RMI

(JAVA) REMOTE METHOD INVOCATION

Panoramica

RMI permette di invocare metodi di oggetti (remoti) indipendentemente dalla loro posizione (purché accessibili)

Estende il concetto di RPC (Remote Procedure Call) al paradigma OO

Tuttavia:

- Il destinatario è un oggetto (e non un processo, almeno in apparenza)
- Gli argomenti sono valori o oggetti (locali o remoti)
- Il metodo può invocare metodi dei suoi argomenti (e.g. callback)

Distribuzione e invocazione sono trasparenti (per quanto è lecito attendersi)

- Irraggiungibilità dell'oggetto
- Non tutti gli argomenti sono leciti e.g. un filestream

Definire oggetti remoti

Per definire un oggetto remoto abbiamo bisogno di:

1. Un interfaccia (da fornire agli utenti/client)
2. Un sua implementazione (la parte remota da mantenere sul server)
3. (opzionalmente) politiche d'accesso all'implementazione

Al resto provvederà l'infrastruttura RMI.

Esempio: un dizionario…

Dictionary

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

public interface Dictionary <V extends Serializable> extends Remote{
     V get(String key) throws RemoteException;
     void put(String key, V value)  throws RemoteException;
}
```

- L'interfaccia «si dichiara» all'infrastruttura RMI ereditando **`java.rmi.Remote`**
- Remote è vuota, funge da «tag»
- I metodi remoti possono sollevare eccezioni legate alla comunicazione con l'istanza remota (e.g. policy, I\O, etc.).
- Queste sono raccolte sotto **`RemoteException`**

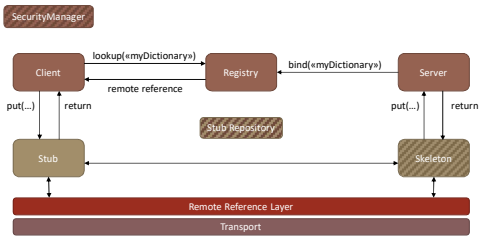
RemoteDictionary

```
import…
public class RemoteDictionary<V extends Serializable>
     extends RemoteObject implements Dictionary{
     …
     private final Map<String,V> values = new HashMap<>();
     public V get(String key) {
          return values.get(key);
     }

     public void put(String key, V value) {
          values.put(key, value);
     }
}
```

L'implementazione deve estendere `java.rmi.server.`**`RemoteObject`** (o una sua sottoclasse e.g. **UnicastRemoteObject** o Activable)

L'infrastruttura RMI



Stub e Skeleton

Stub e skeleton sono sintetizzati a partire all'implementazione (RemoteDictionary)

In alcuni casi (e.g. versioni Java meno recenti) la sintesi è fatta manualmente lanciando il compilatore RMI (direttamente sui binari) e.g.:

```
rmic RemoteDictionary
```

Lo stub traduce le invocazioni del client in comunicazioni verso il server

Lo skeleton accetta le comunicazioni dal client e le traduce in invocazioni

(Da Java 2, lo skeleton è rimpiazzato dalla reflection)

Viene garantita una semantica di invocazione «al più una volta».

Registry e binding

Il **registry** (rmiregistry) tiene traccia degli oggetti remoti disponibili
- Le istanze sono registrate dai server legandole a nomi testuali
- I client richiedono un riferimenti all'oggetto associato ad un dato nome

I riferimenti remoti restituiti dal registry contengono:
- Host
- Porta
- Object Identifier (OID)

Il riferimento è gestito dallo stub che appare al programmatore come istanza dell'interfaccia remota (e.g. Dictionary)

Gli stub possono essere inviati al client (e.g. rilasciati su appositi repository)

Le operazioni sono monitorate dai SecurityManager
- (presenti in ogni applicazione Java, ma disattivabili)

Security Manager

Istanze di `java.lang.SecurityManager` definiscono politiche di sicurezza per l'applicazione.

L'esecuzione di un'azione proibita scatena una `SecurityException` e.g.:

```
SecurityManager security = System.getSecurityManager();
security.checkAccept(String host, int port);
```

solleva un'eccezione se il thread chiamante non può accettare connessioni da `host:port`

È possibile caricare configurazioni predefinite o personalizzate:

```
System.setSecurityManager(new RMISecurityManager());
```

`AppletSecurityManager`: gli stub hanno i permessi delle applet

`RMISecurityManager`: consente solo l'accesso agli stub e connessioni RMI

`Null`: solo stub nel classpath locale

SecurityManager

È possibile definire file policy con controllo molto fine sui permessi:

```
grant {//alcuni esempi
  permission java.security.AllPermission;

  permission java.net.SocketPermission "127.0.0.1:1024-",
    "accept, connect, listen, resolve";

  permission java.net.SocketPermission "localhost:1024-",
    "accept, connect, listen, resolve";

  permission java.net.SocketPermission "*:1024-65535",
    "connect, accept";

  permission java.net.SocketPermission "*:80", "connect";

  permission java.io.FilePermission
    "/home/user/public/classes/-", "read";
};
```

Indicando il file presso java.security.policy (e.g. come parametro passato alla JVM)

DictionaryServer

```
import java.rmi.registry.LocateRegistry;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;

public class DictionaryServer {

    public static void main(String[] args)
           throws RemoteException, MalformedURLException {
           //crea un registro locale, porta standard 1099
           LocateRegistry.createRegistry(1099);
           //crea l'oggetto remoto
           RemoteDictionary<Person> dict = new RemoteDictionary<Person>();
           //lo pubblica sul registry
           Naming.rebind("EmployeesDictionary", dict);
     }
}
```


DictionaryClient

```
import java.rmi.RemoteException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.net.MalformedURLException;

public class RMIClient {
     public static void main(String[] args)
           throws RemoteException, MalformedURLException, NotBoundException {

      Registry registry = LocateRegistry.getRegistry("localhost");

      Dictionary dict = (Dictionary) registry.lookup("EmployeesDictionary");

      Person mario = dict.get("MarioRossi");
            …
      }
}
```

Estensioni di RemoteObject

RemoteServer
- Fornisce le funzioni comuni di un oggetto remoto (e.g. gestione socket etc)

UnicastRemoteObject (extends RemoteServer)
- Quasi tutti gli oggetti remoti «centralizzati» sono derivati da questa classe
- È un oggetto remoto non replicato, i riferimenti sono validi finché è «vivo»
- Serializzablie
- Accessibile tramite TCP (solitamente)

Activable (extends RemoteServer)
- Oggetti persistenti nel tempo, possibilmente replicati ed attivabili su richiesta

Parametri

Tipi primitivi
- Passati per valore

Oggetti remoti
- Passati per riferimenti

Oggetti locali (serializzabili)
- Passati per valore
- È usata la Java Object Serialization

Ogni oggetto che implementa java.io.Serializable è serializzabile/deserializablie

Oggetti non serializzabili (e.g. FileReader) possono essere wrappati in oggetti remoti e passati per riferimento.

Callbacks

«blocco di codice» come parametri (e.g. continuation passing style) e.g.:
- Funzioni
- Puntatori a funzione
- Etc..

Usate (ma non solo) per la programmazione asincrona o event-driven e.g.:
- Interazione grafica con l'utente
- Interazione con device
- I/O
- Operazioni lunghe
- Observer pattern
- Etc.

(alternativa: polling, semplice ma inefficiente)

In java oggetti che implementano/estendono interfacce/classi opportune.

Callbacks

Esempio: la funzione generica **sort** riceve in argomento una collezione da ordinare e una funzione (**callback**) per confrontarne gli elementi.

- [HS] `sort :: (a -> a -> Bool) -> [a] -> [a]`
- [C]
```
void sort(void *base, size_t nmemb, size_t size,
      int(*compar)(const void *, const void *));
```

In OO una callback è *un'istanza che fornisce l'implementazione di un metodo specificato da un'interfaccia*.

Esempio nelle API Java: Comparator e i metodi di sorting di Array

```
String[] stringArray = {"B", "c", "A", "C"};

Arrays.<String>sort(
        stringArray,
        Collections.reverseOrder());
```


Callbacks in Java

```
class SomeLongTask{

  public interface AsyncOpCallback{
    void onCompleted(OpResult result)
  }

  …

  public void asyncOp(AsyncOpCallback cb){
    new Thread(new longTaskWorker(cb)).start();
  }

  class longTaskWorker implements Runnable{
    private final AsyncOpCallback callback;
    …
    public void run(){
      …
      callback.onCompleted(operationResult);
    }
  }
}
```


Callbacks in Java

```
task.asyncOp(new AsyncOpCallback(){
  public void onCompleted(OpResult result){
    if(result.isSuccessfull()){
      system.out.println("task completed");
    }else{
      system.out.println("task completed :(");
    }};
…
```

Le interfacce possono essere comodamente implementate da *classi anonime*

Le interazioni possono essere arbitrariamente complesse e.g.:

- Callbacks specifiche e.g. onUpdate(int progress), onError(Exception ex)
- Con risultato e.g. `Credentials onLoginRequired()`
- asyncOp restituisce oggetti e.g. per sync, polling, abort etc…

Callbacks RMI

Medesimo approccio, ma con oggetti remoti.

Il server fornisce un'interfaccia «di callback»
- (e.g. `AsyncOpCallback extends java.rmi.Remote`)

Il client implementa l'interfaccia e rende remote le istanze…

1. Il client registra il proprio oggetto remoto callback presso il register rendendolo pubblico (nei limiti delle policy)…*ok, ma c'è di meglio*.
2. Il client registra l'oggetto remoto callback **solo** presso il «server» ossia solo presso l'oggetto a cui la callback deve essere fornita. A tale scopo invoca `UnicastRemoteObject.exportObject(myCallbackInst,…)`

(Esempio…)

RemoteObjects «on demand»

Mantenere oggetti remoti poco usati può essere costoso

In questi casi si utilizzano oggetti «attivabili»
- Sono creati/risvegliati all'invocazione di un loro metodo
- Disattivazione volontaria o su eventi (e.g. timeout, server load threshold)
- Gestiti dal demone **rmid**

Un oggetto attivabile estende `java.rmi.activation.Activatable`

E qualche passetto di setup in più…
- Info per creazione/ripristino dell'oggetto (`ActivationDesc`)
- Gruppo di attivazione (`ActivationGroup`) anche su più JVM
- Info per creazione/ripristino del gruppo (`ActivationGroupDesc`)

(Esempio…)

Esercizi

1. Estendere Dictionary con metodi sincroni
   - Verificare l'esistenza di una chiave
   - Cancellazione di chiavi/valori
   - Richiedere/rilasciare/verificare lock su singole chiavi (o gruppi)
   - (Java.util può fornire spunti per estendere ulteriormente l'API)
2. Estendere Dictionary secondo il pattern Observer
   - Notifiche su eventi e.g. aggiunta, modifica, cancellazione etc
3. Introdurre le notifiche nel Dictionary socket-based
4. ActivableDictionary
5. Implementare una chat rudimentale
   - Niente stanze, storico o autenticazione
6. ActivableChat con persistenza (e.g. storico/statistiche)
7. Emulare altre semantiche di invocazione (e.g. chiamate idempotenti)