Programmazione Avanzata e di Rete

Massimo Merro

Anno Accademico 2006/2007

Verona, 10 Gennaio 2007

Programmazione di Sistemi Distribuiti

- I *Sistemi Distribuiti*, per loro natura, prevedono che computazioni differenti possano essere eseguite in JVM differenti, possibilmente su host differenti, e comunicare tra loro.

- Al fine di consentire la programmazione basilare di sistemi distribuiti il linguaggio Java supporta i *socket*, un meccanismo flessibile e sufficientemente potente per consentire programmazione di sistemi distribuiti.

- I socket sono un meccanismo di basso livello e, come tutti i linguaggi a basso livello, presentano non poche difficoltà per il programmatore. In sintesi, i socket richiedono da parte del programmatore

  - la progettazione dei protocolli di comunicazione;
  - la verifica (operazione tutt'altro che banale) della funzionalità dei protocolli;
  - la codifica e decodifica dei dati trasmessi.

- Lavorando a livello di sockets le difficoltà maggiori derivano dall'eterogeneità delle entità in rete. Infatti, in generale, host differenti possono avere CPU, sistemi operativi, linguaggi, ed implementazioni differenti.

Remote Procedure Call

- Un'alternativa ai sockets è rappresentato da una tecnologia di più alto livello, comunemente chiamata RPC (*Remote Procedure Call*), in cui l'interfaccia di comunicazione è rapprensentata dall'invocazione (remota) di procedura.

- La tecnologia RPC consente di spezzare un'applicazione di rete in due moduli, una per il *server* ed una per il *client*. Tali moduli sono distribuiti su due macchine differenti ed espletano funzioni differenti.

- Utilizzata ancor oggi, la tecnologia RPC consente di invocare procedure che appartengano ad applicazioni remote, in maniera del tutto trasparente all'utente.

- Più precisamente il client invoca una procedura del server remoto, il quale si occupa di eseguire la procedura (con i parametri passati dal client) e di ritornare a quest'ultimo il risultato dell'esecuzione.

- La connessione remota è trasparente al client che ha l'illusione di invocare una procedura locale.

- Grazie alla RPC, il programmatore non deve più preoccuparsi di sviluppare dei protocolli che si occupino del trasferimento dei dati, della verifica, e della codifica/decodifica. Queste operazioni sono interamente gestite dalla RPC.

- La tecnologia RPC presenta comunque dei **limiti**:

  - parametri e risultati devono avere tipi primitivi;
  - la programmazione è essenzialmente procedurale;

- la localizzazione del server non è trasparente (il client deve conoscere l'IP e la porta su cui il server è in esecuzione);
- non vi è programmazione ad oggetti e quindi mancano i concetti di ereditarietà, incapsulamento, polimorfismo, ect...

- Dagli inizi degli anni '90 sono state proposte delle tecnologie, dette *Middleware*, per superare i limiti di RPC:
  - **CORBA**: supporta applicazioni scritte in linguaggi differenti su piattaforme differenti.
  - **Java RMI**: supporta applicazioni Java su una piattaforma Java, ovvero le applicazioni possono essere distribuite su differenti JVM.
  - **DCOM**: supporta applicazioni scritte in linguaggi differenti, ma su piattaforme Win32. Esistono delle implementazioni per sistemi Unix.
  - **.NET remoting**: supporta applicazioni scritte in linguaggi differenti, su piattaforma Win32.

Java RMI

- In questo corso studieremo Java RMI, un'estensione di Java semplice ed espressiva. Vedi:
  `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`

- Tutte le funzionalità standard di Java sono disponibili in Java RMI. In particolare, i meccanismi di sicurezza, di serializzazione dei dati, basi di dati JDBC, ect..).

- I principali vantaggi nell'utilizzare Java RMI sono i seguenti:

  – È un linguaggio ad oggetti. Di conseguenza i concetti di *riutilizzabilità* (ereditarietà), *protezione dell'informazione* (incapsulamento), ed *accesso dinamico* (polimorfismo) sono già definiti.

  – Consente il passaggio di referenze ad oggetti remoti da una JVM ad un'altra.

  – Supporta un *Java Security Manager* per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite.

– Contrariamente ad altre tecnologie, scrivere applicazioni in Java RMI è relativamente facile e veloce.

– Supporta un meccanismo di *Distributed Garbage Collection* (DGB) per disallocare quegli oggetti remoti per cui non esistano più referenze attive.

– Consente di avere oggetti remoti attivabili, ovvero servers che si attivano "on demand", cioè a seguito di una invocazione, e che si disattivano quando non utilizzati.

– RMI rende completamente trasparente l'utilizzo di oggetti remoti. Infatti, una volta localizzato l'oggetto appropriato, il programmatore utilizza i metodi dell'oggetto come se questi fosse locale. Tutto il lavoro di codifica, decodifica, verifica, e trasmissione dei dati è effettuato dal runtime RMI in maniera trasparente all'utente.

– Supporta le specifiche IIOP (Internet Inter-ORB Protocol).

Questa nuova funzionalità detta RMI-IIOP, consente l'integrazione tra applicazioni Java RMI e CORBA.

Panoramica dell'architettura Java RMI

– L'obiettivo di Java RMI è quello di fornire una tecnologia che consenta agli oggetti Java di comunicare tra loro indipendentemente dalla dislocazione delle JVM su cui sono in esecuzione. Ciò vuol dire che un client deve poter accedere un oggetto remoto in maniera del tutto simile a come accede un oggetto locale.

– L'architettura RMI si basa sulla seguente idea: la *definizione* e l'*implementazione* di un server remoto sono due concetti distinti. In particolare, il codice che definisce un server e quello relativo all'implementazione restano separati ed operano su JVM differenti.

– Questo risponde alle esigenze di programmazione di sistemi distribuiti in cui i client devono conoscere la *definizione* di un servizio remoto (cioè fornito remotamente), mentre i server devono *implementare* il servizio remoto (ovviamente per implementare qualcosa bisogna conoscerne la definizione).

– Entrando più nello specifico, la definizione di un server remoto è codificato usando un'interfaccia Java particolare[1], mentre l'implementazione è codificata attraverso una classe Java appropriata.

Creare un'applicazione distribuita in Java RMI

Nello sviluppare un'applicazione distribuita in Java RMI si seguono essenzialmente quattro passi:

1. Progettazione ed implementaz. delle componenti distribuite
2. Compilazione dei sorgenti e generazione degli stub
3. Eventuale preparazione di classi (bytecode) che verranno caricate dinamicamente dai client

[1] Si ricordi che un'interfaccia Java non contiene codice eseguibile.

4. (Eventuale) lancio del registro RMI
5. Lancio e quindi esportazione del server remoto.

Vediamo queste fasi punto per punto.

1) Progettazione ed implementazione delle componenti distribuite

In questa fase si decide la struttura dell'applicazione e si stabilisce quali funzionalità devono essere espletate da oggetti locali e quali da server remoti. Vi sono sostanzialmente tre diversi moduli la cui progettazione richiede particolare cura:

– *Definizione dell'interfaccia remota.* Un'interfaccia remota è una particolare interfaccia Java che denota quei metodi del server che possono essere invocati remotamente da un client. Tale interfaccia remota deve essere conosciuta sia dal server che dal client. Quest'ultimo, infatti, deve conoscere il nome dei metodi remoti ed il tipo dei parametri e dei risultati che vengono passati. L'interfaccia remota rappresenta quindi la modalità attraverso cui un client ed un server interagiscono tra loro.

- *Implementazione dei server remoti.* I server remoti devono implementare una o più interfacce remote (insieme ad ogni altra interfaccia che sia necessaria per usi locali). Un server può implementare più di una interfaccia remota nel caso voglia fornire "views" differenti a client differenti.
- *Implementazione dei client* Clienti che interagiscono con server remoti possono essere implementati in ogni momento, anche successivamente all'implementazione del server, purchè facciano un uso appropriato dell'interfaccia remota per accedere il server.

2) Compilazione sorgenti e generazioni stubs

La compilazione avviene in due fasi, nella prima si utilizza il compilatore **javac** per compilare i sorgenti del server e del client. Più precisamente presso il server si compilano le interfacce remote e l'implementazione del server, mentre sulla macchina client si compilano l'interfaccia remota e l'applicazione client (si noti che l'interfaccia remota deve essere conosciuta sia dal server che dal client).

Nella seconda fase si usa il compilatore **rmic** per generare nella macchina server le classi *stub* e *skeleton* relative ai server remoti. **IMPORTANTE:** Si noti che a partire dalla J2SE 5.0 tale compilazione non sarebbe più necessaria in quanto le classi stub vengono create a tempo d'esecuzione. Il compilatore **rmic** va comunque usato quando si ha a che fare con clients che utilizzano versioni precedenti di Java.

Stubs

- In Java RMI, un client non accede un server remoto direttamente, ma attraverso un oggetto ben preciso, che prende il nome di *stub*, il quale va in esecuzione sulla macchina del client ed agisce da rappresentante locale (proxy) al server remoto.

- Un oggetto **stub** traduce automaticamente ogni invocazione al server remoto in termini di comunicazione di rete, con relativo passaggio di parametri ed eventuale codifica/decodifica (marshalling/demarshalling).
- *Un oggetto stub è in pratica una referenza al server remoto.* Referenza di cui il client deve necessariamente venire in possesso se vuole invocare i metodi del server remoto.
- È compito dello **skeleton** (per lo meno nelle prime versioni di Java RMI) quello di ricevere presso l'host del server le richieste da parte dello stub, decodificarle, inoltrarle al server, codificare i risultati ed inviarli allo stub affinchè li passi al client.
- Com'è fatto un oggetto stub? Lo stato di un oggetto stub contiene essenzialmente tre informazioni: l'IP dell'host su cui gira il server, la porta su cui è in esecuzione il server, ed un identificativo RMI associato al server remoto.
- Per quanto riguarda i metodi, *la classe stub implementa la medesima interfaccia remota del server remoto.* Quindi per

il client interagire localmente con lo stub è come interagire direttamente col server remoto. Ovviamente il contenuto di questi metodi non ha niente a che vedere coi corrispondenti metodi del server, ma piuttosto si occupano di interagire col sistema RMI runtime per poter contattare il server remoto.

– Come fa un Client ad ottenere una referenza ad un server remoto (cioè l'oggetto stub associato) ad un server remoto? Ci sono due modi:

  * La riceve come risultato di un'invocazione di un metodo ad un server remoto di cui già conosce la referenza.
  * Attraverso un *servizio di Naming*, come ad esempio un registro RMI, in esecuzione presso l'host del server, in cui l'oggetto remoto viene registrato inserendo il nome e la sua referenza remota. Il client deve perciò accedere il registro RMI (che è un server remoto esso stesso) attraverso l'IP dell'host e la porta su cui è in esecuzione il registro.

3) Accesso alle classi caricabili dinamicamente

In questa fase ci si preoccupa di dislocare le classi che devono essere caricate dinamicamente dal client (quali ad esempio la classe dell'oggetto stub, etc) in un'area del server che possa essere acceduto dall'esterno, usando un qualche protocollo URL (ad esempio, HTTP, FTP, file, etc).

4) Lancio del registro RMI

Prima di lanciare il server remoto (e quindi l'applicazione client) è necessario lanciare presso il server il registro di Naming RMI. In modo che il client possa recuperare le referenze ai server remoti di cui ha bisogno.

5) Esportazione e lancio del server

Dopo essere lanciato (o nello stesso momento) il server viene *esportato* al sistema RMI cioè viene messo in ascolto su una porta (in generale definibile dall'utente) da cui accetta richieste di invocazioni remote da parte del sistema RMI. Ricorda: il client non può interagire direttamente con il server, ma solo indirettamente tramite lo stub che agisce da proxy.

Vediamo adesso un po' più in dettaglio com'è fatta un'interfaccia remota, un'implementazione di un server e come si compila il codice relativo.

Interfaccia remota

- Un'interfaccia remota è un'interfaccia Java che contiene tutti i metodi del server che possono essere invocati remotamente.

- Per comunicare con un server remoto, il codice del client deve essere a conoscenza dell'interfaccia remota implementata dal server.

- Una interfaccia remota per definirsi tale *deve* estendere l'interfaccia `java.rmi.Remote` o un'altra interfaccia remota che estenda `java.rmi.Remote`.

  L'interfaccia `java.rmi.Remote` è una semplice interfaccia "marker" e non contiene usata per distinguere un'interfaccia remota da una non-remota.

- Ad esempio, questa potrebbe essere l'interfaccia remota di un server remoto con un solo metodo:

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

public interface IntRemota extends Remote {
   public int remoteHash (String s) throws RemoteException;
}
```

- **Importante:** Tutti i metodi che appartengono ad un'interfaccia remota (inclusi quelli ereditati da una superclasse) devono lanciare `java.rmi.RemoteException` oppure una sua superclasse (ad es., `java.io.IOException`, `java.lang.Exception`, etc) per gestire eccezioni remote che possono avere origine al livello di client, server, o della rete che li connette. In caso contrario il codice non supererà la compilazione `rmic`.

- **Importante:** L'interfaccia deve esplicitamente essere dichiarata come *public*.

- **Importante:** Quando si passa un server remoto (ovvero una sua referenza remota) come argomento o risultato di un metodo remoto, la variabile relativa deve essere dichiarata del tipo dell'interfaccia remota e non dell'implementazione.

- Questo perchè quello che si riceve non è ovviamente un server remoto (il quale resta dov'è) ma un sua referenza remota (uno stub) la cui classe implementa la medesima interfaccia remota del server.

Implementazione del server remoto

- L'implementazione di un oggetto remoto deve estendere la classe `java.rmi.server.RemoteObject`, o una sua sottoclasse, e deve implementare tutte le interfacce remote che intende supportare.
- Cominceremo col vedere server remoti della sottoclasse `java.rmi.server.UnicastRemoteObject`. Un esempio:

```
public class ServerRemoto extends UnicastRemoteObject
                          implements IntRemota {
    public ServerRemoto () throws RemoteException {
      super();
    }
    public int remoteHash (String string) {
      return string.hashCode ();
    }
    public static void main (String args[]){ .......
   }
}
```

- Notate che estendiamo `UnicastRemoteObject` ed implementiamo l'interfaccia remota.

- Notate anche che poichè la superclasse `UnicastRemoteObject` può emettere eccezioni di tipo `RemoteException`, il costruttore della nostra implementazione deve passare (per lo meno) tale eccezione (o una sua super-classe).

Compilazione del server

Come detto, la compilazione avviene in due parti:

1. Sull'host del server si compila l'interfaccia remota e l'implementazione del server. Ad esempio:

   ```
   javac IntRemota.java ServerRemoto.java
   ```

2. Dopodichè, sempre presso l'host del server, vengono generati Le classi stub e sleketon attraverso il compilatore `rmic`.

```
rmic ServerRemoto
```

Questo comando produce le classi `ServerRemoto_Stub.class` e `ServerRemoto_Skel.class`. Entrambi le classi sono necessarie per l'esecuzione dell'implementazione dell'oggetto remoto.

Un client in possesso di una referenza remota, ovvero un oggetto stub, per poterlo utilizzare deve accedere alla classe di appartenenza dell'oggetto stub:

```
ServerRemoto_Stub.class
```

Come ogni classe Java essa contiene il codice da eseguire (la referenza remota) deve quando viene invocato un metodo dello stub.

IMPORTANTE: In effetti quando un client riceve una referenza remota (uno stub) oltre a ricevere l'oggetto stub caricherà *dinamicamente* in maniera trasparente la classe dell'oggetto stub attraverso un meccanismo detto *codebase* che in Java RMI gioca un ruolo molto importante durante la trasmissione di referenze remote.

Naming: registro RMI

- Per poter invocare un metodo di un server remoto, un client deve prima ottenere una referenza remota (cioè uno stub) del server. Per comodità, tale referenza remota può essere mantenuta in un registro RMI (che è esso stesso un server remoto in esecuzione presso l'host remoto) in cui vengono registrate coppie della forma *(NomeServer, ReferenzaRemota)*.

- Il registro di naming può essere lanciato o da linea di comando, attraverso `rmiregistry`, oppure da programma, attraverso i metodi della classe `LocateRegistry`. Come vedremo nelle esercitazioni tale fase è molto delicata e spesso fonte di errori.

- Il registro RMI presso l'host del server è semplicemente identificato con l'IP dell'host e la porta su cui è in esecuzione il registro (di default la 1099).

- Quindi, una volta acceduto il registro RMI, il client può ricavare la referenza remota di un server remoto in esecuzione presso l'host remoto semplicemente indicando il nome del server.

- Comunque, come già detto, la referenza remota in sé, ovvero lo stub, non è sufficiente per poter invocare un metodo remoto del server. Infatti l'oggetto stub agisce localmente presso il client come un proxy verso il server. Ma per poter eseguire un metodo dello stub il client ha bisogno la classe (`ServerRemoto_Stub.class`) dell'oggetto stub. Tale codice risiede presso l'host del server ed il client dovra potersela caricare a tempo di esecuzione. Vedremo come.