RMI — Remote Method Invocation

[Pagina intenzionalmente vuota]

RMI: nozioni di base

Un'applicazione RMI è un *applicazione distribuita ad oggetti*.

Applicazione RMI tipica, strutturata in:

- *server*: JVM che:
  - crea oggetti *remoti*
  - ne rende accessibili i riferimenti
  - ne attende le invocazioni da parte dei client e le serve
- *client*: JVM che:
  - ottiene riferimenti a oggetti remoti (presenti su un server)
  - ne invoca metodi, che, eseguiti sul server, restituiscono un valore

$\Rightarrow$ se un oggetto `x` mette a disposizione i suoi metodi (ad altri oggetti), attraverso la rete:

  - i metodi di `x` si dicono *remoti*
  - l'oggetto `x` si dice *remoto*

N.B.: quelli di *client*, *server* e *oggetto remoto* sono ***ruoli***, che i componenti possono rivestire simultaneamente.

P.es. dati due oggetti `x` e `y`, componenti di un'applicazione RMI e residenti su JVM separate *X* e *Y*:

- sia `x` che `y` possono essere remoti e, in particolare:
- `x` può invocare i servizi (metodi) di `y`
- `y` può invocare i servizi (metodi) di `x`

$\Rightarrow$ per la terminologia introdotta:

  - `x` (o un sistema residente su *X* che lo contiene) fa da cliente,
  - nei confronti del server `y` (o del sistema su *Y* che lo contiene)
  - le stesse osservazioni valgono scambiando `x` e `y`

Layer RMI

Nell'architettura di un'applicazione distribuita Java, l'astrazione "oggetto remoto" è supportata dal "layer" RMI, che:

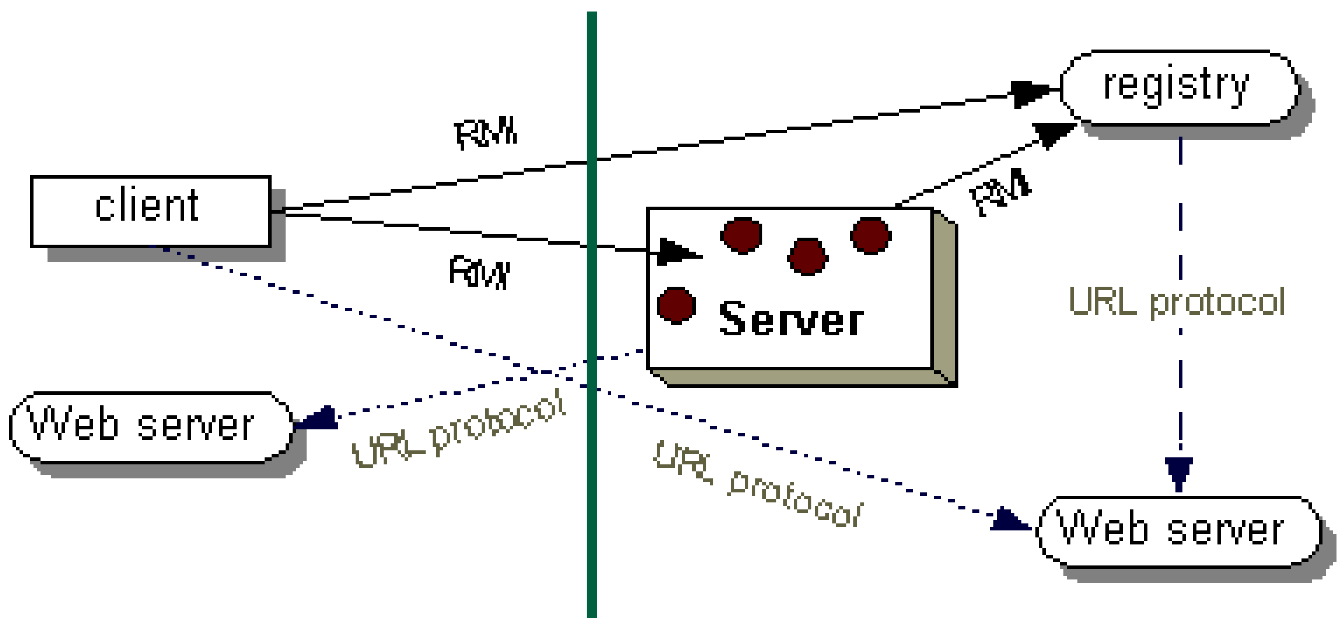
- fornisce i servizi richiesti alle applicazioni
- implementa i meccanismi necessari per questi.

Principali servizi/meccanismi del layer RMI:

- *comunicazione* tra oggetti remoti:
  - nascosta dall'astrazione fornita dagli oggetti remoti
  - implementata dai layer RMI e di trasporto (sotto RMI)
- *localizzazione* (cf. p. 110); consente a un cliente di ottenere riferimenti di oggetti remoti, mediante:
  - servizio `rmiregistry`
  - meccanismi standard del linguaggio (parametri, `return`)
- *caricamento* remoto di bytecode di classi (non remote) (cf. p. 112):
  - richiesto dalla trasparenza
  - evita limitazioni a parametri/return-value di metodi remoti

Architettura RMI

Ottenere riferimenti a oggetti remoti



`rmiregistry` è un name server distribuito per oggetti, individuati da un nome-stringa:

- il server vi registra un oggetto, associandolo al suo nome (*binding*)
- il client ottiene un riferimento all'oggetto (per lui) remoto, in cambio del nome (*lookup*)
- il cliente può ora invocare ogni metodo (dell'oggetto) remoto

Localizzazione di iferimenti remoti. Due possibilità:

1. Inizialmente, un "primo" riferimento remoto dovrebbe essere messo in circolazione via `rmiregistry` .
2. In seguito, il cliente può ottenerne altri (cf. esempi "factory", p.**??**):
   - chiamando metodi (di oggetti) remoti ...
   - che restituiscono (riferimenti a) oggetti remoti

Infine, i riferimenti remoti sono "cittadini di 1a classe":

⇒ possono "circolare" liberamente come parametri di metodi, anche remoti (p. 111).

RMI: riferimenti remoti come parametri

Come detto, un oggetto (tipicamente remoto):

- può ottenere un riferimento a un altro oggetto remoto
- ricevendolo come parametro dell'invocazione di un proprio metodo

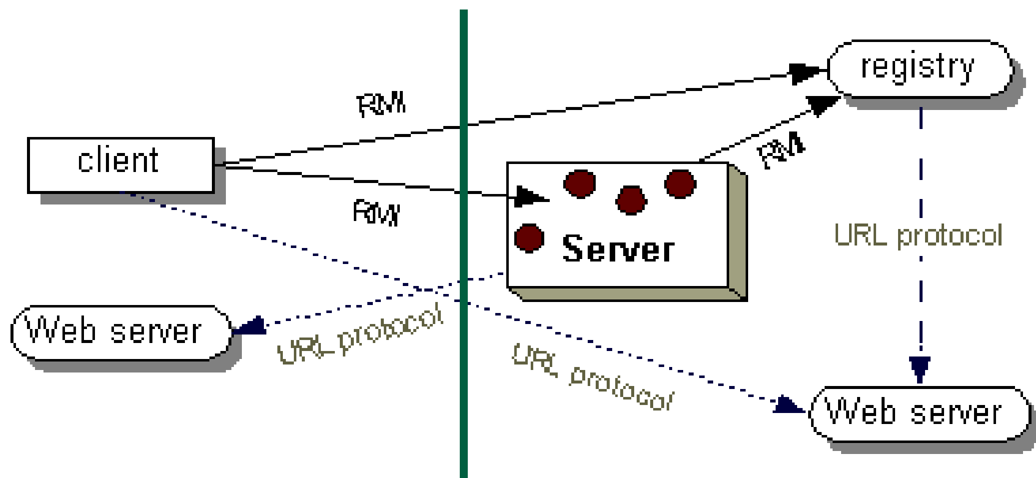
Esempio:

- `a`, `b`, `c` siano oggetti remoti sulle JVM A, B, C rispettivamente
- `b` possiede i riferimenti:
  1. `ref _a` ad `a`
  2. `ref _c` a `c`
- se `b` chiama un metodo `f()` di `c` e gli passa come argomento (il riferimento ad) `a`, p.es.:

```
ref_c.f(ref_a,...);
```

- `c` sulla JVM C otterrà, come 1° argomento del suo metodo `f()` , un riferimento all'oggetto remoto `a` sulla JVM A.

RMI: caricamento remoto di oggetti non remoti



Il sistema RMI può trasferire il bytecode di una classe *non remota* `MyClass`, dal client al server e viceversa.

Questi trasferimenti possono essere supportati anche dal protocollo http e da Web server, come in figura.

Essi possono aver luogo se un oggetto (non remoto) `MyClass` figura:

- come parametro di un metodo remoto invocato dal client (direzione client $\longrightarrow$ server)
- come oggetto restituito da un metodo remoto eseguito sul server (direzione server $\longrightarrow$ client)

Osservazioni:

$\Rightarrow$ trasparenza: nessun vincolo su parametri/return val di metodi remoti

- RMI passa automaticamente gli oggetti con il loro vero tipo Java $\Rightarrow$loading remoto 100% trasparente e *implicito*
- ciò consente di fornire nuove classi a JVM remote a *run time*, e aiuta a rendere *dinamico* il comportamento delle applicazioni (p. 123)

Programmazione con oggetti remoti in Java

Un'applicazione RMI è fatta, come le altre applicazioni Java, di:

- *interfacce*: definiscono (intestazioni di) di metodi
- *classi*, di cui alcune implementano le interfacce

Caratteristiche specifiche RMI:

- (i bytecode del)le classi possono risiedere su JVM distinte o *remote*
- su una JVM si possono invocare metodi di oggetti residenti su una JVM remota;
- tali oggetto e i suoi metodi si dicono *remoti*

RMI: aspetti linguistici

Dal punto di vista del linguaggio:

- un'interfaccia è remota se estende `java.rmi.remote` :

```
public interface RmtInt extends java.rmi.Remote {
    public int get() throws java.rmi.RemoteException;
    ...
}
```

- un oggetto è remoto se la sua classe implementa un'interfaccia remota

Dal punto di vista del "deployment":

- gli oggetti remoti e il loro bytecode vivono sulla JVM "server"
- sulla JVM "cliente", non esiste una copia dell'oggetto remoto
- esiste invece una classe *stub* che:
  - rappresenta localmente l'implementazione di una o più interfacce remote;
    p.es. per l'interfaccia remota `RmtInt` ci sarà una classe stub

    ```
    class RmtInt
    ```
  - intercetta le chiamate ai metodi remoti
  - provvede a *marshalling/unmarshalling* di parametri/risultato
  - usa i layer RMI e sottostanti per comunicare con la JVM remota

Creazione di un'applicazione RMI

Passi principali:

1. progetto e implementazione dei componenti remoti, tipicamente:
   - cliente
   - server
   - interfaccia remota
   - implementazione dell'interfaccia remota

   p.es.:

   `Client.java` `RmtInt.java` `RmtIntImpl.java` `Server.java`

2. compilazione dei sorgenti in altrettante classi
3. generazione con `rmic` di classi stub per l'interfaccia remota
4. distribuzione delle classi generate tra client e server:
   - lato client: cliente, interfaccia remota e stub
   - lato server: server e, per la classe remota: interfaccia, implementazione, stub
5. avvio dell'applicazione:
   - lato server: `rmiregistry &` e server
   - lato client: client

Generazione di applicazione RMI: strumenti

1. Progetto e implementazione dei componenti remoti; porta ai file sorgente:

   `Client.java` `RmtInt.java` `RmtIntImpl.java` `Server.java`

2. compilazione dei sorgenti; p.es.:

   `javac RmtInt.java` crea `RmtInt.class`
   `javac RmtIntImpl.java` crea `RmtIntImpl.class`
   richiede `RmtInt.class`
   `javac Client.java` crea `Client.class`
   richiede `RmtInt.class`
   `javac Server.java` crea `Server.class`
   richiede `RmtInt.class`, `RmtIntImpl.class`

3. generazione con `rmic` di:
   - stub (anche lato server, per marshalling)
   - solo per JDK 1.1, skeleton (solo lato server)

   `rmic -v1.2 RmtIntImpl` crea `RmtIntImpl_Stub.class`
   richiede `RmtInt.class` e `RmtIntImpl.class`

4. distribuzione classi e avvio dell'applicazione:
   - lato server: `rmiregistry &`, da dir in cui sono le classi di:
     - interfaccia della classe remota: `RmtInt.class`
     - stub della classe remota: `RmtIntImpl_Stub.class`
   - lato server: `java Server &`; nel class path servono:
     - interfaccia della classe remota: `RmtInt.class`
     - stub della classe remota: `RmtIntImpl_Stub.class`
     - implementazione della classe remota `RmtIntImpl.class`
   - lato client: `java Client ...` ; nel class path servono:
     - interfaccia della classe remota: `RmtInt.class`
     - stub della classe remota: `RmtIntImpl_Stub.class`

Progetto e implementazione dei componenti remoti

Passi:

- scelta dell'architettura e di come distribuire i componenti
- definizione delle interfacce *remote*, cioè che
  - linguisticamente estendono `java.rmi.Remote`
  - logicamente definiscono un'astrazione dei servizi remoti (a uso dei clienti) attraverso i loro metodi
    - per questi metodi occorre introdurre risultato/parametri, ognuno con un tipo
    - per ogni tipo nuovo va definita una classe o interfaccia
- implementazione dei servizi remoti, attraverso:
  - classi che implementano (i metodi del)le interfacce remote, e, se necessari, altri metodi (locali)

    N.B.: una classe può implementare più interfacce remote
  - implementazione di classi/interfacce di risultato/parametri dei metodi delle interfacce
- implementazione dei clienti (svincolata da quella dei servizi)

Come esempio, si veda `examples/rmi/rmtint` .

RMI: Nozioni avanzate

Oggetti remoti multipli

Come si è visto, nell'uso di base c'è una corrispondenza 1-1 tra stringhe e oggetti remoti:

1. il server istanzia un oggetto remoto
2. il server registra l'oggetto con un nome-stringa
3. il cliente ottiene un riferimento all'oggetto remoto attraverso la stessa stringa

NB: dopo (2) il server non termina, per quanto non contenga un `Thread.wait()`.
Spiegazione: il thread `main()` ha rilasciato un riferimento all'esterno (all'rmiregistry).

D'altra parte, un server che elimina il riferimento remoto, p.es. con `unbind()`, terminerà.

Problema: supponiamo che un'applicazione RMI necessiti di istanze multiple allocate dinamicamente di un oggetto remoto:

- il server potrebbe istanziare e registrare più oggetti, ma
- il client dovrebbe conoscere tutte le stringhe con cui verranno registrati gli oggetti: poco pratico!

Soluzione:

- il client fa il `lookup` di un solo oggetto remoto, di "base"
- quindi ne invoca più volte un metodo che gli restituisce ogni volta un riferimento a un nuovo oggetto remoto

Esempi: `examples/rmi/factoryRem` e `examples/rmi/factoryImp`.

Multithreading

Quando un client su una JVM A invoca un metodo di un oggetto remoto, residente sulla JVM B:

- se il client è il primo a far questo dalla JVM A, su B viene attivato un nuovo thread per l'elaborazione
- se no, viene riutilizzato il thread attivato in precedenza

Quindi:

- richieste concorrenti da thread concorrenti dalla stessa JVM (p.es. A) vengono in effetti trattate sequenzialmente
- se si desidera un maggiore parallelismo dal lato server, che rispecchi il paralleismo introdotto dal lato client, conviene che:
  - sul server (p.es. JVM B) girino istanze multiple degli oggetti remoti, ognuna eseguita da un thread distinto
  - idealmente un'istanza/thread distinto per ciascun thread dei clienti

  per questo scopo si usano gli schemi *factory* o *dispenser*

Binding all'inizializzazione

Il binding su `rmiregistry` di un oggetto remoto `x` di classe `RemXImpl` può essere effettuato:

1. dal server, subito dopo l'istanziazione:

```
x = new RemXImpl();
java.rmi.Naming.rebind( "PIPPO", x );
```

2. oppure dal costruttore della classe `RemXImpl`

```
x = new RemXImp("PIPPO");
```

   il costruttore si occuperà del binding

La seconda soluzione risulta:

- più comoda: non si deve ripetere la chiamata a `rebind()` dopo ogni istanziazione di un oggetto remoto;
- più elegante: in generale, in Java i compiti di inizializzazione degli oggetti vanno delegati, per quanto possibile, ai costruttori.

Esempio: `examples/rmi/rmtintSelfBind` .

Loading del client a run-time

Il cliente avrà un metodo, p.es. `run()` che sostituisce `main()`.

Vedi `examples/rmi/rmtintClntLoad`.

Esempio: remote compute engine

Interfaccia:

- `examples/rmi/compute/Compute.java`
  `Compute.executeTask(Task T)` esegue un `Task` generico
- `examples/rmi/compute/Task.java`
  `Task` è un'interfaccia di cui si sa solo che contiene il metodo `execute()`