niveau supplémentaire d'indirection
hyp : le serveur dispose d'un prog. CGI qui redirige les req. vers l'obj. serv. RMI
le client essaie de contacter le serveur à l'@
`http://hostname:80/cgi-bin/java-rmi?forward=<port>`

4. Fonctionnalités additionnelles

RMI et les *firewalls*

Conclusion

- ça marche
- mais perte de performances
- mais aléas de sécurité éventuels

⇨ à n'utiliser que lorsqu'on ne peut vraiment pas faire autrement

4. Fonctionnalités additionnelles

Personnalisation des communications

RMI utilise des *sockets* TCP

- côté client et serveur
- attribuées automatiquement par défaut

Possibilité de personnaliser ces sockets pour

- forcer l'utilisation de *sockets* précises
- tracer les communications
- crypter et/ou signer les données
- introduire des traitements "à l'insu" des objets clients/serveurs RMI

⇨ redéfinir le constructeur

```
protected UnicastRemoteObject( int port,
     RMIClientSocketFactory csf, RMIServerSocketFactory ssf )
```

- ou utiliser `static UnicastRemoteObject.export(Remote,int,..,..)`

4. Fonctionnalités additionnelles

Personnalisation des communications

```
interface RMIClientSocketFactory {
    java.net.Socket createSocket( String host, int port );
}
interface RMIServerSocketFactory {
    java.net.ServerSocket createServerSocket( int port );
}
```

⇨ déf. 2 classes qui implantent ces interfaces
⇨ déf. des sous-classes de `Socket` et `ServerSocket`
pour personnaliser le fonctionnement des sockets
ex : `javax.net.ssl.SSLSocket`, `javax.net.ssl.SSLServerSocket`

5. Protocole

Protocole JRMP

Protocole de transport des invocations de méthodes distantes pour Java RMI

Messages sortants

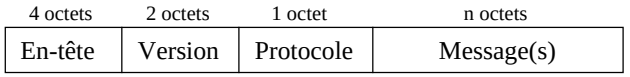
- *Call* : véhicule une invoc. de méth. (+ *callData*)
- *Ping* : teste le bon fonctionnement d'un serveur
- *DcgAck* : utilisé par l'algo. de ramasse-miettes pour signaler que les obj. dist. retournés par le serveur ont été reçus par le client

Messages entrants

- *Return* : véhicule le retour de l'invoc. (+ *returnValue*)
- *HttpReturn* : idem mais encapsulé dans une requête HTTP (+ *returnValue*)
- *PingAck* : acquittement d'un message *Ping*

5. Protocole

Structure des paquets échangés par le protocole RMI



En-tête : *magic number* (JRMI)
Version : numéro de version du protocole

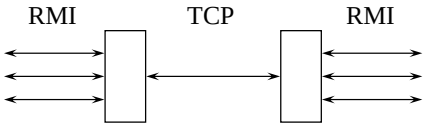
Protocole : 3 possibilités

- SingleOpProtocol : 1 seule invoc. par paquet (cas de RMI sur HTTP)
- StreamProtocol : +sieurs invoc. **vers un même obj.** les unes à la suite
- MultiplexProtocol : +sieurs invoc. **vers une même machine** multiplexées sur la même connexion

5. Protocole

Multipléxage de connexion

But : transmettre +sieurs invocations RMI sur une même connexion TCP



Protocole de multiplexage

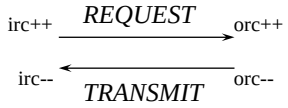
- *OPEN*, *CLOSE*, *CLOSEACK* : pour gérer une connexion RMI multiplexée
- *REQUEST*, *TRANSMIT* : pour assurer le transport de l'information

5. Protocole

Mécanisme de contrôle de flux

But : éviter qu'un buffer plein pour 1 connexion bloque toutes les autres
éviter qu'1 connexion qui ne se termine pas bloque toutes les autres
(par exemple en cas d'appels récursifs)

2 compteurs (en nombre d'octets) pour chaque cx RMI multiplexée
- *input request count* (irc)
- *output request count* (orc)



- irc et orc ne doivent jamais être négatifs
- irc ne doit pas dépasser une valeur (en nb d'octets) qui le bloquerait

Bibliographie

- **Java RMI**. W. Grosso. O'Reilly.

- **jGuru RMI Tutorial**
`http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html`

- **Guide RMI**. Sun.
`http://java.sun.com/products/jdk1.4/docs/guide/rmi/`

- **RMI-IIOP**. IBM.
`http://www.ibm.com/java/jdk/rmi-iiop/`