



UNIVERSITÀ DEGLI STUDI DI FIRENZE  
Facoltà di Scienze Matematiche, Fisiche e Naturali

---

Corso di Laurea in  
SCIENZE DELL'INFORMAZIONE

# Una infrastruttura per l'interoperabilità e lo scambio di oggetti in rete

Tesi di Laurea di  
Daniele Falassi

Relatore:

Prof. Rocco De Nicola

Correlatore:

Dott. Michele Loreti

---

Anno Accademico 2001/02

# Indice

<b>1</b>	<b>Introduzione</b>	<b>5</b>
1.1	Interoperabilità . . . . .	6
1.2	Contenuto della Tesi . . . . .	8
1.3	Struttura della tesi . . . . .	11
<b>2</b>	<b>Interoperabilità Software</b>	<b>13</b>
2.1	Problematiche di interoperabilità . . . . .	15
2.2	Framework per l'interoperabilità . . . . .	16
2.2.1	CORBA . . . . .	16
2.2.2	COM e DCOM . . . . .	17
2.2.3	L'infrastruttura Java . . . . .	18
2.3	Le ragioni per un'infrastruttura diversa . . . . .	19
<b>3</b>	<b>Linguaggi per l'infrastruttura</b>	<b>21</b>
3.1	Caratteristiche dei linguaggi per l'infrastruttura . . . . .	21
3.2	Java e C#: due linguaggi per l'infrastruttura . . . . .	24
3.2.1	Java . . . . .	24
3.2.2	Microsoft .NET e C# . . . . .	25
3.2.3	Brevi considerazioni sull'implementazione del paradigma ad oggetti di C# e Java . . . . .	28

---

3.2.4	Organizzazione delle librerie . . . . .	29
3.3	Reflection . . . . .	33
3.3.1	Java . . . . .	33
3.3.2	C# . . . . .	35
3.4	Meccanismo di Serializzazione . . . . .	36
3.4.1	C# . . . . .	36
3.4.2	Java . . . . .	38
3.5	Caricamento dinamico del codice . . . . .	39
3.5.1	Java . . . . .	39
3.5.2	C# . . . . .	40
3.6	Multithreading . . . . .	41
3.6.1	Java . . . . .	41
3.6.2	C# . . . . .	42
3.7	Programmazione di rete . . . . .	43
3.7.1	Java . . . . .	43
3.7.2	C# . . . . .	44
<b>4</b>	<b>Un linguaggio per lo scambio dei dati:XML</b>	<b>46</b>
4.1	Introduzione a XML . . . . .	46
4.1.1	Utilità di XML . . . . .	47
4.1.2	Regole sintattiche . . . . .	48
4.1.3	Namespaces . . . . .	49
4.1.4	Semantica di un documento XML: DTD e XSD . . . . .	51
4.2	SOAP . . . . .	52
4.2.1	Specifiche SOAP . . . . .	52
4.3	XML, SOAP e la comunicazione tra Java e C# . . . . .	54
4.3.1	La scelta di XML . . . . .	54
4.4	Gestione di XML in Java e C# . . . . .	55

---

4.4.1	Una premessa sui tipi di dati . . . . .	55
4.5	L'XmlSerializer di C# . . . . .	56
4.6	Un serializzatore personalizzato per Java: <i>XmlSerializer</i> . . . . .	58
4.6.1	Dinamicità nella serializzazione . . . . .	61
<b>5</b>	<b>Xml e interoperabilità tra Java e C#: l'infrastruttura</b>	<b>62</b>
5.1	Panoramica generale sull'infrastruttura . . . . .	63
5.2	Funzionamento dell'infrastruttura . . . . .	65
5.3	L'interfaccia ISendable . . . . .	67
5.3.1	Versione C# . . . . .	68
5.3.2	Versione Java . . . . .	69
5.4	L'XmlServer . . . . .	70
5.4.1	Dettagli implementativi . . . . .	71
5.5	KSender . . . . .	75
5.5.1	Pack . . . . .	79
5.5.2	Unpack . . . . .	81
5.5.3	Il metodo request . . . . .	83
5.5.4	Il metodo record . . . . .	84
5.6	I server di classi . . . . .	85
<b>6</b>	<b>Interoperabilità e mobilità: il framework FAL</b>	<b>86</b>
6.1	FAL . . . . .	87
6.2	Descrizione del sistema . . . . .	87
6.3	Dettagli implementativi . . . . .	89
6.3.1	La classe Agent . . . . .	89
6.3.2	La classe Node . . . . .	90
6.3.3	La classe Locality . . . . .	93
6.3.4	La classe Channel . . . . .	93

<b>7</b>	<b>Conclusioni</b>	<b>95</b>
7.1	Limiti . . . . .	96
7.2	Possibili Estensioni . . . . .	96

# Capitolo 1

## Introduzione

Un tempo la connettività di rete era solo uno strumento di supporto opzionale e costoso per un tipo di utenza evoluta e specializzata. Infatti la connessione alla rete era prevalentemente utilizzata da enti universitari, grosse aziende o enti governativi.

Nell'ultimo decennio le cose sono cambiate drasticamente. Accanto al fenomeno Internet, i cui risultati sono sotto gli occhi di tutti, anche altri tipi di reti, come quelle aziendali hanno conosciuto un'enorme evoluzione. Questa evoluzione è stata principalmente in termini di una maggiore *semplicità d'utilizzo* e di un miglioramento drastico delle *performance* di comunicazione.

Come conseguenza del primo aspetto si è avuto l'incremento esponenziale dell'utenza della rete: persone di ogni estrazione sociale e culturale sono oggi in grado di navigare su Internet, anche solo in modo approssimativo. Molti nuovi tipologie di utenti sono oggi presenti sulla rete, ognuna con le sue esigenze e necessità. Allo stesso tempo anche le grosse realtà, che già facevano uso della rete anche in prima del boom degli anni novanta, hanno iniziato a manifestare l'esigenza di sfruttare la "connettività" stessa in modo più proficuo. Si è vista, ad esempio, la nascita delle cosiddette applicazioni

Business-to-Business, tramite le quali transazioni via rete anche piuttosto complesse vengono completamente automatizzate, con notevole risparmio di tempo e risorse per le aziende stesse. Ne è un esempio Biztalk[23].

L'incremento dell'utenza ha comportato un ampliamento delle *dimensioni* e quindi un aumento del *traffico* all'interno della rete. Questo ha comportato la necessità di migliorare le *performance* delle connessioni di rete, sia da un punto di vista hardware che software. Accanto all'ormai consueta evoluzione dell'hardware con la nascita di nuove tecnologie, come le reti *wireless* o le connessioni *firewire*, si è assistito anche all'evoluzione dei meccanismi di progettazione software. Ormai, i programmi non sono più pensati —e utilizzati— per un'esecuzione in locale ma la loro computazione coinvolge anche la rete e le architetture eterogenee che la compongono. È in questo ambito di ricerca, ad esempio, che nasce il concetto di *computazione mobile*, ovvero la possibilità per un programma di migrare durante la sua computazione attraverso piattaforme diverse.

## 1.1 Interoperabilità

Come si è già accennato in precedenza la rete non è cresciuta solo di dimensioni ma si sono diversificati enormemente anche i tipi di utenza, e dunque sulla rete interagiscono molti linguaggi e sistemi operativi diversi. Di conseguenza è sempre più difficile, oggi, utilizzare del software che non possa poi essere sfruttato in un contesto così eterogeneo. Ciò significa anche che questo software deve poter sfruttare le nuove tecnologie che nascono e vengono utilizzate sulla rete.

Tuttavia aggiornare un sistema pre-esistente al fine di sfruttare nuove tecnologie rischia di essere un processo molto costoso. Infatti spesso le nuove

tecnologie sono meglio sfruttate dai nuovi linguaggi e riscrivere una intera applicazione da capo in un linguaggio nuovo può essere impraticabile.

Per ovviare a queste problematiche si è affermata la necessità di avere un meccanismo che consenta a linguaggi diversi di scambiare dati strutturati, per sfruttarne le diverse caratteristiche, riducendo così al minimo le necessarie modifiche sulle applicazioni pre-esistenti.

Questa interoperabilità, però non si dovrebbe limitare alla possibilità di scambiare semplici dati, ma dovrebbe fornire anche funzionalità per lo scambio di codice. Questa esigenza è estremizzata nell'ambito della computazione mobile, esposto in precedenza. Infatti, affinché un programma possa effettivamente migrare con tutto il suo codice, i suoi dati e il suo contesto di esecuzione, occorre che sia possibile scambiare informazione complessa — ovvero interi programmi — tra sistemi eterogenei.

Le problematiche di interoperabilità si possono riassumere, quindi, nella necessità di far cooperare tra loro, a basso livello, applicazioni scritte in linguaggi diversi e in esecuzione su sistemi operativi eterogenei.

Per risolvere queste problematiche si sono proposte due classi di soluzioni, di genere diverso: da un lato architetture per l'interoperabilità, dall'altro l'utilizzo di linguaggi e protocolli aperti e quindi accessibili a più piattaforme.

Possono essere citati ad esempio, CORBA[18] e DCOM[20]: due architetture che consentono un certo livello di interoperabilità fra applicazioni distribuite sulla rete. Entrambe consentono ad una applicazione di invocare una procedura presente su un'altra macchina tramite interfacce standardizzate. Questo meccanismo consente un buon livello di interoperabilità tra linguaggi e sistemi diversi, ma, ad esempio, non permette la mobilità del codice. Questa è invece possibile sfruttando ad esempio l'infrastruttura di Java[3] poiché permette di eseguire il medesimo codice su due piattaforme distinte ma aventi

entrambe l'ambiente di esecuzione di Java, risolvendo l'interoperabilità di sistemi ma costringendo all'uso di un singolo linguaggio. Chiaramente utilizzare un singolo linguaggio non risolve i problemi descritti in precedenza. Questo perché, come si è accennato all'inizio di questa sezione, è importante che un software possa essere in condizione di poter interagire col maggior numero di applicazioni, indipendentemente dal linguaggio di programmazione.

Si sono allora iniziati a sviluppare anche linguaggi per lo scambio di messaggi fra applicazioni sul web che fossero indipendenti da una infrastruttura specifica.

Negli ultimi anni si è affermato il linguaggio XML(eXtensible Markup Language[1]), sviluppato come estensione del noto linguaggio per il web HTML[1]. Con XML, oltre che costruire messaggi, è anche possibile descrivere dei dati in un modo formale e standardizzato. Da XML è stato poi sviluppato il *Simple Object Access Protocol* (SOAP)[1], pensato, dai suoi autori, per scambiare messaggi tra applicazioni, in un modo più semplice rispetto a XML. L'indipendenza da una specifica infrastruttura permette a SOAP e XML di essere utilizzati in svariati contesti, come ad esempio nei servizi web realizzati tramite diversi framework, come Sun One[16] di Sun Microsystems e .NET[4] di Microsoft.

## 1.2 Contenuto della Tesi

Tutte le soluzioni a cui si è accennato risolvono però solo parzialmente la problematica per l'interoperabilità che si è esposta. Ad esempio CORBA consente l'interoperabilità fra linguaggi ma non la mobilità del codice.

L'obiettivo della tesi è quello di fornire un'infrastruttura che consenta di scambiare istanze di oggetti attraverso la rete, utilizzando linguaggi dif-

ferenti. Questo è un primo passo nella realizzazione di un meccanismo di interoperabilità diverso da quelli già esistenti. Questo permetterebbe infatti lo scambio del codice (che non avviene su infrastrutture come CORBA e DCOM) e permetterebbe a framework che già forniscono primitive per lo scambio di codice (come ad esempio Java), di interagire a basso livello con applicazioni scritte in linguaggi differenti.

Vediamo quindi lo scenario che vogliamo andare ad affrontare. Osserviamo la figura 1.1. In essa sono date due macchine A e B collegate in rete, due programmi PL1 e PL2 scritti rispettivamente con un linguaggio L1 e con un linguaggio L2. I linguaggi L1 e L2 sono orientati a oggetti. Sulla macchina A è presente il programma  $P_{L1}$  che utilizza al suo interno un oggetto  $O_{L1}$ . Si vuole trasferire tale oggetto sul programma  $P_{L2}$  in esecuzione sulla macchina B. Nel nuovo contesto l'oggetto  $O_{L1}$  diventerà un oggetto  $O_{L2}$ , equivalente al precedente ma definito nel linguaggio L2.

Figura 1.1: Scenario

Sulla base dello scenario appena descritto si propone una infrastruttura che ha carattere generale, poiché, in linea di principio, essa potrà essere utilizzata da un qualunque linguaggio orientato a oggetti. Nella tesi l'infrastruttura proposta viene realizzata utilizzando i linguaggi Java e C#[15].

La figura 1.2 mostra le diverse parti di cui è composta l'infrastruttura. Abbiamo anzitutto i programmi  $P_{L_1}$  e  $P_{L_2}$  che devono scambiarsi le istanze di oggetti. Essi sono scritti in due linguaggi diversi  $L_1$  e  $L_2$  e possono comunicare fra loro via rete. Le Application Program Interfaces (API) dell'infrastruttura per il linguaggio  $L_1$  rappresentano le classi e i metodi tramite i quali il programma  $P_{L_1}$  può utilizzare l'infrastruttura. Analogamente esistono delle API per il linguaggio  $L_2$  che vengono sfruttate da  $P_{L_2}$  allo stesso modo. Vi sono poi un numero variabile di *server di classi* per il linguaggio  $L_1$  che consentono ad un programma di caricare le definizioni delle classi necessarie ad istanziare un oggetto richiesto. Chiaramente ogni classe presente sui server di classi per il linguaggio  $L_1$  deve avere una classe corrispondente nel linguaggio  $L_2$ . Infine esiste un *server di informazioni* che consente ad un programma di individuare da quale server di classi è possibile scaricare una data definizione. La comunicazione tra i due programmi  $P_{L_1}$  e  $P_{L_2}$  e tra l'infrastruttura e il server di informazioni è realizzata tramite messaggi in formato XML.

Veniamo ora a descrivere il funzionamento dell'infrastruttura. Quando il programma  $P_{L_2}$  vuole inviare un oggetto  $O_{L_2}$  al programma  $P_{L_1}$ , lo invia alle API dell'infrastruttura che ne codificano lo stato in XML.

$P_{L_2}$  invia quindi l'oggetto così codificato a  $P_{L_1}$ , che passa direttamente il messaggio ricevuto all'API. Quest'ultime esaminano il messaggio e, tramite il server di informazioni e i server di classi, recupera la classe che nel nuovo linguaggio  $L_1$  corrisponde a quella dell'oggetto  $O_{L_2}$  di partenza. Infine, l'infrastruttura costruisce un'oggetto  $O_{L_1}$  nello stato individuato dal messaggio XML ricevuto e lo restituisce al programma  $P_{L_1}$ .

Nella tesi si è utilizzata l'infrastruttura per costruire un semplice linguaggio per agenti mobili (FAL). L'esempio permette di apprezzare i vantaggi

Figura 1.2: Schema dell'infrastruttura

dell'infrastruttura in un simile contesto.

### **1.3 Struttura della tesi**

Nel capitolo 2 sono descritte alcune delle strutture per l'interoperabilità più diffuse per contestualizzare il lavoro della tesi. Nel capitolo 3 vengono illustrate alcune caratteristiche utili ai fini della costruzione dell'infrastruttura e viene motivata la nostra scelta di utilizzare Java e C#. Nel capitolo 4 viene introdotto il linguaggio XML, utilizzato per la comunicazione dello stato degli oggetti scambiati fra le applicazioni. Nel Capitolo 5 viene quindi presentata

in dettaglio l'infrastruttura nella sua implementazione Java/C#. Nel capitolo 6, per mostrare l'utilizzo dell'infrastruttura presentiamo FAL, un semplice framework per la programmazione di applicazioni distribuite con mobilità.

## Capitolo 2

# Interoperabilità Software

Fino a una ventina di anni fa l'interazione fra diversi ambienti software era pressoché inesistente. La rete Internet era ancora in uno stato primitivo, utilizzata principalmente da enti governativi o universitari. Il solo modo che l'utente medio aveva per trasferire dati fra due macchine era tramite un supporto di registrazione magnetico come ad esempio un floppy. La rete stessa aveva prestazioni molto inferiori a quelle di oggi. In questo contesto parlare di interoperabilità fra due applicazioni software presenti su due sistemi diversi era praticamente impossibile. Lo scenario in cui ci si trovava ad operare era quello di centri di calcolo con computer enormi e costosissimi dotati di sistemi operativi e applicazioni proprietarie. In questa situazione se per esempio si voleva far comunicare due applicazioni, delle quali la prima dava come output alcuni dati per l'input della seconda, si doveva stampare l'output della prima su carta, e successivamente inserire a mano sul terminale della macchina su cui era in esecuzione la seconda applicazione i dati di input che si potevano estrapolare dall'output della prima. era cosa comune utilizzare meccanismi proprietari sia per la registrazione dei dati che per la comunicazione interna ad una rete locale, senza che questo fosse considerato un problema.

Il primo vero esempio di interoperabilità può essere, considerato Postscript [12], un linguaggio per la descrizione di documenti interpretabile da apposite stampanti. Si tratta di un vero e proprio linguaggio di programmazione. Una volta che un documento Postscript viene ricevuto da una stampante i comandi in esso contenuti vengono eseguiti e come risultato si ottiene la stampa del documento. In questo modo se avevo una applicazione che stampava l'output in formato Postscript, potevo utilizzare un qualunque modello di stampante in grado di interpretarla, senza essere più vincolato ad utilizzare le sole stampanti compatibili con il mio sistema operativo. Diventa così possibile stampare quello stesso documento su un sistema operativo diverso ma comunque in grado di stampare documenti postscript.

Non è un caso che questo sia stato l'ambito per la nascita di un primo meccanismo di interoperabilità. La stampante era — ed è tutt'oggi — uno strumento indispensabile per un ufficio di qualunque tipo. È presente ovunque e ogni sistema — sia dal punto di vista hardware che software — è in grado di operare con una periferica di stampa. Anche la stessa necessità di stampare è comune a qualunque sistema di elaborazione dati. Avendo tutto questo insieme di esigenze e strumentazioni comuni, la nascita di meccanismi di funzionamento standard, e quindi in grado di interagire fra loro, è una conseguenza abbastanza logica.

Tutte queste caratteristiche sono oggi possedute anche dalla connettività di rete — e da Internet. Infatti un qualsiasi computer che possiamo acquistare oggi ha la possibilità di connettersi alla rete in modo semplice e a basso costo. Anche i sistemi operativi ormai considerano la rete il principale strumento per la comunicazione dei dati, fornendo le funzionalità di accesso come parti integranti del sistema stesso.

Ci troviamo dunque in una situazione in cui è estremamente semplice ed

economico realizzare una piccola LAN anche in casa. Le nuove tecnologie, come le reti wireless o la telefonia con WAP[24](Wireless Application Protocol) e GPRS[11](General Packet Radio Services) consentono la connessione alla rete praticamente da *qualunque luogo*.

## 2.1 Problematiche di interoperabilità

Il forte processo di integrazione dell'hardware ha finito, inevitabilmente, con interessare anche il software. Le problematiche da affrontare sono moltissime, e non è possibile farne un sunto completo in questa sede. Verranno descritti brevemente solo alcuni esempi che riteniamo più significativi per la contestualizzazione della tesi.

Molte problematiche di interoperabilità nascono, ad esempio, dalla gestione di una rete aziendale. Spesso le grosse aziende acquisiscono reti appartenenti a ditte precedenti, oppure grazie alle nuove tecnologie più potenti ed economiche riescono a mettere in comunicazione reti appartenenti all'azienda ma che erano storicamente separate. O ancora, si cerca di sfruttare una nuova tecnologia integrandola all'interno. In tutti questi casi le necessità sono molte: occorre riutilizzare il software già presente; si vuole che alcune applicazioni siano accessibili da ogni nodo del sistema aziendale. Inoltre, volendo sfruttare una nuova tecnologia, si richiede anche che sia possibile estendere con nuove funzionalità le applicazioni già presenti nel sistema. Teoricamente, è sempre possibile adottare una scelta drastica: riscrivere tutto il sistema. Di fatto questa scelta, quando attuabile, comporta dei costi elevatissimi.

Quindi, se si vuole riutilizzare il materiale già presente (composto da vecchi database, vecchie applicazioni realizzate da terzi, reti con protocolli diversi, terminali e pc con sistemi operativi diversi), sono necessari degli

opportuni meccanismi di interoperabilità software. Per riutilizzare vecchi database occorre ad esempio avere uno stesso standard per la descrizione dei dati. Inoltre per estendere le funzionalità di applicazioni pre-esistenti e sconosciute, sono necessari, ad esempio, dei meccanismi di interoperabilità tra i diversi linguaggi che utilizzo per realizzare una applicazione.

## 2.2 Framework per l'interoperabilità

Tutte le problematiche esposte sin qui hanno portato alla nascita di numerose soluzioni. Fino a pochi anni fa, l'unico meccanismo per l'interoperabilità utilizzato era quello delle *Remote Procedure Calls*(RPC) dette anche *chiamate a procedura remota*. Tramite di esse un processo client è in grado di richiedere l'esecuzione di una procedura su una macchina server, remota. Successivamente, con lo sviluppo delle tecnologie per la mobilità del codice, si sono sviluppati anche altri paradigmi come quello degli *Agenti Mobili* nel quale un processo (chiamato Agente) è in grado di sospendere la propria esecuzione e migrare su un nuovo sistema, riprendendo lì la propria esecuzione autonomamente[17].

Nel seguito sono presentati alcuni dei principali framework per l'interoperabilità che esistono oggi, e che si avvalgono anche dei meccanismi appena citati.

### 2.2.1 CORBA

Un prima infrastruttura per l'interoperabilità che andiamo a illustrare è CORBA(*Common Object Request Broker Architecture*). Questa architettura (mostrata in figura 2.1) consente di far comunicare fra loro oggetti scritti in linguaggi diversi sulla rete. La comunicazione tra i diversi oggetti avviene

tramite l'uso di interfacce scritte in un linguaggio comune —Interface Definition Language o IDL[18]— e tramite un apposito server centralizzato —l'*Object Request Broker* (ORB). L'ORB gestisce i messaggi tra le diverse interfacce. Una applicazione che voglia richiamare un metodo su un dato oggetto che si trova sulla rete CORBA invia un messaggio (che è praticamente una RPC) all'ORB che provvederà a recapitarlo al destinatario opportuno in un formato ad esso comprensibile, restituendo infine la risposta ricevuta all'oggetto di partenza. In sostanza il meccanismo di comunicazione è una sorta di chiamata a procedura remota, con la differenza che tutte le chiamate sono fatte passando dal server ORB centralizzato, per cui non è necessario conoscere la locazione esatta dell'oggetto che si intende utilizzare. Tramite una architettura come questa è possibile, ad esempio avere un oggetto **O** in esecuzione su una macchina remota e poterne invocare i metodi in modo trasparente, come se l'oggetto fosse in locale.

Figura 2.1: Architettura CORBA

### 2.2.2 COM e DCOM

Una architettura simile a CORBA è il Component Object Model (COM), di cui DCOM (Distributed COM) è l'estensione per la programmazione distribuita. Questa architettura, sviluppata da Microsoft, consente di realizzare

applicazioni costituite da componenti che comunicano fra loro tramite apposite interfacce, indipendenti dal linguaggio con cui la particolare componente viene scritta. Utilizzando COM è possibile costruire delle applicazioni strutturate come quelle in figura 2.2. In questo caso l'applicazione suddivisa nelle componenti A, B, C, D. Tuttavia le componenti C e D si trovano su macchine remote. Il meccanismo di interoperabilità di DCOM utilizza delle componenti C' e D', che, rispettivamente, si occupano di comunicare con le componenti C e D. Questo meccanismo dà l'illusione che C e D siano locali.

Figura 2.2: Una applicazione DCOM

Come si può vedere DCOM non utilizza un server centralizzato come CORBA. Tuttavia l'intera infrastruttura lavora soltanto in ambiente Windows. Questo rappresenta il limite vero per COM e DCOM in un contesto di interoperabilità più generale.

### 2.2.3 L'infrastruttura Java

Nei due esempi analizzati sin qui si è visto come si possa riuscire a far dialogare delle applicazioni eterogenee sulla rete. Tuttavia né CORBA né DCOM

permettono la mobilità del codice. Le informazioni che vengono scambiate sono essenzialmente i parametri per poter richiamare una funzione remota, e il risultato di quest'ultima. Un meccanismo di interoperabilità diverso, nel quale è effettivamente possibile la migrazione del codice è quello dell'infrastruttura del linguaggio Java.

L'idea dell'infrastruttura è quella di avere su ciascuna macchina, dove vogliamo far migrare il codice, una macchina virtuale — chiamata Java Virtual Machine — che fa da intermediario tra il sistema operativo vero e proprio e l'applicazione. Di fatto l'applicazione compila in un linguaggio intermedio — il `bytecode` — che la macchina virtuale poi interpreta e traduce in comandi per il particolare sistema operativo su cui è in esecuzione.

Questo meccanismo — assieme a molte altre caratteristiche importanti del linguaggio — si è rivelato ottimo per realizzare ambienti di programmazione per agenti mobili. Ad esempio Klava[7], Mole [21], Concordia[25], Voyager[10] o Aglets[13] sono tutti ambienti per agenti mobili sviluppati con Java.

Il linguaggio Java sarà oggetto di una trattazione più completa nel capitolo 3.

Tramite Java è dunque possibile attuare un meccanismo di interoperabilità tra sistemi operativi anche diversi, rimanendo, tuttavia legati all'uso di un unico linguaggio di programmazione.

## 2.3 Le ragioni per un'infrastruttura diversa

Gli strumenti software che abbiamo illustrato consentono di attuare numerose tecniche per l'interoperabilità. Ciononostante quando uno strumento consente l'interazione tra linguaggi diversi — CORBA —, non permette la mo-

bilità del codice. D'altra parte laddove, quest'ultima viene permessa — Java — non si consente l'utilizzo di linguaggi diversi.

É quindi interessante provare a realizzare una infrastruttura che consenta sia un'interoperabilità tra linguaggi, sia la mobilità del codice —, e dunque, garantisca un buon grado di interoperabilità tra sistemi operativi eterogenei.

# Capitolo 3

## Linguaggi per l'infrastruttura

In questo capitolo si descrivono i linguaggi utilizzati per l'implementazione dell'infrastruttura oggetto della tesi.

Poiché l'infrastruttura deve permettere anche la mobilità del codice, la scelta dei linguaggi è stata limitata ai soli linguaggi orientati ad oggetti. Questo perché l'oggetto racchiude in se sia il codice che i dati ad esso relativi, individuando così una unità computazionale a se stante, in grado di muoversi in modo indipendente.

### 3.1 Caratteristiche dei linguaggi per l'infrastruttura

Per prima cosa vengono illustrate le caratteristiche che un linguaggio dovrebbe supportare per poter essere utilizzato nel contesto del lavoro della tesi. Successivamente illustreremo come i due linguaggi scelti le implementano. Le caratteristiche riguardano sia la parte per l'interoperabilità che per la mobilità del codice.

**Reflection** La proprietà di Reflection consiste nel poter ottenere informazioni sul codice che è in esecuzione. È possibile, ad esempio, interrogare un oggetto per conoscerne il tipo a runtime. Ricordiamo che tramite il meccanismo di polimorfismo ad un medesimo oggetto base, cui si può far riferimento nel codice al tempo della compilazione, può corrispondere a runtime una qualunque istanza di una sua classe derivata. Questo meccanismo risulta molto utile per la parte sulla mobilità, poiché posso ricevere codice di cui non conosco nulla, se non che implementa una determinata interfaccia. In tal modo posso verificare se un oggetto ricevuto dalla rete può essere gestito dalla mia applicazione.

**Serializzazione** Tramite il meccanismo di *Serializzazione* è possibile registrare un oggetto su uno stream e poterlo successivamente recuperare tramite il processo inverso, quello di *Deserializzazione*. Questo significa, ad esempio, poterlo registrare su file oppure poterlo inviare sulla rete e recuperarlo da un'altra macchina. Ciò che viene registrato non è semplicemente lo stato dell'oggetto e il suo codice. In realtà poiché un oggetto può contenere altri oggetti al suo interno, la serializzazione produce un grafo di oggetti. Tale processo non è semplice e pertanto è importante che un linguaggio lo supporti di default, poiché la serializzazione è fondamentale per poter inviare e ricevere oggetti tramite la rete.

**Multitasking e Multithreading** Un linguaggio che supporti la gestione del *multitasking* consente di gestire l'esecuzione di più processi contemporaneamente. Questo, nella pratica, significa che tale linguaggio è in grado di eseguire operazioni evolute nella gestione dei processi in esecuzione, come controllarne l'avvio e la terminazione. Il *multithreading* è il multitasking relativo alla gestione dei thread. Un *thread* è una sorta di processo in ese-

cuzione più “leggero” dal punto di vista delle risorse rispetto ad un normale processo. Infatti è possibile mandare in esecuzione più versioni dello stesso codice ciascuno come un thread distinto, ottenendo un risultato analogo a quello del multitasking. Tuttavia tutti i thread in esecuzione condividono la stessa area di memoria, a differenza dei normali processi che utilizzerebbero uno spazio di memoria distinto. Per contro, l'utilizzo dello stesso spazio di memoria comporta un accesso concorrente alle risorse e la necessità dell'uso di primitive di sincronizzazione.

Tutte questi meccanismi sono fondamentali in un contesto di programmazione concorrente come quello in cui si trova ad operare un progetto pensato anche per la mobilità.

**Programmazione di rete** Per realizzare un minimo livello di interoperabilità fra due macchine remote occorre quantomeno che queste possano scambiarsi delle stringhe. Tuttavia l'infrastruttura che abbiamo presentato brevemente nell'introduzione ha la necessità di fare un uso molto intensivo della rete. Questo significa che un linguaggio pensato per realizzarla deve essere in grado di gestire strumenti per la connettività di rete — come i *socket* — in modo semplice per facilitare la costruzione dell'applicazione.

**Caricamento dinamico del codice** Il caricamento dinamico del codice consiste nella capacità di caricare nuove parti di codice durante l'esecuzione di un programma. Tali parti di codice possono anche essere sconosciute nel momento in cui si compila l'applicazione. Questa caratteristica è fondamentale se si vuole lavorare in un contesto di mobilità del codice — si veda in proposito anche [5]. È abbastanza ragionevole supporre che un linguaggio capace di caricare dinamicamente il codice possa gestire anche altre caratteristiche come la serializzazione e la reflection, descritte in precedenza.

## 3.2 Java e C#: due linguaggi per l'infrastruttura

Passiamo ora a descrivere i linguaggi che abbiamo scelto per implementare l'infrastruttura. Verranno presentati brevemente e poi si passerà ad analizzare più in dettaglio come ciascuna delle caratteristiche che abbiamo descritto nella sezione precedente viene poi realizzata.

### 3.2.1 Java

Java è un linguaggio di programmazione sviluppato da Sun Microsystems, che ha riscosso un notevole successo negli ultimi anni, soprattutto per il suo largo utilizzo sul web. Si tratta di un linguaggio orientato a oggetti puro, derivato per certi aspetti da C++, ma che presenta molte innovazioni rispetto ad esso. La principale caratteristica che ha determinato il successo di Java è che un linguaggio *multiplatforma* con un altissimo grado di *portabilità*. È multiplatforma perché un programma Java non viene compilato direttamente in linguaggio macchina ma viene compilato in un linguaggio intermedio detto *bytecode*. Il codice così compilato deve essere poi passato all'interprete chiamato Java Virtual Machine (JVM). Questo interprete provvede ad eseguire il codice. E' quindi sufficiente fornire una JVM per un determinato sistema operativo, per poter eseguire del codice bytecode, anche se questo è stato compilato su un sistema completamente diverso e questo spiega perché abbiamo detto che Java ha un alto grado di portabilità. Spesso, però, le diverse macchine virtuali non producono esattamente gli stessi risultati, a causa delle diverse implementazioni. Questo grosso problema non ha però impedito a Java di diventare un linguaggio enormemente diffuso.

### 3.2.2 Microsoft .NET e C#

Nel 1999 alcune aziende — tra cui Microsoft, Sun, HP, Netscape e IBM — hanno presentato all' ECMA<sup>1</sup> lo standard catalogato col codice TC39/TG3[9]. In tale documento si descrive la **Common Language Infrastructure** (CLI). L'obiettivo di tale infrastruttura è quello di fornire le specifiche per **una piattaforma multilinguaggio**. Il *Common Language Runtime* (CLR) costituisce l'implementazione di Microsoft della CLI. Spesso ci si riferisce ad esso anche come al .NET Framework. In realtà con .NET si intendono un insieme di strumenti diversi per il web di cui, comunque il CLR costituisce una parte fondamentale.

Il meccanismo con cui il CLR realizza una piattaforma multilinguaggio è simile a quello di Java: ogni linguaggio all'interno dell'infrastruttura viene compilato in un linguaggio intermedio il *Microsoft Intermediate Language*(MSIL) che poi può essere interpretato da una macchina virtuale (in questo caso chiamata *Virtual Execution Engine*).

#### Common Language Specifications e Common Type System

Esistono poi delle specifiche a cui un linguaggio (e quindi anche il suo compilatore) che voglia essere utilizzato all'interno del framework deve conformarsi. Tali specifiche, dette **Common Language Specifications**(CLS) costituiscono la vera novità rispetto a Java. Esse sono costituite, da alcune regole sulla struttura del linguaggio —come ad esempio il requisito che il linguaggio sia orientato a oggetti—. Sempre queste specifiche è dato anche l'obbligo di sfruttare un insieme di librerie di classi comuni, che costituiscono il **Common**

---

<sup>1</sup>European Computer Manufacturer Association un ente internazionale fondato nel 1961 dedicato alla standardizzazione dei sistemi informativi e di comunicazione, è ormai esteso anche a società extraeuropee. Sito ufficiale: [www.ecma.ch](http://www.ecma.ch)

**Type System** (CTS). Utilizzando il CTS i linguaggi definiti all'interno di .NET possono scambiarsi le definizioni dei tipi, sfruttando le caratteristiche di diversi linguaggi per la scrittura di una applicazione.

### Il linguaggio C#

Il linguaggio C#, nel contesto della multiplatforma .NET, è un linguaggio pensato per costruire applicazioni per la rete. Questo linguaggio prende molte caratteristiche di altri linguaggi come Java, Visual Basic e C++. È un linguaggio orientato ad oggetti puro, in cui tutte le classi derivano da una medesima classe base *object*, che implementa ereditarietà singola per le classi e multipla per le interfacce. Questo linguaggio possiede alcune caratteristiche peculiari, rispetto ai linguaggi appena citati.

La prima grossa differenza è l'assenza dei tipi primitivi, ovvero di tipi che non sono oggetti (ad esempio `integer` o `char`) presenti invece in C++ e Java. In C#, infatti, questi vengono sostituiti dai *value type*[15] che in pratica corrispondono alle classi wrapper di Java (`Integer`, `Boolean`, ecc). Tramite il meccanismo di *boxing/unboxing*[15, 2] il programmatore può lavorare su di essi come se fossero i vecchi tipi primitivi di Java.

Ecco un esempio di come funziona questo meccanismo:

Somma di due interi (tipi primitivi e oggetti wrapper) in Java:

```
1.int x,y,z; // tipi primitivi
2.Integer a,b,c; // oggetti wrapper
.....
3.z=x+y; //somma tra tipi primitivi
4.a=new Integer(x);
5.b=new Integer(y);
6.c=new Integer(a.getValue()+b.getValue()); //somma tra oggetti wrapper
```

```
7.System.out.println(c.toString());//scrive l'output
Somma tra due oggetti int in C#:
8.int x,y,z;
...
9.z=x+y;//nota che x,y e z sono Oggetti.
//la riga seguente scrive l'output,notare che viene richiamato un metodo
su z
10.Console.WriteLine(z.ToString())
```

É il compilatore C# ad eseguire tutte quelle operazioni che in Java vengono eseguite al rigo 6.

## Gli Attributi

Gli attributi costituiscono una peculiarità di C#, rispetto a Java. Tramite un attributo è possibile associare una particolare meta-informazione ad una classe, ad un campo o anche ad un metodo di una classe. Tale informazione può essere poi recuperata tramite la classe `Type` e il metodo `GetAttributes` a runtime.

La sintassi è la seguente:

```
[<nomeattributo>]
<classe, campo o metodo>
```

Per capire cosa sia un attributo si pensi alla keyword `public`. Con essa si intende dire al compilatore che la classe o il membro che segue la keyword è visibile anche da oggetti esterni alla classe. Questa caratteristica è una *metainformazione* ovvero una informazione sul codice e non un dato del problema risolto dal codice.

Di solito la meta-informazione deve essere registrata su file esterni, che il Runtime non può controllare. Ad esempio si potrebbe voler associare

ad una classe l'URL dove è localizzata la sua documentazione. Una simile informazione non ha nulla a che vedere col funzionamento della classe stessa. La differenza tra registrare questa informazione su un file o piuttosto registrarla nel codice di un attributo è che nel secondo caso questa informazione può essere gestita dal Runtime e la classe stessa diventa maggiormente auto-espressiva.

In C# è possibile creare attributi personalizzati. Tuttavia vi sono numerosi attributi predefiniti. Uno di questi è `[Serializable]`, che indica che quel particolare campo o classe è serializzabile. Di questo attributo si parlerà meglio nella sezione 3.4.1.

**Assembly** Il compilatore C# costruisce degli eseguibili in MSIL, che poi vengono gestiti dall'ambiente runtime del CLR di .NET. Effettivamente il prodotto di una compilazione di codice C# è un *assembly*. Un assembly è un insieme di file che costituiscono una applicazione. In uno dei file che costituiscono l'assembly viene inserito un *manifest* che elenca tutti i file che fanno parte dell'assembly. Gli altri file, che costituiscono l'assembly, possono essere o file risorsa —immagini-suoni, ecc.— oppure *moduli*. Un modulo è in pratica una libreria di codice (con estensione *.netmodule*) che non può essere utilizzata se non viene integrata all'interno di un assembly.

### 3.2.3 Brevi considerazioni sull'implementazione del paradigma ad oggetti di C# e Java

Per quanto detto sin qui i due linguaggi possono apparire piuttosto simili. In effetti il modo con cui entrambi implementano il paradigma orientato a oggetti è molto simile: entrambi utilizzano un modello *object-based* ovvero tutti le classi ereditano da una medesima classe comune. Inoltre in entrambi

i linguaggi non è stata sviluppata l'ereditarietà multipla per le classi, per ragioni di efficienza.

### 3.2.4 Organizzazione delle librerie

Per comprendere la realizzazione dell'infrastruttura, è importante descrivere i meccanismi di gestione delle librerie dei due linguaggi scelti. Questo, ad esempio, al fine di comprendere alcune scelte riguardanti la realizzazione dei server di librerie. Le similitudini fin qui riscontrate fra i due linguaggi scompaiono quando si analizza l'organizzazione delle librerie.

#### Java

L'output di una compilazione in Java è un file `class` scritto in *bytecode*. Per essere più precisi viene prodotto un file `class` per ogni classe presente nel file da compilare. In realtà è possibile gestire la creazione dei file `class` tramite i concetti di *package* e *CLASSPATH*.

**I package** I package sono delle raccolte di classi. È possibile dichiarare una classe come appartenente ad una di queste raccolte semplicemente utilizzando la parola chiave `package` seguita dal nome della raccolta nel file dove la classe è definita. Dopo la compilazione della classe, il file `class` relativo ad essa si troverà in una directory chiamata come il package che si era definito. In questo modo è anche possibile costruire raccolte di package, le cui classi verranno inserite in opportune sottodirectory. Ad esempio la seguente dichiarazione

```
package pippo.pluto.topolino;  
crea l'albero di directory:  
\pippo\pluto\topolino
```

con i file class relativi al package nell'ultima directory (topolino). Per poter accedere ad un package si dovrà utilizzare la parola chiave `import` seguita dal nome del package. Inoltre se si vuole fare riferimento ad un package interno oppure ad una classe specifica, occorre utilizzare il punto: “.” Ad esempio:

```
import pippo.pluto.topolino.pippoclasse;
```

permette di riferirsi ad una classe “pippoclasse” contenuta nel package definito in precedenza.

É inoltre possibile riferirsi a tutte le classi di un package utilizzando il carattere asterisco “\*”, ad esempio il comando:

```
import pippo.pluto.*;
```

importa tutte le classi del package `pippo.pluto`. É da sottolineare come il nome completo di una classe sia costituito dal nome del package per esteso seguito dal nome della classe. Per cui il nome della classe `pippoclasse` di cui si è parlato prima è `pippo.pluto.topolino.pippoclasse`.

Per recuperare una classe, a run-time, la JVM effettua la ricerca delle classi tramite l'utilizzo della variabile d'ambiente `CLASSPATH`, che contiene i nomi delle directory dove sono localizzati i file class, o che costituiscono la radice per un albero di directory di un package.

## C#

Si è appena visto come in Java il legame tra i nomi delle classi e l'organizzazione fisica delle stesse sia profondo. In C# invece questi due concetti sono ortogonali. Infatti l'organizzazione dei nomi delle classi avviene tramite i *Namespace*, mentre quello delle librerie dei file di output utilizza gli *Assembly*.

**Namespace** Quando si definisce una classe in C# è possibile — anzi è caldamente consigliato — definirla all'interno di un namespace. Quello dei

namespace, diversamente dai package Java, è semplicemente un meccanismo per dare nomi univoci alle classi. Ad esempio definisco il seguente:

```
Namespace pippo{
Namespace pluto{
Namespace topolino{
class pippoclasse{
}
}
}
```

La classe creata è: `pippo.pluto.topolino.pippoclasse`

Per poter includere un namespace si utilizza la parola chiave `using` seguita da nome del namespace. Ad esempio il namespace precedente è incluso dal seguente comando:

```
using pippo.pluto.topolino;
```

Con questo comando automaticamente si inserisce tutto il namespace, come in Java quando si utilizza l'asterisco. Questo perché lo scopo stesso dei namespace è un'altro. Qui si sta semplicemente indicando lo spazio dei nomi in cui ricercare una particolare classe. Ci si potrà riferire a quella classe comunque senza utilizzare la parola `using`, ma si dovrà scrivere il nome della classe facendolo precedere dal nome di namespace completo.

A questo punto però le differenze tra il meccanismo di C# e quello di Java si cominciano a sentire. Infatti due namespace chiamati ad esempio `pippo.pluto` e `pippo.topolino` non fanno parte dello stesso namespace `pippo`. Non c'è nessuna relazione fra i due, come ci sarebbe, invece se fossero due package Java.

Ad esempio si consideri la seguente dichiarazione: `Namespace pippo{`

```
class classe classepippo{}  
Namespace pippo.pluto{  
class classepluto{  
}  
}
```

allora il seguente codice *non* si può richiamare: `using pippo;`

.....

```
pluto.classepluto myobject =new pluto.classepluto(); //errore!!
```

Questo perché `pluto` non è un nome di namespace. Il codice corretto può essere:

```
using pippo.pluto;
```

....

```
pluto.classepluto myobject =new pluto.classepluto();
```

**Librerie e Assembly** Veniamo ora a illustrare il ruolo degli assembly e i meccanismi con cui C#, tramite il CLR, carica le classi a runtime. Il meccanismo si appoggia a quello interno delle librerie *dll* (dynamic linked library) di Windows. Poiché un assembly contiene al suo interno tutti i riferimenti ai file di cui ha bisogno — tramite il manifest — quando il runtime carica in memoria la prima *dll* di un assembly, automaticamente sa anche quali sono tutte le altre parti di cui ha bisogno. Seguendo i riferimenti contenuti nel manifest è in grado di ritrovare tutti gli altri file dell'assembly e caricarli in memoria quando necessario. Il meccanismo di ricerca dei file è quello interno di Windows, per cui si ricercano le *dll* in alcuni path di sistema predefiniti (come ad esempio `c:\windows\system`) e in quelli indicati dalla variabile di sistema `PATH`.

In sintesi i meccanismi di gestione delle librerie dei due linguaggi sono simili in quanto entrambi si basano su variabili d'ambiente per il recupero dinamico del codice, sono diversi dal punto di vista dell'organizzazione fisica delle classi, poiché Java obbliga a costruire precisi path di ricerca.

Andiamo ora ad analizzare i meccanismi visti all'inizio del capitolo, descrivendoli in modo dettagliato per ciascuno dei due linguaggi Java e C#.

### 3.3 Reflection

I meccanismi di Reflection per Java e C# sono analoghi. In entrambi abbiamo una classe, (*Type* in C#, *Class* in Java) per la quale ciascuna istanza rappresenta e descrive una classe. Questa istanza contiene informazioni relative alla classe che descrive., come l'elenco dei campi dei metodi e dei costruttori. Sempre comune ad entrambi i linguaggi è la possibilità di interrogare una istanza a runtime per conoscerne il tipo. C# contiene in più la nozione di attributo, per la registrazione di informazione addizionale sui dati.

#### 3.3.1 Java

##### L'oggetto Class

Per comprendere come funziona la Reflection in Java, bisogna prima sapere come il tipo di informazione è rappresentato a run-time. Questo viene realizzato attraverso uno speciale tipo di oggetto, l'oggetto `Class`. C'è un oggetto `Class` per ogni classe che viene creata nel programma Java. Ovvero ogni volta che viene creata una classe viene creato anche un oggetto `Class` ad essa associato (e viene registrato in un file `.class` con lo stesso nome).

A runtime quando si vuole istanziare un nuovo oggetto di una data classe la JVM cerca il file `class` relativo, lo carica e lo istanzia. Il metodo statico

`forName(String)` è utilizzato per istanziare un oggetto conoscendo solo il nome della classe specificato in argomento.

Tramite la keyword `instanceof` è possibile sapere se un oggetto è l'istanza di una particolare classe.

```
Es: if(x instanceof cane) ((cane)x).abbaia();
```

Analogamente il metodo `isInstance(Class)` della classe `Object` ritorna un booleano che indica se l'oggetto chiamante è istanza della classe data in argomento.

### Metodi utili presenti nella classe `Class`

Il metodo più importante è senz'altro `getClass` presente, su tutti i discendenti di `Object`. Esso ritorna un oggetto di tipo `Class` rappresentante la classe di cui l'oggetto in uso è istanza. Ad esempio:

```
pippo.getClass()
```

ritorna un oggetto `Class` che rappresenta la classe di cui `pippo` è istanza. Altro metodo importante è il metodo `newInstance` viene utilizzato per costruire un'istanza della classe rappresentata dall'oggetto `Class`.

Esempio:

```
IoSonoLaClasseInteger= Class.forName(Integer);
```

```
pippo=IoSonoLaClasseInteger.newInstance();// pippo è un Integer
```

Vi sono poi altri metodi che forniscono altri tipi di informazioni. Il metodo `getInterfaces`, ad esempio, ritorna un array di oggetti `Class` ciascuno dei quali rappresenta un'interfaccia contenuta nell'oggetto `Class` in esame, e dunque implementata dalla classe che esso descrive. Infine tramite metodo `getSuperclass` si ottiene l'oggetto `Class` relativo alla sua classe parent.

### 3.3.2 C#

La reflection in C# si realizza principalmente attraverso il Namespace `System.Reflection`. All'interno di quest'ultimo, gli strumenti più utili sono senz'altro la classe *Type* e la classe *Assembly*.

#### La classe `Type`

Un oggetto `Type` descrive le informazioni di tipo di una classe, come ad esempio informazioni sui metodi, le proprietà e i campi. Da un qualunque oggetto C# è possibile ottenere l'oggetto `Type` che ne rappresenta la classe, richiamando il metodo `GetType()`.

Accanto alla classe `Type` vi sono molte altre classi che rappresentano altri costrutti come ad esempio la classe *MethodInfo* che descrive un metodo, o anche la classe *ConstructorInfo*, che rappresenta un costruttore. Da un oggetto `Type` è possibile quindi ottenere ogni possibile informazione riguardo alla classe che rappresenta ottenendola sotto forma di uno di questi oggetti, tramite appositi metodi accessori. Ad esempio per ottenere informazioni sui costruttori di una classe rappresentata da un oggetto `Type` si utilizza `GetConstructors()`.

Un oggetto *Assembly*, infine, rappresenta un assembly. Tramite tale oggetto è possibile ottenere informazioni riguardanti un assembly ad esso associato. Tali informazioni possono essere, ad esempio di quanti e quali file è costituito, le classi che contiene e i loro metodi pubblici.

A ben vedere questa operazione non ha corrispondenti in Java, cioè in Java non è possibile operare allo stesso modo sui package, direttamente dal codice.

## 3.4 Meccanismo di Serializzazione

### 3.4.1 C#

Il meccanismo di serializzazione di C# è molto ordinato e semplice. Vi sono due possibilità, lasciar fare tutto al meccanismo standard di .NET oppure controllare in modo più preciso la serializzazione. Se si vuole utilizzare il meccanismo di serializzazione standard è sufficiente marcare la classe che si vuole serializzare utilizzando l'attributo `[Serializable]`. A questo punto per impedire la serializzazione di particolari valori all'interno di quella classe basta anteporre ad essi l'attributo `[NonSerialized]`. Se si vuole alterare il meccanismo di serializzazione standard di alcuni tipi valore si può utilizzare l'attributo `[MarshalAs]`<sup>2</sup> Invece, se si vuole controllare il meccanismo di serializzazione in modo più preciso basta implementare l'interfaccia `ISerializable`. Questa richiede di implementare un unico metodo, `GetObjectData (SerializationInfo, StreamingContext)`, che riciede in pratica di inserire in un oggetto `SerializationInfo` le informazioni necessarie per la serializzazione, mentre lo `StreamingContext` è la destinazione della serializzazione.

A questo punto per serializzare l'oggetto si possono utilizzare diverse classi. Il meccanismo di base fa uso delle classi che derivano dalla classe astratta `Formatter` che tramite i metodi `Serialize` e `Deserialize` opera la serializzazione verso la destinazione e nel formato desiderato. Le librerie standard forniscono il `BinaryFormatter` per la serializzazione binaria e il `SoapFormatter` per la serializzazione tramite Soap. Esistono poi numerose

---

<sup>2</sup>Si tratta di una necessità molto particolare, di cui un esempio è quello del tipo `String` che viene serializzato normalmente come `BStr` se trasferito a codice `unmanaged`, mentre può essere serializzato come `LPStr` tramite l'utilizzo di questo attributo

classi che nascondono l'utilizzo di questo meccanismo. Ad esempio se voglio serializzare l'oggetto su un file XML posso utilizzare la classe `XmlSerializer` che mette a disposizione una serie di facilitazioni (come l'utilizzo di particolari attributi) per la serializzazione XML. Altro esempio è quello che permette la pubblicazione di oggetti come servizi; infatti le richieste all'oggetto-servizio web vengono passate al `SoapFormatter` in modo trasparente al programmatore. (vedi sezione 3.7)

A questo punto è possibile classificare quattro tipi diversi di oggetti in C#, rispetto alla serializzazione:

- *Not Marshalled* sono oggetti classici non destinati alla serializzazione in alcun modo. Sono quelli di default, che derivano da `object`.
- *UnBound* o **MarshalbyValue** - sono gli oggetti marcati con l'attributo *Serializable*. Tali oggetti possono essere serializzati, ma ciò che viene trasferito è solo il loro stato (valore) viene cioè creata una copia, che ha un'identità propria. Occorre rilevare che l'attributo non si eredita, pertanto si deve precisare sempre in modo esplicito che un dato oggetto è serializzabile.
- *MarshalByref* o *AppDomainBound* Objects - Ooggetti derivati direttamente o indirettamente dalla classe *Marshalbyrefobject*. Per tali oggetti è possibile trasferire al di fuori dell'*Appdomain* cui appartengono un riferimento che permette l'accesso ai loro metodi pubblici anche da parte di un client remoto Tale sistema fa uso di oggetti proxy, ovvero copie locali dell'oggetto remoto con cui si comunica.
- *Context bound*- Si tratta di oggetti legati al proprio *Context*, ovvero al proprio *thread* di esecuzione. Si tratta di una ulteriore specificazione rispetto agli oggetti `MarshalByRef`, utilizzata nel multithread-

ing. Molto semplicemente sono gli oggetti locali di un thread, che risultano quindi non accessibili agli altri thread dello stesso processo. Sono comunque serializzabili se marcati con l'attributo `[Serializable]` o se implementano l'interfaccia `ISerializable`

### 3.4.2 Java

Per poter serializzare una classe in Java occorre che questa implementi l'interfaccia `Serializable`. Questa interfaccia non ha metodi. É una sorta di attributo `[Serializable]` *ante litteram*. Una volta che la classe implementa questa interfaccia si può serializzare un'oggetto di questo tipo utilizzando i metodi `writeObject` e `readObject` delle classi `ObjectInputStream` e `ObjectOutputStream`. Infine se si vuole evitare che un oggetto o una variabile interna alla classe che implementa `Serializable` venga serializzata occorre fare precedere la parola chiave `transient`. Se invece si vuole controllare la serializzazione dell'oggetto si utilizza l'interfaccia `Externalizable`, che estende `Serializable`. Una classe che implementa `Externalizable` deve definire due metodi `readExternal` e `writeExternal`, che vengono poi richiamati automaticamente da `ObjectInputStream` e `ObjectOutputStream`. E fin qui le analogie tra Java e C# sono decisamente evidenti. Infatti abbiamo l'interfaccia `Serializable` di Java che corrisponde all'attributo `[Serializable]` di C# e l'interfaccia `Externalizable` di Java che corrisponde all'interfaccia `ISerializable` di C#. In realtà vi è un'ultima stranezza. Anziché utilizzare `Externalizable`, per controllare completamente il meccanismo di serializzazione è possibile definire, all'interno della classe che implementa `Serializable`, due nuove versioni di `readObject` e `writeObject`, per le quali è obbligatorio seguire la seguente *signature*:

```
private void writeObject(ObjectOutputStream stream)
    throws IOException
private void readObject(ObjectInputStream stream)
    throws IOException
```

Quello che è strano, per non dire assurdo, è che questi due metodi, indicati come privati, in realtà vengono poi utilizzati dall'oggetto `ObjectInputStream` (o `ObjectOutputStream`) al quale passiamo la nostra classe per sostituire i metodi standard. Se poi si ha la necessità di utilizzare i metodi di serializzazione di default, pur avendone ridefinito i due nuovi, o anche all'interno di essi, allora si possono richiamare `defaultReadObject` (e `defaultWriteObject` è l'analogo).

In sintesi, il meccanismo utilizzato da `C#` è certamente più semplice e ordinato di quello utilizzato da Java, il quale risulta meno intuitivo nella realizzazione dell'interfaccia `Serializable`. Tuttavia è da notare che l'attributo `[Serializable]` non è applicabile ad una interfaccia, né è ereditabile, soluzioni che invece sono possibili con Java.

## 3.5 Caricamento dinamico del codice

### 3.5.1 Java

Per il caricamento dinamico delle classi, in Java, si utilizza la classe `ClassLoader` che è responsabile per il caricamento di un oggetto `Class` a Runtime. Tale classe è astratta. Dato il nome di una classe un class loader cerca di localizzare o generare dei dati che costituiscono una definizione per quella classe. Ogni oggetto `Class` contiene un riferimento all'oggetto `ClassLoader` che l'ha definito. Tramite tale riferimento, il metodo `Class.getClassLoader`

restituisce l'oggetto `ClassLoader` opportuno per l'istanza data. Si implementano sottoclassi di **`ClassLoader`** quando si vuole controllare il modo in cui la Java Virtual Machine carica dinamicamente le classi. Il meccanismo di funzionamento della classe **`ClassLoader`** utilizza il principio di delegation nella ricerca di classi e risorse. Ogni istanza di **`ClassLoader`** ha un class loader parent associato. Al momento in cui gli viene richiesto di cercare una classe o una risorsa, per prima cosa il **`ClassLoader`** delega la ricerca al proprio parent. Nel caso questo fallisca esegue la propria ricerca. Il loader interno alla JVM (chiamato *bootstrap class loader*) funge da parent ultimo (non avendo un loader parent) per questo meccanismo di delegation.

Normalmente la JVM carica le classi dal file system locale, secondo un metodo che dipende dalla piattaforma su cui si trova. Per esempio sui sistemi UNIX, la virtual machine carica le classi dalle directory definite nella variabile d'ambiente `CLASSPATH`.

Tuttavia, alcune classi possono non trovarsi su un file, ma possono essere ottenute da altre sorgenti come la rete, oppure possono venire create da una applicazione. Il metodo *defineClass* converte un array di bytes in un'istanza della classe *Class*. Posso quindi ottenere nuovi oggetti della classe così caricata utilizzando il metodo *newInstance* della classe *Class*.

Una volta che un class loader ha creato una classe i suoi metodi e i suoi costruttori possono riferirsi ad altre classi. Per determinare la classe a cui ci si riferisce, la virtual machine richiama il metodo *loadClass* del class loader che ha originalmente creato la classe, che diventa così il loader di default.

### 3.5.2 C#

In C# il meccanismo di caricamento dinamico di classi passa attraverso la classe *Assembly*. Tramite i metodi (statici) *Assembly.Load* e *Assem-*

*bly.LoadFrom* di tale classe, è possibile aggungere un assembly esterno alla *assembly cache* locale.

Una volta caricato il nuovo Assembly posso creare una nuova istanza di una delle classi in esso contenute tramite il metodo *CreateInstance*.

I due metodi citati si differenziano nel fatto che il secondo è specializzato per il caricamento di Assemblies da remoto, in modo anche da poter gestire la politica di sicurezza rispetto alla locazione remota da cui si effettua il download dell'assembly. In particolare il metodo *Assembly.Load(string, Evidence)* permette il caricamento di un assembly a cui viene associata un'informazione — il parametro *Evidence* — circa l'attendibilità dell'assembly caricato. Nonostante questo alcune versioni di *Load* sono deprecate. Lo sono in particolare quelle che consentono il caricamento di un array di byte che rappresenta un assembly. Questo perché il procedimento, in quel caso, funziona a prescindere dal meccanismo di sicurezza interno a .NET, non richiedendo nessun parametro di sicurezza sul codice.

## 3.6 Multithreading

### 3.6.1 Java

In Java i thread sono realizzati tramite l'uso di oggetti che sono derivati dalla classe `Thread`. Questa classe implementa l'interfaccia `Runnable` e dunque possiede un metodo chiamato `run` che va ridefinito con il codice che si vuol far eseguire al thread. Invece nella classe `Thread` è presente il metodo `start()` che, inizializza il thread e poi richiama il metodo `run`. Ovviamente prima di poter richiamare `start` occorre aver costruito l'oggetto con il costruttore appropriato. Infatti il metodo `start` è il metodo che gestisce la creazione del thread come processo all'interno del sistema operativo in cui si trova.

Come si è detto all'inizio del capitolo introducendo i threads, il loro utilizzo comporta la necessità di controllare la condivisione di risorse.

In Java è possibile sincronizzare l'accesso a dei dati condivisi tramite la parola chiave *synchronized*. Questa parola può essere applicata ad un metodo. In tal caso essa garantisce l'accesso esclusivo a quel metodo ad un solo thread alla volta. In pratica il primo processo che richiama un metodo *synchronized* esegue una lock sull'area di memoria dove il codice per quel metodo è registrato. Quando il thread ha terminato l'esecuzione rilascia la lock. Java gestisce una sola lock per tutti i metodi *synchronized* di una classe, quindi un solo thread alla volta può utilizzare tali metodi.

### 3.6.2 C#

In C# il meccanismo di gestione dei thread è diverso da Java. Infatti, tramite un meccanismo simile a quello dei puntatori a funzione, è possibile associare un qualunque metodo ad un thread. Per far questo si utilizzano le classi *Thread* e *ThreadStart*. Il seguente spezzone di codice mostra come si associa un metodo ad un Thread e come si lancia il thread stesso:

```
1 myClass myObj = new myClass();
2 ThreadStart metodoavviothread=new ThreadStart(myObj.myMethod);
[.....]
3 Thread thread = new Thread(metodoavviothread);
4 thread.Start();
```

Alla riga 2 viene costruito un oggetto di tipo *ThreadStart*. Questo oggetto è una sorta di puntatore a funzione, infatti esso viene associato — sempre in riga 2 — al metodo *myMethod* della classe *myClass*, definita in riga 1. In C# oggetti come *ThreadStart* sono chiamati *delegate*. Successivamente — alla riga 3 —, quando si vuole avviare il thread, si costruisce un oggetto

Thread passando come argomento del suo costruttore l'oggetto ThreadStart creato in precedenza. Infine richiamando il metodo *Start* si avvia il thread.

E' da sottolineare che la classe Thread operi solo come wrapper per la gestione del thread stesso. Come si vede, quindi, in C# posso costruire un thread a partire da un qualunque metodo di una qualunque classe. Questo fatto costituisce una importante differenza rispetto a Java, in cui i threads si costruiscono come classi a partire dalla classe base Thread (vedi[8]) Bastano infatti poche righe di codice, come abbiamo visto per poter utilizzare un qualunque metodo come un Thread. Dal lato Java, invece, è necessario costruire una classe ex-novo, cosa che richiede certamente un maggior sforzo implementativo.

## 3.7 Programmazione di rete

Per capire il meccanismo di *remoting* di C#, e le sue scelte conviene prima, a mio avviso, osservare il meccanismo di Java.

### 3.7.1 Java

Il meccanismo di comunicazione di rete di Java utilizza il meccanismo standard dei *socket*. Ciò significa che, per poter comunicare tra un client e un server occorre lavorare su due tipi di oggetti: il *Socket* e il *ServerSocket*. Quest'ultimo è quello che viene utilizzato da un server. Al momento dell'attivazione del server si sceglie un port su cui mettere in ascolto il ServerSocket. Quindi si richiama il metodo *accept()*. Appena un client richiede la connessione questo metodo restituisce un oggetto Socket. L'applicazione server può allora utilizzare questo oggetto per ottenere gli Stream di input e di output, attraverso i quali sarà in grado di comunicare col client. Nel caso si debbano

servire più clients ci si appoggia facilmente ai threads. Il meccanismo di per sé è molto semplice, poiché creato il socket, tutta la comunicazione avviene attraverso gli stessi mezzi degli stream. Tuttavia, compito del programmatore, è anche quello di accertarsi che, anche in caso di errore, i socket occupati vengano chiusi. E non sempre è semplice.

### 3.7.2 C#

In C# (e più in generale in ambiente .NET) si possono scegliere due strade per la comunicazione di rete. O realizzare una connessione di rete in maniera classica tramite la gestione diretta dei socket, come in Java, oppure appoggiarsi al *Remoting Framework* messo a disposizione da .NET. Per la prima scelta c'è poco da dire, in sostanza i meccanismi sono gli stessi di Java, anche se in C# è previsto il supporto per ricezione e spedizione di messaggi in modo asincrono, e si utilizza un unico stream per input e output. Questa scelta è necessaria se si deve costruire una applicazione che comunichi all'esterno del .NET framework.

La seconda scelta, ovvero il meccanismo del .NET Remoting Framework, è pensata per semplificare ulteriormente il lavoro del programmatore, eliminando il controllo diretto dei socket. Infatti il concetto base è che anche un servizio Web è un oggetto. Per tanto ciò che un server deve fare è fornire ai client un oggetto o un riferimento ad un oggetto-servizio. Fatto questo la comunicazione client-servizio è una normale comunicazione tra oggetti. Al programmatore viene richiesto solo di scegliere il port, il nome del servizio, l'indirizzo del server.

Senza dubbio con la modellizzazione C# la costruzione di un servizio Web appare molto più semplice perché il lavoro è incentrato principalmente sulla realizzazione dell'oggetto servizio e non sulla problematica (spesso noiosa e

ripetitiva) di gestire la creazione e l'attivazione dei socket. Purtroppo nella realizzazione della tesi questo meccanismo non si è potuto utilizzare poiché, di fatto non è stato possibile interfacciare il lato Java con questo particolare meccanismo. Per questo motivo non si è descritto il meccanismo di remoting di C# nel dettaglio.

# Capitolo 4

## Un linguaggio per lo scambio dei dati:XML

Nel capitolo precedente sono stati presentati i linguaggi Java e C#, utilizzati per realizzare l'infrastruttura. A questo punto occorre definire una codifica comune tramite la quale i due linguaggi possano scambiarsi i dati. Questo capitolo illustra perché si è scelto di utilizzare XML piuttosto che SOAP per questo fine. Per tale motivo viene prima illustrato il linguaggio XML in modo piuttosto approfondito e vengono inoltre presentati i meccanismi di serializzazione offerti Java e da C# per l'utilizzo di SOAP e XML. Viene, infine descritto perché si è dovuto realizzare un apposito serializzatore XML per Java.

### 4.1 Introduzione a XML

L'eXtensible Markup Language o XML è un linguaggio che nasce come estensione del più noto linguaggio HTML[1] che è considerato la base del World Wide Web. Lo scopo principale con cui fu realizzato HTML era quel-

lo di consentire la visualizzazione dei dati. Per questo motivo HTML prende in considerazione soprattutto il *modo* con cui le informazioni vengono presentate. XML, invece prende in considerazione il *tipo* e la *struttura* di tali informazioni, cercando di descriverli.

Tramite XML è possibile anche descrivere informazioni di tipo strutturale e semantico relative ai dati veri e propri. Questi “*dati sui dati*”, o *meta-dati*, offrono un aiuto addizionale per l’applicazione che utilizza i dati e consente un maggior livello di gestione e manipolazione delle informazioni.

#### 4.1.1 Utilità di XML

Dal punto di vista dell’interoperabilità fra ambienti eterogenei, l’XML presenta caratteristiche importanti per quanto riguarda lo scambio e la condivisione dei dati.

##### Scambio Dati

Il linguaggio XML viene utilizzato per lo scambio di dati tra sistemi e database contenenti dati in formati incompatibili. Una delle problematiche che richiedono più tempo per una soluzione è proprio lo scambio di dati tra tali sistemi su Internet.

Convertendo i dati in XML è possibile semplificare questo lavoro e, inoltre, si codificano le informazioni in modo tale che esse possano essere lette in molti tipi di applicazioni diverse.

##### Dati condivisi

I dati registrati in formato XML sono registrati su comuni file di testo. Questo fatto rende XML un linguaggio non dipendente da una particolare pi-

attaforma hardware/software e rimane legato solo al tipo di codifica testuale che si utilizza.

Da quanto detto seguono naturalmente due cose:

- XML può essere utilizzato per registrare i dati su un DB o file. Si possono scrivere applicazioni specifiche per registrare i dati in formato XML, mentre sarà sufficiente una generica applicazione capace di leggere XML per operare una lettura.
- I dati in formato XML sono accessibili da più utenti. L'indipendenza hardware e software dei dati permette di renderli disponibili a molte diverse applicazioni, oltre che un semplice browser web. Un client può utilizzare il documento XML come una sorgente di dati, come si fa accedendo ai database.

Nell'ambito dell'interoperabilità in rete, un documento XML può essere inserito senza problemi in un documento HTML (tramite il tag `<xml>`), ed è quindi estremamente semplice utilizzare tutti quegli strumenti di comunicazione che sfruttano l'HTML.

### 4.1.2 Regole sintattiche

XML definisce delle regole sintattiche molto rigide rispetto a quelle dell'HTML. Questo è dovuto al fatto che nessun tag XML ha un significato semantico a priori, come invece è per HTML. Per cui, ad esempio, ogni tag aperto deve essere obbligatoriamente chiuso, altrimenti il parser XML si bloccherà dando un errore. Sempre per lo stesso motivo XML è *case-sensitive*.

La prima linea di un documento XML è l'elemento *dichiarazione*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

La dichiarazione non è soggetta alla regola di chiusura obbligatoria dei tag citata in precedenza. Alla

dichiarazione segue il documento XML vero e proprio. Esso è costituito da un primo tag detto *root* la cui chiusura verrà interpretata come la fine del documento XML. Le componenti principali di un documento XML sono gli *elementi* e gli *attributi* ad essi correlati. Gli elementi sono costituiti da un tag di apertura, uno di chiusura e tutto ciò che è contenuto all'interno di questi (testo o altri elementi) Ad esempio:

```
<Pippo> <cane> Pluto </cane> </Pippo>
```

Gli attributi invece costituiscono un'informazione aggiuntiva per un dato elemento e vengono inseriti direttamente nel tag di apertura di un elemento. La sintassi per gli attributi è la seguente:

$$\langle \text{elemento attributo1="val1" ... attributoN="valN"} \rangle \dots \langle / \text{elemento} \rangle$$

Anche se Xml non definisce nessuna semantica specifica per elementi e attributi, gli elementi sono pensati per contenere informazioni, mentre gli attributi sono stati creati pensando di utilizzarli per contenere meta-informazioni sugli elementi. Esempio:

```
<file type="gif">computer.gif</file>.
```

Nell'esempio è irrilevante il tipo del file ai fini del dato (che è solo il nome di un file), ma è importante per il software che vuole manipolare l'elemento.

Una descrizione completa della sintassi Xml è reperibile al sito[1]

### 4.1.3 Namespaces

Poiché i nomi degli elementi XML non sono fissati spesso si rischia di incorrere in conflitti di nomi, per cui uno stesso nome viene dato ad elementi con significato diverso.

## Prefissi

Un primo modo per risolvere un conflitto di nomi può essere quello di aggiungere un prefisso ad un elemento. Se ho, quindi un tag `!pippo!` che ha due significati posso creare `!h:pippo!` e `!f:pippo!` aggiungendo il prefisso anche a tutti gli elementi figli di ciascuno dei due elementi, come nell'esempio seguente:

```
<f:pippo>
  <f:pluto>minni</f:pluto>
  <h:pippo>
    <h:paperino> pippoh </h:paperino>
  </h:pippo>
  <f:topolino>paperino</f:topolino>
</f:pippo>
```

## Utilizzo di Namespaces

L'utilizzo del prefisso rischia, tuttavia, di diventare estremamente tedioso, dovendo andare a modificare tutti i sottoelementi di un elemento dato.

Per ovviare a questo problema si è dunque pensato di utilizzare i *namespace*. L'assegnazione di un namespace ad un dato elemento avviene tramite l'uso dell'attributo `xmlns`, che ha la seguente sintassi:

```
xmlns:prefisso_di_namespace="namespaceURI".
```

Solitamente il namespaceURI è un indirizzo internet, ma è sufficiente che sia una **URI** (Uniform Resource Identifier)

Una volta che viene definito un namespace nel tag iniziale di un elemento, tutti i suoi elementi figli con lo stesso prefisso sono associati a quel namespace, e vengono in questo modo risolti i problemi di omonimia tra elementi.

Se si dichiara un nuovo namespace, quello nuovo sostituisce il vecchio fintanto che ha valore — ovvero all'interno dell'elemento in cui è definito — Per mixare più namespaces si fa come segue: il primo namespace è quello di default e si fa la dichiarazione come prima. Il successivo deve essere dichiarato nello stesso tag e poi deve essere fatto seguire da un'etichetta. Tale etichetta verrà utilizzata nei tag degli elementi che devono riferirsi al namespace secondario. Ad esempio:

Dichiarazione del namespace pluto:

```
<pippo xmlns:http://www.pipposito.com/pippo.xdr
xmlns:pluto=http://www.secondonamespace.com/pluto.xdr>
```

Utilizzo del namespace pluto:

```
<topolino pluto:type=topolino> datoditopolino </topolino>
```

In questo modo si è assegnato all'elemento *topolino* il type *topolino* dello schema nel namespace pluto.

#### 4.1.4 Semantica di un documento XML: DTD e XSD

Come già detto in precedenza, XML non ha a priori nessun tag predefinito (se si eccettua la dichiarazione iniziale, che comunque non è considerata parte propria di un documento XML). Per definire il significato di un documento XML si possono utilizzare due meccanismi: i Document Type Definition (DTD) oppure gli XML Schema Definition (XSD). I primi descrivono i contenuti del documento XML tramite un linguaggio particolare, diverso da XML. I secondi sono invece scritti in XML per cui ne ereditano tutte le caratteristiche, come, ad esempio, l'estendibilità. Per contro sono più complessi da gestire. I DTD sono localizzati all'interno del documento che descrivono.

Per indicare a quale schema faccia riferimento un documento o una porzione di documento, si utilizzano i *namespace*, nel modo già descritto in preceden-

za: il valore del namespace indica in questo caso dove reperire sul web lo schema di definizione per quel documento XML.

Tramite i DTD (oppure gli XSD) un documento XML, corretto dal punto di vista sintattico, viene verificato anche nella sua costruzione semantica, tramite un processo detto di *validazione*. Se supera questo ultimo test, il documento XML è detto *valido*.

## 4.2 SOAP

Il protocollo SOAP (Simple Object Access Protocol) è una particolare evoluzione dell'XML-RPC, ovvero di uno schema per XML pensato per le chiamate a procedura remote. Lo scopo principale di quest'ultimo è quello consentire la comunicazione tra ambienti diversi anche non compatibili tra loro; questo avviene grazie ad un protocollo in grado di adattarsi a sistemi operativi diversi. Sebbene XML-RPC fornisca una sintassi flessibile e di semplice implementazione, se paragonata alla specifica SOAP, quest'ultima risulta di più semplice utilizzo in quanto offre un livello più alto di estensibilità per sistemi orientati ad oggetti.

### 4.2.1 Specifiche SOAP

In sostanza SOAP è un particolare XML Schema Document pensato per lo scambio di messaggi tra oggetti. Un documento SOAP è dunque a tutti gli effetti un documento XML, e come tale ne eredita la portabilità web tramite HTML.

Un messaggio SOAP è un documento XML il cui elemento root si chiama `<Soap:Envelope>`. All'interno di questo elemento vi può essere un elemento

<Soap:Header>, opzionale, mentre vi è sempre un elemento <Soap:Body> che è obbligatorio.

L'elemento <Soap:Envelope> deve obbligatoriamente avere l'attributo di namespace, `xmlns`, settato al valore:

```
http://schemas.xmlsoap.org/soap/envelope.
```

L'elemento <Soap:Header> viene utilizzato per estendere le caratteristiche di un messaggio Soap. Può contenere delle indicazioni che obbligano o meno chi lo riceve a modificare oppure no alcuni elementi del messaggio.

L'elemento `Soap:Body` è il messaggio Soap vero e proprio contenente il messaggio per il destinatario definitivo.

In qualunque elemento all'interno di un messaggio SOAP è possibile inserire l'attributo `encodingStyle`. Esso indica l'indirizzo in cui localizzare le regole di serializzazione dei dati utilizzate nel messaggio SOAP. Tali regole valgono solo all'interno dell'elemento in cui è l'attributo è definito.

Un messaggio SOAP è costituito generalmente da una invocazione di procedura, oppure di una risposta ad una richiesta precedente.

Ciò che interessa particolarmente di SOAP, nel contesto di questo lavoro, è il meccanismo di serializzazione. Questo non è esattamente ciò per cui SOAP è stato pensato ma una sua diretta conseguenza. Infatti SOAP è stato pensato principalmente per eseguire RPC fra ambienti eterogenei. Per far questo occorre quindi poter inviare al server oltre al nome di un servizio anche dei parametri, e sono proprio questi a dover essere serializzati.

## 4.3 XML, SOAP e la comunicazione tra Java e C#

### 4.3.1 La scelta di XML

Per quanto detto sin qui, se si volesse cominciare a costruire una struttura di interoperabilità tra due sistemi, si sarebbe tentati di utilizzare prima di tutto Soap. Questo per la sua semplicità e per il largo supporto offerto da molti linguaggi. C# e Java non sono da meno. In .NET la comunicazione tramite SOAP è quella utilizzata di default dal meccanismo interno di remoting.

Tramite la classe `System.Runtime.Serialization.FormatterServices.Soap` è possibile operare direttamente la serializzazione di molti tipi di oggetti diversi. Anche in Java è supportato Soap. Tutto il lavoro in questo ambito, fino all'avvento di Sun One, era sulle spalle di Apache. Questa organizzazione mette a disposizione tramite il package `org.apache.soap` tutta una serie di classi per la costruzione di servizi web in grado di utilizzare SOAP.

SOAP è uno standard e le due implementazioni dovrebbero, per questo motivo, essere compatibili. In realtà questo non è vero. Come dimostra Catalin Tomescu in un suo articolo[22] le due versioni di SOAP, quella di Microsoft e quella di Apache, non sono compatibili. Il problema è che il lato Java (un server Tomcat) richiede ad ogni elemento serializzato di specificare il tipo in un particolare prefisso `xsi:type`, non richiesto invece dalla versione .NET.

Come conseguenza della serializzazione di un oggetto in .NET si ottiene un documento senza questo attributo, ma il corrispondente deserializzatore SOAP dal lato Java, non riesce a deserializzarlo.

In un articolo successivo[22], Tomescu, propone una soluzione che tut-

tavia è limitata alla serializzazione in un contesto di webservices<sup>1</sup>. Fino alla versione 2.2 di Apache-Soap questo problema rimane irrisolto.

Vista, l'impossibilità di utilizzare la "patch" proposta da Tomescu nel contesto specifico del nostro lavoro, si è optato per l'utilizzo di XML. Infatti essendo pensato per la descrizione dei dati, la serializzazione è un problema affrontata in modo approfondito da entrambe le piattaforme.

Nella recente presentazione del framework Sun One, gli sviluppatori di Sun hanno affermato che tale framework è in grado di interagire tranquillamente con .NET tramite SOAP. Tuttavia, per mancanza di tempo, non è stato possibile verificare tale affermazione.

## 4.4 Gestione di XML in Java e C#

### 4.4.1 Una premessa sui tipi di dati

Anzitutto occorre fare un primo discorso sull'interoperabilità tra Java e C# per quanto riguarda i tipi di dati. Nel paragrafo 3.2.2 parlando del meccanismo di boxing/unboxing di C# si è detto che questo linguaggio non ha tipi primitivi. Questo fatto significa che Java e C# possono scambiarsi solo alcuni tipi di dati in modo trasparente. La tabella 4.1 riassume i tipi che possono essere scambiati all'interno del progetto. Concludendo, è possibile operare la serializzazione solo degli oggetti i cui campi sono di uno dei tipi sopra-descritti, o una composizione di essi.

Quanto appena detto obbliga a limitare i tipi di dati scambiati ai soli tipi che seguono:

---

<sup>1</sup>Infatti propone la modifica di un file di configurazione del servizio web

- I tipi che in Java wrappano i tipi primitivi Integer, Boolean, Short, ecc, e i loro corrispondenti tipi C#;
- le stringhe;
- gli array di oggetti dei tipi appena citati;
- oggetti i cui campi sono a loro volta oggetti dei tipi descritti in precedenza;

## 4.5 L'XmlSerializer di C#

In C# il meccanismo di serializzazione e deserializzazione da e per XML viene controllato tramite la classe *System.Xml.Serialization.XmlSerializer*. Tramite i metodi *Serialize* e *Deserialize* è possibile operare la serializzazione

tipo JAVA	tipo C#
Integer	int
String	string
Boolean	bool
Character	char
Short	short
Byte	byte
Float	float
Double	double
Long	long
Array[wrapper type]	Array[value type]

Tabella 4.1: Tabella dei dati comuni

dei campi pubblici e delle proprietà pubbliche di un qualunque oggetto marcato con l'attributo `[Serializable]`.

Vi sono molti altri meccanismi per il controllo di un documento XML, ad esempio tramite l'attributo `[XmlElementAttribute]`, che consente di modificare il Tag di rappresentazione per un oggetto da serializzare. Tuttavia è da sottolineare che il serializzatore deve essere costruito a partire da una classe specifica. Infatti ogni versione pubblica del costruttore richiede sempre come parametro almeno un oggetto *Type*. Le versioni utilizzate nella tesi sono due. La prima è quella standard che prende come parametro un oggetto *Type* e costruisce un serializzatore per quell'oggetto.

La seconda è quella con la seguente signature:

```
public XmlSerializer(  
    Type type,  
    Type[] extraTypes  
);
```

Questa seconda versione consente di serializzare classi che al loro interno contengono array di object. Infatti la prima versione non è in grado di serializzare un array di object generici. Questa, invece ci riesce, limitatamente ai tipi specificati nell'array `extraTypes`.

In definitiva non è possibile avere un serializzatore generico per ogni tipo di oggetto. Questo punto sarà importante, come si vedrà nel prossimo capitolo per alcune scelte nella costruzione dell'architettura.

## 4.6 Un serializzatore personalizzato per Java: *XmlSerializer*

In Java sono presenti numerosissime implementazioni diverse per lavorare sul codice XML. Le società Apache, Electric e W3C hanno realizzato numerose classi in grado di operare attivamente sul codice XML. Esse permettono di gestire un documento XML tramite il cosiddetto DOM[1] (Document Object Model). In esso un documento XML è rappresentato come un albero costituito da nodi eterogenei. Ciascun nodo è associato ad un elemento del documento XML. Quindi l'elemento root del documento è associato alla radice dell'albero corrispondente, mentre tutti gli altri elementi del documento ne costituiscono gli altri nodi, secondo una scala gerarchica per cui un elemento XML con elementi figli avrà come corrispondente nel DOM un nodo con dei nodi figli associati a quegli stessi elementi figli.

Nella tesi si sono utilizzate le API fornite da W3C. In esse l'interfaccia `org.w3c.Document` rappresenta un documento XML, e consente numerose operazioni su di esso. Tale interfaccia fa parte di una gerarchia di interfacce che nasce da `org.w3c.Node`, che rappresenta un generico nodo XML. Altre interfacce importanti nella gerarchia sono `org.w3c.Attr` e `org.w3c.Element`, che rappresentano attributi ed elementi in un documento XML. Va infine ricordata anche la classe `javax.xml.parsers.DocumentBuilder` che consente, tramite il metodo `parse`, di costruire un oggetto `org.w3c.Document` a partire da un file, da un `InputStream` o da un oggetto `Reader`.

Si è scelto di realizzare un Serializzatore `Xml` personalizzato, utilizzando solo le classi base della W3C per la lettura di documenti XML. In pratica il de-serializzatore opera a partire da un oggetto con interfaccia `org.w3c.Node`, quale appunto `org.w3c.Document`. Le api per la serializzazione già presenti,

infatti non sono facilmente manipolabili per essere rese compatibili con l'equivalente serializzatore C#. Occorre anche aggiungere che il serializzatore utilizzato per il progetto non deve tener conto di tutte le problematiche possibili inerenti ad XML, dovendo operare in un contesto di utilizzo specifico, al contrario dei serializzatori già pronti, che sono più complessi, essendo pensati per un utilizzo più generale.

Il serializzatore personalizzato `XmlSerializer` è quindi pensato per riprodurre un output identico alla sua classe omonima fornita da C#, per quanto riguarda i soli tipi scambiati dal framework. L'idea base è quella di utilizzare delle classi specializzate per il marshalling di una particolare classe. Tali classi devono implementare l'interfaccia `ClassMarshaller`. Inoltre a ciascuna classe deve essere associata una stringa che sarà il nome dell'elemento XML con il quale essa sarà descritta nell'output del serializzatore. Per tener conto di queste due associazioni si utilizza la classe `XmlSerializeRegistry`. Essa contiene tre `HashTable` con le associazioni indicate nella tabella 4.2

nome	key	value
<code>classRegistry</code>	<code>Class</code>	<code>ClassMarshaller</code>
<code>elementRegistry</code>	<code>String</code>	<code>Class</code>
<code>keyRegistry</code>	<code>Class</code>	<code>String</code>

Tabella 4.2: `HashTable` nella classe `XmlSerializeRegistry`

Ovviamente la gestione delle hashtable è trasparente all'utente della classe `XmlSerializer`. Tramite il metodo `setMapping` di `XmlSerializer` l'utente può associare una classe al suo `ClassMarshaller` e alla stringa con la quale vuole che la classe venga descritta in XML. Solo una volta che sono stati stabiliti i mapping per le classi da serializzare, è possibile richiamare i metodi

`serialize` e `deserialize` della classe `XmlSerializer`. In caso contrario verrà generata un'eccezione.

Per quanto detto in precedenza — al punto 4.4.1 — in merito ai tipi di dati che è possibile scambiare con successo tra Java e C#, sono stati realizzati i `ClassMarshaller` solo delle seguenti classi:

- `String`
- `Integer`
- `Vector`
- `Short`
- `Long`
- `Boolean`
- `Character`
- `Float`
- `Double`

Si sono inoltre realizzati tre `ClassMarshaller` speciali:

- `ArraySerializer`: per serializzare `Array`;
- `VectorSerializer`: per serializzare un oggetto della classe `Vector`. In C# la sua corrispondente è la classe `ArrayList`;
- `BeanSerializer`: per serializzare oggetti compositi, javaBean-like, per i quali sono presenti due metodi pubblici `set` e `get` per ciascun campo che si vuole serializzare;

### 4.6.1 Dinamicità nella serializzazione

Questo serializzatore consente di serializzare e deserializzare qualunque classe per la quale sia presente un `classMarshaller`, e di cui è stato settato un mapping appropriato nell'`XmlSerializeRegistry` del serializzatore stesso. Per deserializzare oggetti, aventi al loro interno altri oggetti a loro volta non semplici è necessario costruire un apposito `XmlSerializeRegistry`, contenente anche le associazioni per la deserializzazione di questi oggetti. Questo potrebbe, in linea di principio, non esser fatto a runtime.

Tuttavia, nel contesto in cui il serializzatore deve operare, ci si trova a dover deserializzare oggetti sconosciuti, dei quali non è noto il mapping di serializzazione a priori. Dunque è sconosciuto l'`XmlSerializeRegistry` corretto per tali oggetti. Per risolvere questo problema, è stato realizzato il metodo `joinSerializer(XmlSerializer)`. Tale metodo consente di unire il registry del serializzatore corrente con quello del serializzatore passato come parametro. In tal modo sarà sufficiente avere il serializzatore di ogni oggetto contenuto in quello da deserializzare per costruire a run-time l'oggetto `XmlSerializer` in grado di deserializzarlo correttamente.

## Capitolo 5

# Xml e interoperabilità tra Java e C#: l'infrastruttura

Dopo aver descritto nel capitolo precedente come e perché si è scelto di utilizzare XML come linguaggio intermedio per la comunicazione tra l'ambiente Java e il CLR di Microsoft.NET, passiamo ora alla descrizione dell'infrastruttura vera e propria.

Prima di iniziare la descrizione dell'infrastruttura, occorre sottolineare che il principio di funzionamento su cui essa è basata non è limitato ai soli ambienti Java e C#. Ad esempio è possibile utilizzare un qualunque altro linguaggio .NET (ad esempio Visual Basic.Net e Visual C++.Net) senza troppa fatica. Ma, più in generale, è anche applicabile a qualunque linguaggio o ambiente (sempre orientato a oggetti) che utilizzi un meccanismo di serializzazione e di loading dinamico degli oggetti evoluto come quelli di C# e Java, come ad esempio il linguaggio Eiffel [14].

## 5.1 Panoramica generale sull'infrastruttura

Prima di entrare nei dettagli implementativi ricordiamo il funzionamento generale dell'infrastruttura.

L'ambiente origine invia a quello di destinazione un messaggio XML contenente la codifica dello stato dell'oggetto che intende trasferire, accompagnato da dell'informazione addizionale. Questa informazione addizionale, viene poi utilizzata dall'infrastruttura per rintracciare la classe a cui quel messaggio fa riferimento.

All'utente dell'infrastruttura si richiede,infatti, di scrivere due versioni analoghe delle classi che vuole poter scambiare, una in Java e una in C#.

Figura 5.1: Infrastruttura: Panoramica generale

In figura 5.1 viene mostrato in modo schematico come un oggetto O può essere scambiato tra due applicazioni sfruttando l'infrastruttura. Vediamo cosa indicano gli elementi in figura:

- $O_{Java}$  è l'oggetto di partenza;
- $O_{XML}$  indica un documento XML che descrive l'oggetto  $O_{Java}$ ;
- $O_{C\#}$  è l'oggetto di arrivo;

All'interno dell'infrastruttura, invece si possono riconoscere:

- Uno o più server di classi C#;
- Uno o più server di classi Java;
- Uno o più server di Informazioni;

I *Server di Classi* possono essere un numero variabile per ciascun linguaggio. Su questi server vengono registrate le librerie dove sono definite le classi degli oggetti che verranno scambiati fra le due applicazioni.

Sui *Server di Informazioni* viene registrato l'indirizzo del server di librerie dove viene registrata ciascuna classe.

Vediamo quindi come avviene lo scambio di un oggetto. Si suppone che sia l'applicazione Java a voler trasferire un oggetto  $O_{Java}$  nell'ambiente .NET. A questo scopo essa richiama le opportune funzioni delle API dell'infrastruttura che consentono di ottenere la versione XML dell'oggetto:  $O_{XML}$ . L'applicazione Java, quindi invia tale documento all'applicazione C#. Una volta ricevuto  $O_{XML}$  l'applicazione C# richiama le opportune API che provvedono a restituire l'oggetto  $O_{C\#}$ . Per riuscire in questo, l'infrastruttura:

1. prende  $O_{XML}$ ;
2. contatta il server di informazioni per sapere su quale server di classi è localizzata la definizione per la classe descritta in  $O_{XML}$ ;
3. carica la libreria dal server di librerie localizzato al punto 2;
4. deserializza  $O_{C\#}$  e lo restituisce alla applicazione C#;

## 5.2 Funzionamento dell'infrastruttura

Andiamo ora a vedere più in dettaglio quali sono le componenti utilizzate per realizzare tutto questo.

Vi sono alcuni strumenti fondamentali:

- L'interfaccia **ISendable** che deve essere implementata da ogni classe che si vuole scambiare nell'infrastruttura (scritta sia in Java che C#);
- La classe **KSender** che rappresenta l'interfaccia dell'infrastruttura (scritta sia in Java che C#);
- Il server di informazioni chiamato **XmlServer** (realizzato in C#);

L'utente che decida di utilizzare l'infrastruttura si deve anzitutto accertare di quali siano i tipi di dati comuni tra gli ambienti. Per quanto riguarda Java e C# si è già discusso della questione nel capitolo 4

Fatto questo, l'utente dell'infrastruttura deve scegliere le classi che intende far scambiare tra i due ambienti. Deve, cioè, scrivere una versione in Java e una in C#, per ciascuna classe scelta tenendo conto del fatto che entrambe le versioni delle classi dovranno implementare l'interfaccia *ISendable*. Le librerie che contengono tali classi devono poi essere posizionate sui server di librerie (uno per ogni linguaggio) citati nella sezione precedente. L'utente deve registrare la posizione delle classi stesse sull'*XmlServer*.

La figura 5.2 mostra il funzionamento dell'infrastruttura, relativamente al funzionamento di *KSender* e dell'*XmlServer*. Nella figura vengono inoltre indicati i server di classi, indicati come *Server IIS* (per .NET) e un *Java-ClassServer* (per Java). La spiegazione di questi due strumenti è rimandata all'ultima sezione di questo capitolo.

Come si vede dalla figura 5.2 si considerano due applicazioni, una in ambiente Java una in .NET, che devono comunicare fra loro. L'unica cosa

che si scambiano direttamente sono delle particolari stringhe XML che per comodità, d'ora in avanti chiameremo **KMessage**. Queste stringhe sono documenti XML che contengono lo stato dell'oggetto inviato nonché un identificatore per la classe. Come già detto nel capitolo 3, la capacità di potersi scambiare stringhe è il requisito minimo per poter realizzare una qualche interoperabilità fra due sistemi.

All'interno delle due applicazioni vengono utilizzate le rispettive versioni (Java e C#) della classe *KSender* che come si vede dalla figura, si occupa di tutte le comunicazioni con l'XmlServer e con i server di oggetti, operazioni che risultano quindi trasparenti all'utente dell'infrastruttura. Il lavoro dell'utente sarà solo quello di inviare e ricevere stringhe tra le due applicazioni.

Il grosso del lavoro viene operato dai metodi `KSender.pack` e `KSender.unpack`. Il primo trasforma un oggetto che implementi `ISendable`, in una apposita stringa `KMessage`. Il secondo compie l'operazione inversa, ovvero estrae un oggetto `ISendable` da una stringa `KMessage`.

Figura 5.2: Infrastruttura

Questa seconda operazione è più complessa della precedente. Infatti, se l'oggetto da estrarre non è di una classe nota al sistema, il metodo deve rivolgersi al server di informazioni XmlServer per sapere da quale server di classi ottenerla.

Quindi il metodo `KSender.unpack`, si connette al server di classi opportuno e carica tutte le classi di cui necessita per creare l'oggetto voluto. A questo punto il sistema ha lo stato dell'oggetto in formato XML all'interno della stringa `KMessage`, e ha la classe dell'oggetto caricata in memoria. Per poter ottenere da questi elementi l'oggetto di partenza occorre utilizzare il serializzatore XML specifico per la classe. Ed è a questo punto che entra in gioco l'interfaccia `ISendable`. Questa fornisce il metodo `buildSerializer`, che restituisce il serializzatore XML per l'oggetto stesso. Per cui `KSender.unpack` crea un'istanza vuota della classe voluta e ne richiama il relativo metodo `buildSerializer`. A questo punto il metodo `unpack` può finalmente deserializzare l'oggetto contenuto nel `KMessage`. Sempre all'interno di `KSender` è implementato il metodo `KSender.Record` che permette di registrare le informazioni necessarie sull'`XmlServer`.

Passiamo ora ad analizzare l'interfaccia `ISendable`, la classe `KSender` e l'applicazione `XmlServer` più in dettaglio.

### 5.3 L'interfaccia `ISendable`

Il primo passo per la costruzione dell'infrastruttura è l'interfaccia `ISendable`. Tale interfaccia ha alcune differenze di implementazione a seconda del linguaggio in cui è utilizzata. Tuttavia il metodo più importante che richiede di implementare è il metodo `buildSerializer`. Tale metodo restituisce il serializzatore XML per la classe stessa. Il suo scopo è duplice. Da un lato

permette al deserializzatore di verificare se l'oggetto ricevuto è stato correttamente costruito per operare all'interno dell'infrastruttura; dall'altro, il metodo `buildSerializer` obbliga ciascuna classe a inglobare il proprio serializzatore XML. Infatti, come si è visto nel capitolo precedente, sia in Java che in C# non esiste un serializzatore XML generico, cioè tale che possa deserializzare una qualunque classe. Invece, in entrambi i casi, è richiesto che si conosca la classe da serializzare al momento della costruzione del serializzatore. Questo non è ovviamente possibile in fase di deserializzazione, quindi semplicemente richiamando questo metodo è possibile ottenere il deserializzatore corretto per una data classe.

Per cui, come si vedrà più avanti, in fase di deserializzazione, per prima cosa si costruisce un'istanza di default dell'oggetto `ISendable`, se ne richiama il metodo `buildSerializer`, si ottiene il serializzatore giusto per quella classe, e quindi si può eseguire la deserializzazione del messaggio XML ricevuto in modo corretto.

Per `buildSerializer` sono presenti due versioni; una senza parametri per gli oggetti più semplici, e una che prende come parametro un array di classi. Questa versione è pensata per costruire i serializzatori per oggetti più complessi che hanno altri oggetti al loro interno. Infatti come si è detto nel capitolo precedente un `XmlSerializer` per oggetti complessi richiede, in entrambi i linguaggi, di conoscere tutte le classi che dovrà deserializzare.

### 5.3.1 Versione C#

L'implementazione C# della versione di `buildSerializer` senza parametri è banale. Infatti è sufficiente una sola riga:

```
return new XmlSerializer(this)
```

In questo caso si sfrutta il serializzatore di default di C#.

La seconda versione richiede come parametro un array di oggetti `Type`. In tal modo è possibile utilizzare l'apposito costruttore del serializzatore standard di C#, per cui anche in questo caso l'implementazione è una sola riga, simile alla seguente:

```
return new XmlSerializer(this,extraTypes)
```

dove `extraTypes` indica il vettore di `Type` che viene passato come parametro.

Tuttavia non tutte le classi sono serializzabili con `XmlSerializer`. Ad esempio la classe `HashTable` non può essere serializzata in XML.

### 5.3.2 Versione Java

Dal lato Java occorre più lavoro. Occorre utilizzare il serializzatore personalizzato `XmlSerializer` descritto nel capitolo 4, per cui si dovrà costruire un apposito *mappig* per la propria classe, tramite un `XmlSerializeRegistry`, anch'esso descritto nel capitolo 4.

In questo caso il costruttore con un parametro richiede un `XmlSerializeRegistry`, nel quale si richiede che siano stati registrati i mapping per tutte le altre classi contenute nell'oggetto. Per la versione Java dell'interfaccia sono stati aggiunti anche i metodi `getDefaultRegistry` e `getUsedClassNames`. Il primo restituisce il registro del serializzatore definito per quel dato oggetto `ISendable`. Il secondo restituisce un oggetto `Vector` contenente i nomi delle altre classi contenute nell'oggetto `ISendable`. Questi due metodi sono utili, nei processi di serializzazione e deserializzazione, come si vedrà alla sezione 5.5.

## 5.4 L'XmlServer

L'XmlServer costituisce l'implementazione di un server di informazioni ed è il secondo strumento fondamentale dell'infrastruttura. Il suo compito è quello di fornire e registrare le locazioni — o URL (Unique Resource Locator) — dove vengono poste le librerie con le classi che devono essere scambiate nell'infrastruttura. Tutte le operazioni passano dalla gestione di un piccolo registro, gestito tramite l'oggetto **reg**, istanza della classe *RegistroDati*. In tale registro vengono registrati due tipi di informazioni. La prima è l'associazione tra una classe e il server di librerie su cui è registrata. Per far questo, in tale registro sono inseriti dei record che contengono queste quattro informazioni: nomeclasse, linguaggio, URL e port. Il primo è il nome della classe a cui ci si riferisce, il secondo è il linguaggio in cui è scritta, il terzo e il quarto sono l'url e il port del server di librerie su cui è stata registrata la libreria dove è definita quella particolare classe. La seconda informazione registrata sull'XmlServer è l'associazione tra due classi che sono l'una la traduzione dell'altra in linguaggi diversi. Entrambe queste informazioni sono mantenute tramite l'utilizzo di una HashTable, i cui dettagli di funzionamento saranno descritti nella prossima sottosezione.

Tutto il dialogo con i client che richiedono informazioni all'XmlServer avviene tramite stringhe XML. Il funzionamento del server è brevemente questo. Il server resta in attesa sul port specificato dalla variabile `PORT_R` (se non è specificato il valore di default è il 9001). Quando arriva una richiesta di connessione tramite il metodo *ricevi*, cerca di leggere una stringa XML dal Network. In tal modo può comunicare sia con l'ambiente .NET che con l'ambiente Java, sia con qualunque altra sorgente che sia in grado di inviare stringhe XML. Il documento XML ricevuto viene interpretato come un oggetto *message* serializzato (vedi prossima sezione).

Tale oggetto message può essere una richiesta di dati o una registrazione di nuovi dati.

Nel primo caso viene richiamato il metodo `ManageRequest` che provvede a interrogare l'oggetto *reg* per ottenere l'informazione richiesta, e poi invia la risposta corretta al client, sempre sotto forma di oggetto *message* serializzato in XML.

Nel secondo caso viene richiamato il metodo `ManageRecord` che invece provvederà ad accedere all'oggetto *reg* per inserire l'informazione ricevuta. Se ha successo invierà un messaggio di ok.

In entrambe le situazioni, un errore provocherà il fallimento dell'operazione e l'invio al client di un oggetto *message* di errore, sempre serializzato in XML.

### 5.4.1 Dettagli implementativi

Si è scelto di realizzare il server in C#, ma si può procedere in maniera simile anche in Java.

Vediamo anzitutto quali caratteristiche deve avere il server di informazioni

1. Il server deve essere in grado di soddisfare più richieste contemporaneamente.
2. Il server deve essere in grado di comunicare con tutti gli ambienti diversi che utilizzano l'infrastruttura.

Il punto 1 è stato risolto tramite l'utilizzo dei *thread*. Il meccanismo di funzionamento dei thread in C# è stato descritto nel capitolo 3. I thread sono stati utilizzati per ottenere nuove versioni concorrenti del metodo *ricevi*, una per ciascuna richiesta di connessione.

Il punto 2, ovvero la necessità per il server di comunicare con ogni altro ambiente dell'infrastruttura, è stato risolto utilizzando XML. Come già detto in precedenza il server si aspetta di ricevere delle stringhe XML che rappresentano la versione serializzata della classe *message*.

**La classe message** La classe *message* è costituita dai seguenti campi:

- *type*
- *nomeclasseSource*
- *linguaggioSource*
- *urlSource*
- *portSource*
- *nomeclasseDest*
- *linguaggioDest*
- *urlDest*
- *portDest*
- *info*

Tutti i campi di un oggetto *url* sono stringhe salvo i campi *portSource*, *portDest* che sono interi.

Il campo *type* descrive il tipo di messaggio. Possono esserci quattro tipi di oggetti *message*:

- **INSERT**: Il messaggio indica che i dati in esso contenuti devono essere registrati sul server;

- **REQUEST**: il messaggio è una richiesta di dati;
- **RESPONSE\_KO**: indica un messaggio di errore, transazione non completata correttamente;
- **RESPONSE\_OK**: indica che la transazione è andata a buon fine;

Il campo *info* viene utilizzato per documentare gli errori.

Tutti gli altri campi sono le informazioni che i diversi client ricercheranno per poter localizzare le librerie contenenti le definizioni delle diverse classi.

**Utilizzo di Message** L'utilizzo di questa classe è il seguente. Per registrare una coppia di classi occorre inviare un messaggio del tipo:

```
[ INSERT , "pippo" , "java" , aaa.bbb.ccc.ddd , 1234 , "pluto" ,  
  "sharp" , eee.fff.ggg.hhh , 5678 , - , ]
```

In questo modo si sono registrate sia le informazioni relative ai server di librerie che l'associazione tra le due classi.

Successivamente quando il server riceve un messaggio del tipo:

```
[ REQUEST , "pippo" , "java" , - , - , "sharp" , - , - ]
```

Il server restituirà il messaggio:

```
[ INSERT , "pippo" , "java" , aaa.bbb.ccc.ddd , 1234 , "pluto" ,  
  "sharp" , eee.fff.ggg.hhh , 5678 , - , ]
```

**Il RegistroDati** L'XmlServer registra tutti i propri dati tramite l'oggetto *reg*, di tipo *RegistroDati*. Questa classe gestisce il file delle registrazioni. I suoi metodi più importanti sono **Immetti** ed **Estrai**. Entrambi richiedono un oggetto *message* come argomento. Tuttavia il metodo **Immetti** richiede

che l'oggetto message sia di tipo INSERT, mentre **Estrai** richiede che sia di tipo REQUEST. In caso contrario verrà generata un'eccezione.

Il registro mantiene le sue informazioni in una HashTable bidimensionale che ha come chiave un oggetto di tipo *Chiave* e come valore ad essa associato un oggetto di tipo *languageRecord*.

L'oggetto Chiave<sup>1</sup>. è costituito da tre campi: *nomeclasseSource*, *ling-Source* e *langDest*.

L'oggetto languageRecord contiene invece i campi *classname*, *port*, e *url*.

Quando si richiama il metodo *Immetti*, l'oggetto message che viene passato come parametro determina quali informazioni vengono inserite nella tabella hash.

Supponiamo che l'oggetto message sia di questo tipo:

```
[ INSERT , name1 , lang1 , host1 ,
  port1 , name2 , lang2 , host2 , port2 ]
```

Allora saranno create le associazioni indicate nella tabella 5.1

Tabella Hash						
Chiave				Valore		
name1	lang1	lang2	→	name2	host2	port2
name2	lang2	lang1	→	name1	host1	port1
name1	lang1	lang1	→	name1	host1	port1
name2	lang2	lang2	→	name2	host2	port2

Tabella 5.1: Associazioni in una hashtable

Il metodo *Estrai* invece riceve un oggetto message di tipo REQUEST.

<sup>1</sup>Per utilizzare l'oggetto Chiave come chiave di una tabella Hash si sono dovuti ridefinire i metodi GetHashCode e Equals che, ereditati di default da object, non fornivano il risultato corretto

Da questo costruisce un oggetto Chiave per interrogare la Hashtable. Estrae quindi l'oggetto `languageRecord` ad essa correlato e restituisce infine un nuovo oggetto `message` con i dati ritrovati nella tabella. In caso di errore restituisce `null`.

La tabella viene registrata su disco, su un file denominato `Registro.dat` — ma il nome può essere modificato dall'utente) —, ogni volta che si richiama il metodo `textttAggiorna`.

É interessante notare che la tabella viene utilizzata in un contesto di programmazione concorrente. Infatti, come si è visto in precedenza, vi possono essere più threads che accedono contemporaneamente all'hashtable, e al file in particolare. Per risolvere questo problema, la stessa classe `HashTable` delle librerie `C#` permette di sincronizzare gli accessi alla tabella stessa. La seguente riga di codice mostra come sia possibile estrarre una `HashTable` “a prova di thread” da quella di partenza:

```
Hashtable safe = Hashtable.Synchronized(ash);
```

Sempre utilizzando questo accorgimento si evitano problemi quando si opera sulla `HashTable` con il metodo `Estrai`, richiamato dall'`XmlServer` quando si ricerca una particolare Chiave.

## 5.5 KSender

*KSender* è la classe fondamentale per l'invio e la ricezione degli oggetti nell'infrastruttura. La classe contiene i seguenti metodi:

- `Init` - inizializza il `KSender`;
- `pack` - costruisce un `KMessage` da un oggetto `ISendable`;
- `unpack` - restituisce un oggetto `ISendable` da un `KMessage`;

- record - registra delle informazioni su un XmlServer;
- request - per uso interno, richiede delle informazioni all'XmlServer;
- getNomeclasse - Estrae il nome della classe del KMessage;
- getXmlInfoServerHost - Estrae il contenuto dell'elemento XmlInfoServerHost;
- getXmlInfoServerPort - Estrae il contenuto dell'elemento XmlInfoServerPort;

La classe KSender nasconde all'utente il meccanismo di funzionamento dell'infrastruttura stessa. Una volta che si è attivato un XmlServer è possibile utilizzare i metodi della classe KSender.

La classe contiene tre campi SERVER, PORT e IPSERVER. Essi permettono di individuare l'host e il port dell'XmlServer di riferimento. L'inizializzazione di questi campi avviene tramite il metodo *KSender.Init*. Solo dopo l'inizializzazione si possono utilizzare gli altri metodi. I tre campi saranno quelli a cui faranno implicitamente riferimento i metodi **KSender.pack** e **KSender.unpack**. La figura 5.3 esemplifica il funzionamento di questi due metodi

Anzitutto occorre ricordare che la comunicazione tra i due ambienti avviene in Xml ma, in particolare, vengono scambiate delle stringhe. Per cui all'utente viene lasciato il compito di gestire l'invio e la ricezione di stringhe tra i due ambienti, mentre i metodi *pack* e *unpack* permettono di gestire queste stringhe. In altre parole, se supponiamo di voler inviare un oggetto **ISendable** da una applicazione Java ad una C# si eseguono le seguenti operazioni:

Figura 5.3: Funzionamento dei metodi pack e unpack

1. Si richiama `KSender.pack` che trasforma l'oggetto in una stringa particolare (detta `KMessage`);
2. si invia la stringa `KMessage` all'ambiente `C#`, ad esempio tramite socket;
3. Dal lato `C#`, la stringa ricevuta viene passata al metodo `KSender.unpack` che restituisce l'oggetto `ISendable` ma in versione `C#` e con lo stesso stato di quello di partenza;

**KMessage** Una stringa `KMessage` è un documento Xml la cui struttura è riportata nella figura 5.4

Quindi in breve ecco il significato degli elementi utilizzati: oltre all'elemento radice `KMessage`, elemento root del documento XML, e l'elemento `Packet`, — al punto (2) in figura— che contiene la serializzazione dello stato di un oggetto, troviamo gli elementi `Nomeclasse` e `Language` che ci indicano il nome e il linguaggio della classe che l'oggetto `Packet` contiene. L'elemento `UsedTypes` può contenere — nel punto contrassegnato con (1) in figura —

```
<?xml version="1.0" encoding="utf-8"?>
<KMessage>
  <Nomeclasse>.....</Nomeclasse>
  <Language>.....</Language>
  <UsedTypes>
    <usedType>.....</usedType>
    (1)
    .....
  </UsedTypes>
  <XmlInfoServerHost>.....</XmlInfoServerHost>
  <XmlInfoServerPort>.....</XmlInfoServerPort>
  <Packet>
    ....
    (2)(qui c'è l'oggetto ISendable Serializzato}
    ....
  </Packet>
</KMessage>
```

Figura 5.4: Documento KMessage

un numero variabile di elementi `usedType` ciascuno dei quali contiene il nome di una eventuale classe contenuta all'interno dell'oggetto inviato. Infine abbiamo gli elementi `XmlInfoServerHost` e `XmlInfoServerPort` che indicano dove è localizzato il server di informazioni a cui richiedere la locazione delle librerie (assembly o file class) che definiscono l'oggetto contenuto nel `Packet`.

Vediamo ora in dettaglio come sono state realizzati i diversi metodi nelle due versioni di `KSender`, in Java e C#.

### 5.5.1 Pack

Questo metodo consente di inserire un oggetto `ISendable` in un `KMessage`. Esistono due versioni del metodo `pack`, una pubblica e una protetta per uso interno. In entrambe le versioni Java e C# il metodo richiede come parametro in ingresso un oggetto `ISendable`. Il metodo restituisce una stringa che è un `KMessage`.

Ecco le signature nelle due versioni:

Signature C#:

```
public string pack(ISendable obj)
```

Signature Java:

```
public String pack(ISendable src)
```

Per costruire il `KMessage` il metodo utilizza il serializzatore ottenuto tramite il metodo `buildSerializer` dell'oggetto `ISendable` dato. Si fa uso dei metodi `getTypeUsedList` in C# e del suo analogo Java `getExtraTypes`, per costruire gli elementi `usedType` contenuti nell'elemento `UsedTypes`.

I valori per gli elementi `XmlInfoServerPort` e `XmlInfoServerHost` vengono ricavati dai campi di `KSender` descritti in precedenza. L'elemento `Nomeclasse`

si ricava a partire dall'oggetto `ISendable` dato. Per C# si utilizzano le seguenti linee di comando, dove `src` è l'oggetto da serializzare:

```
Type t=obj.GetType();  
string nomeclasse=t.FullName+", "+t.Assembly.GetName().Name;
```

Si noti che è necessario inserire il nome dell'assembly in cui è localizzata la classe. In Java si utilizza, invece, la seguente unica linea: `String nomeclasse=src.getClass().getName()` Viene ora presentato più in dettaglio il funzionamento dell'algoritmo, analogo in entrambe le versioni.

1. Si invoca il metodo `buildSerializer` sull'oggetto `ISendable` dato, ottenendo il serializzatore XML per l'oggetto;
2. Il metodo serializza lo stato dell'oggetto `ISendable` su una stringa temporanea chiamata `pacco`, utilizzando il serializzatore ottenuto al punto 1;
3. Si inizializza una stringa `result` come stringa vuota.
4. A questo punto si inseriscono nella stringa `result` gli elementi del `KMessage`, secondo quanto descritto in precedenza nella figura 5.4. Ad esempio per aggiungere il nome della classe, contenuto nella stringa `nomeclasse`: `result+=«Nomeclasse»+nomeclasse+«/Nomeclasse»;`
5. Arrivati all'elemento `Packet` si inserisce il contenuto della stringa `pacco`, ottenuta al punto 2, come segue: `result+=«Packet»+pacco+«/Packet»;`
6. Completato il `KMessage`, il metodo restituisce il valore della stringa `result`

### 5.5.2 Unpack

Questo metodo riceve una stringa KMessage e restituisce l'oggetto ISendable in essa contenuto. In caso di errore solleva un'eccezione. Ecco le signature nelle due versioni:

Signature C#

```
public static ISendable unpack(string kmessage)
```

Signature Java

```
public static ISendable unpack(String KMessage)
```

La figura 5.5 mostra il dialogo tra il metodo unpack, l'XmlServer e il server di oggetti.

Vediamo quindi come opera il metodo:

1. Come primo passo il metodo opera il parsing della stringa come documento xml. Per far questo utilizza le apposite API fornite dal linguaggio:
  - In C# costruisce un oggetto *XmlDocument* e opera il parsing tramite il metodo *LoadXml*;
  - In Java utilizza la classe *DocumentBuilder* e il metodo *parse* per costruire un oggetto *Document*;
2. Dal documento così ottenuto, tramite il metodo *KSender.getNomeclasse* si ottiene il nome della classe da localizzare;
3. Tramite il metodo *getXmlUsedTypes* si ottiene, un array gli oggetti *Type* (o *Class* nella versione in Java) ricavandoli dagli elementi *usedType* contenuti nel *KMessage*;
4. Si cerca il file class (o l'assembly) relativo alla classe sulla macchina locale.
5. Se la classe è sconosciuta occorre ricercarla tra i server di oggetti. Si richiama quindi il metodo *KSender.request* per ottenere l'url e il port del server di oggetti in cui è localizzato l'assembly o il file class che contiene la classe data.
6. Se l'operazione di caricamento ha successo il metodo provvede al caricamento della classe e ne crea una istanza generica.
7. Da questa istanza si ottiene, tramite il metodo *buildSerializer* e sfruttando l'array ottenuto al punto 3, il serializzatore per la classe;

8. Infine tramite il serializzatore è possibile deserializzare l'oggetto contenuto nell'elemento `Packet` del *KMessage*.
9. restituisce questo oggetto e termina.

### 5.5.3 Il metodo request

Il metodo `request` serve al metodo `pack` per richiedere all'`XmlServer` l'url del server di oggetti a cui richiedere la libreria (assembly o file class) contenente la classe cercata. Come si è visto nella sezione ?? questo si traduce nell'inviare un message di tipo `REQUEST` all'`XmlServer`.

Le signature del metodo sono:

Signature C#:

```
message request(string nomeclasse, string languagesource)
```

Signature Java

```
message request(String Nomeclasse, String language)
```

Quindi il metodo invierà un oggetto message del tipo:

```
[REQUEST, nomeclassesource, languagesource, -, -, languagedest, -, -].
```

e, come risposta, in caso di successo, otterrà un messaggio del tipo `[INSERT, nomeclasse, languagesource, aaa.bbb.ccc.ddd, 1234, nomeclassedest, languagedest, eee.fff.ggg.hhh, 5678,-]`

Brevemente, il metodo funziona come segue:

1. Per prima cosa si costruisce un oggetto `message` di tipo `REQUEST`, come detto in precedenza.

2. Questo oggetto viene serializzato in Xml tramite l'opportuno serializzatore.
3. L'oggetto message così serializzato viene inviato all'XmlServer. Il metodo resta in attesa della risposta del server.
4. Appena riceve la stringa di risposta dal server la deserializza di nuovo in un oggetto message e lo ritorna. Se si genera un'eccezione ritorna un oggetto nullo.

#### 5.5.4 Il metodo record

Il metodo record, è utilizzato dall'utente per registrare le informazioni circa la localizzazione dell'assembly o del file class di una data classe. Come si è visto nella sezione ?? questo si traduce nell'inviare un oggetto message di tipo INSERT. Ecco le signature per questo metodo:

Signature C#

```
public message record(string nomeclasseSource,
    string linguaggioSource, string urlSource, int portSource,
    string nomeclasseDest, string linguaggioDest, string urlDest,
    int portDest)
```

Signature Java

```
public message record(String NomeclasseSource, String urlSource,
    int portSource, String linguaggioSource,
    String NomeclasseDest, String urlDest,*
    int portDest, String linguaggioDest)
```

il funzionamento è il seguente:

1. si costruisce un oggetto message di tipo INSERT con i campi corrispondenti ai parametri d'ingresso;
2. si serializza tale oggetto con l'opportuno serializzatore;

3. si connette all'XmlServer centralizzato e resta in attesa del message di risposta;
4. appena riceve un message di risposta lo deserializza e lo ritorna;

## 5.6 I server di classi

Come ultimo punto per la descrizione dell'infrastruttura occorre parlare dei server da cui vengono scaricati gli assembly e i file class. I metodi possono essere diversi, nel nostro caso abbiamo utilizzato per il lato C# una semplice directory condivisa gestita dall' IIS di Windows. Per cui il campo url registrato sull'XmlServer è proprio l'indirizzo esatto di dove è localizzato l'assembly.

Dal lato Java, invece si è utilizzato essenzialmente il NetworkClassLoader di Bettini e Cappetta [6], all'interno dell'applicazione JavaClassServer. Il campo url degli oggetti message da Java è quindi necessariamente l'host su cui questo server è attivo — per semplicità si è supposto che il servizio sia attivo sempre sul port 9090. In particolare la versione Java del KSender contiene il metodo *classLoad* che si connette al JavaClassServer. In tal modo per poter rendere disponibili i file class occorre posizionarli nel CLASSPATH del sistema su cui è attivato il JavaClassServer.

## Capitolo 6

# Interoperabilità e mobilità: il framework FAL

Nel capitolo 2 si è messo in evidenza che uno strumento come l'architettura Java, sfruttando le sue caratteristiche di interoperabilità, risulti molto utile per la realizzazione di infrastrutture per la mobilità; a tale proposito si sono citate infrastrutture per la mobilità sviluppate interamente nel linguaggio Java — come Klava o Aglets —, realizzate, dunque, utilizzando un linguaggio unico.

In quello stesso capitolo si faceva anche presente di come sarebbe stato interessante indagare sulla possibilità di sviluppare una infrastruttura per la mobilità tramite un insieme di linguaggi diversi. L'infrastruttura presentata in questa tesi è stata sviluppata proprio con questa idea in mente.

In questo capitolo si mostra come poter trarre vantaggio dall'infrastruttura presentata in un contesto di mobilità, tramite un semplice framework per agenti mobili, FAL. Inizialmente, si è pensato a come integrare l'infrastruttura nell'ambiente Klava. Successivamente, per motivi di tempo, ci si è limitati a realizzare FAL.

## 6.1 FAL

In questa sezione viene descritto il framework per agenti mobili FAL (Framework for Agents and Localities) ispirato al linguaggio DII [19].

Il sistema FAL è costituito da un insieme di *nodi*, ognuno dei quali è l'ambiente di esecuzione per gli *agenti*. Questi interagiscono localmente ad un nodo e possono scambiarsi oggetti tramite appositi *canali*, che costituiscono delle astrazioni per delle code di messaggi.

Gli agenti in un nodo possono eseguire tre operazioni primitive per lo scambio di oggetti: `send(ch,o)`, `receive(ch)`, `receiveW(ch)`.

`send(ch,o)` tramite questa primitiva è possibile mettere un oggetto in un canale.

`receiveW(ch)` tramite questa primitiva viene recuperato un oggetto da un canale specificato. Rimane bloccato finché non ne arriva uno.

`receive(ch)` come `receiveW` ma è non bloccante.

Ogni nodo viene associato ad una *località*, che in pratica ne rappresenta l'indirizzo IP. É inoltre possibile valutare un agente in un nodo remoto. Per far questo si utilizza la primitiva `eval(A,l)` che consente di inviare l'agente A sulla località l per eseguirne la valutazione.

## 6.2 Descrizione del sistema

Veniamo ora a descrivere quali sono le classi che costituiscono il sistema FAL. Prima viene data una descrizione generica di ciascuna classe che ne illustra i membri e il funzionamento rispetto alle altre classi del sistema.

Nella sezione successivamente viene esposta anche la realizzazione del sistema nei linguaggi Java e C# che sfrutta l'infrastruttura realizzata nella tesi.

Le classi che compongono il sistema FAL sono quattro: Node, Agent, Channel e Locality.

**Channel** La classe channel identifica un buffer in cui i processi depositano e recuperano oggetti. È caratterizzata da un nome (channelName).

**Locality** La classe locality serve per individuare un nodo della rete FAL, indicandone url e porta.

**Agent** La classe Agent rappresenta l'unità computazionale di base. Un agente è caratterizzato da un nome —agentName— e un hostnode, ovvero il nodo dove si trova in un dato momento. L'agente contiene un metodo *execute* che verrà invocato al momento dell'esecuzione all'interno di un nodo.

**Node** La classe Node rappresenta un nodo del sistema FAL. Gli oggetti della classe Node vengono utilizzati come ambienti computazionali dagli Agent e forniscono le primitive di comunicazione send, receive e receiveW. Tramite il metodo addAgent, inoltre, il nodo manda in esecuzione un nuovo agente. Infine il metodo Eval(A,l) invia l'agente A per la valutazione alla locazione l.

Per le comunicazioni interne all'infrastruttura — come per il metodo Eval — si sfrutta l'infrastruttura utilizzata nella tesi. Per tale motivo, le classi Agent, Channel e Locality che possono aver bisogno di sfruttare l'infrastruttura implementano l'interfaccia ISendable. Si noti, infatti, che al contrario delle altre la classe Node è statica, nel senso che non migra attraverso la rete.

## 6.3 Dettagli implementativi

In questa sezione verranno descritti i dettagli implementativi delle quattro classi principali di una rete FAL, relativamente alle due implementazioni Java e C#.

Per ciascuna di esse vengono spiegati brevemente i metodi comuni e il loro funzionamento generale. Se vi sono delle differenze particolari tra le due implementazioni si passa alla descrizione dei dettagli specifici che riguardano tali differenze.

### 6.3.1 La classe Agent

La classe Agent rappresenta un Agente nel sistema FAL. È una classe astratta, il metodo *execute()*, infatti, deve essere implementato dall'utente, con il codice personalizzando il codice dell'agente.

La classe Agent ha due campi *hostNode*, che rappresenta il Nodo in cui si trova in un determinato momento l'agente, e *agentName* una stringa che rappresenta il nome dell'agente stesso e consente di identificarlo.

Naturalmente questa classe implementa l'interfaccia *ISendable*, poiché dovrà migrare all'interno della rete FAL.

#### Versione C#

La versione C# di Agent lascia astratti i metodi dell'interfaccia *ISendable*. Questo perché il compilatore non permette di definire l'*XmlSerializer* interno di C# per una classe astratta.

## Versione Java

La versione Java di Agent ha molte particolarità. Anzitutto oltre ad implementare `ISendable`, l'Agent Java deriva dalla classe `Thread`. Pertanto può essere gestita come un `Thread` e, ad esempio implementa il metodo `run`, che però si limita a richiamare `execute`. Solo in questo modo è possibile sfruttare tutti i meccanismi messi a disposizione da Java per la gestione avanzata dei processi.

Oltre a questo, per quanto riguarda il meccanismo di serializzazione è stato realizzato un apposito *ClassMarshaller* chiamato `AgentSerializer`, per permettere la serializzazione XML di un agente. A differenza del caso di C# è stato implementato anche il metodo `buildSerializer`, che sfrutta l'`AgentSerializer`. Tuttavia è comunque opportuno ridefinire il metodo quando si deriva una nuova classe da `Agent`, perché altrimenti il meccanismo di serializzazione non funziona, non essendo stato definito nessun mapping per la nuova classe —per i dettagli si veda il capitolo 4

### 6.3.2 La classe Node

La classe `Node` è la classe più importante del sistema FAL. Essa rappresenta l'ambiente di esecuzione per gli agenti, fornendo le primitive di comunicazione e i canali d'invio e ricezione degli oggetti.

La classe `node`, in entrambe le implementazioni, ha i seguenti campi:

- *sender*: è il `KSender` di riferimento, tramite il quale è possibile sfruttare l'infrastruttura proposta nella tesi.
- *self*: è la *locality* del `Node` stesso;
- *channelQueue* è la tabella hash utilizzata per la gestione degli oggetti depositati sui diversi canali.

- *newChannel* variabile intera, che indica il numero di canali presenti sul nodo;

Veniamo ora ai principali metodi della classe.

**Addagent** *Signature: Addagent(Agent a)*

Questo metodo consente di mandare in esecuzione un agente sul nodo, ovvero invoca il metodo *execute* di quell'agente. Questa operazione è leggermente diversa nelle due implementazioni. Infatti l'esecuzione viene eseguita come un thread. Per cui, dato il diverso meccanismo di gestione dei thread nel caso di C# viene costruito un nuovo Thread del metodo *execute* dell'agente dato, mentre dal lato Java viene semplicemente invocato il metodo *start* dell'agente, poiché, in Java un agente è un thread.

**Eval** *Signature: void Eval(Agent a, Locality l)* Tramite Eval è possibile inviare un agente su un'altra locazione, e dunque su un altro nodo per poterlo valutare. Il funzionamento, analogo per entrambe le implementazioni è il seguente. Eval richiede come parametri un oggetto Agent e un oggetto locality. Se la locality è quella del nodo stesso allora valuta l'agente in locale. Altrimenti tramite il metodo *send* di Locality invia l'agente sull'altro nodo, utilizzando l'oggetto *sender* di riferimento.

**Receive** *Signature: object Receive(Channel ch)* Questo metodo opera la ricezione degli oggetti su un canale. Questo viene realizzato utilizzando una tabella hash *channelQueue*. La prima ha come chiave un oggetto Channel e come valore possibile un oggetto Queue, che rappresenta l'insieme di oggetti presenti sul canale, organizzato come coda FIFO. Quando viene invocato questo metodo, si controlla se l'oggetto Channel, dato come parametro, è già presente come chiave nella tabella hash. In caso affermativo il metodo

ritorna il primo oggetto della coda. Altrimenti, in tutti gli altri casi, ritorna null.

É da sottolineare che poiché la tabella hash è una risorsa condivisa fra tutti i thread in esecuzione su un nodo, l'accesso alla stessa deve essere compiuto tramite mutua esclusione. Si è già detto che i meccanismi che realizzano questo in Java e C# sono molto diversi come già si è visto nel capitolo 5 per la gestione del registro dati dell'XmlServer.

In questo caso in C# il problema è risolto direttamente dall'oggetto HashTable che tramite il metodo statico `Hastable.Synchronized()`, fornisce una tabella hash thread-safe.

In Java, invece, occorre utilizzare il costrutto `synchronized` sull'oggetto channel che fa da chiave per la tabella hash.

**ReceiveW** *Signature:ReceiveW(Channel ch)* Questo metodo realizza la Receive bloccante. Questo viene realizzato, sia in Java che in C# tramite un loop infinito — realizzato con un comando `while(true)` — dal quale è possibile uscire solamente quando si ritorna l'oggetto ricevuto. Per il resto il funzionamento è in tutto analogo a Receive, comprese le considerazioni sull'accesso concorrente alla tabella hash.

**Send** *Signature:Send(Channel ch, object o)* Con il metodo Send è possibile posizionare un oggetto su un canale. Come si è già visto nella descrizione del metodo Receive, questo si traduce con l'inserire tale oggetto in una hashtable. Anche in questo caso valgono tutte le considerazioni fatte per quello che riguarda l'accesso concorrente alla tabella hash.

**Start** Questo metodo viene invocato direttamente dal costruttore e avvia un thread — `NodeReceiver` — per la ricezione degli agenti sul Nodo. Tale

thread resta in attesa sul port di riferimento del Nodo. Ciò che si aspetta è una stringa da passare al KSender di riferimento. Questo deserializza l'oggetto ricevuto. Se l'oggetto è un agente lo valuta tramite Addagent. Altrimenti emette un'eccezione.

### 6.3.3 La classe Locality

La classe Locality, che consente di individuare un nodo all'interno della rete presenta pochissime differenze nelle due implementazioni. Questa classe ha 2 campi url e port che identificano l'url e il port su cui — sperabilmente — è in esecuzione un oggetto Node. Questa classe implementa ISendable, ma soprattutto utilizza l'infrastruttura realizzata nella tesi, nel metodo *send*. La signature esatta del metodo è

$$send(Ksender\ sender, ISendable\ o)$$

L'effetto della chiamata di questo metodo è quello di inviare l'oggetto *o* che implementa ISendable sul Nodo a cui la Locazione fa riferimento, utilizzando il KSender specificato nell'argomento. Il funzionamento è semplice. si richiama il metodo pack del KSender e si ottiene la stringa da inviare. Dopo di che questa stringa viene inviata al nodo, che lo recupera tramite il NodeReceiver.

Non vi sono differenze di rilievo nelle due implementazioni della classe Locality in Java e C#.

### 6.3.4 La classe Channel

La classe Channel rappresenta semplicemente il nome un canale su cui possono venire scambiati oggetti sulla rete. Implementa ISendable e ha un unico

campo che è la stringa `channelName` che rappresenta il nome del canale.

Anche per questa classe non vi sono differenze di rilievo nelle due diverse implementazioni. L'unica differenza è che per Java devo avere i `getChannelName` e `setChannelName` per poter serializzare il `channelName`, tramite il serializzatore personalizzato, mentre dal lato C# è sufficiente che `channelName` sia un campo pubblico.

# Capitolo 7

## Conclusioni

In questa tesi è stata presentata una infrastruttura che consente di far scambiare oggetti tra due linguaggi diversi. In particolare ne è stata realizzata un'implementazione nei linguaggi Java e C#. Sono state realizzate alcune classi di interfaccia tramite le quali è possibile per una qualunque applicazione scritta in uno di questi due linguaggi, utilizzare l'infrastruttura.

Il programmatore che vuole far migrare una classe tra i due ambienti deve implementarne due versioni, una Java e l'altra C#. Ciascuna delle due versioni deve essere posta su un apposito *server di classi*. Infine occorre registrare la posizione di ciascuna classe su un *server di informazioni*. A questo punto lo scambio di oggetti avviene tramite l'invio di messaggi XML, contenenti solo lo stato dell'oggetto da scambiare. L'infrastruttura provvede a recuperare il file class(nel caso Java) o l'assembly (nel caso C#), dell'oggetto inviato tramite i server di informazioni e i server di classi.

A questo punto l'infrastruttura crea un'istanza dell'oggetto di partenza nel linguaggio di destinazione, a partire dal messaggio XML ricevuto.

Nella tesi viene mostrato il framework per agenti mobili FAL, come esemplificazione di utilizzo dell'infrastruttura in un ambiente per la mobilità.

L'infrastruttura realizza un meccanismo per l'interoperabilità tra l'ambiente .NET e l'ambiente Java e di conseguenza con Sun One. Fino ad oggi l'interoperabilità tra questi due ambienti era possibile solo a livello di servizi web, almeno stando alla documentazione ufficiale di Sun One in [16].

Alla tesi viene inoltre allegato un CD con i sorgenti dell'implementazione Java e C# realizzata.

## 7.1 Limiti

Nell'implementazione dell'infrastruttura, i server di informazioni lavorano in modo indipendente. Per ovviare a problemi di inconsistenza sarebbe opportuno che fosse presente uno solo di tali server. Questo rappresenta un limite per l'utilizzo dell'infrastruttura in un contesto più generale, come ad esempio Internet. Inoltre, l'implementazione attuale non consente la modifica dei dati registrati sull'XmlServer. Nella realizzazione della tesi non si sono presi in considerazione le problematiche sulla sicurezza del codice, problematica affrontata in modo diverso da ciascun ambiente che si va ad affrontare, e che meriterebbe uno studio a se.

## 7.2 Possibili Estensioni

Possibili estensioni o lavori futuri sull'infrastruttura possono essere:

- Realizzare le API per l'infrastruttura in altri linguaggi orientati ad oggetti e in grado di caricare codice in modo dinamico a runtime, come ad esempio Eiffel.
- Proseguire l'esempio mostrato con FAL, sfruttando una infrastruttura per la mobilità vero, come ad esempio Klava, sfruttando anche il fatto

---

che l'infrastruttura ha già pronte le API per il linguaggio Java. In tal modo si potrà apprezzare a pieno le possibilità offerte dall'infrastruttura in un contesto per la mobilità.

- Estendere l'infrastruttura per utilizzare un numero variabile di server di informazioni, per esempio utilizzando un meccanismo di replicazione di dati tra i server.
- É possibile sfruttare il meccanismo di interoperabilità fra linguaggi agevolato dall'infrastruttura .NET che consente di costruire delle librerie che possono facilmente interagire col codice scritto in un qualunque altro linguaggio. Tramite il framework realizzato nella tesi si può sfruttare a pieno questa caratteristica provando ad utilizzare, ad esempio, le stesse API realizzate in C#, utilizzando un altro linguaggio .NET come Visual Basic.NET.

# Bibliografia

- [1] [www.w3c.org](http://www.w3c.org). Sito ufficiale dell'organizzazione per gli standard sul World Wide Web.
- [2] T. Archer. *Inside C#*. Microsoft Press, 2001.
- [3] A. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.
- [4] K. Ballinger, J. Hawkins, and P. Kumar. Soap in the microsoft.net framework and visual studio.net. *.NET Framework Documentation*, Novembre 2000.
- [5] L. Bettini. Progetto e realizzazione di un linguaggio di programmazione per codice mobile, 1998.
- [6] L. Bettini and D. Cappetta. A network class loader in java 2. *www.javaworld.it*, 2001.
- [7] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software Practice and Experience*, 2000.
- [8] B. Eckel. *Thinking in Java*. Prentice Hall, 1998.

- 
- [9] ECMA. *Common Language Infrastructure (CLI) Partition I: Concepts and Architecture*, 1999.
- [10] G. Glass. ObjectSpaceVoyager Core Package Technical Overview. White Paper, 1999.
- [11] G. Heine. *Gprs from A-Z*. Artech House, Incorporated, April 2000.
- [12] A. S. Inc., editor. *PostScript Language Reference Manual*. Addison Wesley, 1985.
- [13] D. Lange. *Java Aglets Application Programming Interface*. IBM Corp white paper, Feb. 1997.
- [14] B. Meyer. *Eiffel: The Language*. Prentice Hall, <http://www.eiffel.com/index.html>, 1992.
- [15] Microsoft Corporation. *C# Language Specification*, 2000.
- [16] S. Microsystem. Sun One Architecture Guide. e-book, 2002.
- [17] D. . MIKADO Global Computing Project. Analysis of distribution structures: state of art.
- [18] Object Managemeng Group, <http://www.omg.org>. *CORBA: Architecture and Specification*, Aug. 1998.
- [19] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *25th Annual Symposium on Principles of Programming Languages (POPL)(San Diego, CA)*. ACM, 1998.
- [20] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

- [21] M. Straßer, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In *Proc. of the 2nd ECOOP Workshop on Mobile Object Systems*, 1996.
- [22] M. C. Tomescu. Web services between .net, java and ms soap toolkit. *www.csharpelp.com*, 2002.
- [23] B. E. Travis. *XML and SOAP programming for BizTalk Servers*. Microsoft Press, 2000.
- [24] Wireless Application Protocol Forum Ltd. Wireless Application Protocol - architecture specification. *Internal document*, page 20, 1998.
- [25] D. Wong, N. Paciorek, T. Walsh, and J. DiCelie. Concordia: An Infrastructure for Collaborating Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proc. of the 1st Int. Workshop, MA '97*, number 1219 in LNCS, pages 86–97. Springer-Verlag, 1997.

# Ringraziamenti

Con questa tesi si chiude una parte della mia vita. Quando iniziai, l'università, ormai diversi anni fa, ero una persona molto diversa da quella di oggi.

Molti fatti che mi sono succesi hanno contribuito a cambiarmi. Ma più che gli avvenimenti sono state le persone che ho incontrato.

Ho voluto fare una specie di bibliografia umana. Così magari chiedendo a qualcuno di questi che ringrazio uno potrà chiedere “Ma tu lo conosci Daniele Falassi ....?” e magari verrà a conoscenza di qualche storia bella — o brutta — su di me.

Ringrazio:

Chiara: Uber Alles, senza di lei certe cose di me non si capirebbero;

I miei genitori, Cristina e Vinicio: per l'infinita pazienza nei miei confronti in tutti questi anni;

Leonardo, mio fratello: sponsor di svariati componenti hardware presenti a casa nostra, pagati da lui e usati più spesso da me;

A Michele e al prof. De Nicola: senza i quali questa tesi non ci sarebbe proprio.

Bruno: per la Chiara e per avermi prestato il portatile;

Paola: per la Chiara e per tutte le cene che mi ha offerto;

La mia nonna Luisa che se né andata a 101 anni, ma si ricordava sempre che mi dovevo laureare;

Un ringraziamento particolare lo devo fare a l'Elisa, Massimo, Lapo, Luca e la Raffa, Alessandro (e sua moglie, Stefania, per l'incontro sul vino) e naturalmente la Benedetta (la meglio). E senza dubbio Max, il compagno di corso ideale.

Non posso non ringraziare Don Luigi Giussani, Paolo Nanni e Simone Magherini per tante di quelle cose che si dovrebbe riempire un libro solo per elencarle;

I colleghi di Scienze dell'Informazione con cui ho condiviso esami e altre fatiche: Giovanni Ciruolo (cui DEVO regalare una bottiglia di vino buono) il Saba, il Jack, l'Elena Scapecchi, la Francesca Uncini, la Sara Capecchi, Alessandro Falsetti, Leonardo Ricci, i' Conti, Andrea Fornari, la Nicoletta, l'Lollo e la Chiara , l'Ilaria, il Paba, Roberto Biliotti, la Maddalena e la Veronica;

I giovani di Informatica: Gianluca (steven spielberg) Palazzi , la Lilianna(che è gobba, il che dimostra che nessuno è perfetto), I' Baldi, Lapo, Mark e anche Fabio rimasto un informatico nel cuore;

Veniamo ora a quelli di altri corsi e facoltà:

Fisica: Filippo (Il megaPresidente galattico), Silvio(il fisico più normale), Simone(il fisico normalizzato), Alberto(un fisico con superpoteri), Sebastiano (dalla sicilia con furore), Claudio, Marco, lo Spina (per avermi fatto scoprire il magnete), Papadopulos, Gianluca per aver fatto conoscere a tutti il mio soprannome "Lupo";

Matematica: La Francesca Salvi(doveroso primo posto della categoria dopo che mi ha regalato il signore degli anelli su CD), la Caterina, la Nadia, l'Elena Mancini, la Sonia, la Francesca Buttazzo, Luisa Gianuizzi, Marina Mazzanti, l'Elena Volpini(per avermi comprato metà del mio 1° libro di SDC),Silvia Fontanelli;

Biologia : la Silvia Gasp., Francesca Doni, l'Alessia, la Chiara, la Sabina, la Nadia K., la Giulia (e Alessio), l'Caste, la Valentina e la Sara;

Farmacia: Chiara, l'Elena, Leonardo, la Laura e Tommaso;

Lettere:La Tiziana, la Marta, la Gaia, Francesca Destefanis(quantomeno per il caffè al ghiaccio al mare), le sorelle della Guerri Francesca e Silvia, Letizia, Vanja, Cristina, Debora, Gabriele;

Legge: Francesco Minari per "La minaccia dei Cloni";la Silvia (per il tappetino mouse ad arancia che mi ha accompagnato per tutta questa tesi);

Chimica e dintorni: Alessio, Marco Luisini (e la Donatella), Brenna e la Francesca, la Code e la "Sirena", Marco Capecchi(col senno di poi uno dei migliori), Maranghi, la Ciccia, la Chiara D., la Luisa, la Martina, Just, Piero(per il teatro), Pigna, Rudy, Brando, Guido (the King of jokes!), l'Alessandra, Ienco Isaac, Gabriele (spero di riconoscerlo la prox volta che lo vedo) Dacci e l'Ingrid, Ferro e la Ridi, Dama e la Nicoletta, Lorenzo Scali;

Gruppo misto: Pietro (eterno compagno di stanza in montagna, e nonostante la sua fede romanista)

I miei amici dell'Impruneta e correlati, compagni di vacanze, trekking e altre avventure : Enrico , David, Stefano, la Vale, Francesco;

E ora i Soliti Ignoti, ovvero quelli che conosco solo io e che probabilmente non leggeranno mai questi ringraziamenti): Renato ormai vigile (per avermi portato al torneo di scacchi alla giornata della logica) Ravi(analisi 1) e il Villari, il Vassalli (per gli avventuroni) Il Macca (w la Calabria), Pino per l'invenzione del soprannome, il Fornaio delle cascine del riccio (per avermi salvato una notte che rimasi bloccato con la macchina), Schubert e Amicobit, L'Opera Pia Vanni (Ema e la Vale in particolare ma anche il trio Casini, Barbacci, Della Fonte e Suor Giacinta); il Gallo di Sarzana, la Viviana e la Gloria e l'Humanitas di Salerno nel 1996: li ho capito come NON volevo

vivere.

Cose, luoghi, avvenimenti: La mia vecchia bicicletta “Campagnolo” rossa che mi ha accompagnato fedelmente dai 16 ai 26 anni, morta gloriosamente nell’esercizio delle sue funzioni davanti al nuovo polo didattico; Le alpi apuane (dal Pizzo d’uccello al Monte Pisanino alle Panie); il 30 maggio 1998 a Roma (e ringrazio il Papa che ha avuto l’idea);

Spero di non essermi dimenticato di nessuno, ma temo sia impossibile.