Azzolini Riccardo

Remote Method Invocation

1 Architetture distribuite

Solitamente, nell'ambito delle applicazioni informatiche, si distingue tra:

architetture locali: tutti i componenti sono sulla stessa macchina;

architetture distribuite: le applicazioni e i componenti possono risiedere su nodi diversi, messi in comunicazione da una rete.

Un'architettura distribuita ha diversi vantaggi, tra cui

- accesso a dati non disponibili localmente,
- uso di programmi concorrenti (parallelismo, distribuzione del carico elaborativo, ecc.),

ma anche lo svantaggio di una maggiore complessità, soprattutto per quanto riguarda la comunicazione tra i vari componenti.

Le architetture distribuite possono essere classificate in base al grado di distribuzione:

- **Client–server** (l'architettura di cui si è parlato finora).
- **Multi-livello** (*multi-tier* o *multi-layer*): ogni nodo funge da client nei confronti del livello inferiore e da server nei confronti del livello superiore.

  Un esempio tipico è un'organizzazione a tre livelli (three-tier), che sono:

  1. interfaccia utente ed elaborazione locale;
  2. logica applicativa;
  3. gestione dei dati persistenti.

- **Completamente distribuita**: caratterizzata da oggetti / nodi ciascuno dei quali può comportarsi sia da client che da server, anche nei confronti di uno stesso altro oggetto (cioè un oggetto A può comportarsi da client di B in un certo momento, e da server per B in un altro). Così, il carico elaborativo può essere distribuito in modo molto flessibile.

  L'esempio tipico sono le applicazioni **peer-to-peer** (file sharing, ecc.), così chiamate appunto perché tutti i nodi si comportano "da pari".

2 Middleware

Si chiama **middleware** il software che rende accessibili su Internet risorse hardware o software che prima erano disponibili solo localmente, permettendo a un'applicazione di interagire con altre attraverso la rete senza che l'utente debba capire e codificare le operazioni di basso livello necessarie per tale interazione.

Di fatto, il middleware è la "colla" che tiene insieme le applicazioni distribuite, sgravando il programmatore da incombenze "banali" come ad esempio la gestione esplicita dei socket (anche se già i socket sono un'interfaccia di livello relativamente alto).

Le interazioni tra applicazioni (che il middleware realizza) possono essere di due tipi:

sincrone: il sistema richiedente attende la risposta;

asincrone: non si attende la risposta, che sarà invece accettata quando arriverà, ma intanto il sistema richiedente può continuare a svolgere altre attività.

3 Remote Method Invocation

RMI (**Remote Method Invocation**) è una tecnologia, specifica del mondo Java (a differenza, ad esempio, dei socket), che permette a processi Java distribuiti di comunicare attraverso una rete.

In particolare, con RMI, un processo (client) può fare una chiamata di metodo a un oggetto remoto (server) *come se* l'oggetto fosse sulla sessa macchina. Ad esempio, il codice

```
String response = server.sayHello();
```

potrebbe funzionare ugualmente se `server` fosse un oggetto locale o remoto.

Si dice che RMI consente una **location transparency**: il client può ignorare dove si trovi l'oggetto remoto, e addirittura il fatto che l'oggetto associato a un riferimento sia remoto o locale; se è remoto, ci pensa RMI a gestire la necessaria comunicazione con il server.

In realtà, mentre la sintassi dell'invocazione di metodi su oggetti remoti è identica a quella per gli oggetti locali, la semantica (il significato dell'operazione) è leggermente diversa, quindi la location transparency non è completa. Ciò è inevitabile, poiché non è possibile trascurare completamente il fatto che l'applicazione sia distribuita.

4 Architettura client–server con RMI

Tipicamente, quando si realizza un'architettura client–server con RMI:

- Il server crea degli oggetti remoti (rispetto ai client), li rende visibili, e aspetta che i client invochino metodi su di essi. Questi oggetti devono essere reperibili e referenziabili da programmi che girano su macchine diverse.
- Il client ottiene dei riferimenti a oggetti remoti e invoca metodi su di essi,

In altre parole, RMI fornisce il meccanismo attraverso il quale server e client comunicano per costituire l'applicazione distribuita.

5 Registry

Affinché un client possa interagire con un oggetto server remoto, deve possedere un riferimento a esso. Per ottenere tale riferimento, il client può usufruire del servizio di **registry**:

1. Il server pubblica sul registry un riferimento all'oggetto che vuole mettere a disposizione dei client.
2. Il client (che può anche non conoscere il server) cerca per nome presso il registry il servizio di cui ha bisogno presso, e gli viene restituito un riferimento all'oggetto remoto sul server.
3. Il client usa il riferimento all'oggetto per ottenere il servizio, e può anche passare il riferimento ad altri client.

6 Message passing

Con RMI, l'interazione tra client è server è quella tipica dei sistemi object-oriented, cioè il **message passing**.[1]

In particolare, l'invocazione di un metodo di un oggetto remoto avviene mediante l'invio di un messaggio che contiene il nome del metodo e gli eventuali argomenti, e poi l'oggetto remoto invia a sua volta al chiamante un messaggio contenente il risultato del metodo.

La costruzione, spedizione, ricezione e ricostruzione dei messaggii sono gestite da RMI, in modo trasparente al programmatore.

[1] Si ricorda che, concettualmente, chiamare un metodo di un oggetto significa *passare un messaggio* a tale oggetto.

6.1 Passaggio dei parametri

Il passaggio dei parametri per l'invocazione di un metodo è una delle situazioni in cui il fatto che l'applicazione sia distribuita non è del tutto trasparente.

Considerando, ad esempio, la chiamata

```
String response = server.sayHello(obj);
```

dove `server` è un oggetto remoto, si ha un comportamento diverso a seconda della natura dell'argomento `obj`:

- Se è un riferimento a un oggetto remoto, al metodo remoto viene passato tale riferimento.
- Se è un oggetto locale, viene inviata al metodo remoto una copia dell'oggetto (mediante la serializzazione). In particolare, quello che avviene è il *passaggio per valore con deep copy*: se l'oggetto contiene riferimenti ad altri oggetti, anche questi vengono serializzati e passati al server
- Se è di tipo primitivo, viene passato per valore (copiato dal client all'oggetto remoto).

Potrebbe accadere che l'oggetto remoto non conosca la classe di `obj`. In questo caso, con RMI, è possibile passare al server anche il codice di tale classe. Questa funzionalità è chiamata **dynamic class loading**.

7 Implementazione di RMI

Un'invocazione di un metodo di un oggetto remoto è gestita localmente da uno **stub**, che funge da surrogato locale dell'oggetto remoto: esso si occupa della creazione del messaggio contenente il nome e gli argomenti del metodo (*parameter marshalling*), e dell'invio di tale messaggio al server.

A lato server, il messaggio è ricevuto da uno **skeleton**, che ricostruisce i parametri (*unmarshalling*) e chiama il metodo sull'oggetto vero e proprio (che per lo skeleton è locale).

Infine, lo skeleton prepara un messaggio contenente il risultato, lo invia allo stub, e quest'ultimo passa il risultato al chiamante.

Nota: Originariamente, stub e skeleton andavano creati appositamente, mediante il compilatore RMI. Nelle versioni recenti di Java, invece, skeleton e stub sono impliciti, cioè gestiti in maniera trasparente al programmatore.

7.1 Layer di RMI

RMI è organizzato in diversi strati:

1. lo strato di trasporto è generalmente fornito da TCP;
2. il remote reference layer (RRL) crea e gestisce i riferimenti remoti;
3. stub e skeleton comunicano tra di loro usando i riferimenti remoti, e fanno da intermediari locali per il client e per il server.