Vittorio Scarano

Programmazione con Oggetti Distribuiti: Java RMI

Programmazione con Oggetti Distribuiti: Java RMI

Vittorio Scarano

II edizione - Marzo 2010

A mio padre e a mia madre, ispiratori

A mia moglie, dolce compagna, amica e confidente

Ai miei tre figli, gioielli e luce dei miei occhi

EMERGENCY è un'associazione italiana indipendente e neutrale. EMERGENCY offre assistenza medico-chirurgica gratuita e di elevata qualità alle vittime civili delle guerre, delle mine antiuomo e della povertà.
Dal 1994 più di 3 milioni di persone sono state curate gratuitamente in ospedali, cliniche e centri di riabilitazione di EMERGENCY dove inoltre è garantita la formazione del personale medico locale.
Tre ospedali, un centro di maternità e 29 posti di primo soccorso in Afganistan, un ospedale e un posto di primo soccorso in Cambogia, un centro di riabilitazione in Iraq, un poliambulatorio per migranti a Palermo, un ospedale e un centro pediatrico in Sierra Leone, una clinica pediatrica e un centro di cardiochirurgia in Sudan, due centri pediatrici in Darfur e in Repubblica Centrafricana, e un centro di Ostetricia e Ginecologia in Nicaragua in costruzione.
Ecco gli ospedali che Emergency ha voluto, ha costruito e tuttora gestisce grazie al contributo di migliaia di persone.

EMERGENCY

Via Gerolamo Vida 11
20127, Milano
tel. 02/881881 - fax 02/86316336

http://www.emergency.it
e mail: info@emergency.it

Conto corrente postale
intestato a "EMERGENCY Ong Onlus"
n. 2842 6203
IBAN: IT37 Z076 0101 6000 0002 8426 203

Conto corrente bancario
intestato a "EMERGENCY Ong Onlus"
IBAN: IT 02 X 05018 01600 000000130130
presso Banca Etica, Filiale di Milano

L'autore devolve interamente i propri proventi in favore dei progetti di Emergency.

Indice

1 Introduzione **1**
1.1 I sistemi distribuiti 2
1.2 Un modello di riferimento: Open Distributed Processing 3
1.2.1 Le caratteristiche di un sistema distribuito 4
1.2.2 I requisiti non funzionali di un sistema distribuito 5
1.2.3 La trasparenza di un sistema distribuito 7
1.2.4 I diversi punti di vista (*viewpoints*) per un sistema distribuito 10
1.3 Il Middleware ad Oggetti Distribuiti 13
1.3.1 Il progenitore: Remote Procedure Calls (RPC) 15
1.3.2 Da RPC al Middleware ad Oggetti Distribuiti 15
1.3.3 Esempi rappresentativi: CORBA, Java RMI e .NET 16
1.4 Il Middleware ad Oggetti Distribuiti nel Modello a Componenti 17
Note bibliografiche 19
Spunti per lo studio individuale 20

Introduzione a Java RMI **25**

2 Dai socket agli oggetti remoti **25**
2.1 Introduzione 26
2.2 Un breve richiamo sulla programmazione con i socket 26
2.2.1 I socket TCP 27
2.2.2 Gli stream 27
2.2.3 HelloWorld con i socket 28
2.2.4 Un esempio di client-server con i socket 31
2.3 Da un oggetto locale 34
2.4 ... all'oggetto remoto 36
2.4.1 Il server e la interfaccia remota 37
2.4.2 Il client 39
2.4.3 Lo strato stub/skeleton 39
2.4.4 La sequenza delle invocazioni 43
2.4.5 Per lanciare la applicazione 43
2.5 Indirizzamento dell'oggetto remoto 45
2.5.1 Il Client 45
2.5.2 Lo Skeleton 47
2.6 Alcuni commenti conclusivi 49
2.7 Approfondimenti 49
2.7.1 Il funzionamento di `getOutputStream()` 49
2.7.2 Conversione di `Byte` unsigned 50
Note bibliografiche 50
Spunti per lo studio individuale 51

3 Presentazione di Java Remote Method Invocation 53
3.1 Introduzione . . . 54
3.2 Gli obiettivi della progettazione di Java RMI . . . 54
3.3 Il modello a oggetti distribuiti di Java RMI . . . 56
3.3.1 La struttura delle classi di Java RMI . . . 56
3.3.2 Il meccanismo di invocazione remota . . . 58
3.3.3 La differenza tra il modello a oggetti locale e quello remoto . . . 61
3.4 La architettura di Java RMI . . . 62
3.4.1 I tre layer della architettura . . . 62
3.4.2 Distributed Garbage Collection . . . 66
3.4.3 Caricamento dinamico delle classi . . . 67
3.5 Approfondimenti . . . 68
3.5.1 L'eterogeneità nelle tecnologie ad oggetti distribuiti . . . 68
3.5.2 La trasparenza degli oggetti distribuiti . . . 70
3.5.3 La sicurezza in Java . . . 74
3.5.4 Il meccanismo di marshalling usato da Java RMI . . . 75
Note bibliografiche . . . 76
Spunti per lo studio individuale . . . 78

4 Un primo esempio con Java RMI 81
4.1 Introduzione . . . 82
4.2 Il processo di creazione di un programma Java RMI . . . 82
4.2.1 Definizione della interfaccia remota . . . 82
4.2.2 Implementazione del server . . . 83
4.2.3 Compilazione del server . . . 83
4.2.4 Compilazione con lo stub compiler `rmic` . . . 84
4.2.5 Servizio di naming: `rmiregistry` . . . 85
4.2.6 Esecuzione del server . . . 85
4.2.7 Registrazione del server sul servizio di naming . . . 88
4.2.8 Implementazione del client . . . 88
4.2.9 Compilazione ed esecuzione del client . . . 89
4.3 L'esempio HelloWorld . . . 89
4.3.1 Definizione della interfaccia remota . . . 89
4.3.2 Implementazione del server . . . 90
4.3.3 Compilazione del server . . . 91
4.3.4 Compilazione con lo stub compiler `rmic` . . . 91
4.3.5 Servizio di naming: `rmiregistry` . . . 91
4.3.6 Esecuzione del server . . . 92
4.3.7 Registrazione del server sul servizio di naming . . . 92
4.3.8 Implementazione del client . . . 92
4.3.9 Compilazione ed esecuzione del client . . . 93
4.4 La sicurezza e la policy del Security Manager . . . 93
4.5 Commenti conclusivi . . . 94
Note bibliografiche . . . 95
Spunti per lo studio individuale . . . 96

Programmazione con Java RMI 99

5 Design Pattern con Java RMI 99
5.1 Introduzione . . . 100

5.2 L'Adapter . . . 100
5.2.1 Un esempio di Adapter: un contatore remoto . . . 100
5.2.2 Contatore remoto: il server . . . 101
5.2.3 Contatore remoto: il client . . . 105
5.2.4 Alcuni commenti all'esempio . . . 106
5.3 La Factory . . . 106
5.3.1 Un esempio di Factory: HelloWorld multilingua . . . 107
5.3.2 HelloWorld multilingua: il server . . . 107
5.3.3 HelloWorld multilingua: il client . . . 110
5.3.4 Alcuni commenti all'esempio . . . 111
5.4 L'Observer . . . 111
5.4.1 Un esempio di Observer: la callback per le architetture client-server . 112
5.4.2 Awareness con callback: la architettura . . . 113
5.4.3 Awareness con callback: lato client . . . 114
5.4.4 Awareness con callback: lato server . . . 117
5.4.5 Alcuni commenti all'esempio . . . 119
Note bibliografiche . . . 119
Spunti per lo studio individuale . . . 120

6 Una chat con Java RMI 121
6.1 Introduzione . . . 122
6.1.1 Le funzionalità di una chat . . . 122
6.2 Chat con i socket . . . 123
6.2.1 Un esempio con socket TCP . . . 123
6.2.2 Un esempio con socket UDP Multicast . . . 128
6.2.3 Commenti e ulteriori sviluppi . . . 131
6.3 Una chat client-server con Java RMI . . . 133
6.3.1 La architettura software . . . 133
6.3.2 Il server . . . 133
6.3.3 Il client . . . 136
6.3.4 Commenti e ulteriori sviluppi . . . 139
6.4 Una chat peer2peer . . . 141
6.4.1 Un primo esempio di peer . . . 141
6.4.2 Commenti e ulteriori sviluppi . . . 146
6.5 Approfondimenti . . . 147
6.5.1 UDP e indirizzi IP di gruppo . . . 147
Note bibliografiche . . . 148
Spunti per lo studio individuale . . . 149

7 Alcuni esercizi 151
7.1 Introduzione . . . 152
7.2 Una chat CS per "Cuori solitari" . . . 152
7.2.1 Il testo dell'esercizio . . . 152
7.2.2 Una soluzione . . . 152
7.3 Una chat CS per un docente dispotico . . . 159
7.3.1 Un confronto tra differenti strategie . . . 160
7.3.2 Una soluzione . . . 161
Spunti per lo studio individuale . . . 168

Argomenti Avanzati su Java RMI 173

8 Gestione dinamica di RMI 173
8.1 Introduzione . . . 174
8.2 Caricamento dinamico delle classi . . . 174
8.2.1 Il ClassLoader . . . 174
8.2.2 Caratteristiche del caricamento dinamico . . . 175
8.2.3 Gli scenari di utilizzo . . . 175
8.3 Stub e Skeleton: la evoluzione . . . 183
8.3.1 Stub e Skeleton versione JDK 1.1 . . . 183
8.3.2 Stub versione JDK 1.2 . . . 187
8.3.3 JDK 5: niente Stub e Skeleton! . . . 189
Note bibliografiche . . . 193
Spunti per lo studio individuale . . . 194

9 RMI e Java Enterprise Edition 195
9.1 Introduzione . . . 196
9.2 Java Enterprise Edition . . . 196
9.2.1 Le architetture multi-tier . . . 196
9.2.2 La architettura di Java EE . . . 197
9.3 Corba e IIOP . . . 198
9.4 Java RMI-IIOP . . . 200
9.4.1 Java RMI e Corba . . . 200
9.4.2 Le differenze tra RMI e RMI-IIOP . . . 200
9.4.3 Un esempio in Java RMI-IIOP . . . 201
Note bibliografiche . . . 206
Spunti per lo studio individuale . . . 207

Prefazione

Obiettivi

Questo libro su Java RMI ha due obiettivi. Il primo è quello di fornire allo studente il collegamento tra i tradizionali corsi di programmazione di rete, che sono basati sulla comunicazione basata su TCP/IP ed i socket, e i corsi di programmazione di applicazioni distribuite che tipicamente usano strumenti di alto livello come il modello a componenti distribuite (ambienti Enterprise) oppure il modello orientato ai servizi (ambienti Web Services). Quello che manca, e che si intende fornire con questo libro, è il collegamento che permette allo studente di comprendere in quale maniera Java Enterprise Edition, ad esempio, realizza la infrastruttura di comunicazione tra oggetti, utilizzando Java Remote Method Invocation oppure quanto del modello delle architetture orientate ai servizi (Service Oriented Architecture) derivi direttamente dai modelli di invocazione remota strettamente accoppiati (come Java RMI).

Quindi, il viaggio che si intende far intraprendere allo studente illustrerà come la programmazione di rete serve alla invocazione remota di metodi che servono poi a realizzare la infrastruttura di comunicazione. In un certo senso, si vuole usare la programmazione di socket e la invocazione remota di metodi come la programmazione assembly per la programmazione con un linguaggio evoluto, illustrando i meccanismi e le tecnologie coinvolte nell'ambiente Java. Tutto questo passerà anche attraverso la presentazione di alcuni concetti fondamentali per le architetture di sistemi distribuiti per contestualizzare la presenza di Java RMI.

Il secondo obiettivo è quello di fornire le basi per la programmazione distribuita di oggetti, sviluppando quindi un certo numero di esempi pratici che guideranno lo studente, a partire dalla implementazione con i socket, a realizzare semplici applicazioni che però serviranno ad evidenziare problematiche tipiche dei sistemi client-server e dei sistemi peer2peer.

Dal punto di vista dello stile di programmazione, la scelta è stata quella di evitare di sovraccaricare il codice, senza presentare, ad esempio, interfacce grafiche ma limitandosi alla shell in modo da limitare la quantità di codice da presentare. Il cosiddetto *coding style* di Java viene utilizzato, anche se, per ragioni di spazio e di efficacia grafica, molti dei commenti nel codice sono ridotti all'osso, e mancano del tutto i commenti JavaDoc. Questo stile di programmazione (utilizzato per rendere più agevole lo studio e la presentazione grafica) non è da ritenere un suggerimento, anzi! Sia ben chiaro allo studente che commenti adeguati e informativi sono estremamente utili per la manutenzione e la evoluzione del software e che la presentazione stringata è esclusivamente per limitare la lunghezza del codice stampato.

Struttura del libro

Il libro consta di tre parti. Nella prima si introduce Java Remote Method Invocation attraverso l'utilizzo dei socket. Nella seconda si affrontano i diversi esempi di programmazione

usando Java RMI, allo scopo di introdurre tecniche e metodi utili per la realizzazione di semplici applicazioni remote. Nella terza parte si trovano alcuni approfondimenti sulla struttura interna di Java RMI, con la capacità di gestire a run-time le invocazioni, e RMI-IIOP.

A chi si rivolge il libro

Il libro può essere utilizzato come testo di riferimento per corsi universitari di Programmazione Distribuita di I livello (terzo anno) oppure di II livello (primo anno) in Informatica e Ingegneria Informatica. Si suppone che gli studenti abbiano una buona conoscenza di Java, con esperienza nell'uso di Integrated Develoment Environment (IDE) e qualche conoscenza di base di Ingegneria del Software. Tra le competenze specifiche di Java che sono necessarie, una qualche semplice esperienza con i socket (anche in altri linguaggi) e una semplice conoscenza dei thread.

Il materiale del libro può essere usato per un corso da 4-6 CFU, a seconda del livello di dettaglio e delle esercitazioni da fare in laboratorio (qualora previsto).

Gli strumenti utilizzati

Composizione tipografica. Per scrivere questo libro è stato utilizzato LaTeX ed in particolare la versione per Windows distribuita con MiKTeX 2.8 (`http://www.miktex.org`). TeXlipse 1.2.2 (`http://texlipse.sourceforge.net/`) è un plugin di Eclipse che è stato usato per facilitare la composizione.

Per la traduzione di programmi Java in LaTeX si è utilizzato, invece, Java2HTML versione 1.5.0 (`http://www.java2html.de`) che permette anche la conversione da Java in LaTeX, con il syntax-coloring ed è disponibile anche come plugin per Eclipse. Per la creazione delle immagini, si usa il plugin di Eclipse eDump 1.7.1 (`www.bdaum.de`) che fornisce la possibilità di catturare immagini delle varie componenti di Eclipse con notevole flessibilità.

Linguaggio e ambiente integrato di sviluppo. In generale, si è utilizzata la ultima versione disponibile al momento della Java VM di Sun, vale a dire la versione Java SE Development Kit (JDK) 6 Update 11. Insieme ad essa, suggeriamo di scaricare in locale la documentazione Java disponibile sul sito, che porta una notevole quantità di informazioni e di guide per i vari argomenti.

Per la scrittura e la documentazione dei programmi di esempio si + usato Eclipse (`http://www.eclipse.org`), versione 3.4.1 con il plugin per eUML2 Free di Soyatec versione 3.2.1 (`http://www.soyatec.com/euml2/`).

Convenzioni

L'uso e l'abuso della lingua inglese. In questo libro si cerca di evitare l'uso di forzate traduzioni di termini inglesi che sono oramai diventati di uso tecnico. Ben lungi dal prefigurarsi come sudditanza linguistica verso oltreoceano, questa scelta è motivata dal fatto che questo ben prepara gli studenti di oggi che, da professionisti dell'informatica di domani, si troveranno, necessariamente, a contatto con documentazione di tecnologia recente che sarà disponibile, probabilmente, solamente in inglese tecnico.

Convenzioni tipografiche. Per indicare una sessione di lavoro ad una shell (come ad esempio per la compilazione o esecuzione di un programma) si è scelto di utilizzare come esempio una semplice shell DOS[1]. Per rappresentare una sessione useremo la seguente convenzione:

```
Z:\>echo Ciao Ciao

Z:\>
```

Per indicare un listato di una classe Java di nome `Hello.java` si utilizzerà questo standard:

HelloWorld.java

```
1 // Una classe di esempio
2 public class HelloWorld {
3   public static void main(String[] args) {
4     System.out.println("Hello World!");
5   }
6 }
```

Fine: HelloWorld.java

In generale, comandi da shell vengono indicati in grassetto, come ad esempio, il compilatore Java **javac**.

[1] In alternativa, si può utilizzare un plugin per Eclipse chiamato WickedShell (`http://www.wickedshell.net`) che permette di creare shell molto flessibili come view di Eclipse.

Prefazione alla seconda edizione

Questa edizione aggiunge alla precedente la terza parte, con gli argomenti avanzati: gestione dinamica e RMI-IIOP. Ovviamente, sono stati corretti anche gli errori, così come segnalati dai precisi e puntuali commenti che gli studenti del corso di Programmazione Distribuita del corso di Laurea in Informatica dell'Università di Salerno hanno fornito, durante l'anno accademico 2008-2009.

Ringrazio sentitamente tutti gli studenti che hanno inviato commenti, suggerimenti, correzioni, perchè il libro ne ha beneficiato notevolmente, e con questo, spero, ne beneficeranno anche i loro colleghi che lo useranno per lo studio negli anni successivi.

Anche i diritti di questa II edizione saranno interamente versati ad Emergency, nel ricordo di Teresa Sarti Strada, scomparsa il 1° settembre 2009.

Capitolo 1

Introduzione

Indice

1.1 I sistemi distribuiti **2**
1.2 Un modello di riferimento: Open Distributed Processing **3**
1.2.1 Le caratteristiche di un sistema distribuito 4
1.2.2 I requisiti non funzionali di un sistema distribuito 5
1.2.3 La trasparenza di un sistema distribuito 7
1.2.4 I diversi punti di vista (*viewpoints*) per un sistema distribuito . . . 10
1.3 Il Middleware ad Oggetti Distribuiti **13**
1.3.1 Il progenitore: Remote Procedure Calls (RPC) 15
1.3.2 Da RPC al Middleware ad Oggetti Distribuiti 15
1.3.3 Esempi rappresentativi: CORBA, Java RMI e .NET 16
1.4 Il Middleware ad Oggetti Distribuiti nel Modello a Componenti **17**
Note bibliografiche **19**
Spunti per lo studio individuale **20**

1.1 I sistemi distribuiti

Un sistema distribuito consiste di un insieme di macchine, ognuna gestita in maniera autonoma, connesse attraverso una rete. Ogni nodo del sistema distribuito esegue un insieme di componenti che comunicano e coordinano il proprio lavoro attraverso uno strato software detto *middleware*, in maniera che l'utente (utente del sistema ma anche, in maniera limitata, il programmatore e progettista) percepisca il sistema come un'unica entità integrata.

In generale, i sistemi distribuiti rispondono a motivazioni sia di tipo economico che di natura tecnologica. Per quanto riguarda il contesto sociale ed economico, i sistemi distribuiti rispondono in maniera precisa alle esigenze ed alle richieste della economia di mercato che è caratterizzata da numerose e frequenti acquisizioni, integrazioni e fusioni di aziende. Quindi, la necessità di affrontare in tempi brevi la integrazione dei sistemi di Information Technology di aziende diverse, che si sono fuse insieme, richiede una infrastruttura versatile e agile, che permetta di poter essere operativi in pochissimo tempo. Critica, infatti, è la possibilità di poter essere attivi sul mercato con il nuovo *brand* e gli esempi recenti delle acquisizioni e fusioni di complessi sistemi bancari rendono bene l'idea della necessità di provvedere al più presto alla erogazione di servizi informatici per i clienti della nuova banca. Allo stesso tempo, spesso sistemi informativi di aziende che vengono separate dalla "casa madre" in un meccanismo di cosiddetto "*downsizing*" devono mantenere un certo livello di integrazione con le aziende del gruppo, in una sorta di federazione di sistemi che complica la gestione, prevedendo tre livelli di accesso al sistema informativo: dall'interno della azienda, dall'interno della federazione di aziende e dall'esterno.

I sistemi distribuiti rispondono anche alla esigenza del mercato specifico della Information Technology, dove il tempo necessario per poter arrivare al prodotto finale, dalla ideazione, progettazione e realizzazione (il cosiddetto *time to market*) deve essere reso quanto più breve possibile, sia per il ricambio tecnologico continuo, ma anche perché le richieste dei consumatori variano significativamente in poco tempo. Sistemi che devono offrire funzionalità complesse sono tipicamente assemblati utilizzando componenti preesistenti (i cosiddetti prodotti *off the shelf*) che, però, spesso hanno requisiti hardware e software che rendono necessario separarli in diversi ambienti ed host per utilizzarli attraverso la realizzazione di un sistema distribuito.

Infine, la diffusione di Internet e la naturale evoluzione a fornire servizi attraverso la rete, implica che qualsiasi servizio è potenzialmente accessibile da una platea smisurata di utenti, e quindi soggetto, potenzialmente, a dei picchi di carico non previsti, magari, paradossalmente, dovuti ad una pubblicità positiva[1]. I sistemi distribuiti sono capaci di poter reggere meglio dei sistemi centralizzati o client-server agli improvvisi picchi di carico e quindi rispondono anche a questa esigenza di assicurare la scalabilità del servizio che forniscono in "ogni" condizione.

Insieme alle motivazioni economiche, la tecnologia offre diverse motivazioni all'introduzione dei sistemi distribuiti. Lo sviluppo dell'informatica è sempre stato condotto dal rapidissimo sviluppo delle tecnologie hardware. Le capacità di calcolo, comunicazione, memorizzazione che vengono oggigiorno offerte (già in maniera sorprendente) verranno rapidamente superate nel giro di pochissimi anni. Diverse "leggi" empiriche elaborate negli anni si sono provate fedeli nel prevedere la velocità di evoluzione. Ad esempio, ben conosciuta è la Legge di Moore che afferma che la densità dei transistor nei processori

[1]Verso la fine degli anni '90, quando ancora non si era potuto adeguare le architetture dei sistemi al boom di Internet e alla sua diffusione, capitava che un nuovo servizio su un sito venisse menzionato in articoli di giornali specializzati e questi veniva improvvisamente subissato di richieste e andava in crash ed era irrangiugibile per giorni! Questo veniva chiamato il *kiss of death* della pubblicità positiva.

si raddoppia ogni 18 mesi[2]. In pratica, questa legge afferma che la potenza di calcolo raddoppia ogni 18 mesi e quindi guadagna un ordine di grandezza (quindi 10 volte) in poco più di 5 anni. Altre leggi, meno note ma altrettanto efficaci nel "prevedere" il futuro (finora!), riguardano lo sviluppo delle tecnologie di rete e dei dischi: la Legge di Gilder afferma che la capacità di trasmissione si raddoppia ogni anno, così come il traffico di Internet, mentre per quanto riguarda la capacità di memorizzazione, la Legge di Shugard suggerisce che ogni 4 anni aumenta di un fattore 10. Infine, alcune recenti ricerche [40] sembrano confermare che lo sviluppo di Internet segua anch'esso la Legge di Moore.

Questo sviluppo continuo, a cui assistiamo da diverse decine di anni, ha condotto allo sviluppo di tecniche e metodi per lo sviluppo e la progettazione di sistemi software complessi, in grado di poter utilizzare al meglio queste componenti di sempre maggiori prestazioni. Infatti, se la tecnologia offre macchine sempre più potenti ed economiche, interconnesse sempre di più e con link sempre più veloci ed affidabili, la sfida che viene posta alle tecnologie per realizzare in tempi brevi sistemi complessi ed affidabili risulta sempre più stimolante.

I sistemi distribuiti rappresentano la strada per poter utilizzare al meglio quello che la tecnologia hardware continua a produrre, ma, spesso, si scontrano con notevoli difficoltà di progettazione, sviluppo, gestione ed evoluzione. Infatti, il costo per produrre software complesso risulta notevole, e poter manutenere e fare evolvere un sistema distribuito in linea con i requisiti degli utenti, quelli del committente e quelli della tecnologia è spesso un compito affrontato con difficoltà.

Notevoli sono le problematiche da affrontare per lo sviluppo di sistemi distribuiti. Innanzitutto, le difficoltà sorgono della complessità dell'ambiente in cui i sistemi distribuiti si trovano ad operare: ambienti dove è necessario trattare con i malfunzionamenti di parte delle componenti (riconoscimento e recovery), così come è necessario trattare con i problemi di latenza, quelli generati dall'accesso concorrente alle risorse e quelli derivanti dalla necessità di ottimizzare il carico di lavoro tra i diversi nodi. Tutte queste problematiche vanno affrontate all'interno di un panorama tecnologico in continua evoluzione (cambiamenti ed upgrade di tecnologie hardware e software) e di uno scenario economico in altrettanto rapida mutazione, che richiede la scalabilità dei servizi offerti.

Affianco a queste problematiche ci sono quelle dovute alla difficoltà della realizzazione di sistemi eterogenei e complessi, che, spesso, devono trattare con problemi di portabilità, di mancanza di sistemi evoluti di debugging e di tecniche di progettazione software che si basano su approcci non orientati ad oggetti, che non favoriscono la riusabilità e la estendibilità dei sistemi. La riusabilità e la integrabilità di soluzioni diverse rappresenta un obiettivo importante: non è sempre necessario (anzi spesso non lo è mai) "reinventare la ruota" ad ogni nuovo sistema distribuito, e la possibilità di poter integrare soluzioni pre-esistenti, nonostante le tecnologie e le tecniche di progettazioni diverse, è una priorità importante per gli ambienti di sviluppo per sistemi distribuiti.

1.2 Un modello di riferimento: Open Distributed Processing

Allo scopo di facilitare lo sviluppo dei sistemi distribuiti, è importante la condivisione di un modello comune, che serva come astrazione comune per produttori, sviluppatori e progettisti, in maniera da essere indipendente dalla specifica implementazione tecnologica. Un *modello di riferimento* serve a questo scopo, potendo essere utilizzato anche come terreno comune per la comunicazione durante le fasi iniziali di progettazione, identificando

[2]Alcune interpretazioni suggeriscono che forse 24 mesi è un limite più realistico.

termini e linguaggio da utilizzare, assolvendo, anche, all'obiettivo di permettere confronti tra diverse realizzazioni di sistemi.

Con questo obiettivo, uno sforzo importante per la standardizzazione nei sistemi distribuiti è stato effettuato dall'ISO/IEC[3] per proporre un modello formale delle architetture dei sistemi distribuiti. Questa specifica, realizzata in 4 documenti [22, 23, 24, 25], si chiama "*The Reference Model of Open Distributed Processing*" (RM-ODP) (ISO/IEC 10746-1, 2, 3, 4) ed ha come obiettivo quello di favorire la diffusione dei benefici della distribuzione di servizi di elaborazione di informazione in un ambiente di risorse e nodi eterogenei in multipli domini amministrativi di gestione.

Il modello RM-ODP si basa ed integra il modello tradizionale di rete proposto da ISO/OSI su sette layer, che mostra alcuni limiti negli strati superiori: nei sei strati inferiori si fornisce il modello delle funzionalità proprie della comunicazione, e tutti gli aspetti relativi alla distribuzione delle applicazioni all'interno del sistema distribuito sono troppo limitati dall'unico livello (il settimo) che racchiude funzionalità diverse.

In un certo senso, questo modello ha come obiettivo quello di gestire i problemi di **comunicazione** in un sistema rispetto ai problemi (più semplici) di **connessione**, che vengono trattati principalmente dal modello ISO/OSI. La standardizzazione che viene presentata da RM-ODP è motivata dalla diffusione di applicazioni e sistemi distribuiti che devono assicurare uso di tecnologie e modelli ampiamente condivisi ed aperti, flessibilità, integrazione e interoperabilità. Il modello RM-ODP non si limita, come ISO/OSI, a trattare con i problemi di comunicazione tra sistemi eterogenei, ma punta ad astrarre e standardizzare anche il concetto di portabilità e di trasparenza all'interno di un sistema distribuito. In questo senso, RM-ODP estende ed ingloba, il modello ISO/OSI, usando quest'ultimo come modalità per la comunicazione tra componenti eterogenee.

1.2.1 Le caratteristiche di un sistema distribuito

I sistemi distribuiti si caratterizzano per un discreto numero di aspetti, che vengono utilizzati per permettere di utilizzarne appieno il loro potenziale per assicurare la alta disponibilità dei servizi offerti, le prestazioni, la affidabilità e la ottimizzazione dei costi. Possiamo identificare queste caratteristiche attraverso una serie di *parole chiave* di un sistema distribuito che illustriamo di seguito.

- **Remoto.** Le componenti di un sistema distribuito devono poter essere locali o remote, quindi anche potenzialmente localizzate su macchine diverse.
- **Concorrenza.** Un sistema distribuito è per sua stessa natura concorrente, nel senso più vero del termine, in quanto la contemporanea esecuzione di due (o più) istruzioni è possibile, su macchine diverse, e non esistono strumenti come lock e semafori che sulle architetture multiprocessore (multicore), strettamente accoppiate, permettono di gestire in maniera più "semplice" la sincronizzazione.
- **Assenza di uno stato globale.** Non esiste una maniera per poter determinare lo stato globale del sistema, in quanto la distanza e la eterogeneità del sistema non permette di definire con certezza lo stato in cui si trova ciascun nodo.
- **Malfunzionamenti parziali.** Ogni componente di un sistema distribuito può smettere di funzionare correttamente, in maniera indipendente dalle altre componenti e

[3] L'ISO è un ente per la standardizzazione (International Organization for Standardization) che lavora congiuntamente con l'IEC (International Electrotechnical Commission) per definire standard nell'ambito della tecnologia elettrica, elettronica e altre tecnologie correlate, compreso le Tecnologie della Informazione e della Comunicazione (*Information and Communication Technologies*).

questo fallimento non deve inficiare le funzionalità che sono localizzate altrove nel sistema distribuito.

- **Eterogeneità.** Un sistema distribuito per sua stessa definizione è eterogeneo per tecnologia sia hardware che software. La eterogeneità si realizza in tutti i contesti: hardware, sistema operativo, rete di comunicazione, protocolli di rete, linguaggi di programmazione, applicazioni, etc.
- **Autonomia.** Un sistema distribuito non ha un singolo punto dal quale esso può essere controllato, coordinato e gestito. Quindi, la collaborazione va ottenuta mediando le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo, tramite politiche di condivisione e di accesso, formalmente specificate e rigidamente applicate.
- **Evoluzione.** I sistemi distribuiti devono assecondare la evoluzione dell'ambiente all'interno del quale vengono realizzati e forniscono le loro funzionalità. Questo significa che un sistema distribuito può cambiare anche in maniera sostanziale durante la sua vita, sia perchè cambia l'ambiente sia perchè cambia la tecnologia utilizzata. La flessibilità di un sistema distribuito deve assicurare che la migrazione verso ambienti diversi, tecnologie differenti e applicazioni nuove può essere assecondata con successo e senza costi eccessivi.
- **Mobilità.** Così come appare naturale che gli utenti siano mobili, altrettanto naturale deve essere la mobilità dei nodi e delle risorse (ad esempio, dati) all'interno del sistema in modo da poter adattare al meglio le prestazioni del sistema.

1.2.2 I requisiti non funzionali di un sistema distribuito

La realizzazione di un sistema distribuito non è agevole e comporta la necessità di considerare vari aspetti generali e globali della architettura, standardizzati in maniera tale che possano servire da specifica per i vari fornitori di piattaforme hardware e software allo scopo di fornire strumenti adeguati per rendere più agevole la progettazione, implementazione e manutenzione di un sistema distribuito.

Questi aspetti specifici di un sistema distribuito non hanno a che fare direttamente con le funzionalità che esso deve realizzare, cioé non fanno parte dei requisiti funzionali del sistema, identificati durante la fase di analisi dei requisiti che precede la fase di analisi, progettazione e implementazione. Hanno invece a che fare con i cosiddetti requisiti non funzionali, che indicano essenzialmente la qualità del sistema e non sono identificabili in una specifica parte del sistema, ma sono globali e vanno considerati come fattori che hanno un impatto significativo sulla architettura.

Questi requisiti non funzionali specificano che la progettazione deve puntare a realizzare sistemi distribuiti che:

- ... siano **aperti**, in modo da supportare la portabilità di esecuzione e di interoperabilità (capacità di collaborare insieme tra diverse componenti) attraverso interfacce e servizi ben documentati ed aderenti a standard noti e riconosciuti; questo aspetto risulta importante per poter far evolvere il sistema (si possono aggiungere nuove componenti al bisogno) ma anche per evitare di rimanere legati ad un singolo fornitore: se si usano standard aperti, si può cambiare fornitore senza particolari rischi per la intera architettura (che può essere riutilizzata ed integrata).
- ... siano **integrati**, così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera

efficiente (economica) con il problema della eterogeneità hardware, software e di applicazioni.

- ... siano **flessibili**, per poter evolvere e fare evolvere i sistemi distribuiti in maniera da integrare sistemi *legacy* al proprio interno. Un sistema distribuito dovrebbe anche poter gestire modifiche durante la esecuzione in modo da poter accomodare cambi a run-time, riconfigurandosi dinamicamente.

- ... siano **modulari**, in modo da permettere ad ogni componente di poter essere autonoma ma con un grado di interdipendenza verso il resto del sistema.

- ... supportino la **federazione** di sistemi, in modo da unire diversi sistemi, dal punto di vista amministrativo oltre che architetturale, per lavorare e fornire servizi in maniera congiunta.

- ... siano **facilmente gestibili**, in modo da permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso.

- ... forniscano supporto per la **qualità del servizio** (*Quality of Service*), per poter fornire i servizi con vincoli di tempo, di disponibilità e di affidabilità, anche in presenza di malfuzionamenti parziali, che in un sistema distribuito devono sempre essere considerati possibili ed inevitabili. La **tolleranza ai malfunzionamenti** è una delle principali richieste di qualità del servizio di un sistema distribuito, in quanto i sistemi centralizzati sono particolarmente poco tolleranti ai malfunzionamenti, che possono rendere l'intero sistema inutilizzabile. Un sistema distribuito, invece, è potenzialmente in grado di trattare con i malfunzionamenti, utilizzando (dinamicamente) componenti alternative per fornire funzionalità che alcune componenti non sono in grado temporaneamente di fornire.

- ... siano **scalabili**, perché qualsiasi sistema distribuito accessibile da Internet può essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli; ma si deve anche poter gestire (anche attraverso la flessibilità) che il sistema possa evolvere per accomodare evoluzioni del contesto aziendale che può crescere velocemente, aumentando notevolmente la platea di utenti che accedono ai servizi forniti dal sistema.

- ... siano **sicuri**, così che utenti non autorizzati non possano accedere a dati sensibili. La sicurezza è ovviamente particolarmente complicata dalla natura remota dei sistemi distribuiti e della mobilità degli utenti, nodi e risorse al proprio interno.

- ... offrano **trasparenza** mascherando i dettagli e le differenze della architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema. Questa caratteristica risulta centrale per poter permettere la agevole progettazione ed implementazione: il progettista/programmatore deve avere un certo grado di indipendenza dai dettagli della distribuzione della architettura. È però anche vero che questa trasparenza non deve essere completa e che progettisti/programmatori debbano essere coscienti della natura distribuita di alcune caratteristiche (maggiori dettagli su questa limitazione alla trasparenza, per gli oggetti distribuiti, vengono forniti nel Capitolo 3.5.2).

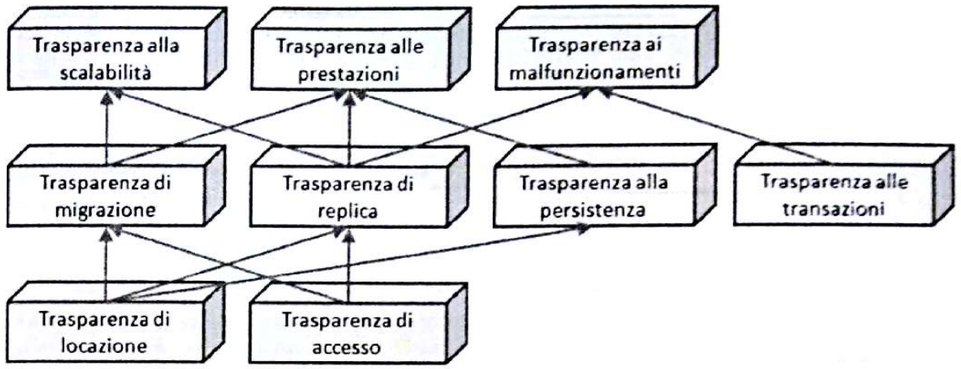

Figura 1.1: *Le dipendenze tra i vari tipi di trasparenza.*

1.2.3 La trasparenza di un sistema distribuito

La definizione che abbiamo dato in questo capitolo di sistema distribuito sottolinea un aspetto importante: i dettagli del sistema che offre le funzionalità operative sono nascosti agli utenti. Questa è la definizione della *trasparenza* di un sistema, che non permette di identificare le singole parti ma viene visto come una unica entità.

Questo permette al progettista/sviluppatore di lavorare in un ambiente che non fornisce informazioni specifiche sulla architettura del sistema: il progettista/sviluppatore ignora, ad esempio, la eterogeneità delle componenti, i fallimenti indipendenti a cui sono soggetti le componenti, la esistenza e la posizione dei diversi nodi che offrono lo stesso servizio e da quale in questo momento esso viene servito e così via. Il progettista/sviluppatore richiede al sistema di fornire un certo tipo ed un certo livello di trasparenza su una certa parte della sua applicazione, attraverso una richiesta al sistema[4].

Ovviamente, il primo vantaggio della trasparenza è la maggiore produttività del lavoro di sviluppo: il progettista e lo sviluppatore possono concentrare il loro lavoro sulla applicazione, utilizzando un sistema "astratto" di cui ignorano i dettagli. Questo si riflette sulla velocità di prototipizzazione e sulla economia della produzione del software. Ma la trasparenza serve anche a permettere un alto riuso delle applicazioni sviluppate che, proprio perché sviluppate nella totale trasparenza dei dettagli sottostanti, possono essere riutilizzate in contesti diversi e su sistemi diversi.

La trasparenza che viene fornita da un sistema distribuito ricade in diverse tipologie, strettamente collegate e dipendenti l'una dall'altra, come illustrato in Figura 1.1 (la schematizzazione che segue integra la presentazione fornita da Emmerich [8]).

Trasparenza di accesso

La trasparenza di accesso nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere la interoperabilità tra oggetti. Questo significa anche che gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto. In questa maniera, un oggetto può essere facilmente spostato a run-time da un nodo ad un altro.

[4]Questo meccanismo è quello che abbiamo chiamato di middleware implicito, in cui la trasparenza viene ottenuta richiedendola al momento del deployment attraverso i metadati di descrizione.

In generale, questo tipo di trasparenza viene fornito di default dai sistemi, in quanto è il tipo di trasparenza necessario per assicurare la interoperabilità in un ambiente eterogeneo.

Trasparenza di locazione

La trasparenza di locazione non permette di utilizzare informazioni circa la localizzazione nel sistema di una particolare componente, che viene identificata ed utilizzata in maniera indipendente dalla sua posizione. Questo tipo di distribuzione fornisce una vista logica del sistema di *naming*, in modo da disaccoppiare il nome da una posizione all'interno della rete.

Anche questo tipo di trasparenza è fondamentale per un sistema distribuito, in quanto senza di esso non si potrebbe spostare componenti da un nodo ad un altro, poiché essi potrebbero essere riferiti secondo la loro posizione e non attraverso un meccanismo di naming logicamente disconnesso dalla locazione.

Trasparenza di migrazione

Il compito di questo tipo di trasparenza è quello di nascondere la possibilità che il sistema faccia migrare un oggetto da un nodo ad un altro, continuando ad essere raggiungibile ed utilizzabile da altri oggetti. Questo viene utilizzato per ottimizzare le prestazioni del sistema (bilanciando il carico tra i nodi oppure riducendo la latenza per accedere ad una componente) o anche per anticipare malfunzionamenti o riconfigurazioni del sistema[5] ma deve essere gestito in maniera automatizzata da parte di parti della infrastruttura del sistema.

La trasparenza di migrazione dipende dalla trasparenza di accesso (che permette di accedere ad un oggetto, anche se locale, solo attraverso la propria interfaccia che viene usata da remoto) e dalla trasparenza di locazione (che nasconde la locazione fisica di un oggetto, permettendone l'accesso attraverso un sistema di naming logico fornito dal sistema).

Trasparenza di replica

Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata da un certo numero di copie (dette *repliche*) che vengono posizionate su altri nodi del sistema, e che offrono esattamente lo stesso tipo di servizio della componente originale. Ovviamente, il sistema si deve occupare di mantenere assolutamente coerente lo stato di tutte le repliche con la componente originale, in maniera tale da rispettare la semantica delle operazioni che vengono compiute sulla componente e dei servizi offerti. Anche questo tipo di trasparenza dipende da quella di accesso e di locazione.

Le repliche vengono utilizzate per diversi scopi. Innanzitutto, per le prestazioni, facendo in modo di replicare componenti laddove (all'interno del sistema) maggiori sono le richieste per quel tipo di servizi, in modo da minimizzare la latenza per accedervi[6]. Ma vengono anche utilizzate per poter far scalare il sistema in presenza di aumento del carico di lavoro.

[5] Ad esempio, se un nodo deve essere spento per aggiornamenti software, gli oggetti che si trovano sul nodo vengono spostati altrove, mantenendo lo stesso nome e tenendo attivi i servizi che stanno fornendo.

[6] Un meccanismo simile a quello che viene realizzto dai cosiddetti Content Delivery Network, tipo Akamai, per il World Wide Web, che spostano parte del contenuto HTML di siti con un alto carico su dei proxy che si trovano nelle reti dei principali Internet Provider regionali, per fornire contenuto vicino all'utente finale.

Trasparenza alle transazioni

Un sistema distribuito è implicitamente concorrente, in quanto la esistenza di diverse risorse accessibili da diversi nodi rende la concorrenza la regola e non la eccezione. La trasparenza alle transazioni (anche chiamata trasparenza alla concorrenza) nasconde all'utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello stato degli oggetti in presenza della concorrenza. Sia l'utente che il progettista e lo sviluppatore sono ignari delle attività che vengono svolte per assicurare la atomicità delle operazioni e possono semplicemente ritenersi gli unici utenti all'interno del sistema.

La gestione delle transazioni distribuite è un compito non semplice e la semplificazione che si ottiene lasciandola al sistema è davvero notevole per lo sviluppatore di applicazioni. La possibilità di poter operare in maniera transazionale una operazione è anche cruciale per assicurare che in presenza di malfunzionamenti una risorsa non si trovi in uno stato non coerente.

Trasparenza ai malfunzionamenti

La trasparenza ai malfunzionamenti nasconde ad un oggetto il malfunzionamento (e, se è il caso, il successivo recovery) di oggetti con i quali sta interoperando. La trasparenza si estende ovviamente sia agli utenti del sistema che non devono avere la sensazione di malfunzionamenti parziali all'interno del sistema, in quanto il sistema deve automaticamente riconfigurare la richiesta e fornire il servizio in maniera alternativa.

Questo tipo di trasparenza si poggia, ovviamente, sulla trasparenza di replica, in quanto quest'ultima fornisce la possibilità di poter ripetere trasparentemente le operazioni che si erano iniziate su una replica di un oggetto, potendo rieseguirle su una altra replica. Ma si basa anche sulla trasparenza alle transazioni, in quanto operazioni complesse eseguite come una transazione, se interrotte a causa di un malfunzionamento non vengono confermate (*commit*) e quindi non alterano lo stato della risorsa e possono essere ripetute su una replica.

Trasparenza alla persistenza

Questo tipo di trasparenza scherma l'utente dalle operazioni che compie il sistema per rendere persistente (cioé in memoria secondaria) un oggetto durante una fase di non utilizzo.

Infatti, per ottimizzare le prestazioni del sistema, gli oggetti di utilizzo raro non vengono mantenuti attivi (nello spazio di indirizzamento della memoria principale) ma vengono de-attivati, e memorizzati (con il loro stato) all'interno della memoria secondaria, mantenendo solamente un *handle* per la loro riattivazione, quando arrivano richieste di operazioni da eseguire. In questo caso, l'oggetto viene riportato in memoria principale e reso attivo per rispondere alla richiesta. Gli oggetti che invocano servizi su un oggetto de-attivato non avvertono la differenza con le invocazioni su un oggetto attivo.

La trasparenza alla persistenza si basa, ovviamente, sulla trasparenza di locazione, in quanto l'accesso indipendente dalla posizione fisica dell'oggetto permette una riattivazione dell'oggetto anche su nodi diversi da quelli su cui era stato de-attivato.

Trasparenza alla scalabilità

La scalabilità è uno dei principali motivi a favore di un sistema distribuito rispetto ad uno centralizzato. Un sistema viene detto scalabile quando è in grado di poter servire carichi di lavoro via via crescenti senza dover modificare la propria architettura e la propria organizzazione.

Progettare un sistema scalabile è necessario, oggigiorno, visto che la platea di utenti ai quali potenzialmente un servizio su Internet viene offerto è smisurata. Un servizio deve potenzialmente poter scalare dalle poche decine alle centinaia di migliaia, o ai milioni di utenti senza che questo comporti la riprogettazione dell'intero sistema. Ovviamente, si dovranno acquisire nuove risorse, ma il sistema dovrà essere in grado di poterle utilizzare senza modifiche sostanziali.

La trasparenza alla scalabilità assicura che il progettista/sviluppatore non deve curarsi dei dettagli di come il proprio servizio scalerà al crescere delle richieste, ma sarà il sistema che provvederà, attraverso il meccanismo di replica e di migrazione, a fare in modo che le nuove risorse aggiunte al sistema vengano utilizzate per fare fronte al carico crescente.

Trasparenza alle prestazioni

Abbastanza simile alla trasparenza alla scalabilità è la trasparenza alle prestazioni. Il sistema distribuito che assicura questo tipo di trasparenza rende il progettista/sviluppatore ignaro dei meccanismi che vengono utilizzati per ottimizzare le prestazioni del sistema, durante la fornitura di servizi. In particolare, il sistema può provvedere ad implementare politiche di bilanciamento del carico, spostando componenti da nodi carichi di lavoro verso nodi che hanno maggiori disponibilità di calcolo a disposizione, oppure politiche di minimizzazione della latenza, avvicinando (repliche di) componenti su nodi più vicini (in termini di topologia di rete) agli utenti che li usano più frequentemente, oppure politiche di ottimizzazione delle risorse di memoria, che prevedono la inattivazione di oggetti che non vengono usati frequentemente e che possono essere re-attivati se necessario. Per questo motivo, la trasparenza alle prestazioni si appoggia sulla trasparenza alla migrazione, alla replica ed alla persistenza.

In conclusione, la trasparenza offerta da un sistema viene definita durante la fase di progettazione e viene richiesta sia per l'intero sistema sia in maniera selettiva solo su alcune componenti, alcune interfacce o alcuni servizi, definendo, per esempio, quali operazioni devono essere eseguiti come transazioni, oppure quali oggetti devono essere replicati dal sistema in dipendenza delle ottimizzazioni richieste per aumentare le prestazioni e la scalabilità del sistema.

Va anche sottolineato che offrire la trasparenza (di tutte le tipologie) ha un costo associato sul sistema, per gestire tutte le operazioni e le situazioni che vanno nascoste dall'utente (utente finale, progettista, sviluppatore, etc.). Quindi la scelta di quali tipi di trasparenza utilizzare ed implementare come servizi per gli sviluppatori richiede una attenta valutazione di costi/benefici. Alcuni tipi di trasparenza, comunque, sono considerati talmente importanti da dover essere sempre presenti come, ad esempio, quelli di locazione e di accesso.

1.2.4 I diversi punti di vista (*viewpoints*) per un sistema distribuito

Un punto di vista (*viewpoint*) per un sistema distribuito è una astrazione che specifica solamente una parte delle caratteristiche del sistema, relative ad uno specifico ambito di interesse. Per i sistemi distribuiti, vengono definiti 5 punti di vista diversi, che coprono diversi aspetti della progettazione delle architetture, di seguito illustrati.

- Il punto di vista *enterprise*, focalizzato sullo scopo, ambito e politiche che governano le attività del sistema distribuito, visto come una parte della organizzazione a cui appartiene.

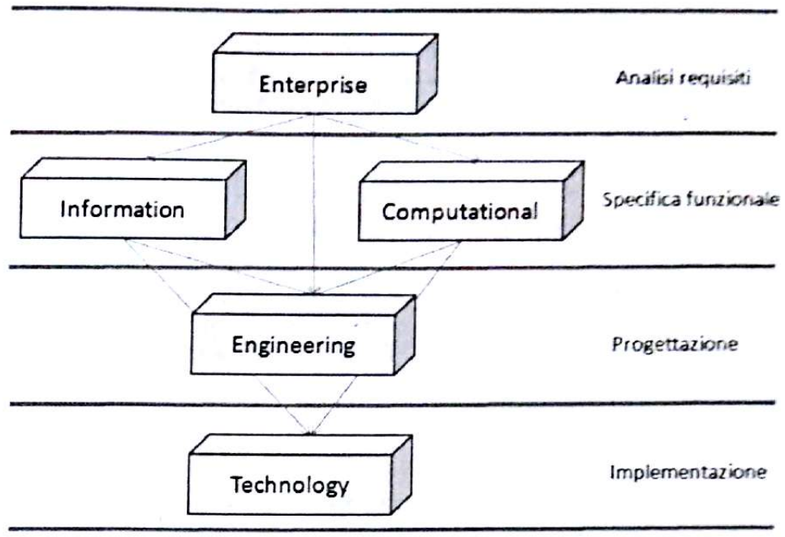

Figura 1.2: *Un esempio di come i diversi punti di vista del modello RM-ODP possono essere utilizzati per specificare il procedimento di realizzazione di un sistema distribuito.*

- Il punto di vista *information*, che si concentra sul tipo di informazioni che vengono gestite dal sistema e sui vincoli sull'uso e la interpretazione di queste informazioni.
- Il punto di vista *computational*, che riguarda la decomposizione della funzionalità che il sistema deve offrire su un insieme di oggetti (o componenti) che interagiscono attraverso le interfacce (che definiscono i servizi offerti) per permettere la distribuzione all'interno della architettura.
- Il punto di vista *engineering*, che si concentra sulla infrastruttura che è necessaria per supportare la distribuzione dei servizi nel sistema.
- Il punto di vista *technology*, che tratta delle scelte tecnologiche per supportare la distribuzione delle funzionalità all'interno del sistema.

È importante notare che i punti di vista non rappresentano dei layer e possono essere interconnessi in maniere diverse per poter illustrare le fasi di realizzazione di un sistema distribuito. Una possibilità viene illustrata in Figura 1.2, dove le fasi di specifica funzionale sono condotte utilizzando i punti di vista information e computational, confluendo poi nel punto di vista engineering per la progettazione dell'intero sistema, che prende input anche dalla analisi dei requisiti specificati utilizzando il linguaggio del punto di vista enterprise.

Il punto di vista *enterprise*

Con questo punto di vista si cerca di definire gli obiettivi del sistema, definendo lo scopo e il contesto di riferimento dal punto di vista della "azienda"[7], definendo quindi i vincoli, i permessi e le politiche che governano le relazioni del sistema sia con il contesto aziendale (che è esterno al sistema ma interno alla azienda) che con l'esterno dell'azienda.

[7] Ovviamente, il termine utilizzato è generico, in quanto possiamo riferirci ad istituzioni pubbliche, private, no-profit, nazionali o transnazionali di qualsiasi dimensione.

In questo punto di vista, si definiscono internamente i ruoli degli utenti del sistema ed il loro comportamento, le politiche di accesso, e le federazioni che servono a definire le relazioni tra sistemi informativi di aziende che decidono di collaborare.

Questo punto di vista, di solito, fornisce specifiche che saranno utilizzate all'interno di altri punti di vista (come quello information e computational) dove verranno definiti i meccanismi per realizzare nel sistema le specifiche definite nel punto di vista enterprise.

Il punto di vista *information*

Questo punto di vista definisce la semantica delle informazioni e delle computazioni all'interno del sistema distribuito, utilizzando tre schemi: lo schema di invarianza, quello statico e quello dinamico.

Lo schema di invarianza definisce le condizioni che devono essere sempre rispettate all'interno del sistema per un dato pezzo di informazione, di solito rappresentata dallo stato di un oggetto. Lo schema statico definisce le condizioni che sono rispettate dopo che un certo evento si è verificato. Lo schema dinamico definisce le modifiche lecite a cui lo stato di un oggetto può essere soggetto, specificando un insieme di precondizioni, che devono essere rispettate affinchè la transizione abbia luogo, e di postcondizioni, che definiscono ciò che devono essere verificate alla fine della transizione. Specifica anche le operazioni di creazione e distruzione delle componenti.

Il punto di vista *computational*

Questo punto di vista riguarda la distribuzione delle computazioni ma non il meccanismo che ne permette la distribuzione. Quindi, si occupa di decomporre il sistema in oggetti che realizzino funzionalità specifiche e di definire le modalità di interazione tra oggetti attraverso un insieme di interfacce definite a tale scopo. Fornisce, in pratica, le basi per le decisioni su come distribuire il lavoro da eseguire (perché le interfacce vengono localizzate indipendentemente) potendo contare sui meccanismi di comunicazione che verranno specificate dal punto di vista engineering.

Il modello degli oggetti di RM-ODP definisce la forma della interfaccia (o delle interfacce) che un oggetto può avere, la maniera in cui interagiscono tra di essi, e le azioni che un oggetto può eseguire, in particolar modo la creazione di nuovi oggetti ed interfacce.

Le interazioni tra oggetti possono essere di tre tipi: le operazioni (simili alle invocazioni di procedure), i *flows* (che sono astrazioni degli stream di comunicazione) ed i *signals* (operazioni atomiche elementari).

Le interazioni tra oggetti hanno luogo solamente se c'è stato un *binding* tra di essi, in modo che si possa comunicare effettivamente da uno verso l'altro. Esempi sono il binding che ha luogo tra un oggetto che rappresenta un file ed il file, attraverso una operazione di *open*, oppure il binding che è necessario per permettere la invocazione di un metodo di un oggetto da parte di un altro oggetto.

Il punto di vista *engineering*

Questo punto di vista ha come obiettivo quello di occuparsi della maniera in cui si ottiene la interazione tra oggetti e sulle risorse necessarie per assicurare tale interazione. Descrive la infrastruttura necessaria per fornire meccanismi trasparenti di distribuzione e le regole per definire canali di comunicazione tra oggetti. Così come il punto di vista computational si occupa di definire quando e perché gli oggetti interagiscono, questo punto di vista definisce sul come essi interagiscono.

Il linguaggio del punto di vista engineering definisce anche la architettura del sistema utilizzando i termini di *nodo* (per indicare la singola macchina, indipendentemente gestita e manutenuta), di *capsula* (che definisce un tradizionale processo del sistema operativo, caratterizzato dall'avere un proprio spazio di indirizzamento) e di *cluster* (che definisce un insieme di oggetti che sono soggetti alle stesse operazioni di gestione, come attivazione, deattivazione, trasferimento, persistenza, etc.).

Il punto di vista *technology*

La enfasi, in questo caso, viene posta sulla implementazione del sistema distribuito, in termini della configurazione della tecnologia hardware e software. È ovviamente vincolata dal costo economico e dalla disponibilità della tecnologia corrente sulla base dei requisiti del sistema. Le specifiche hardware, ad esempio, possono definire il tipo, la potenza di calcolo ed il numero di processori da usare, mentre le specifiche software definiscono i sistemi operativi, gli ambienti di sviluppo e le infrastrutture software che sono necessarie. Ovviamente, al modificarsi degli standard, delle tecnologie o dei requisiti del punto di vista enterprise, vengono modificate anche le scelte effettuate. Infatti, in questo punto di vista si cerca di concentrare tutte le dipendenze dalla tecnologia dell'intero sistema, in modo da non dover modificare consistentemente le decisioni prese utilizzando gli altri 4 punti di vista.

1.3 Il Middleware ad Oggetti Distribuiti

I sistemi distribuiti basati su Oggetti Distribuiti sono uno degli strumenti utilizzati dai sistemi distribuiti per assicurare estendibilità, affidabilità e scalabilità, rendendo minimo lo sforzo per progettare, sviluppare e manutenere sistemi complessi. Gli oggetti distribuiti si trovano alla confluenza di due aree della tecnologia software: i *sistemi distribuiti* che puntano a realizzare un unico sistema integrato basato sulle risorse offerte da diversi calcolatori messi in rete, e lo *sviluppo e la programmazione orientata agli oggetti*, che si focalizzano sulle modalità per ridurre la complessità dei sistemi software, creando artefatti software riutilizzabili in diversi contesti.

Gli oggetti distribuiti, quindi, hanno come obiettivo quello di realizzare servizi distribuiti riutilizzabili, in maniera tale che siano efficienti, flessibili, sicuri e robusti. Il tutto basato su una architettura che utilizza come risorse nodi eterogenei, sia per hardware che per software, raggiungibili attraverso una rete.

Questa integrazione viene realizzata attraverso il *middleware ad oggetti distribuiti*, che risiede tra le applicazioni (che si trovano in alto, verso l'utente) e lo strato del sistema operativo, stack di protocolli di rete e hardware (che si trovano in basso verso l'hardware) con l'obiettivo di permettere alle componenti del sistema di cooperare e comunicare.

È chiaro che la comunicazione attraverso i nodi potrebbe avvenire attraverso le primitive di comunicazione di rete che vengono offerte dal sistema operativo di ogni singolo nodo. Questo però, in pratica, risulta essere troppo complesso (e costoso in termini di tempo e risorse economiche) per la programmazione di applicazioni, in quanto i programmatori dovrebbero prendersi cura di tutti i dettagli di basso livello, come quello di trasformare le strutture dati del livello applicazione in stream di byte oppure datagram che possono essere trasmessi su rete, risolvendo in prima persona tutti i problemi di eterogeneità della rappresentazione dei dati su differenti architetture hardware.

Lo scopo del middleware è quello di rendere semplici questi compiti e di fornire delle astrazioni appropriate per i programmatori, che ben si integrino con i tradizionali strumenti che essi utilizzano per realizzare la applicazione.

Il middleware ad oggetti distribuiti può essere suddiviso in tre strati:

- *Middleware di infrastruttura*. Questo layer si occupa delle comunicazioni tra sistemi operativi diversi e della gestione della concorrenza per evitare lo sforzo di utilizzare meccanismi non portabili (dipendenti dalla singola piattaforma hardware/software di ogni nodo) per sviluppare e manutenere applicazioni distribuite. Un esempio di questo tipo di infrastruttura può essere quella della stessa macchina virtuale Java.

- *Middleware di distribuzione*, che basa i suoi servizi sul middleware di infrastruttura per automatizzare operazioni comuni per la comunicazione. Tra i compiti più importanti ci sono, ad esempio, quelli di:

  - richiedere un servizio ad un altro nodo potendo inviare parametri (*marshalling*). Questo compito non è banale in quanto si deve realizzare l'invio di parametri tra diverse piattaforme hardware/software, ed, oltre ai problemi di eterogeneità della rappresentazione di tipi primitivi (tipicamente affrontata dal layer di infrastruttura) si deve poter inviare anche dati complessi (oggetti) la cui definizione (classe) può anche non essere a disposizione della macchina che riceve la invocazione del servizio;
  - utilizzare lo stesso canale di comunicazione (socket, ad esempio) per diverse richieste, oppure utilizzare una sola macro-richiesta che include diverse richieste;
  - modificare la semantica delle operazioni di invocazione oltre quella tradizionale di unicast, quale ad esempio la invocazione in multicast (invocazioni su diversi oggetti contemporaneamente) oppure la attivazione di oggetti in risposta ad invocazione di servizi;
  - riconoscimento e gestione dei malfunzionamenti di rete, attuando strategie per permettere che la applicazione possa effettuare il recovery di uno stato semanticamente corretto della applicazione.

- *Middleware per servizi comuni di supporto*, un layer che serve a fornire i servizi comuni a tutte le applicazioni distribuite e, quindi, riutilizzabili in tutti i contesti, quali, ad esempio, la persistenza degli oggetti (cioè il loro collegamento automatico ad un sistema di gestione di database), la sicurezza (con gestione di autenticazione e di politiche di accesso), la gestione delle transazioni (assicurando, ad esempio, la atomicità di operazioni che sono effettuate in un contesto altamente concorrente, quale i sistemi distribuiti).

L'obiettivo di questi tre livelli di middleware è quello di assicurare, innanzitutto, che il programmatore di applicazioni concentri i propri sforzi sullo sviluppo della logica di business della applicazione e che non si debba interessare direttamente dei dettagli di comunicazione a livello di rete, il che consuma risorse, tempo e richiede competenze specifiche per evitare implementazioni che risultino, alla lunga, non efficienti e non efficaci. Poi, forniscono astrazioni utili per il programmatore, ben integrate nell'ambiente di sviluppo della applicazione[8]. Infine, proprio perchè il middleware ad oggetti astrae significativamente la parte di distribuzione e di servizio delle applicazioni distribuite, lo sviluppo avviene ad alto livello, permettendo di poter utilizzare e riutilizzare framework e soluzioni, appoggiandosi a metodologie evolute di ingegneria del software per rendere maggiormente proficua, efficiente e efficace la soluzione realizzata.

[8] Esempi tipici, come RMI oppure .NET, si integrano perfettamente nel ciclo di sviluppo di applicazioni basati su Java oppure su ambienti Microsoft .NET, permettendo al programmatore di integrare la distribuzione delle proprie applicazioni in maniera naturale, trasparente e congrua all'ambiente in uso.

1.3.1 Il progenitore: Remote Procedure Calls (RPC)

La computazione distribuita basata su oggetti distribuiti si basa sulla astrazione fornita da *Remote Procedure Call* (RPC), un modello presentato negli anni '80 che permetteva ad una procedura in esecuzione su una macchina di invocarne un'altra che si trovasse su una macchina diversa. RPC realizzava la invocazione in maniera agevole per il programmatore, come se fosse stata una tradizionale chiamata di procedura fatta in locale. RPC è stata la prima tecnologia a dover risolvere problemi significativi e complessi di eterogeneità e di concorrenza. Infatti, in RPC è stato definito per la prima volta il meccanismo di invocazione remota che richiede la traduzione dei tipi di dato a livello applicazione (usati come parametri e come risultati della invocazione di procedura remota) in maniera tale che:

- fosse possibile trasmettere i dati utilizzando stream di byte su socket, in maniera codificata e standardizzata in modo che all'altro capo della comunicazione i dati fossero ricostruiti come erano stati trasmessi (*marshalling*);

- si superassero le differenza nella rappresentazione stessa dei dati da trasmettere, quali ad esempio, la rappresentazione degli interi (con le differenze hardware tra architetture big-endian o little-endian o con le diverse rappresentazioni interne di diversi linguaggi di programmazione o di diversi sistemi operativi) o la rappresentazione delle stringhe di caratteri (che possono essere codificati tramite diversi standard, quali, ASCII e EBCDIC) (*data representation*);

Inoltre, RPC imponeva che fosse rispettata la *sincronia* della invocazione, bloccando il client (processo che invoca il metodo remoto) fino a quando il server (processo che ospita la procedura invocata) non avesse risposto alla invocazione remota di procedura. Questa modalità di invocazione sincrona permetteva al programmatore di rimanere all'interno della tradizionale semantica di chiamata a procedura locale, permettendo di mutuare lo stile di programmazione e le tecniche utilizzate finora in ambito distribuito.

Le operazioni di sincronizzazione, marshalling, data representation e comunicazione tra client e server venivano implementate dai sistemi di RPC attraverso gli *stub*, lato client e lato server che interfacciavano il processo chiamante con il processo che offriva la procedura, fornendo al programmatore una astrazione che facilitava il suo compito di realizzare la applicazione distribuita.

La scrittura degli stub venne poi automatizzata, utilizzando strumenti per creare stub a partire dalla definizione del servizio offerto attraverso un linguaggio specifico, chiamato *Interface Definition Language* (IDL), per il quale esistevano le corrispondenze in ciascun linguaggio di programmazione supportato dallo specifico ambiente.

Per terminare questa rapida carrellata su RPC, va ricordato che esso viene ancora utilizzato per sistemi critici per il funzionamento di Internet, quali DHCP e DNS, e che RCP viene incluso in numerosissimi e diversi sistemi operativi, a partire da quelli Microsoft, a tutti le distribuzioni di Linux e alle versioni di Mac OS X.

1.3.2 Da RPC al Middleware ad Oggetti Distribuiti

Gli oggetti distribuiti sono un modello presentato negli anni '90 che unisce la tecnologia software della programmazione ad oggetti con quella dei sistemi distribuiti: il modello RPC viene esteso in maniera da permettere l'invocazione di metodi di oggetti remoti. Tale estensione non rappresenta, però, semplicemente un aggiornamento della chiamata a procedure per oggetti, in quanto il modello viene esteso integrando al proprio interno le caratteristiche proprie del modello di programmazione ad oggetti come polimorfismo,

ereditarietà e gestione delle eccezioni. In un certo senso, il passaggio che ha portato dal modello RPC al modello ad oggetti distribuiti può essere visto come un ulteriore passo della tecnologia dei linguaggi di programmazione nel cammino verso l'incapsulamento, la modularità e la astrazione, che parte dalla proposta del linguaggio Pascal fatta da Niklaus di Wirth fino ai giorni nostri.

Il passaggio da Remote Procedure Call agli Oggetti Distribuiti è stato motivato dalla evoluzione e dallo sviluppo dei sistemi distribuiti complessi, che, al crescere della disponibilità delle risorse hardware e di comunicazione, diventano sempre più ampi, complessi, eterogenei e difficile da gestire e da manutenere. In effetti, la tecnologia degli oggetti distribuiti ha rappresentato, negli anni '90, la chiave di volta per realizzare sistemi distribuiti *eterogenei* che fossero *scalabili*, *tolleranti ai malfunzionamenti*, *estendibili* e di *agevole gestione*.

A queste esigenze risponde il modello ad oggetti distribuiti, che si basa sulle tecniche della programmazione ad oggetti per offrire ambienti e artefatti riutilizzabili e componenti che si integrano all'interno di architetture software e design pattern. Il risultato è che gli ambienti di programmazione distribuita basati su oggetti distribuiti forniscono uno strumento versatile e efficiente tramite il quale le invocazioni di metodi offerti da oggetti vengano effettuate attraverso la rete, utilizzando macchine diverse che possono anche avere diverse caratteristiche hardware e software (ad esempio, architetture little-endian e big endian, oppure differenze di sistema operativi).

1.3.3 Esempi rappresentativi: CORBA, Java RMI e .NET

Tre sono storicamente le (famiglie di) soluzioni basate su oggetti distribuite, sviluppate negli ultimi 18 anni: CORBA, Java RMI e .NET Remoting.

Storicamente, la prima (1991) proposta significativa di ambiente di programmazione basato su oggetti distribuiti è stato Common Object Request Broker Architecture (CORBA), proposto dalla Object Management Group (OMG). CORBA è uno standard che permette ad oggetti distribuiti, scritti in diversi linguaggi e quindi eterogenei, di comunicare e collaborare per realizzare una applicazione distribuita. Tutta la architettura di CORBA si basa sulla invocazione di un servizio (metodo) su un oggetto distribuito. I servizi sono definiti in termini astratti (cioé specificandone solamente il comportamento) da interfacce scritte in Interface Definition Language, un linguaggio specifico di CORBA. Le implementazioni degli oggetti remoti, invece, possono essere in C, C++, Java, ed altri linguaggi per i quali esista il *binding* con CORBA. Le invocazioni remote vengono realizzate attraverso il servizio fornito dall'Object Request Broker (ORB) che astrae il meccanismo di invocazione in maniera completamente trasparente al client. CORBA rappresenta una soluzione completa di Middleware ad Oggetti, curando sia la parte di infrastruttura, che quella di distribuzione e quella di servizi di supporto, e si è diffuso principalmente negli anni '90 come architettura di riferimento. In seguito, ha scontato diversi problemi [20] tra cui la sua complessità (dovuta anche agli ambiziosi obiettivi di integrazione e di servizi che si proponeva) ma anche la mancanza di interoperabilità[9] tra ORB diverse. L'effetto è che CORBA ha dovuto lasciare il passo nei sistemi distribuiti a soluzioni basate sul modello a componenti, o Enterprise computing, o a soluzioni basate su architetture orientate a servizi (*Service Oriented Architecture*).

[9]CORBA è stato progettato da un gruppo di produttori di software, attraverso un processo che ha provveduto ad una standardizzazione a volte espressa in termini vaghi (in modo che ogni produttore avesse la libertà di interpretarla) e che avesse il vincolo di doversi necessariamente basare su implementazioni reali prototipali (le cosiddette *reference implementation*); di conseguenza, alcune specifiche non erano sufficientemente dettagliate e realistiche da essere realizzabili e/o interoperabili tra diverse implementazioni.

Java Remote Method Invocation è stata la proposta di Sun, interna all'ambiente Java, per realizzare applicazioni distribuite basate su oggetti. RMI è realizzato in Java ed eredita numerose caratteristiche che ne hanno reso l'utilizzo diffuso, dapprima, per realizzare applicazioni distribuite e, poi, come strumento di comunicazione di base per Java Enterprise Edition. Principalmente, Java RMI definisce il Middleware di distribuzione all'interno della piattaforma Java, integrandosi, però, con altre librerie di Java per fornire i servizi comuni di supporto.

Microsoft .NET Framework è la soluzione di Microsoft per la realizzazione di applicazioni ed include una ampia libreria di soluzioni ed un ambiente di esecuzione chiamato Common Language Runtime (CLR) in modo che applicazioni possano essere scritte in uno dei linguaggi supportati da Microsoft (C#, Visual Basic, etc. ma anche linguaggi funzionali come F#). .NET Remoting è una tecnologia Microsoft per realizzare comunicazione tra processi basati su oggetti distribuiti, erede delle tecnologie Microsoft DCOM e COM+. È stata presentata in .NET Framework 2.0 ed è ora inclusa in Windows Communication Foundation[10] (WCF) all'interno di .NET Framework 3.0. Questa tecnologia si occupa anch'essa principalmente della parte di distribuzione, mentre la parte di Middleware di infrastruttura è assicurato dal CLR, e la parte di Middleware di servizi viene realizzata da altre librerie di .NET.

1.4 Il Middleware ad Oggetti Distribuiti nel Modello a Componenti

Le tecnologie basate sugli oggetti distribuiti che abbiamo presentato hanno rappresentato un passo avanti per la progettazione e realizzazione di sistemi affidabili e scalabili. Il loro obiettivo di assicurare le funzionalità e l'efficienza di un sistema ad oggettti distribuiti in un ambiente eterogeneo[11] è stato ottenuto con tecniche ed approcci diversi ma che hanno consentito lo sviluppo e la diffusione dei sistemi distribuiti.

In maniera significativa, le tecnologie sviluppate e l'esperienza maturata (sia in termini positivi che in termini negativi) hanno fortemente influenzato lo sviluppo delle tecnologie basate sul Modello a Componenti Distribuite (anche detto *Enterprise Computing*). Una Componente Distribuita [28] è un blocco riutilizzabile di software che può essere combinato in un sistema distribuito per realizzare funzionalità. All'interno di una componente risiedono servizi e applicazioni che espongono tramite una interfaccia le proprie funzionalità.

Quello che caratterizza e differenzia le componenti da altri moduli software riutilizzabili (come gli oggetti, ad esempio) è che essi possono essere combinati sotto forma di eseguibili binari, piuttosto che sotto forma di azioni da compiere sul codice sorgente. Inoltre, il modello a componenti si basa sul cosiddetto *middleware implicito* che viene contrapposto alle tecnologie di middleware esplicito (come sono tutte quelle descritte finora).

Il middleware implicito, attraverso meccanismi di intercettazione delle richieste e delle interazioni tra gli oggetti, è in grado di fornire servizi comuni e trasversali ad ogni componente senza che essa debba esplicitamente richiederli all'interno del codice. Il server che gestisce la componente (detta *application server* o *container*) fornisce questi servizi sulla base delle richieste (codificate non nel codice ma in un file di metadati di descri-

[10].NET Remoting è il servizio per oggetti distribuiti offerto da .NET 2.0, incluso successivamente all'interno di Web Communication Framework che include un unico modello di comunicazione per web services, invocazioni sincrone per oggetti remoti e invocazioni asincrone su code di messaggi.

[11]Per maggiori dettagli ed approfondimenti circa l'eterogeneità nelle tecnologie ad oggetti distribuiti, vedere il paragrafo 3.5.1.

zione) specificate quando la componente viene messa a disposizione sul server (fase di *deployment*). In questa maniera i servizi vengono messi a disposizione in maniera completamente trasparente allo sviluppatore di software della componente, realizzando una maggiore interoperabilità tra produttori di software diversi.

Tra i servizi che devono essere messi a disposizione da un sistema a componenti, di particolare importanza sono quelli di fornire un protocollo di comunicazione remota, che permette le interazioni tra le componenti remote. In questo ambito, le caratteristiche esibite dai Middleware ad Oggetti Distribuiti (come Java RMI) li hanno resi la base per il recente sviluppo dei sistemi a componenti, in quanto i meccanismi di comunicazione tra oggetti distribuiti permettono di offrire il supporto per le invocazioni di operazioni tra layer diversi di architetture software, e, se non fossero stati disponibili e ben sperimentati, non sarebbero state probabilmente ottenute la scalabilità, la affidabilità e la ampia diffusione dei sistemi distribuiti e delle applicazioni Web che, invece, possiamo riscontrare nel panorama tecnologico odierno.

Ad esempio, uno dei requisiti fondamentali di un sistema distribuito è quello di poter gestire il ciclo di vita di un oggetto distribuito che permette di attivare oggetti su macchine diverse, quando necessario, per assicurare scalabilità (la efficienza all'aumentare del carico attraverso tecniche di bilanciamento del carico tra i vari nodi) e per poter gestire al meglio i malfunzionamenti hardware e software. Inoltre le tecniche per riuso degli oggetti, l'estendibilità e la modularità della programmazione ad oggetti permettono di facilitare la evoluzione del sistema distribuito a seconda delle necessità contingenti, e ne facilita, quindi, la sua manutenzione e gestione.

Come esempio concreto, prendiamo la Java Enterprise Edition di Sun nella quale il ruolo di Java Remote Method Invocation risulta cruciale per la comunicazione. Infatti, la classica *buona pratica* di progettazione suggerisce di separare nettamente il layer di presentazione dal layer di business: il primo si incarica di assemblare il contenuto che viene fornito alla interfaccia utente di una applicazione, mentre il secondo si occupa di gestire la logica della applicazione e di interfacciarsi con i sistemi per la gestione dei dati e con le applicazioni *legacy*[12]. Ora, per poter bilanciare il carico su macchine differenti, in maniera da poter offrire trasparentemente sia la scalabilità che la tolleranza ai malfunzionamenti, questi due layer sono tipicamente distribuiti su più macchine (o su più insiemi di macchine) e quindi si rende necessario che oggetti nel layer di presentazione invochino operazioni di oggetti nel layer di business. Nel caso specifico, Java Remote Method Invocation fornisce la maniera di poter accedere da macchine diverse ai servizi forniti da una componente distribuita (Enterprise Java Bean) di Java Enterprise Edition. Maggiori dettagli su JEE e sulla architettura vengono mostrati nel paragrafo 9.2. Inoltre, la caratteristica di Java di appoggiarsi per la esecuzione su uno strato di software comune e indipendente dalla architettura (Java Virtual Machine) aggiunge una notevole portabilità e flessibilità, permettendo la esecuzione distribuita su nodi eterogenei.

Va anche notato come la rilevanza del modello ad oggetti distribuiti non è limitato all'ambiente Java, in quanto in maniera simile si può verificare la importanza del ruolo che ricopre la libreria .NET Remoting per l'ambiente enterprise .NET in campo Microsoft, anche esso basato su tecniche di virtualizzazione (Common Language Runtime) che assicurano la eterogeneità nel sistema distribuito.

In conclusione, questo è il compito importante e fondamentale che viene svolto dai sistemi a oggetti distribuiti: assicurare una interoperabilità tra componenti distribuite, permettendo di poter creare sistemi che offrano scalabilità, tolleranza ai malfunzionamen-

[12] I sistemi *legacy* (eredità) sono i sistemi informativi obsoleti in una azienda, ma che rappresentano un importante patrimonio informativo aziendale, non possono essere dismessi e sostituiti, ma vanno integrati all'interno dei sistemi informativi che li rimpiazzano.

ti, estendibilità e facilità di gestione. In particolare, queste ultime due caratteristiche sono assicurate dalla natura della programmazione ad oggetti portata nel contesto distribuito.

Quindi, lungi dall'essere stati sostituiti dai sistemi distribuiti *enterprise* basati sul modello a componenti, gli oggetti distribuiti ne rappresentano una componente chiave, senza la quale gran parte dei vantaggi di tali sistemi non sarebbe stata possibile.

Note bibliografiche

Per approfondimenti sui principi architetturali dei sistemi distribuiti si può consultare il libro di Emmerich [8], capitolo 1, dove viene anche riportata la definizione di sistema distribuito del libro di Couloris [7] che usiamo.

Per le motivazioni economiche e tecnologiche e una completa presentazione e confronto di piattaforme di middleware si consulti [9], [28] e [36]. Alcune critiche a CORBA vengono presentate in [20].

Il modello Open Distributed Processing Reference Model viene estesamente descritto nelle pubblicazioni ufficiali che possono essere scaricate liberamente [22, 23, 24, 25].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[R*]** Illustrare le motivazioni tecnologiche ai sistemi distribuiti, fornendo degli esempi.
2. **[R*]** Illustrare le motivazioni economiche ai sistemi distribuiti, fornendo degli esempi.
3. **[P*]** A cosa serve un modello di riferimento?
4. **[R**]** Confrontare i modelli RM-ODP e ISO/OSI per quanto riguarda obiettivi, ambito di riferimento e struttura.
5. **[R**]** Quali sono le caratteristiche di un sistema distribuito? Confrontare ciascuna caratteristica con la tradizionale architettura centralizzata.
6. **[R**]** Definire ciascuno dei requisiti non funzionali di un sistema distribuito.
7. **[P*]** Il requisito di un sistema distribuito di essere "aperto" non si riferisce esplicitamente al software open-source; ma il software open-source è un importante strumento per assicurare questo requisito: perché?
8. **[P*]** Perché il problema della tolleranza ai malfunzionamenti non viene considerato particolarmente importante in una architettura centralizzata?
9. **[P**]** Supponendo che si debba realizzare un sistema distribuito che non è accessibile da qualsiasi utente di Internet, ma che richiede credenziali per l'accesso, perché si deve comunque garantire la scalabilità della architettura?
10. **[R***]** Definire i tipi di trasparenza e le relative dipendenze.
11. **[P*]** Perché la concorrenza è un problema più complesso da affrontare in un ambito distribuito rispetto ad un sistema centralizzato?
12. **[R***]** Definire i 5 punti di vista di un sistema distribuito.
13. **[R*]** A cosa serve il middleware?
14. **[R**]** A cosa serve il marshalling?
15. **[R**]** Quali sono i tre livelli di middleware in cui si divide il Middleware ad Oggetti Distribuiti? E che ruolo hanno?
16. **[R*]** In che maniera la tecnologia ad oggetti distribuiti è utilizzata all'interno dei sistemi di Enterprise Computing?
17. **[R**]** In che maniera una architettura basata su oggetti distribuiti riesce ad assicurare la scalabilità di un sistema distribuito? E la tolleranza ai malfunzionamenti?
18. **[P*]** Perché è importante assicurare la eterogeneità di un sistema distribuito?

19. **[P*]** Quali sono le differenze sostanziali ed i problemi che ha dovuto affrontare il middleware basato su oggetti rispetto a Remote Procedure Call?
20. **[R**]** Quali sono gli utilizzi del Middleware ad Oggetti Distribuiti nell'Enterprise Computing? Fare qualche esempio.
21. **[P*]** Per ciascuno degli ambienti CORBA, Java RMI e .NET Remoting, identificare i livelli di Middleware ad Oggetti Distribuiti che vengono implementati.

Introduzione a Java RMI

Capitolo 2

Dai socket agli oggetti remoti

Indice

2.1 Introduzione **26**

2.2 Un breve richiamo sulla programmazione con i socket **26**

2.2.1 I socket TCP 27

2.2.2 Gli stream 27

2.2.3 HelloWorld con i socket 28

2.2.4 Un esempio di client-server con i socket 31

2.3 Da un oggetto locale **34**

2.4 ... all'oggetto remoto **36**

2.4.1 Il server e la interfaccia remota 37

2.4.2 Il client 39

2.4.3 Lo strato stub/skeleton 39

2.4.4 La sequenza delle invocazioni 43

2.4.5 Per lanciare la applicazione 43

2.5 Indirizzamento dell'oggetto remoto **45**

2.5.1 Il Client 45

2.5.2 Lo Skeleton 47

2.6 Alcuni commenti conclusivi **49**

2.7 Approfondimenti **49**

2.7.1 Il funzionamento di `getOutputStream()` 49

2.7.2 Conversione di `Byte` unsigned 50

Note bibliografiche **50**

Spunti per lo studio individuale **51**

2.1 Introduzione

Nel paradigma di programmazione orientata ad oggetti, la computazione viene effettuata attraverso un insieme di oggetti che contengono uno *stato* (variabili istanza) ed espongono un *comportamento*, vale a dire permettono ad altri oggetti di usare i loro metodi. Per poter estendere questo modello di programmazione nell'ambito distribuito, è necessario permettere di invocare un metodo da parte di un oggetto remoto.

In questo capitolo svilupperemo un semplice esempio in Java che permette la invocazione remota di metodi attraverso i socket e che mette in evidenza soluzioni simili a quelle adottate da Java Remote Method Invocation. Il cammino che seguiremo ci porterà prima a vedere il semplice esempio in locale per poi esaminare una soluzione distribuita che preservi le caratteristiche di invocazione dei metodi, proprie della programmazione ad oggetti

L'esempio che tratteremo è (ovviamente) alquanto semplice: progettiamo un oggetto che mantiene i dati di un impiegato, come il nome, una ID, il suo stipendio e che espone dei semplici metodi *accessori* (che permettono cioè l'accesso alle variabili istanza private[1]) oltre ad un semplice metodo per incrementare lo stipendio e restituire il valore aggiornato.

2.2 Un breve richiamo sulla programmazione con i socket

Iniziamo con il rivedere la programmazione con i socket TCP in Java, in quanto sarà lo strumento che verrà utilizzato per realizzare la invocazione remota dei metodi di un oggetto.

Java fornisce le API di programmazione di rete all'interno del package `java.net` dove vengono definite classi per trattare gli indirizzi di rete come la classe `InetAddress` ed altre classi simili che gestiscono IPv4 e IPv6.

La comunicazione tra programmi su Internet avviene utilizzando la suite di protocolli TCP/IP. La maniera in cui questi protocolli vengono usati è quello di fornire una astrazione (mediante il software di rete) chiamata *socket* che permette di ricevere e trasmettere dati. Questa astrazione permette al programmatore di vedere una comunicazione su rete come un semplice stream di dati.

TCP (Transmission Control Protocol) offre una connessione affidabile tra i due punti della comunicazione. UDP (User Datagram Protocol) non fornisce una connessione tra due punti ma permette semplicemente di inviare pacchetti di dati (*datagram*) da una applicazione ad un'altra. In questo caso non viene fornito alcun meccanismo per garantire il recapito dei pacchetti, a differenza di TCP.

La scelta del programmatore del protocollo da utilizzare per la comunicazione all'interno di una applicazione distribuita dipende essenzialmente:

a) dalla natura della applicazione, cioè se richiede lo scambio di flussi di dati oppure di messaggi di dimensione limitata;

b) dalla quantità di overhead che si vuole pagare: le comunicazioni TCP richiedono maggiori risorse di calcolo per offrire la affidabilità della comunicazione;

c) dalla criticità della affidabilità della comunicazione: se, insomma, perdere un pacchetto nella comunicazione rappresenta un problema oppure può essere sopportato dalla

[1] In generale, è ottima norma di programmazione progettare lo stato di un oggetto come privato (mediante il modificatore di accesso Java `private`) e fornire, solo laddove esplicitamente necessario, dei metodi di accesso (lettura e scrittura). Le convenzioni in uso prevedono per la variabile `var` l'uso dei metodi `getVar()` e `setVar()`.

applicazione. Si pensi, ad esempio, alla trasmissione di video dove la perdita di un numero limitato di pacchetti fa degradare di poco (e per un brevissimo periodo di tempo) la qualità delle immagini mostrate.

La trasmissione avviene assegnando un socket ad una specifica porta che serve ad identificare, tra tutti i pacchetti che arrivano alla macchina, quali sono quelli destinati ad una certa applicazione.

2.2.1 I socket TCP

I socket TCP sono gli endpoint di una comunicazione bidirezionale sulla rete che unisce due programmi. Ad ogni socket viene assegnato un numero di porta che serve ad identificare la applicazione che è incaricata di dover trattare i dati, che è in esecuzione sul computer che li riceve. Quindi un socket viene univocamente definito dalla combinazione di indirizzo IP e numero di porta.

Normalmente, si identificano i due computer coinvolti in un socket con il nome di *client* e *server*. Il server è in esecuzione ed attende che qualche client richieda la connessione. Dal lato client, il programma conosce l'indirizzo della macchina su cui è in esecuzione il server ed il numero di porta. Il client deve anche comunicare al server il numero di porta locale sul quale riceverà i dati (di solito questo viene assegnato dal sistema).

Il procedimento di connessione prevede che il server debba *accettare* la connessione e che assegni un nuovo socket per la comunicazione bidirezionale tra client e server (socket di comunicazione). In questa maniera, il server può tornare ad accettare connessioni da altri client. In questo caso, tipicamente il server lancia un thread per ogni socket stabilito con un client, in modo da permettere la gestione concorrente delle comunicazioni con tutti i client.

Il package `java.net` offre due classi per i socket, proprio per gestire la fase di accettazione da parte del server e la comunicazione bidirezionale tra client e server: la classe `ServerSocket` e la classe `Socket`. La classe `ServerSocket` implementa un socket di connessione che attende richieste da parte di client; quando ne riceve una, assegna un socket alla connessione bidirezionale, restituendo l'oggetto `Socket` che viene usato per la comunicazione tra client e server.

2.2.2 Gli stream

La comunicazione tra client e server avviene attraverso la scrittura e la lettura di stream (flussi) associati con il socket e che permettono una facile interazione (gestita interamente dal linguaggio) per poter trasmettere istanze di classi Java (oggetti) tra client e server, attraverso un meccanismo di *serializzazione*. Gli stream di I/O sono una utile astrazione che Java fornisce al programmatore per trattare con una sequenza di dati che può essere "diretta a" / "proveniente da" diverse entità, quali file, periferiche, memoria e, ovviamente, socket.

Gli stream sono presenti sotto numerose forme: la gerarchia delle classi di stream nel package `java.io` offre una notevole disponibilità di strumenti per il programmatore. Possiamo brevemente esaminare una parte della gerarchia che parte dalla classe astratta `InputStream` (la gerarchia che deriva da `OutputStream` è pressochè speculare). La sottoclasse più importante che utilizzeremo è `ObjectInputStream` che fornisce il meccanismo di deserializzazione quando si riceve un oggetto precedentemente serializzato con `ObjectOutputStream`. Gli oggetti che possono essere trasmessi su questo tipo di stream devono implementare la interfaccia `Serializable` o la interfaccia `Externalizable` (che fornisce un meccanismo per definire una serializzazione proprietaria, non affidandosi a

quella fornita dal linguaggio). I tipi primitivi possono essere letti mediante gli appropriati metodi, come, ad esempio, `readByte()` o `readFloat()`.

La maniera in cui gli stream vengono creati utilizza il meccanismo di *wrapping*: ogni classe via via più specializzata prende come argomento per il costruttore una istanza di classi più alte nella gerarchia. Ad esempio, in un programma dove abbiamo un socket `sock` il costruttore di `ObjectInputStream` prende come argomento l'`InputStream` che è associato al socket:

```
ObjectInputStream inStream = new ObjectInputStream (sock.getInputStream());
```

Un altro esempio di wrapping, è quello che permette di leggere da uno stream dei caratteri di testo. A questo scopo, tra le classi che derivano da `Reader` esiste la classe `InputStreamReader` che rappresenta la connessione tra gli stream binari e quelli di testo. In effetti un oggetto di `InputStreamReader` legge bytes dall'`InputStream` che gli è stato passato (come parametro al costruttore) e li decodifica in caratteri, utilizzando la codifica utilizzata dal sistema in uso (ad esempio, la classica ISO-8859-1 detta anche "ISO Latin-1").

Un'altra classe utile è la classe `BufferedReader` che fornisce una bufferizzazione di uno stream di input di testo allo scopo di aumentare l'efficienza. Un classico esempio di uso di questa classe (che vedremo negli esempi) consiste nell'applicarla allo stream di input di testo `InputStreamReader` ottenuto convertendo lo stream binario offerto dalla classe `System` mediante `System.in`. Questo viene ottenuto attraverso un classico esempio di uso in cascata:

```
BufferedReader bin = new BufferedReader(new InputStreamReader(System.in));
```

Mettendo insieme, quindi, quello che è necessario per aprire un socket, visto nel paragrafo precedente, e le informazioni sugli stream appena presentate, la sequenza di istruzioni classicamente utilizzata per accedere agli stream di un socket lato server è:

```
ServerSocket serverSocket = new ServerSocket(9000);
socket = serverSocket.accept();
System.out.println (''Accettata una connessione...  attendo comandi.'');
ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
ObjectOutputStream outStream = new ObjectOutputStream (socket.getOutputStream());
```

2.2.3 HelloWorld con i socket

Un semplice esempio (il classico HelloWorld su socket!) illustrerà al meglio l'utilizzo di socket e di stream. In questo esempio, il client si connette al server sull'host locale (`localhost`) passando il suo nome (assumiamo `Giovanni`). Il server risponde salutando Giovanni con `Hello Giovanni`. Iniziamo a vedere come è fatto il server.

Server.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.logging.Logger;
4
5  public class Server {
6    static Logger logger = Logger.getLogger("global");
7
8    public static void main(String[] args) {
9      try {
10       ServerSocket serverSocket = new ServerSocket(9000);
11       logger.info("Socket istanziato, accetto connessioni...");
12       Socket socket = serverSocket.accept();
13       logger.info("Accettata una connessione... attendo comandi.");
14       ObjectOutputStream outStream = new ObjectOutputStream (socket.getOutputStream());
```



```
15       ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
16       String nome= (String) inStream.readObject();
17       logger.info("Ricevuto:" + nome);
18       outStream.writeObject("Hello "+ nome);
19       socket.close();
20     } catch (EOFException e)  {
21       logger.severe("Problemi con la connessione." + e.getMessage());
22       e.printStackTrace();
23     } catch (Throwable t)  {
24       logger.severe("Lanciata Throwable:" + t.getMessage());
25       t.printStackTrace();
26     }
27   }
28 }
```

Fine: Server.java

All'interno della classe `Server`, dichiariamo una istanza di un oggetto `Logger`. Questo è un'importante caratteristica che offre il linguaggio: un oggetto `Logger` raccoglie il log dei messaggi di una specifica applicazione. Quello che si dichiara nella linea 6 è un logger globale, utilizzato per le applicazioni più piccole. Il vantaggio è che, oltre alla ricchezza di informazioni che il logger può fornire (ad esempio utilizzando i metodi `logp..()`) si possono loggare informazioni sulla classe, sul metodo etc.. Ogni messaggio di log ha un livello, ed è possibile settare (da codice, mediante `logger.setLevel(..);` oppure da parametro su linea di comando) il livello al di sotto del quale i log devono essere ignorati (e quindi non stampati). E' anche possibile "spegnere" completamente il meccanismo di logging, sia da codice che da parametro o da file di configurazione (ad esempio basta usare `logger.setLevel(Level.OFF);`). Ma la caratteristica più interessante è che il log può essere rediretto su file, su console, su rete, etc. permettendo al nostro log (inizialmente semplice) di evolvere verso uno strumento flessibile per fornire informazioni circa la esecuzione della nostra applicazione. La comodità è che anche per le applicazioni di piccola dimensione, come le nostre, esistono semplici metodi (quali `logger.info()` oppure `logger.severe()`) per generare dei semplici messaggi di log che vengono stampati su console aggiungendo informazioni come la data, la classe ed il metodo.

Ora, alla linea 10 istanziamo su socket di connessione sulla porta 9000 ed alla linea 12, chiamiamo il metodo *bloccante* `accept()`. Da questo metodo si esce solamente quando un client effettua una richiesta di connessione verso il server, ed il metodo restituisce un socket che è stato stabilito tra il server ed il client. E' su questo socket che usiamo gli stream in input e output (linee 14-15) che servono, prima, per chiamare il metodo (anche esso *bloccante*) `readObject()` che restituisce l'oggetto trasmesso (ed opportunamente deserializzato dallo stream) che viene utilizzato per realizzare la risposta (linea 18). Infine il socket è chiuso alla linea 19.

Ora passiamo al client.

Client.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.logging.Logger;
4
5  public class Client {
6    static Logger logger = Logger.getLogger("global");
7
8    public static void main(String args[]) {
9      try {
10       Socket socket = new Socket ("localhost", 9000);
11       ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
12       ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
```

```
13        out.writeObject("Giovanni");
14        System.out.println(in.readObject());
15        socket.close();
16      } catch (EOFException e) {
17        logger.severe("Problemi con la connessione:" + e.getMessage());
18        e.printStackTrace();
19      } catch (Throwable t) {
20        logger.severe("Lanciata Throwable:" + t.getMessage());
21        t.printStackTrace();
22      }
23    }
24  }
25 }
```

Fine: Client.java

Anche il client ha il suo logger (alla linea 6). Apre il socket verso l'host locale dove abbiamo lanciato il server (linea 10), preleva gli stream dal socket e scrive il suo nome sullo stream di output (linee 11-13). Il client termina stampando a video quello che ha ricevuto dalla lettura dello stream di input. Particolare attenzione va data all'ordine con il quale si devono aprire gli stream: prima lo stream di output e poi quello di input (la ragione è spiegata nel paragrafo 2.7.1).

Quello che accade quando si lancia il server è:

```
Z:\>java Server
8-gen-2009 17.15.36 Server main
INFO: Socket istanziato, accetto connessioni...
```

Il server è in attesa di connessioni. Ora lanciamo il client...

```
Z:\>java Client
Hello Giovanni

Z:\>
```

che stampa a video quello che ha ricevuto dal server, mentre sul server compare:

```
8-gen-2009 17.15.50 Server main
INFO: Accettata una connessione... attendo comandi.
8-gen-2009 17.15.50 Server main
INFO: Ricevuto:Giovanni

Z:\>
```

Per concludere questo semplice esempio, il server può essere facilmente reso iterativo, inserendo la `accept()` e la risposta verso il client all'interno di un `while (true)`. Poi, si può rendere il server multi-thread, in modo che ad ogni accept, si faccia partire un thread (che gestisce l'invio della risposta al client) permettendo al server di tornare subito alla `accept()` successiva.

2.2.4 Un esempio di client-server con i socket

Adesso vediamo un semplice esempio di una applicazione client-server che utilizza i socket TCP in Java. Va sottolineato che questa applicazione verrà utilizzata anche successivamente (nel paragrafo 2.5) per implementare un servizio per gli oggetti remoti.

L'esempio implementa sul server un registro che contiene record (composti da due campi stringa, `nome` e `indirizzo`) che vengono inseriti e possono essere reperiti specificando solamente il nome. Realizzeremo un server che, in attesa su una porta, riceve le richieste di inserimento (codificate attraverso un oggetto `RecordRegistro` con tutti i campi riempiti) o le richieste di ricerca (codificate attraverso un oggetto `RecordRegistro` che ha solo il campo `nome` riempito).

Iniziamo dalla definizione del record da memorizzare, un semplice wrapper di due campi stringa, con i relativi metodi di accesso.

RecordRegistro.java

```
1  import java.io.Serializable;
2
3  public class RecordRegistro implements Serializable {
4    private static final long serialVersionUID = -4147133786465982122L;
5
6    // Costruttore
7    public RecordRegistro(String n, String i) {
8      nome = n;
9      indirizzo = i;
10   }
11
12   // Metodi accessori
13   public String getNome() {
14     return nome;
15   }
16   public String getIndirizzo() {
17     return indirizzo;
18   }
19
20   // Variabili istanza
21   private String nome;
22   private String indirizzo;
23 }
```

Fine: RecordRegistro.java

L'unico commento a questa classe lo merita il fatto che le sue istanze sono serializzabili, cioè implementano la interface `Serializable`, in quanto ci aspettiamo che debbano essere trasmesse su socket TCP in formato binario. In generale, è considerato importante che una classe serializzabile dichiari esplicitamente (linea 4) un `serialVersionUID`, e non ci si affidi al calcolo che viene fatto automaticamente (in caso di mancanza di indicazione esplicita da parte del programmatore) e che può essere molto dipendente da implementazioni di compilatori diversi. In pratica, l'algoritmo di deserializzazione usa questo numero per assicurarsi che la classe appena caricata corrisponde ad un oggetto serializzato. Se questo non corrisponde al numero che ha calcolato, allora viene lanciata una eccezione `InvalidClassException`. In caso di uso di compilatori diversi, si potrebbe non riuscire a trasferire (nel nostro esempio) record dal server al client (o viceversa) perché (apparentemente) in fase di deserializzazione lo stream di dati non corrisponde ad una serializzazione di una classe. I `serialVersionUID` possono essere generati con una routine fornita dal JDK che si chiama **serialver**, che prende in input il file della classe Java di cui si vuole calcolare l'ID. Ovviamente, tutte le IDE per programmatori Java forniscono l'inserimento automatico di questo dato nel file. Ma possono anche essere generati manualmente, modificandoli

(incrementandoli di 1, ad esempio) ogni qualvolta la definizione della classe viene effettivamente modificata. In ogni caso, sia che si inserisca: `private static final long serialVersionUID = 1L;` oppure che si usi un altro valore dato da **serialver**, la regola importante è abituarsi ad usarlo.

Ora passiamo al server. Questo server, abbastanza semplice per la verità, non fa altro che attendere sulla porta 7000 che ci siano delle richieste di connessione. Il server è iterativo e serve le richieste così come arrivano, quindi senza multi-thread.

RegistroServer.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5
6 public class RegistroServer {
7   static Logger logger = Logger.getLogger("global");
8
9   public static void main(String[] args) {
10    HashMap<String, RecordRegistro> hash = new HashMap<String, RecordRegistro>();
11    Socket socket = null;
12    System.out.println ("In attesa di connessioni...");
13    try {
14      // Creazione ed accept su socket
15      ServerSocket serverSocket = new ServerSocket(7000);
16      while (true) {
17        socket = serverSocket.accept();
18        ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
19        RecordRegistro record = (RecordRegistro) inStream.readObject();
20        if (record.getIndirizzo()!=null) { // si tratta di una scrittura
21          logger.info ("Inserisco:" + record.getNome() +", "+record.getIndirizzo());
22          hash.put(record.getNome(), record);
23        } else { // è una ricerca
24          logger.info ("Cerco:" + record.getNome());
25          RecordRegistro res = hash.get(record.getNome());
26          ObjectOutputStream outStream=new ObjectOutputStream(socket.getOutputStream());
27          outStream.writeObject(res); // se non c'è il record, res è null
28          outStream.flush();
29        }
30        socket.close();
31      } // fine while
32    } catch (EOFException e)  {
33      logger.severe("Problemi con la connessione:" + e.getMessage());
34      e.printStackTrace();
35    } catch (Throwable t)  {
36      logger.severe("Lanciata Throwable:" + t.getMessage());
37      t.printStackTrace();
38    }
39    finally { // chiusura del socket e terminazione programma
40      try { socket.close();
41      } catch (IOException e) {
42        e.printStackTrace();
43        System.exit(0);
44      }
45    }
46  }
47 }
```

Fine: RegistroServer.java

Tutti i record che arrivano vanno memorizzati in una HashMap che, come chiave di

accesso, utilizza il nome presente nel record (linea 10). Nel ciclo infinito alla linea 16-31 il programma accetta connessioni, riceve un oggetto dallo stream in input (linea 19). Se l'oggetto ha il campo indirizzo non vuoto, allora è una richiesta di inserimento, che viene effettuata (linee 20-22); altrimenti si tratta di una ricerca, che viene effettuata (linea 25) e che viene restituita sullo stesso socket, al client. Se la ricerca è stata infruttuosa, il metodo `get()` (linea 25) restituisce `null` che viene restituito al client (che controllerà questo valore per segnalare una ricerca senza successo all'utente).

Ora passiamo al client. Questo è organizzato con una semplice shell, che permette all'utente di inserire diversi comandi: *inserisci* e *cerca*, oltre che *quit* per uscire.

ShellClient.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.logging.Logger;
4 public class ShellClient {
5   static Logger logger = Logger.getLogger("global");
6
7   public static void main(String args[]) {
8     String host = args[0];
9     String cmd;
10    in = new BufferedReader(new InputStreamReader(System.in));
11    try {
12      while (!(cmd = ask ("Comandi>")).equals("quit")) {
13        if (cmd.equals ("inserisci")) {
14          System.out.println ("Inserire i dati.");
15          String nome = ask("Nome: ");
16          String indirizzo = ask ("Indirizzo: ");
17          RecordRegistro r = new RecordRegistro(nome, indirizzo);
18          Socket socket = new Socket (host, 7000);
19          ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
20          sock_out.writeObject(r);
21          sock_out.flush();
22          socket.close();
23        } else if (cmd.equals ("cerca")) {
24          System.out.println ("Inserire il nome per la ricerca.");
25          String nome = ask("Nome: ");
26          RecordRegistro r = new RecordRegistro(nome, null);
27          // si invia un oggetto con indirizzo vuoto
28          Socket socket = new Socket (host, 7000);
29          ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
30          sock_out.writeObject(r);
31          sock_out.flush();
32          ObjectInputStream sock_in = new ObjectInputStream(socket.getInputStream());
33          RecordRegistro result = (RecordRegistro) sock_in.readObject();
34          // se viene ottenuto un risultato, allora si stampa l'indirizzo
35          if (result != null)
36            System.out.println ("Indirizzo :"+result.getIndirizzo());
37          else // altrimenti non esiste un record con quel nome
38            System.out.println ("Record non presente");
39          socket.close();
40        } else System.out.println (ERRORMSG);
41      } // end while
42    } catch (Throwable t) {
43      logger.severe("Lanciata Throwable:" + t.getMessage());
44      t.printStackTrace();
45      }
46    System.out.println ("Bye bye");
47  }
48
```

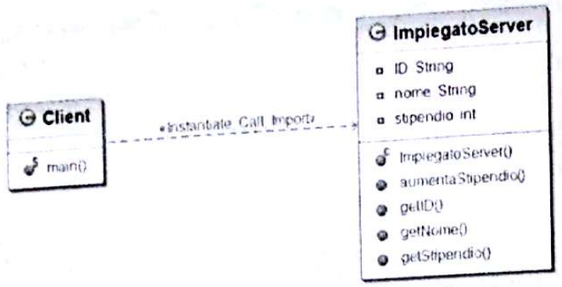

Figura 2.1: *Il diagramma UML delle classi della applicazione di esempio in locale.*

```
49  private static String ask(String prompt) throws IOException {
50    System.out.print(prompt+" ");
51    return (in.readLine());
52  }
53
54  static final String ERRORMSG = "Cosa?";
55  static BufferedReader in = null;
56 }
```

Fine: ShellClient.java

Per ogni richiesta dell'utente, apre il socket, fa la richiesta al server e scrive la risposta (se necessario), chiudendo il socket. Questo avviene sia per l'inserimento (linee 13-22) che per la ricerca (linee 23-39). Per l'inserimento, dopo aver chiesto all'utente di digitare le stringhe per comporre l'oggetto `RecordRegistro` da inviare (linee 14-17), apre il socket, usa lo stream di output, invia l'oggetto e chiude il socket (linee 18-22). Per la ricerca, dopo aver chiesto all'utente il nome del record da ricercare, il client invia un record con il campo indirizzo a null (linee 26-31) e si attende un oggetto in risposta (linea 33). Se l'oggetto non è `null` (linee 36), allora la ricerca è andata a buon fine e si stampa l'indirizzo dell'oggetto ricevuto, altrimenti si stampa un messaggio, e si chiude il socket (linee 35-39).

Il metodo `ask()` serve per fare input da tastiera in maniera semplice, scrivendo un prompt definito nel programma.

2.3 Da un oggetto locale ...

Adesso illustriamo un semplice esempio di un oggetto locale per poi portar lo stesso esempio (usando i socket ed una architettura basata sui meccanismi di Java RMI) in un ambito distribuito e per permettere, alla fine di questo cammino, il confronto tra questo programma e quello distribuito che otterremo alla fine.

L'esempio è quello di una classe che modella un impiegato in una azienda, con un nome, una ID ed uno stipendio. Il diagramma UML delle classi della applicazione che desideriamo progettare viene mostrato in Figura 2.1. La classe `ImpiegatoServer` mantiene i dati necessari fornendo il costruttore, i metodi di accesso alle variabili di istanza ed il metodo `aumentaStipendio()` che prende come parametro l'ammontare dell'aumento e restituisce lo stipendio aggiornato.

Definiamo i ruoli ricoperti nel diagramma di Figura 2.1: la classe `ImpiegatoServer` funge da *fornitore di servizio* cioé *server* mentre la classe `Client` ricopre il ruolo di *fruitore di servizio* cioé *client*. È importante distinguere quando i termini client e server sono

usati nell'ambito di architetture distribuite e quando si descrive invece il ruolo di oggetti distribuiti; in quest'ultimo caso, infatti, ci si riferisce ad una singola chiamata di metodo: l'oggetto server rappresenta l'oggetto che riceve la invocazione che viene effettuata dall'oggetto client. Il ruolo può invertirsi qualora il "server" di una precedente invocazione diventi il client per una invocazione di un metodo su un altro oggetto remoto.

La definizione della classe `ImpiegatoServer`, su cui non sono necessari commenti ulteriori (data la semplicità), viene mostrata di seguito.

ImpiegatoServer.java

```
1 // Server della applicazione ImpiegatoLocale
2 public class ImpiegatoServer {
3
4   // Costruttore
5   public ImpiegatoServer (String n, String I, int s) {
6     nome = n;
7     ID = I;
8     stipendio = s;
9   }
10
11  // Metodi accessori
12  public String getNome() {
13    return nome;
14  }
15  public String getID() {
16    return ID;
17  }
18
19  public int getStipendio() {
20    return stipendio;
21  }
22
23  // Metodi specifici
24  public int aumentaStipendio (int diQuanto) {
25    if (diQuanto > 0)
26      stipendio += diQuanto;
27    return stipendio;
28  }
29
30  // Variabili istanza
31  private String nome;
32  private String ID;
33  private int stipendio;
34 }
```

Fine: ImpiegatoServer.java

L'accesso ai servizi offerti da `ImpiegatoServer` viene ottenuto attraverso una semplice classe `Client` che non fa altro che istanziare un oggetto dalla classe `ImpiegatoServer` ed usarne i metodi.

Client.java

```
1 // Client dell'esempio ImpiegatoLocale
2 public class Client {
3   public static void main(String[] args) {
4     // Si istanzia un impiegato
5     ImpiegatoServer imp = new ImpiegatoServer("Mario Rossi", "01721", 30000);
6
7     System.out.println ("Nome: " + imp.getNome());
8     System.out.println ("ID: " + imp.getID());
9     System.out.println ("Stipendio: " + imp.getStipendio());
```

```
5
6       System.out.println("Aumentiamo lo stipendio di 1000 euro");
7       System.out.println ("Ora il suo stipendio è di: " + imp.aumentaStipendio(1000));
8   }
9 }
```

Fine: Client.java

Va notato che il client si occupa (in questo semplicissimo esempio) di istanziare, prima, l'oggetto server e, poi, di usare il suo riferimento (linee 7-11). In effetti, questi problemi, la istanziazione di un oggetto remoto e il riferimento ad esso, saranno problemi che cambieranno sensibilmente nelle applicazioni distribuite.

2.4 ...all'oggetto remoto

Adesso cerchiamo di rendere distribuito il semplice programma visto nella sezione precedente. I termini che vengono indicati in ***corsivo grassetto*** corrispondono a concetti che verranno formalizzati successivamente per Java Remote Method Invocation.

Adottiamo il principio della astrazione, introducendo uno strato di software che viene utilizzato per nascondere al programmatore la maggior parte del lavoro necessario per permettere la invocazione remota di metodi. Lo strato di software che viene inserito consiste di due componenti: lo ***stub*** e lo ***skeleton*** (anche chiamato *client stub*). La terminologia è stata introdotta da Remote Procedure Call negli anni '80.

Lo ***stub*** è un oggetto che si trova sul client che rappresenta, a tutti gli effetti, l'oggetto server verso il client: presenta ed espone, infatti, gli stessi metodi che vengono esposti sul server. Il compito principale (tra quelli descritti con maggior dettaglio nelle sezioni successive) è quello di comunicare con lo ***skeleton*** che si trova sul lato server. Ogni chiamata del client verso i metodi remoti dello stub genera una comunicazione tra lo stub e lo skeleton; quest'ultimo si occupa di effettuare la invocazione del metodo richiesto sull'oggetto server, ricevere il valore restituito dal metodo e comunicarlo allo stub che, a sua volta, lo restituisce verso il client[2] come valore restituito dalla invocazione del suo metodo da parte del client. Ogni comunicazione tra stub e skeleton (nei due sensi) avviene attraverso un protocollo comune che deve prevedere come si indica il metodo da eseguire e come si inviano i parametri ed il valore restituito.

Avendo introdotto stub e skeleton, passiamo al problema successivo: come può l'oggetto client (o meglio, in questo esempio, lo stub) sapere quali sono i metodi che sono disponibili sull'oggetto server? La soluzione sta nel fatto che sia lo stub che il server implementano una interfaccia comune, detta ***interfaccia remota*** dove sono definiti i metodi che devono essere invocati da remoto.

Vale la pena riassumere i tre componenti fondamentali che abbiamo presentato:

- lo stub, lato client, che rappresenta per il client l'oggetto server in locale;
- lo skeleton, lato server, che serve a gestire la comunicazione da e verso lo stub, ed a effettuare le invocazioni verso il server;
- la interfaccia remota, che rappresenta una definizione dei metodi che il server vuole esporre verso il client e che, essendo implementata sia dallo stub che dal server, lega in maniera inscindibile queste due componenti.

[2]La anticipazione che sia stub che skeleton vengono generati automaticamente servirà probabilmente a tranquillizzare chi inizia a preoccuparsi per la complessità dei compiti che essi svolgono.

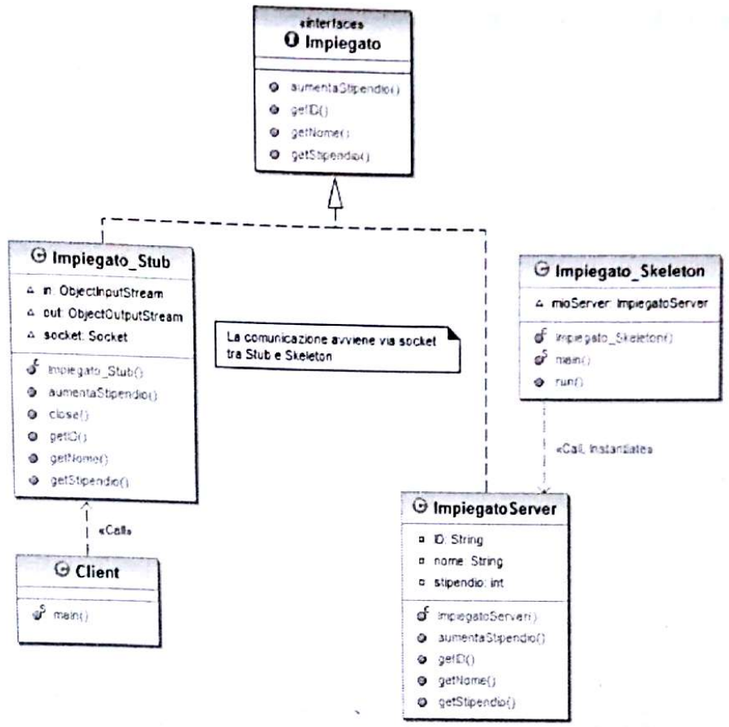

Figura 2.2: *Il diagramma UML delle classi della applicazione di esempio in distribuito.*

Senza addentrarci ora in questi argomenti, che verranno trattati in maniera più ampia successivamente[3], passiamo allora a scrivere l'esempio di `ImpiegatoServer` in distribuito.

L'obiettivo è quello di verificare quale sia la complessità dello strato di software che ci permette di effettuare invocazioni remote di metodi con poco sforzo (come, vedremo, riuscirà a fare RMI).

2.4.1 Il server e la interfaccia remota

Tornando al nostro esempio, scriveremo uno stub ed uno skeleton che si occupano della comunicazione tra `Client` (il client) e `ImpiegatoServer` (il server). Inoltre, forniremo una interfaccia remota che definisce i metodi ai quali il client vuole accedere da remoto. Il diagramma delle classi dell'esempio in distribuito è mostrato in Figura 2.2.

Iniziamo adesso a progettare questa applicazione definendo quali sono i metodi del server che devono essere accessibili da remoto, vale a dire specificando la interfaccia remota. La interfaccia `Impiegato` definisce ognuno dei 4 metodi che devono essere accessibili da remoto. È importante notare che ogni metodo, essendo remoto, viene dichiarato tale da poter lanciare eccezioni, chiaramente dovute all'utilizzo della rete durante la effettiva chiamata. Per Java RMI verranno definite delle ***eccezioni remote*** che serviranno proprio a questo scopo.

Impiegato.java

[3]Ad esempio, vale la pena anticipare che parte del lavoro svolto da stub e skeleton viene svolto in maniera diversa da RMI che arriva ad eliminare (parzialmente) la necessità di queste due classi.

```
1 public interface Impiegato {
2   public String getNome() throws Throwable;
3   public String getID() throws Throwable;
4   public int getStipendio()throws Throwable;
5   public int aumentaStipendio(int diQuanto)throws Throwable;
6 }
```

Fine: Impiegato.java

Ora, passiamo alla definizione del server, vale a dire del fornitore del servizio, nella classe `ImpiegatoServer`.

ImpiegatoServer.java

```
1 public class ImpiegatoServer implements Impiegato {
2
3   // Costruttore
4   public ImpiegatoServer (String n, String I, int s) {
5     nome = n;
6     ID = I;
7     stipendio = s;
8   }
9
10  // Metodi accessori
11  public String getNome() {
12    return nome;
13  }
14
15  public String getID() {
16    return ID;
17  }
18
19  public int getStipendio() {
20    return stipendio;
21  }
22
23  // Metodi specifici
24  public int aumentaStipendio (int diQuanto) {
25    if (diQuanto > 0)
26      stipendio += diQuanto;
27    return stipendio;
28  }
29
30  // Variabili istanza
31  private String nome;
32  private String ID;
33  private int stipendio;
34 }
```

Fine: ImpiegatoServer.java

Il costruttore della classe (linee 3-8) assegna i valori alle variabili di istanza definite alle linee 31-33. I metodi remoti sono implementati in maniera ovvia. Si dovrebbe notare come, in pratica, questa classe risulta identica a quella corrispondente dell'esempio in locale: l'unica differenza è la dichiarazione della interfaccia remota `Impiegato`. In effetti, questo è un primo indizio della netta separazione tra client e server: il client ha a disposizione solamente la interfaccia remota del server, che definisce i servizi offerti dal server, ma senza specificarne la implementazione, che resta a cura del server e che è bene che rimanga nascosta al client, in modo da non esserne dipendente. Questo assicura una evoluzione di client e server che può procedere in maniera indipendente: fin quando il server continua

ad esporre i servizi nella stessa modalità (specificata dalla interfaccia) allora tutti i client possono continuare a fruirne.

Vale la pena anche di sottolineare che in questo esempio la interfaccia remota non è strettamente necessaria (infatti si può eliminare il riferimento alla interfaccia e tutto continua a funzionare!) ma viene usato per abituarci all'uso in RMI della interfaccia remota da parte del server.

2.4.2 Il client

Vediamo ora la parte della nostra applicazione che si riferisce al lato client.

Client.java

```
1 import java.util.logging.Logger;
2
3 public class Client {
4   static Logger logger = Logger.getLogger("global");
5
6   public static void main(String args[]) {
7     try {
8       Impiegato imp = new Impiegato_Stub(args[0]); //host
9       System.out.println ("Nome: " + imp.getNome());
10      System.out.println ("ID: " + imp.getID());
11      System.out.println ("Stipendio: " + imp.getStipendio());
12      System.out.println ("Aumentiamo lo stipendio di 1000 euro");
13      System.out.println ("Ora il suo stipendio è di: "+imp.aumentaStipendio(1000));
14      ((Impiegato_Stub) imp).close();
15    } catch (Throwable t) {
16      logger.severe("Lanciata Throwable:" + t.getMessage());
17      t.printStackTrace();
18    }
19  }
20 }
```

Fine: Client.java

Due sono le modifiche sostanziali al client rispetto alla versione locale. La prima riguarda il fatto che si deve specificare quale è l'host su cui è in esecuzione l'oggetto server. Alla linea 8, il primo parametro sulla linea di comando (`args[0]`) viene passato come parametro al costruttore di `Impiegato_Stub` che si occuperà di contattare il server, invocarne un metodo ed averne il valore restituito. La seconda modifica riguarda il blocco `try .. catch()` presente sulle linee 7-15 che serve perchè le invocazioni remote possono generare eccezioni.

Una osservazione importante riguarda il tipo della variabile `imp`. Infatti, si tratta di un oggetto che implementa la interfaccia (remota) `Impiegato`: niente altro è conosciuto di questo oggetto. Questo comporta che, per poter usare il metodo specifico `close()` (che abbiamo aggiunto per permettere una chiusura "pulita" del programma dal lato client) sia necessario fare un casting esplicito al tipo `Impiegato_Stub` (linea 10).

2.4.3 Lo strato stub/skeleton

Siamo finalmente pronti per vedere come viene implementata la comunicazione tra client e server mediante la definizione di stub (lato client) e skeleton (lato server) che gestiscono tutto quanto necessario alla invocazione ed alla restituzione del valore. Ricordiamo che lo stub deve esporre verso il client tutti i metodi che sono definiti nella interfaccia

remota. Inoltre, come specificheremo in seguito, la classe definisce un metodo di servizio che permette la chiusura del socket. Innanzitutto vediamo il codice della classe Impiegato_Stub.

Impiegato_Stub.java

```
1 import java.io.*;
2 import java.net.*;
3 public class Impiegato_Stub implements Impiegato {
4
5   // Costruttore
6     public Impiegato_Stub(String host) throws Throwable {
7         // Inizializzazione variabili di istanza
8         socket = new Socket (host, 9000);
9         out = new ObjectOutputStream(socket.getOutputStream());
10        in = new ObjectInputStream(socket.getInputStream());
11    }
12
13  // Metodo remoto di accesso al nome
14    public String getNome () throws Throwable {
15        out.writeObject("getNome");
16        out.flush();
17        return (String) in.readObject();
18    }
19
20  // Metodo remoto di accesso alla ID
21    public String getID () throws Throwable {
22        out.writeObject("getID");
23        out.flush();
24        return (String) in.readObject();
25    }
26
27  // Metodo remoto di accesso allo stipendio
28  public int getStipendio () throws Throwable {
29        out.writeObject("getStipendio");
30        out.flush();
31        return in.readInt();
32    }
33
34  // Metodo remoto di aumento dello stipendio
35    public int aumentaStipendio (int diQuanto) throws Throwable {
36        out.writeObject("aumentaStipendio");
37        out.writeInt(diQuanto);
38        out.flush();
39        return in.readInt();
40    }
41
42  // Metodo (locale) per la chiusura del socket con lo skeleton
43    public void close () {
44        try { socket.close();
45        }
46        catch (IOException e) {
47          System.out.println ("Chiusura socket non effettuata con successo!");
48        }
49    }
50
51    // Variabili di istanza
52    Socket socket;
53    ObjectOutputStream out;
54    ObjectInputStream in;
55 }
```


Fine: Impiegato_Stub.java

Il costruttore della classe (linee 6-11) riceve come primo parametro su linea di comando il nome dell'host su cui il server è in esecuzione ed assegna i valori alle 3 variabili di istanza (linee 52-54) creando (sulla porta 9000) un socket prelevandone (ed assegnando alle variabili di istanza `in` e `out`) il flusso di input e di output (come `ObjectStream`).

Passiamo ora alla invocazione remota dei metodi. È necessario stabilire un protocollo per poter comunicare tra stub e skeleton. La (semplice) soluzione scelta consiste nell'inviare allo skeleton una stringa con il nome del metodo da chiamare sul server (inviando successivamente eventuali parametri del metodo) e leggere dal flusso in input dello stesso socket il valore restituito. Tra tutti i metodi remoti offerti dallo stub, l'unico commento necessario si riferisce al metodo `aumentaStipendio()` che è l'unico dei metodi remoti che richiede il passaggio di un parametro: questo viene inviato alla linea 37 del codice, subito dopo la stringa che indica il nome del metodo. Lo skeleton, per la chiamata di questo metodo, attenderà il parametro passato per passarlo al metodo del server.

Concludiamo con il metodo locale `close()` che abbiamo introdotto per poter chiudere in maniera "pulita" (in inglese si dice "*gracefully*") il socket che viene aperto con lo skeleton. Di fatto, se non si chiama il metodo `close()` dello stub, lo skeleton riscontra la improvvisa chiusura del socket quando lo stub termina il suo lavoro e lancia una eccezione.

Adesso passiamo alla controparte dello stub: lo skeleton. Il compito principale risulta nel ricevere le richieste di invocazioni remote che vengono inviate dallo stub e a provvedere ad effettuarle su un oggetto `ImpiegatoServer` che ha provveduto ad istanziare. A causa della sua struttura, la classe costruita è un thread con un metodo `main()` (alle linee 9-15 del listato seguente) che serve sia ad istanziare l'oggetto server e l'oggetto skeleton che a far partire il thread che serve le richieste. Ecco il listato di `Impiegato_Skeleton`.

Impiegato_Skeleton.java

```
1 import java.io.*;
2 import java.net.*;
3 public class Impiegato_Skeleton extends Thread {
4   // Costruttore
5   public Impiegato_Skeleton(ImpiegatoServer server) {
6     mioServer = server;
7   }
8
9   public static void main (String args[]) {
10    // Istanziazione oggetto Server
11    ImpiegatoServer impiegato = new ImpiegatoServer ("Mario Rossi", "01721", 30000);
12    // Istanziazione skeleton e sua esecuzione
13    Impiegato_Skeleton skel = new Impiegato_Skeleton(impiegato);
14    skel.start();
15  }
16
17  // Attivita' dello skeleton
18  public void run() {
19    Socket socket = null;
20    String metodo;
21    int parametro;
22    System.out.println ("Attendo connessioni...");
23    try {
24      // Creazione ed accept su socket
25      ServerSocket serverSocket = new ServerSocket(9000);
26      socket = serverSocket.accept();
27      System.out.println ("Accettata una connessione... attendo comandi.");
```

```
28      ObjectInputStream inStream = new ObjectInputStream (socket.getInputStream());
29      ObjectOutputStream outStream = new ObjectOutputStream (socket.getOutputStream);
30      while (true) {
31        // Lettura del nome del metodo da eseguire
32        metodo = (String) inStream.readObject();
33        System.out.println ("Comando richiesto:" + metodo);
34        if (metodo.equals("getNome")) {
35          outStream.writeObject(mioServer.getNome());
36          outStream.flush();
37        } else if (metodo.equals("getID")) {
38          outStream.writeObject(mioServer.getID());
39          outStream.flush();
40        } else if (metodo.equals("getStipendio")){
41          outStream.writeInt(mioServer.getStipendio());
42          outStream.flush();
43        } else if (metodo.equals("aumentaStipendio")){
44          parametro = inStream.readInt();
45          outStream.writeInt(mioServer.aumentaStipendio(parametro));
46          outStream.flush();
47        } else break;
48      } // fine while
49    } catch (EOFException e)  {
50      System.out.println ("Terminata la connessione");
51    }
52    catch (Throwable t)  {
53      t.printStackTrace();
54      System.out.println ("Skeleton:"+t.getMessage());
55    }
56    finally { // chiusura del socket e terminazione programma
57      try { socket.close();
58      } catch (IOException e) {
59        e.printStackTrace();
60        System.exit(0);
61      }
62    }
63  } // fine run()
64
65  // Variabili di istanza
66  ImpiegatoServer mioServer;
67 }
```

Fine: Impiegato_Skeleton.java

Il metodo `run()` del thread inizia rimanendo in attesa sulla `accept()` del socket (linea 26). Una volta stabilita una connessione, vengono prelevati flusso di input e di output dal socket e si entra in un ciclo infinito (linee 30-48) che consiste nel rimanere in attesa di una stringa che indica il metodo da eseguire (linea 32). A seconda del valore della variabile `metodo`, si esegue il blocco corrispondente della serie di `if..else`. Ad esempio, se il metodo richiesto è `getID()` allora (linee 37-40) viene scritto nel flusso di output il risultato della invocazione sull'oggetto server del metodo corrispondente. Una particolarità è la esecuzione del metodo `aumentaStipendio()` (linee 43-47) che richiede, da parte dello skeleton, la lettura di un parametro (l'intero `parametro`) che viene passato al metodo dell'oggetto server.

Una ultima annotazione riguarda la clausola `finally` (linee 56-62) che, per poter chiudere il socket, deve a sua volta includere un blocco `try.. catch` per poter controllare eventuali eccezioni in chiusura di socket.

2.4.4 La sequenza delle invocazioni

Riassumiamo la sequenza di chiamate e l'uso del socket da parte delle componenti di questo esempio (vedi anche la Figura 2.3). Assumiamo che l'utente del client sia chiamato *Alice* e quello del server sia chiamato *Bob*.

1. Bob lancia il server, eseguendo il programma Skeleton.
2. Lo Skeleton viene lanciato e istanzia (linea 11) l'oggetto ImpiegatoServer.
3. Alice può lanciare il client C. Va notato che Alice può lanciare il client in ogni momento, ma **dopo** che Bob ha lanciato lo Skeleton, in quanto le chiamate su oggetti remoti sono inerentemente *sincrone*, cioè si suppone che siano contemporaneamente in esecuzione sia client che server.
4. All'interno di Client (linea 8) viene istanziato lo Stub.
5. Lo Stub si connette con un socket allo skeleton.
6. Da Client (linea 9) viene fatta la invocazione a `getNome()` di `Impiegato_Stub`.
7. In `Impiegato_Stub` (linee 14-18) viene passata la invocazione del metodo all'oggetto `Impiegato_Skeleton` (come oggetto stringa che contiene il nome del metodo).
8. In `Impiegato_Skeleton` viene ricevuta sul socket la stringa inviata (linea 32) e viene (linee 34-36) effettuata la chiamata al server del metodo `getNome()` ...
9. ...risultato che viene inviato sul socket (linea 35).
10. Lo stub riceve il risultato sul socket (linea 17) e viene restituito al client.

2.4.5 Per lanciare la applicazione

Per lanciare la applicazione sul server si lancia:

```
Z:\>java Impiegato_Skeleton
Attendo connessioni...
Accettata una connessione... attendo comandi.
Comando richiesto:getNome
Comando richiesto:getID
Comando richiesto:getStipendio
Comando richiesto:aumentaStipendio
Terminata la connessione

Z:\>
```

Sul lato client, invece, l'output risulta essere identico a quello lanciato in locale, se non per il fatto che si introduce il nome dell'host su linea di comando (nel nostro esempio localhost).

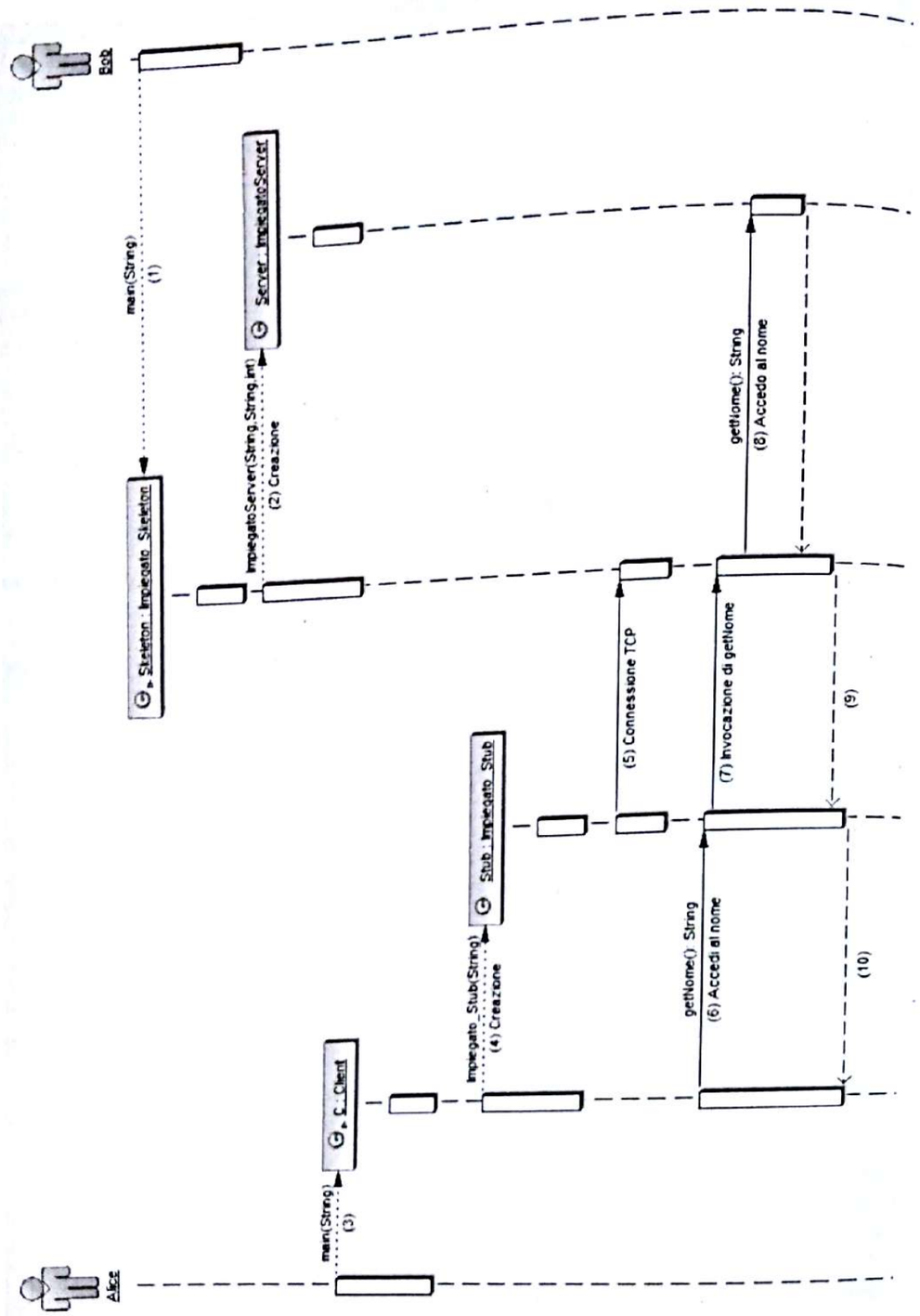

Figura 2.3: *La sequenza delle invocazioni tra client, server, stub e skeleton.*


```
Z:\>java Client localhost
Nome: Mario Rossi
ID: 01721
Stipendio: 30000
Aumentiamo lo stipendio di 1000 euro
Ora il suo stipendio è di: 31000

Z:\>
```

2.5 Indirizzamento dell'oggetto remoto

Ora, il problema che si pone è quello dell'indirizzamento: come si può reperire il riferimento all' "oggetto remoto" da parte del (dei) client? Infatti, il client, nell'esempio precedente, riceveva da linea di comando il nome dell'host su cui si trovava l' "oggetto remoto" (il suo skeleton). Come può fare a sapere queste informazioni?

La soluzione più semplice (che è anche quella adottata da RMI, ma anche da tanti ambienti di oggetti distribuiti, come CORBA) è quella di prevedere che ci sia un *servizio* disponibile, e la cui locazione è conosciuta, che permetta di poter reperire (a tempo di esecuzione) l'indirizzo di un oggetto di cui sappiamo solamente il nome, un identificativo. Questo serve a realizzare la trasparenza di locazione illustrata nel Capitolo 1, ma solo parzialmente, in quanto il servizio di naming è (invece) localizzato: si deve sapere dove si trova per poterlo utilizzare.

A questo scopo, possiamo certamente usare l'esempio di `RegistroServer` fatto in precedenza nel paragrafo 2.2.4. In pratica, si tratta di usare il campo `nome` come l'ID di un oggetto remoto, mentre usiamo il campo `indirizzo` per memorizzare l'host su cui va aperto il socket, verso lo skeleton appropriato. Possiamo limitarci, quindi ad eseguire la applicazione `RegistroServer` vista in precedenza, e modificare la applicazione (sia lato client che server) come segue.

La applicazione che si ottiene è quindi una fusione delle due viste in precedenza, ottenendo il diagramma delle classi mostrato in Fig. 2.4.

Le uniche due classi che vanno modificate sono il Client e lo Skeleton: il Client non dovrà (non potrà) collegarsi direttamente all'oggetto Server (attraverso lo Stub e lo Skeleton) ma dovrà prima richiedere al Registro l'indirizzo IP dove è in ascolto il Server (cioè il suo Skeleton). Lo Skeleton, invece, dovrà provvedere a registrare il suo indirizzo IP sul Registro, per farlo ritrovare dal Client.

2.5.1 Il Client

Iniziamo a vedere le modifiche necessarie per il Client. Innanzitutto, il Client (a differenza di quello visto nel paragrafo 2.4.2) deve accettare come parametro l'host su cui il *servizio* di Registro è disponibile, non quello dove l'oggetto si trova.

Client.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.logging.Logger;
4
5 public class Client {
6   static Logger logger = Logger.getLogger("global");
```

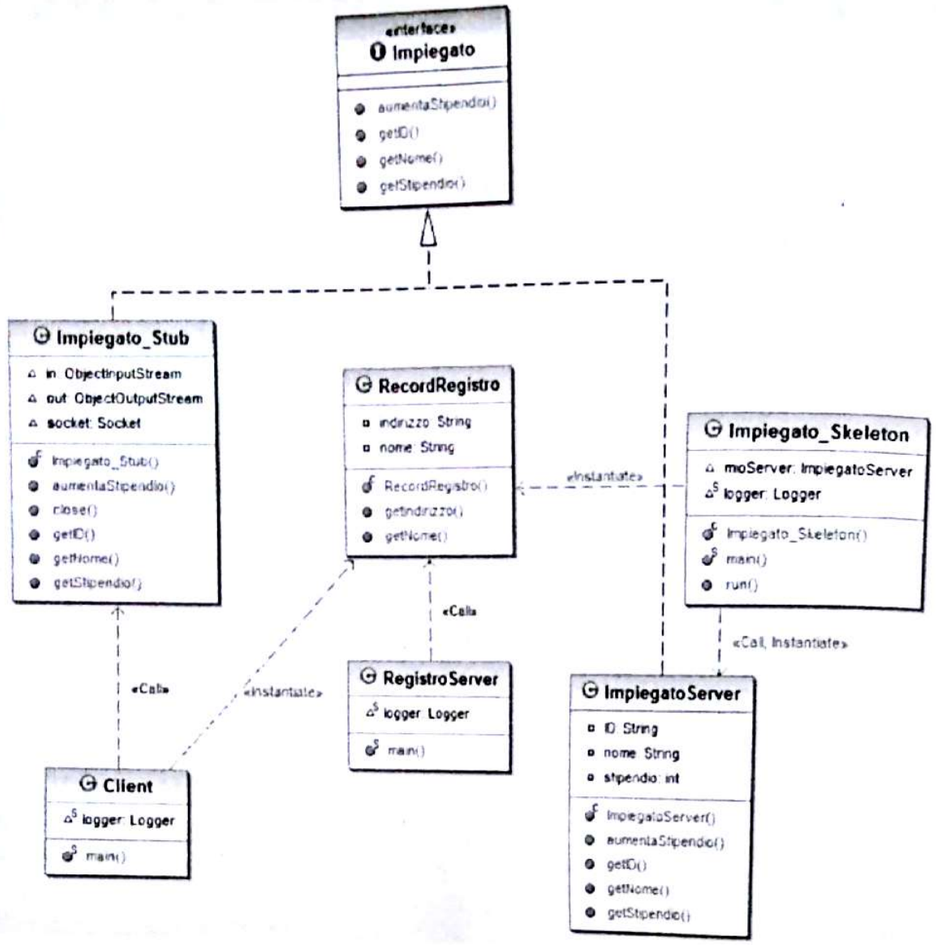

Figura 2.4: *Il diagramma delle classi della applicazione di Impiegato, con un registro remoto.*


```
7
8   public static void main(String args[]) {
9     try {
10      // cerco l'host su cui è registrato il servizio Impiegato
11      RecordRegistro r = new RecordRegistro("Rossi", null);
12      Socket socket = new Socket (args[0], 7000);
13      ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
14      sock_out.writeObject(r);
15      sock_out.flush();
16      // aspetto la risposta
17      ObjectInputStream sock_in = new ObjectInputStream(socket.getInputStream());
18      RecordRegistro result = (RecordRegistro) sock_in.readObject();
19      sock_in.close();
20      // se viene ottenuto un risultato, allora...
21      if (result != null){
22        // ... uso l'indirizzo trovato
23        Impiegato imp = new Impiegato_Stub(result.getIndirizzo());
24        System.out.println ("Nome: " + imp.getNome());
25        System.out.println ("ID: " + imp.getID());
26        System.out.println ("Stipendio: " + imp.getStipendio());
27        System.out.println ("Aumentiamo lo stipendio di 1000 euro");
28        System.out.println ("Ora il suo stipendio è di: "+imp.aumentaStipendio(1000));
29        ((Impiegato_Stub) imp).close();
30      }
31      else // altrimenti non esiste un oggetto remoto con quel nome
32        System.out.println ("Non esiste un oggetto remoto con nome Rossi");
33    } catch (Throwable t) {
34      logger.severe("Lanciata Throwable:" + t.getMessage());
35      t.printStackTrace();
36      }
37  }
38 }
```

Fine: Client.java

Quindi, alla linea 12, il Client cerca di effettuare una ricerca per il riferimento ad un oggetto che abbia come identificativo la stringa `Rossi` (immaginiamo di usare come ID il cognome dell'impiegato). Come dall'esempio del paragrafo 2.2.4, in caso di una richiesta con un `RecordRegistro` con campo indirizzo pari a `null`, allora il server effettuava una ricerca e restituiva il record (se lo trovava) oppure `null`. Quindi, alle linee 12-19, il Client trova (se esiste) l'indirizzo IP del server dove si trova l'oggetto remoto. Se esiste l'indirizzo, allora (linee 21-30) si può effettuare la invocazione dei metodi, passando (linea 23) allo Stub l'indirizzo appena trovato. In caso contrario (linea 32) si stampa un messaggio di errore.

2.5.2 Lo Skeleton

Anche lo Skeleton va modificato principalmente nella parte iniziale. Infatti, deve anche esso ricevere su linea di comando l'host dove si trova il servizio di Registro, presso il quale si registrerà.

Impiegato_Skeleton.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.logging.Logger;
4
5 public class Impiegato_Skeleton extends Thread {
6   static Logger logger = Logger.getLogger("global");
7
```

```
 8  // Costruttore
 9  public Impiegato_Skeleton(ImpiegatoServer server) {
10    mioServer = server;
11  }
12
13  public static void main (String args[]) {
14    // Istanziazione oggetto Server
15    ImpiegatoServer impiegato = new ImpiegatoServer ("Mario Rossi", "01721", 30000);
16    // Istanziazione skeleton e sua esecuzione
17    Impiegato_Skeleton skel = new Impiegato_Skeleton(impiegato);
18    skel.start();
19    // Registrazione dell'oggetto
20    InetAddress addr=null;
21    String ipAddrStr = "";
22    try { // trovo l'indirizzo IP locale
23      addr = InetAddress.getLocalHost();
24      byte[] ipAddr = addr.getAddress();
25      // convertiamo l'indirizzo
26      for (int i=0; i<ipAddr.length; i++) {
27        if (i > 0) ipAddrStr += ".";
28        ipAddrStr += ipAddr[i]&0xFF; // unsigned bytes
29      }
30    } catch (UnknownHostException e) {
31      logger.severe("Non conosco localhost??" + e.getMessage());
32      e.printStackTrace();
33    }
34    logger.info("Registro l'oggetto all'indirizzo "+ipAddrStr);
35    RecordRegistro r = new RecordRegistro("Rossi", ipAddrStr);
36    Socket socket;
37    try {
38      socket = new Socket (args[0], 7000);
39      ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
40      sock_out.writeObject(r);
41      sock_out.flush();
42      socket.close();
43    } catch (UnknownHostException e) {
44      logger.severe("Host non conosciuto" + e.getMessage());
45      e.printStackTrace();
46    } catch (IOException e) {
47      logger.severe("Problemi sul socket per la registrazione" + e.getMessage());
48      e.printStackTrace();
49    }
50  }
51
52  // Attività dello skeleton
53  public void run() {
..  //il resto del metodo è uguale all'esempio del paragrafo 2.4.3
98  } // fine run()
99
100 // Variabili di istanza
101 ImpiegatoServer mioServer;
102 }
```

Fine: Impiegato_Skeleton.java

Questo avviene, dopo aver fatto partire l'oggetto server e il thread dello skeleton, a partire dalla linea 20. Nelle linee 20-30, si provvede a calcolare l'indirizzo IP locale (sulla conversione da un byte senza segno, quale quello dell'indirizzo IP ad intero, vedere gli approfondimenti nel paragrafo 2.7.2).

A questo punto, linea 35-46, si registra un record presso il registro con la propria ID (Rossi) e l'indirizzo calcolato. Il resto della classe è identica a quella dell'esempio nel

paragrafo 2.4.3 mostrato in precedenza.

2.6 Alcuni commenti conclusivi

Ovviamente l'esempio ha come obiettivo esclusivamente quello di dare una idea generale dei compiti che stub e skeleton svolgono. In realtà, essi svolgono altri compiti e quindi sono sicuramente più complessi di quelli esaminati in questo esempio.

Diverse sono le funzionalità e le problematiche che non abbiamo trattato in questo esempio. Innanzitutto, non ci preoccupiamo di controllare le versioni: la classe `Impiegato` potrebbe avere una versione più recente su un lato che su un altro e, di conseguenza, la comunicazione potrebbe avvenire in maniera errata (ad esempio potrebbe esserci un metodo nuovo che il lato server non può trattare). Poi, in questo esempio non abbiamo trattato la gestione di errori di comunicazione né di eccezioni (generate ad un capo della comunicazione ma che devono essere trattate all'altro capo).

In ogni caso, il protocollo di comunicazione tra stub e skeleton che abbiamo descritto risulta alquanto semplicistico e non tratta, ad esempio, della esecuzione di metodi con lo stesso nome ma con firma (elenco dei tipi dei parametri) diversa.

Anche la soluzione dell'indirizzamento è alquanto grezza: ad esempio, abbiamo inserito (per semplicità) le classi di `RecordRegistro` e di `RegistroServer` all'interno del progetto, ma non vorremmo dover fare così ogni volta che vogliamo usare un registro.

Infine, una altra considerazione da fare riguarda la contemporanea esecuzione di diverse chiamate: in effetti questa implementazione dello skeleton non permette che diversi client, contemporaneamente, possano effettuare diverse invocazioni sull'oggetto server. Servirebbe un passo successivo (che potete considerare come un esercizio) che prevede la gestione di un singolo skeleton (in multithread) per ogni client, e della gestione della esecuzione contemporanea con la sincronizzazione dei metodi.

2.7 Approfondimenti

2.7.1 Il funzionamento di `getOutputStream()`

In questo paragrafo trattiamo brevemente di un problema che può capitare e che viene risolto da una semplicissima operazione, apparentemente non spiegabile: per un socket, si deve sempre aprire prima `getOutputStream()` e poi `getInputStream()`!

Infatti, una piccola modifica al programma che mostriamo nel paragrafo 2.2.3 comporta effetti insospettati: basta invertire nel client le linee 11 e 12, quindi aprendo prima lo stream di input e poi lo stream di output e fare lo stesso sul server invertendo le linee 14 e 15 e ..., la applicazione client rimane bloccata sulla apertura dello stream di input.

Il problema è spiegato nella documentazione Java del costruttore di `ObjectInputStream()`: come prima operazione, il costruttore legge dal flusso un header di serializzazione per verificarlo. Questo costruttore si blocca fino a quando il corrispondente `ObjectOutputStream()` (all'altro capo dello stream) ha scritto degli header e fatto il flush dello stream. Ricordiamo che fare il flush dello stream significa assicurarsi che ogni byte che possa essere stato bufferizzato (per efficienza) sia stato effettivamente inviato sullo stream.

Ora, se entrambi i lati del socket creano prima lo stream di input, tutti e due i costruttori di `ObjectInputStream()` rimangono in attesa di leggere questo header, che nessuno invia e, quindi, si bloccano. Come buona pratica, quindi, si deve creare sempre prima l'`ObjectOutputStream()` di un socket e farne il `flush()` (anche se per brevità non faremo il flush in questi esempi), in modo da evitare deadlock.

2.7.2 Conversione di `Byte` unsigned

È importante notare come la conversione di un byte come un intero senza segno passa attraverso la operazione di AND bit a bit. Infatti un byte viene considerato da Java, quando convertito in intero, come un intero con segno (in complemento a due, con il bit più significativo con peso negativo). In pratica, un byte corrisponde ad un intero da -128 a +127. Ovviamente, per gli indirizzi IP si usano, invece, interi senza segno, quindi da 0 a 255.

Per evitare i problemi di conversione, una semplice (ma efficiente) scorciatoia è quella di fare l'AND bit-a-bit del byte con la costante *intera* composta da tutti 1 negli 8 bit meno significativi (vale a dire `0xFF`). Essendo questa costante di 32 bit, il byte viene promosso automaticamente al corrispondente intero e, per un byte con il bit più significativo a 1, questo significa aggiungere tanti 1 a sinistra fino ad arrivare a 32 bit. Il risultato viene trattato come un intero (quindi a 32 bit) e, quindi, si ottiene il valore "come se fosse unsigned".

Vediamo un esempio: se consideriamo il byte `b` con valore `10000010` che rappresenta -126 in complemento a due, la conversione opera secondo questi passi. Poiché la operazione coinvolge operandi interi di dimensione diversa:

```
                                  10000010  &
00000000 00000000 00000000 11111111  =
```

allora il compilatore promuove il byte ad intero (aggiungendo 24 uni a sinistra del bit più significativo):

```
11111111 11111111 11111111 10000010  &
00000000 00000000 00000000 11111111  =
-----------------------------------
00000000 00000000 00000000 10000010
```

ottenendo il risultato, che è un intero il cui valore è quello del byte convertito ad unsigned int (cioè 130).

Note bibliografiche

L'esempio illustrato nella sezione 2.1 è ispirato da un (molto più semplice) esempio mostrato in [32].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[E*]** Rendere iterativo il server dell'esempio di Hello con i socket inserendo la `accept()` e la risposta verso il client all'interno di un `while (true)`.

2. **[E**]** Rendere il server sviluppato nell'esercizio precedente multi-thread, in modo che ad ogni accept, si faccia partire un thread (che gestisce l'invio della risposta al client) permettendo al server di tornare subito alla `accept()` successiva.

3. **[E*]** Creare una applicazione client-server basata su socket TCP, che permette ad ogni client di inviare un intero al server (fornito dall'utente su linea di comando) e di stampare l'intero che riceve ed uscire; il server fa la somma di tutti gli interi ricevuti, restituendo ad ogni client la somma parziale fino ad ora ottenuta.

4. **[E**]** Modificare l'esercizio precedente in modo che il server sia multi-thread, permettendo a diversi client di poter eseguire contemporaneamente l'invio dell'intero.

Capitolo 3

Presentazione di Java Remote Method Invocation

Indice

3.1 Introduzione 54
3.2 Gli obiettivi della progettazione di Java RMI 54
3.3 Il modello a oggetti distribuiti di Java RMI 56
3.3.1 La struttura delle classi di Java RMI 56
3.3.2 Il meccanismo di invocazione remota 58
3.3.3 La differenza tra il modello a oggetti locale e quello remoto 61
3.4 La architettura di Java RMI 62
3.4.1 I tre layer della architettura 62
3.4.2 Distributed Garbage Collection 66
3.4.3 Caricamento dinamico delle classi 67
3.5 Approfondimenti 68
3.5.1 L'eterogeneità nelle tecnologie ad oggetti distribuiti 68
3.5.2 La trasparenza degli oggetti distribuiti 70
3.5.3 La sicurezza in Java 74
3.5.4 Il meccanismo di marshalling usato da Java RMI 75
Note bibliografiche 76
Spunti per lo studio individuale 78

3.1 Introduzione

Java Remote Method Invocation (Java RMI) è una libreria di Java che permette lo sviluppo di applicazioni distribuite, fornendo la possibilità di effettuare comunicazione remota tra programmi scritti in Java. Infatti, Java RMI offre ad un oggetto in esecuzione su una Java Virtual Machine la possibilità di invocare metodi di un oggetto in esecuzione in una altra JVM, anche se essa si trova su una macchina differente.

Il ruolo che viene ricoperto da Java RMI all'interno della Java Platform è quello di *integration library* (libreria per la integrazione) situata immediatamente al di sopra delle librerie base del linguaggio, e fa parte delle specifiche del linguaggio sin dalla versione 1.1 di Java.

Le applicazioni RMI seguono tipicamente[1] una architettura client-server dove il server crea un certo numero di oggetti server accessibili da remoto e attende che gli oggetti client ne utilizzino i servizi, compiendo invocazioni remote sui metodi che espongono. In questo capitolo presentiamo il funzionamento di base di Java RMI per poi illustrare gli obiettivi perseguiti durante la progettazione e, infine, la sua architettura.

3.2 Gli obiettivi della progettazione di Java RMI

Java Remote Method Invocation è stato progettato all'interno della Sun per affiancare all'allora nascente linguaggio Java il supporto per oggetti distribuiti. Il team, condotto da Jim Waldo, ha basato sicuramente il proprio lavoro sulla esperienza con i primi sistemi RPC, sulla quale Sun molto aveva investito negli anni '80 con Open Network Computing, ma anche sulla ricerca (anche al di fuori della Sun) che era stata compiuta sugli oggetti distribuiti in altri linguaggi, come quella con i *Network Objects* di Modula-3.

Particolarmente rilevante era la esperienza maturata in quegli anni per la standardizzazione di CORBA, in quanto Jim Waldo aveva partecipato in maniera significativa alla prima specifica di CORBA quando lavorava per la HP/Apollo. Quindi, la proposta di Java RMI stabiliva obiettivi che, basati sulle esperienze precedenti, potevano garantire una efficace implementazione del modello ad oggetti distribuiti.

Gli obiettivi di Java RMI puntano ad assicurare la *semplicità* e *integrazione* della implementazione. È importante, innanzitutto che il sistema ad oggetti distribuiti di Java RMI sia semplice per un agevole utilizzo ed implementazione. Questo ne permette la diffusione ma impedisce anche che utilizzi errati e scorretti delle potenzialità del sistema possano creare problemi ai sistemi realizzati. Java RMI è integrato all'interno del linguaggio Java e questa scelta di ancorare l'ambiente ad un singolo linguaggio (oltre ad essere dettata chiaramente dalle politiche aziendali della Sun) permette di offrire un ambiente che sia naturale per il programmatore, che conosce e sa usare al meglio le caratteristiche del linguaggio di programmazione che già usa per implementare logica della applicazione e presentazione.

Analizziamo ora, singolarmente, gli obiettivi che si poneva la realizzazione di Java Remote Method Invocation.

Invocazione trasparente di metodi remoti

Java RMI deve poter offrire al programmatore un meccanismo semplice per la invocazione di metodi che sono offerti da un oggetto remoto. La invocazione deve avvenire fornendo al programmatore la "illusione" che essa avvenga su un oggetto che risiede all'interno dello stesso spazio di indirizzamento utilizzato dall'oggetto che compie la invocazione.

[1] La architettura client-server è quella che meglio si presta alle invocazioni remote, ma non è un requisito indispensabile, è possibile creare applicazioni, ad esempio, peer2peer in Java RMI.

Integrazione in Java

Una importante caratteristica è quella di fare in modo che il modello distribuito si integri in maniera naturale all'interno del linguaggio Java. Questo permette di offrire un ambiente familiare allo sviluppatore Java, che può usare gli stessi strumenti, modelli e astrazioni che vengono utilizzati per oggetti locali. Ciò deve avvenire cercando di preservare, quanto più possibile, la semantica degli oggetti che sono all'interno del linguaggio. Le somiglianze e le differenze tra il modello ad oggetti locale e quello distribuito verranno illustrate nel paragrafo 3.3.3

La integrazione in Java deve essere quanto più ampia possibile. Per questo motivo, Java RMI fornisce un garbage collector distribuito in modo da preservare la modalità di gestione della memoria di Java che solleva il programmatore dal doversi occupare esplicitamente della allocazione e deallocazione della memoria. Nei linguaggi dove la gestione della memoria è a carico del programmatore (come C/C++), la allocazione/deallocazione a richiesta (tramite, ad esempio, `malloc()` e *free()*) rende la esecuzione del programma più efficiente (non c'è attività di garbage-collection che periodicamente deve essere effettuata) ma in caso di errori di programmazione (i cosiddetti *memory-leak*) un programma può continuare ad allocare memoria per oggetti e non deallocarne mai, esaurendo in breve lo spazio di memoria a disposizione. Questo è particolarmente pericoloso per applicazioni distribuite in quanto i server sono in esecuzione continua: anche il più piccolo memory-leak può portare in pochi giorni un server a esaurire la memoria a disposizione. La Garbage Collection distribuita viene illustrata con maggiori dettagli nel paragrafo 3.4.2.

Non-trasparenza della natura locale/remota di un oggetto

Nonostante l'obiettivo di assicurare la semplicità di uso per il programmatore, esistono diverse caratteristiche del linguaggio che *non devono* essere nascoste al programmatore. Quindi, il fatto che un oggetto sia remoto oppure locale deve essere chiaro ed evidente, sia in fase di progettazione che in fase di implementazione. Le motivazioni per questa *non-trasparenza* sono descritte ampiamente all'interno del paragrafo 3.5.2.

Rendere minima la complessità di client e server

Comunque, si deve assicurare la minima complessità alla applicazione distribuita basata su Java RMI, compatibilmente con gli altri obiettivi. In particolare, il livello di complicazione che viene introdotto da un oggetto distribuito per l' *oggetto client*[2] (cioè l'oggetto che compie la invocazione remota) e per l'*oggetto server* (cioè l'oggetto che riceve ed esegue la invocazione) deve essere limitato.

Preservare la sicurezza fornita da Java

Il modello ad oggetti distribuito fornito da Java RMI non deve alterare il livello di sicurezza che viene offerto dalla piattaforma Java. Infatti, sin dalla presentazione del linguaggio, la "sicurezza" e la "robustezza" del linguaggio sono state al centro delle attenzioni dei progettisti, principalmente a causa della natura distribuita del linguaggio, che prevede la esecuzione in locale di programmi che vengono scaricati dalla rete (come, ad esempio, gli applet).

La sicurezza del linguaggio Java consiste nell'eseguire una applicazione (od un applet) all'interno di una *sandbox*, un ambiente dedicato, ristretto e controllato, all'interno del

[2] In questo contesto, l'uso del termine client/server è ristretto alla singola invocazione (e quindi viene riferito agli *oggetti*) e non influenza la architettura della applicazione distribuita.

quale le operazioni che il programma può eseguire non risultano pericolose (sia accidentalmente che intenzionalmente). Diverse componenti del linguaggio Java contribuiscono ad assicurare la sicurezza del linguaggio: una loro analisi più approfondita è presentata nel paragrafo 3.5.3.

Modalità di invocazione

Java RMI deve prevedere la possibilità che esistano diversi tipi di invocazione, fornendo quella di tipo *unicast* da un client verso un server ma permettendo (in futuro) anche la estensione verso invocazioni di tipo *multicast* vale a dire verso diversi server replicati[3]. Inoltre deve essere possibile che l'oggetto server sia attivato solo al momento della invocazione e che i riferimenti ad oggetti persistenti siano persistenti.

Livelli di trasporto multipli

Infine, Java RMI deve essere aperto verso future espansioni che prevedano che il protocollo di trasporto (basato su socket) che viene utilizzato possa essere modificato. Questa caratteristica è stata particolarmente utile nella interazione di Java RMI con l'ambiente CORBA, utilizzando il protocollo Internet-InterORB Protocol (IIOP) come livello di trasporto.

3.3 Il modello a oggetti distribuiti di Java RMI

Un *oggetto remoto* è un oggetto i cui metodi possono essere acceduti da un altro spazio di indirizzamento, e potenzialmente da un'altra macchina. La descrizione dei servizi offerti da remoto da un oggetto remoto è contenuta all'interno di una *interfaccia remota* che è una interfaccia Java che dichiara i metodi remoti. Una *invocazione di metodi remoti* (Remote Method Invocation) rappresenta la invocazione di un metodo su un oggetto remoto (specificato nella interfaccia remota) e ha la stessa sintassi di una invocazione di un metodo locale. Questo rappresenta una importante facilitazione per il programmatore che non deve pagare nessuna complessità addizionale per poter usare una invocazione remota (ma commenteremo ampiamente su questo nel paragrafo 3.5.2).

L'oggetto client di oggetti remoti server utilizza esclusivamente la interfaccia remota dell'oggetto, non la sua implementazione. Questo garantisce che le funzionalità remote risultino astratte verso il client e disaccoppia le due implementazioni, permettendo, ad esempio, evoluzioni dell'oggetto server (cioé della sua implementazione) senza che il client debba essere modificato.

3.3.1 La struttura delle classi di Java RMI

Java RMI è contenuto in 5 package: `java.rmi` e `java.rmi.server` che contengono il meccanismo basilare di funzionamento delle invocazioni remote, `java.rmi.activation` per gli oggetti attivabili, `java.rmi.dgc` per la Distributed Garbage Collection e `java.rmi.registry` per il servizio di localizzazione.

[3] In effetti questa estensione non è poi avvenuta, probabilmente perché Java RMI viene ora utilizzato per la comunicazione di sistemi di Enterprise che prevedono tecniche a più alto livello per gestire server replicati.

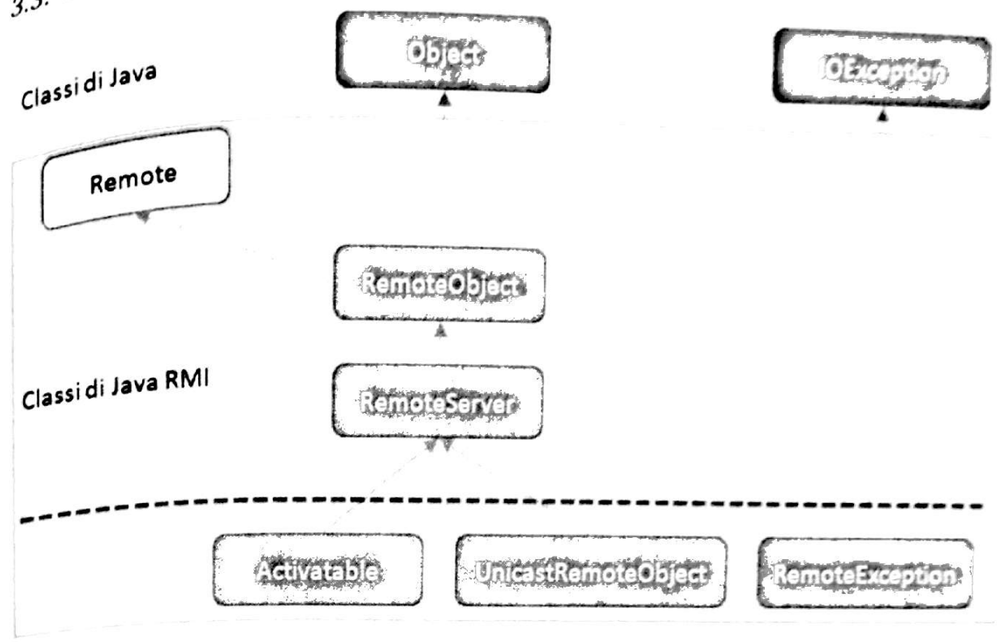

Figura 3.1: *Le relazioni tra le classi di Java RMI e le classi di Java (le dipendenze con linea continua indicano* `extends` *mentre quella linea tratteggiata indica* `implements`*). Le classi al di sotto della linea doppia tratteggiata sono quelle che il programmatore usa, di solito, per la programmazione con Java RMI.*

Interfacce ed eccezioni remote

Prima di definire un oggetto remoto, si deve definire necessariamente una interfaccia remota per l'oggetto, in modo che vengano esposti i servizi che l'oggetto remoto intende mettere a disposizione per un utilizzo da parte dei client.

Una *interfaccia remota* per Java RMI deve estendere (implementare) la interfaccia `java.rmi.Remote` che è una interfaccia cosiddetta *marker*, cioè una interfaccia vuota che, in questo caso, serve solamente per poter segnalare che essa definisce dei metodi accessibili da remoto.

Ogni metodo descritto in una interfaccia remota deve essere un *metodo remoto* cioé deve soddisfare entrambe le seguenti condizioni:

- Un metodo remoto deve dichiarare esplicitamente la eccezione `java.rmi.RemoteException`. Questa è una delle conseguenze delle considerazioni circa la non-trasparenza che deve avere un oggetto distribuito: poiché la semantica dei malfunzionamenti di un oggetto remoto è profondamente diversa da quella dei malfunzionamenti di un oggetto locale (ed i motivi verranno ampiamente trattati nel paragrafo 3.5.2) allora il programmatore deve rendere esplicita la natura remota dei metodi nella dichiarazione della interfaccia remota. In questa maniera si forza lo sviluppo successivo, che comporterà la invocazione di questi metodi, a gestire la eccezione remota in maniera esplicita, in quanto il compilatore controlla che la eccezione sia gestita (*checked exception*). Va notato come, comunque, questa eccezione remota deriva dalla "tradizionale" `java.io.IOException` del linguaggio Java.

- I parametri remoti di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota, non utilizzando la classe della implementazione remota. Questo

permetterà di poter passare riferimenti remoti sia come parametri che come valori restituiti, come sarà chiaro nel paragrafo 3.3.2.

Il meccanismo della interfaccia remota aggiunge, in pratica, un livello ulteriore di accessibilità ai modificatori di accesso dei metodi (`public`, `protected`, etc.): i metodi remoti, dichiarati in una interfaccia remota sono *più* accessibili dei metodi `public` che risultano accessibili ma solamente da invocazioni all'interno della stessa macchina virtuale.

Va infine ricordato come nelle interfacce (e quindi anche in quelle remote) sia possibile definire delle costanti. Questo rappresenta uno strumento interessante per poter condividere dati costanti tra le componenti della applicazione distribuita.

Implementazioni remote

Per realizzare la *implementazione remota* che deriva (`implements`) da una interfaccia remota per offrire verso l'esterno i metodi remoti in essa definiti, si può procedere in due modi:

- il primo (detto di *riuso della implementazione remota*) prevede che la classe che contiene l'implementazione dell'oggetto derivi esplicitamente da `java.rmi.server.UnicastRemoteObject` ereditando di conseguenza il comportamento definito dalle classi `java.rmi.server.RemoteObject` e `java.rmi.server.RemoteServer`;

- la seconda modalità (detta *classe di implementazione locale*) permette che la classe derivi il comportamento da qualche altra classe (non remota) e che si debba quindi occupare esplicitamente di esportare l'oggetto (tramite il metodo statico `exportObject()` di `java.rmi.server.UnicastRemoteObject`) e di implementare la semantica di alcune operazioni di `Object` per oggetti remoti che sono ridefiniti in `java.rmi.server.RemoteObject` e `java.rmi.server.RemoteServer`.

Va sottolineato come la implementazione remota può implementare anche altri metodi, oltre a quelli remoti definiti nella interfaccia remota ma che questi saranno accessibili esclusivamente in locale, secondo le direttive di accesso fornite dai modificatori di accesso.

3.3.2 Il meccanismo di invocazione remota

Riferimenti remoti

Nel modello distribuito, gli oggetti client interagiscono con un oggetto *stub* (detto anche surrogato) che espone localmente esattamente le stesse interfacce remote definite dall'oggetto remoto.

In questa maniera, lo stub rappresenta la interfaccia remota dell'oggetto remoto in locale, sulla macchina dove l'oggetto client è in esecuzione. Dal punto di vista della Java Virtual Machine del client, infatti, il tipo dello stub è lo stesso di quello dell'oggetto server (per quanto riguarda la parte accessibile da remoto, rappresentata dall'insieme di interfacce remote che implementa). Quindi, il client può accedere tradizionalmente al tipo di un oggetto remoto, controllando quale interfaccia remota implementa, attraverso `istanceof`.

Il sistema fornisce anche un meccanismo per il caricamento dinamico dello stub, per rendere dinamicamente disponibile lo stub, a run-time, agli oggetti client, ma il funzionamento verrà descritto di seguito, nel paragrafo 3.4.3.

Localizzazione e invocazione di oggetti remoti

Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il riferimento remoto. Questo può essere reperito in due maniere: *(a)* ottenuto come risultato di altre invocazioni (locali o remote) di metodi; *(b)* attraverso un servizio di directory.

La prima modalità è standard per quanto riguarda le invocazioni locali e sarà approfondita nel paragrafo successivo che tratta come vengono passati i parametri durante le invocazioni remote.

Per quanto riguarda il secondo metodo, invece, Java RMI fornisce un semplice meccanismo di *name server* nella classe `java.rmi.Naming`, che permette di gestire riferimenti a oggetti remoti accessibili specificando un ID (stringa). Tale classe fornisce metodi per ricercare (`lookup()`), registrare (`bind()`, `unbind()`, `rebind()`) ed elencare (`list()`) gli identificativi registrati, accedendo al servizio attraverso una specifica che segue lo standard Uniform Resource Locator (URL). Questo meccanismo è utilizzabile anche come tool da linea di comando (*rmiregistry*), eseguito sulla macchina sulla quale l'oggetto server si trova. Un meccanismo di localizzazione più robusto verrà proposto successivamente quando tratteremo della implementazione di RMI su IIOP, basato su Java Naming and Directory Interface.

La invocazione di un metodo remoto ha la stessa sintassi di una invocazione locale. Poiché i metodi remoti devono includere `RemoteException` nella propria firma, il codice dell'oggetto client viene forzato dal compilatore a gestire questo possibile malfunzionamento della chiamata remota, in aggiunta ad altre eccezioni che dipendono dalla semantica della applicazione.

Il significato di una `RemoteException` durante una invocazione remota non è chiaro: l'unica cosa che il client conosce è che qualche problema di comunicazione è avvenuto prima, durante o dopo la chiamata. Questo significa che il client non sa se la invocazione è stata effettuata e che magari l'errore sia capitato solamente nella restituzione del valore oppure se il metodo non è stato proprio invocato. Quindi, per poter preservare la semantica delle operazioni, è bene che i metodi remoti siano progettati in maniera tale da essere idempotenti, vale a dire che ripetute applicazioni dello stesso metodo con lo stesso parametro devono generare lo stesso risultato. I metodi locali, invece, non devono obbedire a questi requisiti stringenti di idempotenza. Questo è un esempio delle differenze sostanziali che devono esserci tra gli oggetti locali e quelli remoti, trattati con maggiore dettaglio nel paragrafo 3.5.2.

Passaggio di parametri

Un metodo remoto può dichiarare solo parametri o valori restituiti che siano serializzabili, vale a dire che implementino la interfaccia `Serializable`. Le classi dei parametri o dei valori restituiti che non siano disponibili localmente vengono scaricate dinamicamente.

Un *oggetto locale* passato come parametro o restituito come valore da una invocazione remota viene passato per copia, vale a dire che il contenuto dell'oggetto viene copiato prima di essere serializzato dal meccanismo di serializzazione di Java. Quindi la semantica per il passaggio di parametri locali è diversa da quella di Java che, ricordiamo, passa il riferimento di un oggetto.

Quindi, quando vengono passati come parametri più riferimenti allo stesso oggetto ad una altra macchina virtuale utilizzando invocazioni diverse, allora i riferimenti sulla macchina server saranno ad oggetti distinti. Ad esempio, se `A` è un riferimento ad un oggetto remoto che offre il metodo remoto `comunicaInizio(Date x)` , e se `B` e `C` sono riferimenti a oggetti locali di tipo Date, dopo avere eseguito il codice:

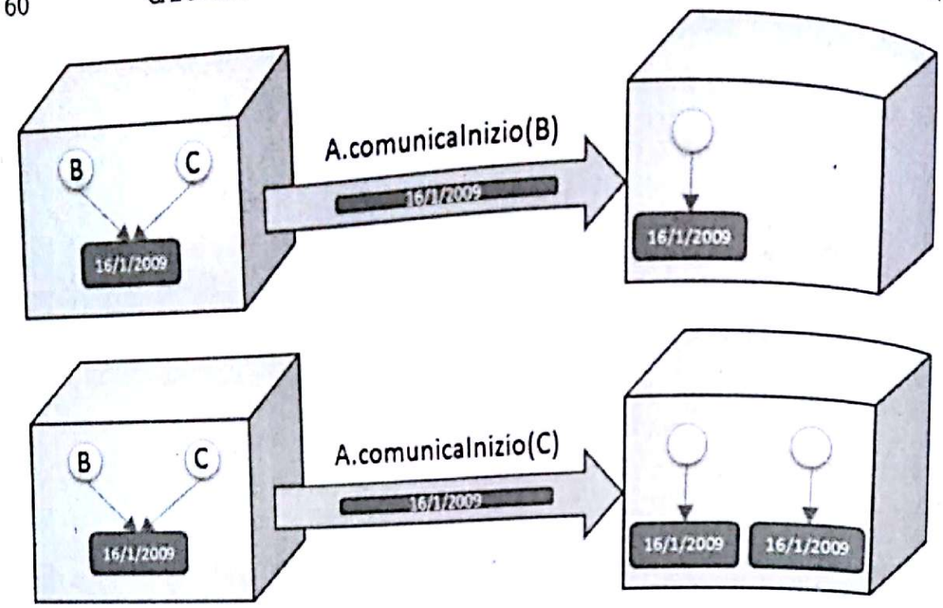

Figura 3.2: *Le due invocazioni, separate, che passano due riferimenti allo stesso oggetto, il cui effetto è quello di ottenere due oggetti distinti sulla macchina server.*

```
Date B = new Date();
Date C = B;
A.comunicaInizio(B);
A.comunicaInizio(C);
```

sulla macchina remota, ci saranno due oggetti diversi di tipo `Date()` a cui punteranno le due variabili, mentre sulla macchina locale, dalla quale provenivano, essi facevano riferimento allo stesso oggetto. L'esempio delle due invocazioni viene mostrato in Figura 3.2.

Il meccanismo implementato da Java RMI assicura la cosiddetta *integrità referenziale*: quando vengono passati più riferimenti allo stesso oggetto nella *stessa invocazione*, allora viene garantito che anche sulla macchina remota alla quale sono stati passati i riferimenti punteranno allo stesso oggetto. Continuando l'esempio precedente, se l'oggetto `A` fornisse anche il metodo remoto `comunicaInizioFine(Date x, Date y)`, allora dopo avere eseguito il codice:

```
Date B = new Date();
Date C = B;
A.comunicaInizioFine(B, C);
```

sulla macchina remota ci sarà un solo oggetto `Date` a cui punteranno i due riferimenti passati come parametri. I due esempi vengono illustrati in Figura 3.3.

Quando si passa un *oggetto remoto* come parametro o lo si ottiene come valore restituito, viene passato il suo stub, e non la implementazione.

Nel paragrafo 3.5.4 vengono forniti i dettagli sul funzionamento di questo meccanismo di marshalling, basato sulla specializzazione effettuata da Java RMI della serializzazione usata sullo stream di output e di input.

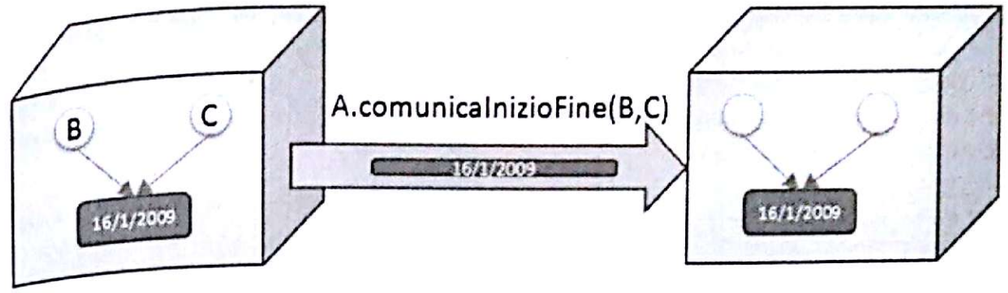

Figura 3.3: *Quando due riferimenti locali allo stesso oggetto vengono passati attraverso la stessa invocazione, la integrità referenziale fa sì che sul server essi puntino allo stesso oggetto.*

3.3.3 La differenza tra il modello a oggetti locale e quello remoto

Innanzitutto, va tenuto presente che uno degli obiettivi di Java RMI è di mantenere simile la maniera di programmare oggetti locali e remoti. Quindi, per quanto possibile (e per quanto *giusto*, maggiori dettagli nel paragrafo 3.5.2) i due modelli sono stati differenziati il meno possibile.

Una prima modifica necessaria è stata quella della implementazione di alcuni metodi della classe `Object` che non sono appropriate per il contesto distribuito e che sono state rimpiazzate dalla implementazione di `java.rmi.server.RemoteObject`, che ridefinisce i metodi `equals()`, `hashCode()`, `toString()` come segue:

- Il metodo `X.hashCode()` viene ridefinito in maniera che restituisca lo stesso codice per due stub diversi di oggetti remoti che si riferiscono allo stesso oggetto remoto. In questa maniera gli oggetti remoti (o meglio i loro stub, che sono quelli che vengono serializzati e trasmessi) possono essere utilizzati come chiavi nelle tabelle hash.

- Il metodo `X.equals()` restituisce un booleano che è vero se il riferimento remoto passato è uguale a quello di `X`. In effetti, il confronto viene fatto sugli stub e quindi la uguaglianza è effettuata sui riferimenti e non sul contenuto, visto che per poter controllare il contenuto di un oggetto remoto si dovrebbe fare una chiamata remota, che può generare una `RemoteException` che non viene lanciata[4] dalla firma del metodo `equals()` come definito in `Object()`.

  Se l'oggetto passato `Y` non è un oggetto remoto, allora il risultato viene delegato al risultato di `Y.equals(X)` che quindi permette di verificare la uguaglianza tra uno stub locale e un oggetto remoto (di cui si usa lo stub).

  Questo metodo viene utilizzato, in particolare, nei metodi di accesso alle tabelle hash, quando si devono fare confronti per determinare cosa si deve restituire.

- Il metodo `toString()` su un oggetto remoto deve poter restituire informazioni addizionali circa la macchina sulla quale quell'oggetto remoto si trova, oltre alle consuete informazioni circa il nome della classe ed un codice hash (come nella implementazione standard del metodo in `Object`).

Va ricordato che queste modifiche al comportamento dei metodi non vengono effettuate qualora si usi il metodo della implementazione locale tramite la classe di implementazione locale (con il metodo `exportObject()`) e quindi, è responsabilità del programmatore

[4] Ovviamente, anche se il linguaggio lo permettesse in qualche maniera, ci sarebbero delle serie conseguenze circa l'efficienza, visto che il metodo `equals()` è un metodo che può essere chiamato molto spesso.

prendersi cura del corretto comportamento dell'oggetto remoto, ad esempio, qualora il riferimento fosse utilizzato in una tabella hash.

Infine, a differenza degli oggetti locali, i client di oggetti remoti (cioé gli oggetti che invocano i metodi) non interagiscono con la implementazione dell'oggetto remoto ma solo con la interfaccia remota.

Rispetto al *passaggio di parametri*, questo è possibile in maniera trasparente nei due modelli, nel senso che i riferimenti remoti possono essere passati a (o restituiti da) metodi come parametri (o valori restituiti). La semantica del passaggio di parametri differisce, invece, in quanto argomenti locali a invocazione remote sono passati per copia invece che per riferimento, mentre quelli remoti sono passati per riferimento.

Per quanto riguarda la *gestione dei tipi*, si può effettuare il casting Java di un oggetto remoto ad una qualsiasi delle interfacce remote supportate dalla implementazione dell'oggetto utilizzando la sintassi del linguaggio e si può utilizzare l'operatore `istanceof` per verificare a tempo di esecuzione le interfacce supportate da un oggetto remoto, applicando l'operatore al riferimento remoto.

Infine, la differenza nella *invocazioni di metodo* consiste nel fatto che le invocazioni remote forzano il programmatore (attraverso il meccanismo di *checked exception* utilizzato dal compilatore) a dover gestire esplicitamente i fallimenti di invocazioni di metodi remoti (le eccezioni remote); a differenza delle invocazioni locali che, di norma, non richiedono l'intervento del programmatore. Questo è dovuto al fatto che la invocazione di metodo remota è intrinsecamente più complessa e i malfunzionamenti possibili più numerosi e difficili da trattare, e il programmatore deve essere cosciente di questo (maggiori dettagli verranno dati nel paragrafo 3.5.2).

3.4 La architettura di Java RMI

3.4.1 I tre layer della architettura

Il sistema di Java RMI è strutturato su tre livelli (*layer*), mostrati nella Figura 3.4:

- *Stub/Skeleton Layer* che comprende gli stub lato client e gli skeleton lato server;
- *Remote Reference Layer* che specifica il comportamento della invocazione e la semantica del riferimento (unicast, multicast, etc.);
- *Transport Layer* che si occupa della connessione e della sua gestione.

La applicazione dell'utente si trova in cima a questi livelli e interagisce (in maniera moderata) con il livello di stub/skeleton.

Una architettura di questo tipo, a livelli, permette di astrarre le funzionalità fornite da un livello verso il livello immediatamente superiore, con il quale comunica attraverso un protocollo ben definito, ma che si basa solamente sulle funzionalità offerte e non sulla loro implementazione. In questa maniera la architettura può evolvere mantenendo la sua struttura tramite la sostituzione di un livello con un altro equivalente (dal punto di vista dei servizi offerti verso l'alto e richiesti verso il basso). Questa struttura si è provata lungimirante, in quanto RMI ha potuto assecondare la evoluzione del calcolo distribuito, fornendo, ad esempio, diversi protocolli di trasporto, come IIOP oppure introducendo gli oggetti attivabili su richiesta (da JDK 1.2) cambiando il livello di trasporto nel primo caso e aggiornando il livello di reference nel secondo.

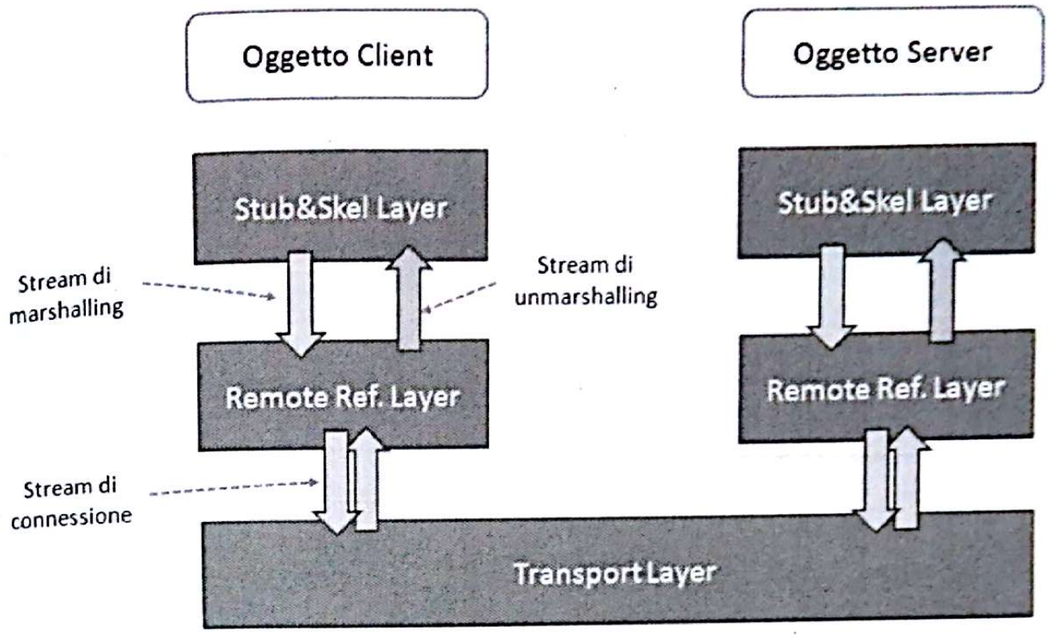

Figura 3.4: *La architettura a layer di RMI con la esplicitazione della tipologia di stream utilizzati per comunicare tra i layer.*

Prima di esaminare in dettaglio i layer, ricapitoliamo i passi che vengono compiuti da Java RMI, simili a quelli che abbiamo già preliminarmente introdotto nell'esempio del capitolo 2.

- Un client che invoca un metodo su un oggetto server remoto, fa uso di uno stub per portare a termine la sua richiesta in quanto il riferimento remoto che è in possesso del client è solamente un riferimento al suo stub, presente in locale.
- Lo stub implementa l'interfaccia remota dell'oggetto remoto (verso il client) e inoltra le richiesta all'oggetto server attraverso il remote reference layer del client.
- Il remote reference layer del client si occupa di gestire la semantica delle invocazione remota lato client (se, ad esempio, si tratta di una invocazione unicast oppure multicast).
- Il livello di trasporto si occupa di stabilire la connessione con la macchina remota e della successiva gestione della connessione, curando il *dispatching* (recapito) delle invocazioni verso gli oggetti remoti. A questo scopo, inoltra la richiesta al livello di reference del server.
- Il livello di reference del server si occupa di inoltrare la richiesa allo skeleton e di curare la semantica delle invocazione lato server (gestendo, ad esempio, i riferimenti ad oggetti persistenti per curarne la loro attivazione).

Stub/Skeleton Layer

Essendo il livello più alto di Java RMI, esso si occupa di essere la interfaccia tra la applicazione (cioé le classi Java scritte dal programmatore) ed il resto del sistema[5]. Questo livello, al pari dei livelli inferiori, è stato modificato, via via che le versioni di Java aggiornavano il linguaggio, semplificando sempre di più il compito del programmatore, ma non è scomparso, in quanto nelle nuove versioni i compiti svolti da stub e skeleton sono stati solamente implementati in maniera diversa, evitando di dover necessariamente richiedere uno sforzo da parte del programmatore. La interfaccia verso il basso, in direzione del remote reference layer, consiste nel fornire uno *stream di marshal* di oggetti Java che vengono passati al remote reference layer. Il meccanismo di marshalling viene descritto in dettaglio nel paragrafo 3.5.4. Gli oggetti che vengono passati al remote reference layer vengono passati per copia.

Lo stub è incaricato di:

- iniziare la connessione con la macchina virtuale remota, chiamando il remote reference layer;
- effettuare il marshalling verso uno stream di marshal, fornito dal layer di remote reference;
- attendere il risultato della invocazione;
- effettuare l'unmarshalling dei valori restituiti (o delle eccezioni verificatesi);
- restituire il valore verso l'oggetto client che ha richiesto la invocazione.

Lo skeleton è incaricato di effettuare il *dispatching* verso l'oggetto remoto, vale a dire, curare che la invocazione sia effettuata sull'oggetto remoto in attesa di invocazioni.

Quando uno skeleton riceve una invocazione in entrata, si occupa di:

- effettuare l'unmarshalling dal remote reference layer (lato server) dei parametri per la invocazione;
- invocare il metodo sulla implementazione che si trova nella sua JVM;
- effettuare il marshalling del valore restituito (compreso di eventuali eccezioni) verso chi ha invocato il metodo.

Stub e skeleton vengono creati dall' *RMI compiler* `rmic`, un tool fornito con il JDK, che, a partire da una classe compilata che rappresenta la implementazione di un oggetto remoto, ne genera i file stub e skeleton, con il nome della classe a cui viene, rispettivamente, aggiunto alla fine `_Stub` e `_Skel`.

Sembra evidente che i compiti dello skeleton sono meno complessi di quello dello stub, ed in effetti, uno dei primi miglioramenti ottenuti dalle successive versioni è stato proprio quello di modificare il protocollo usato dallo stub che elimina la necessità di skeleton (Java 2 SDK) in quanto codice generico (spostato nel layer sottostante) viene usato per realizzare i compiti dello skeleton.

È importante notare che non è corretto dire che, a partire da Java 2, siano stati "eliminati" gli skeleton, in quanto i compiti che venivano effettuati da essi sono solamente effettuati altrove, in maniera generica, e che, quindi, va enfatizzato che il layer stub/skeleton

[5]Per ragioni didattiche, presentiamo in questo capitolo la versione originale, JDK 1.1, basata su stub e skeleton, mentre nei capitoli successivi introdurremo le due modifiche sostanziali che sono intervenute successivamente alla presentazione della prima versione di Java RMI, eliminando (solo apparentemente!) prima lo skeleton (JDK 2) e poi anche lo stub (JDK 5) dalla "visione" del programmatore.

è comunque esistente nelle versioni successive. Il miglioramento ha riguardato, effettivamente, solamente l'utilizzo da parte del programmatore, il cui lavoro è stato ridotto, a spese di una maggiore complessità (a carico della piattaforma RMI) e di una lieve perdita di efficienza.

A partire, poi, da Java 5, è stata anche eliminata la necessità di generare gli stub in maniera statica, liberando il programmatore dall'utilizzo di `rmic`: ancora una volta, i compiti dello stub sono rimasti e sono eseguiti da un oggetto `Proxy` che viene generato dinamicamente. Quindi, in conclusione, il layer stub/skeleton rimane presente (anche se maggiormente o totalmente nascosto) in tutte le versioni di Java, fino a quella attuale, Java 6. Maggiori dettagli su questa evoluzione si trovano nel paragrafo 8.3.

Remote Reference Layer

Questo layer si occupa di interfacciare il livello di trasporto con quello di stub/skeleton fornendo e supportando la semantica della operazione di invocazione di un metodo. In un senso, si occupa della parte di protocollo indipendente dalle invocazioni remote specifiche, che vengono gestite da stub e da skeleton, creati da `rmic`.

Infatti, ogni implementazione di oggetto remoto può scegliere un protocollo generale che determina le modalità di invocazione, fissato per la durata di vita dell'oggetto. Ad esempio, diverse sarebbero, in questa maniera, le modalità permesse alle invocazioni:

- invocazioni *unicast*, vale a dire da un singolo client verso un singolo server;
- invocazioni *multicast*, vale a dire un singolo client fa una invocazione ad una "batteria" di server replicati, in maniera da poter garantire la ridondanza: se uno di questi è in esecuzione allora risponderà alla invocazione (e si dovrebbe definire cosa si deve fare con le invocazioni successive);
- invocazioni di *oggetti attivabili*: le invocazioni potrebbero essere effettuate ad un oggetto remoto che è persistente, vale a dire viene attivato se arrivano delle invocazioni;
- invocazioni con riconnessione: le invocazioni potrebbero tentare connessioni alternative se l'oggetto remoto originariamente contattato non risponde alla invocazione.

Si deve, però, immediatamente precisare che Java RMI, attualmente, implementa esclusivamente la modalità unicast (presente sin dalla prima presentazione in JDK 1.1) e la modalità di invocazione di oggetti attivabili (sin dalla versione Java 2).

Il protocollo di invocazione utilizza le due componenti client e server del remote reference layer. Il lato client ha informazione circa il server della invocazione (se unicast) e comunica attraverso il livello di trasporto verso il lato server dello stesso layer. Durante la invocazione, da stub a skeleton e ritorno, il remote reference layer può intervenire per forzare uno specifico protocollo di invocazione. Ad esempio, nel caso di una invocazione multicast[6] la parte client del livello può inoltrare la richiesta ad un insieme di server, e selezionare la prima risposta che arriva, scartando le altre. Il lato server, invece viene utilizzato per la attivazione di oggetti persistenti in caso di invocazione remota.

Questo layer fornisce verso l'alto (lo stub/skeleton layer) un riferimento ad un oggetto che implementa la interfaccia `java.rmi.server.RemoteServer`, che espone un metodo `invoke()` per effettuare l'inoltro della invocazione, che viene chiamato dallo stub. Va ricordato che le modifiche intercorse tra la versione JDK 1.1 e Java 2 che hanno portato ad eliminare lo skeleton, sono state ottenute proprio modificando la semantica delle operazioni compiute da `RemoteServer`.

[6]Ricordiamo che Java RMI non offre invocazioni multicast.

Il remote reference layer interagisce verso il basso con il livello di trasporto, verso il quale utilizza la astrazione di una connessione orientata ai flussi (stream di connessione). Questa astrazione viene fornita dal livello di trasporto che, però, non è obbligato a realizzare una connessione e potrebbe utilizzare protocolli *connectionless* senza alterare la modalità con cui il remote reference layer comunica i dati.

Transport Layer

Il livello di trasporto ha il compito di:

- stabilire la connessione verso macchine con indirizzi IP remoti;
- gestire le connessioni e monitorare il loro stato;
- rimanere in ascolto per connessioni in arrivo;
- gestire una tabella degli oggetti remoti che risiedono nello spazio di indirizzamento locale;
- stabilire una connessione per le chiamate in entrata;
- identificare l'oggetto dispatcher a cui inoltrare la connessione.

Il protocollo utilizzato da RMI è chiamato Java Remote Method Protocol, un protocollo proprietario Sun. Ma la modularità ed astrazione fornita dalla architettura a livelli ha permesso l'inserimento di vari altri protocolli. Il più importante, sicuramente è la introduzione del protocollo di CORBA, Internet Inter-ORB Protocol, che ha reso compatibile il mondo Java (RMI) con il mondo CORBA. RMI-IIOP (così è chiamata la versione di RMI che usa IIOP) è, poi, utilizzato in maniera stabile da Java Enterprise Edition, diventando, quindi, uno standard de-facto per quanto riguarda la interoperabilità tra applicazioni distribuite Java.

3.4.2 Distributed Garbage Collection

La gestione della memoria è uno degli aspetti sul quale i progettisti di linguaggi di programmazione hanno posto particolare attenzione per poter fornire le soluzioni migliori al programmatore. Infatti, essendo la memoria una risorsa finita, il linguaggio deve fornire degli strumenti che ne permettano e facilitino la gestione. Le filosofie principali di gestione della memoria oscillano tra due estremi ognuno con le proprie giuste motivazioni.

La prima supporta il punto di vista che "*La gestione della memoria è troppo importante per essere lasciata al sistema*" e quindi deve essere gestita interamente dal programmatore, che ha a disposizione strumenti per allocare e deallocare la memoria (come la `free()` e la `malloc()` in C). Ovviamente, in questo scenario, il programmatore è responsabile della deallocazione della memoria ed è quindi, più facile commettere errori; di contro, il programmatore ha il completo controllo della gestione della memoria, e può decidere se, in che momento, e quale parte di essa deve essere deallocata.

La seconda filosofia sostiene, invece, che "*La gestione della memoria è troppo importante per essere lasciata al programmatore*" e fornisce un servizio per la allocazione/deallocazione che è assolutamente trasparente al programmatore, la cosiddetta *garbage collection*. È questo il caso di Java, dove tutti gli oggetti vengono automaticamente allocati e deallocati quando il sistema lo ritiene necessario e opportuno, in quanto esso mantiene traccia dei riferimenti attivi ad ogni oggetto. Tra i vantaggi di questo tipo di soluzioni, sicuramente possiamo citare quello della produttività, in quanto la progettazione e l'implementazione

possono ignorare l'aspetto della gestione della memoria, e quello della facilità di uso del linguaggio, mentre, ovviamente, si perde il controllo della gestione della memoria e c'è un moderato degrado di prestazioni (ad esempio, ad intervalli fissati, nella macchina virtuale Java parte il thread della Garbage Collection e questo può influire su programmi che hanno obiettivi particolari e stretti vincoli di tempo). La gestione della memoria di Java è una delle caratteristiche del linguaggio più apprezzate: essa rappresenta un aspetto critico per assicurare la robustezza dei sistemi, specialmente per programmi che devono essere in esecuzione per lungo tempo. La Garbage Collection in locale funziona mantenendo e calcolando il numero di riferimenti che fanno riferimento ad un oggetto: se un oggetto non è più riferito da nessuno allora è candidato ad essere eliminato dall'heap alla prossima esecuzione del Garbage Collector.

La proposta di Java Remote Method Invocation, avendo tra gli obiettivi quello di mantenere una stretta integrazione con Java, fornisce un sistema di Garbage Collection per gli oggetti remoti. Questo meccanismo si basa su una estensione della garbage collection locale: la JVM tiene traccia di tutti i riferimenti all'oggetto remoto che risultano essere attivi (*live*). Alla prima invocazione di un oggetto, la JVM ritiene quell'oggetto referenziato e quindi da non eliminare. Al termine delle invocazioni, il client fa in modo di inviare un messaggio di dereferenziazione dell'oggetto e, la JVM server quindi considera quel riferimento *weak* (debole) e quindi passibile di eliminazione alla prossima invocazione del garbage collector.

Il meccanismo, fino a questo punto, sembra essere simile a quello locale. In effetti, ci sono dei notevoli problemi in più da trattare nel caso remoto: ad esempio, il client potrebbe chiudersi per qualche malfunzionamento, oppure potrebbe perdere la connessione verso il server, ed il server si troverebbe con un oggetto remoto che risulta essere referenziato (e quindi da non passare al garbage collector) ma in effetti non sarà mai utilizzato dal client, e quindi rappresenta un potenziale problema di *memory leak*. A questo scopo si introduce il meccanismo di *lease*: ogni riferimento che viene assegnato al client ha un tempo di vita specificato. Al termine del periodo, se non vengono effettuate altre invocazioni, il server considera quel riferimento non più valido e quindi l'oggetto diventa weak e collezionabile dal garbage collector. Questo significa che, in generale, il programmatore che scrive l'oggetto client deve prevedere che il lease possa scadere e, per evitare che l'oggetto server sia eliminato, fornire dei metodi che (a intervalli di tempo prefissati) provvedano a "rinnovare" il lease, effettuando delle chiamate fittizie a metodi che non hanno nessun effetto.

3.4.3 Caricamento dinamico delle classi

Java Remote Method Invocation permette il passaggio di oggetti come parametri, valori restituiti o eccezioni, attraverso la serializzazione. Questo permette di poter mantenere il sistema di tipi che il linguaggio offre, e, quindi, continuare ad offrire il familiare ambiente di programmazione. Questo, però, si scontra con una altra caratteristica di Java, quella del caricamento delle classi a tempo di esecuzione che risulta essere più complesso nel momento in cui stiamo passando ad un metodo offerto da un server remoto (ad esempio) un parametro che è una istanza di una sottoclasse della classe dichiarata nella firma del metodo. In questo caso, l'oggetto remoto può trovarsi nella situazione in cui non conosce esattamente come è strutturata la classe di cui l'oggetto passato è istanza.

Questo viene risolto da Java RMI attraverso il caricamento dinamico delle classi. In breve, quando si fa il marshalling degli oggetti per la trasmissione (ad esempio, come parametri nella invocazione da client a server), essi vengono anche annotati con il *codebase* cioé con la Uniform Resource Locator (URL) di un server WWW da dove è possibile trovare la definizione della classe (cioé il file `.class`). Quando viene effettuato l'unmarshalling

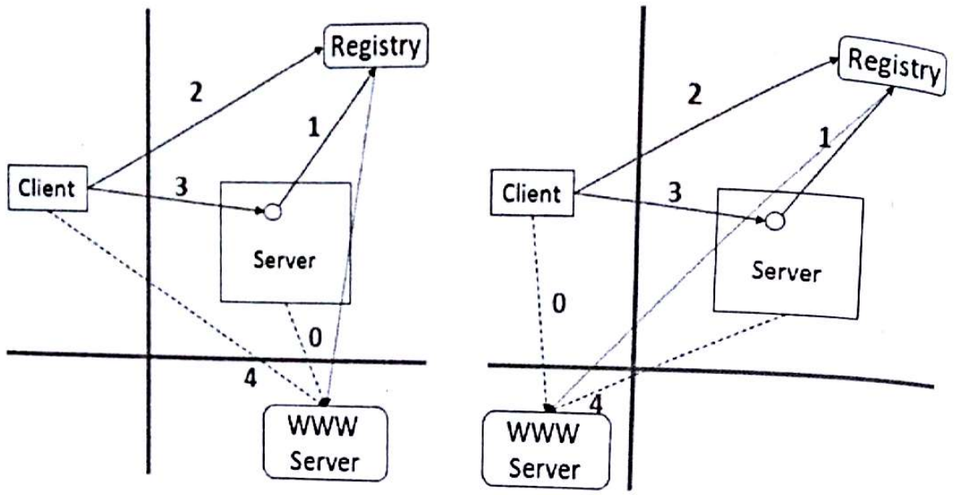

Figura 3.5: *Due esempi di utilizzo del caricamento dinamico. A sinistra, si illustra il caso in cui un valore (la cui classe non è conosciuta dal client) viene restituito al client: la definizione (il file* `.class`*) viene messa (0) dal server su un server WWW; il server poi, registra (1) l'oggetto sul registro (servizio di naming rmiregistry), da cui (2) il client ottiene il riferimento remoto che usa per fare la invocazione (3) per poi accedere (4) al server WWW per poter caricare la classe per fare l'unmarshalling del parametro che è stato restituito. Le linee doppie indicano il partizionamento degli host, da notare che il registry deve essere localizzato (per motivi di sicurezza) sulla stessa macchina del server. A destra, un simile diagramma spiega il comportamento nel caso di un parametro passato al server.*

dell'oggetto, il Classloader cerca di risolvere il nome della classe nel suo contesto, poi, in caso non sia possibile, viene acceduta la definizione della classe per poter ricreare l'oggetto all'altro capo della comunicazione (nel nostro esempio, sul server). Un diagramma illustrativo dei passi che vengono effettuati si trova nella Figura 3.5, dove non vengono spiegati tutti i dettagli, che invece verranno trattati successivamente nel paragrafo 8.2.

Questo meccanismo di caricamento dinamico permette anche il caricamento dinamico degli stub, con lo stesso meccanismo di annotazione che viene effettuato nel marshalling. Questo, ad esempio, avviene quando si passa un riferimento remoto, che consiste nell'inviare lo stub. Questo meccanismo viene spiegato in dettaglio nel paragrafo 3.5.4.

3.5 Approfondimenti

3.5.1 L'eterogeneità nelle tecnologie ad oggetti distribuiti

Il problema della eterogeneità è sempre stato considerato particolarmente importante per la progettazione dei sistemi distribuiti. Anzi, i sistemi distribuiti sono proprio caratterizzati dal fatto che, essendo composti da diversi nodi, possibilmente gestiti da organizzazioni diverse, presentano notevoli differenze per hardware e software (sistemi operativi, middleware, etc.). Vale la pena di sottolineare come la esigenza di considerare la eterogeneità è importante anche per possibili estensioni del sistema: anche se, al momento della progettazione e realizzazione il sistema può essere omogeneo, non è detto che questo continui ad

essere vero in futuro, e progettare un sistema facendo in modo di poter reggere la continua evoluzione della produzione informatica è una priorità importante.

È interessante approfondire i tipi di soluzioni forniti dalle tecnologie ad oggetti distribuiti negli ultimi 18 anni al problema della eterogeneità dei sistemi distribuiti. Come già detto, le principali proposte tecnologiche in questo campo sono: Common Object Request Broker Architecture (CORBA) del consorzio Object Management Group (OMG), .NET Remoting di Microsoft .NET e Java Remote Method Invocation di Sun.

Due sono le principali categorie che possiamo usare per le soluzioni basate su oggetti distribuiti che sono presenti in letteratura: le soluzioni *platform-neutral* (neutre rispetto alla piattaforma) e le soluzioni *platform-centric* (centrate attorno alla piattaforma) a seconda del fatto che siano basate o meno su una specifica piattaforma (hardware/software/linguaggio di programmazione).

CORBA rientra sicuramente nella categoria delle soluzioni neutre rispetto alla piattaforma in quanto, anzi, uno dei punti di forza dell'ambiente è la sua disponibilità in diversi linguaggi, compresi C, C++, Java ma anche ADA e COBOL. La possibilità di astrarre così tanti linguaggi, e così diversi tra di loro, è offerta dalla definizione della interfaccia in un linguaggio neutrale, chiamato Interface Definition Language, che definisce le funzionalità offerte da remoto, implementate nel linguaggio specifico. Diversi sono i problemi incontrati da questo tipo di soluzione. Innanzitutto, la espressività del passaggio di parametri viene limitata per la necessità di dover ricondurre ad una comune semantica diversi linguaggi, ciascuno con le sue caratteristiche. Poi, proprio per la differenza di implementazione, esiste una forte limitazione al tipo di oggetti che possono essere passati come parametri, in quanto solamente un insieme di oggetti predefiniti nell'ambiente possono essere passati per copia, perché la implementazione del metodo potrebbe essere realizzata con un altro linguaggio. Si rinuncia alla dinamicità della definizione di sottoclassi, con la possibilità di estendere il sistema dei tipi a tempo di esecuzione (cosa, invece, permessa in Java RMI tramite il caricamento dinamico delle classi). Insomma, l'intero sistema risulta essere alquanto statico e difficile da far evolvere, cosa che rappresenta il prezzo necessario per offrire una piattaforma che cerca di raccogliere a fattor comune diversi linguaggi, ciascuno con il proprio approccio e le proprie peculiarità. Infine, non va trascurato l'impatto sul programmatore che deve necessariamente apprendere un altro linguaggio (l'IDL).

Le soluzioni *platform-centric* (centrate sulla piattaforma) sono invece basate su una specifica piattaforma (hardware, software o linguaggio di programmazione) in modo da poter fare leva su una maggiore compatibilità "da progetto" dell'intero ambiente. Le due soluzioni che ricadono in questa categoria rappresentano un approccio diverso per fornire una piattaforma astratta unica: nel caso di Java Remote Method Invocation l'ambiente di riferimento è quello della macchina virtuale Java che offre un unico linguaggio per astrarre le diverse piattaforme (sistema operativo e hardware); nel caso della soluzione proposta da Microsoft, invece, il Common Language Runtime permette di astrarre le differenze tra linguaggi (C#, Visual C++, Visual Basic, etc.) su un unica piattaforma (il sistema operativo Microsoft). In un certo senso, la prima soluzione è *language-centric* mentre la seconda è *system-centric*.

Il vantaggio delle soluzioni centrate sulla piattaforma è sicuramente quello di adattarsi al linguaggio che viene utilizzato dal programmatore per implementare il proprio oggetto remoto. Questo significa, innanzitutto, ridurre e minimizzare l'impatto sulla produzione di applicazioni distribuite (perchè i programmatori non devono apprendere un nuovo strumento) ma significa anche che i programmatori possono fare leva sulle specifiche del linguaggio utilizzandole in pieno. In generale, un sistema centrato sulla piattaforma permette di passare oggetti per copia da un processo all'altro: saranno eseguiti dalla macchina virtuale Java oppure dal CLR in maniera trasparente. Questo significa poter far

migrare computazioni in maniera semplice ed efficace[7]. Inoltre, utilizzando un ambiente di programmazione tradizionale, si possono integrare le funzionalità distribuite come una naturale estensione del modello ad oggetti, cioé i meccanismi utilizzati dai programmatori in uno spazio di indirizzamento unico possono essere naturalmente estesi in uno scenario distribuito.

Come commento finale, vale la pena di sottolineare come non sia un caso che CORBA sia stato progettato da un consorzio di aziende (e quindi non potesse essere legato a nessun ambiente specifico) mentre Java e .NET (ed i loro modelli di oggetti distribuiti) sono stati invece progettati, ciascuno, da una singola azienda, il che ha permesso la progettazione unica dell'intero ambiente. Questo ha anche comportato alcune conseguenze nella usabilità dei vari ambienti: CORBA è noto essere un ambiente con API estremamente complesse e dalla curva di apprendimento molto lenta, mentre gli ambienti forniti dai produttori (Sun e Microsoft) facilitano al massimo il primo approccio ai propri ambienti, ma forniscono anche una alta produttività mediante ambienti integrati di sviluppo (Integrated Development Environment, IDE) come NetBeans e Visual Studio.

3.5.2 La trasparenza degli oggetti distribuiti

Avendo trattato con le soluzioni per risolvere la eterogeneità nei sistemi ad oggetti distribuiti, discutiamo ora come si può progettare un linguaggio di programmazione (ovviamente ad oggetti) che includa al proprio interno questo modello. Questa progettazione può avvenire in 2 maniere diverse:

1. gli oggetti (locali e remoti) vengono trattati come oggetti locali e uno strato di software si occupa di nascondere completamente al programmatore la conoscenza della natura degli oggetti;
2. gli oggetti locali sono trattati in maniera diversa dagli oggetti remoti, ed il programmatore conosce in ogni momento la natura di ogni oggetto e progetta ed implementa in maniera conseguente.

Queste sono le possibili scelte[8] che riguardano la *trasparenza* del modello ad oggetti distribuito, cioè quanto di esso debba essere visibile al programmatore.

La opzione 1 possiamo chiamarla di *totale trasparenza*: il programmatore ignora del tutto che esistono oggetti distribuiti e, ovviamente, non *sa* (cioè il linguaggio non gli permette di sapere) quale dei suoi oggetti è distribuito e quale è locale. Da un certo punto di vista, questa scelta sembra attraente: il linguaggio astrae anche la natura di essere distribuito di un oggetto e si occupa di tradurre accessi, invocazioni, ciclo di vita all'interno di una architettura distribuita, senza che il programmatore debba essere coinvolto.

La scelta 2 è invece la scelta di *non trasparenza* della natura dell'oggetto: il programmatore *deve* scegliere (vale a dire è forzato dal linguaggio) quali dei suoi oggetti sono locali e quali sono distribuiti, dovendo differenziare il meccanismo tramite il quale ne gestisce stato, comportamento, ciclo di vita e invocazione dei servizi. Questa soluzione appare sicuramente più onerosa per il programmatore: non solo deve gestire gli oggetti distribuiti in maniera diversa (e più complessa) rispetto a quelli locali, ma deve anche farsi

[7] Ricordiamo che gli oggetti distribuiti rivestono particolare importanza proprio nei sistemi enterprise per poter bilanciare il carico e offrire tolleranza ai malfunzionamenti su più server replicati.

[8] Ovviamente, abbiamo scartato la soluzione (estremista) in cui gli oggetti (locali e remoti) vengono considerati tutti come oggetti distribuiti e il programmatore tratta ogni oggetto come se si trovasse in un altro spazio di indirizzamento. Infatti, in questo caso, qualsiasi programma, anche il più semplice (pensate ad un `HelloWorld` dove la stringa `HelloWorld` deve essere allocata in remoto!) risulterà essere tanto complesso da rendere il linguaggio non utilizzabile e difficilmente apprendibile.

carico di decidere *quali* oggetti saranno locali e quali no. A prima vista, questa soluzione sicuramente sembra meno agevole ma, vedremo, risulterà maggiormente efficace per la realizzazione di sistemi distribuiti affidabili.

Il dibattito sulla trasparenza degli oggetti distribuiti rappresenta un fenomeno ciclico: spesso, nella storia dei linguaggi di programmazione, si è imputato alle insufficienze del modello di programmazione distribuito il numero relativamente piccolo di applicazioni distribuite realizzate. Quindi, la discussione su quanto e come debbano essere trasparenti le "parti" che compongono un sistema distribuito è una discussione che continua da un lungo periodo di tempo, con diverse tecnologia, una sorta di "Dèjà vu" tecnologico.

In effetti, la programmazione in un ambito distribuito risulta essere profondamente diversa dalla programmazione di oggetti nello stesso spazio di indirizzamento locale, questo per una serie di motivi che ora approfondiremo. La conseguenza di questa differenza strutturale è che la scelta migliore per poter costruire sistemi distribuiti affidabili e robusti è quella di obbligare il progettista ed il programmatore a conoscere i dettagli circa la natura dell'oggetto, se esso sia, cioé, distribuito o locale. Un modello di totale trasparenza, che permette a progettista/programmatore di non conoscere la natura degli oggetti, è destinato infatti ad ignorare degli aspetti fondamentali del sistema che viene costruito e quindi a realizzare sistemi inaffidabili e non sicuri.

Iniziamo con il definire esattamente l'aspetto che differenzia sostanzialmente la computazione locale da quella distribuita:

- per *computazione locale* intendiamo il calcolo che è confinato all'interno di un singolo spazio di indirizzamento;
- per *computazione distribuita* intendiamo il calcolo che viene effettuato coinvolgendo ed utilizzando diversi spazi di indirizzamento, possibilmente anche su macchine diverse.

Basandosi su questa distinzione sostanziale, analizziamo le differenze fondamentali che esistono tra computazione distribuita e computazione locale in modo da evidenziare i motivi per cui la non trasparenza degli oggetti distribuiti è il modello migliore.

La comunicazione: latenza e banda

La differenza più evidente di un oggetto distribuito da un oggetto locale è sicuramente la latenza. Il tempo che viene richiesto per invocare un metodo remoto è sostanzialmente diverso (4-5 ordini di grandezza) da quello che è necessario per fare invocazioni locali. E questa differenza è sostanzialmente non eliminabile, in futuro, visto che alla velocità della luce, sono necessari almeno 30ms per effettuare un ping dall'Europa negli Stati Uniti (andata e ritorno) anche se tutta la elaborazione (router, TCP/IP, etc.) fosse fatta in tempo 0.

Anche su Local Area Network, la latenza è sostanzialmente molto maggiore delle invocazioni locali e questo non può essere ignorato, in quanto le invocazioni remote richiedono più tempo e diminuiscono la efficienza della soluzione, persino in un ambiente controllato come una LAN.

Anche in futuro, probabilmente, la situazione non cambierà: ai miglioramenti alla latenza che si otterranno sulle Local Area Network e sulle Wide Area Network, corrisponderanno altrettanto significativi miglioramenti nella efficienza e tempo di risposta alle invocazioni da parte dei processori e delle memorie.

Una altra differenza immediatamente visibile è la banda di comunicazione. In effetti, la banda a disposizione sulle reti è aumentata in maniera consistente[9] rispetto ai miglioramenti che si sono ottenuti sulla latenza. Ciò nonostante, esiste una differenza notevole tra la banda che viene messa a disposizione delle architetture locali tra CPU, cache (vari livelli), memoria e memoria di massa. Quindi, questo aspetto non può essere ignorato nella progettazione: la scelta, ad esempio, se una invocazione di un metodo deve trasferire oppure no degli oggetti (serializzati) di notevole dimensione si deve basare sulla conoscenza, in fase di progettazione, della natura locale o remota dell'oggetto su cui si invoca il metodo. Inoltre, se anche la banda aumenta sulle reti geografiche e locali, aumenta anche la quantità di dati che si devono poter scambiare le applicazioni distribuite (dati multimediali, quali audio, video, etc.).

La non trasparenza degli oggetti distribuiti obbliga progettisti e implementatori a considerare dall'inizio gli oggetti distribuiti come "diversi" da quelli locali. In effetti, ignorare la latenza e la banda significa incorrere sicuramente in problemi di prestazioni per la applicazione realizzata. Eppure, sebbene queste caratteristiche siano le più evidenti e le maggiormente citate, latenza e banda non sono caratteristiche più difficili da riconciliare tra oggetti distribuiti e oggetti locali. Altre insuperabili differenze tra le due visioni esistono e rendono irrealizzabile l'obiettivo di una visione "unificante" offerto dalla trasparenza del modello.

Accesso alla memoria

Dal punto di vista del programmatore, questa differenza è fondamentale: i puntatori alla memoria locale non hanno senso in un contesto distribuito. Se il sistema dovesse completamente nascondere questi dettagli al programmatore, dovrebbe "forzare" un approccio interamente basato su oggetti (ad esempio, eliminando i tipi di dato primitivi, che sono presenti in molti linguaggi, come Java, per motivi di efficienza) ed, allo stesso tempo, impedire al programmatore di utilizzare puntatori che siano relativi ad una memoria locale. E questo rende necessario per il programmatore imparare ad usare un nuovo linguaggio, solo per poter mantenere la trasparenza locale/remoto, il che, di per sè, rappresenta un serio ostacolo alla diffusione del calcolo distribuito, oltre che (ovviamente) rappresentare una maniera paradossalmente non-trasparente per forzare la trasparenza: si deve cambiare il linguaggio utilizzato. E lo stesso paradosso, rendere non-trasparente la trasparenza, si otterrebbe se si volesse comunque fornire un sostituto dei puntatori da utilizzare in spazi di indirizzamento diversi: il programmatore sarebbe forzato (non trasparentemente) ad usare questo nuovo tipo di "puntatore" per poter assicurare la trasparenza.

Insomma, l'accesso alla memoria è profondamente diverso tra modello locale e distribuito, e cercare di nasconderlo crea più problemi e contraddizioni della soluzione di non-trasparenza che riconosce e rende evidente le differenze tra i due modelli.

Malfunzionamenti parziali del sistema

Nei sistemi distribuiti, una parte del sistema può essere soggetta a malfunzionamenti in maniera totalmente impredicibile mentre il resto deve continuare a funzionare e fornire i servizi per cui il sistema è stato progettato. Questa caratteristica è talmente radicata nei sistemi distribuiti che il malfunzionamento parziale (insieme alla concorrenza, di cui parliamo dopo) viene considerato da alcuni autori come i fattori che danno la "definizione" di un sistema distribuito.

[9] In circa 10 anni, la banda è migliorata di circa 1000 volte mentre la latenza è migliorata solo di un fattore 10. Questo suggerisce, in generale, che per migliorare le prestazioni si deve strutturare la applicazione in modo che le invocazioni remote siano meno frequenti ma che trasportino più dati.

La differenza tra il caso locale e quello remoto è davvero di natura fondamentale. In locale, un malfunzionamento è potenzialmente in grado di bloccare completamente la macchina e le conseguenze sono che o ci riesce (errori hardware non recuperabili) oppure viene gestito da un coordinatore centrale (il sistema operativo, ad esempio) che ha sempre il completo controllo di tutto il sistema. Questo coordinatore centrale non esiste nei sistemi distribuiti, dove "*il fallimento di un computer di cui ignoravi persino la esistenza, può rendere inutilizzabile il* tuo *computer*", come recita una famosa definizione (provocatoria) di Leslie Lamport del 1987. Tutto può accadere in un sistema distribuito geograficamente, dai problemi sulla rete, a mancanza di corrente, fino ad arrivare a qualcuno che inciampa nel cavo di alimentazione oppure in quello di rete e disconnette una macchina che stava fornendo un servizio remoto ad un client, ad esempio. Ed i problemi si influenzano a cascata e magari il problema di un nodo A rende il nodo B non funzionante (magari bloccato aspettando risposte) che rende il nodo C non funzionante... e così via, fino alla nostra macchina (Z) che rimane bloccata e non se ne capisce la ragione.

Il problema centrale del calcolo distribuito (che non esiste nel calcolo locale) è quello di assicurarsi, dopo un malfunzionamento, che lo stato dell'intero sistema sia coerente. Immaginiamo una invocazione remota da un oggetto A verso un oggetto B. Se la chiamata fallisce, il nodo B non ha maniera di sapere cosa è successo e se, ad esempio, l'errore si è verificato nella invocazione, nel passaggio di parametri, sull'hardware del nodo B, oppure se la chiamata è stata effettuata correttamente ma solamente la trasmissione del valore restituito è fallita a causa di un router che si è spento, da qualche parte nel mondo. In quel caso, la nostra applicazione deve essere progettata per evitare, ad esempio, che la re-invocazione dello stesso metodo non alteri lo stato del sistema, realizzando dei metodi idempotenti che permettano la invocazione dello stesso metodo con un certo insieme di parametri in modo che esattamente (o al più) una di queste invocazioni sia eseguita. Pensiamo, ad esempio, ad un meccanismo di *timestamp* calcolati dal client che permetta al server di riconoscere una invocazione "ripetuta" di un metodo in modo da evitare (ad esempio) di effettuare due bonifici bancari invece di uno se la prima invocazione è ritornata al client con un errore, il server la aveva in effetti eseguita, ed il client la ha reinviata. Ma se il metodo deve passare anche un timestamp, allora la sua interfaccia deve essere modificata, e quindi il metodo deve essere esplicitamente dichiarato come remoto.

Rendere un oggetto distribuito robusto rispetto ai malfunzionamenti parziali che possono capitare richiede di far gestire situazioni non ben determinate e di adottare strategie che evitino di alterare e rendere incoerente lo stato del sistema. Invece, in un sistema locale, se una invocazione locale fallisce, allora il sistema è in uno stato tale che probabilmente il malfunzionamento è globale e non c'è maniera di recuperare uno stato coerente.

E questa differenza non è sormontabile: un oggetto remoto può fallire in maniera diversa da un oggetto locale, ed il programmatore deve essere cosciente di questo per poter adottare contromisure adeguate.

Concorrenza

Infine, una differenza fondamentale sta nella concorrenza. Un oggetto remoto è soggetto ad invocazioni che sono generate in maniera "genuinamente" concorrente da nodi che esattamente nello stesso istante fanno arrivare diverse invocazioni allo stesso oggetto. Inoltre, ritardi nella comunicazione possono fare arrivare diverse invocazioni successive dello stesso client verso un server in ordine diverso, portando quindi ad una loro esecuzione in ordine diverso rispetto a quello in cui il client le ha effettuate.

La situazione, nel costesto locale, è profondamente diversa: esiste un gestore centralizzato di risorse (il sistema operativo) che può gestire ed orchestrare le comunicazioni tra oggetti e supportare nella sincronizzazione e nelle tecniche di recovery. Un oggetto

remoto non può contare su questo supporto e quindi le tecniche di gestione della concorrenza risultano essere più complesse, cosa di cui, quindi, il programmatore deve essere a conoscenza. Quindi, in conclusione, anche questa motivazione spinge ad adottare la non-trasparenza del modello degli oggetti distribuiti, se si vuole che il sistema sia robusto e affidabile.

3.5.3 La sicurezza in Java

La *sandbox* fornita dal linguaggio permette di eseguire applicazioni (e applet) in maniera tale che le operazioni che esse compiono siano controllate e ristrette così da prevenire danni dovuti ad errori di programmazione oppure da intenzionali tentativi di operazioni illegali. Java fornisce la sandbox basandosi su 4 livelli di sicurezza: la sicurezza intrinseca del linguaggio, il Classloader, il Bytecode Verifier ed il Security Manager (mostrati in Fig. 3.6).

Il linguaggio *Java* è considerato *inerentemente sicuro* per vari motivi. Innanzitutto, il linguaggio è fortemente tipizzato (*strongly typed*): tutte le variabili hanno un tipo definito a tempo di compilazione e solamente (poche) implicite conversioni (*casting*) vengono effettuate dal compilatore e dalla macchina virtuale a run-time. Tutte le altre operazioni di casting devono essere esplicitate dal programmatore. Questo evita errori possibili durante la esecuzione. Poi, Java offre la gestione automatica della memoria attraverso un meccanismo di garbage collection, che impedisce di esaurire lo spazio di indirizzamento del processo. La assenza di puntatori, e la impossibilità di poter fare aritmetica o assegnamenti con i riferimenti rende impossibile effettuare accessi illegali in memoria. Inoltre, l'accesso alla memoria reale non viene determinato a tempo di compilazione (quando si usano degli *handle* simbolici) ma a tempo di esecuzione, quindi non si può conoscere in anticipo in che zona di memoria verranno memorizzati gli oggetti e, quindi, non si può scrivere codice per alterarli. Infine, a tempo di esecuzione vengono controllati i limiti di array per prevenire accessi a elementi non esistenti.

Il *Classloader* si occupa di caricare la classe a tempo di esecuzione, anche da locazioni remote. Il suo compito principale è quello di caricare la classe in un namespace separato rispetto a quello delle classi locali, in modo che classi del linguaggio built-in, locali, non possono essere rimpiazzate da altre (ad esempio, come il Security Manager). Infatti, quando si fa riferimento ad una classe, viene *prima* cercata tra le classi del sistema locale (built-in) e, solo successivamente, nel namespace della classe dalla quale viene riferita. In questa maniera non c'è possibilità di "sovrascrivere" una classe di sistema. Insomma, il Class loader controlla che non possiamo scrivere una classe `String` che, ogni volta che viene usata, manda il contenuto della stringa via mail a qualcuno, in modo da rivelare numeri di carta di credito ed altre informazioni: la classe utilizzata sarà sempre quella presente nel package `java.lang.*`.

Infine, il Classloader fa in modo che ogni classe abbia il proprio namepace separato dalle altre classi caricate in modo da prevenire possibili interferenze.

Dopo che il Classloader ha caricato una classe per la esecuzione, il *Bytecode verifier* controlla che essa non sia "volontariamente" ostile. Controlla, quindi che essa sia conforme alle specifiche del linguaggio (ad esempio, correttezza degli operandi per ogni operazione, casting illegali, etc.), che non ci siano stack underflow/overflow, e che non ci siano violazioni alla regole specificate dai modificatori di accesso (ad esempio, che non si acceda illegalmente ad un campo il cui accesso è dichiarato `private`). Questo livello di controllo si rende necessario per il fatto che il bytecode viene generato in maniera nettamente distinta (sia come spazio che come tempo) dal suo caricamento ed esecuzione e permette

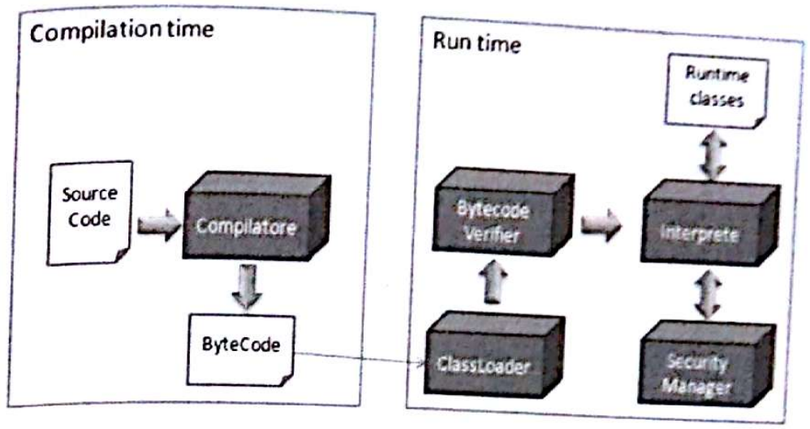

Figura 3.6: *I 4 livelli di sicurezza di Java, dalla definizione del linguaggio (vincolo forzato dal compilatore e dall'interprete, in due fasi diverse) ai successivi Classloader, Bytecode Verifier e Security Manager.*

di garantire che il bytecode non sia stato artatamente manomesso oppure che sia stato generato da un compilatore Java di bassa qualità che ha omesso alcuni controlli.

A questo punto il *Security Manager* si occupa di definire i confini della sandbox. In effetti, esistono operazioni la cui pericolosità dipende dalla politica che l'utente ha scelto. Ad esempio, leggere dal disco non è di per sè una operazione insicura ma dipende se a farlo è una applicazione scaricata da remoto che legge dati privati e sensibili per trasmetterli silenziosamente ad un server oppure una nostra applicazione che sta leggendo un file di configurazione. Il Security Manager viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa, e fornisce le autorizzazioni sulla base della politica (*policy*) che ha stabilito l'utente lanciando la macchina virtuale. In questo modo, l'utente può eseguire una applicazione sconosciuta con una policy restrittiva, mentre una applicazione fidata o ben nota può essere lanciata con una policy più aperta. Il Security Manager viene caricato a tempo di esecuzione e non può essere esteso, sovrascritto o rimpiazzato. La politica di accesso può essere specificata per dominio di provenienza, in modo da poter differenziare agevolmente il comportamento del Security Manager (maggiori dettagli sono forniti nel paragrafo 4.4.

3.5.4 Il meccanismo di marshalling usato da Java RMI

Fare il marshalling di un oggetto in Java significa effettuare una serializzazione modificando la semantica dei riferimenti remoti (invece di un riferimento remoto viene inserito lo stub dell'oggetto remoto) e aggiungendo informazioni all'oggetto (il codebase della classe dell'oggetto).

Il meccanismo di marshalling di Java RMI si basa sulla specializzazione del meccanismo tradizionale di serializzazione effettuata da `ObjectOutputStream`. Infatti, questa classe offre la possibilità di poter modificare il comportamento tramite il quale gli oggetti vengono scritti come stream di byte. Più precisamente, la specializzazione del meccanismo di serializzazione per il marshalling avviene modificando tre metodi della classe `ObjectOutputStream`:

- Il metodo `replaceObject()` che può definire un metodo alternativo per serializzare un oggetto sullo stream.

- Il metodo `enableReplaceObject()` che restituisce un booleano e stabilisce se la istanza deve oppure no specializzare il meccanismo di serializzazione, usando il metodo `replaceObject()`.

- Il metodo `annotateClass()`, che permette di inserire informazioni addizionali sulla classe, viene usato per specificare il codebase e permettere quindi il caricamento dinamico.

La operazione più complessa è quella di `replaceObject()`. Il meccanismo di serializzazione di RMI specifica (abilitando il flag booleano restituito da `enableReplaceObject()`) che questo metodo deve essere richiamato ogni qualvolta viene invocato `writeObject()`, *prima* che questo provveda alla serializzazione dando la possibilità di sostituire l'oggetto da serializzare. Il metodo `replaceObject()` fa le seguenti operazioni:

1. Se l'oggetto da serializzare è una istanza di `java.rmi.Remote` e quindi è un riferimento remoto, e l'oggetto risulta esportato al runtime di RMI, allora viene restituito lo stub per quell'oggetto remoto utilizzando il metodo `java.rmi.server.RemoteObject.toStub()`. Nel caso in cui l'oggetto remoto non sia esportato, allora si restituisce l'oggetto remoto stesso.

2. Se l'oggetto da serializzare non è una istanza di `java.rmi.Remote` allora viene restituito a `writeObject()`.

Tramite questo meccanismo viene risolta una apparente incongruità nelle invocazioni: quando viene passato un riferimento remoto come parametro, questo viene copiato sullo stream, ma non abbiamo il vincolo di dichiarare un riferimento remoto come serializzabile. Infatti, quando si passa un riferimento remoto che è esportato (quindi attivo), esso viene sostituito dal suo stub che è una istanza di `java.rmi.server.RemoteStub`, che è una classe che implementa la interfaccia `Serializable`.

Questo meccanismo spiega anche la modalità con la quale viene assicurata la integrità referenziale: poiché i parametri di una stessa invocazione remota utilizzano lo stesso stream di output, parametri che si riferiscono allo stesso oggetto *nella stessa invocazione* verranno serializzati nel flusso come facenti riferimento allo stesso oggetto, e verranno deserializzati nella stessa maniera all'altro capo dello stream.

Note bibliografiche

Architettura e Modello di Java Remote Method Invocation

La presentazione della architettura di Java RMI è approfondita nell'articolo che ha presentato il modello, del 1996, [39] oltre che nella documentazione Sun [2]. Alcuni commenti sulla progettazione di Java RMI si trovano in interviste sulla rete con Jim Waldo [37] e con Ann Wollrath [1]. Un confronto con Corba e DCOM fatto da Jim Waldo si trova in [38].

Eterogeneità e Trasparenza

La trasparenza degli oggetti distribuiti viene discussa ampiamente, con alcuni esempi pratici in [27] e alcuni commenti più recenti di Henning sono presenti in [19].

Una raccolta di errori comuni e ricorrenti nella progettazione e implementazione dei sistemi distribuiti si trova a [34]. La famosa mail di Leslie Lamport che ha originato la provocatoria definizione di Sistema Distribuito si trova a [29].

Sicurezza in Java

La sicurezza di Java puè essere approfondita con la documentazione Sun (in locale o sul sito) [4, 10] mentre il Java Language Environment viene descritto in [14].

Marshalling e Serializzazione

La serializzazione è affrontata in maniera completa in [18] ed, in particolare, nel Capitolo 10 (disponibile anche online [16]). La differenza tra serializzazione e Marshalling è ampiamente descritta nella RFC 2173 [35] ed un primo articolo dove la tecnica viene chiamata con il suo nome originario (pickling) si trova in [33].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* [P], Esercizi, *indicati con una* [E], *oppure* Domande di ricapitolazione, *segnalate da una* [R]. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

Gli obiettivi della progettazione di Java RMI

1. [R**] Quali sono i vantaggi dall'integrare il modello distribuito all'interno di un linguaggio di programmazione?

2. [R**] Quale è la differenza tra invocazione *unicast* e invocazione *multicast* nel contesto distribuito? E per cosa quella multicast può essere utilizzata?

La architettura di Java RMI

3. [R**] A cosa serve una interfaccia remota?

4. [R***] Per quale motivo la non trasparenza degli oggetti distribuita ha come (non unica) conseguenza la esistenza di una eccezione remota che **deve** essere dichiarata da metodi remoti?

5. [R**] Come funziona il servizio di localizzazione offerto da Java RMI?

6. [R***] Cosa è la integrità referenziale?

7. [P***] Perché non si possono passare oggetti locali per riferimento come parametri, in Java RMI?

8. [R***] Quali sono le differenze tra modello ad oggetti locale e quello distribuito, in Java RMI?

9. [P**] Perché si deve ridefinire il metodo `equals` per un oggetto remoto?

10. [P***] Perché si deve ridefinire il metodo `hashCode` per un oggetto remoto?

11. [R***] Cosa sono le *checked exceptions* e perché le eccezioni remote sono tali?

12. [R***] Definire i compiti e le interazioni tra i layer della architettura di Java RMI.

13. [R***] Descrivere la politica di gestione della memoria distribuita usata da Java RMI.

14. [P**] Perché si rende necessario aggiungere il meccanismo del *lease* alla Garbage Collection distribuita?

15. [P***] Perché la gestione della memoria a cura del programmatore (come in C, ad esempio) rende le applicazioni più efficienti?

La eterogeneità nelle tecnologie ad oggetti distribuiti

16. [P**] Perché è stato scelto che un parametro locale ad un metodo remoto venga passato per copia? Quali sarebbero stati i problemi se si fosse scelto di mantenere la semantica delle invocazioni locali Java?

17. [P**] Perché i metodi remoti devono (dovrebbero) essere idempotenti mentre quelli locali no? E perché questo ha a che fare con le transazioni distribuite?

18. [P***] Perché la possibilità di passare oggetti per copia dipende dalla maniera in cui la piattaforma tratta il problema della eterogeneità?

La trasparenza degli oggetti distribuiti

19. [R***] Descrivere le motivazioni fondamentali che rendono necessario distinguere, in un linguaggio, un oggetto locale da uno distribuito.

20. [R***] Descrivere perché in un sistema locale non ha senso di parlare di malfunzionamenti parziali.

La sicurezza in Java

21. [P**] Perché è necessario che il Bytecode verifier "ripeta" i controlli che il compilatore ha già effettuato?

22. [P*] Supponiamo che una classe `String` caricata da remoto possa rimpiazzare la classe `String` del linguaggio Java. In che maniera questo potrebbe inficiare la sicurezza di una applicazione?

Capitolo 4

Un primo esempio con Java RMI

Indice

4.1 Introduzione 82
4.2 Il processo di creazione di un programma Java RMI 82
4.2.1 Definizione della interfaccia remota 82
4.2.2 Implementazione del server 83
4.2.3 Compilazione del server 83
4.2.4 Compilazione con lo stub compiler `rmic` 84
4.2.5 Servizio di naming: `rmiregistry` 85
4.2.6 Esecuzione del server 85
4.2.7 Registrazione del server sul servizio di naming 88
4.2.8 Implementazione del client 88
4.2.9 Compilazione ed esecuzione del client 89
4.3 L'esempio HelloWorld 89
4.3.1 Definizione della interfaccia remota 89
4.3.2 Implementazione del server 90
4.3.3 Compilazione del server 91
4.3.4 Compilazione con lo stub compiler `rmic` 91
4.3.5 Servizio di naming: `rmiregistry` 91
4.3.6 Esecuzione del server 92
4.3.7 Registrazione del server sul servizio di naming 92
4.3.8 Implementazione del client 92
4.3.9 Compilazione ed esecuzione del client 93
4.4 La sicurezza e la policy del Security Manager 93
4.5 Commenti conclusivi 94
Note bibliografiche 95
Spunti per lo studio individuale 96

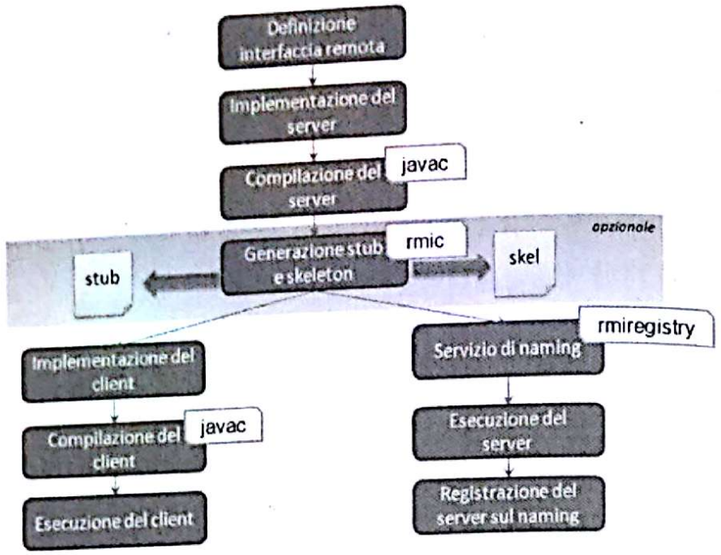

Figura 4.1: *Il processo di creazione di un programma che usi oggetti remoti con Java RMI.*

4.1 Introduzione

In questo capitolo cercheremo di presentare un primo esempio di un programma, illustrando, però, dapprima il processo di creazione che porta alla sua realizzazione, analizzando i vari passi.

La descrizione di questo processo risulterà essere dettagliata e pedante per motivi didattici: probabilmente, una volta appreso il meccanismo non ci sarà necessità di seguire questo processo passo-passo.

4.2 Il processo di creazione di un programma Java RMI

Il processo per la creazione di un programma Java RMI può essere riassunto dalla Figura 4.1 e si suddivide, dopo alcuni passi preliminari, in due sottoprocessi: uno che procede verso lo sviluppo e esecuzione del server, l'altro che va in direzione dello sviluppo e della esecuzione del client. Alcuni passi sono dipendenti dai precedenti e altri, invece, possono proseguire indipendentemente. In particolar modo, lo sviluppo del client risulta essere solo minimamente influenzato da quello del server: come vedremo, il programmatore lato client ha necessità esclusivamente della interfaccia remota (ed eventualmente dello stub) per potere proseguire nello sviluppo del programma client.

4.2.1 Definizione della interfaccia remota

Il primo passo consiste nella definizione di quali sono i servizi che vengono offerti dal nostro server, specificati all'interno di una interfaccia. Questa interfaccia rappresenta il "contratto" che vincola il server ad offrire determinati servizi se il client utilizza la interfaccia che il server espone. La implementazione, ovviamente, sarà a carico del server e totalmente nascosta al client.

Con Java Remote Method Invocation non è necessario utilizzare un linguaggio ad-hoc per specificare la interfaccia: basta utilizzare il meccanismo della `interface` Java. Questo a differenza di quanto accade in Corba, dove, data la natura multilinguaggio di tale piattaforma di middleware, si rende necessario utilizzare un linguaggio neutrale, chiamato Interface Definition Language (IDL) .

La interfaccia remota per RMI deve avere le seguenti caratteristiche:

- la `interface` deve derivare dalla interface mark-up `Remote`;
- tutti i metodi remoti devono lanciare la eccezione `java.rmi.RemoteException`.

Un classico errore che viene fatto è quello di inserire nella interfaccia remota "a tappeto" tutti i metodi del server, anche quelli che, in effetti, sono locali. Questo è un errore per (almeno) un paio di motivi: innanzitutto, è concettualmente errato indicare come remoti dei metodi che non verranno mai chiamati da remoto e può confondere per la incoerenza qualcuno che legga il codice; poi, in questa maniera viene permesso che un client (scritto ad hoc) possa chiamare da remoto quei metodi (che originalmente dovevano essere chiamati solo in locale), generando dei problemi di sicurezza che nelle nostre applicazioni "giocattolo" di solito non consideriamo, ma che sono ben presenti nella realtà.

4.2.2 Implementazione del server

A questo punto, il server viene implementato. Un oggetto remoto in Java deve essere istanza di una classe che:

- implementi una o più interfacce remote (cioè che estendano `Remote`);
- derivi da `java.rmi.UnicastRemoteObject`.

Il fatto che il server derivi da `UnicastRemoteObject` implica che il costruttore del nostro server debba esplicitamente essere scritto (anche se vuoto!) in quanto è necessario che il costruttore della sottoclasse (il nostro server) lanci esplicitamente la eccezione `RemoteException` che viene lanciata dal costruttore della superclasse. Del perché il costruttore di `UnicastRemoteObject` debba poter lanciare questa eccezione parleremo successivamente.

4.2.3 Compilazione del server

Questa è la parte più semplice: in generale una buona IDE, come Eclipse, lo fa per noi, automaticamente.

Attenzione! Diventa importante, da questo punto in poi, sapere esattamente dove vengono messi i file `.class` che vengono creati dal compilatore `javac` a seconda di come avete configurato il vostro progetto Java in Eclipse quando lo avete creato. Infatti, se avete accettato il valore di default forniti dal wizard di creazione di un progetto Java, allora i sorgenti vengono messi in una directory `src` mentre i file in bytecode vengono messi in una directory `bin` che (dal Package Explorer di Eclipse) non viene visualizzata. Per poterla visualizzare (vedremo perché è importante nel prossimo passo) è necessario utilizzare una view Navigator (disponibile sotto la perspective `Resource`).

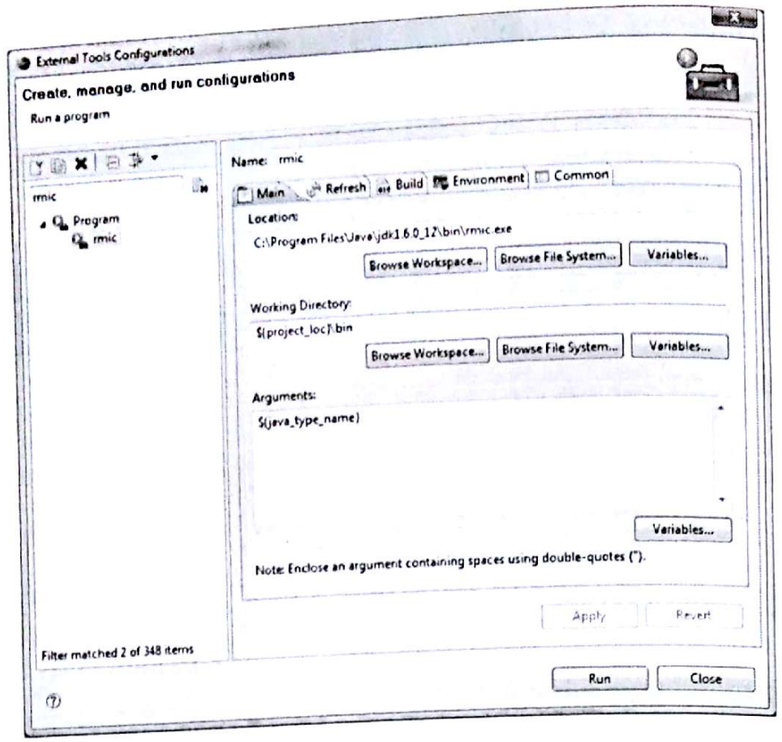

Figura 4.2: *La configurazione di esecuzione di* `rmic` *come applicazione esterna in Eclipse: si deve selezionare il pathname del comando e indicare come directory corrente quella dell'elemento selezionato, usando* `${workspace_loc}/${container_path}/`*, se si trova nel workspace oppure* `${project_loc}/bin` *se il progetto si trova altrove (ma sotto la directory* `bin`*).*

4.2.4 Compilazione con lo stub compiler `rmic`

A questo punto, entra in ballo il supporto che RMI fornisce al programmatore. Avendo specificato interfaccia remota e server (che deriva da `UnicastRemoteObject`) si possono generare automaticamente i file che sono necessari (stub e skeleton) senza che debbano essere scritti dal programmatore.

A questo scopo, RMI fornisce uno *stub compiler*, chiamato `rmic` che, eseguito su il file .class del nostro server, genera lo stub (e lo skeleton). Qui è necessario una veloce anticipazione: RMI è evoluto con le versioni di Java ed alle iniziali versioni che prevedevano la generazione di stub e skeleton (JDK 1.1 e 1.2) sono seguite versioni in cui viene generato solamente lo stub che può fare a meno dello skeleton, in maniera più evoluta, tramite il package `java.lang.reflect` (maggiori dettagli sono forniti nel paragrafo 8.3. L'esempio che faremo adesso usa il modello pre-JDK 1.5 che prevede la creazione solamente dello stub.

Quindi, dobbiamo eseguire lo stub sul file .class del nostro server. A tale scopo, seguendo le istruzioni in Figure 4.2, possiamo definire una applicazione esterna da chiamare dall'interno di Eclipse in modo da poterlo eseguire selezionando (nel Navigator) il file .class e generare lo stub del nostro oggetto server.

Attenzione... normalmente, non viene effettuato il refresh della directory nella quale si trova la classe di cui avete generato lo stub. Quindi sembra che `rmic` non abbia funzionato. Basta usare il tasto F5, oppure, nella configurazione della esecuzione di applicazioni esterne (presentata nella Figura 4.2), selezionare, nella scheda "Refresh", la checkbox che permette di fare il refresh dell'intero progetto (o della directory corrente) dopo la esecuzione della applicazione.

4.2.5 Servizio di naming: `rmiregistry`

A questo punto è necessario che il nostro oggetto remoto possa essere accessibile dai client che ne vogliono invocare i servizi. A tale scopo, Java RMI propone un servizio di naming (abbastanza semplice) chiamato `rmiregistry`. Questo programma deve essere lanciato prima di eseguire l'oggetto server, in quanto il server (tipicamente) tra le prime operazioni farà in modo di registrarsi presso il servizio di naming con una etichetta. I client che vogliono accedere ai servizi di un oggetto remoto, fanno una richiesta (operazione di *lookup*) al registry per ottenere un riferimento remoto da usare per le invocazioni remote.

Il servizio di naming *deve* essere lanciato dalla directory dove si trova il file .class dello stub, dell'oggetto remoto e della interface remota, quindi (nei nostri esempi) dalla directory `bin` del progetto. In Figura 4.3 si trova un esempio in cui si evidenzia il valore del campo `working directory` nella configurazione di esecuzione. Attenzione! Per eseguire rmiregistry con questa configurazione, deve essere selezionata (nel progetto) la directory dove si trova lo stub, e poi eseguire il comando esterno definito come in Figura 4.3.

4.2.6 Esecuzione del server

Ora si può eseguire il server, tradizionalmente, eseguendo con la macchina virtuale la classe appropriata. L'unica cosa particolare da fare è scegliere una politica di sicurezza per la macchina virtuale, operazione che non compiamo per applicazioni locali, di solito. Infatti, la macchina virtuale, per poter eseguire operazioni potenzialmente pericolose (ed accedere la rete lo è) ha bisogno che venga esplicitamente permesso in una *policy* che viene fornita alla macchina virtuale su linea di comando.

La scelta di una policy è abbastanza complessa (si può arrivare a scegliere le porte sulle quali ci si può connettere, oppure se è possibile scrivere/leggere/modificare un file, etc.) ma nel nostro caso (assolutamente di esempio!) possiamo evitare i problemi di configurazione adottando una politica estremamente liberale (che chiameremo, appropriatamente, `policyall`) nella quale "tutto è permesso!":

```
grant {
        permission java.security.AllPermission;
};
```

Dobbiamo quindi fornire su linea di comando alla macchina virtuale la indicazione del file di policy da seguire, tramite il comando `java -Djava.security.policy=policyall xxx` oppure, dall'interno di Eclipse, creando una configurazione di run come indicato in Figura 4.4, dove viene indicato tra i parametri della macchina virtuale. Particolare attenzione va data al percorso del file `policyall`: se la macchina virtuale non trova il file di policy non protesta in maniera evidente, ma semplicemente adottando una policy estremamente restrittiva, quindi impedendoci accesso (dal server) al servizio di naming. In caso di malfunzionamenti (al primo esempio) sarebbe consigliabile la esecuzione via shell di comandi, in modo da controllare precisamente directory corrente e parametri forniti alla macchina virtuale.

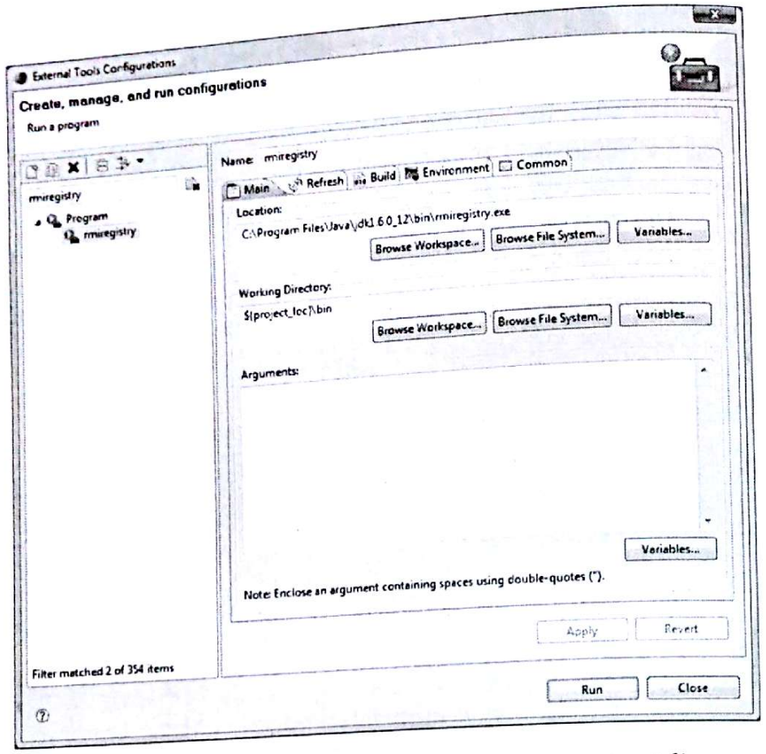

Figura 4.3: *La configurazione di esecuzione di rmiregistry in Eclipse: nella directory corrente (working directory) si specifica la directory dell'elemento selezionato* `${workspace_loc}/${container_path}/` *dove si deve trovare lo stub, oppure si usa* `${project_loc}/bin` *se il progetto si trova altrove (ma sotto la directory* `bin`*). Si suppone che si sia selezionata nel Navigator la directory* `bin` *dove è stato eseguito il comando* `rmic` *e dove si trova anche il server RMI da eseguire.*

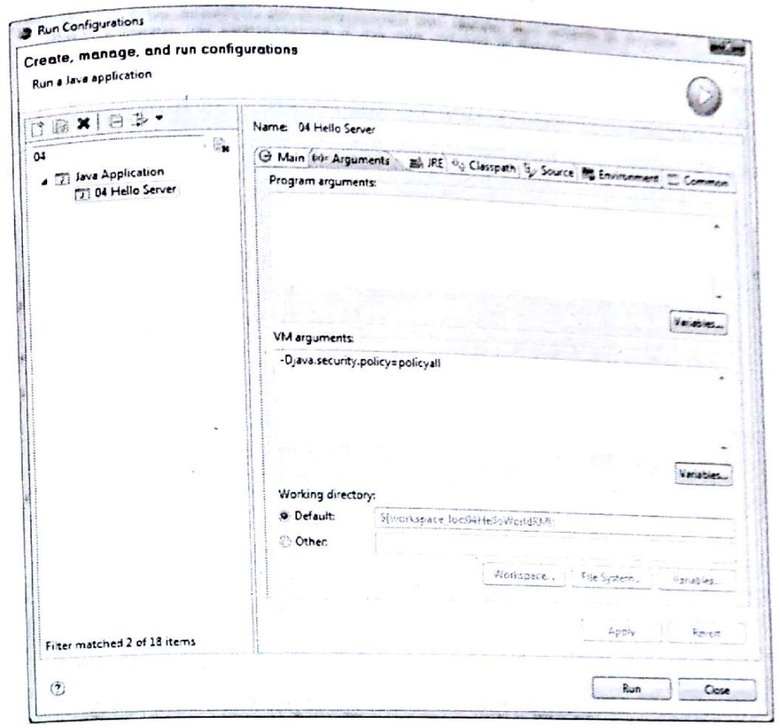

Figura 4.4: *La configurazione di esecuzione di un server RMI in Eclipse: negli argomenti alla macchina virtuale si deve specificare un file che specifichi una politica di sicurezza (in questo caso* `policyall`*). Attenzione al pathname: si suppone che il file policyall si trovi nella directory selezionata sul Navigator/Package Explorer, altrimenti il server non potrà accedervi e ci saranno eccezioni dovute alla sicurezza.*

```
24-gen-2009 15.28.57 HelloImpl main
INFO: Creo l'oggetto remoto...
24-gen-2009 15.29.00 HelloImpl main
INFO: ... ora ne effettuo il rebind...
java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
        at java.security.AccessControlContext.checkPermission(Unknown Source)
        at java.security.AccessController.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkConnect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(Unknown Source)
        at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(Unknown Source)
        at sun.rmi.transport.tcp.TCPEndpoint.newSocket(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.newConnection(Unknown Source)
        at sun.rmi.server.UnicastRef.newCall(Unknown Source)
        at sun.rmi.registry.RegistryImpl_Stub.rebind(Unknown Source)
```

Figura 4.5: *La eccezione lanciata quando non esiste il file di configurazione: una semplice* `java.security.AccessControlException` *che può trarre in inganno.*

Va ricordato che se dimenticate di creare il file di policy, ma lo indicate nella configurazione di esecuzione, l'errore che si verifica può essere fuorviante.. una "semplice" `java.security.AccessControlException` che può trarre in inganno, mentre avete semplicemente dimenticato di creare il file (vedi Fig.4.5).

4.2.7 Registrazione del server sul servizio di naming

Appena lanciato il server, esso deve (di norma) registrarsi sul servizio di naming. Vengono usati metodi del package `java.rmi.Naming` che prevedono semplici modalità di registrazione, richiesta e deregistrazione. Per motivi di sicurezza, non è possibile che il server si registri su un servizio di naming che non sia sul suo stesso *host*, quindi non è possibile avere un servizio di naming "esterno", il che riduce la efficacia di soluzioni distribuite in alcuni casi. Qualora fosse necessario per la nostra applicazione un sistema di naming distribuito, ci si può rivolgere a RMI-IIOP (RMI sul protocollo Internet Inter-ORB Protocol, dell'ambiente Corba) che permette di aggirare questo problema, con il vantaggio ulteriore di aggiungere compatibilità con Corba ed alcuni (piccoli) svantaggi (che vedremo in seguito).

4.2.8 Implementazione del client

La implementazione del client non richiede molte modifiche rispetto ad una eventuale applicazione scritta in locale. In effetti, si deve essenzialmente risolvere il problema della localizzazione dell'oggetto remoto, effettuata tramite i servizi di `java.rmi.Naming` che permettono di ottenere un riferimento remoto ad un oggetto server. A questo punto, la invocazione dei metodi remoti procede in maniera tradizionale, con la sola differenza che si deve gestire la eccezione `RemoteException` che viene lanciata da tutti i metodi remoti. La implementazione dell'invocazione remota risulta essere a carico dello stub.

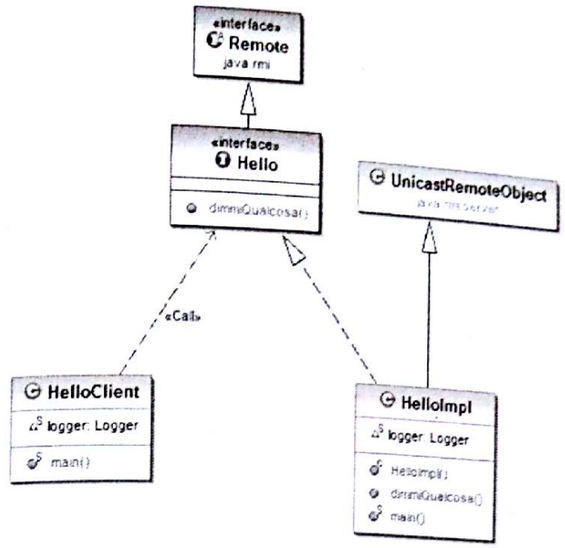

Figura 4.6: *Il diagramma UML delle classi della applicazione* `HelloWorld` *di esempio in RMI.*

4.2.9 Compilazione ed esecuzione del client

La compilazione del client è anche'essa notevolmente semplificata da IDE tipo Eclipse. L'unica attenzione che si deve prestare è al fatto che deve essere presente lo stub[1] del server nella directory dove il client viene compilato. La stessa attenzione deve essere compiuta per la esecuzione.

4.3 L'esempio HelloWorld

Ora, provvediamo a realizzare un esempio "classico" di HelloWorld in Java RMI. La nostra applicazione consisterà di un server che offre un metodo remoto che prende come parametro il nome dell'utente che sta invocando il metodo e restituisce una stringa, che verrà stampata a video dal client. Ovviamente, alcune stampe di controllo ci permetteranno di seguire i vari passi che il nostro server ed il nostro client stanno seguendo. Il diagramma UML delle classi è presentato in Figura 4.6.

Adesso seguiamo i passi precedentemente illustrati con gli esempi specifici.

4.3.1 Definizione della interfaccia remota

Dobbiamo specificare una interface Java che derivi da `java.rmi.Remote` e indicare il metodo remoto con la indicazione dei parametri e del tipo di dati restituito.

Hello.java

```
1 // Interfaccia Hello per HelloWorld
2 public interface Hello extends java.rmi.Remote {
3   String dimmiQualcosa(String daChi) throws java.rmi.RemoteException;
```

[1]Sarete sorpresi dal vedere che nel caso in cui non lo fate, funziona lo stesso! Quello che succede è che a runtime si usa il modello di invocazione JDK 1.5 e successivi, che prevede che lo stub venga (silenziosamente) creato a runtime, prima di essere utilizzato. Di questo ci occuperemo successivamente.

```
4 }
```

Fine: Hello.java

Non ci sono particolari commenti sul codice appena visto; solamente che il metodo `dimmiQualcosa()` deve lanciare esplicitamente la `RemoteException` del package `java.rmi`.

4.3.2 Implementazione del server

Ora, dobbiamo implementare il server. Una convenzione utilizzata per denotare una classe che è la implementazione di una interfaccia, è quello di chiamare il programma come la interfaccia, aggiungendo `Impl` al nome della interfaccia[2]. Quindi, ecco la definizione della classe `HelloImpl.java`

HelloImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.UnicastRemoteObject;
3 import java.util.logging.Logger;
4
5 public class HelloImpl extends UnicastRemoteObject implements Hello {
6   // Serial UID
7   private static final long serialVersionUID = -4469091140865645865L;
8   // Logger per la classe
9   static Logger logger= Logger.getLogger("global");
10
11  // Costruttore
12  public HelloImpl() throws RemoteException {
13    // vuoto
14  }
15
16  // Metodo remoto dimmiQualcosa
17  public String dimmiQualcosa(String daChi) throws RemoteException {
18    logger.info("Sto salutando "+daChi);
19    return "Ciao!";
20  }
21
22  public static void main(String args[]) {
23    System.setSecurityManager(new RMISecurityManager());
24    try {
25      logger.info("Creo l'oggetto remoto...");
26      HelloImpl obj = new HelloImpl();
27      logger.info("... ora ne effettuo il rebind...");
28      Naming.rebind("HelloServer", obj);
29      logger.info("... Pronto!");
30    } catch (Exception e) {
31      e.printStackTrace();
32    }
33  } // end main
34 } // end classe HelloImpl
```

Fine: HelloImpl.java

Iniziamo a commentare le parti sostanziali. Notiamo, innanzitutto, la definizione di un costruttore vuoto alla linee 12-14 , necessario perché (per fare il match con il costruttore della superclasse) deve lanciare la eccezione `RemoteException`. Poi implementiamo il metodo remoto della interface remota, linee 17-20, con il classico comportamento che ci

[2]Una altra convenzione, usata anch'essa, è quella di chiamare le interfacce con un nome che inizi per `I` mentre il nome della classe che la implementa non usa la `I` all'inizio. In questo secondo caso, avremmo avuto `IHello.java` come interface e `Hello.java` come classe remota.

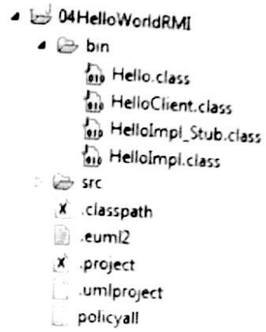

Figura 4.7: *Come appare il progetto HelloWorld a questo punto.*

aspettiamo (restituisce una stringa di benvenuto, mentre stampa sul log chi sta utilizzando il metodo remoto).

È necessario un primo commento sul fatto che il costruttore di un oggetto remoto possa lanciare una eccezione remoto. Durante la costruzione di un oggetto remoto, in pratica, il costruttore di `UnicastRemoteObject` attiva il meccanismo che permette di ricevere invocazioni remote, tra le quali (ad esempio) quella di istanziare, in qualche maniera, un `ServerSocket` in ascolto su qualche porta. Se la rete non è disponibile, o se qualche altro problema si verifica, deve essere possibile rendere evidente questo malfunzionamento, fornendo la informazione che si tratta di un problema dovuto alla natura distribuita dell'oggetto.

Più significativo è invece il `main()` (linee 22-33). Alla linea 23 istanziamo il Security Manager di RMI che serve a poter caricare classi dinamicamente dalla rete (non serve in questo caso) se esse non sono presenti nel `CLASSPATH` della macchina virtuale e che implementa la politica data nel file `policyall`.

Alle linee 24 e 30 si trova il blocco `try catch` che serve per le eccezioni che possono essere lanciate dalla istanziazione dell'oggetto remoto (linea 26) e dall'utilizzo del servizio di Naming attraverso il metodo `rebind()` (linea 28) che registra l'oggetto, con nome `HelloServer` sul registry attivato sullo stesso host dove eseguiremo il programma server.

4.3.3 Compilazione del server

Questa fase è semplice, come detto. Basta lasciar fare alla IDE. Maggiori commenti nel prossimo passo.

4.3.4 Compilazione con lo stub compiler `rmic`

Quello che si ottiene, dopo aver compilato, è la classe `HelloImpl.class`, sulla quale (selezionandola) possiamo eseguire `rmic` configurato come indicato nella sezione precedente. Quello che si ottiene è una situazione simile a quanto mostrato in Fig. 4.7, visualizzato con la Navigator view di Eclipse (non con il Package Explorer che non fa vedere la directory `bin`.

4.3.5 Servizio di naming: `rmiregistry`

A questo punto si può lanciare il registry di RMI, utilizzando le attenzioni illustrate nella sezione precedente.

4.3.6 Esecuzione del server

Possiamo lanciare il server, ed ottenere, dalla console, la informazione che il server è stato creato come oggetto remoto (linea 26).

4.3.7 Registrazione del server sul servizio di naming

Poi, la esecuzione della linea 28 porta alla registrazione dell'oggetto sul registry, che viene indicata da console dal corrispondente messaggio, che indica anche che il server è pronto per effettuare i servizi che gli verranno chiesti.

4.3.8 Implementazione del client

La implementazione di un client di un server remoto, come detto nella sezione precedente, non è particolarmente differente dal caso locale. Alcune differenze principali possono essere evidenziate dal semplice client illustrato di seguito.

HelloClient.java

```
1 import java.rmi.*;
2 import java.util.logging.Logger;
3
4 public class HelloClient {
5   // Logger per la classe
6   static Logger logger= Logger.getLogger("global");
7
8   public static void main(String args[]) {
9     try {
10      logger.info("Sto cercando l'oggetto remoto...");
11      Hello obj = (Hello) Naming.lookup("rmi://localhost/HelloServer");
12      logger.info("... Trovato! Ora invoco il metodo...");
13      String risultato = obj.dimmiQualcosa("Pippo");
14      System.out.println("Ricevuto:"+ risultato);
15    } catch (Exception e) {
16      e.printStackTrace();
17    }
18  }// fine main
19 }// fine classe HelloClient
```

Fine: HelloClient.java

Innanzitutto, trattandosi di invocazioni remote (sia l'utilizzo del servizio di naming che la vera e propria invocazione remota del metodo) questa parte del programma va raccolta in un `try catch` (linee 9 e 15). Alla linea 11, viene effettuato il `lookup()` del server sul servizio di Naming. Questo viene effettuato passando come parametro una URL che specifica `rmi:` come protocollo, poi indica il server `localhost` ed infine indica la id `HelloServer` con il quale il server si è registrato.

Va anche notato che il metodo `lookup()` restituisce un risultato di tipo `java.rmi.Remote` (cioè la interfaccia marker che denota tutte le interfaccie remote) e che dobbiamo necessariamente fare il casting di tale risultato alla interfaccia remota specifica per il servizio `Hello`.

In questa maniera, il riferimento `obj` è un riferimento remoto ad un oggetto server di cui si ignora la implementazione e si conosce esclusivamente i servizi che vengono offerti tramite la interfaccia remota `Hello`.

A questo punto, alla linea 13 viene invocato il metodo remoto, passando come parametro il nome dell'utente ("Pippo" nell'esempio) e poi (linea 14) si stampa quello che è stato restituito dalla invocazione remota del metodo `dimmiQualcosa()`.

4.3.9 Compilazione ed esecuzione del client

Anche questo, in questo caso, è molto semplice ed è affidato alla IDE. Attenzione va rivolta a rendere accessibile (in caso di progetti separati) le classi di cui ha bisogno il client: la interfaccia remota `Hello.java` e (se si usa il modello RMI pre-JDK 1.5) lo stub dell'oggetto server `HelloImpl_stub.java`

4.4 La sicurezza e la policy del Security Manager

Come presentato nel paragrafo 3.5.3, Java è un linguaggio che, sin dal momento della presentazione, ha posto una certa enfasi sulla sicurezza, scelta progettuale che è stata implementata di 4 livelli di sicurezza garantiti dal linguaggio, dal Classloader, dal Bytecode verifier e dal Security Manager.

Per poter eseguire un programma (come l'HelloWorld che abbiamo scritto) che sappiamo utilizzerà risorse "critiche" quali la rete, dobbiamo fare in modo che il server venga lanciato soggetto al controllo di un Security Manager che permetta, in questo caso, alcune operazioni che di solito, invece, vengono proibite, in quanto potenzialmente pericolose.

Ricordiamo che il Security Manager effettua un controllo a run-time di tutti gli accessi a risorse sensibili (file locali, socket, informazioni sull'ambiente della VM, etc.) e che deve essere istallato per poter attivare i controlli di accesso. Ad esempio, applet e applicazioni WebStart sono caricate sempre con un Security Manager istallato, mentre le applicazioni locali non caricano automaticamente un Security Manager, che, come nel caso delle applicazioni RMI, deve essere esplicitamente caricato.

RMI fornisce un proprio Security Manager, sottoclasse di `SecurityManager`, chiamato `RMISecurityManager`. Il class loader non caricherà alcuna classe da siti remoti se non è stato istanziato un security manager. Comunque, `RMISecurityManager` non fa altro che implementare la stessa politica di `SecurityManager` quindi è ininfluente quale dei due venga utilizzato. Questo permette che si possano anche usare security manager specifici per la applicazione, derivati da `SecurityManager`. Si può istanziare un security manager direttamente da codice:

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

oppure si può invocare la macchina virtuale con il parametro su linea di comando `java -Djava.security.manager=java.rmi.RMISecurityManager` oppure `java -Djava.security.manager=default` o anche `java -Djava.security.manager`.

Nel semplice esempio appena mostrato, non è stato necessario istanziare un security manager, in quanto non si prevede di dover caricare classi da remoto. In caso questo sia necessario, se non istanziamo un security manager, il class loader lancerà una eccezione quando cercheremo di effettuare questa operazione, impedendola.

I permessi che vengono offerti dal security manager sono determinati in base alle informazioni che caratterizzano la classe (determinati al livello precedente dal Class loader): da dove viene il codice e se risulta essere certificato (con una firma digitale) (oltre ovviamente ai permessi di default).

Le politiche di sicurezza sono evolute durante le versioni di Java. Con la prima versione (1.0), era possibile solamente distinguere tra codice locale e remoto, nella 1.1 fu possibile distinguere tra codice certificato caricato da remoto, in modo da fornire accesso con le stesse possibilità del codice locale. A partire dalla versione 1.2 viene inserita la politica di accesso attuale, con il Security Manager che è in grado di distinguere la politica di accesso a seconda del dominio di provenienza.

Un Security Manager (o meglio una sua istanza) delimita i confini della *sandbox*, una area protetta i cui confini (permessi di default) sono estremamente ben definiti e preci-

si. Il file di policy che viene fornito ad un Security Manager specifica esplicitamente le (uniche) possibili vie di uscita dalla sandbox, cioè in quali casi è possibile derogare dalla politica di accesso di default. È possibile specificare un file di policy per macchina virtuale (nella directory `jre/lib/security/java.policy`) oppure a livello utente, oppure a livello programma (come abbiamo visto, mediante il parametro `java -Djava.security.policy=policyfile`).
Il formato dei permessi è di questa forma:

```
grant [signedByName][codebase URL]{
        // lista dei permessi
};
```

L'esempio che segue è quello di un file di permessi standard (versione JDK 1.5), dalla facile lettura e comprensione, che ci permette di valutare la fine granularità che si può ottenere nel controllo dei permessi, fino a poter controllare la possibilità di poter fermare thread, le singole operazioni permesse su ogni porta, la lettura delle proprietà specifiche dell'ambiente (nome e versione della macchina virtuale, etc.).

```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {
  permission java.security.AllPermission;
};

// default permissions granted to all domains
grant {
  // Allows any thread to stop itself using the java.lang.Thread.stop()
  // method that takes no argument.
  permission java.lang.RuntimePermission "stopThread";

  // allows anyone to listen on un-privileged ports
  permission java.net.SocketPermission "localhost:1024-", "listen";

  // "standard" properies that can be read by anyone
  permission java.util.PropertyPermission "java.version", "read";
  permission java.util.PropertyPermission "java.vendor", "read";
  permission java.util.PropertyPermission "java.vendor.url", "read";
  permission java.util.PropertyPermission "java.class.version", "read";
  permission java.util.PropertyPermission "os.name", "read";
  permission java.util.PropertyPermission "os.version", "read";
  permission java.util.PropertyPermission "os.arch", "read";
  ...
};
```

4.5 Commenti conclusivi

Concludiamo presentando alcune delle semplificazioni introdotte a scopo didattico in questo capitolo.

- Il client ed il server si trovano nello stesso progetto. Questo di solito non avviene (chi scrive il server e chi scrive il client sono separati nello spazio (team diversi) e nel tempo (client vengono scritti anche molto dopo che il server è stato realizzato). Questo significa che alcune problematiche sono state eluse, quali, ad esempio, capire

esattamente di quali file ha necessità il server ed il client, e come fare per poterli reperire (magari a run-time).

- La politica di security utilizzata è molto grezza: se ne sconsiglia vivamente l'utilizzo in un ambiente reale di utilizzo.
- Si prevede l'utilizzo del modello di stub/skeleton precedente a JDK 1.5, che rende evidente (a scopi didattici) la creazione e l'utilizzo dello stub (almeno!). In effetti, con il modello JDK 1.5 e successivi, è possibile non creare neanche lo stub e farlo creare (silenziosamente) a runtime.

Note bibliografiche

L'esempio di HelloWorld è simile un pochino in tutti i testi indicati nella introduzione, e si può anche trovare online nei tutorial di Java RMI. Maggiori dettagli sulla sicurezza e sui file di policy possono essere trovati in [14] ed nel cap. 8 di [31]. Maggiori dettagli sul security manager si trovano nel cap. 6 di [13].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, contraddistinti da una **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

Il processo di creazione di un programma RMI

1. **[R*]** Quali sono i passi che permettono di creare un oggetto server con Java RMI?
2. **[R*]** Quali sono i passi che permettono di creare un oggetto client con Java RMI?
3. **[R**]** Quali passi della creazione di un oggetto server devono essere effettuati prima di quella di un oggetto client?
4. **[R*]** A cosa serve `rmic`?
5. **[R**]** Quali sono tutte le classi generate per poter eseguire un client-server con Java RMI 1.2? E con le versioni successive 1.3, 1.4 e 1.5?
6. **[P**]** Una volta creata una applicazione client-server in Java RMI, quali sono i passi strettamente necessari se (1) si modifica la implementazione del server; (2) si modifica la implementazione del client; (3) si modifica la interfaccia?

La sicurezza e la policy del Security Manager

7. **[R*]** Come vengono specificate le politiche di sicurezza alla JVM?
8. **[R**]** Per quale motivo è necessaria una certa cautela nelle politiche di sicurezza se si usa Java RMI? Quali sono i potenziali pericoli?

Programmazione con Java RMI

Capitolo 5

Design Pattern con Java RMI

Indice

5.1 Introduzione **100**

5.2 L'Adapter **100**

5.2.1 Un esempio di Adapter: un contatore remoto 100

5.2.2 Contatore remoto: il server 101

5.2.3 Contatore remoto: il client 105

5.2.4 Alcuni commenti all'esempio 106

5.3 La Factory **106**

5.3.1 Un esempio di Factory: HelloWorld multilingua 107

5.3.2 HelloWorld multilingua: il server 107

5.3.3 HelloWorld multilingua: il client 110

5.3.4 Alcuni commenti all'esempio 111

5.4 L'Observer **111**

5.4.1 Un esempio di Observer: la callback per le architetture client-server 112

5.4.2 Awareness con callback: la architettura 113

5.4.3 Awareness con callback: lato client 114

5.4.4 Awareness con callback: lato server 117

5.4.5 Alcuni commenti all'esempio 119

Note bibliografiche **119**

Spunti per lo studio individuale **120**

5.1 Introduzione

Iniziamo con il proporre alcuni semplici esempi di uso di Java RMI che introdurranno alcune problematiche del calcolo distribuito, ma che serviranno anche a presentare alcuni problemi tipici, la cui soluzione rappresenta un esempio di *design pattern* software.

Un *design pattern* rappresenta una soluzione riutilizzabile a problemi che occorrono e si ritrovano spesso all'interno del processo di sviluppo software. In un certo senso, rappresentano una formalizzazione della esperienza di un programmatore esperto, che ha verificato la validità di un certo tipo di soluzione a problemi che possono essere caratterizzati in maniera simile. I pattern vengono descritti utilizzando la descrizione del problema, con un contesto che ne permette la definizione in termini concreti, una presentazione sintetica delle considerazioni che hanno portato alla soluzione, una soluzione generale, con commenti circa gli aspetti positivi e negativi di usare tale soluzione ed una lista di pattern correlati.

In questo capitolo presentiamo tre semplici applicazioni distribuite con Java RMI che rientrano in tre design pattern principali: l'Adapter, la Factory e l'Observer.

5.2 L'Adapter

Una classe Adapter implementa una interfaccia conosciuta dai suoi *Client*[1] e fornisce accesso ad una classe (chiamata *Adaptee*) non conosciuta da essi. Un Adapter fornisce funzionalità ma senza doversi basare su una implementazione, che viene schermata completamente ai Client dell'Adapter.

Questo pattern rappresenta uno dei pattern strutturali più importanti. Attraverso di esso, una classe Adapter fornisce un servizio (tramite la interfaccia) per una classe Adaptee, che non deve (o non può) essere modificata per fornire i servizi direttamente ai Client, implementando la interfaccia appropriata.

5.2.1 Un esempio di Adapter: un contatore remoto

In questo esempio di uso di Adapter vediamo una semplice applicazione remota, nella quale, però, introduciamo anche alcune altre interessanti caratteristiche.

Innanzitutto, all'interno di questa applicazione, dal lato server, distingueremo due classi diverse, quella che offre il servizio remoto al client (la classe di *service*) e quella che, invece, gestisce il ciclo di vita del service (la classe *server*) che si occuperà di istanziare il service, di registrarlo sul registry e di controllarne il funzionamento. La seconda caratteristica che introdurremo sarà quella di distinguere e tenere separate (come faremo d'ora in poi) la implementazione del server e quella del client: come accade nella realtà lo sviluppo viene effettuato in tempi diversi, posti/team diversi e quindi dedicheremo allo sviluppo del server e del client un progetto separato.

L'esempio in cui applichiamo l'Adapter è alquanto semplice: un contatore che è stato definito in locale deve essere acceduto (anche) da remoto attraverso un oggetto remoto che deve offrire verso l'esterno i metodi del contatore locale. Inoltre, l'oggetto remoto mantiene anche un log degli accessi e delle operazioni effettuate, in modo da permettere un controllo in locale. Quindi, in questo caso, l'Adapter sarà l'oggetto remoto server, mentre l'Adaptee sarà il contatore locale.

[1] Per la definizione di Adapter, si usa il termine Client per indicare oggetti che ne invocano i metodi e non presuppone che gli oggetti "Client" siano remoti alla classe Adapter.

5.2.2 Contatore remoto: il server

Iniziamo con il definire la (semplicissima) logica di business della nostra applicazione, definendo un contatore intero locale, definito nella seguente classe.

LocalCounter.java

```
1 public class LocalCounter {
2   // costruttore
3   public LocalCounter(int v)   {
4     value = v;
5   }
6
7   // legge il valore del contatore
8   public synchronized int localGetValue() {
9     return value;
10  }
11
12  // incrementa il valore del contatore
13  protected synchronized void increment() {
14    value++;
15  }
16
17  //variabile istanza
18  private int value;
19 }
```

Fine: LocalCounter.java

Ovviamente, questo semplice esempio non ha bisogno di significativi commenti, se non per sottolineare due caratteristiche. Innanzitutto, è da rimarcare che questo contatore è costruito in maniera tale da non permettere di poter essere decrementato: abbiamo solamente i metodi di lettura (`localGetValue()`) e di incremento (`increment()`). Questo tecnica di wrapping (anche essa, in un certo senso, esempio dell'Adapter) permette di forzare la esecuzione di operazioni semanticamente corrette su un contatore. Ad esempio, un numero di protocollo per registrazioni contabili non può essere mai decrementato ed una classe progettata in questa maniera garantisce "*by design*" la correttezza delle operazioni. Inoltre, sottolineiamo come i metodi di accesso in lettura e scrittura del contatore sono sincronizzati, in maniera da permettere a thread diversi di operare in maniera atomica sul valore, stabilendo una relazione di dipendenza temporale tra le invocazioni (l'ordine in cui sono accadute le operazioni).

In Figura 5.1 vediamo la architettura della applicazione lato server. Il semplice contatore locale che abbiamo definito permette di offrire da remoto i servizi definiti dalla interfaccia remota `Counter`, semplici servizi di lettura del valore e di incremento del valore. Ogni metodo deve anche portare la "firma", con un campo `from` per poter distinguere chi ha effettuato l'accesso.

Counter.java

```
1 import java.rmi.*;
2
3 public interface Counter extends Remote {
4   int getValue(String from) throws RemoteException;
5   void sum(String from, int valore) throws RemoteException;
6 }
```

Fine: Counter.java

Dal diagramma in Figura 5.1 si evidenzia, ora, una particolarità. Una delle maniere di offrire l'adapter, in ambito remoto, è quello di estendere la classe locale, i cui metodi

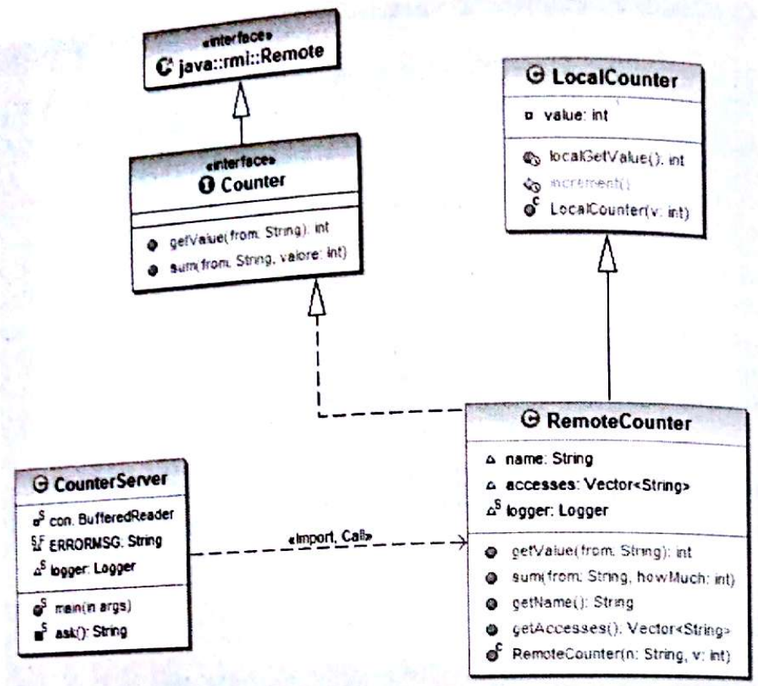

Figura 5.1: *Il diagramma UML delle classi della applicazione* `RemoteCounter` *lato server.*

locali vanno esposti in remoto, aggiungendo i metodi remoti specifici. Ma questo pone un problema: un oggetto remoto dovrebbe estendere `UnicastRemoteObject` e in Java non esiste la ereditarietà multipla. Quindi, usiamo la maniera alternativa in cui una classe può definire oggetti remoti: utilizzare il metodo `exportObject()` per esportare l'oggetto come remoto. Prima di passare al codice dell'oggetto remoto server, è necessaria una osservazione: ovviamente, non è possibile ridefinire i metodi di LocalCounter in `RemoteCounter` in modo che siano accessibili da remoto, in quanto la firma del metodo (la *signature*) andrebbe cambiata per poter lanciare la eccezione *RemoteException*. Questo, in un certo senso, rende necessario l'utilizzo del pattern Adapter.

Alcuni commenti sono necessari per quanto riguarda la esportazione esplicita dell'oggetto. Innanzitutto, esiste anche il metodo per rendere non più accessibile da remoto un oggetto remoto, ma senza distruggerlo: si usa il metodo `unexportObject(java.rmi.Remote obj, boolean force)` che può forzare la operazione anche se ci sono chiamate pendenti, usando il flag booleano `force` a `true`. Va anche notato che, come detto in precedenza, la implementazione dei metodi `hashCode()`, `equals()` e `toString()` rimane a carico del programmatore (in questo caso non li abbiamo implementati).

RemoteCounter.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5
6 public class RemoteCounter extends LocalCounter implements Counter {
7   static Logger logger = Logger.getLogger("global");
8
```



```
9   public RemoteCounter(String n, int v) throws RemoteException {
10    super (v);
11    name = n;
12    UnicastRemoteObject.exportObject(this);
13    try {
14      Naming.rebind(name,this);
15    } catch (Exception e){
16      logger.severe("Problemi con la rebind:" + e.getMessage());
17      e.printStackTrace();
18    }
19    String cr ="Nuovo counter creato il: " + new Date();
20    accesses.add(cr);
21    logger.info(cr);
22    }
23
24  public int getValue(String from) throws RemoteException {
25    int app = this.localGetValue();
26    String cr = "getValue da "+from+" (" + new Date()+"): "+app;
27    accesses.add(cr);
28    logger.info(cr);
29    return app;
30  }
31
32  public void sum (String from, int howMuch) throws RemoteException {
33    for (int i=1; i<= howMuch; i++)
34      this.increment();
35    String cr = "sum "+howMuch+" da "+from+" (" + new Date()+"): nuovo "+this.localGetValue();
36    accesses.add(cr);
37    logger.info(cr);
38  }
39
40  public String getName() {
41    return name;
42  }
43
44  public Vector<String> getAccesses() {
45    return accesses;
46  }
47
48  String name;
49  Vector<String> accesses = new Vector<String>();
50 }
```

Fine: RemoteCounter.java

Iniziamo con l'osservare che, come necessario, la classe deriva da `LocalCounter` e non da `UnicastRemoteObject`, ma che implementa una interfaccia remota `Counter` (linea 6). Il costruttore (linee 9-22) assolve diversi compiti. Innanzitutto, essendo un oggetto che deve essere remoto, e dovendo chiamare il metodo `exportObject` deve necessariamente dichiarare che lancia la eccezione remota `RemoteException`. Alla linea 14 viene effettuato il rebind dell'oggetto, e poi si inizia ad inserire in un vettore di accessi `accesses`, dichiarato alla linea 49, la creazione del counter. Infatti, uno dei servizi aggiuntivi che si richiede a questo oggetto adapter è quello di fornire anche un log degli accessi, che può essere visualizzato in locale.

I due metodi remoti sono offerti alle linee 24-30 (il metodo `getValue()`) ed alle linee 32-38 (il metodo `sum()`). Entrambi lanciano (come da "contratto" vincolante della interfaccia remota) la eccezione remota. Il metodo `getValue()` non fa altro che prelevare, usando il metodo `localGetValue()` che si eredita da `LocalCounter` per poi scrivere nei log (ed a video tramite un logger, se si è abilitato il livello di log `info`) la informazione su

chi ha fatto richiesta del valore. Il metodo `sum()` deve necessariamente implementare una operazione elementare per effettuare la somma, visto che `LocalCounter` offre solamente la operazione incremento. Quindi, effettua il solito logging in accesses e a video.

Due metodi locali di servizio, per accedere al nome del contatore remoto ed al vettore di accessi sono forniti alle linee 40-42 e 44-46.

A questo punto, mettiamo in pratica il proposito di suddividere le funzionalità da *service* dell'oggetto remoto (il fornire accesso remoto ad un contatore da parte di `RemoteCounter`, nel nostro esempio) dalla gestione del service (oggetto remoto) stesso, con istanziazione, gestione, interrogazione etc. localizzati in una classe detta di *server* che adesso andiamo a definire.

CounterServer.java

```
1 import java.io.*;
2 import java.rmi.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5
6 public class CounterServer {
7   static Logger logger = Logger.getLogger("global");
8   private static  BufferedReader con=new BufferedReader(new InputStreamReader(System.in));
9
10  public static void main (String args[]) {
11    String cmd;
12    if (System.getSecurityManager() == null)
13      System.setSecurityManager(new RMISecurityManager());
14    try {
15      RemoteCounter cont = new RemoteCounter ("Contatore",0);
16      System.out.println ("Pronto!");
17      while (!(cmd = ask ()).equals("quit")) {
18        if (cmd.equals ("valore"))
19          System.out.println ("localGetValue:"+cont.localGetValue());
20        else if (cmd.equals ("valore(remoto)"))
21          System.out.println ("getValue:"+cont.getValue("Server"));
22        else if (cmd.equals ("nome"))
23          System.out.println ("getName:"+cont.getName());
24        else if (cmd.equals ("movimenti")) {
25          Vector<String> v = cont.getAccesses();
26          synchronized (v) {
27            for (Enumeration<String> e = v.elements() ; e.hasMoreElements() ;)
28              System.out.println(e.nextElement());
29          } //end synchronized
30        } // end if movimenti
31        else System.out.println (ERRORMSG);
32      } // end while
33    }   catch (RemoteException e) {
34      logger.severe("Problemi con oggetti remoti:" + e.getMessage());
35      e.printStackTrace();
36    }    catch (Exception e) {
37      logger.severe("C'è qualche altro problema:" + e.getMessage());
38      e.printStackTrace();
39    }
40    System.out.println("Ciao!");
41    System.exit(0);
42  }
43
44  private static String ask() throws IOException {
45    System.out.print(">> ");
46    return (con.readLine());
47  }
```


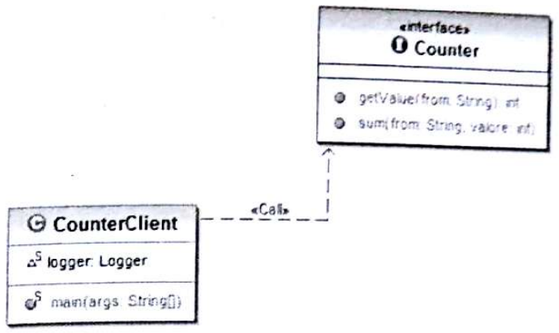

Figura 5.2: *Il diagramma UML delle classi della applicazione* `RemoteCounter` *lato client.*

```
48
49   static final String ERRORMSG = "Uhh? non capisco.";
50 }
```

Fine: CounterServer.java

La classe `CounterServer` ha come obiettivo quella di gestire interrogazioni in locale del contatore, in modo da verificarne il valore, gli accessi effettuati, il nome e (più in generale) le operazioni del ciclo di vita dell'oggetto server. Tutto il lavoro si concentra nel metodo `main` (linee 10-42). Dopo aver istanziato il contatore remoto (con nome `Contatore` e valore 0) alla linea 15, si effettua un ciclo di input dall'utente, che (fino a quando non digita "`quit`") può interrogare il valore del contatore, il nome, e stampare la lista dei movimenti effettuati (linee 24-29). Va segnalato che la differenza tra la lettura del valore in locale (linee 18-19) oppure utilizzando (da locale) il metodo remoto (linee 20-21) si differenzia per due soli motivi: il primo, ovviamente, una (seppur minima) differenza in efficienza visto che nella invocazione remota vengono coinvolti gli strati di rete per effettuare una invocazione remota verso localhost; la seconda sta nel fatto che della lettura mediante accesso locale non rimane traccia nei log, che vengono definiti solamente a livello di oggetto remoto (adapter) `RemoteCounter`.

5.2.3 Contatore remoto: il client

La applicazione client è, ovviamente, alquanto più semplice. Come si vede dal diagramma UML delle classi in Figura 5.2, il client interagisce solamente con la interfaccia remota del contatore `Counter`. Dobbiamo brevemente introdurre e commentare la classe del client.

CounterClient.java

```
1 import java.rmi.*;
2 import java.util.logging.Logger;
3
4 public class CounterClient {
5   static Logger logger = Logger.getLogger("global");
6
7   public static void main (String args[]) {
8     String host = "localhost";
9     if (System.getSecurityManager() == null)
10      System.setSecurityManager(new RMISecurityManager());
11    String nome = args[1];
12    if (args.length==3)
```

```
13        host = args[2];
14      try {
15        Counter cont = (Counter) Naming.lookup("rmi://"+host+"/Contatore");
16        int valore=Integer.parseInt(args[0]);
17        cont.sum (nome, valore);
18        System.out.println ("Totale="+cont.getValue(nome));
19      }  catch (RemoteException e) {
20        logger.severe("Problemi con oggetti remoti:" + e.getMessage());
21        e.printStackTrace();
22      }   catch (Exception e) {
23        logger.severe("C'è qualche altro problema:" + e.getMessage());
24        e.printStackTrace();
25      }
26    }// end main
27  }
```

Fine: CounterClient.java

Il programma prende da linea di comando il valore da incrementare ed il nome del client (linea 11) e (opzionalmente) il nome dell'host su cui risiede il contatore remoto (linee 12-13), altrimenti si usa per default `localhost`. Dopo aver ottenuto il riferimento remoto al contatore (linea 18) effettua la invocazione del metodo remoto (linea 20) passando come parametro l'intero passato su linea di comando. Infine, viene stampato il totale (linea 21) chiamando il metodo per la lettura del valore del contatore remoto.

5.2.4 Alcuni commenti all'esempio

L'utilizzo dell'Adapter permette di stabilire tre livelli di accesso al contatore locale, a seconda della posizione di chi accede (su che macchina virtuale si trova) e della modalità (quale classe viene utilizzata per accedere).

1. A livello locale, nella stessa Java Virtual Machine si può accedere da una classe locale (non riportata nell'esempio) al contatore `LocalCounter` attraverso i metodi `localGetValue()` e `increment()`.

2. È possibile accedere al contatore, sempre a livello locale (nella stessa macchina virtuale), ma attraverso la classe `RemoteCounter` ed in questa maniera, oltre ai metodi di `LocalCounter` ci sono i metodi offerti specifici della sottoclasse, offerti da remoto (ad esempio, `sum()`).

3. A livello remoto, attraverso l'interfaccia `Counter` è permesso l'accesso solamente tramite `getValue()` e `sum()`.

Va fatto notare che alcune possibilità sono offerte a puro scopo didattico, quali, ad esempio, la possibilità di poter accedere in lettura al valore del contatore utilizzando il metodo non "loggato" `localGetValue()`. Se si vuole schermare e proteggere il contenuto del contatore, e regolamentare gli accessi attraverso l'Adapter, si dovrebbe evitare che queste invocazioni possano avvenire, ad esempio, dalla classe di service.

5.3 La Factory

Un altro pattern di particolare importanza è rappresentato dalla Factory[2], un pattern di creazione (*creational pattern*). L'obiettivo è realizzare un meccanismo tramite il quale una

[2]In alcuni testi, viene riferito come *Factory Method*.

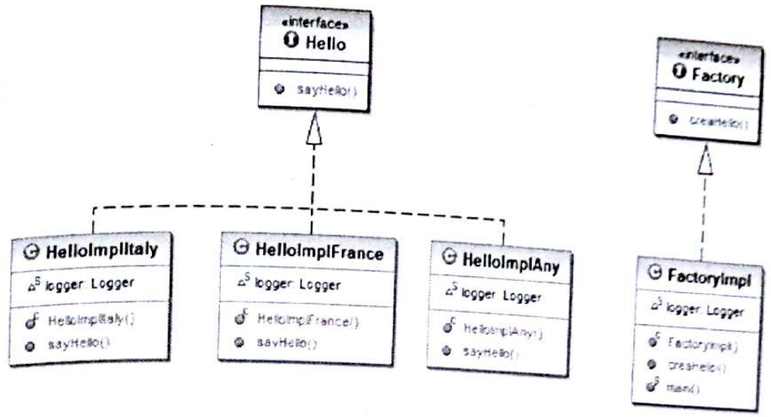

Figura 5.3: *Il diagramma UML delle classi della applicazione* `Factory` *lato server.*

classe (la Factory) può istanziare diversi oggetti ConcreteProduct, anche da diverse classi, che offrano la stessa interfaccia Product verso i CreationRequestor, che richiedono la creazione dei prodotti. La Factory offre verso l'esterno i servizi definiti da una interfaccia IFactory, che permette di rendere questo schema generico e riutilizzabile, in quanto diverse implementazioni di Factory generano diversi insiemi di ConcreteProduct. Inoltre, la classe CreationRequestor è indipendente dalla classe degli oggetti ConcreteProduct, che può essere dinamicamente determinata.

Introduciamo, adesso, un esempio di Factory che usa il pattern per creare un tipo di oggetto diverso, per rispondere ad una richiesta di "Hello" che avviene da client di diversa nazionalità.

5.3.1 Un esempio di Factory: HelloWorld multilingua

Trattiamo adesso un semplice esempio: supponiamo di avere diversi oggetti remoti del tipo "HelloWorld", che si differenziano per la nazionalità[3] con la quale gli utenti si presentano. Quindi esisterà una classe che definisce oggetti che rispondono in italiano, una classe che definisce quelli che rispondono in francese e così via. Quello che vogliamo è permettere ad un client generico, che vuole interagire con un oggetto remoto (ottenuto fornendo la propria nazionalità) che risponde al metodo `sayHello()` in maniera coerente (cioè ad un italiano risponde in italiano, ad un francese in francese, e così via).

5.3.2 HelloWorld multilingua: il server

La applicazione che presentiamo, innanzitutto, lato server è contraddistinta dal diagramma delle classi definite nella Figura 5.3.

Come esempio, abbiamo provveduto a fornire 3 classi di "HelloWorld" diverse, per la lingua italiana, francese ed tutte le altre lingue (in quel caso risponde in inglese!). La interfaccia comune a tutte e 3 le classi è definita di seguito.

Hello.java

[3]Ovviamente la factory viene usata qui solo per illustrare la tecnica. Nel caso specifico esistono soluzioni enormemente più semplici!

```
1 import java.rmi.*;
2
3 public interface Hello extends Remote {
4   String sayHello(String myName) throws RemoteException;
5 }
```

Fine: Hello.java

Ora, la factory deve fornire oggetti che implementino la interfaccia nota al client (unica informazione di cui ha necessità). Ecco la semplice definizione

Factory.java

```
1 import java.rmi.*;
2
3 public interface Factory extends Remote {
4   Hello creaHello(String from) throws RemoteException;
5 }
```

Fine: Factory.java

Ora veniamo alla parte più complicata: ecco la factory.

FactoryImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.logging.Logger;
4
5 public class FactoryImpl extends UnicastRemoteObject implements Factory {
6
7   private static final long serialVersionUID = 1L;
8   static Logger logger = Logger.getLogger("global");
9   public static void main (String args[]) {
10    System.setSecurityManager (new RMISecurityManager());
11    try {
12      new FactoryImpl ("FactoryRemoteHello", args[0]);
13    } catch (Exception e){
14      logger.severe("Problemi con la creazione della factory:" + e.getMessage());
15      e.printStackTrace();
16    }
17    System.out.println ("Factory pronta!");
18  } // end main
19
20  public FactoryImpl (String nome, String h) throws RemoteException{
21    super();
22    host = h;
23    try {
24      Naming.rebind("rmi://"+host+"/"+nome,this);
25    } catch (Exception e){
26      logger.severe("Problemi con la rebind:" + e.getMessage());
27      e.printStackTrace();
28    }
29  } // fine costruttore
30
31  public Hello creaHello(String from) throws RemoteException {
32    if (from.equals("Italia")) {
33      return new HelloImplItaly("Italy"+italian++);
34    } else if (from.equals("France")){
35      return new HelloImplFrance("France"+french++);
36    }
37    return new HelloImplAny("Any"+any++);
38  }
39
```



```
40  private String host;
41  private int italian = 1;
42  private int french = 1;
43  private int any = 1;
44 }
```

Fine: FactoryImpl.java

La implementazione della Factory deve offrire il metodo `creaHello()`, in accordo con la interfaccia `Factory`, per permettere ai client di ottenere un riferimento remoto ad un oggetto che si comporta come detta la interfaccia `Hello` ma la cui implementazione viene scelta dalla Factory. Il metodo `main()` (linee 9-18) invoca solamente il costruttore della Factory (linee 20-29) che provvede a effettuare il rebind dell'oggetto sul registry con il nome fornito dal `main`. Il metodo `creaHello()` (linee 31-38) non fa altro che scegliere il client appropriato, in dipendenza dalla nazionalità espressa dal parametro passato al metodo dal client. Per l'esempio, abbiamo a disposizione tre implementazioni di "HelloWorld" diverse: una italiana, una francese ed una per le altre lingue (che risponderà in inglese). Come si evince dal diagramma in Figura 5.3, tutte le classi di "HelloWorld" implementano la stessa interfaccia: al proprio interno, però, possono essere diverse (e nel nostro caso la differenza starà solamente nella lingua in cui rispondono al client). A seconda della nazionalità, la Factory, quindi, risponderà restituendo un `HelloImplItaly` (linea 33), un `HelloImplFrance` (linea 35) oppure un `HelloImplAny` (linea 37). Ad ogni oggetto "HelloWorld" viene passato (tramite parametro sul costruttore) un identificativo, che viene ottenuto concatenando un intero che tiene conto di quanti oggetti di un certo tipo sono stati creati. Quindi, ad esempio, si usa la variabile `italian`, definita alla linea 41, concatenata alla stringa `Italy` per definire la id degli oggetti che rispondono in italiano. La variabile viene incrementata subito dopo essere stata utilizzata (linea 33 per l'oggetto "Hello" italiano, ad esempio).

Ecco la definizione della classe che implementa l'oggetto che risponde in italiano.

HelloImplItaly.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.logging.Logger;
4
5 public class HelloImplItaly extends UnicastRemoteObject implements Hello {
6   private static final long serialVersionUID = 1L;
7   static Logger logger = Logger.getLogger("global");
8
9   public HelloImplItaly (String id) throws RemoteException{
10    super();
11    myid = id;
12  } // fine costruttore
13
14  public String sayHello(String myName) throws RemoteException {
15    try {
16      logger.info(myid+": Ricevuto "+myName+"@"+getClientHost());
17    } catch (ServerNotActiveException e) {
18      logger.severe("Problemi con la getClientHost:" + e.getMessage());
19      logger.info(myid+": Ricevuto "+myName+"@"+"unknown");
20      e.printStackTrace();
21    }
22    return "Ciao "+myName+"!";
23  }
24
25  private String myid;
```

```
Factory pronta!
2-feb-2009 19.44.06 HelloImplItaly sayHello
INFO: Italy1: Ricevuto Vittorio@10.0.0.136
2-feb-2009 19.44.26 HelloImplFrance sayHello
INFO: France1: Ricevuto Jacques@10.0.0.136
2-feb-2009 19.44.44 HelloImplAny sayHello
INFO: Any1: Ricevuto John@10.0.0.136
```

Figura 5.4: *Le stampe a video che vengono effettuate lato della factory, mentre ogni client ottiene "Ciao", "Bonjour" oppure "Hello" a seconda della nazionalità che ha specificato.*

```
26 }
```

Fine: HelloImplItaly.java

La classa risulta abbastanza semplici: il metodo `sayHello()` è definito alle linee 20-30. L'unica caratteristica di Java RMI che abbiamo introdotto è l'utilizzo del metodo `getClientHost` che restituisce l'indirizzo del client che sta effettuando la chiamata (per la stampa nel log, linea 23). Il metodo è ereditato dalla classe `UnicastRemoteObject`.

La classe "HelloWorld" in francese è simile, tranne che non fa altro che rispondere usando "Bonjour" al posto di "Ciao", mentre quelle per le altre lingue risponde con "Hi!".

5.3.3 HelloWorld multilingua: il client

Ecco ora, il client che accede ad un oggetto remoto che implementa la interfaccia `Hello`, fornito da una Factory.

Client.java

```
1 import java.rmi.*;
2
3 import java.util.logging.Logger;
4 public class Client {
5   static Logger logger = Logger.getLogger("global");
6
7   public static void main (String args[]) {
8     try {
9       Factory fact = (Factory) Naming.lookup ("rmi://"+args[0]+"/FactoryRemoteHello");
10      Hello hello = (Hello) fact.creaHello(args[2]); // nazionalità
11      System.out.println(hello.sayHello(args[1])); // nome
12    } catch (Exception e){
13      logger.severe("Problemi con le invocazioni:" + e.getMessage());
14      e.printStackTrace();
15    }
16  }
17 }
```

Fine: Client.java

Il client è alquanto semplice: riceve su linea di comando 3 parametri: l'host al quale fare la richiesta per il lookup, il proprio nome e la nazione di provenienza. La invocazione remota avviene alla linea 11, sull'oggetto che è stato restituito dalla Factory nella invocazione remota alla linea 10.

Quello che accade quando viene eseguito da 3 client diversi, ognuno con nazionalità diversa è mostrato in Figura 5.4.

5.3.4 Alcuni commenti all'esempio

Come abbiamo già osservato, l'esempio è particolarmente semplice ma si presta ad alcuni commenti. Innanzitutto, risulta chiaro come il client sia completamente disaccoppiato dalla esistenza delle classi di implementazione, che vengono scelte dalla Factory. I client non sono legati a sapere le diverse lingue in cui il servizio di "HelloWorld" viene offerto dal server, che quindi può offrire servizi piu complessi senza che il client debba essere modificato.

Una altra osservazione va fatta per quanto riguarda il lavoro svolto dall'rmiregistry in questo esempio: seguendo il pattern della Factory, solamente un riferimento remoto viene memorizzato sul registry, mentre i riferimenti remoti vanno forniti direttamente ai client tramite il metodo definito dalla interfaccia `Factory`. Questo significa innanzitutto, spostare il carico dal registry verso la factory, ma soprattutto comporta che la factory risulta essere in grado di poter forzare una politica di accesso e di bilanciamento del carico sugli oggetti ConcreteProduct. Infatti, gli oggetti "HelloWorld" potrebbero essere riutilizzati da client diversi, per ottimizzare le prestazioni (non dovendo effettuare la costruzione di un nuovo oggetto prima di poterlo restituire). La Factory potrebbe avere un *pool* di oggetti "HelloWorld" che sono disponibili per ciascun linguaggio, e restituire a ogni richiesta quello che in questo momento risulta essere meno carico di lavoro[4]. Inoltre, la Factory può fornire oggetti remoti che si trovano su altre macchine virtuali (e quindi su altri host). Immaginiamo che abbiamo a disposizione un cluster di 10 nodi, dove abbiamo lanciato una schiera di oggetti "HelloWorld" in varie nazionalità, pronti a servire i client. Potremmo realizzare una interfaccia addizionale `FullRegistry`, implementata da `FactoryImpl` che permetta la registrazione degli oggetti nel pool di risorse della Factory. In questa maniera, la Factory può suddividere il lavoro tra i vari nodi del cluster, applicando politiche di bilanciamento del carico.

Questo schema può essere usato anche in situazioni dove non si ha a disposizione (o necessità di) un cluster, ma semplicemente si vuole evitare la limitazione di rmiregistry che non permette la registrazione (metodi `bind()` e `rebind()`) ad oggetti che non siano sullo stesso host su cui rmiregistry è stato lanciato. Questa scelta progettuale di Java RMI è stata fatta per problemi di sicurezza, in quanto così si evita (ad esempio) che un oggetto remoto malizioso "esterno" possa sovrascrivere un riferimento remoto esistente, sostituendosi al legittimo oggetto remoto precedentemente registrato, potendo quindi interagire in maniera non lecita con i client. Ovviamente, sarebbe compito della `FullRegistry` prevedere una fase/modalità di autenticazione per permettere di registrare oggetti sulla Factory solo se in possesso delle autorizzazioni necessarie.

5.4 L'Observer

Il pattern di Observer rappresenta un altro fondamentale pattern di tipo comportamentale (*behavioural* pattern). Consiste nello stabilire un meccanismo tramite il quale è possibile registrare delle dipendenze tra oggetti, in modo che un oggetto Observable riceve delle richieste da parte di un oggetto Observer di essere informato (tramite notifica) quando accade un evento. Ovviamente, sono necessarie, per poter interagire, due interfacce: una per poter permettere all'oggetto Observer di registrarsi sull'Observable, ed una da parte dell'Observer per poter ricevere la notifica quando necessario.

Come si può facilmente verificare, questo pattern è ampiamente utilizzato nella gestione degli eventi (anche, ad esempio, per la progettazione di user interface) anche se alcune

[4] Ovviamente, per l'esempio in questione il carico di lavoro è minimo, ma per servizi reali questo potrebbe essere un reale problema.

considerazioni vanno necessariamente esplicitate. Innanzitutto, per poter effettuare tutte le notifiche, un oggetto Observable può impiegare una quantità di tempo non preventivabile in anticipo, in quanto la registrazione è dinamica e permessa a tutti: se vengono registrati 10.000 oggetti Observer, anche un semplice broadcast di un breve messaggio (persino in locale) può ritardare in maniera significativa la applicazione. Una altra considerazione è che questo pattern può essere inserito l'uno dentro l'altro, quindi potrebbero esserci delle dipendenze cicliche: due oggetti possono chiamare ricorsivamente il metodo di notifica dell'altro fino a riempire lo stack e mandare in crash la macchina virtuale (e la applicazione).

Ora presentiamo un esempio, leggermente più complesso dei precedenti, che realizza tramite Observer un meccanismo di callback, una importante tecnica per la programmazione client-server.

5.4.1 Un esempio di Observer: la callback per le architetture client-server

Nelle architetture client-server, le funzionalità sono rigidamente divise tra le due componenti della architettura: il server fornisce servizi ed il client li richiede. Questo significa che lo scambio di informazioni viene sempre iniziato dal client ed il server può solamente rispondere ad una invocazione dei servizi da parte del server.

Questo semplice modello è una notevole semplificazione per il progettista: la asimmetria dei ruoli nella architettura permette di identificare immediatamente la componente nella quale inserire certe funzionalità, e rende estremamente semplice il flusso di dati e le interazioni (che vanno da client verso server per la richiesta, la cui risposta è contenuta nel valore restituito dal metodo/procedura remoto).

Questa asimmetria, però, limita fortemente la natura delle interazioni tra client e server. Come già detto, il server non è in grado di iniziare una comunicazione con il client, problema che in tante architetture client-server viene affrontato con diverse soluzioni. Una classica soluzione è quella del *polling*: il client effettua chiamate ripetute, automaticamente, in modo che il server, se ha bisogno di comunicare con il client, deve solamente attendere che arrivi la prossima richiesta del client, mettendo in un buffer le informazioni/operazioni da inviare. Questo, ad esempio, è la strategia utilizzata da AJAX nel Web 2.0, per rimediare alla irrinunciabile architettura client-server di HTTP. Ovviamente, una considerazione da fare è che la tecnica del polling pone un certo stress sia sul server (che riceve le chiamate ripetute da ogni client) che sul client (che deve inviare ripetutamente le chiamate) che sulla rete (che deve tramettere un numero frequente di richieste/risposte da ogni client al server).

La soluzione che viene adottata, invece, quando si è in grado di poter alterare la natura del protocollo (cosa che in HTTP non è possibile!) è quella della *callback* che segue, tipicamente, il pattern dell'Observer. Con questo meccanismo, il client effettua una richiesta di essere "richiamato" (significato di *callback* in italiano) per ottenere delle informazioni. In questa maniera, le invocazioni di callback vengono effettuate da server verso il client. Questo non significa che i ruoli nella architettura vengono scambiati (come invece avverrà nelle architetture peer-to-peer!): il server continua a essere la componente che offre i servizi, che, però, prevedono che il server faccia richieste anche ai client connessi.

Diversi sono gli utilizzi della tecnica del callback. Innanzitutto, la invocazione di callback può servire per offrire alcune delle caratteristiche delle invocazioni asincrone. Il meccanismo di *remote procedure call* o di *remote method invocation* è essenzialmente sincrono, in maniera da preservare il tradizionale comportamento delle invocazioni in locale: l'oggetto/procedura che sta effettuando la invocazione di metodo/procedura è bloccato

fino a quando il metodo/procedura non ha terminato. Nelle invocazioni asincrone, invece, la richiesta viene inviata al server ed il client continua le sue elaborazioni. Quando i risultati saranno disponibili, la risposta verrà inviata al client. È importante puntualizzare che la asincronia che riusciamo ad ottenere con sistemi di invocazione di metodi/procedure è solamente parziale (ed a questo tipo ci riferiremo nel resto del paragrafo). Infatti, le soluzioni che vedremo non riescono a garantire il disaccoppiamento temporale di client e server, vale a dire non riescono a garantire che la invocazione venga permessa anche se client e server non siano contemporaneamente in esecuzione (completa asincronia). Questa tipologia di chiamate è tipica dei sistemi a scambio di messaggi (*Message Oriented Middleware*) dove un intermediario si occupa di effettuare le invocazioni asincrone da client a server e non verrà trattata.

Per poter avere delle invocazioni asincrone, senza fare uso del polling, ci sono due strade. La prima è percorribile se il linguaggio permette l'uso del multithreading, come Java. In questo caso, infatti, è possibile eseguire invocazioni remote in un thread separato, con il sovraccarico di dover gestire la sincronizzazione delle attività quando la invocazione finalmente termina e, ottenuto il risultato, deve comunicarlo agli altri thread della computazione.

Una altra soluzione è quella della callback. Quando un client vuole invocare un servizio asincrono di un server, fa in modo che il server possa effettuare callback, effettuando una *registrazione* che permetta al server di memorizare le informazioni sufficienti per poter effettuare la callback. Ad esempio, in Java RMI, farà in modo di passare il suo riferimento remoto al server, che permetterà al server di fare la callback. Poi, il client può invocare un metodo remoto "asincrono" ed il server fa in modo di compiere il lavoro in un thread separato, risponde contemporaneamente al client ed effettuerà la invocazione di callback verso il client quando il suo compito sarà terminato ed avrà la risposta.

5.4.2 Awareness con callback: la architettura

Una applicazione della tecnica di callback è quella della "consapevolezza" (*awareness*) delle componenti di una applicazione distribuita: ogni componente deve essere in grado di poter conoscere (in maniera asincrona) cosa stanno facendo le altre componenti. Quindi, ad esempio, un client di una chat deve sapere chi è entrato nella chat e chi ne è uscito. A questo scopo, il client può esportare un metodo di *notifica* che il server chiamerà per far sapere cosa stanno facendo gli altri client.

Partiamo dal classico Esempio di `HelloWorld` che è stato realizzato nel Capitolo 4. La modifica funzionale (per poter applicare la callback) è che il client, dopo aver inviato il suo nickname e ricevuto (e stampato) il messaggio di benvenuto che il server invia, rimane in attesa fino a quando l'utente non preme INVIO, in modo che il server possa informarlo dei successivi client che si connettono, realizzando una primitiva consapevolezza del client rispetto a tutti gli altri client che si connettono dopo di lui.

Ovviamente, quindi, il server, ad ogni ricezione di una invocazione di un nuovo client, dovrà avvisare tutti i client di cui ha informazione, notificandoli della invocazione remota da parte del nuovo client.

Per poter realizzare la callback, abbiamo bisogno di fare in modo che anche l'oggetto client diventi remoto, dovendo esportare il metodo di notifica verso il server.

Quindi, innanzitutto, dobbiamo definire come client e server si scambieranno informazioni. Infatti, se è ovvio che si debba definire una interfaccia di callback per il client per poter esporre il metodo di notifica, verifichiamo quali altri metodi debba esporre il server al client per poter permettere la callback. La scelta è quella di esporre lato server dei metodi che servano per assicurare la possibilità di poter effettuare la callback, permettendo la *registrazione* e la *de-registrazione* del client tra quelli che il server sceglierà di avvertire.

Per separare nettamente le funzionalità specifiche della applicazione (l' *HelloWorld*) e la introduzione della callback per la consapevolezza delle azioni degli altri client, scegliamo di introdurre due nuove interfacce: una che espone il metodo di notifica del client verso il server, chiamata `ClientCallBack`, ed un'altra che serve ad esporre i metodi di registrazione del server verso il client, chiamata `ServerCallBack`.

ClientCallback.java

```
1 import java.rmi.*;
2
3 public interface ClientCallback extends Remote {
4   public void notifyMe(String message) throws RemoteException;
5 }
```

Fine: ClientCallback.java

La specifica del metodo `notifyMe()` è estremamente semplice (in questo caso): viene passato come parametro dal server una stringa che contiene le informazioni sul nuovo client che si è connesso al server.

ServerCallback.java

```
1 import java.rmi.*;
2
3 public interface ServerCallback extends Remote {
4   public void registerForCallback(ClientCallback cl) throws RemoteException;
5   public void unregisterForCallback(ClientCallback cl) throws RemoteException;
6 }
```

Fine: ServerCallback.java

La interfaccia lato server serve esclusivamente a fornire due metodi generici di registrazione e de-registrazione del client sul server. Innanzitutto, commentiamo il tipo del parametro che viene passato ai due metodi. Infatti, entrambi i metodi `registerForCallback()` e `unregisterForCallback()` prendono come parametro un oggetto che implementi la interfaccia `ClientCallback`. Innanzitutto, questo significa che l'oggetto che viene passato è un oggetto remoto (`ClientCallback` eredita da `java.rmi.Remote`) ma ha come effetto quello di obbligare un oggetto che si voglia registrare sul server ad implementare la interfaccia che permette la callback. Quindi, garantisce che nella lista di riferimenti remoti che manterrà il server, tutti potranno essere richiamati per la callback, offrendo il metodo `notifyMe()` definito in `ClientCallback`.

5.4.3 Awareness con callback: lato client

Ora passiamo a vedere la struttura del client. Il diagramma delle classi è riportato in Figura 5.5.

Notiamo, innanzitutto, il ruolo delle due interfacce presentate nella architettura, `ClientCallback` e `ServerCallback`: una di queste `ClientCallback` viene implementata dal client `HelloClient`, mentre l'altra viene utilizzata per la invocazione remota per poter registrare e de-registrare il client. Oltre a queste due interfacce ed al client, risulta presente anche la interfaccia `Hello` che definisce (in maniera analoga all'esempio del Capitolo 4) le funzionalità specifiche della applicazione (vale a dire il metodo `dimmiQualcosa()`.

Hello.java

```
1 import java.rmi.*;
2
3 public interface Hello extends Remote {
4   String dimmiQualcosa(String daChi) throws RemoteException;
```


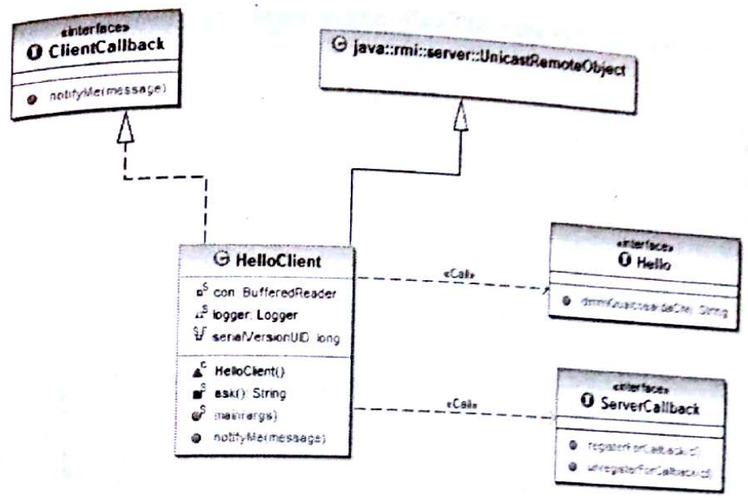

Figura 5.5: *Il diagramma UML delle classi della applicazione* `HelloWorldConCallback` *lato client.*

```
5 }
```

Fine: Hello.java

A questo punto, trattiamo in maniera dettagliata il client. In effetti, il suo compito è alquanto semplice: invocare il metodo remoto messo a disposizione dal server nell'Interfaccia `Hello`, non prima però, di essersi registrato come client interessato alla callback, invocando, quindi, il metodo di `register()` messo a disposizione dal server.

HelloClient.java

```
1 import java.io.*;
2 import java.rmi.*;
3 import java.rmi.server.*;
4 import java.util.logging.Logger;
5
6 public class HelloClient
7     extends UnicastRemoteObject
8     implements ClientCallback {
9
10   private static final long serialVersionUID = 1L;
11   private static  BufferedReader con =
12     new BufferedReader(new InputStreamReader(System.in));
13   static Logger logger = Logger.getLogger("global");
14
15   HelloClient() throws RemoteException {
16     super();
17   }
18
19   public static void main(String args[]) {
20     String host = "localhost";
21     String nome = "Pippo";
22     ServerCallback objCallback = null;
23     HelloClient client= null;
24     if (args.length > 0 ) host = args[0];
25     if (args.length > 1 ) nome = args[1];
26     if (System.getSecurityManager() == null)
```

```
27        System.setSecurityManager(new RMISecurityManager());
28      try {
29        client = new HelloClient();
30        Object obj = Naming.lookup("rmi://"+host+"/HelloServer");
31        Hello objHello = (Hello) obj;
32        objCallback = (ServerCallback) obj;
33        objCallback.registerForCallback(client);
34        System.out.println("Ricevuto:" + objHello.dimmiQualcosa(nome));
35        System.out.println("Premi invio per terminare..");
36        ask();
37      }   catch (RemoteException e) {
38        logger.severe("Problemi con oggetti remoti:" + e.getMessage());
39        e.printStackTrace();
40      }  catch (Exception e) {
41        logger.severe("C'è qualche altro problema:" + e.getMessage());
42        e.printStackTrace();
43      }
44      finally {
45        System.out.println ("Esco..");
46        try {
47          objCallback.unregisterForCallback(client);
48        } catch (RemoteException e1) {
49          logger.severe("Problemi con la unregister:" + e1.getMessage());
50          e1.printStackTrace();
51        }
52        System.exit(0);
53      }
54    }// fine main
55
56    // implementazione metodo di callback
57    public void notifyMe(String message) throws RemoteException {
58      System.out.println ("Notifica:"+message);
59    }
60
61    // metodo di servizio per input
62    private static String ask() throws IOException {
63      System.out.print(">> ");
64      return (con.readLine());
65    }
66  }// fine classe HelloClient
```

Fine: HelloClient.java

Dopo i soliti preliminari delle linee 10-17 (dichiarazione del `serialVersionUID`, dichiarazioni del lettore da standard input, di logger e metodo costruttore), iniziamo a presentare il metodo `main()`. Il programma prende su linea di comando due parametri (opzionali): hostname del server e il nickname da utilizzare. Se non sono presenti come parametro (linee 24-25) vengono usati "`localhost`" e "`Pippo`", definiti alle linee 20-21. Dopo aver installato se necessario un Security Manager, viene istanziato (linea 31) un oggetto remoto di `HelloClient`. Nella linea 39 viene fatto il lookup del server. A questo punto, abbiamo un riferimento remoto di un server, che implementa (come vedremo) due interfacce. Infatti, da un lato, il server offre le funzionalità di un server di *HelloWorld*, quindi implementa la interfaccia remota `Hello`, mentre, d'altro canto, deve permettere la callback e quindi, implementa anche `ServerCallback`. Quindi, per chiarezza e per motivi didattici, distinguiamo in maniera esplicita tra il riferimento remoto che serve per le funzionalità della applicazione cioè `objHello`, ottenuto nella linea 31 facendo il casting di `obj` tramite `Hello`, e il riferimento remoto che serve per permettere le callback dal server, cioè `objCallback`, ottenuto nella linea 32 facendo il casting di `obj` tramite `ServerCallback`.

Alla linea 33, quindi si registra l'oggetto client presso il server, per la callback, mentre

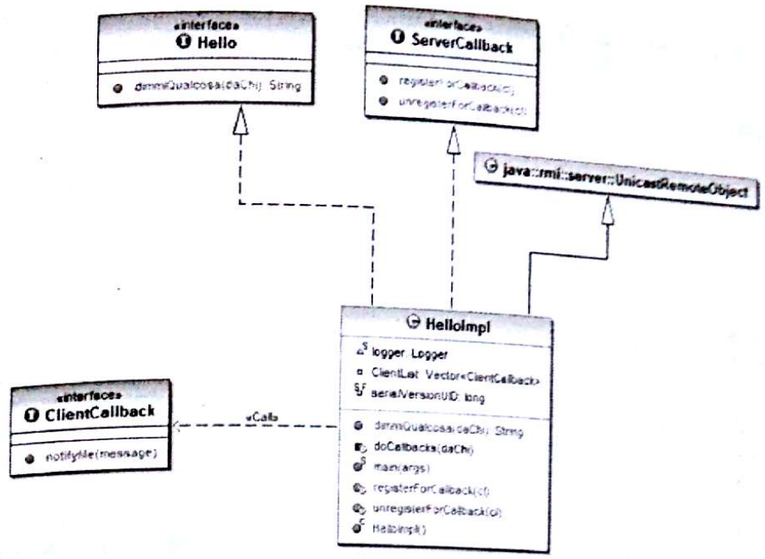

Figura 5.6: *Il diagramma UML delle classi della applicazione* `HelloWorldConCallback` *lato server.*

alla linea 34 viene effettuata la chiamata remota del metodo `dimmiQualcosa`, passando il proprio nome. Le linee 35-36 servono a mantenere il client in attesa di concludere, per permettere tutte le callback che arrivano: quando l'utente preme INVIO, il metodo `ask()` ritornerà e si uscirà dal programma (linea 52). Va infine notato che, dopo le `catch`, il blocco `finally` serve per effettuare la `deregister`, in ogni caso, alla fine del blocco, cioé anche in caso di malfunzionamenti di qualche tipo, che hanno fatto eseguire il blocco di `catch` delle eccezioni.

5.4.4 Awareness con callback: lato server

A questo punto, per analizzare il server presentiamo la architettura in Figura 5.6. Visto che la definizione delle interface `Hello`, `ClientCallback` e `SererCallback` è stata data per il client, dobbiamo solamente presentare il codice del server.

HelloImp.java

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.util.Vector;
4  import java.util.logging.Logger;
5
6  public class HelloImpl   extends UnicastRemoteObject
7    implements Hello, ServerCallback {
8
9    private static final long serialVersionUID = 1L;
10   static Logger logger = Logger.getLogger("global");
11
12   public HelloImpl() throws java.rmi.RemoteException {
13     super();
14     ClientList = new Vector<ClientCallback>();
15   }
16
```

```
17  public static void main(String args[]) {
18    System.setSecurityManager(new RMISecurityManager());
19    try {
20      HelloImpl obj = new HelloImpl();
21      logger.info("Effettuo il rebind...");
22      Naming.rebind("HelloServer", obj);
23      System.out.println("Pronto!");
24    }  catch (RemoteException e) {
25      logger.severe("Problemi con oggetti remoti:" + e.getMessage());
26      e.printStackTrace();
27    }   catch (Exception e) {
28      logger.severe("C'è qualche altro problema:" + e.getMessage());
29      e.printStackTrace();
30    }
31  } // end main
32
33  public String dimmiQualcosa(String daChi) throws RemoteException {
34    logger.info("Saluto "+daChi);
35    doCallbacks(daChi);
36    return  "Ciao!";
37  }
38
39  private synchronized void doCallbacks(String daChi) throws RemoteException {
40    for (int i=0; i< ClientList.size(); i++) {
41      logger.info ("Effettuo callback n."+i);
42      ClientCallback cl = (ClientCallback) ClientList.elementAt(i);
43      cl.notifyMe("Salutato "+daChi);
44    }
45  }
46
47  public synchronized void registerForCallback(ClientCallback cl) throws RemoteException {
48    logger.info ("Sto aggiungendo un client..");
49    ClientList.add(cl);
50  }
51
52  public synchronized void unregisterForCallback(ClientCallback cl) throws RemoteException {
53    logger.info ("Sto rimuovendo un client..");
54    ClientList.removeElement(cl);
55  }
56
57  private Vector<ClientCallback> ClientList;
58 } // end classe HelloImpl
```

Fine: HelloImp.java

Saltando a piè pari la solita parte iniziale, sottolineiamo soltanto nella linea 57 la dichiarazione di un `Vector` che memorizza i riferimenti remoti dei client, che devono implementare la interface `ClientCallback`. Il costruttore alle linee 12-15 non fa altro che inizializzarlo.

Poi, alle linee 33-37 c'è la implementazione del metodo remoto di `Hello`, che risulta diverso dalla prima applicazione vista come esempio esclusivamente per la chiamata ad un metodo privato `doCallbacks()`. Questo viene descritto nelle linee 39-45, e non fa altro che scorrere il `Vector` di client, e di chiamarne il metodo remoto `notifyMe()` che *devono* aver implementato, visto che implementano la interface `ClientCallback`.

Nel `main`, alle linee 17-31, si effettua la classica istanziazione (linea 20) e registrazione (linea 22) del server.

Alle linee 47-55 vengono implementati i due metodi di `ServerCallback`, che non fanno altro che aggiungere il riferimento remoto passato alla lista di client connessi (il metodo `registerForCallback()`, linee 47-50) oppure cancellarlo (il metodo `unregisterForCallback()`, linee 52-55).

5.4.5 Alcuni commenti all'esempio

Innanzitutto, dobbiamo sottolineare come la suddivisione dei servizi remoti in due classi di interfacce, quelle funzionali (`Hello`) e quelle per la callback, abbiamo facilitato la astrazione permettendo lo sviluppo separato della semantica della applicazione e delle funzionalità di *awareness* che vengono fornite alla applicazione. Inoltre tale astrazione permette la compatibilità "verso il basso" in quanto è possibile usare il server visto nell'esempio con il client dell'esempio semplice di `HelloWorld`: non verrà fornito la consapevolezza, ma solamente il semplice servizio di Hello.

Un altro commento merita la architettura delle due interfacce che vengono realizzate per la callback lato server e lato client. Consideriamo, innanzitutto che una delle maniere di presentare una interfaccia remota è quello di presentarla come un contratto tra client e server:

> *"Il server si impegna a fornire questo servizio, se la richiesta del client avverrà usando questo metodo con questa signature."*

Ora, nella nostra architettura, il contratto può essere espresso in questa forma

> *"Se il client si impegna a rispettare il contratto ClientCallback, allora il server si impegna a fornirgli il servizio con il contratto ServerCallback"*

Note bibliografiche

I design pattern sono stati presentati per la prima volta nel noto libro [12] della cosiddetta "Banda dei Quattro" (*Gang of Four*): Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, . Un bel catalogo di pattern in Java è presentato da Mark Grand in [15].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

L'Adapter

1. **[P**]** Ipotizziamo che i metodi di counter locale non siano sincronizzati e lo siano solo quelli corrispondenti del wrapper `RemoteCounter`. In cosa, una soluzione del genere, offre una diversa flessibilità a chi realizza la applicazione?
2. **[P***]** Se usiamo `RemoteCounter` così come è scritta, si possono incontrare dei problemi nell'uso di diverse istanze in una collection Java. Perché?

La Factory

3. **[P*]** Perché la Factory può assolvere alcuni compiti del rmiregistry, prendendosi parte del suo carico di lavoro?
4. **[P**]** In che maniera una Factory può utilizzare un pool di oggetti per favorire il bilanciamento del carico su più macchine?

L'Observer

5. **[R*]** Definire il design pattern dell'Observer.
6. **[R**]** Quali sono i possibili problemi che possono derivare dall'utilizzo di Observer?
7. **[R**]** All'interno dell'esempio di callback, identificare i ruoli di Observer, Observable e delle rispettive interfacce.
8. **[P**]** Perché il metodo `doCallbacks()` deve essere sincronizzato? Cosa potrebbe accadere se non lo fosse?
9. **[P*]** Nell'esempio di callback, cosa succede se un client non si deregistrasse, per errore hardware oppure per errore software? Come potremmo modificare il programma server in maniera da renderlo resistente a malfunzionamenti di questo tipo?
10. **[E*]** A proposito dell'esempio del callback, modificare il programma in modo che ad ogni nuovo client connesso al server venga fornito la lista dei client che sono connessi e in attesa delle notifiche.
11. **[E**]** Eliminare il metodo di `deregister` dall'esempio di callback e fare in modo che venga gestito dal server la assenza di un client che si era registrato.
12. **[P*]** Nell'esempio della callback, come si può fare in modo che si garantisca che chi invoca il metodo offerto da `Hello` implementi anche l'interfaccia di callback?

Capitolo 6

Una chat con Java RMI

Indice

6.1 Introduzione **122**
6.1.1 Le funzionalità di una chat 122
6.2 Chat con i socket **123**
6.2.1 Un esempio con socket TCP 123
6.2.2 Un esempio con socket UDP Multicast 128
6.2.3 Commenti e ulteriori sviluppi 131
6.3 Una chat client-server con Java RMI **133**
6.3.1 La architettura software 133
6.3.2 Il server 133
6.3.3 Il client 136
6.3.4 Commenti e ulteriori sviluppi 139
6.4 Una chat peer2peer **141**
6.4.1 Un primo esempio di peer 141
6.4.2 Commenti e ulteriori sviluppi 146
6.5 Approfondimenti **147**
6.5.1 UDP e indirizzi IP di gruppo 147
Note bibliografiche **148**
Spunti per lo studio individuale **149**

6.1 Introduzione

L'equivalente di "Hello World!" in ambito distribuito è rappresentato da un programma di chat, vale a dire un programma che permetta a tutti gli utenti collegati di comunicare in maniera sincrona con gli altri utenti. La chat viene spesso presentata come l'esempio didattico per eccellenza per la programmazione distribuita [11, 21].

In questo capitolo presentiamo diversi esempi di applicazioni che realizzano una chat, declinandoli secondo le diverse tecnologie e le diverse architetture utilizzate. Il primo tentativo sarà quello di realizzare una chat con i socket, prima con TCP e poi con UDP Multicast, esaminandone le caratteristiche che ne rendono problematica la estensione usando questi strumenti.

Si passa poi all'uso di Java RMI, partendo da un primo esempio, con una semplice architettura client-server, per poi passare ad una architettura peer-2-peer. Discuteremo, negli esempi, di come sia possibile aggiungere facilmente nuove funzionalità, caratteristica offerta dagli ambienti di oggetti distribuiti rispetto alle implementazioni basate sui socket.

6.1.1 Le funzionalità di una chat

Prima di presentare gli esempi, discutiamo che cosa vogliamo che una chat offra ai suoi utenti. Per prima cosa, una chat deve permettere, ovviamente, la possibilità di scambiare messaggi con tutti gli utenti collegati[1]. Questo significa che ogni utente avrà un proprio nickname (dato all'atto del collegamento) con il quale tutti i suoi interventi saranno trasmessi a tutti gli utenti connessi. L'utente (nei nostri esempi) userà una semplice chat testuale, dalla quale potrà uscire, terminando la partecipazione alla chat, utilizzando il comando "`!quit`". Altri comandi possono essere inseriti nei programmi, tipicamente con un carattere iniziale che ne permetta il riconoscimento dal programma (come, ad esempio, il punto esclamativo).

Poi, si deve offrire la cosiddetta *consapevolezza* (awareness) del comportamento degli utenti, informando gli utenti collegati di chi si connette e chi si disconnette dalla chat. In questa maniera, un utente riesce a sapere la composizione del gruppo man mano che essa cambia ed è consapevole degli utenti a cui sta scrivendo.

Una chat deve essere *estendibile* cioé deve permettere di poter aggiungere funzionalità modificando o integrando la semantica delle operazioni compiute. Ad esempio, una chat dovrebbe permettere ad un utente di comunicare solamente con uno dei partecipanti oppure con un sottogruppo (modalità di *whisper*). La chat potrebbe prevedere meccanismi di blocco di un utente (da parte di un coordinatore), oppure la possibilità di prevedere interventi anonimi, senza la indicazione del nome di chi ha effettuato il contributo. Potrebbe essere gestita una politica di accesso, prevedendo una fase autenticazione per poter entrare nella chat. Alcuni interventi potrebbero automaticamente scatenare eventi che devono essere gestiti dal sistema (ad esempio, nel caso di giochi in cui gli interventi degli utenti devono essere interpretati come comandi) oppure devono essere inibiti in presenza di particolari condizioni (ancora in un gioco, ad esempio, si può effettuare un certo comando solamente se in presenza di una condizione del giocatore). Ogni partecipante alla chat deve essere in possesso di alcune informazioni per il bootstrap della applicazione, da fornire al proprio programma (tipicamente attraverso la linea di comando). Ad esempio, queste informazioni possono riguardare l'indirizzo di un server, la porta del socket su cui il server è in ascolto, oppure un indirizzo IP di gruppo multicast. È chiaro che queste informazioni

[1] Una semplicissima evoluzione di questa funzionalità potrebbe essere che i messaggi che un utente invia non vengono visualizzati all'utente stesso.

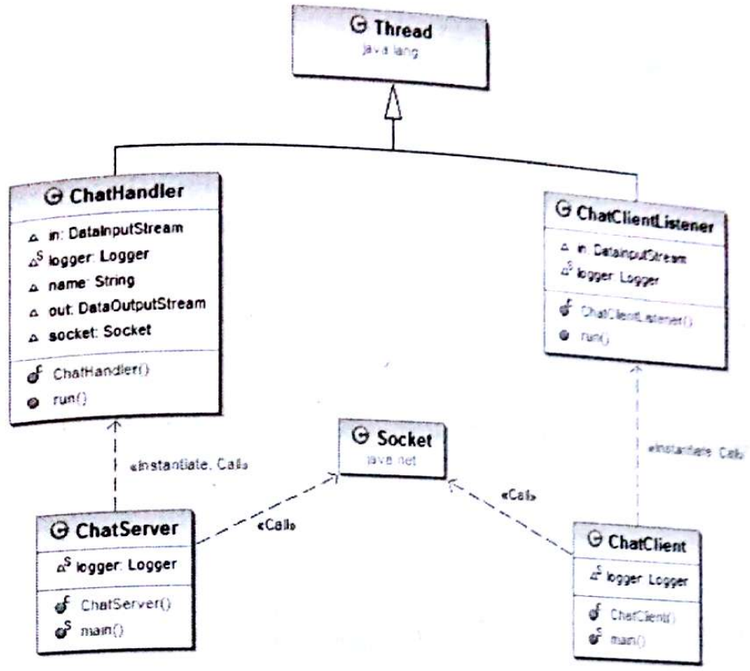

Figura 6.1: *Il diagramma UML delle classi della applicazione della chat in TCP.*

esplicitano la dipendenza della applicazione chat da una infrastruttura che permetta di reperire le funzionalità sulla rete, una sorta di servizio di *naming*.

6.2 Chat con i socket

Per poter meglio apprezzare la programmabilità degli oggetti distribuiti, sperimentiamo un paio di implementazioni di chat basate sui socket TCP e UDP Multicast.

6.2.1 Un esempio con socket TCP

La applicazione di chat che usa i socket TCP ha una architettura client-server. Ogni client si collega al server, usando un socket TCP, ed il server fa in modo di reinviare (usando lo stesso socket) tutto quello che ogni client invia al server (*broadcast*). Il diagramma delle classi della applicazione viene mostrato in Figura 6.1.

Iniziamo con il descrivere la attività del client di chat. Prima, però, di partire con la descrizione della applicazione, dobbiamo definire un semplice protocollo tramite il quale il socket aperto tra client e server viene utilizzato anche per scambiare informazioni di servizio, quale (nel nostro esempio) il nickname utilizzato dal client. Questo va definito in maniera esplicita all'interno della applicazione per sincronizzare client e server. La scelta è semplice: la prima stringa che viene inviato dal client al server rappresenta il proprio nickname. A partire dalla seconda stringa inviata, il server farà il broadcast di quanto ricevuto a tutti i client connessi.

ChatClient.java

```
1 import java.net.*;
2 import java.io.*;
3 import java.util.logging.Logger;
4
5 public class ChatClient {
6   static Logger logger = Logger.getLogger("global");
7
8   public ChatClient(String name, String server, int port) {
9     String cmd;
10    try {
11      socket = new Socket(server,port);
12      in = new DataInputStream(socket.getInputStream());
13      out = new DataOutputStream(socket.getOutputStream());
14      sendTextToChat(name);
15      ChatClientListener lis = new ChatClientListener(in);
16      lis.start();
17      while (!(cmd = ask ()).equals("!quit"))
18        sendTextToChat(cmd);
19      socket.close();
20      System.out.println ("Ciao!");
21    } catch (IOException e) {
22      logger.severe("Problemi con il socket:" + e.getMessage());
23      e.printStackTrace();
24    }
25  }
26
27  protected void sendTextToChat(String str) {
28    try {
29      out.writeUTF(str);
30    } catch (IOException e) {
31      logger.severe("Problemi con l'invio di " + str+"." + e.getMessage());
32      e.printStackTrace();
33    }
34  }
35
36  private static String ask() throws IOException {
37    System.out.print(">> ");
38    return (con.readLine());
39  }
40
41  public static void main (String args[]) {
42    if (args.length == 3) {
43      int port=Integer.parseInt(args[2]);
44      new ChatClient(args[0], args[1], port);
45    } else {
46      logger.severe("Syntax: java ChatClient <name> <serverhost> <port>");
47      System.exit(-1);
48    }
49  }
50
51  private Socket socket;
52  private DataInputStream in;
53  private DataOutputStream out;
54  private static BufferedReader con = new BufferedReader(new InputStreamReader(System.in));
55 }
```

Fine: ChatClient.java

Il client accetta su linea di comando 3 parametri: il nickname, l'host su cui il server è in attesa e la porta. Nel metodo `main()` (linee 41-49) si controlla che i parametri ci siano prima di istanziare un oggetto di tipo `ChatClient` (linea 44) al quale vengono passati i tre

parametri.

Il costruttore, innanzitutto, effettua la connessione al server (linea 11) e preleva dal socket ottenuto gli stream di input e di output (linee 11-12) per le successive comunicazioni. Poi, alla linea 13, fornisce il proprio nickname (come dal protocollo precedentemente descritto) utilizzando il metodo di servizio `sendTextToChat()` definito alle linee 27-34, e poi istanzia (e fa partire) un *listener* che rimane in attesa per ricevere dal server quanto viene inviato dagli altri client (linea 15-16). Alle linee 17-18 c'è il loop per ricevere da tastiera quanto deve essere inviato al server, fino a quando l'utente non digita "`!quit`". La lettura avviene utilizzando il metodo di servizio `ask()` che usa lo stream `con` definito alla linea 54. In uscita dal ciclo, alla linea 19, si chiude il socket e si termina il programma.

Il client, quindi, si occupa solamente di inviare quello che digita l'utente verso il server, sullo stream di output che è stato prelevato alla linea 13. Lo stream di input (linea 12) viene passato al `ClientListener` (linea 15) per poter svolgere, in maniera concorrente, la lettura di quanto il server invia a ciascun client connesso. Passiamo a vedere il codice del listener, che è un thread istanziato ed avviato dal client.

ChatClientListener.java

```
1 import java.io.*;
2 import java.util.logging.Logger;
3
4 public class ChatClientListener extends Thread {
5   static Logger logger = Logger.getLogger("global");
6
7   public ChatClientListener(DataInputStream i) {
8     in = i;
9   }
10
11  public void run() {
12    while (true) {
13      try {
14        System.out.print("\n"+in.readUTF()+"\n>> ");
15      } catch (IOException e) {
16        logger.warning("Chiusura del socket sulla lettura:" + e.getMessage());
17        break;
18      }
19    }
20  }
21
22  DataInputStream in;
23 }
```

Fine: ChatClientListener.java

Il costruttore riceve (linee 7-9) lo stream di input dal quale deve effettuare la lettura di quanto invia il server. Nel metodo `run()`, in un ciclo infinito (linee 12-19), il thread non fa altro che leggere (attraverso la chiamata bloccante `readUTF()`) una stringa dallo stream input e stamparla a video, ristampando il prompt a capo (i caratteri ">>" dopo il carattere di *newline* "`\n`") alla linea 14. La chiusura del thread (con la uscita dal ciclo) avviene quando `ChatClient` effettua la chiusura del socket (alla linea 19), il che fa lanciare una `IOException` dalla `receive` che è semplicemente un segnale che il programma sta terminando (ecco perchè il logger usa il metodo `warning` e non il metodo `severe` per segnalare la eccezione) e che quindi anche il thread deve terminare il proprio lavoro. Questa tecnica per terminare un thread (che useremo ancora) viene ritenuta la modalità corretta per chiudere un thread quando questi sta leggendo in maniera bloccante da un socket, piuttosto che usare metodi deprecati come il metodo `stop()`[2]. Verrà anche utilizzata per segnalare

[2] In [3] vengono citate le motivazioni per le quali il metodo di `stop()` viene deprecato.

al server che il client si è disconnesso: alla chiusura del socket il server riconoscerà la disconnessione del client e si comporterà di conseguenza.

Ora passiamo al server. Il server accetta come parametro la porta sulla quale deve accettare le richieste di connessione da parte dei client. Poi usa anch'esso un thread per gestire la singola connessione verso ogni client e per gestire (attraverso un metodo statico) il broadcast a tutti i client.

ChatServer.java

```
1 import java.net.*;
2 import java.util.logging.Logger;
3 import java.io.*;
4
5 public class ChatServer {
6   static Logger logger = Logger.getLogger("global");
7
8   public ChatServer (int port) throws IOException {
9     ServerSocket server = new ServerSocket (port);
10    while (true) {
11      Socket client = server.accept();
12      DataInputStream in = new DataInputStream(client.getInputStream());
13      String name = in.readUTF();
14      logger.info ("New client " + name + " from " + client.getInetAddress());
15      ChatHandler c = new ChatHandler (name, client);
16      c.start ();
17    }
18  }
19
20  public static void main (String args[])  {
21    if (args.length != 1) {
22      logger.severe("Syntax: java ChatServer <port>");
23      System.exit(-1);
24    }
25    try {
26      new ChatServer (Integer.parseInt (args[0]));
27    } catch (NumberFormatException e) {
28      logger.severe("la porta indicata non è corretta:"+ e.getMessage());
29      System.exit(-1);
30    } catch (IOException e) {
31      logger.severe("Problemi con l'uso dei socket:"+ e.getMessage());
32      e.printStackTrace();
33    }
34  }
35 }
```

Fine: ChatServer.java

Il metodo `main()` (linee 20-34) controlla che il parametro passato su linea di comando ci sia (linea 21-24) e che sia un intero corretto (linea 27-30) prima di istanziare una istanza di un oggetto `ChatServer` (linea 26). il costruttore ha un loop infinito (linee 10-17) utilizzato per accettare le connessioni (linea 11), prelevare lo stream di input dal socket restituito dalla `accept` (linea 12), leggere il nome dell'utente (secondo il protocollo che ogni client utilizza) (linea 13) e istanziare e far partire un istanza del gestore della singola connessione, `ChatHandler`, alle linee 15-16.

Gran parte della logica del server si trova sulla classe di gestione della singola connessione. Infatti, oltre a fungere da controparte dell'istanza di `ChatClientListener` sul client (e quindi a scambiare dati attraverso il socket stabilito con il client), serve anche a mantenere memorizzati, in una struttura `Vector` statica, tutti gli handler che sono istanziati, per

poterne utilizzare lo stream di output per inviare in broadcast i metodi. Ecco, in dettaglio, il codice.

ChatHandler.java

```
1 import java.net.*;
2 import java.io.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5
6 public class ChatHandler extends Thread {
7   static Logger logger = Logger.getLogger("global");
8
9   public ChatHandler (String name, Socket socket) throws IOException {
10    this.name = name;
11    this.socket = socket;
12    in = new DataInputStream (new BufferedInputStream (socket.getInputStream()));
13    out = new DataOutputStream (new BufferedOutputStream (socket.getOutputStream()));
14  }
15
16  public void run () {
17    try {
18      broadcast("INFO", "Entra " + name);
19      handlers.addElement (this);
20      while (true) {
21        String message = in.readUTF ();
22        broadcast(name,message);
23      }
24    } catch (IOException ex) {
25      logger.warning("Connessione persa con "+ name +"."+ex.getMessage());
26      ex.printStackTrace();
27    } finally {
28      handlers.removeElement (this);
29      broadcast("INFO", name+" e' andato via");
30      try {
31        socket.close ();
32      } catch (IOException ex) {
33        logger.severe("Il socket era già chiuso?"+ex.getMessage());
34        ex.printStackTrace();
35      }
36    }
37  }
38
39  protected static void broadcast (String from, String message) {
40    synchronized (handlers) {
41      Enumeration<ChatHandler> e = handlers.elements ();
42      while (e.hasMoreElements ()) {
43        ChatHandler handler =  e.nextElement ();
44        try {
45          if (!from.equals(handler.name)) {
46            handler.out.writeUTF (from+":"+message);
47            handler.out.flush ();
48          } // end if
49        } catch (IOException ex) {
50          logger.warning("Connessione persa con "+handler.name+"."+ex.getMessage());
51        }
52      }
53    }
54  }
55
56  Socket socket;
57  DataInputStream in;
```

```
58  DataOutputStream out;
59  String name;
60  protected static Vector<ChatHandler> handlers=new Vector<ChatHandler> ();
61 }
```

Fine: ChatHandler.java

Il duplice compito appena descritto del `ChatHandler` viene evidenziato dai due metodi principali. Il metodo di `run()`, richiamato dal server per far partire il thread, provvede a rimanere in attesa sul socket specifico, passato al costruttore (linee 9-14) e del quale sono stati prelevati stream di input (linea 12) e stream di output (linea 13). La prima operazione che fa `run()` è quella di usare `broadcast()` (che vedremo successivamente) per informare tutti della avvenuta connessione dell'utente (linea 18). Poi passa a "registrare" l'oggetto appena creato (linea 19) nel `Vector` di handler, dichiarato ed inizializzato alla linea 60 come variabile `static`.

Il ciclo infinito del `run()` serve esclusivamente a ricevere ciò che il singolo client, con il quale la istanza del thread è connessa attraverso il socket, invia. Questo viene fatto alla linea 21 mentre alla linea 22 si chiama il metodo statico `broadcast()`. L'uscita dal loop infinito del `run()` avviene in caso il socket venga chiuso (dal client, o dietro esplicita richiesta dell'utente o per un errore di trasmissione). In ogni caso, il server cattura la eccezione generata dalla `readUTF()` bloccante alla linea 24-26 e, in ogni caso con la `finally`, provvede ad eliminare dall'elenco degli handler quello che gestisce l'handler (linea 28), avvisa tutti i rimanenti nella chat (linea 29) e prova, in ogni maniera, a chiudere il socket (per terminare in maniera pulita in caso di errori di qualche tipo).

Il metodo di `broadcast` (linea 39-54) non fa altro che scorrere in maniera sincronizzata (linea 40) il vettore di handler (linee 42-52), utilizzando la variabile `out` di `DataOutputStream`, alla quale ogni handler permette l'accesso, per inviare il messaggio a tutti (linee 46-47). Viene anche effettuato il controllo per non inviare il messaggio al client che lo ha generato (linea 45).

6.2.2 Un esempio con socket UDP Multicast

Il secondo tentativo per costruire una chat si basa sull'uso dei socket multicast UDP. Per alcune applicazioni di rete, non si ha la necessità del tipo di servizio che fornisce TCP, in particolare, della affidabilità del recapito dei pacchetti e della connessione punto-punto attraverso il canale "logico" del socket. UDP fornisce questa semplice modalità di trasferimento di pacchetti (*datagram*) che avviene indipendentemente l'uno dall'altro (quindi non c'è garanzia circa l'ordine di ricezione, ma anche sull'avvenuto recapito dei precedenti datagram).

In Java, il protocollo UDP viene offerto all'interno di `java.net` che fornisce le classi `DatagramPacket`, per i datagram, `DatagramSocket`, per il socket UDP, e `MulticastSocket` che permette di inviare un singolo pacchetto a tutte le applicazioni che stanno leggendo dal socket. Questo realizza la modalità di comunicazione *multicast* dove un messaggio viene ricevuto da molti utenti e non è diretta da un utente ad un altro specifico, come invece capita nella modalità *unicast*.

Ovviamente, questa modalità di comunicazione multicast sembra particolarmente adatta al'uso in una chat. Nella modalità multicast vengono utilizzate quattro operazioni di base:

- *join*, che permette ad una applicazione di unirsi ad un cosiddetto "gruppo" di multicast, per poter inviare e ricevere messaggi al/dal gruppo;
- *leave*, che permette ad una applicazione di abbandonare un gruppo di multicast;

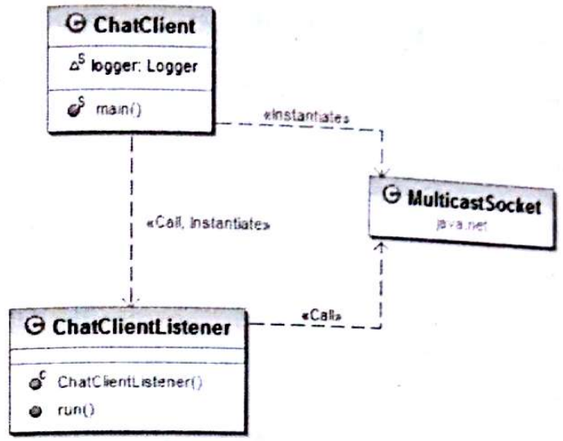

Figura 6.2: *Il diagramma UML delle classi della applicazione della chat in UDP.*

- *send*, che invia un pacchetto al gruppo;
- *receive*, che permette di ricevere un pacchetto che è stato inviato al gruppo sul quale la applicazione ha fatto il join.

Utilizzando il protocollo UDP, presentiamo ora la applicazione della chat con i socket multicast, la cui architettura è mostrata nella Figura 6.2.

Come si vede, la struttura del programma presenta solamente il client. Infatti, in pratica, non esiste server, in quanto la operazione di multicast viene effettuata dallo strato di rete, che si sostituisce alla attività del server. Quindi, ogni client deve semplicemente istanziare un socket Multicast sul quale, contemporaneamente, agirà per la scrittura, e leggerà attraverso un thread che è in ascolto sul socket (`ChatClientListener`).

Iniziamo ad esaminare, quindi, il client della chat.

ChatClient.java

```
import java.net.*;
import java.io.*;
import java.util.logging.Logger;

public class ChatClient {
  static Logger logger = Logger.getLogger("global");

  public static void main (String args[]) {
    int port;
    String nome;
    InetAddress gruppo;
    MulticastSocket chatSocket;
    String cmd="";
    if (args.length!=3) {
      logger.severe("Sono necessari tre parametri: nome IPAddressGroup port");
      System.exit(1);
    }
    nome = args[0];
    port = Integer.parseInt(args[2]);
    try {
      gruppo = InetAddress.getByName(args[1]);
      chatSocket = new MulticastSocket(port);
      chatSocket.setTimeToLive(1);
```

```
24        chatSocket.joinGroup(gruppo);
25        cmd= "(entra " + nome +")";
26        DatagramPacket p = new DatagramPacket(cmd.getBytes(), cmd.length(),gruppo, port);
27        chatSocket.send(p);
28        ChatClientListener lis = new ChatClientListener(chatSocket);
29        lis.start();
30        while (!(cmd = ask ()).equals("!quit")) {
31            cmd = nome + ": "+cmd;
32            p = new DatagramPacket(cmd.getBytes(), cmd.length(),gruppo, port);
33            chatSocket.send(p);
34        }
35        chatSocket.close();
36      } catch (IOException e) {
37        logger.severe("Problemi sulla creazione/uso/chiusura del socket "+e.getMessage());
38        e.printStackTrace();
39      }
40      System.out.println ("Ciao!");
41    }
42
43    private static String ask() throws IOException {
44      String s = "";
45      System.out.print(">> ");
46      try{
47        s= con.readLine();
48      } catch (IOException e){
49        logger.severe("Errore nell'input da console "+e.getMessage());
50      }
51      return (s);
52    }
53
54    private static BufferedReader con = new BufferedReader(new InputStreamReader(System.in));
55  }
```

Fine: ChatClient.java

Il programma prende su linea di comando tre parametri: il nome dell'utente (il nickname), l'indirizzo IP di classe D e la porta su cui si deve inviare/ricevere i datagram in multicast. Una volta controllato che i parametri siano correttamente presenti (linee 14-17), si deve istanziare l'indirizzo di gruppo (linea 21). La creazione del socket su una porta avviene alla linea 22, e, dopo (linea 24), viene effettuato il join del socket al gruppo istanziato alla linea 21. Questo meccanismo, apparentemente "contorto", va sottolineato: il socket Multicast viene creato solamente con l'indicazione della porta e poi, successivamente, viene definito l'indirizzo a cui deve inviare e dal quale deve ricevere, di classe D. Va anche sottolineato come il settare il *TimeToLive* a 1 (linea 23) fa in modo che i pacchetti generati non escano dalla rete locale (questo valore indica il numero di router che il pacchetto può passare), anche se il valore di default è proprio 1.

Il primo messaggio inviato è la indicazione che l'utente si è collegato: si crea la stringa (linea 25) e si costruisce (linea 26) il `DatagramPacket` per contenere la stringa, convertendo la string in bytes. Va anche notato come indirizzo multicast e porta vadano anche essi all'interno del datagram appena creato, che viene poi inviato alla linea 27. A questo punto, si istanzia il thread di ricezione e lo si fa partire (linee 28-29). Questo thread (come vedremo) si occuperà della ricezione dei messaggi inviati da altri client sul socket: il compito di questa classe sarà quello invece di curare l'invio dei messaggi.

Il ciclo nelle linee 30-34 serve proprio a prendere l'input dall'utente (attraverso il metodo di servizio `ask()`, linee 43-52) fino a quando l'utente non digita `!quit` (linea 30). Quello che l'utente digita viene concatenato al suo nickname (linea 31) e poi impacchettato in un datagram (linea 32) che viene poi inviato (linea 33). Quando si termina, il socket

viene chiuso (linea 35) e poi si esce dal programma (linea 40).

Va fatto notare che diverse sono le eccezioni lanciate dai vari metodi che usano la rete, tutte istanza di `IOException`, e per motivi di compattezza di codice abbiamo ritenuto opportuno trattare solamente il catch dell'intero blocco, ma che si dovrebbe differenziare tra di esse per comprendere la ragione del malfunzionamento. Inoltre, la eccezione lanciata da `InetAddress.getByName()` è di tipo `UnknownHostException` ma che essendo sottoclasse di `IOException` viene trattata nel caso generale.

Ora possiamo passare al thread che si occupa di gestire la ricezione dei datagram.

ChatClientListener.java

```
1  import java.io.*;
2  import java.net.*;
3
4  public class ChatClientListener extends Thread {
5    private MulticastSocket multisocket;
6    private byte[] buffer = new byte[100];
7    private DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
8
9    public ChatClientListener(MulticastSocket sock) {
10     multisocket = sock;
11   }
12
13   public void run() {
14     while (true) {
15       try {
16         java.util.Arrays.fill(buffer,new Integer(0).byteValue());
17         multisocket.receive(packet);
18         System.out.print("\n"+(new String(buffer).trim())+"\n>> ");
19       } catch (IOException e) {
20         System.out.println ("Connessione terminata");
21         break;
22       }
23     }
24   }
25 }
```

Fine: ChatClientListener.java

Il thread deve semplicemente continuare a leggere "fin quando può" (vedremo cosa significa) da un socket multicast che viene passato al costruttore (linea 9-11). Il ciclo di lettura dal socket avviene alle linee 14-23: per prima cosa si azzera il buffer di byte utilizzato per la lettura (linea 16), poi si rimane in attesa bloccante sulla `receive` (linea 17), per stampare poi la stringa risultante dalla lettura del buffer (eliminando la parte dei caratteri messi a zero dalla linea 16 con il metodo `trim()`). Questo ciclo viene effettuato senza uscire mai, se non quando viene generata una eccezione, dalla `receive`, che indica che il socket è stato chiuso (dalla linea 35 del client), segnale della conclusione della shell interattiva offerta dal client.

6.2.3 Commenti e ulteriori sviluppi

Affrontiamo adesso una analisi critica del tipo di soluzioni che possono essere determinate dall'utilizzo dei socket (TCP e UDP) per realizzare una pur semplice applicazione distribuita come la chat.

Innanzitutto, partiamo da una considerazione generale valida per i due esempi forniti: entrambe le soluzioni utilizzano meccanismi propri del livello inferiore (il trasporto)

per gestire, invece, operazioni che hanno a che fare con la semantica della applicazione (quali la disconnessione di un client, per TCP, oppure il broadcasting dei messaggi con UDP). Questo rappresenta una limitazione notevole per la applicazione che risulta vincolata progettualmente ad essere basata sullo specifico trasporto utilizzato.

Poi, per quanto riguarda la soluzione basata su TCP, va considerato che una applicazione basata su stream risulta difficilmente estendibile con nuove operazioni e funzionalità. Infatti, per poter effettivamente gestire altre operazioni della applicazione distribuita (quali, ad esempio, il blocco di un utente, l'invio di messaggi ad un solo utente (*whisper*), l'invio di immagini/file, etc.) sarebbe necessario implementare un protocollo per poter inviare comandi via socket, da far processare all'altro capo. Questo renderebbe, innanzitutto, più complesso sia client e server, con una fase di parsing dei comandi inviati sul socket, ma renderebbe anche molto poco estendibile le applicazioni, in quanto qualsiasi nuovo comando per una nuova funzionalità va inserito all'interno del codice specifico del parser. Se le soluzioni che si affidano al livello di trasporto per alcune operazioni sono sicuramente più semplici, sono anche, però, notevolmente limitate in quanto, ad esempio, il server TCP non è facilmente in grado di distinguere malfunzionamenti sul socket da disconnessioni volontarie del client (e simili limitazioni valgono anche per l'uso di UDP, come vedremo). D'altro canto, le soluzioni che prevedono di codificare all'interno dello stream la gestione di comandi permettono sì di aggiungere funzionalità ma ad un costo significativo che rende non facilmente estendibile le applicazioni.

Anche la soluzione basata su UDP, anche se molto semplice nella architettura e non soggetta alle limitazioni della soluzione TCP di parsing dei comandi, soffre di diversi problemi. Innanzitutto, ci sono problemi di tipo operativo dettati dalla natura del protocollo di trasporto utilizzato: la configurazione di molti firewall e router non permette ai pacchetti UDP di uscire da/entrare in reti interne. Inoltre, il meccanismo è di per sé limitato dalla dimensione (fissata) del buffer utilizzato, che limita la lunghezza del messaggio (nell'esempio fissato a 100) e che non permette di gestire in maniera semplice errori che possano sorgere da client "diversi" che usano dimensioni diverse per il buffer (un client che permette l'invio di messaggi di 200 bytes causerebbe problemi a quelli che usano dimensioni più piccole).

I problemi nascono, poi, dalla natura *connectionless* della trasmissione UDP che non solo non permette di verificare i malfunzionamenti ma non offre neanche meccanismi per rilevare la semplice presenza dei client sulla rete. Non c'è alcun modo di conoscere chi è connesso e se, ad esempio, ci sia qualcuno che silenziosamente ascolta quanto trasmesso senza avere informato alcuno della sua presenza. Infatti, in questo esempio di UDP, tutta la gestione del ciclo di vita dell'utente (entrata/uscita dal gruppo) è esclusivamente a carico dello strato di rete (e non della logica di applicazione) e non c'è maniera di controllare se un nodo è uscito per davvero oppure no dal gruppo (se non fidandosi del fatto che tutti stiano eseguendo esattamente lo stesso client!). Questo impedisce, tra l'altro, di gestire qualsiasi politica di controllo degli accessi al sistema: basta essere in grado di inviare/ricevere pacchetti multicast ed un client (qualsiasi client!) è in grado di essere nel sistema.

Ma, soprattutto, questa architettura, che basa il multicasting esclusivamente sullo strato di rete, non permette di poter alterare in maniera agevole la semantica della operazione di multicast. Infatti, non è possibile effettuare modifiche al comportamento, come ad esempio, la possibilità di poter inviare messaggi solamente ad una parte dei client (la modalità di *whisper*) oppure di gestire attivazioni/disattivazioni del meccanismo di scambio di messaggi. Infine, usando il multicasting non è possibile verificare malfunzionamenti di recapito (non vengono lanciate eccezioni se, ad esempio, qualche problema si verifica).

Gli esempi forniti sono abbastanza semplici, e si potrebbero integrare (cosa che lascia-

mo per esercizio). Ad esempio, si potrebbe migliorare la fase di bootstrap in modo da fare in modo che il server usi UDP multicast solamente per fare il broadcast del proprio indirizzo IP in un pacchetto che viene detto di *beacon* (faro). I client leggono la "presenza" del server tramite l'ascolto sul socket multicast, leggono il beacon che contiene indirizzo IP e porta e si connettono tramite socket TCP. In questa maniera, se l'indirizzo di multicast risulta essere "cablato" nel codice (cioè non passato sulla linea di comando ma definito come costante), la chat sembra partire senza alcun parametri: i client "troveranno" il server automaticamente e l'utente dovrà solamente fornire il proprio nickname.

6.3 Una chat client-server con Java RMI

Adesso esaminiamo una soluzione con Java RMI di una chat che si basa su una architettura client-server. L'obiettivo di questo esempio è quello di verificare come la architettura ad oggetti distribuiti facilita la scrittura di una applicazione distribuita dove le funzionalità ed il protocollo risultano chiaramente specificate dalla definizione di classi di oggetti distribuiti, dove lo stato dell'oggetto e il comportamento (i metodi) sono esposti e visibili (attraverso le interfacce remote) ma la cui implementazione risulta essere indipendente (e quindi facilmente modificabile). Inoltre, la strutturazione con un linguaggio orientato ad oggetti permette anche di poter espandere le funzionalità presenti attraverso il meccanismo di derivazione, permettendo la estendibilità della architettura.

6.3.1 La architettura software

In questo primo semplice esempio adotteremo la tradizionale architettura client-server. Ogni client rappresenta un utente e si registra presso il server. In questa maniera, quando uno dei client vuole inviare un messaggio agli altri utenti, lo invia al server che lo reinvia a tutti i client che si sono connessi.

Dal punto di vista di Java RMI, il programma, quindi, ha bisogno di due modalità di interrogazione remota e, quindi, di due interfacce:

- una interfaccia per la chat, composta dai metodi che il server offre verso ciascun client per implementare le funzionalità di base di scambio messaggi e di consapevolezza dello stato.
- una interfaccia per permettere al server di fare *callback* sul client, in modo da fare broadcast del messaggio.

6.3.2 Il server

Il diagramma delle classi della applicazione lato server viene mostrato nella Figura 6.3. Innanzitutto, definiamo la interfaccia per la funzionalità di chat, esposta ed implementata dal server, descritta della interfaccia `IChat.java`.

IChat.java

```
1 import java.rmi.*;
2
3 public interface IChat extends java.rmi.Remote {
4   public void dici (Messaggio m) throws RemoteException;
5   public void iscrivi (IClientCallback idRef, String nickname) throws RemoteException;
6   public void abbandona (IClientCallback idRef, String nickname) throws RemoteException;
```

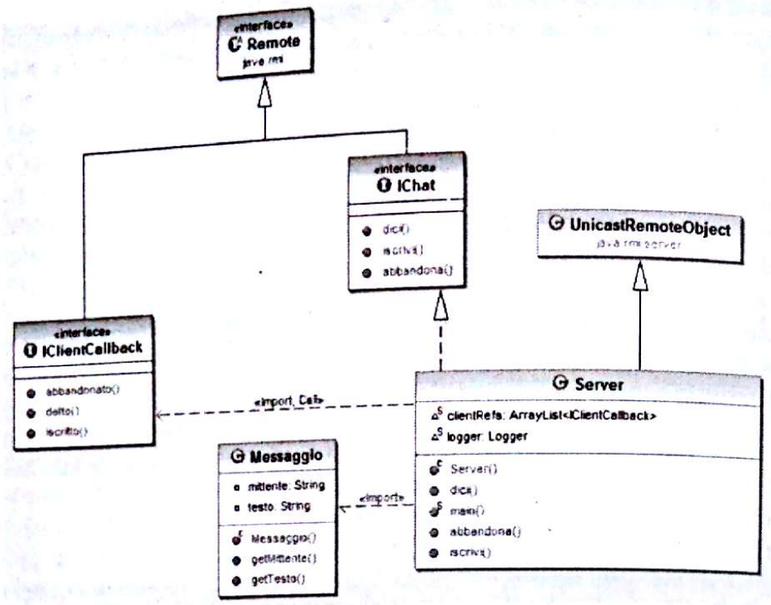

Figura 6.3: *Il diagramma UML delle classi del server della Chat con RMI.*

```
7 }
```

Fine: IChat.java

In questa interfaccia abbiamo messo insieme la funzionalità di chat, esposta dal metodo `dici()` che il server offre ai client, ma anche le funzionalità di iscrizione/abbandono della chat, che servono sia per dare consapevolezza agli utenti su chi sono gli utenti che sono presenti, ma anche per assicurare la attività di call-back. Per semplicità abbiamo inserito le funzionalità in una unica interfaccia, ma sarebbe stato possibile definire due diverse interfacce una per la chat (con il metodo `dici()`) ed una per la consapevolezza/call-back (con i metodi `iscrivi()` e `abbandona()`).

Da notare come la definizione della signature dei metodi `iscrivi()` e `abbandona()` (linee 5 e 6) prevede che il primo parametro sia un riferimento remoto ad un oggetto remoto che implementi la interfaccia `IClientCallback`, in modo da assicurare a tempo di compilazione il buon funzionamento della funzionalità di call-back.

Adesso passiamo alla interfaccia di call-back.

IClientCallback.java

```
1 import java.rmi.*;
2
3 public interface IClientCallback extends java.rmi.Remote {
4     public void detto (Messaggio m) throws RemoteException;
5     public void iscritto (String nickname) throws RemoteException;
6     public void abbandonato (String nickname) throws RemoteException;
7 }
```

Fine: IClientCallback.java

In questa classe, definiamo i metodi che il server richiamerà, sia per la funzionalità di chat (call-back) che per la awareness. I nomi metodi (per metterli in diretta relazione con

quelli della interfaccia `IChat.java`) sono i corrispondenti degli imperativi *dici*, *iscrivi* e *abbandona* ma al participio passato: `detto()` (linea 4), `iscritto()` (linea 6) `abbandonato()` (linea 6).

Adesso passiamo alla parte server.

Server.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5
6 public class Server extends UnicastRemoteObject implements IChat {
7
8   private static final long serialVersionUID = 1L;
9   static Logger logger= Logger.getLogger("global");
10
11  public Server() throws java.rmi.RemoteException {
12  }
13
14  public static void main(String args[]) {
15    System.setSecurityManager(new RMISecurityManager());
16    try {
17      Server obj = new Server();
18      Naming.rebind("ChatServer", obj);
19      System.out.println("Pronto per le connessioni alla chat");
20    } catch (Exception e) {
21      logger.severe("Problemi con oggetti remoti:" + e.getMessage());
22      e.printStackTrace();
23    }
24  } // end main
25
26  public void iscrivi (IClientCallback idRef, String nickname) throws RemoteException {
27    logger.info ("\nEntra "+ nickname +".");
28    for (int i = 0; i < clientRefs.size(); i++) {
29      IClientCallback unClient = ((IClientCallback) clientRefs.get(i));
30      unClient.iscritto(nickname) ;
31    }
32    clientRefs.add(idRef);
33  }
34
35  public void abbandona (IClientCallback idRef, String nickname) throws RemoteException {
36    clientRefs.remove(idRef);
37    logger.info ("\n"+ nickname + " ha abbandonato la chat.");
38    for (int i = 0; i < clientRefs.size(); i++) {
39      IClientCallback unClient = clientRefs.get(i);
40      unClient.abbandonato(nickname) ;
41    }
42  }
43
44  public void dici (Messaggio m) throws RemoteException {
45    for (int i = 0; i < clientRefs.size(); i++) {
46      IClientCallback unClient =  clientRefs.get(i);
47      unClient.detto(m) ;
48    }
49  }
50
51  static ArrayList<IClientCallback> clientRefs = new ArrayList<IClientCallback>();
52 }
```

Fine: Server.java

Questo oggetto remoto implementa la interfaccia di chat (IChat, linea 6) e mantiene una lista di riferimenti remoti dei client che sono connessi nell' ArrayList clientRefs, dichiarato alla linea 51.

Dopo il solito costruttore vuoto[3] parte il metodo main() che non fa altro (linee 16-23) che istanziare e registrare l'oggetto remoto per la chat, e gestire le eccezioni.

La implementazione dei 3 metodi remoti della interfaccia remota IChat si trova successivamente. Alle linee 26-33, il metodo iscrivi() invia la notifica a tutti i client, per la awareness, scorrendo tutti i client presenti in ClientRefs ed invocandone il metodo di awareness iscritto() (linee 28-31). Infine, linea 32, viene aggiunto il riferimento remoto del client iscritto nella lista dei client connessi, in modo da permettergli di ricevere tutti i messaggi inviati dagli altri client.

Alle linee 35-42, si trova il metodo abbandona(), speculare rispetto al precedente: prima elimina il riferimento remoto dalla lista di client (linea 36) e poi invia la notifica a tutti i client rimanenti (linee 38-41).

Il metodo dici() si trova alle linee 44-49 e non fa altro che fare broadcast del messaggio ricevuto a tutti i client connessi.

La classe che definisce il messaggio da inviare è alquanto semplice ed è presentata senza commenti.

Messaggio.java

```
1 public class Messaggio implements java.io.Serializable {
2   private static final long serialVersionUID = 1L;
3
4   public Messaggio(String mit, String tes) {
5     mittente = mit;
6     testo = tes;
7   }
8
9   public String getMittente() {
10    return mittente;
11  }
12
13  public String getTesto() {
14    return testo;
15  }
16
17  private String mittente;
18  private String testo;
19 }
```

Fine: Messaggio.java

6.3.3 Il client

Come si vede dalla Figura 6.4, il client della chat deve necessariamente implementare la interfaccia IClientCallback descritta nel paragrafo precedente.

Il client accetta come parametro il nickname dell'utente che vuole partecipare, mentre il nome dell'host su cui il server sta in esecuzione (e dove, quindi, si trova anche il registry

[3] Ricordiamo che per gli oggetti remoti, il costruttore vuoto deve essere inserito esplicitamente in quanto il costruttore di UnicastRemoteObject lancia la eccezione RemoteException ed il costruttore vuoto, inserito per default dal compilatore per una sottoclasse che non lo dichiari, non lancerebbe questa eccezione.

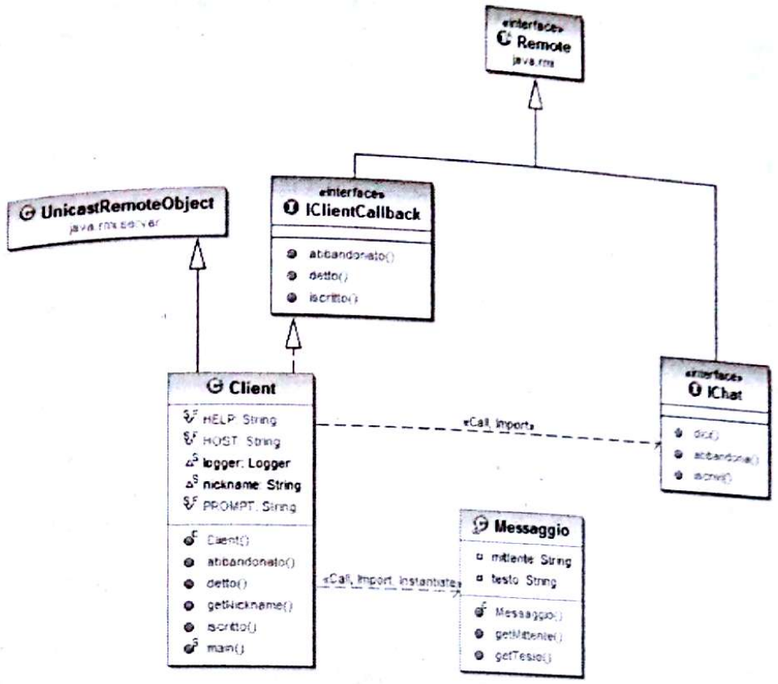

Figura 6.4: *Il diagramma UML delle classi del client della Chat con RMI.*

sul quale effettuare la lookup()) si trova in una costante del programma. Vediamo il codice del client.

Client.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.logging.Logger;
import java.io.*;

public class Client extends UnicastRemoteObject implements IClientCallback {
  private static final long serialVersionUID = 1L;
  static Logger logger= Logger.getLogger("global");

  public Client() throws java.rmi.RemoteException {
  }

  public static void main(String args[]) {
    Client myself = null;
    IChat serverRef = null;
    if (args.length > 0 )   nickname  = args[0];
    else {
      logger.severe("E' richiesto il nickname. Esco..");
      System.exit(1);
    }
    System.setSecurityManager(new RMISecurityManager());
    try {
      serverRef = (IChat) Naming.lookup ("rmi://"+HOST+"/ChatServer");
      myself = new Client();
      serverRef.iscrivi (myself, nickname);
```

```
26      } catch (Exception e) {
27        logger.severe("Non riesco a trovare il server o a iscrivermi. Esco..");
28        e.printStackTrace();
29        System.exit(1);
30      }
31      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
32      String cmd="";
33      System.out.println ("Benvenuto "+nickname);
34      for (;;) {
35        System.out.print(PROMPT);
36        try {
37          cmd = in.readLine();
38        } catch (Exception e) {
39          e.printStackTrace();
40        }
41        if (cmd.equals ("!quit")) {
42          try {
43            serverRef.abbandona (myself, nickname);
44          } catch (RemoteException e) {
45            logger.severe("Non riesco a invocare abbandona() sul server. Esco..");
46            e.printStackTrace();
47            System.exit(1);
48          }
49          break;
50        } else if (cmd.equals ("!help")) {
51          System.out.print(HELP);
52        } else { // si tratta di un messaggio da inviare a tutti
53          if (cmd.length() !=0) {// se la stringa è non vuota
54            Messaggio m = new Messaggio (nickname, cmd);
55            try {
56              serverRef.dici(m);
57            } catch (RemoteException e) {
58              logger.severe("Non riesco a invocare dico() sul server. Esco..");
59              e.printStackTrace();
60              System.exit(1);
61            }
62          }
63        }
64      }// fine for
65      System.exit(0);
66    }// fine main
67
68    public void detto (Messaggio m) throws RemoteException {
69      if (!m.getMittente().equals(nickname))
70        System.out.print ("\n"+m.getMittente()+": "+m.getTesto()+"\n"+PROMPT);
71    }
72
73    public void iscritto (String nickIscritto) throws RemoteException{
74      System.out.print ("\nEntra "+nickIscritto+"."+"\n"+PROMPT);
75    }
76
77    public void abbandonato (String nickAbbandona) throws RemoteException{
78      System.out.print ("\n"+nickAbbandona+ " ha abbandonato la chat"+"\n"+PROMPT);
79    }
80
81    static String nickname;
82    public static final String HOST="localhost";
83    public static final String PROMPT = "Comandi >";
84    public static final String HELP= "Qualsiasi stringa digitata verrà inviata a tutti.\n"+
85    "I comandi disponibili sono: \n"+
86    "\t !quit \t per uscire\n"+
```



```
87    "\t !help \t per stampare il messaggio di help\n";
88  }
```

Fine: Client.java

La classe si compone concettualmente di due parti: una nella quale si implementa la shell per far interagire l'utente con la chat, ed una nella quale si implementano i metodi definiti nella interfaccia remota di callback. Iniziamo a descrivere il main (linee 13-66). Dopo aver controllato la presenza del nickname su linea di comando (linee 16-20), si setta il Security Manager (linea 21) e si provvede ad effettuare la ricerca del server, facendo la richiesta di `lookup()` sul registry (linea 23), localizzato sull' `HOST` definito alla linea 82; poi, si istanzia un oggetto Client (per le callback) alla linea 24 e si provvede ad invocare il metodo per la iscrizione sul server (linea 25).

La shell per i comandi è strutturata in maniera semplice ed usa lo stream di input preso dalla console (linea 31). Un ciclo infinito (linee 34-64) fornisce il prompt (definito alla linea 83) e prende l'input dall'utente (linee 36-40). A questo punto si deve effettuare il parsing del comando: vengono previsti due semplici comandi, uno per uscire ("`!quit`") ed uno per avere un help all'utente ("`!help`"), definito come la stampa della string costante definita alle linee 86-89 (per leggibilità viene definita come concatenazione di stringhe). In caso di richiesta di uscita dal programma, viene invocato il metodo `abbandona()` sul server (linea 43) e poi si esce dal ciclo (linea 49). Altrimenti, se il comando è la richiesta di aiuto, viene stampato l'help (linee 50-52). L'ultimo caso (linee 52-63) è quando si tratta di un messaggio da inviare a tutti i partecipanti attraverso la invocazione remota del metodo `dici()` sul server (linea 56) di un messaggio costruito con il nickname ed il testo digitato (linea 54).

La seconda parte,concettualmente divisa dalla prima, è quella in cui si provvede ad implementare i metodi definiti nella interfaccia `IClientCallback`. I metodi sono `detto()` (linee 68-71), `iscritto()` (linee 73-75) e `abbandonato()` (linee 77-79). I metodi sono alquanto semplici ed auto-esplicativi. Infine, la definizione della classe del client si conclude con la definizione, alle linee 81-87, delle variabili statiche di utilizzo pratico nella classe.

Le altre classi necessarie (`IClientCallback`, `IChat` e `Messaggio`) sono state già illustrate nel paragrafo precedente, quando è stato presentato il server.

6.3.4 Commenti e ulteriori sviluppi

Questo esempio aveva come obiettivo quello di riuscire a rendere evidente la maggiore programmabilità e la superiore capacità di evolvere di una soluzione basata su oggetti distribuiti rispetto a quella basata su socket.

Sottolineiamo, innanzitutto, la versatilità e la scalabilità del meccanismo delle interfacce remote: la possibilità di poter aggiungere nuove funzionalità, aggiungendo nuove interfacce, permette di fare evolvere la architettura senza rinunciare alla compatibilità con le versioni precedenti. Il meccanismo utilizzato per differenziare il meccanismo di chat vero e proprio (nella interfaccia `IChat`) e il meccanismo di consapevolezza (nella interfaccia `IChatCallback`) può essere replicato per poter offrire, ad esempio, un meccanismo di scambio di file, realizzato da client e server che implementino ciascuno una interfaccia `IChatFileExchangeClient` e `IChatFileExchangeServer`, che può essere utilizzato tra una coppia di client "aggiornati" alla nuova funzionalità (semplicemente verificando che un riferimento remoto sia `istanceof` della appropriata interfaccia) mentre i client "obsoleti" possono continuare a utilizzare solo le funzionalità di base offerte da `IChat` e `IChatCallback`.

Poi, va enfatizzato come la natura del protocollo è chiaramente esposta a livello di progettazione dal comportamento definito per ogni oggetto remoto, cioè dai metodi. Non

c'è bisogno (in un certo senso) di dover passare al codice per definire il protocollo, ma basta progettarlo tramite i metodi offerti dalle interfacce: il fatto che, ad esempio, la invocazione della operazione di disconnessione corrisponda alla invocazione da parte del client del metodo remoto `abbandona()` facilita la progettazione del sistema, e rende la codifica molto più semplice e estendibile.

La gestione dei malfunzionamenti è un altro aspetto importante di questo tipo di soluzione, gestione realizzata da Java RMI in maniera esplicita in ossequio ai dettami della non-trasparenza degli oggetti distribuiti discussi nel paragrafo 3.5.2. Innanzitutto, la struttura ad oggetti distribuiti già permette di non affidarsi al livello di trasporto per effettuare operazioni con una valenza semantica sulla applicazione (quali ad esempio la disconnessione) e, quindi, di non sovraccaricare di significato la caduta/chiusura del socket (ad esempio). Java RMI aggiunge a questa caratteristica la possibilità di poter intercettare e trattare esplicitamente i malfunzionamenti in modo da poter agire di conseguenza. Ad esempio, se il server verifica un errore nella invocazione remota con un client, può agire di conseguenza prevedendo di fare, ad esempio, tentativi ripetuti per un certo intervallo di tempo dopo il quale, se nessuno dei tentativi ha avuto successo, il client viene considerato non più collegato e viene eliminato dalla lista dei client connessi.

Utilizzare gli oggetti distribuiti permette anche di limitare al massimo l'impatto della scelta del trasporto. Poichè la applicazione non usa, per le proprie operazioni, alcun meccanismo offerto dal livello di trasporto (come invece facevano sia la chat con TCP che quella con UDP), allora la applicazione risulta essere indipendente dal trasporto utilizzato. Va sottolineato, comunque, che un certo livello di dipendenza permane, in quanto la scelta della classe di oggetti remoti definisce il protocollo utilizzato e quindi, se si volesse usare una altra implementazione di RMI che offrisse un diverso protocollo di trasporto si dovrebbe necessariamente fare qualche piccola modifica (ad esempio, far derivare la classe `Server` non da `UnicastRemoteObject` ma da una altra classe che usa un altro trasporto). Comunque, si è ottenuti una notevole indipendenza nel codice.

Alcuni possibili spunti per miglioramenti o ampliamenti della chat TCP rappresentano degli esercizi per l'approfondimento individuale. Si potrebbe, ad esempio, realizzare il meccanismo di controllo di nick duplicati. Per fare questo si potrebbero prevedere due soluzioni: la prima potrebbe consiste nel lanciare una eccezione (sia lato client che lato server) quando il server si rende conto che un utente si sta connettendo con un nickname già presente. Una seconda soluzione potrebbe, invece, essere quella di modificare il metodo `iscrivi()` sul server in maniera che restituisca un booleano che codifica la avvenuta registrazione (`true`) oppure un errore che si è verificato per nick duplicato (`false`). Una altra possibilità è quella di permettere agli utenti di cambiare nickname durante la conversazione, cosa di cui andrebbero avvisati tutti gli altri utenti.

Una modifica strutturale all'esempio appena visto potrebbe essere quello di modificare la posizione nella architettura dove viene trattato il meccanismo di non stampare a video di un client i messaggi che esso stesso invia. Nell'esempio TCP fornito, questa caratteristica viene trattata lato client: il metodo `detto()` viene invocato dal server su tutti i client connessi, ma ogni client controlla che il mittente del messaggio sia diverso dal proprio prima di stamparlo a video. Sarebbe possibile implementarla lato server, facendo in modo che il server non faccia broadcast del messaggio al client che lo ha inviato: si può fornire ogni client di un metodo accessore remoto per il nickname, oppure si può inserire nella lista di client connessi, una istanza di una classe che mantenga la relazione nickname / riferimento remoto. Questo permette anche di controllare che un client non invochi `iscrivi()` oppure `abbandona()` fornendo un nickname "arbitrario" come parametro.

Ulteriori modifiche alla chat verranno illustrate nel capitolo successivo, dove, sulla base della chat, verranno implementati alcuni esempi di applicazioni distribuite con alcune semplici funzionalità per per l'insieme dei partecipanti.

6.4 Una chat peer2peer

In questo secondo esempio adotteremo una architettura peer2peer: viene eliminata la tradizionale separazione funzionale tra client e server, separazione che, se da un lato facilita la realizzazione ed il deployment, dall'altro rende la applicazione poco fault-tolerant e poco scalabile. Inoltre, va notato che il server non svolge una funzione sostanziale nella chat: serve solamente a fare il broadcast verso tutti i client di ogni messaggio inviatogli da un client. Va però ricordato che la presenza del server assolve anche (in taluni casi) la funzione di fornire la implementazione di politiche di accesso che sono facilmente realizzate avendo a disposizione un singolo punto di accesso alla intera architettura.

La architettura prevede una sola componente, lanciata da ogni utente, uguale per tutti, che agisce da *peer*: questa componente gestirà in proprio tutte le azioni che venivano svolte dal server (registrazione e broadcast dei messaggi) e dal client (shell e metodi di callback per awareness).

In effetti, come si vedrà, la architettura (esplicitata dal diagramma delle classi mostrato in Figure 6.5) contiene in pratica una sola componente significativa, il peer, che implementa una interfaccia remota e che utilizza una classe wrapper per il messaggio da inviare.

Un problema significativo, che affronteremo per gradi, è quello del *bootstrap* di una applicazione P2P. In effetti, in mancanza di un preciso server al quale rivolgersi, un peer che vuole connettersi agli altri peer si trova in difficoltà[4] per effettuare il discovery (scoperta) degli altri nodi.

In questo esempio, adotteremo una soluzione semplicistica (che ha una seria limitazione che tratteremo poi) per poterci concentrare sugli aspetti funzionali: usiamo un `rmiregistry` come una componente che deve mantenere il riferimento remoto di tutti i peer. In questa maniera, un nodo, per poter scoprire chi sono i peer che sono connessi alla chat, si farà dare dal registry la lista degli oggetti remoti registrati e poi, successivamente, li informerà della propria presenza. In questo semplice esempio, assumeremo, anche, che il registry sia utilizzato solamente dalla applicazione di chat, e quindi non discrimineremo (una volta ottenuta la lista dei riferimenti remoti dal registry) tra i riferimenti remoti che appartengono alla applicazione chat (banalmente, quelli che implementano la interfaccia remota di chat) e quelli che invece appartengono ad altre applicazioni.

6.4.1 Un primo esempio di peer

Iniziamo con il definire la interfaccia remota, implementata da tutti i peer. Essa definisce i classici metodi di chat, di iscrizione, di uscita dalla chat e un metodo per reperire da remoto il nickname di un nodo.

Chat.java

```
1 import java.rmi.*;
2
3 public interface Chat extends java.rmi.Remote {
4   public void dico (Messaggio m) throws RemoteException;
5   public void iscrivi (Chat idRef) throws RemoteException;
```

[4] Non sono applicabili nella pratica le tecniche di multicasting con UDP in quanto esse, come abbiamo detto, sono facilmente messe in crisi dalle configurazione dei router che filtrano questi pacchetti.

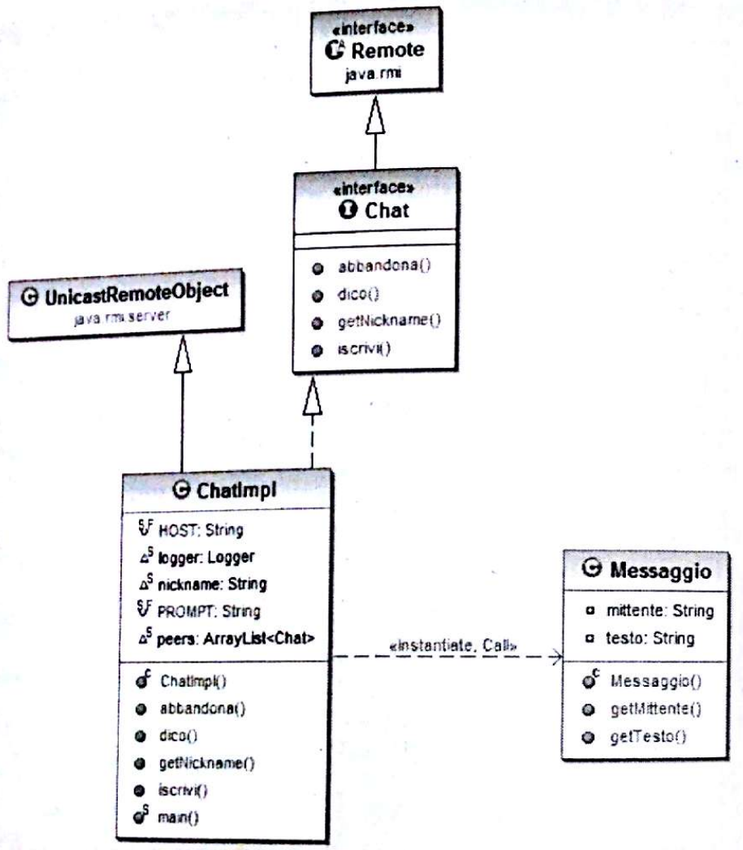

Figura 6.5: *Il diagramma UML delle classi della applicazione Chat P2P.*


```
6   public void abbandona (Chat idRef) throws RemoteException;
7   public String getNickname() throws RemoteException;
8 }
```

Fine: Chat.java

Adesso passiamo alla definizione del peer. Come abbiamo visto anche per il client nell'esempio precedente, ci sono due parti concettualmente separate: una nella quale viene implementata la interfaccia verso l'utente, ed una che implementa i metodi remoti. La differenza sostanziale con l'esempio precedente sta nel fatto che ogni client deve mantenere una propria lista di peer da gestire per inviare i messaggi. Quindi ogni peer invierà (autonomamente) i propri messaggi a ciascuno peer che ha nella propria lista. Ma passiamo a vedere ed illustrare il codice.

ChatImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.*;
4 import java.util.logging.Logger;
5 import java.io.*;
6
7 public class ChatImpl extends UnicastRemoteObject implements Chat {
8
9     private static final long serialVersionUID = 1L;
10    static Logger logger= Logger.getLogger("global");
11
12    public ChatImpl() throws java.rmi.RemoteException {
13    }
14
15    public static void main(String args[]) {
16        Chat myself = null;
17        if (args.length > 0 )   nickname  = args[0];
18        else {
19            System.out.println ("E' richiesto il nickname");
20            System.exit(1);
21        }
22        System.setSecurityManager(new RMISecurityManager());
23        try {
24            String nomi[] = Naming.list("rmi://"+HOST);
25            logger.info ("Risultano connessi "+nomi.length+" utenti:");
26            for (int i=0 ;  i < nomi.length; i++)
27                logger.info ("\t" + nomi[i]);
28            for (int i=0 ;  i < nomi.length; i++)
29                peers.add((Chat) Naming.lookup (nomi[i]));
30            myself = new ChatImpl();
31            Naming.rebind(nickname, myself);
32            logger.info ("Iscrizione presso i partecipanti:");
33            for (int i=0 ;  i < nomi.length; i++) {
34                logger.info ("\tIscrizione a "+ peers.get(i).getNickname()+" in corso..") ;
35                peers.get (i).iscrivi (myself) ;
36                logger.info ("Iscrizione effettuata!");
37            }
38        } catch (Exception e) {
39            logger.severe("Problemi con accesso a oggetti remoti:" + e.getMessage());
40            e.printStackTrace();
41        }
42        try {
43            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
44            String cmd;
45            System.out.println ("Benvenuto "+nickname+". Con te ci sono "+ peers.size()+" ute
46            for (int i = 0; i < peers.size(); i++)
```

```
                System.out.println ("\t"+ peers.get(i).getNickname() );
47          for (;;) {
48              System.out.print(PROMPT);
49              cmd = in.readLine();
50              if (cmd.equals ("!quit")) {
51                  for (int i = 0; i < peers.size(); i++) {
52                      peers.get(i).abbandona(myself) ;
53                  }
54                  break;
55              } else { // si tratta di un messaggio da inviare a tutti
56                  if (cmd.length() !=0) {// stringa non vuota
57                      Messaggio m = new Messaggio (nickname, cmd);
58                      for (int i = 0; i < peers.size(); i++) {
59                          peers.get(i).dico(m) ;
60                      }
61                  }
62              }
63          }// fine for
64      } catch (IOException e) {
65          logger.severe("Problemi con input da tastiera:" + e.getMessage());
66          e.printStackTrace();
67      }
68      try { // eliminiamo il riferimento remoto dal registry
69          Naming.unbind(nickname);
70      }catch (Exception e) {
71          logger.severe("Problemi con input da tastiera:" + e.getMessage());
72          e.printStackTrace();
73      }
74      System.exit(0);
75  }// fine main
76
77  public void dico (Messaggio m) throws RemoteException {
78      System.out.print ("\n"+m.getMittente()+": "+m.getTesto()+"\n"+PROMPT);
79  }
80
81  public void iscrivi (Chat idRef) throws RemoteException {
82      peers.add(idRef);
83      System.out.print ("\nEntra "+idRef.getNickname()+"."+"\n"+PROMPT);
84  }
85
86  public void abbandona (Chat idRef) throws RemoteException {
87      peers.remove(idRef);
88      System.out.print ("\n"+ idRef.getNickname()+" ha abbandonato la chat"+"\n"+PROMPT);
89  }
90
91  public String getNickname() throws RemoteException {
92      return nickname;
93  }
94
95  static ArrayList<Chat> peers = new ArrayList<Chat>();
96  static String nickname;
97  public static final String HOST="localhost";
98  public static final String PROMPT = "Comandi >";
99  }
100 }
```

Fine: ChatImpl.java

Il peer accetta il nickname come unico parametro su linea di comando. Nel `main()`, dopo aver controllato che il parametro sia stato passato (linee 17-21), si istanzia e si usa il Security Manager e si passa alla fase si registrazione nella chat (linee 23-41). Il

meccanismo che viene usato[5] prevede queste fasi:

- Si preleva dal registry la lista di tutti gli oggetti remoti registrati, usando il metodo `list()` (linea 24); ricordiamo che qui si assume che non ci siano altri oggetti registrati se non i peer della chat. Il metodo restituisce un array di stringhe, che vengono stampate a video (linee 26-27).
- Le stringhe con la identificazione di tutti gli oggetti remoti vengono associate a riferimenti remoti, attraverso una serie di `lookup()` (linee 28-29), che vengono aggiunti ad un `ArrayList` di `peers` (definito alla linea 96) che contiene l'elenco iniziale di peer che si sono connessi prima del peer stesso.
- Si istanzia un peer (linea 30) e lo si registra sul registry (linea 31).
- Si registra il proprio indirizzo presso tutti i peer della chat, invocando su ciascuno di loro il metodo `iscrivi()` (linee 33-37).

Ora va trattata la parte di shell interattiva (linee 42-68) che si occupa di comunicare all'utente il nome degli utenti connessi, scorrendo l'array di peer ed invocandone il metodo remoto `getNickname()` (linee 46-47). Poi si entra nella shell (linee 48-64), dove il comando "`!quit`" viene interpretato come richiesta di uscita dalla shell, invocando il metodo `abbandona()` su tutti i peer nell'array e poi uscendo dal ciclo (linee 51-56). Le altre stringhe digitate vengono interpretate come messaggi da inviare a tutti i peer (se non sono stringhe vuote, linea 57) invocando il metodo `dico()` su tutti i peer (linee 57-62).

In uscita dal ciclo, si provvede a eliminare il proprio riferimento dal registry, in modo da non essere trovato dai peer che si dovessero collegare successivamente (linee 69-74) e poi si esce dal programma (linea 75).

A questo punto si passa alla seconda parte, che implementa i metodi remoti. Il metodo `dico()` è alquanto semplice: stampa a video quello che è stato ricevuto, andando a capo e ristampando il prompt (linee 78-80). I metodi successivi sono lievemente più complicati. Il metodo `iscrivi()` (linee 82-85) aggiunge il riferimento remoto alla lista di peer mantenuta in locale. Il metodo `abbandona()` (linee 87-90) fa praticamente la operazione inversa, cioè elimina dall'array il riferimento remoto passato.

La classe Messaggio è molto semplice ed è riportata di seguito.

Messaggio.java

```
1  public class Messaggio implements java.io.Serializable {
2    private static final long serialVersionUID = 1L;
3
4    public Messaggio(String mit, String tes) {
5      mittente = mit;
6      testo = tes;
7    }
8
9    public String getMittente() {
10     return mittente;
11   }
12
13   public String getTesto() {
14     return testo;
15   }
16   private String mittente;
```

[5] Va detto che questa soluzione non è esente da critiche, per quanto riguarda l'accesso contemporaneo di più peer alla chat. Ma tratteremo di questi problemi nella sezione successiva.

```
17   private String testo;
18 }
```

Fine: Messaggio.java

6.4.2 Commenti e ulteriori sviluppi

Innanzitutto, possiamo verificare che la dimensione (in linee di codice) del peer testimonia il fatto che implementa le funzionalità che nell'esempio precedente erano distribuite su client e su server.

Il vantaggio notevole offerto da questa architettura è la *fault-tolerance*: anche in presenza di malfuzionamenti localizzati, il resto delle componenti è in grado di assolvere alcuni dei compiti. Ad esempio, se uno dei peer non funziona, la applicazione continua a funzionare, generando delle eccezioni remote quando gli altri peer provano a contattarlo, ma permettendo (qualora queste eccezioni vengano gestite) il funzionamento degli altri. Anche se il registry dovesse cadere per qualche motivo, il resto dei peer che si trovano connessi possono continuare a chattare: il registry viene usato solamente nel bootstrap. Il miglioramento ottenuto da questa architettura, rispetto alla tolleranza ai malfunzionamenti, è evidente verificando che nella architettura client-server, se il server non funzionava più la *intera* applicazione era impossibilitata a funzionare.

Tuttavia, ci sono due significativi restrizioni/problemi alla soluzione esposta, che descriveremo, insieme ad alcune possibili soluzioni (che possono essere implementate come esercizi).

Il primo problema della soluzione appena presentata è dipendente da una limitazione di Java RMI che impedisce che possano registrarsi sul rmiregistry oggetti remoti che non siano sullo stesso host del registry. In pratica, significa che la soluzione appena presentata, architetturalmente corretta, non funziona in pratica per niente bene: dovremmo avere tutti i peer lanciati sullo stesso host, il che, magari va bene per le nostre prove ma non nella realtà. Il motivo per il quale il registry non permette di fare operazioni di `bind()` e `rebind()` da parte di processi che non siano sulla stessa macchina è quello di assicurare la sicurezza (almeno in minima parte) del sistema: se fosse permesso a chiunque di accedere al registry, una applicazione malevola potrebbe scaricare la lista di tutti gli oggetti remoti registrati, potrebbe fare il rebind di oggetti remoti propri, al posto di quelli presenti, e instradare, in pratica, tutte le invocazioni verso oggetti remoti propri. Invece, con la limitazione appena vista, solamente chi ha accesso alla stessa macchina fisica su cui è in esecuzione il registry (e quindi soggetto ad una certa politica di accesso) può effettuare queste operazioni.

Esistono alcune possibili soluzioni a questo problema. La prima è quella di avere una implementazione leggermente diversa, che preveda che ogni peer faccia partire un proprio registry, e che ogni peer possa scaricare (o avere cablato nel codice) una lista di host noti, da interrogare, verificando la esistenza di un registry, che indica che almeno un peer si trova su quell'host.

Una altra soluzione, è quella di implementare un servizio di bootstrap per la partenza, che sia un oggetto remoto[6] che abbia solamente l'obiettivo di restituire uno o tutti i riferimenti remoti attualmente attivi. La differenza con la soluzione client-server è che questo servizio di bootstrap serve solamente all'inizio e per il broadcast di informazioni esso non viene usato. Questo tipo di soluzione è utile anche per il secondo problema che tratteremo successivamente.

[6] A tutti gli effetti, si tratta di un *server* di bootstrap, anche se il termine server in una architettura P2P non dovrebbe essere utilizzato.

Infine, basta modificare leggermente il programma per utilizzare Java RMI-IIOP che usa servizi di naming che non hanno la limitazione del rmiregistry per ottenere il nostro esempio funzionante anche su host diversi, anche se rimane ovviamente il vincolo di dover conoscere dove è disponibile (su che host) il servizio di naming.

Un secondo problema, questa volta di progetto e non di ambiente, è quello dell'aver completamente ignorato (per semplicità) i problemi dovuti alla concorrenza degli accessi. In effetti, due peer possono provare ad iscriversi contemporaneamente e, quindi, con una interleaving delle operazioni di aggiornamento presso le tabelle dei peer potrebbero non riuscire ad aggiornare la propria tabella uno con l'indirizzo dell'altro. In questo caso, la soluzione potrebbe essere quella di fornire sul servizio di bootstrap una variabile di lock che faccia in modo che la fase di registrazione sia in effetti sincronizzata: un peer accede ad un metodo del servizio di bootstrap in maniera sincronizzata ed evita che durante i suoi aggiornamenti alla tabella ed al registro un altro peer possa accedere. La soluzione di questo problema è lasciato per esercizio.

Va detto, infine, che di solito le applicazioni P2P si basano su tecniche più complesse per la gestione del bootstrap e della localizzazione, quali sistemi di Distributed Hash Tables, infrastrutture di comunicazione, meccanismi di flooding etc.

6.5 Approfondimenti

6.5.1 UDP e indirizzi IP di gruppo

Gli schemi di indirizzamento IP permettono di definire indirizzi di classe diversa, a seconda della quantità di bit che vengono definiti per l'identificazione delle rete e quelli per l'host. In questa maniera vengono definiti gli indirizzi di classe A, B e C che servono per definire reti di dimensione diversa. Lo schema è il seguente:

- Indirizzi di classe A: il primo bit è 0, 7 bit per la rete, 24 bit per l'indirizzo dell'host. Permette 128 reti con più di 16 milioni di host ciascuna, con indirizzo che va da 0.0.0.0 a 127.255.255.255.

- Indirizzi di classe B: i primi 2 bit sono 10, poi ci sono 14 bit per la rete, 16 bit per l'host, per un totale di 16K reti, ciascuna con 64K host. Gli indirizzi vanno da 128.0.0.0 a 191.255.255.255.

- Indirizzi di classe C: i primi tre bit sono 110, poi vengono dedicati 21 bit per la rete e 8 bit per gli host, offrendo un totale di 4M di reti con 256 host ciascuna. Gli indirizzi vanno da 192.0.0.0 a 223.255.255.255.

A questi schemi di indirizzamento vengono aggiunte le reti di classe D (multicast) e E (indirizzamento riservato per future espansioni). Gli indirizzi di gruppo multicast, in classe D, sono caratterizzati dall'avere i primi 4 bit a 1110 e poi ci sono 28 bit per rappresentare la id del gruppo. Gli indirizzi di multicast hanno quindi un dominio che va da 224.0.0.0 a 239.255.255.255 oltre, ovviamente, al numero di porta UDP. Per poter usare un indirizzo multicast, ci sono diverse opzioni: si può ottenere un indirizzo assegnato staticamente dalla Internet Assigned Numbers Authority (IANA); si può usare un indirizzo arbitrario o si può ottenere un indirizzo transitorio (tramite un protocollo come Session Announcement Protocol).

Note bibliografiche

La chat con TCP è una rielaborazione della chat presentata da Boger [6].

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[P*]** Nell'esempio della chat con i socket TCP, perchè nel metodo `broadcast` di `ChatHandler` si rende necessario di fornire accesso in mutua esclusione (`synchronized`) al vettore di handler, quando `Vector` offre già metodi sincronizzati?

2. **[P**]** Scrivere il servizio di bootstrap per la implementazione con RMI di una chat con architetture Peer2Peer.

3. **[P***]** Scrivere il servizio di bootstrap per la implementazione con RMI di una chat con architetture Peer2Peer, in modo da gestire anche i problemi di concorrenza.

Capitolo 7

Alcuni esercizi

Indice

7.1 Introduzione **152**
7.2 Una chat CS per "Cuori solitari" **152**
7.2.1 Il testo dell'esercizio 152
7.2.2 Una soluzione 152
7.3 Una chat CS per un docente dispotico **159**
7.3.1 Un confronto tra differenti strategie 160
7.3.2 Una soluzione 161
Spunti per lo studio individuale **168**

7.1 Introduzione

In questo capitolo presentiamo alcuni esercizi basati sulla applicazione di chat, con commenti e descrizioni. L'obiettivo è quello di presentare alcune semplici soluzioni e di favorire la messa in pratica dei concetti precedentemente appresi.

7.2 Una chat CS per "Cuori solitari"

Questo esercizio prevede una registrazione presso il server dei client in due categorie: "Uomini" e "Donne". Il server fa partire delle chat 1 a 1 tra un uomo ed una donna, in modalità diretta, cioé senza che il server venga coinvolto. La difficoltà maggiore consiste nella gestione del meccanismo di attesa degli client uomini/donne per i quali non esiste ancora un partner.

7.2.1 Il testo dell'esercizio

Progettare e realizzare un programma distribuito in Java (che usi Remote Method Invocation) che realizzi una Chat client-server a coppie (per un Club di "cuori solitari"). La Chat deve offrire la possibilità di fare discussioni a due tra un partecipante "uomo" ed un partecipante "donna". Inizialmente, ogni partecipante si iscrive (comunicando il proprio sesso) al server che provvederà a costruire le coppie: invierà a ciascuno dei due partecipanti il riferimento remoto dell'altro in modo che essi possano "chattare" solamente tra di loro (direttamente, senza l'apporto del server). Ad ogni iscrizione presso il server, il server controllerà se esistono client in attesa (dell'altro sesso): se esistono crea una coppia e li mette in grado di fare chat, altrimenti mette il client appena iscritto in una lista di attesa (coda FIFO), in attesa che qualche altro client si iscriva. Opzionalmente, si può gestire la situazione in cui un partecipante in una coppia esce (con il comando quit) e l'altro partecipante viene automaticamente inviato per una re-iscrizione al server che lo mette in grado di fare chat con un altro partecipante oppure lo mette in lista di attesa, a seconda della disponibilità.

La caratteristica di questo esercizio sta nel funzionamento in due fasi: prima viene una fase di iscrizione e di "accoppiamento", gestita dal server, dove i client non fanno altro che chiamare i metodi per la iscrizione e rimanere in attesa del partner; poi, viene una fase, strutturata in maniera peer2peer, dove i due client dialogano tra di loro, senza più l'intervento del server.

Quindi, cercheremo di suddividere i due problemi da affrontare:

- l'accoppiamento tra client di sesso diverso;
- la chat peer2peer tra due client.

7.2.2 Una soluzione

Iniziamo con la presentazione delle due interfacce. La prima interfaccia riguarda i servizi offerti dal server.

CuoriSolitariServer.java

```
1 import java.rmi.*;
2
```


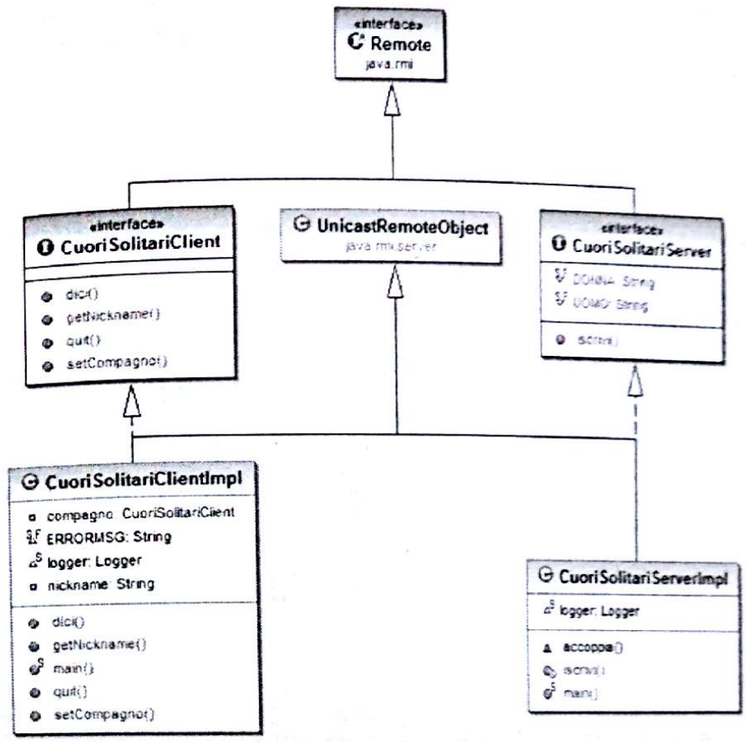

Figura 7.1: *Il diagramma UML delle classi della applicazione della chat Cuori Solitari che comprende, insieme, sia il lato server che quello client.*

```
3 public interface CuoriSolitariServer extends Remote {
4   public static final String UOMO = "uomo";
5   public static final String DONNA = "donna";
6   boolean iscrivi(String tipo, CuoriSolitariClient rif) throws RemoteException;
7 }
```

Fine: CuoriSolitariServer.java

Questa interfaccia semplicemente definisce (linea 6) un metodo per la iscrizione, passando come parametri il tipo di client (uomo/donna) ed il riferimento remoto ad un oggetto che implementi la interface `Client`. Il metodo restituisce un booleano che servirà semplicemente ad indicare al client se ha trovato subito un partner pronto, oppure se è in attesa. Si deve notare anche che è possibile definire delle costanti (linee 4-5) all'interno della interfaccia, costanti che in questo caso semplificano il lavoro di gestire il tipo di dati "sesso", che dovrebbe essere gestito con una classe apposita. Adesso esaminiamo la interfaccia lato client.

CuoriSolitariClient.java

```
1 import java.rmi.*;
2
3 public interface CuoriSolitariClient extends Remote {
4   void setCompagno(CuoriSolitariClient riferimento) throws RemoteException;
5   void dici(String cosa) throws RemoteException;
6   String getNickname() throws RemoteException;
7   void quit() throws RemoteException;
8 }
```

Fine: CuoriSolitariClient.java

In questa interfaccia vengono definiti alcuni metodi critici per il funzionamento, sia lato server, nella fase di accoppiamento, sia nella chat a due che segue. Innanzitutto, il metodo di callback più importante è quello di `setCompagno()` (linea 4) che permette al server di settare la variabile `compagno` del client in modo da contenere il riferimento remoto dell'oggetto client del partner selezionato dal server. Quindi il server, per eseguire un accoppiamento, assegnerà alla variabile `compagno` di uno il riferimento remoto dell'altro, e viceversa.

Il metodo `dici()` (linea 5) ha la particolarità che non viene chiamato dal server, ma sarà invocato solamente nella chat a 2 che seguirà, quindi dal partner selezionato. Da notare come, in questo caso, non serve il *wrapping* di una classe Messaggio, che contenga anche il mittente, in quanto colui che invia è sempre noto a priori (cioè solamente il compagno invoca questo metodo remoto).

Il metodo `quit()` (linea 7) serve a notificare ad un client la uscita del suo partner, resettando a `null` il suo riferimento al `compagno` e rendendolo quindi disponibile ad un nuovo round di accoppiamento con un nuovo client sul server.

Il funzionamento del server consiste principalmente nell'assicurare l'accoppiamento di client di sesso diverso. Non implementeremo in maniera specifica una coda, visto che non veniva richiesto dalla traccia, e utilizzeremo dei semplici `ArrayList` dal quale rimuoveremo l'elemento in testa.

CuoriSolitariServerImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.ArrayList;
4 import java.util.logging.Logger;
5
6 public class CuoriSolitariServerImpl extends UnicastRemoteObject
7                 implements CuoriSolitariServer {
```



```
8   private static final long serialVersionUID = 1L;
9   static Logger logger = Logger.getLogger("global");
10
11  protected CuoriSolitariServerImpl() throws RemoteException {
12    super();
13  }
14
15  public static void main(String[] args) {
16    System.setSecurityManager(new RMISecurityManager());
17    try {
18      CuoriSolitariServerImpl server = new CuoriSolitariServerImpl();
19      Naming.rebind("ServerClub", server);
20      System.out.println("Pronto per accettare nuovi iscritti!");
21    } catch (Exception e) {
22      logger.severe("Errori su creazione/rebind"+ e.getMessage());
23      e.printStackTrace();
24    }
25  }// end main
26
27  synchronized public boolean iscrivi(String tipo, CuoriSolitariClient rif)
28                          throws RemoteException {
29    logger.info ("Richiesta iscrizione: "+ rif.getNickname() + ", "+ tipo);
30    if (tipo.equals(UOMO)) {
31      logger.info ("Essendo un uomo lo aggiungo alla lista appropriata");
32      listaUomini.add(rif);
33      logger.info(" che ora contiene " + listaUomini.size() +" elementi");
34    } else if (tipo.equals(DONNA)) {
35      logger.info("Essendo una donna la aggiungo alla lista appropriata");
36      listaDonne.add(rif);
37      logger.info(" che ora contiene " + listaUomini.size() +" elementi");
38    } else
39      throw new RemoteException("Tipo partecipante non corretto");
40    return accoppia();
41  }
42
43  boolean accoppia() {
44    if (listaUomini.size()== 0 || listaDonne.size()==0)
45      return false;
46    CuoriSolitariClient uomo = (CuoriSolitariClient) listaUomini.remove(0);
47    CuoriSolitariClient donna = (CuoriSolitariClient) listaDonne.remove(0);
48    try {
49      uomo.setCompagno(donna);
50      donna.setCompagno(uomo);
51    } catch (RemoteException e) {
52      e.printStackTrace();
53    }
54    return true;
55  }
56
57  private ArrayList<CuoriSolitariClient> listaUomini = new ArrayList<CuoriSolitariClient>();
58  private ArrayList<CuoriSolitariClient> listaDonne = new ArrayList<CuoriSolitariClient>();
59 }
```

Fine: CuoriSolitariServerImpl.java

Il funzionamento principale della lista di attesa per i due tipi diversi di client (uomo/donna) consiste nell'accodare ogni nuovo client nella lista appropriata. Immediatamente dopo ogni inserimento, si provvede a verificare se è possibile o no l'accoppiamento, cioè se esistono due client in attesa di sesso diverso. Se, ad esempio, c'era una coda di un tipo piena di utenti in attesa, appena si iscrive un client del sesso opposto allora questi viene prima inserito nella propria coda e poi viene fatto il match per mettere in comuni-

cazione i due elementi in testa alla propria coda. Per fa comunicare due "cuori solitari" il server fa in modo di comunicare a ciascuno il riferimento remoto dell'altro e quindi a farne partire la chat. Se invece non esistono elementi che si possono accoppiare, non si fa nulla.

Dopo il solito costruttore vuoto (linee 11-13) si trova il `main()` (linee 15-25), che svolge il classico compito di istanziazione e registrazione dell'oggetto remoto (linee 18-19), dopo aver istanziato un Security Manager (linea 16).

Il metodo remoto `iscrivi()` (linee 27-41) è il fulcro della creazione delle coppie. In effetti, dopo aver aggiunto nella lista appropriata il client iscritto (linee 30-39) (da notare il controllo di correttezza nella linea 39), viene richiamato il metodo `accoppia()` (linea 40) che restituisce un booleano che viene a sua volta restituito dal metodo. Il metodo `accoppia()` (linee 43-55), restituisce `false` se una delle due liste è vuota (linee 44-45): non ci sono coppie che possono essere create. Invece, in caso sia possibile creare una coppia, alle linee 46-47 vengono eliminati dalla coda i due client uomo e donna, e vengono messi in comunicazione, passando all'uomo il riferimento remoto della donna e viceversa (linee 48-53) e, infine, si restituisce `true`.

Infine, alle linee 57-58, vengono definiti (ed inizializzati) due `ArrayList` che conterranno la lista di uomini e di donne. Da notare che il tipo di dati contenuto è di oggetti (remoti) che implementino la interface `Client`.

A questo punto, si può passare al client. Ricordiamo i suoi compiti principali. Una volta iscritto, il client deve offrire al suo utente una shell per interagire con l'utente. Questa shell, se il `compagno` non è stato ancora settato, permette solamente la esecuzione di comandi (che iniziano per il carattere `!`: le altre stringhe non possono essere inviate a nessuno! Questo tipo di soluzione rappresenta un'interessante uso della variabile `compagno` come flag di abilitazione: se avessimo una componente grafica (invece della primitiva shell di testo) potremmo usare questa variabile come segnale per poter fare la abilitazione o disabilitazione della componente.

CuoriSolitariClientImpl.java

```
 1 import java.io.*;
 2 import java.rmi.*;
 3 import java.rmi.server.*;
 4 import java.util.logging.Logger;
 5
 6 public class CuoriSolitariClientImpl extends UnicastRemoteObject
 7                    implements CuoriSolitariClient {
 8   private static final long serialVersionUID = 1L;
 9   static Logger logger = Logger.getLogger("global");
10
11   protected CuoriSolitariClientImpl() throws RemoteException {
12     super();
13   }
14
15   public static void main(String[] args) {
16     String cmd;
17     try {
18       client = new CuoriSolitariClientImpl();
19     } catch (RemoteException e1) {
20       logger.severe("Errore su creazione oggetto remoto"+ e1.getMessage());
21       e1.printStackTrace();
22     }
23     if (args.length < 2) {
24       System.out.println("Inserire: uomo/donna nick server(facoltativo)");
25       System.exit (1);
26     }
27     client.tipo = args[0];
28     client.nickname = args[1];
```



```
29     if (args.length > 2 ) client.host = args[2];
30     iscriviAlServer(client);
31     BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
32     try {
33       while (!(cmd = ask ("Comandi>", in)).equals("!quit")) {
34         if (cmd.startsWith("!")) {// esecuzione comandi
35           if (cmd.equals ("!status")) {
36             String s=ACCOPPIATO;
37             if (client.compagno == null)
38               s = NON_ACCOPPIATO;
39             System.out.println ("Status: " + s);
40           } else if (cmd.equals ("!sesso"))
41             System.out.println ("Sesso: "+client.tipo);
42           else if (cmd.equals ("!scappa")) {
43             try {
44               client.compagno.quit();
45               System.out.println("Mi riiscrivo");
46               client.setCompagno(null);
47               iscriviAlServer(client);
48             } catch (RemoteException e) {
49               logger.severe("Errori su invocazioni remote "+ e.getMessage());
50               e.printStackTrace();
51             }
52           }
53           else System.out.println (ERRORMSG);
54         } else { // stringhe da inviare
55           if (client.compagno != null ) {
56             try {
57               client.compagno.dici(cmd);
58             } catch (RemoteException e) {
59               logger.severe("Errore su invocazione remota "+ e.getMessage());
60               e.printStackTrace();
61             }
62           }
63         }
64       } // end while
65     } catch (IOException e) {
66       logger.severe("Errore su input da tastiera "+ e.getMessage());
67       e.printStackTrace();
68     }
69     System.out.println("Esco");
70     // posso digitare !quit anche se ero senza compagno
71     if (client.compagno !=null )
72       try {
73         client.compagno.quit();
74       } catch (RemoteException e2) {
75         logger.severe("Errore invocazione remota in uscita "+ e2.getMessage());
76         e2.printStackTrace();
77       }
78       System.exit(0);
79   }
80
81   public void setCompagno(CuoriSolitariClient riferimento) throws RemoteException {
82     compagno = riferimento;
83     if (compagno != null)
84       System.out.println("Ho trovato un compagno: "+ compagno.getNickname());
85     else
86       System.out.println("Sono solo!");
87   }
88
89   public void dici(String cosa) throws RemoteException {
```

```
90      System.out.println ("["+compagno.getNickname()+"]:"+cosa+"\nComandi>");
91    }
92
93
94    private static void iscriviAlServer(CuoriSolitariClientImpl c) {
95      try {
96        CuoriSolitariServer server =
97          (CuoriSolitariServer) Naming.lookup("rmi://"+c.host+"/ServerClub");
98        System.out.println("Ho la reference del server");
99        if (server.iscrivi (c.tipo, c))
100         System.out.println ("Che fortuna, c'era qualcuno che mi aspettava!");
101       else
102         System.out.println ("Che sfortuna, beh, aspettero'!");
103     } catch (Exception e) {
104       logger.severe("Errore in registrazione sul server"+ e.getMessage());
105       e.printStackTrace();
106     }
107   }
108
109   private static String ask(String prompt, BufferedReader in) throws IOException {
110     System.out.print(prompt+" ");
111     return (in.readLine());
112   }
113
114   public String getNickname() throws RemoteException {
115     return nickname;
116   }
117
118   public void quit() throws RemoteException {
119     setCompagno(null);
120     System.out.println ("Se ne è andato...Ci rimettiamo in fila!");
121     iscriviAlServer(client);
122   }
123
124   private String host="localhost";
125   private CuoriSolitariClient compagno = null;
126   private String nickname = null;
127   private String tipo = null;
128   private static CuoriSolitariClientImpl client = null;
129   final static String ERRORMSG = "Comando non riconosciuto";
130   private static final String ACCOPPIATO = "accoppiato";
131   private static final String NON_ACCOPPIATO = "non accoppiato";
132 }
```

Fine: CuoriSolitariClientImpl.java

Dopo il costruttore (linee 11-13), si trova il metodo `main()` (linee 15-79). Dopo aver istanziato il client (linea 18) e prelevato i parametri da linea di comando (linee 23-29) (tipo (uomo/donna) e nickname obbligatori, host del server facoltativo), effettua una chiamata (linea 30) al metodo `iscriviAlServer()` definito alle linee 94-107 che svolge un semplice compito: invocare il metodo remoto `iscrivi()` sul server (linea 99) dopo averne cercato il riferimento (linea 97). Il booleano restituito dal metodo `iscrivi()` ha come solo effetto quello di stampare messaggi appropriati a video (linea 100 e 102).

Torniamo al main, alla linea 33, dove parte la shell di comandi. Da questa shell si esce se si digita `!quit`, altrimenti si rimane nel while alle linee 33-64. L' `if.. else` alle linee 34-54 e 55-62 distingue tra la esecuzione dei comandi (che iniziano con `!`) e le stringhe da inviare al compagno (se presente). I comandi che vengono riconosciuti sono alcune semplici query (`!status` (linee 35-40) che restituisce informazioni se si è accoppiati oppure no e `!sesso` (linee 40-42) che restituisce il proprio tipo) ed il comando `!scappa`

che finge una uscita da parte del client[1], invocando il metodo `quit()` sul compagno e reiscrivendosi subito dopo sul server (linee 42-52). Qualsiasi altro comando che inizia con `!` viene trattato come un errore (linea 53).

Le stringhe che non sono considerate comandi vengono inviate al compagno, se presente (con il controllo alla linea 55). In questa maniera, come già anticipato, il valore non nullo della variabile compagno funge da flag per la abilitazione della shell testuale e potrebbe essere utilizzata (in una interfaccia grafica) per abilitare una componente.

Dal ciclo di while (linee 33-64) si esce se l'utente digita `!quit`. In questo caso, le operazioni da effettuare sono quella di informare il proprio compagno (se esiste, linea 71) della propria uscita invocandone il metodo `quit()` (linea 73) e poi di uscire (linea 78).

Il metodo `setCompagno()`, alle linee 81-87, non fa altro che assegnare alla variabile `compagno` il riferimento passato (e stampare a video alcune informazioni, tra cui il nickname del compagno appena trovato, linea 84). Il metodo `dici()` stampa la stringa passata, usando il nickname del compagno come mittente.

Oltre a metodi semplici e di servizio (`ask()` e `getNickname()`), si trova il metodo `quit()` alle linee 118-122, dove, dopo aver messo a `null` il riferimento del compagno (linea 119), si rifà la propria iscrizione al server (linea 121), motivo per il quale la operazione di iscrizione al server è stata rifattorizzata all'esterno e resa un metodo.

In conclusione, alle linee 124-131 vengono dichiarate alcune variabili dal nome e utilizzo auto-esplicativo. Tra le più importanti per le funzionalità, il riferimento remoto al `compagno` (linea 125), ed alcune stringhe da stampare (linee 26-30).

7.3 Una chat CS per un docente dispotico

Questo esercizio prevede una chat che simula una classe in cui il "docente" (sul server) è alquanto dispotico (ovviamente è solamente un esempio) ed è l'unico a poter parlare, tranne, al massimo, uno studente che viene abilitato dal docente dopo aver richiesto la parola (metaforicamente "alzando la mano"): in questo caso, il client selezionato può parlare a tutti. Inoltre il docente può terminare l'intervento di uno studente (levandogli la parola) o addirittura cacciarlo via dalla classe, cioé forzarne l'uscita dalla chat.

Il testo dell'esercizio

Progettare e realizzare un programma distribuito in Java (che usi Remote Method Invocation) che realizzi una classe virtuale con architettura client-server. La Classe deve offrire la possibilità ad un docente (il server) di poter comunicare con tutti gli studenti (i client) connessi in broadcast. I client (di norma) possono solamente vedere quello che scrive il docente mentre quello che scrivono loro non viene inviato in broadcast; i client possono, però, "alzare la mano" per fare una domanda, cioè richiedere di poter parlare digitando il comando "!domanda". Il docente vede sul suo schermo il nome degli studenti che "alzano la mano" per parlare e può decidere di dare la parola ad uno di loro (oppure a qualcun altro) digitando il comando "!parla" seguito dal nickname dello studente. Quando uno studente S ha la parola, gli altri studenti vengono avvisati e quello che scrive S viene inviato in broadcast a tutti. Il docente può comunque intervenire e la discussione tra docente e lo studente selezionato viene comunque inviata a tutti gli studenti (facendo in modo di distinguere da chi proviene ogni linea scritta). Comunque, il

[1] Non era richiesto dalla traccia, ma era alquanto semplice da implementare e quindi viene inserito nella soluzione all'esercizio.

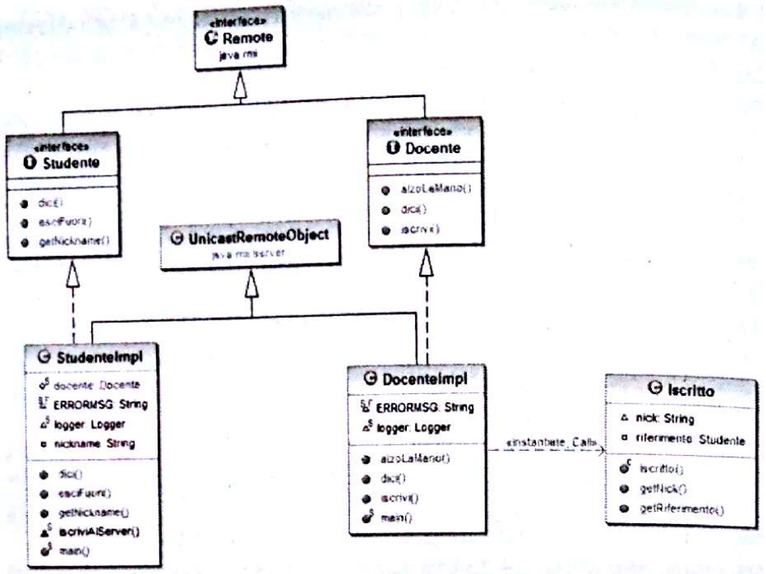

Figura 7.2: *Il diagramma UML delle classi della applicazione della chat con un Docente Dispotico che comprende, insieme, sia il lato server che quello client.*

docente può terminare l'intervento dello studente con il comando "!riprendiamo". Opzionalmente, si può implementare anche che un docente possa "cacciare via" uno studente dalla classe, digitando "!caccia" seguito dal nickname dello studente: in tal caso il server (docente) chiama un metodo remoto del client (studente) selezionato che termini il programma client.

7.3.1 Un confronto tra differenti strategie

Questo esercizio è un classico esempio che permette la soluzione sia lato server che lato client. Esaminiamo ciascuna delle due tipologie di soluzione e poi le confrontiamo.

Iniziamo con il presentare la soluzione lato server. Il meccanismo che permette di bloccare i contributi degli studenti consiste nel memorizzare sul server, la macchina docente, il riferimento remoto dell'unico client (studente) che è in grado di effettuare contributi. Questa variabile verrà aggiornata quando il docente userà il comando `!parla`. Ogni client, quindi, invoca il metodo `dici()` fornendo, insieme alla stringa da inviare, anche il proprio riferimento remoto[2]. Il server controlla, per ogni contributo, se il client è quello (unico) autorizzato (confrontandolo con il riferimento remoto che ha memorizzato) e, in tal caso, invia a tutti il contributo. Nel caso contrario, può ignorare la invocazione (o inviare singolarmente allo studente un messaggio di errore). Per ripristinare la situazione in cui nessuno parla, basterà ripristinare a `null` il valore di tale variabile.

La soluzione lato client, invece, prevede che ogni client memorizzi una propria variabile (booleana) che indica se è abilitato a parlare oppure no. Il server effettuerà la abilitazione/disabilitazione cambiando (con un appropriato metodo remoto esposto da ogni client) il valore di tale variabile.

[2]In effetti potrebbe passare anche solamente il proprio nickname, però questo permetterebbe una semplice maniera di falsificare interventi da parte di uno studente, che modifica il proprio client, inserendo un nickname arbitrario.

Le due soluzioni sono entrambe appropriate per gli scopi dell'esame (questo significa che vengono valutate in maniera assolutamente equivalente) ma hanno caratteristiche diverse:

- La soluzione lato server mantiene un certo controllo sul meccanismo. Ad esempio, non è possibile che un client falsifichi (in maniera semplice) un riferimento remoto altrui, e non è possibile (anche cambiando il codice del client) cercare di parlare se non si ha diritto.

- La soluzione lato client, invece, offre la possibilità ad un utente malizioso di scrivere un client modificato che non modifichi la propria variabile booleana (o semplicemente ignorandone il valore quando si deve spedire il messaggio) e che quindi sia capace di scrivere sempre, indipendentemente dal controllo del server. Un vantaggio rispetto alla soluzione lato server, invece, consiste nel fatto che in caso un utente non possa scrivere, non viene proprio effettuata la richiesta remota al server, che, invece, nel caso della soluzione lato server viene sempre effettuata.

7.3.2 Una soluzione

Presentiamo una soluzione che implementa lato server il meccanismo di blocco/sblocco di un utente. Iniziamo con il presentare le interfacce sia per il docente che per lo studente. La interfaccia per il docente è la seguente:

Docente.java

```
1 import java.rmi.*;
2
3 public interface Docente extends Remote {
4   void iscrivi(Studente rif, String nick) throws RemoteException;
5   void dici(String cosa, Studente rif) throws RemoteException;
6   void alzoLaMano(Studente rif) throws RemoteException;
7 }
```

Fine: Docente.java

Come si può notare, il classico metodo di iscrizione `iscrivi()` (linea 4) permette di passare sia il proprio riferimento remoto (di tipo `Studente`) che il nick. Alla linea 5 viene definito il classico metodo `dici()` funzionale per la chat, mentre alla linea 6 viene definito il metodo `alzaLaMano()` che usa uno studente per segnalare al docente che vorrebbe parlare.

Adesso vediamo la interfaccia per lo studente.

Studente.java

```
1 import java.rmi.*;
2
3 public interface Studente extends Remote {
4   void dici(String cosa) throws RemoteException;
5   String getNickname () throws RemoteException;
6   void esciFuori() throws RemoteException;
7 }
```

Fine: Studente.java

Come al solito, nel metodo `dici()` alla linea 4 viene definito il metodo che il server richiama per la chat. Oltre al solito metodo di servizio per la richiesta del nickname (li-

nea 5), è presente il metodo `esciFuori()` che gestirà la richiesta da parte del server (il docente) di terminare ed uscire fuori dalla classe.

Prima di passare al docente, vediamo velocemente la definizione di un bean per racchiudere le informazioni (riferimento remoto e nickname) di un iscritto alla classe, che non necessita di particolari spiegazioni.

Iscritto.java

```
1 public class Iscritto {
2   public Iscritto(Studente rif, String nick) {
3     setRiferimento(rif);
4     setNick (nick);
5   }
6
7   public Studente getRiferimento() {
8     return riferimento;
9   }
10
11  private void setRiferimento(Studente riferimento) {
12    this.riferimento = riferimento;
13  }
14
15  public String getNick() {
16    return nick;
17  }
18
19  private void setNick(String nick) {
20    this.nick = nick;
21  }
22
23  private Studente riferimento = null;
24  String nick = "non conosciuto";
25 }
```

Fine: Iscritto.java

Ora presentiamo la implementazione `DocenteImpl` della interface `Docente`.

DocenteImpl.java

```
1 import java.io.*;
2 import java.rmi.*;
3 import java.rmi.server.*;
4 import java.util.ArrayList;
5 import java.util.logging.Logger;
6
7 public class DocenteImpl extends UnicastRemoteObject implements Docente {
8   private static final long serialVersionUID = 1L;
9   static Logger logger = Logger.getLogger("global");
10
11  protected DocenteImpl() throws RemoteException {
12    super();
13  }
14
15  public void iscrivi(Studente rif, String nick) throws RemoteException {
16    Iscritto i = new Iscritto(rif, nick);
17    iscritti.add(i);
18    logger.info("Si e' iscritto "+nick);
19  }
20
21  public void dici(String cosa, Studente rifRem) throws RemoteException {
22    if (rifRem.equals(studenteCheParla))  {
23      String s ="["+rifRem.getNickname()+"]:"+cosa ;
```



```
24      diciATutti(s);
25      System.out.println(s);
26    } else {
27      logger.info("Sta cercando di parlare "+rifRem.getNickname());
28      rifRem.dici("Non puoi parlare! Silenzio!");
29    }
30  }
31
32  public void alzoLaMano(Studente rif) throws RemoteException{
33    logger.info (rif.getNickname()+" alza la mano...");
34  }
35
36  private void diciATutti (String cosa) {
37    Iscritto tmp;
38    for (int i = 0; i< iscritti.size(); i++) {
39      tmp = iscritti.get(i);
40      try {
41        ((Studente) tmp.getRiferimento()).dici(cosa);
42      } catch (RemoteException e) {
43        logger.info ("Sembra che "+tmp.getNick()+" se ne sia andato.");
44        iscritti.remove(i);
45        logger.info ("Lo cancelliamo");
46      }
47    }
48  }
49
50  public static void main(String[] args) {
51    String cmd;
52    DocenteImpl docente = null;
53    System.setSecurityManager(new RMISecurityManager());
54    try {
55      docente = new DocenteImpl();
56      Naming.rebind("Docente", docente);
57      System.out.println("Pronto per accettare studenti!");
58    } catch (Exception e) {
59      logger.severe("Errori su creazione/rebind"+ e.getMessage());
60      e.printStackTrace();
61    }
62    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
63    try {
64      while (!(cmd = ask ("Comandi>", in)).equals("!quit")) {
65        if (cmd.startsWith("!")) {// esecuzione comandi
66          if (cmd.equals ("!parla")) {
67            String nome = ask ("Nome?", in);
68            Iscritto tmp=null;
69            int i;
70            for (i= 0; i < docente.iscritti.size(); i++) {
71              tmp = docente.iscritti.get(i);
72              if (tmp.getNick().equals(nome)) break;
73            }
74            if (i != docente.iscritti.size()) {// trovato
75              String s = "Parla "+tmp.getNick();
76              System.out.println (s);
77              docente.diciATutti (s);
78              docente.studenteCheParla = (Studente) tmp.getRiferimento();
79            } else // non trovato
80              System.out.println ("non trovo il nick");
81          } else if (cmd.equals ("!riprendiamo")) {
82            docente.studenteCheParla.dici("Scusa ma dobbiamo riprendere.");
83            docente.studenteCheParla = null;
84            System.out.println ("Riprendiamo...");
```

```
85          docente.diciATutti ("Riprendiamo...");
86        } else if (cmd.equals ("!caccia")) {
87          String nome = ask ("Nome da cacciare?", in);
88          Iscritto tmp=null;
89          int i;
90          for (i= 0; i < docente.iscritti.size(); i++) {
91            tmp = docente.iscritti.get(i);
92            if (tmp.getNick().equals(nome)) break;
93          }
94          if (i != docente.iscritti.size()) {// trovato
95            System.out.println ("lo caccio fuori!");
96            docente.iscritti.remove(i);
97            try {
98              ((Studente) tmp.getRiferimento()).esciFuori();
99            } catch (RemoteException e) {//la connessione viene resettata
100             System.out.println ("Fatto!");
101           }
102         } else // non trovato
103           System.out.println ("non trovo il nick");
104       }
105       else System.out.println (ERRORMSG);
106     } else { // stringhe da inviare
107       docente.diciATutti (cmd);
108     }
109   } // end while
110 } catch (Exception e) {
111   e.printStackTrace();
112 }
113 System.out.println("Esco");
114 System.exit(0);
115 }
116
117 private static String ask(String prompt, BufferedReader in) throws IOException {
118   System.out.print(prompt+" ");
119   return (in.readLine());
120 }
121
122 final static String ERRORMSG = "Comando non riconosciuto";
123 private ArrayList<Iscritto> iscritti = new ArrayList<Iscritto>();
124 private Studente studenteCheParla = null;
125 }
```

Fine: DocenteImpl.java

Iniziamo con il descrivere le variabili istanza più importante (che si trovano alla fine del file): alla linea 123 dichiariamo la lista di iscritti (istanze di `Iscritto`) dove il docente metterà tutti gli studenti iscritti. Di particolare importanza è la variabile `studenteCheParla` (linea 124), che contiene il riferimento remoto dello studente che è autorizzato a parlare, e che viene inizializzata a `null` che indica che nessuno è autorizzato.

Dopo il costruttore vuoto (linee 11-13), la implementazione dei metodi remoti prevede il metodo `iscrivi()` (linee 15-19) che prevede l'inserimento nella lista degli iscritti del bean costruito con il riferimento ed il nickname passati al metodo remoto dal client che si è appena iscritto.

Il metodo `dici()` (linee 21-30) è diverso dal classico metodo di una chat semplicemente perchè effettua il controllo sul riferimento remoto di chi è iscritto con l'if alla linea 22. Sottolineiamo, in velocità, che l'uguaglianza si controlla con il metodo `equals()` e non con l'operatore `==` che controlla esclusivamente se entrambe le variabili hanno come riferimento lo stesso oggetto e non (come serve a noi) che le due variabili facciano riferimenti a due oggetti che sono semanticamente uguali. Se è lo studente autorizzato a parlare, allora viene inviata a tutti (invocazione del metodo `diciATutti()` della linea 24) la stringa passata come parametro. Altrimenti (linee 27-28) il client non autorizzato viene avvertito (singolarmente, attraverso la callback `dici()` invocata in linea 28) che non può parlare.

Il metodo per segnalare che si vuole parlare (linee 32-34) stampa semplicemente a video una stringa per avvisare il docente.

Il metodo `diciATutti()` alle linee 36-48, scorre semplicemente la lista di iscritti per inviare la stringa. L'unica particolarità è il trattamento della eccezione remota che non fa altro che rimuovere il client dalla lista.

A questo punto, nel file, alla linea 50 inizia il `main()`. Dopo aver installato il Security Manager (linea 53), si istanzia e fa il rebind dell'oggetto remoto sul server (linee 55-56). La shell di interazione con l'utente inizia alla linea 64. I comandi vengono gestiti dall'if a linea 65. Il comando per assegnare ad uno studente il diritto di parlare viene gestito nelle linee 66-81. Dopo aver chiesto il nome dello studente a cui si vuole dare la parola, viene ricercato nella lista (linea 70-73), se si è usciti dal for perchè si è eseguito il break, allora il valore della variabile `i` è inferiore alla dimensione della lista di iscritti, e quindi è stato trovato il nick. Si avvisano tutti gli altri iscritti che il diritto passa allo studente (linea 77) e (importante!) si memorizza il suo riferimento (linea 78) nella variabile `studenteCheParla`. Se non lo si è trovato si stampa un messaggio (linea 80).

La shell permette anche di togliere la parola allo studente che sta parlando, attraverso il comando `!riprendiamo` gestito alle linee 81-86, che semplicemente mette a `null` il riferimento di `studenteCheParla`. Infine, tra i comandi, c'è la esecuzione di `!caccia` che permette di "allontanare fuori dalla classe" uno dei client (linee 86-104). Viene chiesto il nome dello studente e cercato nella lista di iscritti (linee 87-93) e poi viene eliminato dalla lista degli iscritti (linea 96) e ne viene chiamato il metodo remoto `esciFuori()` linea 98. Tutte le altre stringhe che iniziano per "!" sono considerati comandi non corretti (linea 105). Il docente dispotico (in quanto tale!) ha sempre diritto di parola e tutte le altre stringhe che digita (che non siano comandi) vengono inviate a tutti (linea 107).

A questo punto, possiamo descrivere il client dello studente.

StudenteImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.logging.Logger;
4 import java.io.*;
5
6 public class StudenteImpl extends UnicastRemoteObject implements Studente {
7   private static final long serialVersionUID = 1L;
8   static Logger logger = Logger.getLogger("global");
9
10  protected StudenteImpl() throws RemoteException {
11    super();
12  }
13
14  public static void main(String[] args) {
15    String cmd;
16    try {
17      studente = new StudenteImpl();
18    } catch (RemoteException e) {
19      logger.severe ("Errore nella creazione oggetto remoto "+e.getMessage());
20      e.printStackTrace();
21    }
22    if (args.length < 1) {
23      System.out.println("Inserire nick e l'indirizzo del server");
24      System.exit (1);
25    }
```

```
26      studente.nickname = args[0];
27      if (args.length > 1 ) studente.host = args[1];
28      iscriviAlServer(studente);
29      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
30      try {
31        while (!(cmd = ask ("Comandi>", in)).equals("!quit")) {
32          if (cmd.startsWith("!")) {// esecuzione comandi
33            if (cmd.equals ("!domanda")) {
34              logger.info ("Alzo la mano...");
35              docente.alzoLaMano(studente);
36            } else System.out.println (ERRORMSG);
37          } else { // stringhe da inviare
38            docente.dici(cmd, studente);
39          }
40        } // end while
41      } catch (Exception e) {
42        logger.severe ("Errore in invocazione metodi remoti "+e.getMessage());
43        e.printStackTrace();
44      }
45      System.out.println("Esco");
46      System.exit(0);
47    }// fine main
48
49    public void dici(String cosa) throws RemoteException {
50      System.out.println (cosa);
51    }
52
53    public String getNickname() throws RemoteException {
54      return nickname;
55    }
56
57    public void esciFuori() throws RemoteException {
58      System.out.println ("Sono stato buttato fuori!");
59      System.exit(1);
60    }
61
62    private static String ask(String prompt, BufferedReader in) throws IOException {
63      System.out.print(prompt+" ");
64      return (in.readLine());
65    }
66
67    static void iscriviAlServer(StudenteImpl stud) {
68      try {
69        docente = (Docente) Naming.lookup("rmi://"+stud.host+"/Docente");
70        logger.info("Ho la reference del docente e mi iscrivo");
71        docente.iscrivi (stud, stud.nickname);
72      } catch (Exception e) {
73        logger.severe ("Errore in registrazione sul server "+e.getMessage());
74        e.printStackTrace();
75      }
76    }
77
78    private String nickname;
79    private static StudenteImpl studente;
80    public static Docente docente;
81    private String host = "localhost";
82    final static String ERRORMSG = "Comando non riconosciuto";
83  }
```

Fine: StudenteImpl.java

La implementazione dello Studente è abbastanza semplice e simile al client della chat.

Nel `main()`, dopo aver istanziato l'oggetto remoto (linea 17), viene effettuata (linee 27-28) la iscrizione al server fornito su linea di comando (default `localhost`, vedi definizione alla linea 81) invocando il metodo `iscriviAlServer()` alle linee 67-76). La shell di comandi (che inizia alla linea 31) serve solamente a distinguere tra la domanda da effettuare (linee 32-36) e le stringhe da inviare al docente. Ricordiamo che sarà il server a controllare se lo studente è abilitato a parlare oppure no.

La implementazione dei tre metodi remoti si trova alle linee 49-60 e l'unica particolarità è la uscita "brusca" causata dalla invocazione da parte del server del metodo `esciFuori()` alle linee 57-60.

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[E**]** Progettare e realizzare un programma distribuito in Java (che usi Remote Method Invocation) con architettura client-server che implementi un gioco di alto-basso con una chat. È una classica chat dove è possibile scambiarsi messaggi tra i client. Il server, prima di far partire il gioco (da linea di comando o da input) sceglie un intero positivo. Gli utenti devono provare a indovinare quale è il numero del server, ed hanno tre tentativi a disposizione. Ogni volta che un utente vuole provare a indovinare, digita "*!prova*" sulla shell, viene richiesto l'intero da inviare e lo sottopone al server, che risponde (a tutti quanti) dicendo se ha indovinato (e in questo caso il gioco termina) oppure se il numero che ha tentato l'utente è più basso o più alto di quello da indovinare. Ogni utente può continuare a chattare normalmente e riceve, oltre i messaggi dagli altri utenti, anche le risposte che il server invia a chi prova ad indovinare (in questa maniera l'utente può aumentare le possibilità di vincita). Il controllo che l'utente non faccia più di 3 prove deve essere effettuato dal server e non dal client. Inoltre, si può chiedere (una volta sola per client, anche questo controllato lato server) un "aiutino" personale: se il client digita "*!aiutino*", allora il server invia solamente a quel client la informazione di chi (quale client) si è avvicinato di più al numero da indovinare e di quanto si è avvicinato (ad esempio, "`Si è avvicinato di più Giovanni.. di 4`" se il numero da indovinare è 130 e Giovanni in passato ha provato a indovinare con 126).

2. **[E**]** Progettare e realizzare un programma distribuito in Java (che usi Remote Method Invocation) con architettura client-server che implementi un gioco di indovinelli a punti che funzioni, comunque anche da chat, cioè in ogni momento tutti possono usare una shell standard per comunicare con tutti gli altri. Il primo utente (client) che entra è detto "speaker" in quanto pone "gli indovinelli" agli altri. Lo speaker X comunica gli indovinelli al server digitando sulla shell una stringa del tipo "`?domanda:risposta`". Il server porrà la domanda facendo comparire sullo schermo un messaggio del tipo "`X chiede: domanda ?`" ed il primo che risponde con "*risposta*" al server vince un punto ed il server comunica a tutti chi ha indovinato. Quindi ad esempio X digita "`?10+2:12`", tutti gli altri vedono "`X chiede: 10+2?`" ed il primo che risponde "12" vince il punto. Ovviamente la domanda può essere anche non aritmetica, tipo "`?Chi ha vinto il mondiale di calcio nel 1970:Brasile`". È importante notare che durante gli indovinelli è possibile chattare normalmente: l'unica cosa che caratterizza una domanda è il fatto che proviene dallo speaker e che incomincia per "?".

3. **[E**]** Progettare e realizzare un programma distribuito in Java (che usi Remote Method Invocation) con architettura client-server che implementi un sistema di discussione con votazione. È una classica chat, alla quale partecipa anche il server, dove è possibile scambiarsi messaggi. L'utente del server può lanciare una votazione, digitando il comando "`!voto`" (e facendo seguire la domanda del voto) ed a tutti i client compare il tema del voto (ad esempio, "`L'Italia ha giocato bene all'ultimo mondiale?`"). Da questo momento, è possibile (oltre al classico scambio di messaggi nella chat) che i client possano digitare "`!si`" per votare a favore oppure

"`!no`" per votare contro. Senza l'intervento dell'utente, il server tiene "silenziosamente" conto dei voti a favore o contro. Quando tutti i presenti hanno votato, allora annuncia automaticamente i risultati e la votazione termina. Per semplicità, si può assumere che dal momento dell'inizio del voto nessun client entri o esca. Si deve controllare che (a) prima di attivare una votazione, sia terminata quella attiva; (b) che un utente non voti più di una volta; (c) che l'utente non può votare se non è attiva una votazione.

Nella migliore tradizione politica, si deve anche implementare il meccanismo di influenza "sottobanco" del voto: a tale scopo, ogni utente client può cercare di convincere altri client a votare a favore o contro, e per poter comunicare "discretamente" può usare un comando "`!sussurra`" che, inserito nickname e messaggio, fa recapitare un messaggio privato, tramite il server, esclusivamente al destinatario. Attenzione: tutte le operazioni di lancio della votazione, di voto e di "sussurro" devono comunque permettere la chat classica, cioè ogni stringa digitata dall'utente (che non inizia per "!") va inviata a tutti.

Argomenti Avanzati su Java RMI

Capitolo 8

Gestione dinamica di RMI

Indice

8.1 Introduzione **174**

8.2 Caricamento dinamico delle classi **174**

8.2.1 Il ClassLoader 174

8.2.2 Caratteristiche del caricamento dinamico 175

8.2.3 Gli scenari di utilizzo 175

8.3 Stub e Skeleton: la evoluzione **183**

8.3.1 Stub e Skeleton versione JDK 1.1 183

8.3.2 Stub versione JDK 1.2 187

8.3.3 JDK 5: niente Stub e Skeleton! 189

Note bibliografiche **193**

Spunti per lo studio individuale **194**

8.1 Introduzione

Dopo aver trattato delle motivazioni di Java RMI, della sua architettura e delle sua funzionalità (con alcuni semplici esempi), è il caso di approfondire, in questo capitolo, alcuni argomenti di natura più avanzata sul suo funzionamento. In particolare modo, spiegheremo due meccanismi che permettono a RMI di reagire in maniera dinamica (a run-time) durante le invocazioni remote.

Tratteremo, innanzitutto, della possibilità di far caricare alla Java Virtual Machine classi dalla rete a run-time, in modo da permettere una agevole evoluzione di applicazioni distribuite, in quanto non è necessario distribuire con client/server la definizione di classi che possano essere modificate/aggiunte alla applicazione, ma si può fare in modo che client/server carichino dinamicamente ciò di cui hanno bisogno.

Poi, parleremo della architettura di Java RMI, esaminando come il layer Stub & Skeleton sia stato modificato nelle diverse versioni, portando alcuni dei compiti che esso svolgeva all'interno del Reference Layer e del Transport Layer, con una continua evoluzione verso un deployment via via più semplificato. Per fare questo, introdurremo i proxy dinamici in Java per poter spiegare il loro ruolo nella generazione dinamica degli stub.

8.2 Caricamento dinamico delle classi

La piattaforma Java offre la possibilità di poter caricare dinamicamente le classi da utilizzare, siano esse localizzate su disco o sulla rete. Nel paragrafo 3.4.3 abbiamo presentato la architettura di questo meccanismo ed adesso ne approfondiamo i dettagli.

Questa caratteristica, particolarmente enfatizzata durante la presentazione del linguaggio nell'ormai lontano 1995, effettivamente rappresenta un punto di forza notevole: una applicazione distribuita Java risulta essere (quasi automaticamente) espandibile, in quanto è la JVM stessa che all'atto della esecuzione si preoccupa di reperire, caricare ed utilizzare classi che non erano disponibili al momento della "installazione" della applicazione distribuita.

Pensiamo, per esempio, alla necessità di dover facilmente aggiornare il comportamento di una applicazione, permettendo di aggiungere delle funzionalità ad una classe, e facendo in modo che tutti i client (che potrebbero essere davvero in grande numero) possano automaticamente reperire (a run-time) la classe necessaria.

Tutto questo viene realizzato attraverso l'utilizzo da parte della Java Virtual Machine del `ClassLoader`, un meccanismo del Java Runtime Environment per caricare componenti software a run-time nella Java Virtual Machine.

8.2.1 Il ClassLoader

In Java, come si sa, la unità della distribuzione di software è fissata alla *classe*, che, scritta in una rappresentazione indipendente dalla architettura, viene eseguita dalla Java Virtual Machine. Le caratteristiche di portabilità e di indipendenza dalla architettura del linguaggio Java sono rese ancora più importanti dalla possibilità di poter caricare dinamicamente le classi da qualsiasi sorgente, sia essa la unità di memoria di massa oppure (meno tradizionalmente) dalla rete. Questo viene effettuato dai ClassLoader, oggetti veri e propri, istanze della classe `ClassLoader` che hanno come compito quello di "tradurre" una classe (definita con il nome) in una sua implementazione (espressa in termini di bytecode).

La Java Virtual Machine ha un ClassLoader di *bootstrap*, che serve a fare riferimento alla librerie Java di base del linguaggio, un ClassLoader per le estensioni (localizzate

nella directory <JAVA_HOME>/lib) ed un ClassLoader di sistema, che fa riferimento alla variabile di ambiente `CLASSPATH`.

8.2.2 Caratteristiche del caricamento dinamico

Risulta particolarmente importante analizzare le caratteristiche del caricamento dinamico delle classi introdotte da Java. Questo meccanismo, infatti, rappresenta una parte significativa del successo della piattaforma Java.

Le caratteristiche più importanti del caricamento dinamico delle classi della piattaforma Java sono:

Lazy Loading: le classi vengono caricate dalla JVM esclusivamente su richiesta, quindi facendo risparmiare al sistema memoria (per memorizzare il bytecode), banda sulla rete (qualora le classi da caricare si trovino in rete) e tempo di CPU (nessuna risorsa di calcolo viene impiegata per classi che non vengono utilizzate).

Type-safety: il caricamento dinamico preserva il controllo sulla integrità dei tipi sfruttando il meccanismo della ereditarietà della programmazione ad oggetti.

Politica di caricamento definibile dall'utente: i class-loader sono oggetti che possono essere istanziati dal programmatore, con il comportamento specifico come richiesto dalle esigenze, in modo da offrire al programmatore il completo controllo. In questa maniera, ad esempio, si possono determinare specifiche locazioni di caricamento, oppure particolari politiche di sicurezza da applicare.

Namespace multipli: poter usare class loaders diversi contemporaneamente permette di usare namespace differenti per componenti diverse all'interno della stessa applicazione. Questo permette una notevole flessibilità nella progettazione, garantendo una specifica politica di accesso e di sicurezza per ciascuno namespace.

Sicurezza: attraverso la politica di sicurezza della JVM (vedi il Capitolo 3.5.3).

8.2.3 Gli scenari di utilizzo

Il caricamento dinamico delle classi può avvenire in tre scenari distinti:

1. Caricamento dello stub, da parte del client.
2. Caricamento di una classe da parte del server, in quanto viene passato come parametro ad una invocazione remota un oggetto istanza di una classe che non è a disposizione del server (nei suoi ClassLoader).
3. Caricamento di una classe da parte del client, in quanto viene restituito dal server, da una invocazione remota, un risultato che è istanza di una classe che non è a disposizione del client (nei suoi ClassLoader).

Il primo caso è attualmente meno utilizzato, vista la possibilità di usare il modello di generazione automatica dello stub da parte delle recenti versioni di Java. I due casi successivi, invece, rappresentano una maniera importante per permettere ad una applicazione distribuita di evolvere in maniera sostenibile, al sorgere di nuove necessità e requisiti.

Nei prossimi paragrafi, esamineremo in dettaglio ciascuno dei tre scenari, cercando di comprendere il meccanismo mediante il quale viene effettivamente trasmessa la informazione della locazione della classe da caricare, dal server al client o viceversa, a seconda dello scenario.

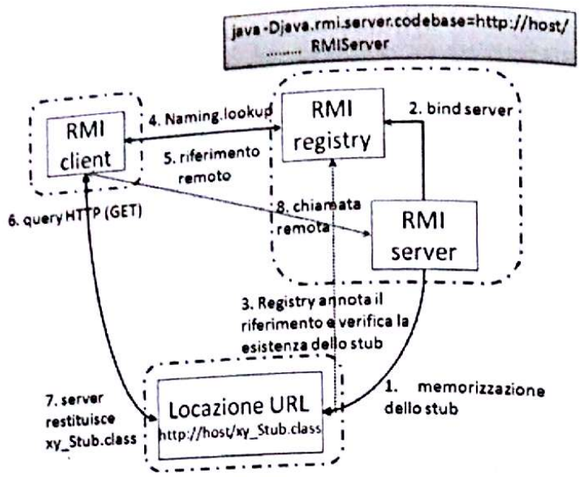

Figura 8.1: *Lo schema del caricamento dinamico dello stub da parte del client.*

Il caricamento dinamico dello stub

In questo scenario, attualmente poco utilizzato, la informazione sulla posizione della classe che definisce lo stub si trova sul lato server e deve essere passata al client. La modalità è quella di utilizzare il registry come posizione dove il server memorizza la URL della posizione della classe e dove il client può accedere, quando fa lookup, ottenendo il riferimento remoto, annotato con la URL del server HTTP che contiene la classe.

Le operazioni effettuate per il caricamento dinamico dello stub vengono riassunte nella Figura 8.1.

1. Lo stub, generato con `rmic`, viene memorizzato sul server HTTP.
2. Il server (lanciato in modo da indicare in `java.rmi.server.codebase` il codebase dal quale scaricare lo stub) effettua il bind sul registry dell'oggetto remoto.
3. Il registry annota la definizione della URL per la classe dell'oggetto appena registrato e verifica la disponibilità della classe sul server HTTP.
4. Il client effettua il lookup sul registry ...
5. ...ed ottiene un riferimento remoto che è annotato con la locazione dello stub, con la URL indicata dal server.
6. Il client effettua la richiesta dello stub al server HTTP con una richiesta GET ...
7. ...ed ottiene lo stub.
8. Quindi il client può effettuare la invocazione remota.

Il caricamento dinamico da parte del server

In questo scenario, viene previsto il caricamento di una classe da parte del server, in quanto viene passato come parametro ad una invocazione remota un oggetto istanza di una classe che non è a disposizione del server (nei suoi ClassLoader). Una illustrazione di massima del meccanismo viene illustrato nella Figura 3.5 (destra).

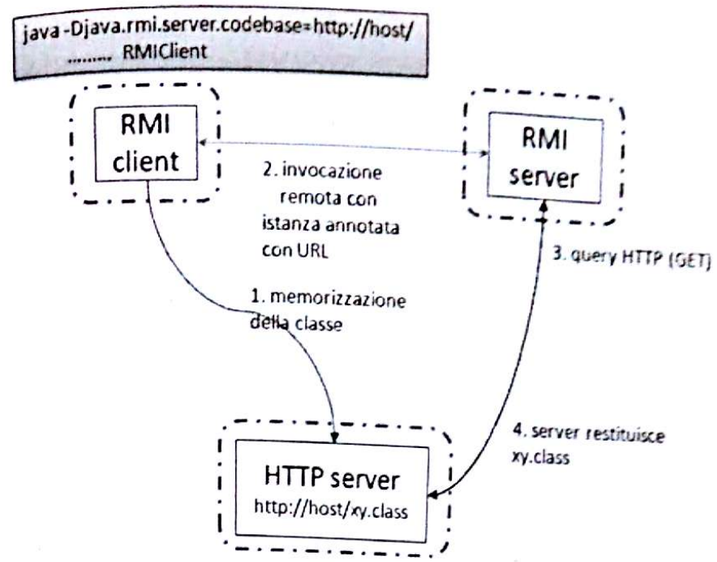

Figura 8.2: *Lo schema del caricamento dinamico di una classe da parte del server.*

Essendo Java un linguaggio fortemente tipizzato, cioé che prevede i controlli di tipo anche durante la compilazione, l'unica maniera di poter passare un parametro la cui classe non è a disposizione del server è quando il parametro passato è istanza di una sottoclasse derivata della classe presente nella firma del metodo. Questo garantisce la type-safety: il parametro passato implementa lo stesso comportamento della superclasse, specializzando opportunamente metodi e aggiungendo (se necessario) metodi e stato all'istanza della superclasse.

In questo caso, il meccanismo di Java RMI prevede che il client che vuole passare un parametro, istanza di una sottoclasse della classe prevista come parametro nella firma del metodo, debba indicare in qualche maniera la posizione della definizione della sottoclasse. Per fare questo si utilizza il meccanismo della annotazione della classe, spiegato nel dettaglio nel paragrafo 3.5.4.

Per poter annotare le classi, la macchina virtuale del client deve essere lanciata fornendo la URL del server come parametro `java.rmi.server.codebase` e poi i passi che vengono realizzati sono i seguenti (vedi anche la Figura 8.2):

1. Innanzitutto, la classe deve essere memorizzata sul server HTTP.
2. Quando il client viene lanciato ed invoca un metodo con un parametro istanza della sottoclasse xy, questa viene annotata (con il metodo `annotateClass()`) con la URL fornita alla macchina virtuale.
3. Il ClassLoader del server, quando riceve questa istanza di un oggetto annotata con la URL, provvede a caricare la classe dal server usando un tradizionale metodo GET del protocollo HTTP.
4. Il server HTTP fornisce la classe richiesta al server RMI che a questo punto usa la definizione per istanziare l'oggetto serializzato trasmesso.

Possiamo vedere un esempio di questo meccanismo. Supponiamo di avere un database di studenti, e che (tra i tanti metodi, che non dettagliamo) ci sia quello tramite il quale una istanza della classe `Studente` viene aggiunta al DB. Scriveremo due diversi progetti,

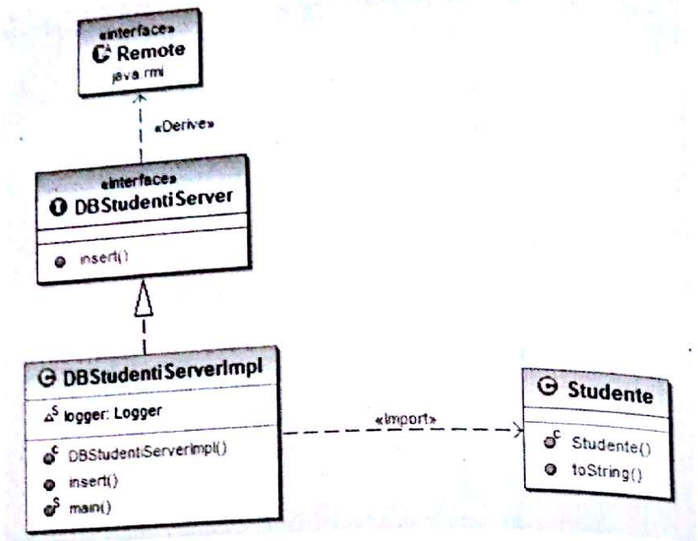

Figura 8.3: *Il diagramma delle classi del server.*

separati, per il client e per il server, in maniera da rendere evidente la differenza delle classi disponibili a ciascuno di essi.

Il diagramma del progetto server è definito in definizione della classe che racchiude i dati dello studente, che non richiede particolari commenti. Fig. 8.3. Iniziamo con la semplice

Studente.java

```
1 public class Studente implements java.io.Serializable{
2
3     private static final long serialVersionUID = 1L;
4
5     public Studente (String n, int m, String c) {
6         nome = n;
7         matricola = m;
8         corsoDiLaurea =c;
9     }
10
11    public String toString(){
12        return ("Nome="+ nome +
13                ", matricola="+ matricola +
14                ", corso di laurea="+ corsoDiLaurea);
15    }
16
17    private String nome;
18    private int matricola;
19    private String corsoDiLaurea;
20 }
```

Fine: Studente.java

Il metodo che implementiamo sulla interfaccia remota `DBStudentiServer` è uno solo (per semplicità), si chiama `insert()` e prevede l'inserimento di uno studente, con la restituzione di un booleano per indicare se l'inserimento è avvenuto correttamente (`true`) oppure no (`false`). Ecco la definizione.

DBStudentiServer.java

```
1 import java.rmi.*;
2
3 public interface DBStudentiServer extends Remote {
4     public boolean insert (Studente s) throws RemoteException;
5 }
```

Fine: DBStudentiServer.java

Ora, veniamo alla implementazione del server, nella classe `DBStudentiServerImpl`, che è estremamente semplice: dopo aver istanziato il server (linea 18) e fatto il bind sul registry (linea 19), rimane in attesa di client che invochino il servizio di inserimento (linee 25-30), che per semplicità non implementiamo, ma che provvede semplicemente a stampare a video lo studente inserito, usando il metodo `toString()` che è stato ridefinito nella classe `Studente`.

DBStudentiServerImpl.java

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.logging.Logger;
4
5 public class DBStudentiServerImpl extends UnicastRemoteObject implements DBStudentiServer {
6
7     private static final long serialVersionUID = 1L;
8     static Logger logger=Logger.getLogger("global");
9
10    public DBStudentiServerImpl() throws java.rmi.RemoteException {
11    }
12
13    public static void main(String[] args) {
14        System.setSecurityManager(new RMISecurityManager());
15
16        DBStudentiServerImpl server;
17        try {
18            server = new DBStudentiServerImpl();
19            Naming.rebind("DBServer", server);
20            System.out.println("Pronto per connessioni al DB..");
21        } catch (Exception e) {
22            logger.severe("Problemi con oggetti remoti:"+e.getMessage());
23        }
24    }
25    public boolean insert(Studente s) throws RemoteException {
26        // inserisce in un DB lo studente,
27        // se è ok, restituisce true altrimenti false
28        System.out.println ("Inserito:"+s);
29        return true;
30    }
31 }
```

Fine: DBStudentiServerImpl.java

Nel progetto del client, descritto in Fig. 8.4, l'unica classe da descrivere è il client, anche esso estremamente semplice, riportato di seguito nella classe `DBStudentiClient`: l'utente digita i campi richiesti (linee 15-22) e poi viene cercato il riferimento al server (linea 30) e invocato il metodo per l'inserimento (linea 31).

DBStudentiClient.java

```
1 import java.io.*;
2 import java.rmi.*;
3 import java.util.logging.Logger;
```

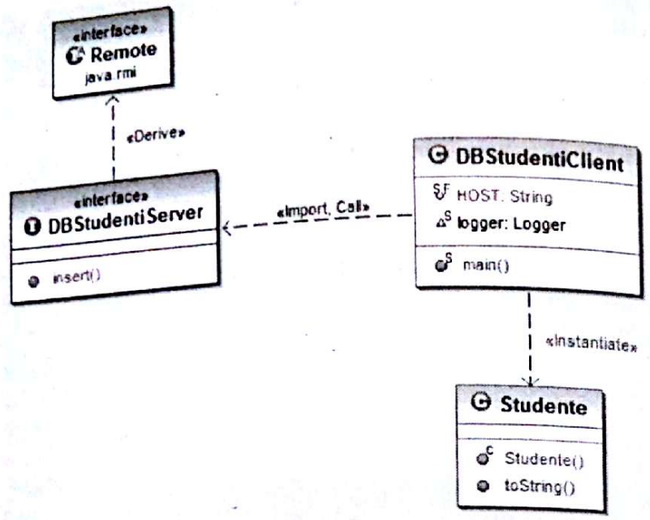

Figura 8.4: *Il diagramma delle classi del client.*

```
4
5 public class DBStudentiClient {
6     static Logger logger=Logger.getLogger("global");
7
8     public static void main(String[] args) {
9         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
10        String nome = "";
11        int matricola=0;
12        String corsoDiLaurea="";
13
14        System.out.println ("Inserimento studenti, benvenuto!");
15        try {
16            System.out.print ("Inserire nome:");
17            nome = in.readLine();
18            System.out.print ("Inserire matricola:");
19            matricola = Integer.parseInt(in.readLine());
20            System.out.print ("Inserire corso di laurea:");
21            corsoDiLaurea = in.readLine();
22        } catch (Exception e) {
23            logger.severe("Problemi con input dati:"+e.getMessage());
24            e.printStackTrace();
25            System.exit(-1);
26        }
27        System.setSecurityManager(new RMISecurityManager());
28        DBStudentiServer server;
29        try {
30            server = (DBStudentiServer) Naming.lookup ("rmi://"+HOST+"/DBServer");
31            server.insert(new Studente(nome, matricola, corsoDiLaurea));
32        } catch (Exception e) {
33            logger.severe("Non riesco a trovare il server o a inserire lo studente. Esco..");
34            e.printStackTrace();
35            System.exit(1);
36        }
37    }
```


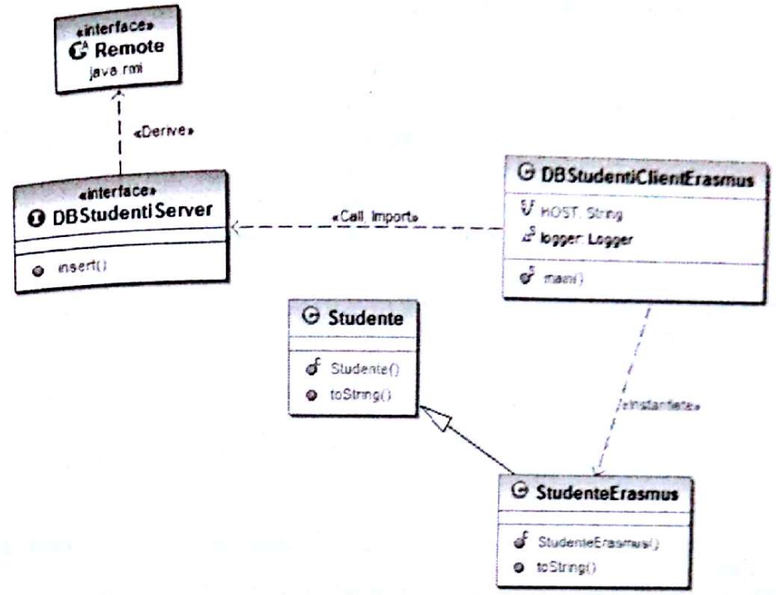

Figura 8.5: *Il diagramma delle classi del client modificato in modo che inserisca studenti del progetto Erasmus.*

```
38     public static final String HOST = "localhost";
39 }
```

Fine: DBStudentiClient.java

Ora, modifichiamo la applicazione. In questo caso, pensiamo che serva un client per poter inserire degli studenti del progetto Erasmus, che, quindi, hanno anche un altro campo da inserire (quello della nazione della università di provenienza) e il cui metodo `toString()` viene modificato in maniera conseguente. In questo caso, la situazione della applicazione client è la seguente: creiamo una classe, derivata da `Studente`, che si chiama `StudenteErasmus` e che contiene i dati e le modifiche al comportamento (metodi) necessari. Il diagramma delle classi viene mostrato nella Fig. 8.5.

La definizione della classe per lo studente Erasmus è semplice e riportata qui di seguito.

StudenteErasmus.java

```
1 public class StudenteErasmus extends Studente {
2
3     private static final long serialVersionUID = 1L;
4
5     public StudenteErasmus(String n, int m, String c, String naz) {
6         super(n, m, c);
7         nazione = naz;
8     }
9
10    public String toString(){
11        return (super.toString() +
12                ", nazione=" + nazione);
13    }
14
15    private String nazione;
```

```
16 }
```

Fine: StudenteErasmus.java

Le uniche modifiche sono quelle di aggiungere un nuovo campo `nazione` alla classe `Studente`, ridefinendo il costruttore (linee 5-8) e modificando il metodo `toString()` in modo che venga stampato anche il nuovo campo (linee 11-12).

Il client per l'inserimento di uno studente Erasmus è una semplice estensione del client visto in precedenza, che provvede a chiedere anche la nazionalità. Eccone il listato.

DBStudentiClientErasmus.java

```
1 import java.io.*;
2 import java.rmi.*;
3 import java.util.logging.Logger;
4
5 public class DBStudentiClientErasmus {
6     static Logger logger=Logger.getLogger("global");
7
8     public static void main(String[] args) {
9         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
10        String nome = "";
11        int matricola=0;
12        String corsoDiLaurea="";
13        String nazione="";
14
15        System.out.println ("Inserimento studenti Erasmus, benvenuto!");
16        try {
17            System.out.print ("Inserire nome:");
18            nome = in.readLine();
19            System.out.print ("Inserire matricola:");
20            matricola = Integer.parseInt(in.readLine());
21            System.out.print ("Inserire corso di laurea:");
22            corsoDiLaurea = in.readLine();
23            System.out.print ("Inserire nazione provenienza:");
24            nazione = in.readLine();
25        } catch (Exception e) {
26            logger.severe("Problemi con input dati:"+e.getMessage());
27            e.printStackTrace();
28            System.exit(-1);
29        }
30        System.setSecurityManager(new RMISecurityManager());
31        DBStudentiServer server;
32        try {
33            server = (DBStudentiServer) Naming.lookup ("rmi://"+HOST+"/DBServer");
34            server.insert(new StudenteErasmus(nome, matricola, corsoDiLaurea, nazione));
35        } catch (Exception e) {
36            logger.severe("Non riesco a trovare il server o a inserire lo studente. Esco..");
37            e.printStackTrace();
38            System.exit(1);
39        }
40    }
41    public static final String HOST = "localhost";
42 }
```

Fine: DBStudentiClientErasmus.java

Ora, il risultato sembra identico a quello del caso precedente, in cui si usava il client che usava la classe `Studente`: il server introduce uno studente nel DB, stampandone nome, matricola, corso di laurea e nazione. La cosa particolare, ovviamente, è che il server non contiene alcuna informazione circa la classe `StudenteErasmus`, che viene caricata in

maniera dinamica dal server HTTP nel cui file di log, infatti, si possono vedere gli accessi della macchina virtuale, indicati da linee del tipo:

```
10.0.0.136 - - [data] GET /StudenteErasmus.class HTTP/1.1 200 924 - Java/1.6.0_17
```

Da notare che, se non si chiude il server, agli inserimenti successivi di altri studenti Erasmus, il server non ha bisogno di ricaricare la classe, che evidentemente mantiene in cache.

Alcuni suggerimenti pratici per provare questi esempi. Innanzitutto, la URL che viene fornita deve terminare con / in quanto il meccanismo che viene utilizzato accoda il nome della classe da reperire alla URL. Poi, nella configurazione e lancio del server HTTP, attenzione al fatto che alcuni programmi (come Skype) usano la porta 80 e creano problemi ad Apache (ed altri server WWW) che "pretendono" di usare la porta che gli è dedicata! Quindi, in caso di problemi, controllare di avere chiuso Skype ed altri programmi simili. Infine, per verificare l'accesso ed il corretto funzionamento, è sempre il caso di verificare che la classe sia effettivamente accessibile alla URL che si usa, mediante un semplice accesso con il browser HTTP.

Il caricamento dinamico da parte del client

L'ultimo scenario è quello duale del precedente: il server ha modificato la classe del valore restituito da un metodo, e deve indicare ai client che ne invocano il metodo, come poter reperire la definizione della sottoclasse del metodo.

I passi, illustrati nella figura 8.6, sono i seguenti:

1. Il server provvede a memorizzare la definizione della classe sul server HTTP.
2. Il client effettua la invocazione remota e viene restituito, come valore, una istanza di una sottoclasse della classe indicata nella firma del metodo, annotata con la URL del server.
3. Il ClassLoader del client effettua una richiesta HTTP GET per la classe indicata ...
4. ... che viene fornita dal server HTTP, in modo che il client può usarla per poter creare l'oggetto restituito dalla invocazione remota.

8.3 Stub e Skeleton: la evoluzione

Cerchiamo adesso di approfondire il compito di stub e skeleton e come essi si sono evoluti nelle diverse versioni di Java che si sono succedute. Ricordiamo la architettura di Java RMI e di come i tre layer, Stub&Skeleton, Reference e Transport, vengano utilizzati sia lato client che lato server (si veda la figura 3.4). In questi esempi, daremo alcuni cenni su come vengono implementati i tre layer.

Utilizziamo come esempio di base il programma di HelloWorld che abbiamo visto nel paragrafo 9.4.3.

8.3.1 Stub e Skeleton versione JDK 1.1

Se vogliamo generare stub e skeleton utilizzando lo standard in uso nella versione 1.1, dobbiamo invocare il comando `rmic` passando come parametro `-v1.1`. I file stub e skeleton

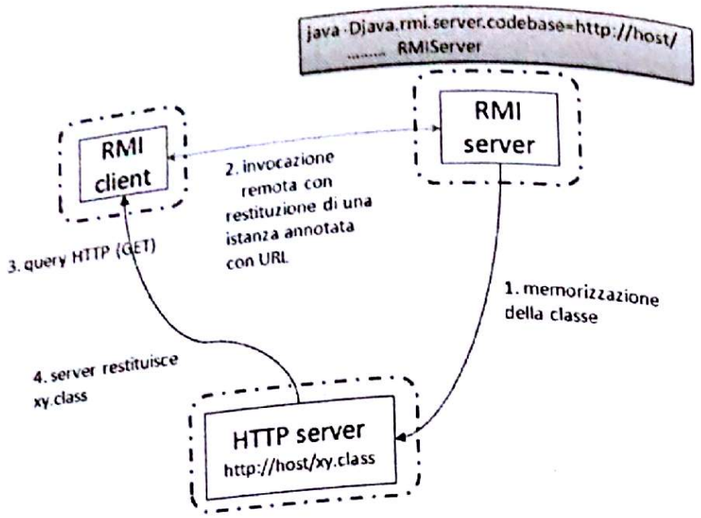

Figura 8.6: *Lo schema del caricamento dinamico di una classe da parte del client.*

(caratterizzati dal nome della classe remota, con suffisso `_Stub` e `_Skel`) sono file .class che vengono realizzati compilando i seguenti sorgenti[1], generati automaticamente.

Iniziamo con il trattare lo stub analizzando i suoi compiti. Lo stub, innanzitutto, contiene i dati che permettono al client di mandare messaggi verso il server. Quindi la *istanza* dello stub ha le informazioni sufficienti (codificate nello stato e nel comportamento (metodi) dell'oggetto stub) per trasmettere su un tradizionale socket TCP (ad esempio) le invocazioni remote del client verso il server, con i parametri di ingresso ed il valore restituito. Poi, lo stub ha anche un altro compito importante: rappresenta un meccanismo per riutilizzare i socket per le trasmissioni RMI. I socket sono costosi da aprire, in termini di tempo e di messaggi scambiati, e quindi RMI cerca di riutilizzare socket precedentemente aperti (per altre invocazioni verso la stessa macchina).

Prima di trattare il codice sorgente dello stub, generato automaticamente da `rmic`, va ricordato che molte delle classi a cui faremo riferimento sono *astratte* (cioè sono interfacce), definite in `java.rmi` e `java.rmi.server`. Se si volesse arrivare alla implementazione del meccanismo, si devono consultare (per la JVM di Sun) i package `sun.rmi`, `sun.rmi.server` e `sun.rmi.transport`[2].

HelloImpl_Stub.java

```
1 // Stub class generated by rmic, do not edit.
2 // Contents subject to change without notice.
3
4 public final class HelloImpl_Stub
5             extends java.rmi.server.RemoteStub
6             implements Hello, java.rmi.Remote {
7     private static final java.rmi.server.Operation[] operations = {
```

[1]Questi sorgenti, di norma, sono cancellati e non visibili dal programmatore, ma possono essere mantenuti invocando `rmic` con la opzione `-keep`.

[2]Ad esempio, per la interfaccia `java.rmi.RemoteCall`, la corrispondente implementazione è `sun.rmi.transport.StreamRemoteCall`, mentre per `RemoteRef` le implementazioni sono contenute in diverse classi, tra cui `sun.rmi.server.UnicastServerRef` per il lato server e `sun.rmi.server.UnicastRef` per il lato client. Senza entrare in estremo dettaglio, comunque, va ricordato che `UnicastServerRef` (server) e `UnicastRef` (client) rappresentano una parte consistente del layer di Reference, mentre `StreamRemoteCall` rappresenta la codifica del protocollo di trasporto, corrispondente al Transport Layer.


```
        new java.rmi.server.Operation("java.lang.String dimmiQualcosa(java.lang.String)")
6   };
9
10  private static final long interfaceHash = 1246442348947728853L;
11
12  // constructors
13  public HelloImpl_Stub() {
14      super();
15  }
16  public HelloImpl_Stub(java.rmi.server.RemoteRef ref) {
17      super(ref);
18  }
19
20  // methods from remote interfaces
21
22  // implementation of dimmiQualcosa(String)
23  public java.lang.String dimmiQualcosa(java.lang.String $param_String_1)
24                          throws java.rmi.RemoteException {
25      try {
26          java.rmi.server.RemoteCall call = ref.newCall(
27                                  (java.rmi.server.RemoteObject) this,
28                                  operations, 0, interfaceHash);
29          try {
30              java.io.ObjectOutput out = call.getOutputStream();
31              out.writeObject($param_String_1);
32          } catch (java.io.IOException e) {
33              throw new java.rmi.MarshalException("error marshalling arguments", e);
34          }
35          ref.invoke(call);
36          java.lang.String $result;
37          try {
38              java.io.ObjectInput in = call.getInputStream();
39              $result = (java.lang.String) in.readObject();
40          } catch (java.io.IOException e) {
41              throw new java.rmi.UnmarshalException("error unmarshalling return", e);
42          } catch (java.lang.ClassNotFoundException e) {
43              throw new java.rmi.UnmarshalException("error unmarshalling return", e);
44          } finally {
45              ref.done(call);
46          }
47          return $result;
48      } catch (java.lang.RuntimeException e) {
49          throw e;
50      } catch (java.rmi.RemoteException e) {
51          throw e;
52      } catch (java.lang.Exception e) {
53          throw new java.rmi.UnexpectedException("undeclared checked exception", e);
54      }
55  }
56 }
57
```

Fine: HelloImpl_Stub.java

Innanzitutto, lo stub è una estensione di `RemoteStub` che è una sottoclasse di `RemoteObject` che mantiene un riferimento `ref`, come handle per l'oggetto remoto. Poi, si vede come (alla linea 6) questo stub implementi la interfaccia `Hello` in modo da permettere al client di usarlo "al posto" dell'oggetto, in quanto espone gli stessi metodi dell'oggetto remoto (in particolare, l'unico metodo definito è `String dimmiQualcosa(String)`).

Ricordiamo che la istanza serializzata dello stub viene passata al client attraverso la operazione di `lookup()` sul registry. Quindi, il server ha memorizzato nel `RemoteObject` un

campo `ref` con il riferimento remoto (istanza di `java.rmi.server.RemoteRef`) che viene utilizzato dallo stub serializzato e passato al client. Questo riferimento remoto è quello utilizzato nel costruttore (linee 17-19) e che serve a costruire lo stub, lato client, una volta trasmesso dalla `lookup`. Questo riferimento, `ref`, è quello utilizzato per le invocazioni remote.

Lo stub viene costruito in modo da implementare i metodi della interfaccia remota `Hello`, in particolare `dimmiQualcosa()`, alle linee 24-56. A parte la gestione delle varie eccezioni, il lavoro effettivo della modalità di invocazione versione 1.1 viene realizzato nelle linee 27-29, 31-32, 36, 39-40, 46 che corrispondono a 5 fasi distinte: preparazione della invocazione (`newCall()`), marshalling parametri, invocazione con passaggio al client delle eccezioni (`invoke()`), unmarsalling del valore restituito e conclusione (`done()`).

Analizziamo queste 5 fasi una alla volta. La preparazione della invocazione con la chiamata di `newCall()` ha come argomenti l'array di operazioni, l'indice del metodo da invocare e l'hash dello stub (per riconoscere problemi di versione) (linee 27-29). Nella seconda fase, viene passato il parametro (linee 31-32) attraverso una tradizionale serializzazione dei parametri. Poi si effettua la vera e propria invocazione, usando il metodo `invoke()` sull'oggetto `call` che serve per far passare all'oggetto client che invoca ed eventuali eccezioni (programmate dall'utente) che debbono essere passate "verso l'alto" (infatti è l'unica istruzione non in un blocco `try ...catch`). Infine, la lettura del parametro restituito avviene alle linee 39-40, distinguendo, nelle eccezioni, da generici errori di I/O alle linee 41-42, dagli errori dovuti al passaggio di parametri la cui classe non è a disposizione del client (linee 43-44). Va detto, comunque, che in entrambi i casi viene lanciata la eccezione `UnmarshalException` con lo stesso[3] messaggio di errore. La conclusione del processo avviene nella quinta fase, con la invocazione di `done()` che permette di terminare la invocazione e poter riutilizzare il socket per altre invocazioni.

Ora possiamo passare allo skeleton, che ha il compito di offrire verso il client i metodi dell'oggetto remoto. Lo skeleton viene utilizzato dall'implementazione lato server della `RemoteRef`, chiamata `ServerRef`[4]. Come si può vedere dal codice che segue, il suo compito è molto semplice.

HelloImpl_Skel.java

```
1 // Skeleton class generated by rmic, do not edit.
2 // Contents subject to change without notice.
3
4 public final class HelloImpl_Skel
5             implements java.rmi.server.Skeleton {
6
7     private static final java.rmi.server.Operation[] operations = {
8         new java.rmi.server.Operation("java.lang.String dimmiQualcosa(java.lang.String)")
9     };
10
11    private static final long interfaceHash = 1246442348947725853L;
12
13    public java.rmi.server.Operation[] getOperations() {
14        return (java.rmi.server.Operation[]) operations.clone();
15    }
16
17    public void dispatch(java.rmi.Remote obj, java.rmi.server.RemoteCall call, int opnum,
18                         long hash)
19                        throws java.lang.Exception    {
20        if (hash != interfaceHash)
```

[3]Probabilmente la versione 1.1. è stata una versione primitiva, non particolarmente efficiente, dove, quindi, ci si può aspettare qualche imprecisione del genere.

[4]implementata in `sun.rmi.server.UnicastServerRef`.


```
21            throw new java.rmi.server.SkeletonMismatchException("interface hash mismatch");
22        HelloImpl server = (HelloImpl) obj;
23        switch (opnum) {
24        case 0: // dimmiQualcosa(String)
25        {
26            java.lang.String $param_String_1;
27            try {
28                java.io.ObjectInput in = call.getInputStream();
29                $param_String_1 = (java.lang.String) in.readObject();
30            } catch (java.io.IOException e) {
31                throw new java.rmi.UnmarshalException("error unmarshalling arguments", e);
32            } catch (java.lang.ClassNotFoundException e) {
33                throw new java.rmi.UnmarshalException("error unmarshalling arguments", e);
34            } finally {
35                call.releaseInputStream();
36            }
37            java.lang.String $result = server.dimmiQualcosa($param_String_1);
38            try {
39                java.io.ObjectOutput out = call.getResultStream(true);
40                out.writeObject($result);
41            } catch (java.io.IOException e) {
42                throw new java.rmi.MarshalException("error marshalling return", e);
43            }
44            break;
45        }
46
47        default:
48            throw new java.rmi.UnmarshalException("invalid method number");
49        }
50    }
51 }
52
```

Fine: HelloImpl_Skel.java

Gli unici metodi da implementare per la interfaccia `Skeleton` (linea 5) sono `dispatch()` e `getOperations()`. Notiamo che vengono definite le operazioni permesse (linee 7-9) ma con il solo scopo di poter interrogare lo skeleton per poterle leggere (metodo `getOperations()`, linee 13-15). In effetti, non si useranno le operazioni ma unicamente l'indice della operazione richiesta. Ricordiamo, infatti, che lo skeleton e lo stub vengono generati insieme e, quindi, sono sincronizzati: indici (ed hash) saranno uguali e coordinati. Infatti, il primo controllo del metodo `dispatch()` è quello di verificare che stub e skeleton corrispondano alla stessa istanza del client (linee 20-21) mediante il controllo dell'hash della invocazione e dello stub.

Nello `switch` alle linee 24-50 vengono eseguiti i vari metodi remoti, caratterizzati da un indice. In questo caso, abbiamo un solo metodo, e quindi un solo `case` alle linee 25-46. Viene letto l'input sullo stream, corrispondente ad una stringa (linee 27-37), viene effettuata la invocazione sull'oggetto server (linea 38) che è in questo caso un riferimento ad un oggetto locale (lo skeleton si trova sul server) e viene inviato il risultato (linee 39-44).

8.3.2 Stub versione JDK 1.2

A questo punto, sembra evidente che il lato skeleton (versione JDK 1.1) è effettivamente abbastanza semplice: se la chiamata remota deve semplicemente essere "tradotta" nella invocazione corrispondente sull'oggetto locale, questo può essere fatto senza utilizzare uno skeleton, inserendo maggiori informazioni su quanto lo stub passa agli strati inferiori di Ja-

va RMI ed aumentando leggermente la complessità del Layer di Reference, che si occuperà di fare marshalling dei parametri (che prima veniva effettuato dallo strato stub/skeleton).

Questo è quello che viene realizzato dalla versione 1.2 del JDK. Lo stub viene reso in maniera diversa, utilizzando la *reflection* del package `java.lang.reflect` per poter passare alla invocazione (dello strato sottostante) un oggetto "metodo", cioè un oggetto che codifica le informazioni di un metodo dell'oggetto remoto, ottenuto attraverso l'uso della definizione della classe dell'oggetto remoto. In effetti, quello che basta, ovviamente, è che si possa accedere alla definizione della "interfaccia" dell'oggetto remoto, essendo ovviamente la implementazione remota non disponibile lato client. Ecco il sorgente dello stub versione 1.2.

HelloImpl_Stub.java

```
1 // Stub class generated by rmic, do not edit.
2 // Contents subject to change without notice.
3
4 public final class HelloImpl_Stub
5             extends java.rmi.server.RemoteStub
6             implements Hello, java.rmi.Remote {
7     private static final long serialVersionUID = 2;
8
9     private static java.lang.reflect.Method $method_dimmiQualcosa_0;
10
11    static {
12        try {
13            $method_dimmiQualcosa_0 = Hello.class.getMethod("dimmiQualcosa",
14                    new java.lang.Class[] {java.lang.String.class});
15        } catch (java.lang.NoSuchMethodException e) {
16            throw new java.lang.NoSuchMethodError(
17                    "stub class initialization failed");
18        }
19    }
20
21    // constructors
22    public HelloImpl_Stub(java.rmi.server.RemoteRef ref) {
23        super(ref);
24    }
25
26    // methods from remote interfaces
27
28    // implementation of dimmiQualcosa(String)
29    public java.lang.String dimmiQualcosa(java.lang.String $param_String_1)
30                    throws java.rmi.RemoteException {
31        try {
32            Object $result = ref.invoke(this, $method_dimmiQualcosa_0,
33                    new java.lang.Object[] {$param_String_1}, -1794656071447706533L);
34            return ((java.lang.String) $result);
35        } catch (java.lang.RuntimeException e) {
36            throw e;
37        } catch (java.rmi.RemoteException e) {
38            throw e;
39        } catch (java.lang.Exception e) {
40            throw new java.rmi.UnexpectedException("undeclared checked exception", e);
41        }
42    }
43 }
```

Fine: HelloImpl_Stub.java

Notiamo come, alle linee 4-6, la dichiarazione dello stub è simile a quella della versione 1.1. Alla linea 7 si differenzia la versione (in modo da controllare eventuali disallineamenti

nell'utilizzo di stub di versioni differenti). Nel blocco di inizializzazione della classe, linee 11-19, viene istanziato l'oggetto `Method` che si ottiene attraverso l'uso della definizione della *interfaccia* `Hello`, che serve, qui, come classe astratta per permettere solo di recuperare le informazioni circa il metodo[5], non avendo il client a disposizione, ovviamente, la implementazione dell'oggetto remoto.

Il compito dello stub, rispetto alla versione precedente, è molto semplificato e consiste nell'utilizzo del metodo `invoke()` del riferimento remoto. Alle linee 32-33, questa invocazione prende l'oggetto metodo, un array di parametri, e un long di hash. Il risultato viene restituito alla linea 34. Va detto che i dati passati alla `invoke()` servono scopi differenti: l'oggetto "metodo" che viene passato serve (nella definizione del metodo in `sun.rmi.UnicastRef`, ad esempio) ad ottenere la lista delle classi dei parametri in input, mentre l'hash che viene passato (il terzo parametro) serve a contraddistinguere il metodo da invocare[6].

Va detto, in conclusione, che la semplificazione nella versione 1.2 non è estremamente significativa. Anche se l'eliminazione dello skeleton nella versione 1.2 aggiunge un minimo overhead rispetto alla versione precedente (lo skeleton versione 1.1 è più efficiente, in quanto più semplice), il miglioramento nel deployment non è impressionante: il problema, infatti, non è aggiornare il server (che è uno solo) ma i client (che sono tanti) e quindi la eliminazione dello stub è quella che avrebbe sicuramente avuto un maggiore impatto sulla semplicità dell'ambiente. Questo è quello che viene realizzato nella versione 1.5 della JDK.

8.3.3 JDK 5: niente Stub e Skeleton!

In effetti, un certo numero di versioni della JDK si sono succedute, prima di riuscire ad eliminare la necessità anche dello stub, all'interno di RMI. Questo è stato realizzato attraverso una serie di passaggi logici che hanno permesso di utilizzare i proxy dinamici per generare a run-time uno stub. Il primo passo logico è stata la introduzione dei proxy dinamici.

Proxy dinamici

Il design pattern del Proxy è alquanto popolare (non solo in campo distribuito): una classe che accetta richieste e le inoltra ad una altra classe che compie il lavoro. In effetti, il meccanismo dello stub è una realizzazione del design pattern del proxy.

I proxy dinamici sono stati introdotti nella versione 1.3 di Java. Si tratta di una classe che implementa una lista di interfacce, che sono specificate a tempo di esecuzione (e non scritte nel codice). In questa maniera, una invocazione di un metodo su una istanza di un proxy dinamico viene realizzata attraverso una modalità unica, e viene recapitata ad un oggetto che implementi una delle interfacce del proxy. I proxy dinamici possono servire ad alcuni utilizzi interessanti, quali quello di aggiungere una fase di autenticazione a tutti i metodi di una classe. Per fare questo, si può realizzare un proxy che intercetta tutte le invocazioni e le esegue solo se viene passata una fase di autenticazione.

[5] In effetti, come vedremo, serve solamente per recuperare informazioni circa la sua firma, cioè il tipo dei suoi parametri.

[6] Per distinguere tra le invocazioni secondo il modello 1.1 ed il modello 1.2, lo `StreamRemoteCall` usa un indice di metodo negativo (-1) per le invocazioni 1.2, utilizzando il codice hash, che caratterizzava nella 1.1 la corrispondenza stub/skeleton, per definire il metodo da invocare.

Un esempio di proxy dinamico

Per chiarire l'utilizzo dei proxy dinamici, vediamo un esempio, abbastanza semplice, ma efficace nel rappresentare le caratteristiche significative di questo strumento.

Supponiamo di voler che l'accesso ad un `ArrayList` avvenga solamente dietro un controllo della autenticità dell'utente e, a seconda del suo ruolo e del metodo invocato, fare in modo di garantire o meno l'accesso.

Vediamo una classe che faccia questo controllo esibirebbe un metodo `check()` che, dato il metodo, restituisce un booleano che è vero se l'accesso è garantito e falso altrimenti.

MasterControl.java

```
1 import java.lang.reflect.*;
2
3 public class MasterControl {
4     public static boolean check(Method m) {
5         System.out.println ("Controllo l'esecuzione di "+ m.getName());
6         // qui ci andrebbe il controllo...
7         System.out.println ("ok!");
8         return true;
9     }
10 }
```

Fine: MasterControl.java

Come si vede, per semplicità il metodo `check()` in questo momento non effettua nessun controllo e stampa solo a video alcune informazioni. La domanda ora è: come facciamo a inserire questo controllo per *tutti* i metodi della classe `ArrayList`? Probabilmente, con un design pattern dell'adapter possiamo andare a riscrivere tutti i metodi, inserendo una invocazione a `check()` prima della invocazione del metodo dell'oggetto ArrayList (che fa la parte dell'Adaptee nel pattern).

A questo può servire, invece, il pattern del proxy, creando un proxy dinamico che faccia questo lavoro in poche righe di codice. Vediamo come in questo esempio.

ListProxyFactory.java

```
1 import java.lang.reflect.*;
2 import java.util.List;
3
4 public class ListProxyFactory {
5     public static List<String> getListProxy(final List<String> lis) {
6         return (List<String>) Proxy.newProxyInstance (
7                 lis.getClass().getClassLoader(),
8                 new Class[] { List.class },
9                 new InvocationHandler() {
10                     public Object invoke(Object proxy, Method method, Object[] args)
11                                                 throws Throwable {
12                         if (MasterControl.check(method))
13                             return method.invoke(lis, args);
14                         else
15                             return (List<String>)null;
16                     } // fine metodo invoke
17                 } // fine invocation handler
18         ); // fine dichiarazione proxy
19     } // fine metodo getListProxy
20 } // fine classe
```

Fine: ListProxyFactory.java

La classe `ListProxyFactory` ha solo un metodo statico `getListProxy()`, che restituisce un riferimento ad un oggetto che implementa una `List<String>` (linea 5). All'interno

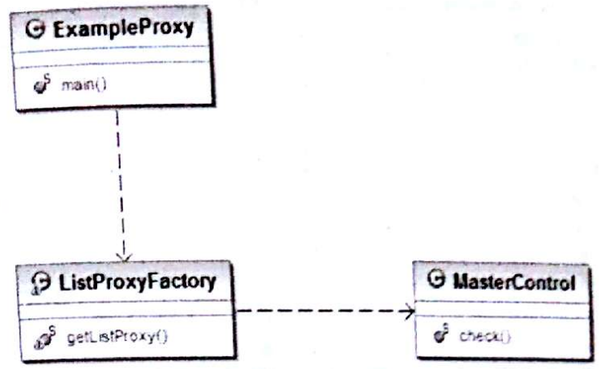

Figura 8.7: *Il diagramma delle classi dell'esempio sui proxy dinamici.*

c'è una sola linea di codice (di `return`) che effettua tutto quanto necessario per creare un proxy dinamico. Innanzitutto, si usa il metodo statico `newProxyInstance` della classe `java.lang.reflect.Proxy` (linea 6) per creare un proxy dinamico, che implementi un certo numero di interfacce ed usi un invocation handler particolare. I parametri del metodo sono tre: il primo (linea 7) è il ClassLoader da usare e si usa il ClassLoader utilizzato dal parametro passato al metodo (`lis`). Il secondo parametro (linea 8) definisce l'array di interfacce che vanno definite dal proxy (in questo caso solamente la classe che definisce `List` che, ricordiamo, è una interfaccia) mentre il terzo parametro (linee 10-17) serve a definire l'invocation handler.

Come invocation handler, definiamo una classe anonima che implementa l'unico metodo necessario, quello di `invoke()` che prende tre parametri: il proxy, il metodo e la lista degli argomenti. Lo scopo di `invoke()` è quello di invocare il metodo su un proxy dinamico. Il corpo di questo metodo è ovviamente molto semplice: se il controllo di `MasterControl` viene superato si invoca il metodo, altrimenti si restituisce `null`.

Vediamo, ora, come si può usare questo esempio, in una semplice applicazione schematizzata dal diagramma delle classi in Figura 8.7.

In questa semplice applicazione, vogliamo semplicemente aggiungere dati in un ArrayList (con la autenticazione realizzata da `MasterControl`). Vediamo il sorgente del programma.

ExampleProxy.java

```
1 import java.io.*;
2 import java.util.*;
3
4 public class ExampleProxy {
5     public static void main(String[] args) {
6         String cmd;
7         List<String> list = ListProxyFactory.getListProxy(new ArrayList<String>());
8 //        List<String> list = new ArrayList<String>();
9         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
10        try {
11            while (!(cmd = ask ("Comandi>", in)).equals("!quit")) {
12                if (cmd.equals("!list")) {
13                    Iterator<String> itr = list.iterator();
14                    while(itr.hasNext())
15                        System.out.println(itr.next());
16                } else
17                    list.add(cmd);
```

```
18          }
19      } catch (IOException e) {
20          e.printStackTrace();
21      }
22  }
23
24  private static String ask(String prompt, BufferedReader in) throws IOException {
25      System.out.print(prompt+" ");
26      return (in.readLine());
27  }
28 }
```

Fine: ExampleProxy.java

Alla linea 7, si crea (a run-time!) il proxy dinamico: viene passato un oggetto `ArrayList` sul quale viene creato un proxy dinamico che viene utilizzato per tutte le successive invocazioni (che verranno gestite dall'invocation handler del proxy). Alla linea 8, si trova la linea di codice che è equivalente alla creazione statica di un oggetto, che è equivalente alla linea 7, ma ovviamente senza alcun adapter al comportamento.

L'inserimento di dati viene realizzato in un ciclo (linee 11-18: tutto quanto viene digitato dall'utente (fino a quando non digita `''!quit''`, linea 11) viene aggiunto alla lista, invocando (linea 17) il metodo add. Come si vede, la invocazione è assolutamente equivalente alle invocazioni tradizionali, solamente che questa viene intercettata dal proxy che "sovraimpone" il proprio comportamento e la propria gestione, codificata dal programmatore nel metodo `invoke()`.

Se l'utente digita `''!list''`, allora viene stampata a video tutta la lista (linee 12-16). Anche qui, il metodo `iterator()` per prelevare un iteratore da `list` (linea 13) verrà intercettato e soggetto alla autenticazione di `MasterControl`.

Come si vede, i proxy dinamici rappresentano una maniera interessante per poter modificare in maniera completa il comportamento di un oggetto. A parte l'utilizzo che descriveremo successivamente, questi strumenti possono essere utili nella cosiddetta programmazione *orientata agli aspetti*, dove gli aspetti sono entità esterne che offrono caratteristiche trasversali alle interazioni tra tutti gli oggetti, in maniera coordinata e strutturata. Ad esempio, la autenticazione (e in generale, le politiche di sicurezza) si presta bene ad essere implementate efficientemente ed efficacemente con gli aspetti, come abbiamo visto usando i proxy dinamici.

Generazione dinamica dello Stub

Ovviamente, i proxy dinamici sembrano costruiti appositamente per risolvere il problema dello stub. Per un oggetto remoto, si ha a disposizione la definizione della interfaccia, quindi si ha la possibilità, quando l'oggetto remoto viene esportato (cioè reso accessibile mediante il costruttore di `UnicastRemoteObject` oppure `exportObject()`) di creare lo "stub" a tempo di esecuzione. Quello che accade è che, se in quel momento non è accessibile uno stub pregenerato, allora lo stub dell'oggetto remoto sarà una istanza di `java.lang.reflect.Proxy` che usa come handler per la gestione delle invocazioni quello definito in `java.rmi.server.RemoteObjectInvocationHandler`. Questo proxy implementerà tutti i metodi indicati nella interfaccia remota, e la modalità con cui vengono eseguiti vengono definiti dall'handler. Nella implementazione di Sun, una implementazione del meccanismo si trova in `sun.rmi.server.Util.createProxy()`.

In effetti, questo tipo di soluzione semplifica di molto non solo il compito del programmatore, ma rende anche abbastanza lineare il codice del layer di Reference e di Transport, che rimane complesso solamente perchè deve assicurare la compatibilità con gli altri due meccanismi di Stub & Skeleton. Inoltre, è possibile forzare il meccanismo con i proxy

dinamici anche in presenza di stub generati con rmic, assegnando a `true` la proprietà `java.rmi.server.ignoreStubClasses` (da linea di comando della macchina virtuale).

Note bibliografiche

Caricamento dinamico delle classi

Una presentazione dettagliata del meccanismo di ClassLoader all'interno della macchina virtuale (senza particolare riferimento a RMI) viene presentata in [30] mentre informazioni specifiche sull'uso in RMI vengono date in [5] e [38]. Altre informazioni, sul sito della Sun, sono disponibili nel tutorial sulle estensioni della Java Virtual Machine (disponibile a `http://java.sun.com/docs/books/tutorial/ext/basics/load.html`).

Stub e Skeleton

Lo studio della evoluzione di Stub e Skeleton è stato effettuato sulla base dei codici sorgenti e della documentazione del JDK SE Java 6(in particolare le Releases Notes di RMI della versione J2SE 5.0). In [17], si trova un interessante articolo di confronto e di studio dell'evoluzione del deployment in Java RMI. Per i sorgenti, si possono scaricare all'indirizzo `http://www.sun.com/software/opensource/java/`. I sorgenti che abbiamo utilizzato sono relativi alla versione 6.

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[R**]** A cosa serve in Java il caricamento dinamico delle classi in ambito distribuito?
2. **[R*]** Cosa fa il ClassLoader?
3. **[R**]** Quali sono le caratteristiche del caricamento dinamico delle classi in Java?
4. **[R***]** Descrivere ciascuno dei tre scenari in cui viene caricata dinamicamente una classe in Java RMI, con un esempio.
5. **[P**]** Estendere l'esempio degli Studenti Erasmus, in modo che anche i client siano obbligati a caricare dinamicamente una classe.
6. **[R*]** Perché non è più utilizzato (nella pratica) il caricamento dinamico dello stub?
7. **[R***]** Quali sono (dai listati di Java inclusi nel testo) alcune delle classi coinvolte nel Reference Layer?
8. **[R**]** A cosa serve il pattern dei Proxy Dinamici?
9. **[R***]** Quale è il ruolo dei proxy dinamici per la generazione dinamica dello stub?

Capitolo 9

RMI e Java Enterprise Edition

Indice

9.1 Introduzione **196**
9.2 Java Enterprise Edition **196**
9.2.1 Le architetture multi-tier 196
9.2.2 La architettura di Java EE 197
9.3 Corba e IIOP **198**
9.4 Java RMI-IIOP **200**
9.4.1 Java RMI e Corba 200
9.4.2 Le differenze tra RMI e RMI-IIOP 200
9.4.3 Un esempio in Java RMI-IIOP 201
Note bibliografiche **206**
Spunti per lo studio individuale **207**

9.1 Introduzione

In questo capitolo puntiamo a definire una estensione di Java Remote Method Invocation che usi un protocollo di trasporto diverso: invece di quello nativo, JRMP (Java Remote Method Protocol), viene usato lo standard CORBA Internet Inter-ORB Protocol (IIOP). Questo permette, innanzitutto, di ottenere la interoperabilità a livello di trasporto con l'ambiente CORBA, cosa che rappresentava un importante passo per Java RMI (all'epoca della sua introduzione).

Questa estensione, che si chiama RMI-IIOP, è importante anche perché è il protocollo utilizzato per implementare le invocazioni remote tra client e Enterprise JavaBeans all'interno di Java Enterprise Edition, la cui struttura brevemente descriveremo, prima di passare a RMI-IIOP.

9.2 Java Enterprise Edition

Java Enterprise Edition (Java EE) è un insieme di tecnologie integrate, basate su Java, con l'obiettivo di ridurre il costo e la complessità di sviluppare, installare e gestire applicazioni basate su server che siano basate su una *architettura multi-tier*.

Gli obiettivi di riduzione dei costi sono particolarmente importanti per le aziende, in quanto è possibile diminuire il tempo di risposta ai cambiamenti del mercato e poter, quindi, fornire in maniera puntuale servizi a clienti, partner, impiegati e fornitori. Le applicazioni che devono fornire questi servizi tipicamente devono integrare diversi sistemi informativi aziendali con l'obiettivo di fornire servizi che assicurino alta disponibilità, sicurezza, affidabilità e scalabilità.

Questi obiettivi sono raggiunti da una architettura realizzata su diversi *tier*, dove tra i client (sul front end) ed i dati (sul back end) vengono frapposti alcuni tier intermedi che fungono da piattaforma per la esecuzione della logica della applicazione. Questi *middle tier* sono il posto, nella architettura, dove vengono implementati (in maniera coordinata ed omogenea) i servizi che nascondono all'utente la complessità e la eterogeneità dei sistemi informativi i cui dati vengono integrati.

9.2.1 Le architetture multi-tier

Una popolare categorizzazione delle funzionalità di una applicazione distribuita le suddivide in tre layer software:

- Layer di presentazione, che consiste nelle funzionalità di presentazione verso l'utente del risultato della applicazione distribuita.
- Layer di business, che implementa la logica della applicazione.
- Layer di accesso ai dati, che realizza l'uso delle informazioni memorizzate nei DBMS.

Questi tre layer di software possono essere collocati in maniera diversa all'interno di una architettura hardware che, in maniera analoga, può essere strutturata in *tier*.

Le architetture *single tier* sono quelle utilizzate dalle applicazioni "monolitiche" degli anni '70-'80, dove tutti e tre i layer venivano offerti da un singolo computer (mainframe) mentre l'utente accedeva alla applicazione attraverso un semplice terminale (tipicamente alfanumerico) che non aveva alcuna capacità di elaborazione.

La presenza sempre maggiore di Personal Computer, con una capacità di calcolo non trascurabile (anche in quegli anni), introdusse di fatto una architettura *two-tiers* dove i

layer di presentazione e di logica di business venivano implementati, mentre solamente l'accesso ai dati veniva gestito dal secondo tier, dove un DBMS forniva l'accesso ai dati.

Il passo successivo è consistito nel frapporre tra DBMS e client[1] un server che centralizzasse la logica di business, per facilitarne la gestione e migliorare la efficacia, lasciando il layer dipresentazione sui client. In queste prime architetture 3-tier, il middle-tier forniva servizi centralizzati, spesso acceduti attraverso protocolli di Remote Procedure Call.

La naturale evoluzione di questa architettura è consistita nell'utilizzare gli oggetti (distribuiti) nel middle-tier, potendo contare su un paradigma di programmazione che permetteva maggiorigaranzie di stabilità, robustezza e riusabilità del codice (scalabilità efacile manutenzione). Il middle tier è utile per poter gestire funzionalità di base per la logica della applicazione, quale concorrenza,transazioni, etc. I protocolli che vengono utilizzati dal client per comunicarecon gli oggetti distribuiti sono differenziati a seconda del frameworkapplicativo usato, da DCOM (per ambienti Microsoft) a RMI (per ambienti Java) ad Internet Inter-Orb Protocol[2](IIOP) (per CORBA).

La successiva evoluzione vede inserire un altro tier (e generalmente si parla di architettura n-tier) basato sul browser HTTP nella architettura del World Wide Web. In questa maniera il client diventa il browser, che funziona come "client universale" per tutte le applicazioni, che diventano tutte "Web-based". In questa maniera i tier diventano (almeno) 4: un tier di presentazione lato client sul client (browser web), un tier di presentazione lato server su un server HTTP (come Apache) che fornisce l'assemblaggio delle informazioni fornite dalla applicazione, mediante tecnologie come JSP, Servlet, ASP, etc.. Un terzo tier consiste nel server (di applicazione) dove la logica di business viene effettivamente realizzata, ed un quarto tier di accesso ai dati. La comunicazione tra il server HTTP e il server dalla applicazione continua ad utilizzare tecnologie e protocolli per la comunicazione verso oggetti distribuiti (DCOM, RMI, IIOP).

In questo tipo di architettura, gli obiettivi principali da ottenere sono quelli di distinguere, separare e localizzare le responsabilità delle componenti della applicazione, in modo da permetterne una evoluzione indipendente e che permette la scalabilità al mutare delle condizioni. Ad esempio, se si decide di cambiare il DBMS, si deve solamente modificare la maniera in cui i dati vengono reperiti dal tier del server di applicazione verso il nuovo DBMS. In generale, quindi, si punta ad ottenere riusabilità, flessibilità e scalabilità della architettura. Va comunque evidenziato che restano a carico del progettista/programmatore numerose problematiche che sono "trasversali" alla architettura, quali sicurezza, bilanciamento del carico su più server, gestione della concorrenza, transazioni, gestione dinamica ed ottimale delle risorse del sistema, etc. Queste problematiche sono trasversali in quanto ricoprono in maniera completa tutti i layer/tier della architettura (sia software che hardware, quindi) e richiedono un approccio unitario (e tra essi stessi interoperabile) per poter essere risolti al meglio. Questi sono gli obiettivi della piattaforma Java 2 Enterprise Edition (JEE).

9.2.2 La architettura di Java EE

Java EE introduce un modello di programmazione simplificata, in quanto facilita lo sviluppo di applicazioni "enterprise" tramite l'utilizzo di API (Application Programming Interface) e di meccanismi integrati per assicurare servizi complessi con semplicità.

Come illustrato in Fig. 9.1, la architettura di Java EE si basa su 3 tier, ma il tier centrale viene suddiviso in due parti, realizzando, in pratica, una architettura n-tier. Il motivo della

[1] Ilclient con elevate capacità di calcolo veniva spesso indicato con il termine *difat client*.
[2] Il protocollo IIOP verrà illustrato nelparagrafo successivo.

aggregazione in un singolo tier centrale è che tipicamente esso risiede su un singolo server (possibilmente duplicato e ridondato per tolleranza ai malfunzionamenti ed efficienza).

I client in Java EE possono essere di due tipi: Web client ed application client. Nel primo caso, si tratta di pagine WWW dinamiche che, attraverso tecnologie di markup, vengono visualizzate da un Web browser. Tipicamente, questo tipo di client viene chiamato *thin* (in opposizione ai *fat* client) in quanto non fanno alcuna operazione onerosa, quali richieste dirette allo strato di business oppure al database.

Un altro tipo di client sono gli application client, applicazioni Java che offrono una interfaccia più ricca di quella che può essere fornita da un browser Web e da un linguaggio di markup. In questo caso, ovviamente, i client possono anche bypassare lo strato di presentazione implementato sul Web server (in quanto sono dotati di funzionalità di presentazione evoluta) ed interfacciarsi direttamente con lo strato di business che viene eseguito sul server di applicazioni. In questo caso, viene usato il protocollo RMI-IIOP (che descriveremo di seguito in questo capitolo) per interrogare e interagire con le componenti distribuite (Enterprise JavaBeans) che compongono lo strato di business sull'application server.

Lo strato di business, con gli EJB, rappresenta una importante caratteristica della architettura, in quanto permette di ottenere (molto semplicemente[3]) quei servizi "trasversali" per la applicazione, precedentemente menzionati, in maniera dichiarativa, cioè creando degli oggetti distribuiti che rispondono a certi standard (componenti distribuite) e facendone gestire sicurezza, bilanciamento del carico, transazioni, etc. all'application server, sulla base di una serie di richieste dichiarate, appunto, dal programmatore. In questa maniera, senza dover re-implementare politiche, oppure neanche inserire del codice per poter richiamare API di uno specifico servizio, si effettua il "deploy" di una componente distribuita e la si lascia gestire all'application server, che offre i servizi evoluti richiesti dal programmatore.

Un Enterprise JavaBean viene realizzato scrivendo del codice Java (secondo una modalità detta POJO: "Plain Old Java Object") con dei metadati (annotazioni Java) che permettono all'application server (detto EJB container) di trattare questa componente in maniera da offrire (automaticamente) servizi evoluti, quali (in aggiunta a quelli citati in precedenza) il pooling di risorse (un pool di istanze che vengono condivise automaticamente dai client), gestione dello stato (persistenza), accesso mediante web services, ottimizzazione delle prestazioni con tecniche di caching.

Punto cardine di questa architettura, dal punto di vista della comunicazione, è l'uso dei protocolli RMI e RMI-IIOP per la comunicazione sia tra application client ed EJB che tra le varie parti (potenzialmente distribuite anch'esse) del Java EE application server.

9.3 Corba e IIOP

Corba ha rappresentato, negli anni '90, un tentativo organizzato di standardizzare la computazione con oggetti distribuiti (come abbiamo brevemente descritto nel paragrafo 1.3.3).

Per permettere la interoperabilità tra oggetti remoti, che potevano essere su piattaforme eterogenee, venivano fornite delle specifiche, indipendenti dalla piattaforma utilizzata. Il primo livello era quello della specifica delle richieste di invocazione remota, permettendo la definizione con un Interface Definition Language (IDL) delle funzionalità remote offerte

[3]Java Enterprise Edition è passato attraverso diverse edizioni a partire dalla introduzione nel 1998, a J2EE 1.2 (1999), a J2EE 1.3 (2001) a J2EE 1.4 (2003), a JEE 1.5 (2006) fino alla attuale versione JEE 1.6. Si deve osservare che solamente le ultime due versioni permettono effettivamente un lavoro semplice al programmatore, mentre le altre erano abbastanza macchinose.

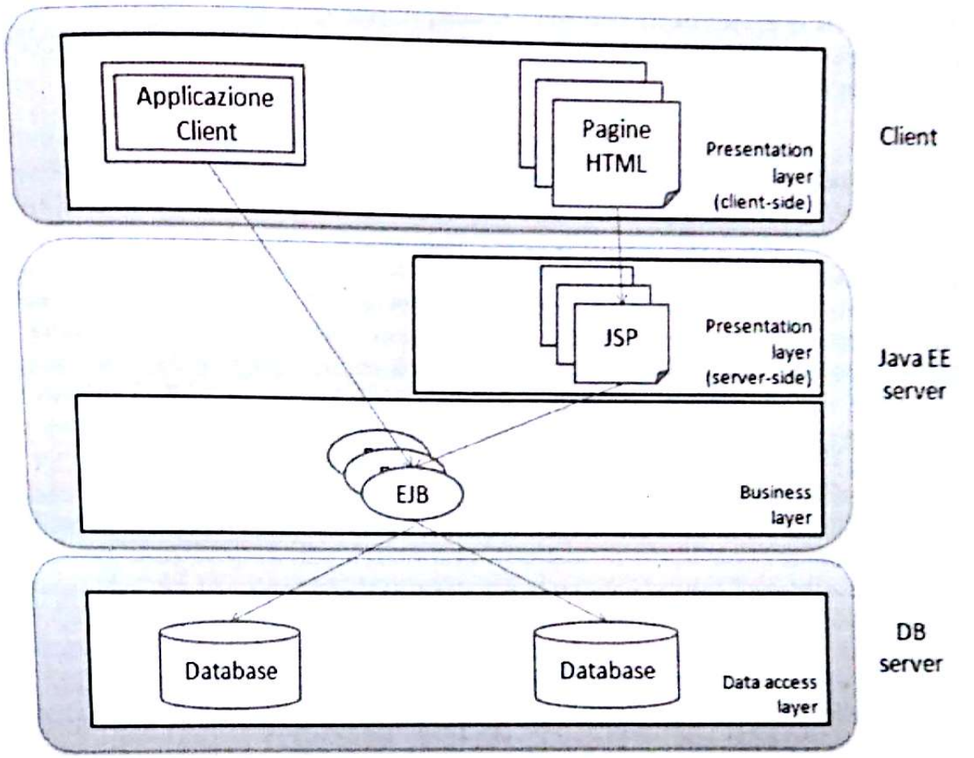

Figura 9.1: *La architettura di Java Enterprise Edition.*

dagli oggetti (definizione dei metodi da invocare). Per fare un paragone, la soluzione di Java RMI per questo tipo di problemi, consiste nell'usare le specifiche del linguaggio stesso, usando la definizione di una `interface` con una specifica interfaccia marker `Remote`.

Il livello della sintassi dei messaggi scambiati tra Object Request Broker (ORB) per permettere la comunicazione tra oggetti remoti viene definito, in maniera astratta, da General Inter-ORB Protocol (GIOP). Il livello di trasporto venivarealizzato a seconda dei protocolli di rete utilizzati, e nello standard CORBAviene definito come obbligatoria la implementazione di GIOP suTCP/IP che viene realizzata da Internet Inter-ORB Protocol (IIOP). Questo protocollo aveva diversi vantaggi (implementazione semplice ed efficiente) ed aveva lo scopo di garantire a diversi ORB, anche di produttori diversi, di poter collaborare, in maniera standard, alla gestione di oggetti remoti. La caratteristica di CORBA di essere non solo una piattaforma eterogenea (in termini di software e di hardware) ma di cercare anche di indirizzare i problemi di compatibilità tra vendor diversi, era una importante aspetto dello standard, anche se risolto, in molti casi, in maniera non del tutto soddisfacente.

9.4 Java RMI-IIOP

L' obiettivo di Java *Remote Method Invocation over IIOP* (RMI-IIOP) è quello di combinare insieme la facilità di uso di Java RMI con la flessibilità e lo standard fornito da IIOP per gli ORB di CORBA, in maniera da permettere di poter anche integrare componenti legacy scritte in altri linguaggi che usano CORBA. Quindi, l'obiettivo era quello di aprire le applicazioni CORBA alle modalità di invocazione di Java Remote Method Invocation.

9.4.1 Java RMI e Corba

Volendo scrivere applicazioni direttamente in CORBA (rinunciando, quindi, alle invocazioni stile RMI) è possibile usare Java IDL, in modo da rispettare lo standard delle applicazioni CORBA. L'uso di RMI-IIOP, che si concentra solamente sullo strato di trasporto, permette in maniera agevole di potersi interfacciare a servizi CORBA preesistenti con applicazioni RMI (usando RMI-IIOP).

In effetti, prima della introduzione di RMI-IIOP, le modalità di invocazione che usavano RMI e Corba erano nettamente distinte (Fig. 9.2 (sinistra)) mentre con RMI-IIOP la interoperabilità è aumentata. Infatti, è possibile che un client Corba possa invocare servizi di un server RMI-IIOP, così come un client Java che usi RMI-IIOP possa invocare servizi di un server RMI, di un server RMI-IIOP e di un server CORBA, in quest'ultimo caso, con alcune limitazioni. Infatti, IDL permette la ereditarietà multipla che non è permessa in Java e, qualora il server Corba usi queste caratteristiche, il client non è in grado di accedere a questa specifica che è più ricca semanticamente di quanto permette il linguaggio stesso (situazione indicata da una freccia tratteggiata nel diagramma a destra della Fig. 9.2).

9.4.2 Le differenze tra RMI e RMI-IIOP

Per poter usare RMI-IIOP, si devono generare stub e skeleton particolari, utilizzando, però, lo stesso strumento fornito da RMI, `rmic`. In effetti, con un parametro, `rmic -iiop` genera lo stub ed il *tie* (nome Corba per lo skeleton) per poter usare RMI-IIOP come protocollo di trasporto, mentre con `rmic -idl` genera la interfaccia in IDL, in modo da permettere ad un client Corba (non Java) di poter accedere al server Java che usa RMI-IIOP.

Una importante differenza tra RMI e RMI-IIOP sta nel servizio di naming da utilizzare. Per poter usare RMI-IIOP, si deve usare Java Naming and Directory Interface (JNDI), un

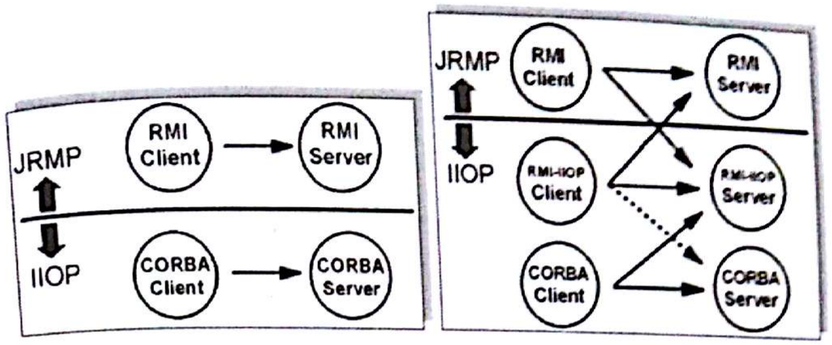

Figura 9.2: *La distinzione tra il mondo RMI e quello CORBA prima di RMI su IIOP (sinistra) e dalla introduzione di RMI-IIOP (destra). La freccia tratteggiata indica che solo una parziale interoperabilità è permessa.*

servizio di Java che fornisce una interfaccia unica per accedere a diversi servizi di naming e di directory. Una caratteristica di JNDI è che può essere programmato, fornendo (in aggiunta ai tradizionali servizi di naming) nuovi servizi di naming per nuovi protocolli. Infatti, JNDI consiste sia di una API (Application Programming Interface) per il programmatore che vuole usare uno dei servizi JNDI ma anche di una SPI (Service Provider Interface) in modo da poter aggiungere nuovi servizi, qualora necessario.

Tra i servizi forniti con JNDI, troviamo il DNS, LDAP, il tradizionale RMI registry ed il servizio di naming di Corba (Corba Common Object Services). Per poter usare un servizio di naming di JNDI, si deve istanziare un oggetto specifico (definito con una *factory* che definisce il tipo di servizio di naming da utilizzare e con una URL, che definisce dove il servizio è localizzato). Questo può essere realizzato sia passando parametri su linea di comando per settare dei parametri, sia nel codice. Per usare i parametri su linea di comando, si deve utilizzare (per il servizio di naming di Corba) il seguente formato:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
                    -Djava.naming.provider.url=iiop://hostname:port server
```

Questi parametri verranno forniti[4] sia alla esecuzione della JVM server che a quella client.

9.4.3 Un esempio in Java RMI-IIOP

Per poter vedere le differenze con Java RMI, scriviamo un esempio del classico programma di HelloWorld che abbiamo incontrato nel paragrafo .

Iniziamo dalla interfaccia che rimane identica e definisce l'unico metodo remoto.

Hello.java

```
1 // Interfaccia Hello per HelloWorld
2 public interface Hello extends java.rmi.Remote {
3     String dimmiQualcosa(String daChi) throws java.rmi.RemoteException;
4 }
```

Fine: Hello.java

Adesso passiamo al server.

HelloImpl.java

[4]Nella pratica, si ponga attenzione alle lettere maiuscole/minuscole: il nome della factory è `CNCtxFactory` con le prime tre lettere maiuscole, oltre alla F di factory.

```
1 import javax.rmi.*;
2 import javax.naming.*;
3 import java.rmi.RemoteException;
4 import java.util.logging.Logger;
5
6 public class HelloImpl extends PortableRemoteObject implements Hello {
7     // Serial UID
8     private static final long serialVersionUID = -4469091140865645865L;
9     // Logger per la classe
10    static Logger logger= Logger.getLogger("global");
11
12    // Costruttore
13    public HelloImpl() throws RemoteException {
14        // vuoto
15    }
16
17    // Metodo remoto dimmiQualcosa
18    public String dimmiQualcosa(String daChi) throws RemoteException {
19        logger.info("Sto salutando "+daChi);
20        return "Ciao!";
21    }
22
23    public static void main(String args[]) {
24        try {
25            logger.info("Creo l'oggetto remoto...");
26            HelloImpl obj = new HelloImpl();
27            logger.info("... ora ne effettuo il rebind...");
28            Context ic = new InitialContext();
29            ic.rebind("HelloServer", obj);
30            logger.info("... Pronto!");
31        } catch (Exception e) {
32            e.printStackTrace();
33        }
34    } // end main
35 } // end classe HelloImpl
```

Fine: HelloImpl.java

Come si può vedere facilmente, le principali differenze stanno nei package utilizzati (linee 1-2) che prevedono di usare `javax.rmi...` invece di `java.rmi....` Poi, linea 6, viene dichiarato un oggetto remoto che deriva da `PortableRemoteObject` ed, infine, la differenza del naming (linee 28-29). Infatti, per poter usare un servizio JNDI, si deve reperire dai parametri dati alla macchina virtuale Java un riferimento ad un oggetto di tipo `Context` che utilizza le informazioni sul servizio richiesto (factory ed indirizzo). Il metodo `rebind()`)dalla sintassi simile al servizio di naming di Java RMI) viene chiamato su questo oggetto e, quindi, non attraverso un metodo statico. Ovviamente, si deve badare a lanciare il server specificando sui parametri le indicazioni al servizio di naming.

Infine, tocca al client, dove le modifiche sono anche esse identificabili in pochi limitati punti del codice.

HelloClient.java

```
1 import javax.rmi.*;
2 import javax.naming.*;
3 import java.util.logging.Logger;
4
5 public class HelloClient {
6     // Logger per la classe
7     static Logger logger= Logger.getLogger("global");
8
9     public static void main(String args[]) {
```



```
10        Context ic;
11        Object obj;
12        Hello server;
13        try {
14            logger.info("Sto cercando l'oggetto remoto...");
15            ic = new InitialContext();
16            obj =  ic.lookup("rmi://localhost/HelloServer");
17            server = (Hello) PortableRemoteObject.narrow(obj, Hello.class);
18            logger.info("... Trovato! Ora invoco il metodo...");
19            String risultato = server.dimmiQualcosa("Pippo");
20            System.out.println("Ricevuto:"+ risultato);
21        } catch (Exception e) {
22            e.printStackTrace();
23        }
24    }// fine main
25 }// fine classe HelloClient
```

Fine: HelloClient.java

Dopo l'uso dei package `javax` alle linee 1-2, la differenza sostanziale sta nel lookup (che viene anche esso fatto usando, alle linee 15-16, un oggetto `Context`) e nella maniera in cui si ottiene un riferimento remoto al server. Il riferimento remoto, infatti, viene ottenuto effettuando un casting che viene fatto non attraverso gli strumenti del linguaggio Java, ma attraverso un metodo diverso, offerto in Java dal metodo `narrow()`, che segue lo standard Corba. Infatti, in Corba viene definita la modalità con cui viene effettuato il casting, in maniera indipendente dai linguaggi, in quanto Corba viene definito anche per linguaggi che non sono orientati agli oggetti (ad esempio il C) e quindi deve fornire un'unica modalità di casting, indipendente dalla piattaforma software (il linguaggio di programmazione) utilizzata.

Va notato che anche il client deve essere lanciato con le informazioni sul servizio di naming JNDI da utilizzare (factory e URL).

A questo punto si devono generare stub e tie con la esecuzione del comando:

```
rmic -iiop HelloImpl
```

che genera i due file `_HelloImpl_Stub.class` e `_HelloImpl_Tie.class` (lo skeleton con la terminologia Corba). Ricordiamo che lo stub deve essere presente sul client e lo skeleton sul server.

Il servizio di naming viene lanciato con il comando

```
orbd -ORBInitialPort 1050
```

che lancia un servizio di naming compatibile con il Corba Commmon Object Service. Va sottolineato il fatto che in `orbd` viene rimossa la limitazione che aveva `rmiregistry`, di non accettare binding di oggetti istanziati su macchine (host fisici) diverse.

Esistono anche altre differenze tra le implementazioni di RMI e di RMI-IIOP. Innanzitutto, non esiste la Distributed Garbage Collection: quando si vuole che un oggetto remoto venga reso disponibile per il Garbage Collector (interno alla JVM) si deve esplicitamente fermare la sua esportazione come oggetto remoto, attraverso la invocazione di `PortableRemoteObject.unexportObject()`. Non viene fornito, per motivi di compatibilità con Corba, alcun servizio di caricamento dinamico dello stub, anche se alcuni fornitori di application server JEE offrono delle funzionalità simili ma, ovviamente, non portabili. Infine, non si possono usare strumenti particolari di Java RMI come, ad esempio, la possibilità di usare socket particolare (ad esempio, che usino Secure Socket Layer) per la comunicazione. Per riassumere, si confrontino gli schemi riassuntivi in Fig. 9.3 e 9.4.

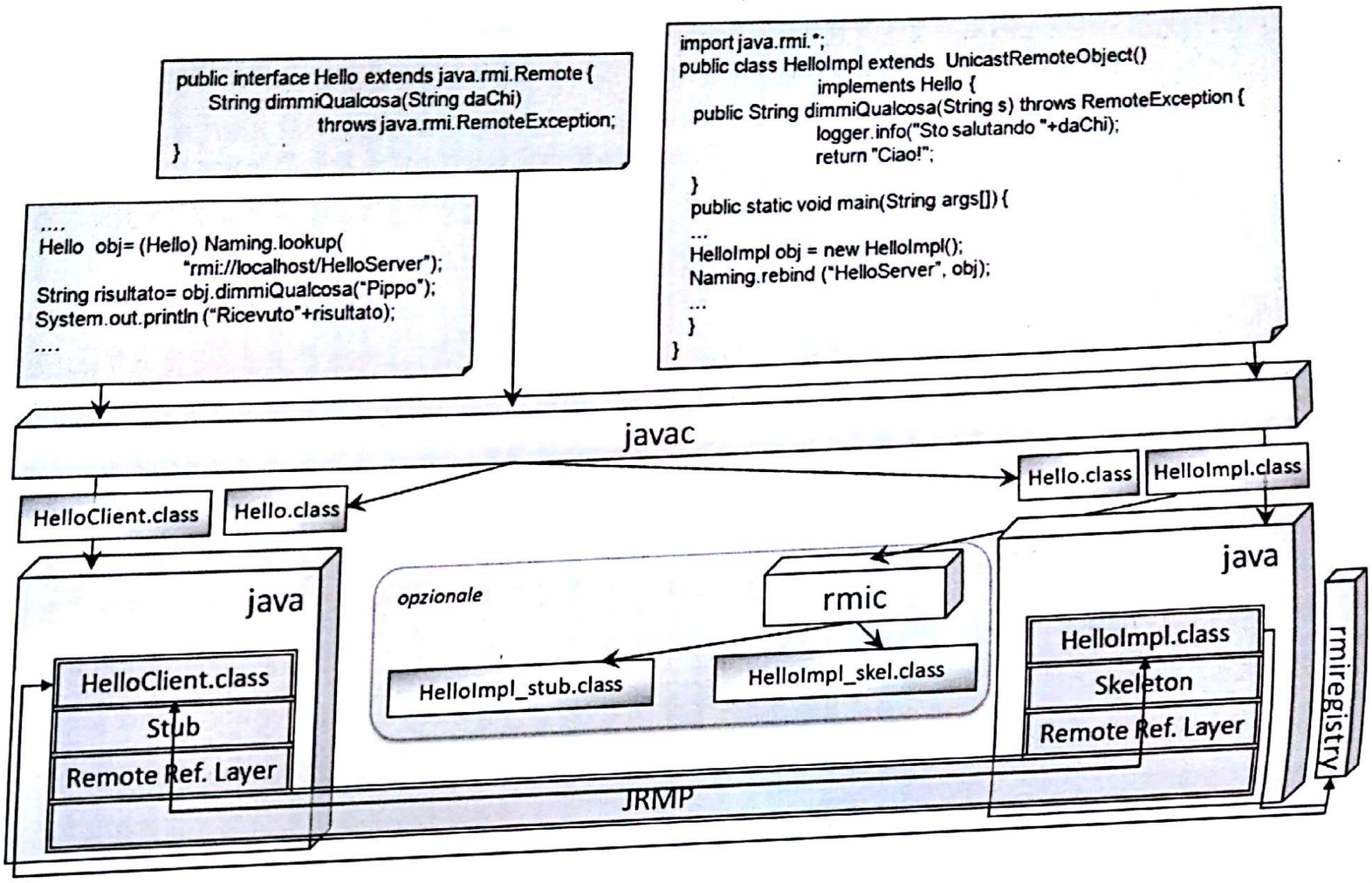

Figura 9.3: *Uno schema di funzionamento di Java RMI.*

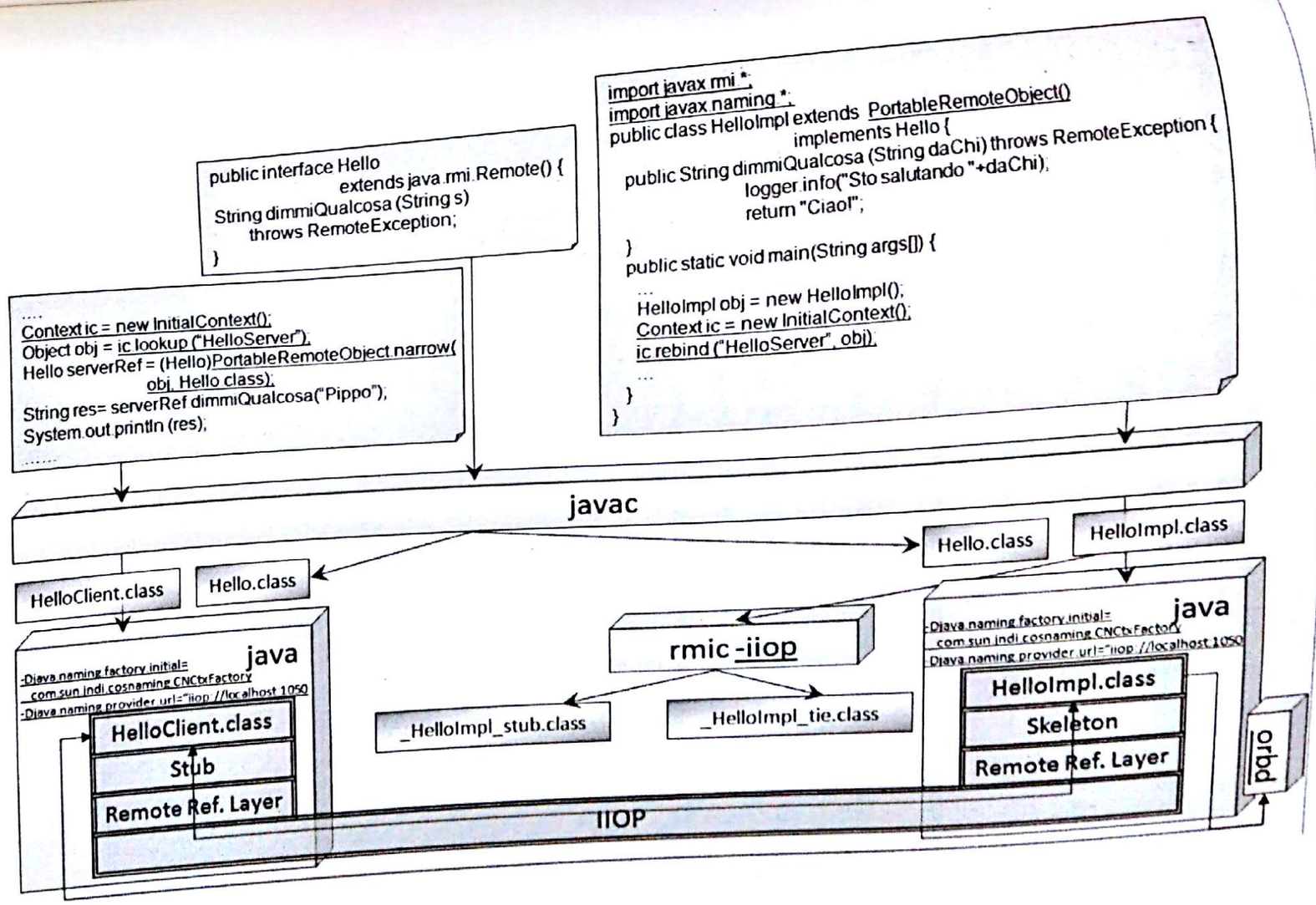

Figura 9.4: *Uno schema di funzionamento di Java RMI-IIOP: sottolineate sono le differenze rispetto ad RMI.*

Note bibliografiche

Le informazioni iniziali su Java Enterprise Edition sono tratte dai capitoli introduttivi del Java Tutorial [26]. Ulteriori informazioni si trovano nella documentazione Java e nei tutorial RMI-IIOP.

Spunti per lo studio individuale

Per l'approfondimento e lo studio individuale, ecco alcuni spunti di riflessione, che possono essere Problemi, *contraddistinti da una* **[P]**, Esercizi, *indicati con una* **[E]**, *oppure* Domande di ricapitolazione, *segnalate da una* **[R]**. *Per ogni problema, esercizio o domanda di ricapitolazione viene indicato anche il livello di difficoltà da Facile *, Medio ** e Difficile ***.*

1. **[R***]** Descrivere le architetture 1-tier, 2-tier, 3 tier ed n-tier ed i relativi pro e contro.
2. **[R**]** Definire i layer software in cui le funzionalità delle applicazioni distribuite possono essere suddivise.
3. **[R***]** Quali sono i compiti "trasversali" che vengono lasciati in JEE all'application server?
4. **[R**]** In che maniera vengono usati i metadati per creare un Enterprise JavaBean?
5. **[R**]** Cosa è IIOP?
6. **[R***]** Quale è la differenza tra GIOP e IIOP?
7. **[R**]** Descrivere Java Naming and Directory Services.
8. **[R***]** Quali sono le differenze tra le Application Programming Interfaces (API) e le Service Provider Interface (SPI) in JNDI?

Bibliografia

[1] Interview with Ann Wollrath about Remote Method Invocation (RMI). http://www.genady.net/rmi/v20/external/chat2002.html.

[2] Java RMI Specification. http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html. Disponibile in locale, scaricando i docs di JDK, sotto technotes/guides/rmi/index.html.

[3] Java Thread Primitive Deprecation. Disponibile in locale, scaricando i docs di JDK, sotto technotes/guides/concurrency/threadPrimitiveDeprecation.html.

[4] Overview of Java SE Security. http://java.sun.com/javase/6/docs/technotes/guides/security/overview/jsoverview.html. Disponibile in locale, scaricando i docs di JDK, sotto technotes/guides/security/overview/jsoverview.html.

[5] The Lifecycle of an RMI Server and Dynamic Class Loading in RMI. Java Developer Connection on Sun Developer Network http://java.sun.com/developer/JDCTechTips/2001/tt0227.html.

[6] Marko Boger. *Java in Distributed Systems: Concurrency,Distribution,and Persistence*. John Wiley \& Sons, Inc., 2001.

[7] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems. Concepts and design*. Addison Wesley, third edition, 2001. http://www.dck3.net.

[8] Wolfgang Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000. http://www.distributed-objects.com.

[9] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. The impact of research on the development of middleware technology. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–48, 2008.

[10] J. S. Fritzinger and M. Mueller. Java Security. Technical report, Sun Microsystems, Inc., 1996.

[11] Greg Gagne. To java.net and beyond: teaching networking concepts using the java networking api. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 406–410, New York, NY, USA, 2002. ACM.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[13] Li Gong. Java 2 platform security architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html. Disponibile in locale, scaricando i docs di JDK, sotto technotes/guides/security/spec/security-spec.doc.html.

[14] James Gosling and Henry McGilton. Java Language Environment. http://java.sun.com/docs/white/langenv/Security.doc.html.

[15] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml,Volume 1*. John Wiley \& Sons, Inc., 2002.

[16] William Grosso. Java RMI. Chapter 10: Serialization. http://oreilly.com/catalog/javarmi/chapter/ch10.html.

[17] William Grosso. RMI, Dynamic Proxies, and the Evolution of Deployment. Java.net, 1/6/2004, http://today.java.net/pub/a/today/2004/06/01/RMI.html.

[18] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., 2001.

[19] Michi Henning. Another note on distributed computing. http://zeroc.com/blogs/michi/2008/07/17/another-note-on-distributed-computing/.

[20] Michi Henning. The rise and fall of CORBA. *Commun. ACM*, 51(8):52–57, 2008.

[21] Mark A. Holliday, J. Traynham Houston, and E. Matthew Jones. From sockets and rmi to web services. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2008, Portland, OR, USA, March 12-15, 2008*, volume 40, pages 236–240, New York, NY, USA, 2008. ACM.

[22] ISO/IEC. Information technology – open distributed processing – reference model: Architecture (iso/iec 10746-3:1996). http://standards.iso.org/ittf/PubliclyAvailableStandards/s020697_ISO_IEC_10746-3_1996(E).zip, 1996.

[23] ISO/IEC. Information technology – open distributed processing – reference model: Foundations (iso/iec 10746-2:1996). http://standards.iso.org/ittf/PubliclyAvailableStandards/s018836_ISO_IEC_10746-2_1996(E).zip, 1996.

[24] ISO/IEC. Information technology – open distributed processing – reference model: Architectural semantics (iso/iec 10746-4:1998). http://standards.iso.org/ittf/PubliclyAvailableStandards/c020698_ISO_IEC_10746-4_1998(E).zip, 1998.

[25] ISO/IEC. Information technology – open distributed processing – reference model: Overview (iso/iec 10746-1:1998). http://standards.iso.org/ittf/PubliclyAvailableStandards/c020696_ISO_IEC_10746-1_1998(E).zip, 1998.

[26] E. Jendrock, D. Carson, I. Evans, D. Gollapudi, K. Haase, and C.Srivathsa. The Java Enterprise Tutorial . Java website, http://java.sun.com/javaee/6/docs/tutorial/doc/.

[27] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Sun Microsystems, Inc., 1994.

[28] D. Krieger and R.M. Adler. The emergence of distributed component platforms. *Computer*, 31(3):43–53, 1998.

[29] Leslie Lamport. Mail originale sulla definizione di Sistema Distribuito. http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt.

[30] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *ACM SIGPLAN Notices*, volume 33, pages 36–44, New York, NY, USA, 1998. ACM.

[31] Sun Microsystems. Java Security Overview. http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf, April 2005.

[32] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, third edition, 2001.

[33] Roger Riggs, Jim Waldo, Ann Wollrath, and Sun Microsystems Inc. Pickling State in the Java System. In *In Proceedings of the USENIX 1996 conference on Object-Oriented Technologies*, 1996.

[34] Arnon Rotem-Gal-Oz. Fallacies of Distributed Computing Explained. http://www.rgoarchitects.com/Files/fallacies.pdf.

[35] Vincent Ryan, Scott Seligman, and Rosanna Lee. RFC 2713 - Schema for Representing Java Objects in an LDAP Directory. http://tools.ietf.org/html/rfc2713, October 1999.

[36] D.C. Schmidt. Distributed object computing. *Communications Magazine, IEEE*, 35(2):42–44, 1997.

[37] Jim Waldo. Jini Network Technology Fulfilling its Promise (interview with Jim Waldo). http://java.sun.com/developer/technicalArticles/Interviews/waldo_qa.html.

[38] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[39] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9, 1996.

[40] Guo-Qing Zhang, Guo-Qiang Zhang, Qing-Feng Yang, Su-Qi Cheng, and Tao Zhou. Evolution of the internet and its cores. *New Journal of Physics*, 10(12):1–11, December 2008.

Programmazione con Oggetti Distribuiti: Java RMI
© Copyright 2010 Vittorio Scarano

Responsabile della pubblicazione Vittorio Scarano

Libro pubblicato dall'autore

Stampato in Italia presso Thefactory,
per Gruppo Editoriale L'Espresso S.p.A.

L'autore è un utente del sito

ILMIOLIBRO

Java Remote Method Invocation è una libreria di Java che permette la invocazione remota di metodi, operazione fondamentale per le computazioni basate su oggetti distribuiti.

Questo libro presenta Java RMI insieme al suo contesto di riferimento, il modello ad oggetti distribuiti. Gli obiettivi di questo libro sono due: il primo è quello di fornire allo studente il collegamento tra i tradizionali corsi universitari di programmazione di rete, che sono basati sulla comunicazione su TCP/IP e socket, e i corsi di programmazione di applicazioni distribuite che tipicamente usano strumenti di alto livello come il modello a componenti distribuite (ambienti di Enterprise Computing) oppure il modello orientato ai servizi (ambienti Web Services). Il secondo obiettivo è quello di fornire basi per la programmazione distribuita di oggetti, sviluppando quindi un certo numero di esempi pratici che guideranno lo studente, a partire dalla implementazione con i socket, a realizzare semplici applicazioni con oggetti distribuiti.

Il libro viene utilizzato come libro di testo per il corso di Programmazione Distribuita al Corso di Laurea in Informatica (I livello) della Università degli Studi di Salerno.

L'autore devolve interamente i propri proventi in favore dei progetti di Emergency.