



UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE  
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA  
INFORMATICA

# Orchestrazione di servizi compositi in Scala

*Laureando:*  
Nicholas DALL'ARMELLINA

*Relatore:*  
Ch.mo Prof.  
Massimo MARESCA  
*CoRelatore:*  
Ing. Michele STECCA

Anno accademico 2010/2011



*“To those who,  
by the dint of glass and vapour,  
discover stars,  
and sail in the wind’s eye”*

*“A coloro che,  
attraverso lenti a vapore,  
scoprono le stelle,  
e navigano nel filo del vento.”*

LORD GEORGE GORDON NOEL BYRON



# Ringraziamenti

Sono già trascorsi diversi anni da quando ho iniziato la mia avventura universitaria. Ho incontrato molte persone, instaurato nuove amicizie, perdute altre, ma sono poche le persone alle quali sento di dover ringraziare con tutto il cuore per avermi supportato (e sopportato) ed essermi rimaste vicino lungo questo cammino.

In primis non posso che ringraziare i miei genitori che, con la loro pazienza hanno creduto in me e mi hanno sostenuto (e spero continuino con lo stesso amore e pazienza anche in futuro nonostante la mia testardaggine) in tante cose.

In secundis più che un ringraziamento, voglio augurare un sentito “in bocca al lupo” a mio fratello Andrea che, seppur all’inizio non entusiasta, ha intrapreso lo scorso anno la carriera di perito informatico, nella speranza che un giorno possa maturare un vivido interesse per questa disciplina e che possa avere un brillante futuro con essa.

In terzis, voglio ringraziare Mirco ed “*il Case*” Enrico, compagni e amici da ormai  $\left(\frac{27\pi\sqrt{2}}{10}\right) \int_{-\infty}^{\infty} \delta(x) dx$  anni, nonché la mia cara amica Rita a cui rompo le scatole da  $\lim_{x \rightarrow \infty} \log_a x$  anni, seguendo da sempre le mie vicissitudini e ascoltando le mie lamentele.

In quartis, infine, voglio ringraziare il gruppo di ricerca del CIPI, in particolare il mio relatore Prof. Massimo Maresca che mi ha dato l’opportunità di progredire professionalmente permettendomi di svolgere questo lavoro di tesi, e soprattutto, porgo un sentito ringraziamento all’Ing. Michele Stecca che in questi mesi è sempre stato presente durante il mio lavoro con i preziosi consigli che, con pazienza, mi ha rivolto.

Infine, non posso che ringraziare la mia forza d’animo che più volte mi ha permesso di rialzarmi dopo i numerosi momenti di sconforto e insuccessi iniziali. Anche se banale, voglio augurare al me stesso del futuro (sperando che non sia così futuro) una brillante carriera, una buona salute e la speranza di trovare presto un’anima pia che voglia condividere la vecchiaia con lui.

Grazie anche a te che sei arrivando fino in fondo a questo poema senza esserti addormentato o voltato pagina.

Con affetto, sempre vostro  
*Nicholas*

*Padova, lì 25 Ott. 2011*



# Indice

|                                                            |            |
|------------------------------------------------------------|------------|
| <b>Indice</b>                                              | <b>i</b>   |
| <b>Elenco delle tabelle</b>                                | <b>v</b>   |
| <b>Elenco delle figure</b>                                 | <b>vii</b> |
| <br>                                                       |            |
| <b>I Introduzione</b>                                      | <b>1</b>   |
| <b>1 Introduzione</b>                                      | <b>3</b>   |
| <b>2 La composizione dei servizi</b>                       | <b>7</b>   |
| 2.1 Event Driven Mashup . . . . .                          | 7          |
| 2.1.1 Interazione con Servizi Esterni . . . . .            | 9          |
| <b>3 Problematiche dei Servizi Compositi</b>               | <b>11</b>  |
| 3.1 Multi-Threading . . . . .                              | 12         |
| 3.2 Multi-Threading e Web-Server . . . . .                 | 13         |
| <br>                                                       |            |
| <b>II Scala</b>                                            | <b>19</b>  |
| <br>                                                       |            |
| <b>4 Il linguaggio Scala</b>                               | <b>21</b>  |
| 4.1 Caratteristiche e sintassi di Scala . . . . .          | 26         |
| <b>5 Il modello ad Attori</b>                              | <b>33</b>  |
| 5.1 Multi-core e Java . . . . .                            | 33         |
| 5.2 Threadless Concurrency su JVM e Scala Actors . . . . . | 36         |
| 5.2.1 Il modello ad Attori . . . . .                       | 37         |
| 5.2.2 Attori e JVM . . . . .                               | 38         |

|            |                                                              |            |
|------------|--------------------------------------------------------------|------------|
| <b>6</b>   | <b>Il modello ad Eventi</b>                                  | <b>43</b>  |
| 6.1        | React vs. Receive (Event-based vs. Thread-based) . . . . .   | 43         |
| 6.1.1      | React . . . . .                                              | 45         |
| 6.1.2      | Receive . . . . .                                            | 46         |
| 6.2        | Test sulla funzione react . . . . .                          | 47         |
| 6.2.1      | Test1 . . . . .                                              | 48         |
| 6.2.2      | Test2 . . . . .                                              | 49         |
| 6.2.3      | Test3 . . . . .                                              | 51         |
| 6.2.4      | Test4 . . . . .                                              | 52         |
| 6.2.5      | Test5 . . . . .                                              | 53         |
| 6.2.6      | Test6 . . . . .                                              | 54         |
| <b>7</b>   | <b>Scala Scheduler</b>                                       | <b>57</b>  |
| <b>III</b> | <b>Asynchronous I/O</b>                                      | <b>63</b>  |
| <b>8</b>   | <b>Introduzione all'Asynchronous I/O</b>                     | <b>65</b>  |
| 8.1        | Java Non-Blocking I/O, il pacchetto java.nio . . . . .       | 66         |
| 8.2        | Scala Non-Blocking I/O, Scala Continuation . . . . .         | 72         |
| 8.3        | Un approccio ibrido, Java NIO e Scala Delimited Continuation | 77         |
| 8.3.1      | NioSelector . . . . .                                        | 77         |
| 8.3.2      | NioListener . . . . .                                        | 79         |
| 8.3.3      | NioConnection . . . . .                                      | 82         |
| 8.3.4      | NioServer . . . . .                                          | 84         |
| <b>IV</b>  | <b>Netty</b>                                                 | <b>87</b>  |
| <b>9</b>   | <b>L'architettura Netty</b>                                  | <b>89</b>  |
| 9.1        | Single Thread Pattern - Reactor . . . . .                    | 92         |
| 9.2        | Multi-Threaded Pattern - Worker Thread Pool . . . . .        | 95         |
| 9.3        | Multi-Threaded Pattern - Multiple Reactor Thread . . . . .   | 97         |
| 9.4        | Esempi di applicazione in Netty . . . . .                    | 98         |
| <b>V</b>   | <b>Prototipo di Orchestratore in Scala</b>                   | <b>105</b> |
| <b>10</b>  | <b>L'orchestratore di servizi basato su eventi in Scala</b>  | <b>107</b> |
| 10.1       | Mashup basato sugli eventi . . . . .                         | 107        |
| 10.2       | Una piattaforma Mashup basata su Scala . . . . .             | 109        |
| 10.2.1     | Mapping dei componenti MEP negli attori Scala . . . . .      | 109        |

|           |                                                                                        |            |
|-----------|----------------------------------------------------------------------------------------|------------|
| 10.2.2    | L'algoritmo di orchestrazione . . . . .                                                | 113        |
| 10.2.3    | Conclusioni . . . . .                                                                  | 114        |
| <b>11</b> | <b>Prestazioni dell'Orchestratore</b>                                                  | <b>117</b> |
| 11.1      | L'Orchestratore in Scala . . . . .                                                     | 117        |
| 11.1.1    | Funzionamento in locale . . . . .                                                      | 117        |
| 11.1.2    | Orchestratore e Service Proxy locali e Client remoto . .                               | 123        |
| 11.1.3    | Orchestratore, Service Proxy e Client remoti . . . . .                                 | 131        |
| 11.1.4    | Client e Server Service remoti con Orchestratore e Ser-<br>vice Proxy locali . . . . . | 141        |
| 11.2      | L'orchestratore in Java . . . . .                                                      | 149        |
| <b>VI</b> | <b>Conclusioni</b>                                                                     | <b>165</b> |
| <b>12</b> | <b>Conclusioni e Sviluppi futuri</b>                                                   | <b>167</b> |
|           | <b>Bibliografia</b>                                                                    | <b>169</b> |



# Elenco delle tabelle

|       |                                                                           |     |
|-------|---------------------------------------------------------------------------|-----|
| 4.1   | Any, AnyVal, AnyRef, Null, Nothing . . . . .                              | 27  |
| 4.2   | Sintassi Scala vs Java - Parte I . . . . .                                | 28  |
| 4.3   | Sintassi Scala vs Java - Parte II . . . . .                               | 29  |
| 5.1   | Metodi di un attore . . . . .                                             | 39  |
| 11.1  | Risultati relativi alle prove in locale . . . . .                         | 119 |
| 11.2  | Percentuale metodi occupazione CPU . . . . .                              | 119 |
| 11.3  | Risultati relativi alle prove con Orchestratore e Client remoto . . . . . | 127 |
| 11.4  | Percentuale metodi occupazione CPU . . . . .                              | 127 |
| 11.5  | Risultati relativi alle prove con attori remoti . . . . .                 | 134 |
| 11.6  | Percentuale metodi occupazione CPU . . . . .                              | 134 |
| 11.7  | Risultati relativi alle prove con Server Service . . . . .                | 144 |
| 11.8  | Percentuale metodi occupazione CPU . . . . .                              | 144 |
| 11.9  | Risultati relativi alle prove in locale . . . . .                         | 156 |
| 11.10 | Risultati relativi alle prove in remoto . . . . .                         | 160 |



# Elenco delle figure

|      |                                                                                     |     |
|------|-------------------------------------------------------------------------------------|-----|
| 2.1  | Esempio di Event Driven Mashup . . . . .                                            | 8   |
| 2.2  | Architettura d'alto livello per la piattaforma Mashup lato server                   | 9   |
| 3.1  | Esempi dei modelli di Thread . . . . .                                              | 15  |
| 3.2  | Modelli architetturali Server . . . . .                                             | 16  |
| 4.1  | Gerarchia delle classi in Scala . . . . .                                           | 26  |
| 5.1  | Gerarchia delle cache nei sistemi multiprocessore . . . . .                         | 36  |
| 7.1  | Algoritmo di work-stealing . . . . .                                                | 58  |
| 7.2  | Diagramma UML delle Classi . . . . .                                                | 59  |
| 8.1  | Architettura client/server del modello NIO di Java . . . . .                        | 66  |
| 8.2  | diagramma sequenziale UML . . . . .                                                 | 85  |
| 9.1  | Architettura del framework Netty di Java . . . . .                                  | 89  |
| 9.2  | Design classico dei servizi . . . . .                                               | 91  |
| 9.3  | Design con Reactor a Single Thread . . . . .                                        | 92  |
| 9.4  | Design Multi-Thread con Worker Thread . . . . .                                     | 95  |
| 9.5  | Design Multi-Thread con Multiple Reactor Thread . . . . .                           | 97  |
| 10.1 | Esempio di Mashup basato sugli eventi . . . . .                                     | 108 |
| 10.2 | Architettura d'alto livello per la piattaforma Mashup lato server                   | 109 |
| 11.1 | Architettura per l'esecuzione in locale . . . . .                                   | 118 |
| 11.2 | Andamenti temporali per l'esecuzione in locale . . . . .                            | 120 |
| 11.3 | Risorse impiegate per l'esecuzione in locale - Parte I . . . . .                    | 121 |
| 11.4 | Risorse impiegate per l'esecuzione in locale - Parte II . . . . .                   | 122 |
| 11.5 | Architettura per l'esecuzione con Orchestratore e Client remoto                     | 123 |
| 11.6 | Andamenti temporali per l'esecuzione con Orchestratore e Client<br>remoto . . . . . | 128 |

|                                                                                                   |     |
|---------------------------------------------------------------------------------------------------|-----|
| 11.7 Risorse impiegate per l'esecuzione con Orchestratore e Client<br>remoto - Parte I . . . . .  | 129 |
| 11.8 Risorse impiegate per l'esecuzione con Orchestratore e Client<br>remoto - Parte II . . . . . | 130 |
| 11.9 Architettura per l'esecuzione remota . . . . .                                               | 131 |
| 11.10 Andamenti temporali per l'esecuzione remota . . . . .                                       | 135 |
| 11.11 Risorse impiegate per l'esecuzione remota dell'Orchestratore -<br>Parte I . . . . .         | 137 |
| 11.12 Risorse impiegate per l'esecuzione remota dell'Orchestratore -<br>Parte II . . . . .        | 138 |
| 11.13 Risorse impiegate per l'esecuzione remota del Service Proxy -<br>Parte I . . . . .          | 139 |
| 11.14 Risorse impiegate per l'esecuzione remota del Service Proxy -<br>Parte II . . . . .         | 140 |
| 11.15 Architettura per l'esecuzione con Server Service . . . . .                                  | 141 |
| 11.16 Andamenti temporali per l'esecuzione con Server Service remoto                              | 145 |
| 11.17 Risorse impiegate per l'esecuzione con Server Service remoto<br>- Parte I . . . . .         | 146 |
| 11.18 Risorse impiegate per l'esecuzione con Server Service - Parte II                            | 147 |
| 11.19 Architettura Orchestratore in Java . . . . .                                                | 149 |
| 11.20 Andamenti temporali per l'esecuzione in locale . . . . .                                    | 157 |
| 11.21 Risorse impiegate per l'esecuzione in locale - Parte I . . . . .                            | 158 |
| 11.22 Risorse impiegate per l'esecuzione in locale - Parte II . . . . .                           | 159 |
| 11.23 Andamenti temporali per l'esecuzione remota . . . . .                                       | 161 |
| 11.24 Risorse impiegate per l'esecuzione remota - Parte I . . . . .                               | 162 |
| 11.25 Risorse impiegate per l'esecuzione remota - Parte II . . . . .                              | 163 |

# Parte I

## Introduzione



# Capitolo 1

## Introduzione

Negli ultimi anni il mercato dell'IT sta spostando i propri servizi sul web adottando una tecnologia sempre più diffusa chiamata Web Service. In passato i provider hanno sempre fornito i loro servizi attraverso piattaforme “chiuse” rendendo difficile l'integrazione con tecnologie diverse. Negli ultimi anni, questa tendenza sta via via scomparendo lasciando spazio ad architetture open e alla diffusione di API (Application Programming Interface) come quelle catalogate da [programmableweb](http://programmableweb.com)<sup>1</sup> nella quale sono raccolte quelle di piattaforme come Twitter, YouTube, Facebook, Google Maps ecc... La tecnologia che, seguendo questa tendenza, permette di esporre i servizi sono proprio i Web Service.

I Web Service, seppur a livello di singole interazioni, mantengono il vecchio modello client/server; i servizi vengono visti come componenti di base per la creazione di altri servizi più complessi e con questa concezione quindi, spariscono i ruoli di fornitore di un servizio e del suo utilizzatore in quanto anche il client a sua volta può diventare fornitore di servizi per altri. Il concetto importante nell'architettura dei Web Service quindi non è più il ruolo ricoperto da un'applicazione bensì il servizio che essa offre. I motivi che sono alla base della diffusione dei Web Services si possono ricondurre a:

- Permettono l'accessibilità, la disponibilità, la scalabilità e la riusabilità dei diversi servizi;
- Specifiche chiare e semplici;
- Utilizzo di tecnologie standard open già consolidate come XML;
- Una sempre più diffusa adozione da parte delle aziende;

---

<sup>1</sup>[www.programmableweb.com](http://www.programmableweb.com)

- Numerosi tool a disposizione per lo sviluppo.

Esistono anche degli svantaggi legati ai Web Service in quanto, solitamente, viene richiesto un grosso volume di traffico di dati nella rete e un elevato carico computazionale per la de/codifica dei dati.

La recente tendenza in ambito di Web Service vede l'applicazione degli stessi in due ambiti, quello aziendale (con soluzioni business) e il mercato dei Web Service liberamente accedibili attraverso Internet. Per quanto riguarda l'aspetto aziendale, l'architettura che predomina il panorama dei Web Service è SOA (Service Oriented Architecture), che è in grado di unire molteplici servizi capaci di comunicare tra di loro per mezzo di connessioni sfruttando degli standard basati su XML. In ambito open, invece, la community spinge per una diffusione sempre maggiore di API come citato in precedenza.

Le implementazioni dell'architettura SOA attualmente a disposizione si possono suddividere in due categorie, quelle cioè derivate da SOAP (Simple Object Access Protocol) e quelle da REST (REpresentational State Transfer). Esempi di implementazioni dell'architettura SOAP si possono citare WSDL<sup>2</sup> (Web Services Description Language) e SOAP<sup>3</sup> entrambi standardizzati e ben consolidati nell'ambiente al contrario di REST, che rimane uno standard solo de facto nonostante sia più leggero di SOAP e abbia visto negli ultimi anni un notevole incremento dell'adozione rispetto alla controparte SOAP.

I servizi che si ottengono dalla composizione di quelli più elementari si possono suddividere in due categorie:

- **Orchestrazione:** È un metodo centralizzato che descrive l'interazione dei servizi a livello di scambio dei messaggi, includendo la business logic e l'ordine con cui le interazioni devono essere eseguite. Tali interazioni possono includere più applicazioni creando così un modello transazionale di processo. L'Orchestrazione si focalizza quindi sul singolo processo;
- **Coreografia:** A differenza dell'Orchestrazione, la Coreografia è un metodo distribuito che tiene traccia dei messaggi che vengono scambiati tra più parti e più sorgenti. La Coreografia non è associata ad un singolo servizio web eseguito da un singolo attore, bensì allo scambio di messaggi pubblici che intercorre tra diversi servizi web.

Le specifiche di un servizio Web sono espresse in WSDL, che dà disposizioni sulla sintassi da utilizzare nei messaggi che vengono scambiati da un client

---

<sup>2</sup>[www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl)

<sup>3</sup>[www.w3.org/TR/soap](http://www.w3.org/TR/soap)

ma non comprende i messaggi scambiati tra i servizi che, invece, devono essere descritti a parte. WSDL fornisce indicazioni sul tipo di operazioni che possono essere messe a disposizione dal servizio, come utilizzare il servizio (formato dei messaggi, protocolli accettati, formato dei dati restituiti, ecc...) e l'URI del Web Server. Il linguaggio più diffuso basato su XML con cui vengono descritti i servizi implementati sui Web Service si chiama BPEL<sup>4</sup> (Business Process Execution Language). BPEL implementa lo standard BPEL4WS (Business Process Execution Language for Web Service); altri linguaggi utilizzano standard, sempre basati su XML, come WSFL<sup>5</sup> (Web Services Flow Language), BPML (Business Process Modeling Language), WSCI<sup>6</sup> (Web Service Choreography Interface), BPSS<sup>7</sup> (Business Process Specification Schema). Questi linguaggi coprono per lo più lo stesso contesto e le loro maggiori differenze sono insite nella sintassi e nel consorzio che le propone.

I requisiti principali per un sistema basato su Web Service si possono suddividere in 5 categorie:

- **Interoperabilità:** un WS deve consentire lo sviluppo di servizi web interoperabili tra una vasta gamma di ambienti. I WS devono indipendenti dal modello e dal linguaggio di programmazione adottato;
- **Affidabilità:** un WS deve essere affidabile e stabile nel tempo;
- **Sicurezza:** un WS deve provvedere a garantire un ambiente sicuro per l'esecuzione dei processi online;
- **Scalabilità ed Estensibilità:** un WS deve permettere applicazioni scalabili ed estensibili, provvedendo a fornire un architettura modulare, sufficientemente estensibile per evoluzioni future, applicando i principi di semplicità della applicazioni non imponendo alte barriere in entrata;
- **Fault Tolerance:** un WS deve prevedere sistemi di gestione dei guasti permettendo all'applicazione di essere sempre disponibile nel tempo.

Anche rispettando i requisiti di appena citati e sfruttando linguaggi come BPEL non sempre si ottengono i risultati sperati come ben descritto in [10], gli autori dimostrano la possibilità di ottenere prestazioni per nulla soddisfacenti.

In questo scenario, lo studio effettuato e descritto nella tesi, ha portato ad

---

<sup>4</sup>[www.ibm.com/developerworks/library/specification/ws-bpel/](http://www.ibm.com/developerworks/library/specification/ws-bpel/)

<sup>5</sup>[xml.coverpages.org/wsfl.html](http://xml.coverpages.org/wsfl.html)

<sup>6</sup>[www.w3.org/TR/wsci/](http://www.w3.org/TR/wsci/)

<sup>7</sup>[www.ebxml.org/specs](http://www.ebxml.org/specs)

analizzare il problema della composizione di servizi esplorando alcuni linguaggi di programmazione, avendo come obiettivo la minimizzazione del numero di Thread.

### Struttura della Tesi

In questa prima parte della tesi viene presentata una panoramica sulla composizione dei servizi e sullo stato dell'arte, analizzando gli aspetti significativi derivati dalla composizione orientata agli eventi. Saranno caratterizzati gli attori principali e la logica di composizione che permette la comunicazione attraverso scambi di messaggi e loro struttura. Verranno analizzati i punti di forza e le problematiche che conseguono la composizione di servizi partendo da un'analisi del modello di Threading dei Sistemi Operativi moderni e i modelli ai quali è affidata la gestione e la comunicazione, fino ad arrivare ad una critica del modello di Thread che sta alla base del linguaggio di programmazione Java.

Nella seconda parte della Tesi, sarà introdotto un linguaggio di programmazione abbastanza giovane quale *Scala* dove sarà analizzato nella sua struttura e negli aspetti sintattici più significativi per concentrarsi, successivamente, sul modello ad Attori di Scala, alternativa al modello di Threading di Java per la quale è stata ispirata questa Tesi. A questo proposito, ci si concentrerà sul concetto di continuation insito nella primitiva *React* per lo scambio di messaggi, punto cardine che sta alla base di questo studio. Particolare attenzione sarà rivolta anche all'algoritmo di Scheduling utilizzato dal linguaggio perché, come si vedrà, giocherà un ruolo fondamentale nelle prestazioni dell'applicazione che sarà presentata.

Nella terza e quarta parte saranno analizzate, con meno dettaglio della precedente, due architetture alternative al modello ad attori, l'*Asynchronous I/O* con l'intento di definire una base per creare un'applicazione che non effettua chiamate bloccanti, e il framework *Netty* che cerca di definire un pattern architetturale per lo scambio asincrono di messaggi basato sul modello ad eventi.

Nella quinta parte, infine, viene presentato il progetto dell'orchestratore[35] creato in precedenza in Java e relativi test di performance, comparati con il porting in Scala dello stesso, sviluppo e studio del quale sarà punto focale di questo lavoro.

## Capitolo 2

# La composizione dei servizi

### 2.1 Event Driven Mashup

Un Mashup Event Driven, generalmente modellato come un insieme di servizi (Svc), è di solito inserito tramite uno strumento di editing grafico che fa parte di una piattaforma chiamata **S**ervice **C**reation **P**latform. Il creatore di Mashup posiziona i blocchi corrispondenti ai Svc e disegna le corrispondenti dipendenze tra loro. Una volta specificato, un Mashup (o **C**omposite **S**ervice), viene salvato come file XML e memorizzato in un repository. Quando necessario, viene caricato dal repository, installato su una piattaforma denominata **S**ervice **E**xecution **P**latform, ed eseguito. Il deployment di un Mashup richiede il deployment delle sue specifiche, il deployment delle sue Svc e la configurazione del SEP.

Il compositore di servizi esegue una o più azioni e, a seconda delle variazioni dello stato interno, notifica gli eventi agli altri Svcs. Ogni Svc è associato ad un insieme di proprietà di input che vengono elaborate secondo la logica interna del Svc per emettere nuovi eventi e, facoltativamente, per generare nuove proprietà di output. Un evento ha origine in un base service che è collegato tramite un link (vedi Figura 2.1) a un'azione in un altro base service che ha come conseguenza l'innescio dell'azione corrispondente. Le proprietà di uscita del Svc che notifica l'evento, possono essere copiate sulle proprietà di input degli Svc le cui azioni sono invocate. In questo modo, ogni Svc può essere implementato come modulo generico che può essere combinato con tutti gli altri Svc per creare nuovi Mashup. Per esempio in Figura 2.1 il Svc *GMailMonitor* è associato alla proprietà in uscita “target mail content” e all'evento “when-target-mail-received”, mentre il Svc *SendSMS* è associato ad una proprietà di input denominata “sms content” per l'azione “SendSMS” e all'evento “when-sms-sent”. Grazie al fatto che gli Svc possono scambiarsi i



Figura 2.1: Esempio di Event Driven Mashup

valori delle proprietà, al momento della notifica degli eventi è possibile avere la proprietà di input “sms content” del set *SendSMS* al valore di uscita “target email content” del Svc *GMailMonitor*.

Possono essere inclusi nelle specifiche di Mashup due blocchi di controllo: il blocco *If-Then-Else* e il blocco *Merge*. Il primo permette di indicare due percorsi mutuamente esclusivi nel servizio composito, che saranno presi a seconda del valore di una o più proprietà specificate come parametri di input, mentre il secondo si occupa della fusione di due percorsi indipendenti in esecuzione in parallelo.

Un Mashup Server Side deve poter essere eseguito da un gran numero di utenti in più istanze, dette *Sessions*. Alcune delle proprietà di input del Mashup non possono essere definite al momento della creazione perché in sessioni differenti sono diverse. La differenza più importante rispetto al Mashup call/response, in cui si invoca un'azione che può portare ad una sola risposta, è che gli Svc sono autorizzati ad emettere uno o più eventi, innescando così l'invocazione di uno o più Svc.

Una volta che la creazione di un Mashup è conclusa, la sua descrizione viene salvata in formato XML e memorizzata in una repository da cui può essere recuperata per poi essere distribuita.

I ruoli degli attori coinvolti nella creazione di un Mashup sono:

- **Mashup creator:** è l'attore che crea nuovi Mashup mediante il tool Service Creation Platform (SCP);
- **Mashup utente:** è l'attore che vuole usare un Mashup;
- **Svc creator:** è l'attore che implementa un Svc secondo l'interfaccia definita dal Platform Administrator;

- **Platform Amministratore:** è l'attore che amministra la piattaforma dal lato server, (vale a dire problemi hardware, di rete, l'implementazione di nuovi Svc/SC, problemi di prestazioni, ecc...)

### 2.1.1 Interazione con Servizi Esterni

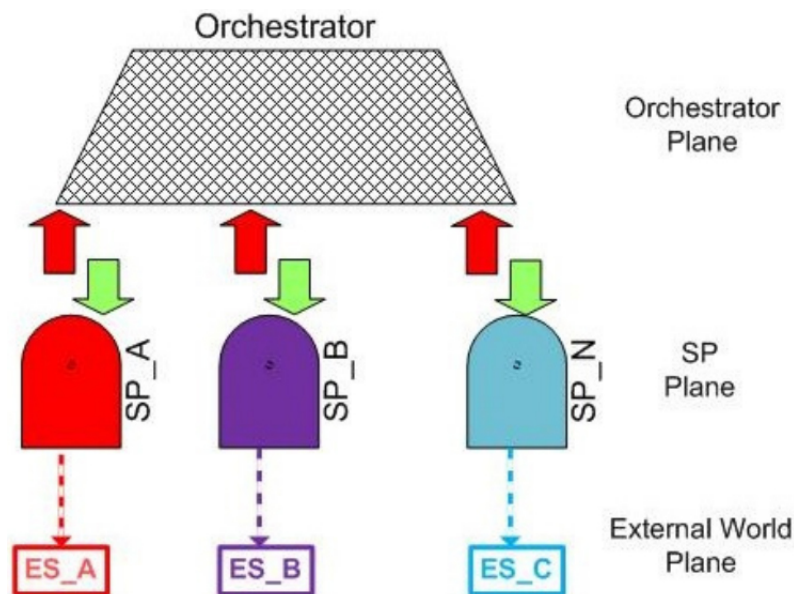


Figura 2.2: Architettura d'alto livello per la piattaforma Mashup lato server

L'architettura della piattaforma lato server che esegue il Mashup è mostrata in Figura 2.2, dove i rettangoli colorati rappresentano le risorse (funzionalità/contenuti) disponibili su Internet (Web Service API, server di posta, mappe, feed RSS, ecc...) che possono essere combinate per creare un Mashup. Queste risorse non possono essere direttamente collegate al modello event-driven introdotto nella Sezione 2.1, perché non espone un'interfaccia compatibile per come sono stati progettati i servizi stand alone. Inoltre, le risorse esterne possono essere realizzate utilizzando diverse tecnologie e con diversi formati di dati e non possono essere direttamente combinate in un Mashup. Per questo è stato introdotto il concetto di Service Proxy (SP). Un componente SP viene eseguito all'interno del Domain Service Provider e incorpora la risorsa esterna per renderla compatibile con il modello event-driven definito nella sezione precedente. È necessario un Sp diverso per ogni servizio esterno. In particolare ogni SP:

- Comunica con l'orchestratore utilizzando un'interfaccia standard;

- Comunica con una risorsa esterna utilizzando un protocollo di comunicazione proprietario. In particolare, esso gestisce gli Eventi Esteri che arrivano da Risorse Esterne estraendone i dati di interesse e convertendoli in “Eventi d’Orchestrazione”.

La componente SP rende l’architettura estensibile, così come un nuovo servizio può essere facilmente aggiunto per aumentare il numero dei blocchi di base che possono essere utilizzati per creare un Mashup. Quando nuove funzionalità (ad esempio, un nuovo servizio sul Web) o nuove fonti di dati (ad esempio, i nuovi dati memorizzati in un database) saranno disponibili, possono essere rapidamente aggiunte alla piattaforma per sostenere la creazione di Mashup che si appoggia ad essi.

La componente “Orchestrator” si occupa di eseguire la logica di Mashup, combinando gli Svc come definito nel SCP. Questo sistema centralizzato non comunica direttamente con i servizi esterni mentre riceve/invia eventi da/per SP che si trovano all’interno del Domain Service Provider, che dovrebbero gestire le comunicazioni con i servizi situati esternamente. Gli SP eseguono anche la traduzione del formato dei dati, cioè, traducono le proprietà di input nel formato dati richiesto dal servizio esterno specifico e analizzano la risposta del Svc per completare le proprietà di output dello stesso.

In sintesi, sono stati definiti i seguenti concetti:

- Servizio (Svc): la funzionalità di base che può essere combinata con altri servizi per creare un Mashup (ad esempio il Svc “SendSMS”);
- Mashup (o servizio composito): un insieme di Svc che compongono un nuovo servizio complesso seguendo il paradigma definito nella sezione 2.1;
- Servizio Proxy (SP): il software che si comporta come un proxy che permette al server di comunicare con le risorse esterne;
- Orchestrator: il software che si occupa di eseguire il Mashup.

## Capitolo 3

# Problematiche dei Servizi Compositi

Per tenere separate le diverse applicazioni eseguite in una macchina, i sistemi operativi fanno uso di oggetti chiamati processi. Collegati ai processi ci sono delle entità chiamate Thread che eseguono il codice contenuto in un processo ai quali, il sistema operativo, assegna degli slot temporali del processore. Ciascun Thread coordina i gestori delle eccezioni, una coda prioritaria sullo scheduling e un insieme di strutture dati utilizzate per salvare il contesto del Thread fino alla sua esecuzione. Il contesto di un Thread rappresenta l'insieme di informazioni necessarie alla ripresa dell'esecuzione come lo stato dei registri della CPU e dello stack.

I normali sistemi operativi multitasking simulano l'esecuzione contemporanea di più Thread di processi diversi; questa operazione viene eseguita dividendo il tempo del processore tra i Thread che ne richiedono l'utilizzo. Il Thread ancora in esecuzione allo scadere della porzione di tempo concessagli, viene sospeso e il processore viene assegnato ad un altro Thread. Quando il sistema cambia Thread, viene salvato il contesto del Thread interrotto caricando quindi quello del Thread successivo dall'apposita coda.

La quantità di tempo assegnata ad un Thread dipende dal sistema operativo e dal processore in uso. Grazie al fatto che ogni porzione di tempo è limitata, più Thread simulano un'esecuzione contemporanea anche in architetture con un solo processore. Questa situazione si verifica anche in ambienti multi processore in quanto i Thread da eseguire sono distribuiti tra i processori disponibili.

### 3.1 Multi-Threading

**Vantaggi** L'utilizzo di più Thread rappresenta la tecnica migliore di gestione dei processi in quanto incrementa la velocità di risposta di un sistema e permette l'elaborazione parallela dei dati. In ambienti mono processore, queste capacità si ottengono dall'utilizzo di più Thread che elaborano i dati in background, sfruttando i brevi periodi di tempo concessi tra un evento utente e il successivo.

**Svantaggi** Utilizzando il minor numero di Thread possibile si riduce al minimo l'impiego di risorse del sistema e se ne migliorano le prestazioni. Nella fase di progettazione di un'applicazione, bisogna considerare le problematiche che ne conseguono dall'utilizzo di Thread in quanto presentano dei requisiti per quanto riguarda l'utilizzo delle risorse e si possono generare potenziali conflitti. I requisiti da considerare sono i seguenti:

- Il sistema occupa spazio in memoria per la conservazione delle informazioni riguardanti il contesto dei processi e dei Thread. Da ciò ne consegue che in sistemi a 32 bit, il numero di processi e di Thread che è possibile creare è limitato dalla memoria disponibile in quanto, lo spazio degli indirizzi è di  $2^{32} = 4GB$  (il problema non sussiste nei sistemi a 64 bit);
- La gestione di un elevato numero di Thread comporta un elevato consumo del tempo del processore. Se la maggior parte dei Thread in esecuzione sono concentrati in un unico processo, quelli contenuti negli altri saranno pianificati meno frequentemente;
- Gestire l'esecuzione del codice attraverso l'impiego di molti Thread e con tecniche di accesso condiviso, sono operazioni complesse che possono causare numerosi errori. È necessario quindi una corretta sincronizzazione o un meccanismo di controllo dell'accesso alle risorse condivise. I problemi che conseguono da una non corretta sincronizzazione dell'accesso alle risorse possono portare a situazioni di *deadlock* (è una situazione anomala in cui più Thread non concludono la loro esecuzione in quanto tutti attendono che l'altro/i completino la loro esecuzione) e *race condition* (si verifica a causa di una dipendenza critica imprevista legata alla durata di due eventi).

La riduzione del numero di Thread garantisce un miglior controllo della sincronizzazione delle risorse, in quanto potrebbero presentarsi pattern per i quali un minore numero di Thread è gestibile in modo più efficace.

## 3.2 Multi-Threading e Web-Server

Nella progettazione di un Server si devono tener conto dei requisiti che esso, in generale, deve avere. Un Server deve poter gestire in modo rapido un numero elevato di Client rispondendo rapidamente ad ogni richiesta, anche nella situazione in cui cresca rapidamente il numero di Client. In presenza di un elevato numero di Client che non stanno effettuando nessuna richiesta e che per i quali si archivia lo stato, tali Client devono avere un effetto trascurabile sul tempo di risposta. Per i Client che, al contrario, stanno eseguendo richieste che potrebbero richiedere del tempo per esser eseguite, esse non devono aumentare in modo significativo i tempi di risposta delle altre richieste.

Abbiamo dunque i seguenti requisiti:

- Minimizzazione della dimensione dello stato salvato per ogni Client attivo;
- Il tempo per notificare una richiesta da parte di un Client non deve dipendere dal numero di Client attivi che non fanno richieste;
- Quando un Client fa una richiesta ad un Server, tale richiesta non deve essere influenzata dalla presenza di altri Client che concorrono per l'utilizzo del Server.

Come suggerito in<sup>1</sup>, ci sono tre modelli dominanti di mapping per i Thread, ognuno dei quali ha i propri trade-off (Figura 3.1):

- Molti Thread in un Light-Weight Process (LWP) - approccio multi-a-uno;
  - **Vantaggi:**
    - \* Portabilità;
    - \* Facile da implementare con poche dipendenze di sistema.
  - **Svantaggi:**
    - \* Si perdono i vantaggi del parallelismo;
    - \* Necessario meccanismo di lock per il Synchronous I/O.
- Un Thread per LWP - approccio uno-a-uno;
  - **Vantaggi:**

---

<sup>1</sup><http://www.cs.indiana.edu/classes/b534-plal/ClassNotes/sched-synch-details4.pdf>

- \* Sfruttamento del parallelismo, blocking e system call.
- **Svantaggi:**
  - \* La creazione di Thread coinvolge la creazione di LWP;
  - \* Ogni Thread utilizza risorse prelevate dal kernel;
  - \* Numero limitato di Thread totali.
- Molti Thread in molti LWP - approccio multi-a-molti.
- Due tipi di Thread:
  - \* *Bounded*;
    - I bounded Thread sono mappati ciascuno su un singolo LWP. Gli user Thread possono esser mappati solo su un singolo kernel Thread mentre questi ultimi possono servire più user Thread.
  - \* *Unbounded*.
    - Gli unbounded Thread possono esser mappati sullo stesso LWP

Un server HTTP riceve le richieste dai propri Client tramite connessioni TCP. Il server rimane in ascolto su una porta nota per le richieste di nuova connessione. Quando una nuova richiesta di connessione arriva, il sistema offre la connessione al server tramite una chiamata a sistema *accept*. Il server attende che il Client gli invii una richiesta su questa connessione, analizza la richiesta e poi restituisce la risposta sulla stessa connessione. Il Web Server di solito ottenere la risposta dal file system locale, mentre i Proxy ottengono le risposte da altri Server, tuttavia, entrambi i tipi di server possono utilizzare una cache per velocizzarne il recupero.

I modelli tipici di architettura di un Server si possono riassumere essenzialmente in tre categorie:

- **process-per-connection**
  - Il Server inizializza un nuovo processo per gestire ogni connessione. L'overhead da fork (l'overhead per la creazione del processo) è un problema in quanto le architetture server utilizzano una serie di processi pre-inizializzati. In questo modello, mostrato nella Figura 3.2(a) , un processo master accetta nuove connessioni e passandole ad un processo di pre-inizializzazione.
- **thread-per-connection**

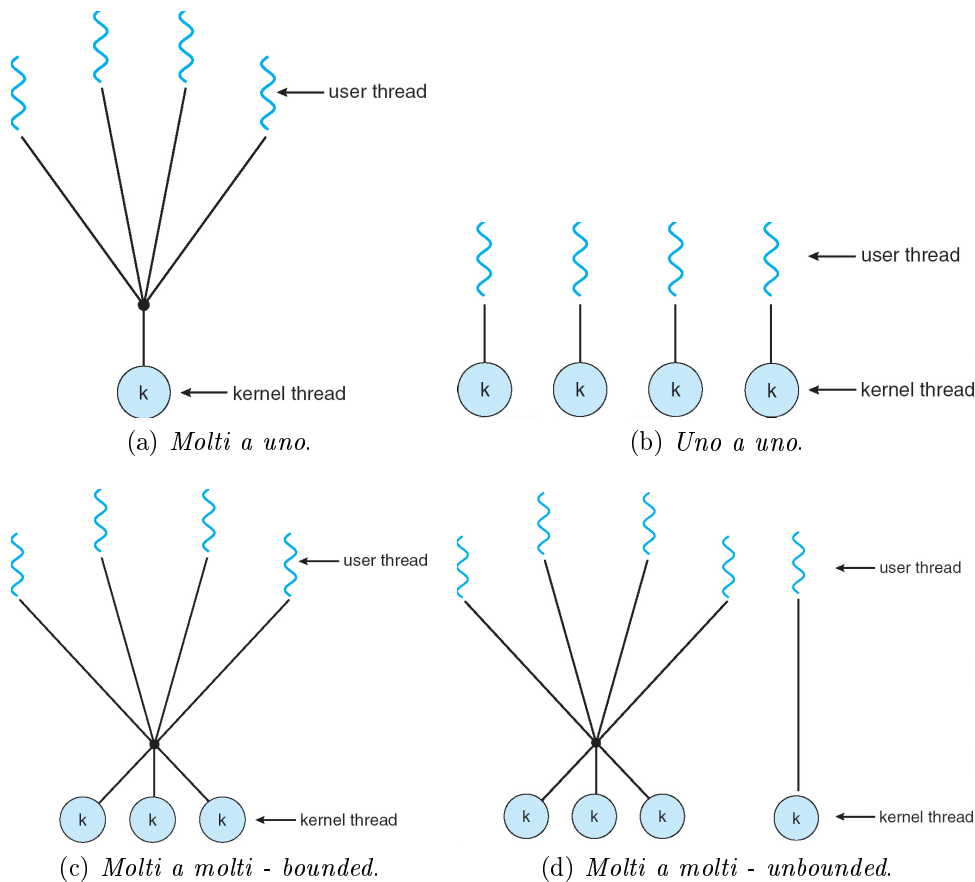


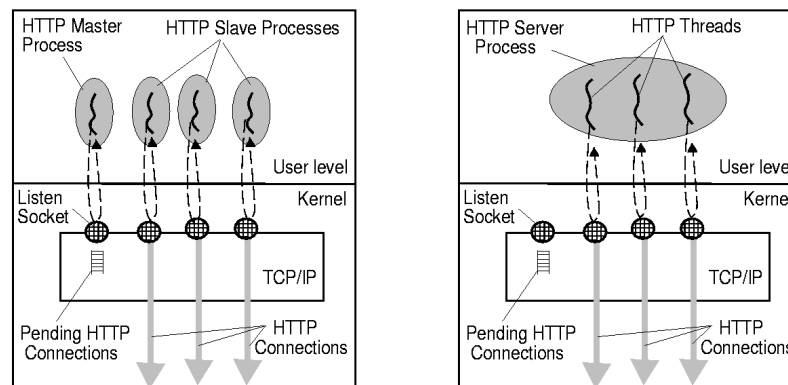
Figura 3.1: Esempi dei modelli di Thread

- Ogni connessione viene assegnata ad un unico Thread (Figura 3.2(b)). Questi possono essere kernel o user-level Thread. Il Thread Scheduler gestisce la condivisione dell'utilizzo della CPU tra i Thread del server. I Thread inattivi rimangono in ascolto sul proprio socket ed accettano nuove connessioni in ingresso sullo stesso.

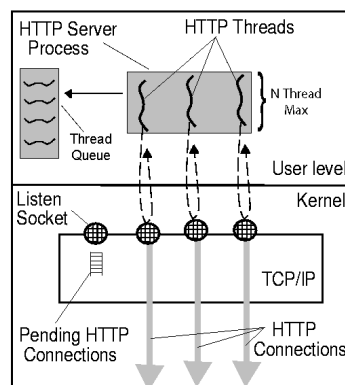
#### • thread pool

- È una variante del thread-per-connection. Il vantaggio principale è la sua semplicità. La generazione dinamica di un Thread per la gestione di ogni nuova richiesta, causa un elevato impiego delle risorse se il numero di richieste diventa molto grande. Il modello di Figura 3.2(c), per evitare tale overhead, all'avvio del Server inizializza un numero prefissato di Thread che possa soddisfare tutte le richieste in arrivo. Questa soluzione permette di contenere i costi dovuti alla creazione dei Thread e aggira i limiti dovuti

all'impiego delle risorse del sistema operativo. Le richieste dei client sono eseguite parallelamente fino a quando il numero di richieste simultanee non superi il numero di Thread presenti nel pool. A questo punto, ulteriori richieste devono essere accodate (o rifiutate) fino a quando un Thread non diventa disponibile.



(a) *Modello process-per-connection.* (b) *Modello thread-per-connection.*



(c) *Modello thread pool.*

Figura 3.2: Modelli architetturali Server

Prendiamo in considerazione un Web Server che fa riferimento al modello thread-per-connection visto in precedenza. Se per ogni connessione viene creato un Thread, nel sistema sono presenti  $\#ConessioniAttive = \#ThreadNelSistema$ .

Un Server può memorizzare il suo stato per-Client in un Thread per ogni Client attivo e bloccarsi in attesa di una richiesta su ogni Thread. Lo Scheduler dei Thread si occuperà quindi di accorgersi quando l'input è disponibile da una di quelle richieste bloccate in lettura e continuare l'esecuzione sul Thread associato. Tuttavia, i Thread sono relativamente costosi e consumano più memoria del necessario.

Una realizzazione che conserva lo stato di ogni Client manualmente potrebbe memorizzare quello stato in modo molto più compatto e quindi gestire molti più Client, ma oltre ad essere più complicato da attuare, l'applicazione deve ora includere un meccanismo per il multiplex di più Client su un minor numero di Thread e quindi ci sarebbero Client attivi senza che ci sia alcun Client bloccato da altri Client.

In questo capitolo sono state presentate varie problematiche concernenti l'utilizzo di Thread, per cui, nei capitoli successivi, verranno presentate delle metodologie alternative come il linguaggio di programmazione Scala (nel capitolo 4), analizzando il problema di tenere bassi il numero di Thread nel sistema in un'applicazione Client/Server anche con un ottica di integrazione in altre architetture come Netty (nel capitolo 9) e al sistema di Asynchronous I/O (nel capitolo 8).



# Parte II

## Scala



# Capitolo 4

## Il linguaggio Scala

Scala è un linguaggio che affronta le principali esigenze degli sviluppatori moderni. Si tratta di un linguaggio *statically typed* con *mixed-paradigm*, un linguaggio compilato per JVM, elegante e con una sintassi flessibile, un sistema di tipo sofisticato con costrutti sintattici che promuovono la scalabilità da un piccolo script interpretato a grandi applicazioni sofisticate. Più in dettaglio:

**Statically typed** Un linguaggio *statically typed* vincola il tipo ad una variabile per tutta la durata della stessa. Al contrario, un linguaggio *dynamically typed* vincola il tipo al *valore* effettivo a cui fa riferimento la variabile, il che significa che il tipo di una variabile può cambiare insieme al valore a cui riferisce.

**Mixed paradigm - object-oriented programming** Scala supporta pienamente la programmazione orientata agli oggetti (OOP). Scala migliora il supporto Java per OOP con l'aggiunta di *trait* (discusso più avanti). In Scala, tutto è un oggetto. In Scala non sono presenti i tipi primitivi come in Java. Invece, tutti i tipi numerici sono veri e propri oggetti. Tuttavia, per ottenere prestazioni ottimali, Scala utilizza i tipi primitivi di base del runtime quando possibile. Inoltre, Scala non supporta il costrutto *static* a livello di classe o membri di tipi, dal momento che non sono associati a un'istanza effettiva. Invece, Scala supporta un oggetto Singleton per sostenere quei casi in cui è necessario avere esattamente un'istanza di un tipo.

**Mixed paradigm - functional programming** Scala supporta pienamente la programmazione funzionale (FP). FP è un paradigma di programmazione precedente al OOP, ma è stato ripreso da poco tempo. L'interesse

per la FP è in aumento a causa dei costrutti che agevolano la risoluzione di problemi di progettazione, specialmente in ambito concorrente. I linguaggi puramente funzionali non consentono alcun stato mutabile, evitando così la necessità di sincronizzazione nell'accesso condiviso allo stato mutevole. I programmi scritti in linguaggi funzionali puri comunicano attraverso uno scambio di messaggi tra processi autonomi e concorrenti. Scala offre anche le *closure*, una caratteristica che i linguaggi dinamici come Python e Ruby hanno adottato dalla programmazione funzionale e assente nelle ultime versioni di Java<sup>1</sup>. Le closure sono funzioni le cui variabili referenziate racchiudono la definizione della funzione. Cioè, le variabili non vengono passate come argomenti o definite come variabili locali all'interno della funzione. Le closure sono una potente astrazione con la quale vengono implementati sistemi ad oggetti e le strutture di controllo fondamentali.

**Linguaggio compilato per JVM** Scala è noto soprattutto per essere un linguaggio compilato per JVM, quindi Scala genera bytecode riconosciuti da una JVM. Il compilatore Scala utilizza tecniche intelligenti per mappare le estensioni Scala in bytecode validi. Da Scala, si può facilmente richiamare il bytecode che ha avuto origine da un sorgente Java (per la JVM). Al contrario, è possibile richiamare codice Scala da Java, per mezzo di artifici di programmazione, eseguendolo sulla JVM consentendo allo sviluppatore Scala di sfruttare le librerie disponibili e di interoperare con altri linguaggi ospitati su quelle runtime.

**Sintassi concisa, elegante e flessibile** La sintassi di Java può essere prolissa. Scala utilizza una serie di tecniche per ridurre al minimo la sintassi non necessaria rendendo il codice molto più conciso come il codice dei linguaggi dinamici (esempio nel codice 4.1 e 4.2). L'inferenza di tipo riduce al minimo la necessità di informazioni di tipo esplicito in molti contesti. Le dichiarazioni di tipi e le funzioni sono molto concise.

Listing 4.1: Esempio codice Java

```
1 public class BasicBlockEdge {  
2     public BasicBlockEdge(CFG cfg, int fromName, int toName) {  
3         from = cfg.createNode(fromName);  
4         to = cfg.createNode(toName);  
5         from.addOutEdge(to);  
6         to.addInEdge(from); cfg.addEdge(this); }  
}
```

<sup>1</sup>Sono state introdotte per la prima volta nella versione Java SE 7 del 28 luglio 2011 [wiki.java.net/bin/view/JDK/ClosuresSyntaxInJava7](http://wiki.java.net/bin/view/JDK/ClosuresSyntaxInJava7)

```

7 public BasicBlock getSrc() { return from; }
8 public BasicBlock getDst() { return to; }
9 private BasicBlock from, to; }

```

Listing 4.2: Esempio codice Scala

```

1 class BasicBlockEdge(cfg : CFG, fromName : Int, toName : Int) {
2   var from : BasicBlock = cfg.createNode(fromName)
3   var to : BasicBlock = cfg.createNode(toName)
4   from.addOutEdge(to)
5   to.addInEdge(from)
6   cfg.addEdge(this) }

```

La sostanziale differenza nella versione Scala dello stesso codice Java sta nel fatto che la dichiarazione della classe consente il passaggio di parametri che diventano istanze di variabili per ogni oggetto della classe. Il costruttore può essere inserito direttamente nella definizione della classe. Le variabili `from` e `to` diventano così accessibili ai membri della classe attraverso una funzione `get` generata automaticamente (diversamente da Java che deve essere dichiarata esplicitamente). Poiché le funzioni senza parametri non richiedono l'uso delle parentesi, le funzioni di accesso come `get` e `set` possono sempre essere riscritte in un secondo momento. Scala produce automaticamente anche la funzione di `set` posponendo il carattere speciale underscore al nome della variabile (esempio `from_(int)` e `to_(int)`) senza dover dichiarare esplicitamente una funzione `set` come in Java. L'ultimo valore calcolato è il valore restituito da una funzione in modo da risparmiare sulla dichiarazione del valore restituito.

**Scalable - architectures** Scala è progettato per essere scalabile da piccoli e interpretati script a grandi applicazioni distribuite. Scala fornisce quattro meccanismi del linguaggio che promuovono la composizione scalabile dei sistemi:

- self type espliciti;

Listing 4.3: Esempio senza l'utilizzo delle self reference che genera errore in compilazione

```

class NodeImpl {
  def connectWith(node: Node): Edge = {
    val edge = newEdge(this, node) // genera errore perché
    edges = edge :: edges          // this è di tipo
    edge }}                        // NodeImpl e non Node

```

Listing 4.4: Esempio con l'utilizzo delle self reference che compila correttamente

```
class NodeImpl {  
  self: Node =>  
  def connectWith(node: Node): Edge = {  
    val edge = newEdge(this, node) // ora funziona  
    edges = edge :: edges  
    edge }}
```

- Nel primo caso viene generato un'errore in quanto `this` fa riferimento ad un'istanza della classe `NodeImpl` mentre lo si utilizza in un contesto che ne richiede una di tipo `Node`. In Scala è possibile legare una classe ad un altro tipo (che può anche essere implementata in un secondo momento) passando il riferimento `self` di `this` all'altro tipo.
- tipi di dato astratto e generici;
  - per un esempio di utilizzo di dato astratto si veda il Listing 4.7. Per l'uso dei generici si veda la differenza tra Java e Scala nei Listing riportati di seguito:

Listing 4.5: Esempio di utilizzo dei generici in Java

```
public class Stack<T> {  
  private List<T> stack = new ArrayList<T>();  
  public void push(T t) { ... }  
  public T pop() { ... }  
}
```

Listing 4.6: Esempio di utilizzo dei generici in Scala

```
class Stack[T] {  
  private val stack:List[T] = new List[T]  
  def push(t:T):Unit = { ... }  
  def pop:T { ... }  
}
```

- classi nidificate
  - per un esempio di utilizzo di classe nidificata si veda il Listing 4.7
- mixin composition.
  - Scala permette di riutilizzare le definizioni di un nuovo membro di una classe nella definizione di una nuova classe. Ciò è espresso come mixin-class composition.

Listing 4.7: Esempio di mixin composition

```
abstract class AbsIterator {...}

trait RichIterator extends AbsIterator {...}

class StringIterator(s: String) extends AbsIterator {...}

object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator with RichIterator
      {...}
  }
}
```

Come si può vedere dal codice, con una mixin composition si può mettere nello stesso codice una classe astratta, un trait che estende la classe astratta, una classe, un oggetto, classi che estendono altre e che implementano interfacce, ecc...

**Scalable - performance** Poiché il codice Scala viene eseguito sulla JVM, beneficia di tutte le ottimizzazioni e delle prestazioni fornite da tale linguaggio e di tutti gli strumenti di terze parti che supportano prestazioni e scalabilità, come profiler, librerie cache distribuite, meccanismi di clustering, ecc...

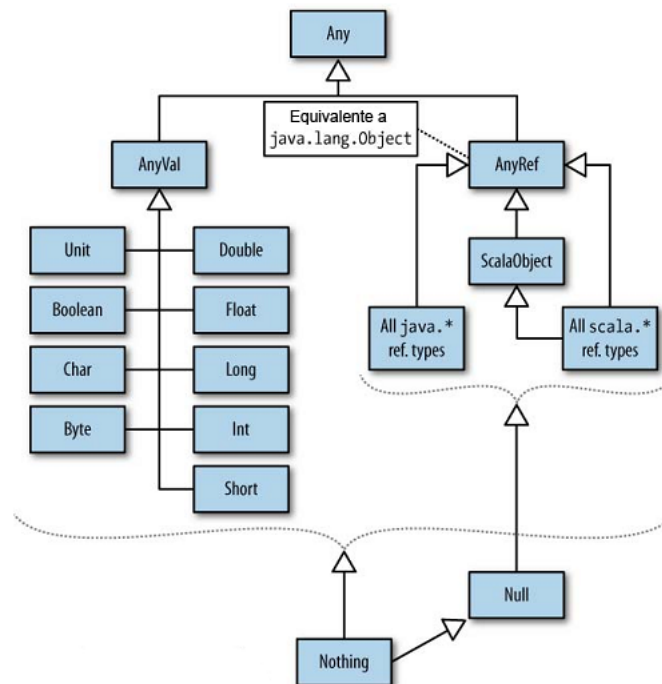


Figura 4.1: Gerarchia delle classi in Scala

## 4.1 Caratteristiche e sintassi di Scala

Nella Tabella 4.2 vengono riassunte le principali caratteristiche sintattiche del linguaggio e confrontate con la controparte in Java.

Scala permette di decidere, nella dichiarazione, quando una variabile può essere immutabile (2) (in sola lettura) o mutabile (1)(lettura-scrittura). Una variabile immutabile è dichiarata con la parola chiave **val**. Una variabile immutabile deve essere inizializzata quando viene dichiarata.

Esempio: **val** array: Array[String] = **new** Array(5)

Una variabile mutabile viene dichiarata con la parola chiave **var**. Anche per questa, Scala richiede che sia inizializzata quando viene dichiarata. Si può assegnarne un nuovo valore ogni volta che si vuole ma non è possibile modificarne il tipo (vedi esempio seguente) perché, ad esempio, Double è immutabile.

Esempio: **var** price: Double = 100.

Tabella 4.1: Any, AnyVal, AnyRef, Null, Nothing

| Nome    | Genitore                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Any     | nessuno                      | È la radice della gerarchia. Definisce alcuni metodi <i>final</i> come “==”, “!=”, “asInstanceOf[T]”, “asInstanceOf[T]”, così come le versioni predefinite di <i>equals</i> , <i>hashCode</i> e <i>toString</i> che sono progettate per essere ridefinite dalle sottoclassi.                                                                                                                                                                                                                                                                                                                                                                                       |
| AnyVal  | Any                          | Il genitore di tutti i tipi primitivi (Boolean, Byte, Char, Short, Int, Long, Float, Double, Unit). Tutte le istanze di AnyVal sono di valore immutabile, e tutti i tipi di AnyVal sono <i>abstract final</i> . Quindi, nessuno di essi può essere istanziato con <i>new</i> . Piuttosto, vengono create nuove istanze con valori letterali o chiamando i metodi sulle istanze che restituiscono il nuovo valore.                                                                                                                                                                                                                                                  |
| AnyRef  | Any                          | Il genitore di tutti i tipi reference, inclusi tutti i tipi <i>java.*</i> e <i>scala.*</i> . È equivalente a <i>java.lang.Object</i> per la JVM e a <i>object</i> ( <i>System.Object</i> ) per la runtime .NET. Le istanze dei tipi reference vengono create con <i>new</i> .                                                                                                                                                                                                                                                                                                                                                                                      |
| Null    | tutti i tipi di Anyref       | È definito come <i>final trait</i> (quindi non può avere sottotipi) ed ha un'unica istanza, <i>null</i> . Dal momento che Null è un sottotipo di tutti i tipi AnyRef, è sempre possibile assegnare null come istanza di una di questi tipi. Java, al contrario, tratta semplicemente null come parola chiave con un trattamento speciale da parte del compilatore. Tuttavia null, in Java, si comporta effettivamente come se si trattasse di un sottotipo di tutti i tipi, proprio come Null in Scala. D'altra parte, dal momento che Null non è un sottotipo di AnyVal, non è possibile assegnare null ad un Int, che è anche coerente con la semantica di Java. |
| Nothing | qualsiasi classe, anche Null | Anche Nothing è <i>final trait</i> , ma non ha istanze. Tuttavia, è ancora utile per definire i tipi. L'esempio migliore è Nil, la lista vuota, che è un <i>case object</i> . È di tipo <i>List[Nothing]</i> . Nil è un'istanza di <i>List[T]</i> , per ogni tipo T.                                                                                                                                                                                                                                                                                                                                                                                               |

Tabella 4.2: Sintassi Scala vs Java - Parte I

| Scala                                                                                                        | Java                                                                   |
|--------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| Definizione esplicita di classi e oggetti<br>(object definisce una classe con una singola istanza)           |                                                                        |
| <i>id</i> : <i>type</i> per definizioni e parametri                                                          | <i>type id</i>                                                         |
| Tutte le definizioni iniziano con <i>def</i>                                                                 |                                                                        |
| Non è richiesto il ; alla fine di ogni riga                                                                  |                                                                        |
| Inserimento di “=” dopo firma metodo e prima di {}                                                           |                                                                        |
| <i>def</i> metodo( <i>variabile</i> : <i>Tipo</i> ) : <i>tipoRestituito</i> = {}                             | <i>public tipoRestituito</i> metodo( <i>Tipo</i> <i>variabile</i> ) {} |
| <i>Array</i> [ <i>T</i> ] con <i>Array</i> keyword e <i>T</i> tipo di dato generico                          | <i>tipoDato</i> []                                                     |
| Accesso ad array con <i>a</i> ( <i>i</i> )                                                                   | <i>a</i> [ <i>i</i> ]                                                  |
| <i>Unit</i>                                                                                                  | <i>void</i>                                                            |
| Wildcard “_” perché “*” è una keyword<br>nella programmazione funzionale                                     | *                                                                      |
| (1) <i>var</i> <i>variabile</i> : <i>tipoDato</i> = inizializzazione                                         | <i>tipoDato</i> <i>variabile</i> = inizializzazione                    |
| (2) <i>val</i> <i>variabile</i> = inizializzazione<br>(variabile immutabile in sola lettura)                 | <i>final tipoDato</i> <i>variabile</i> = inizializzazione              |
| (3) Superclasse “scala.Any”                                                                                  |                                                                        |
| (4) Derivate di Any:<br>▽ AnyVal (Unit, Double, Int, ecc...)<br>▽ AnyRef (ScalaObject, String, List, ecc...) | Superclasse “java.lang.Object” corrisponde a AnyRef                    |
| (5) Nothing → Null → classi derivate da AnyRef<br>↘ classi derivate da AnyVal                                |                                                                        |
| <i>Nil</i>                                                                                                   | <i>null</i>                                                            |
| “==”                                                                                                         | “==” sui tipi                                                          |
| “==”                                                                                                         | <i>equals</i> sugli oggetti                                            |
| Ogni operazione viene vista come un invio di un messaggio<br>es: <i>x + y</i> corrisponde a <i>x. + (y)</i>  |                                                                        |

Quando un’istanza di una classe è seguita da parentesi con un elenco di zero o più parametri, il compilatore invoca il metodo `apply` su tale istanza. Questo è vero per un oggetto con definito un metodo `apply`, così come un’istanza di una classe che definisce un metodo `apply`. Nel caso di un *object*, `apply` si comporta come factory method restituendo una nuova istanza.

Esempio:

```

type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}

```

Tabella 4.3: Sintassi Scala vs Java - Parte II

|                                                                                                                                                                    |                                                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|
| supertipi +A                                                                                                                                                       |                                                 |
| “.:” come estrattore di head/tail da liste                                                                                                                         |                                                 |
| (6) “←” compressore<br>es: <i>var</i> v ← List(...)                                                                                                                |                                                 |
| (7) “→” separatore<br>es: <i>val</i> mappa = Map("Stat o" → "Capitale")                                                                                            |                                                 |
| “=>” usato nelle funzioni letterali<br>es: <i>val</i> mult = (i : Int) => i * 2                                                                                    |                                                 |
| <i>for</i> , <i>forall</i> , <i>foreach</i>                                                                                                                        | <i>for</i>                                      |
| (8) <i>apply</i> [T](i : T)                                                                                                                                        |                                                 |
| (9) <i>unapply</i> [T](i : Tuple2)                                                                                                                                 |                                                 |
| (10) trait                                                                                                                                                         |                                                 |
| yeld                                                                                                                                                               |                                                 |
| Chiamata per nome:<br>Il parametro di un'espressione non viene valutato finché non chiama la funzione<br><i>def</i> metodo(parametri: ()) => <i>tipoRestituito</i> |                                                 |
| Chiamata per valore:<br>Il valore del parametro viene valutato prima che sia passato al metodo                                                                     | Chiamata per valore:<br>metodo(parametri)       |
|                                                                                                                                                                    | Chiamata per riferimento:<br>metodo(parametri*) |
| lazy vals<br>es: <i>val</i> value: Int<br><i>lazy val</i> inv = utilizzo value...                                                                                  |                                                 |

Quindi si può creare un nuovo oggetto *Pair* con `val p = Pair(1, "one")`. Può sembrare che si stia creando un'istanza di *Pair* senza un *new*, in realtà, invece di invocare direttamente il costruttore di *Pair*, viene invocato `Pair.apply`, che chiama `Tuple2.apply` sull'oggetto `Tuple2`.

In Java, una classe può implementare un numero arbitrario di interfacce. Questo modello è molto utile per dichiarare che una classe espone più astrazioni, ma ha un grave inconveniente. Per molte interfacce, molte delle funzionalità possono essere implementate con codice standard che sarà valido per tutte le classi che utilizzano l'interfaccia. Java non fornisce alcun meccanismo intrinseco per la definizione e l'utilizzo di tale codice riutilizzabile. I programmatori Java devono utilizzare convenzioni ad hoc per il riutilizzo del codice d'implementazione per una data interfaccia. Spesso, l'implementazione di un'interfaccia ha membri che non sono collegati al resto dei membri dell'istanza. Il termine *mixin* viene spesso utilizzato per questi pezzi di codice potenzialmente riutilizzabili di un'istanza che potrebbe essere mantenuta in modo indipendente.

Scala fornisce una soluzione mixin chiamata *trait*. È possibile definire l'astra-

zione di una callback in un trait, come in un'interfaccia Java, ma è possibile anche realizzare l'astrazione del trait (o derivare un trait). In Scala è possibile anche mescolare i trait e allo stesso tempo crearne le istanze. I trait preservano la separazione degli argomenti, mentre dà la possibilità di comporre un comportamento su richiesta.

Esempio comparativo da Java e Scala:

In Java si possono usare le interfacce per simulare un trait

Listing 4.8: Esempio codice Java

```
public interface Singer {  
    public void sing();  
}  
  
public class Bird implements Singer{  
    ...  
    public void sing(){...}  
}  
...  
  
public class Cicala extends Singer{  
    ...  
    public void sing(){...}  
}
```

In Scala è più elegante la definizione del comportamento “sing()” e il riutilizzo del tutto indipendente, separando questa funzione come un trait e l'utilizzo di quel trait come un mixin.

Esempio di utilizzo *statico*:

```
trait Singer{  
    def sing { println("Singing...") }  
}  
...  
  
class Bird extends Singer
```

I trait si possono anche estendere:

```
class Insect  
class Cicada extends Insect with Singer  
class Bird extends Singer with Flyer //Flyer è un altro trait
```

Esempio di utilizzo *dinamico*: Nel caso di una classe `Person`, ci si trova ad affrontare un altro problema particolare; Si vuole che solo alcune istanze di `Persona` siano cantanti. Non si è in grado di implementare l'interfaccia `Singer` della classe `Person` poiché ciò trasforma ogni istanza di persona in un cantante.

Scala permette di mixare “dinamicamente” un trait quando si crea una nuova istanza di una classe. In questo caso, solo l'istanza speciale sarà un cantante e vengono forniti i metodi di quel trait:

```
class Person {  
  def tell { println("Hello...") }  
}  
val singingPerson = new Person with Singer  
person.sing
```

Come si può vedere, è stata creata una nuova istanza di tipo `Person`, dicendo che in questo caso è anche una cantante, utilizzando la parola chiave `with`.

```
println( "class of singing person: " + singingPerson.getClass ) //  
-> com.mgi.traits.TraitsAsMixins$$anon$2  
println( "class of singing person is a Person? " +  
  singingPerson.isInstanceOf[Person] ) // -> true  
println( "class of singing person is a Singer? " +  
  singingPerson.isInstanceOf[Singer] ) // -> true
```

A differenza di Java si può chiamare un metodo su tale istanza, senza type-cast, non importa se il metodo è stato originariamente definito in `Persona` o in `Singer`.

Naturalmente si possono incontrare alcuni problemi durante l'invio di tale istanza a un metodo che si aspetta un parametro di tipo `Person`. In tale ambito, un `Person` (in generale) non è un cantante, quindi la chiamata `sing()` potrebbe causare un errore dal momento che `sing()` non è membro regolare della classe `Person`.

In conclusione, i trait in Scala sono un modo elegante per separare i concetti. Ogni funzione può essere separata all'interno di un proprio trait e può essere mixato in ogni tipo o istanza che deve possedere quel trait.



# Capitolo 5

## Il modello ad Attori

### 5.1 Multi-core e Java

Negli ultimi anni, le architetture delle CPU sono migrate dal single-core verso il multi-core. L'architettura di base, fondamentalmente ha diversi core, ma una bassa velocità di clock per core.

Questo è un problema per Java perché:

- Java ha un path length per File e Cartelle più lungo rispetto a linguaggi come C e la bassa velocità di clock non aiuta. Infatti Java limita la lunghezza dei percorsi a 2031 caratteri<sup>1</sup> <sup>2</sup> mentre il filesystem ne può contenere al massimo 247, quindi i percorsi vengono troncati oppure si genera un errore;
- JavaEnterpriseEdition promuove un modello di Thread che presenta molteplici punti deboli, tra i quali:
  - La JVM gestisce i Thread secondo il modello *shared memory* la cui gestione è pesante ed incorre in penalizzazioni dovute a overhead causate da frequenti context switch. Da notare anche il problema dovuto alla gestione dell'heap. L'heap infatti è condiviso da tutti i Thread. Quando viene creato un oggetto, esso entra in concorrenza con i Thread presenti nella struttura dati, limitato anche dalla forma di sincronismo. Ci sono lavori di ricerca che, partendo dall'ipotesi di avere sulla JVM Thread indipendenti tra di loro, cercano di creare zone di memoria private per ogni Thread

---

<sup>1</sup>[bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=4165006](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4165006) e

<sup>2</sup>[publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=/com.ibm.java.doc.user.zos.60/user/limitations.html](http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=/com.ibm.java.doc.user.zos.60/user/limitations.html)

- senza incorrere nei problemi di sincronismo e abbassando i livelli di overhead nel sistema;
- Nei sistemi a 32bit, lo spazio di indirizzamento è di  $2^{32} = 4GB$  ed è limitante nella gestione di numerosi Thread (anche se il problema è stato ovviato con l'introduzione dei sistemi a 64bit);
  - I Thread hanno tutti una dimensione fissa dello stack associato, sprecando molta memoria.
- La Garbage Collector è fortemente dipendente dalla velocità di clock nelle situazioni in cui interviene per le operazioni di pulizia.
  - Come sopracitato, tra i problemi noti del modello vi è l'overhead da frequenti context switch. Un context switch è il processo di memorizzazione e di ripristino dello stato di una CPU, detto contesto, in modo che l'esecuzione possa essere ripresa dallo stesso punto in un secondo momento. Ciò consente a più processi di condividere una singola CPU. Il context switch è una caratteristica essenziale per un sistema operativo multitasking; sono di solito computazionalmente intensivi e gran parte della progettazione dei sistemi operativi ne riguarda l'ottimizzazione dell'uso. Un cambio di contesto può interessare il registro, delle attività o intercorrere in un Thread o un processo. Le informazioni che costituiscono il contesto sono determinate dal processore e dal sistema operativo. cambiare di processo richiede una certa quantità di tempo per l'amministrazione, il salvataggio, il caricamento dei registri e delle aree di memoria, l'aggiornamento delle varie tabelle, ecc. . . Ci sono tre situazioni in cui si verifica un context switch:
    - **Multitasking:** In un algoritmo di scheduling, un processo deve essere commutato fuori dalla CPU in modo che un altro processo possa essere eseguito. All'interno di un sistema operativo di tipo preemptive multitasking, lo scheduler permette ad ogni task di esser eseguito per una certa quantità di tempo, detta *time slice*. Se un processo non cede volontariamente la CPU, si attiva un interrupt timer e il sistema operativo schedula un altro processo da eseguire. Ciò assicura che la CPU non possa essere monopolizzata da un uso intensivo del processore da parte di una sola applicazione;
    - **Interrupt handling:** Le architetture moderne sono di tipo *interrupt driven*. Ciò significa che se la CPU richiede dei dati ad un disco, per esempio, non ha bisogno di sistema di busy-wait finché

la lettura non è conclusa; può sottomettere la richiesta e proseguire con l'esecuzione di altri task. Quando la lettura è finita, la CPU può essere interrotta. Quando si verifica un interrupt, l'hardware passa automaticamente una parte del contesto al gestore di interrupt del programma che l'ha invocato. Nella maggioranza dei casi, solo una minima parte del contesto è cambiata al fine di minimizzare la quantità di tempo speso nella gestione dell'interrupt. Il kernel non crea o schedula alcun processo particolare per gestire gli interrupt, invece il gestore viene eseguito in un contesto stabilito all'inizio della gestione degli interrupt. Una volta che il servizio di interrupt è completato, il contesto verificatosi prima dell'interrupt viene ripristinato in modo che il processo interrotto possa riprendere l'esecuzione nel suo stato corretto.

- **User e Kernel mode switching:** Quando in un sistema operativo è necessaria una transizione tra l'user mode e il kernel mode, non è necessario un context switch; una transizione non è di per sé un cambio di contesto. Tuttavia, a seconda del sistema operativo, un cambio di contesto può avvenire comunque.

L'overhead da context switch è il costo intrinseco che non si può evitare durante un interrupt hardware/software. Attualmente il meccanismo hardware degli context switch è molto lento in quanto consente di salvare quasi tutto lo stato della CPU. I sistemi operativi più moderni non impiegando tecniche di segmentazione della memoria, evitano così di caricare un segmento di registro durante i cambi di contesto; questi sistemi operativi abbandonano il meccanismo di context switch hardware per ragioni prestazionali. Oltretutto le CPU a 64bit non impiegano i context switch hardware quando lavorano a 64bit.

Il problema del path length e dell'esecuzione in un ambiente gestito ha come conseguenza che il codice impiegherà più tempo per l'esecuzione. Anche se ci sono più core, dal punto di vista di una richiesta individuale, la velocità di clock più bassa è quella che peggiorerà i tempi di risposta, aggravato anche dal fatto che tutti i core sono connessi da un'unica cache L3 (vedi Figura 5.1), cache che dovrebbe essere dimensionata per avere un alto hit rate per non peggiorare la situazione.

L'introduzione di una Concurrent GC è stato un punto importante. Il garbage a breve termine viene raccolto in modo molto efficiente. Il problema sono i garbage a lungo termine, come i dati conservati in una cache nella JVM. Se tali dati non cambiano allora si potrebbe dimensionare l'heap a lungo termine in modo che li conservi. Ma, se la cache contiene dati che

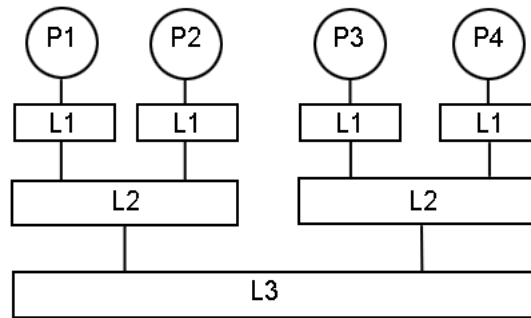


Figura 5.1: Gerarchia delle cache nei sistemi multiprocessore

cambiano (una scrittura attraverso la cache, per esempio) allora quell'heap di grandi dimensioni si riempirà e quando succederà, ci saranno molti di dati contenuti in esso e questo richiederà una pausa per la pulizia lunga. Le cache ora hanno, in aggiunta, funzionalità come l'indicizzazione e questo sottintende che richieda una quantità maggiore di memoria.

## 5.2 Threadless Concurrency su JVM e Scala Actors

I due paradigmi dominanti della concorrenza sono:

- **Shared State con Monitor**, dove la concorrenza viene raggiunta attraverso l'esecuzione sincronizzata di più thread utilizzando lock, ecc...
- **Message Passing**, che è un modello di shared-nothing che utilizza uno scambio di messaggi asincrono attraverso processi leggeri o thread.

Il paradigma Message Passing offre un maggiore livello di astrazione rispetto allo shared state, in quanto l'utente non ha bisogno di interagire direttamente con le primitive di basso livello dei Thread.

Erlang<sup>3</sup> (da cui Scala deriva il suo sistema di gestione degli attori) supporta questo modello di programmazione (e quindi anche Scala) ed è stato ampiamente utilizzato nel settore delle reti di telecomunicazioni raggiungendo un alto grado di parallelismo.

Java supporta il primo modello e la progettazione di applicazioni concorrenti in Java risulta complessa.

---

<sup>3</sup>[www.erlang.org](http://www.erlang.org)

### 5.2.1 Il modello ad Attori

Un attore è un'entità computazionale che, in risposta a un messaggio che riceve, può contemporaneamente:

- Inviare un numero finito di messaggi ad altri attori;
- Creare un numero finito di nuovi attori;
- Indicare il comportamento da utilizzare per il successivo messaggio che riceve.

Non c'è una sequenza già presunta sulle azioni sopra citate e potranno essere svolte in parallelo.

Il modello ad attori è un modello matematico di programmazione concorrente che tratta gli “attori” come tipi primitivi: in risposta ad un messaggio che riceve, un attore è in grado di prendere decisioni locali, creare più attori, inviare più messaggi e determinare come rispondere al successivo messaggio ricevuto.

Il modello adotta la filosofia “*ogni cosa è un attore*”. Questo è simile alla filosofia “*tutto è un oggetto*” usata da alcuni linguaggi OOP, ma differisce in quanto i software OOP sono in genere eseguiti in modo sequenziale, mentre il modello ad attori è intrinsecamente concorrente.

Il disaccoppiamento del mittente dalle comunicazioni inviate è stato un progresso fondamentale del modello che permette la comunicazione asincrona e strutture di controllo come il modello Message Passing. I destinatari dei messaggi sono identificati da un indirizzo, a volte chiamato “*mailing address*” e l'attore può comunicare solo con gli attori di cui ha noti gli indirizzi, ottenuti da un messaggio che riceve o se l'indirizzo è per un attore che ha creato da sé.

Il modello è caratterizzato da un concorrenza intrinseca interna e tra gli attori, la creazione dinamica di attori, l'inclusione di indirizzi degli attori nei messaggi e l'interazione solo attraverso il passaggio diretto di messaggi asincroni senza restrizioni su l'ordinamento del messaggio di arrivo.

### 5.2.2 Attori e JVM

La concorrenza basata sugli attori in Erlang è altamente scalabile e offre un livello più grossolano di modello di programmazione agli sviluppatori. Scala prende da Erlang lo stile di concorrenza basata sugli attori su JVM. Gli sviluppatori possono progettare applicazioni concorrenti scalabili sulla JVM utilizzando il modello degli attori di Scala da cui ne trarranno vantaggio i processori multi-core, senza utilizzare il modello di Thread di Java. Nelle applicazioni che richiedono un grande numero di processi simultanei in una quantità limitata di memoria, i Thread della JVM, dimostrano di essere un peso significativo a causa del sovraccarico dello stack e locking contention. Gli attori in Scala forniscono un modello ideale per la programmazione in un ambiente virtual machine non cooperativo. Assieme alla capacità di pattern matching del linguaggio Scala, si possono sfruttare tutti i benefici della concorrenza di Erlang sulla JVM.

In Scala, un attore è un oggetto che riceve un messaggio e compie determinate azioni sulla base dello stesso. L'ordine in cui si ricevono i messaggi non è importante anche se in Scala esiste una coda di messaggi per ogni attore.

Un attore può gestire un messaggio internamente, spedire un altro messaggio oppure creare un nuovo attore sulla base del messaggio ricevuto.

Un attore non esegue una sequenza o un ordine di azioni ed ha uno stato globale condiviso. Queste due cose permettono l'esecuzione in parallelo di più attori.

Listing 5.1: Esempio definizione attore

```
import scala.actors.Actor
def Attore extends Actor
{
    def act()
        { logica implementativa... }
}
val x = new Attore
x.start
```

Un attore riceve qualsiasi tipo di oggetto come messaggio, per questo gli attori fanno uso di pattern matching anche se un attore agisce su un messaggio di una famiglia di tipi mentre la Pattern Matching Machine (PMM) sulla classe e/o contenuto del messaggio.

Listing 5.2: Esempio logica implementativa

```

...
loop
{
    receive
    {
        case msg1 => action1
        ...
        case msgN => actionN
    }
}

```

`loop` esegue un `while(true)` del suo corpo. `receive` è una chiamata bloccante finché non riceve un messaggio del tipo accettato dalla PMM (i vari case...). La tabella 5.1 mostra l'elenco dei metodi a disposizione di un attore.

Tabella 5.1: Metodi di un attore

| Metodo                     | Return                   | Descrizione                                                                     |
|----------------------------|--------------------------|---------------------------------------------------------------------------------|
| <code>act</code>           | <code>Unit</code>        | Contiene uno dei metodi sottoelencati                                           |
| <code>receive</code>       | messaggio                | <b>Bloccante</b> fino a che non riceve un messaggio corretto                    |
| <code>receiveWithin</code> | messaggio                | Come <code>receive</code> ma si <i>sblocca</i> dopo un certo # di ms            |
| <code>react</code>         | <code>Nothing</code>     | Richiede meno Thread di <code>receive</code>                                    |
| <code>reactWithin</code>   | <code>Nothing</code>     | Come <code>react</code> ma si <i>sblocca</i> dopo un certo # di ms              |
| <code>!</code>             | <code>Unit</code>        | Invia in modo asincrono un messaggio qualunque                                  |
| <code>!!</code>            | <code>Future[Any]</code> | Invia il messaggio e restituisce il valore future che rappresenta la risposta   |
| <code>?!</code>            | <code>Option[Any]</code> | Invia il messaggio e rimane in attesa della risposta (metodo di invio sincrono) |

Si usa `react` quando possibile mentre, se si vuole il risultato del messaggio, meglio usare `receiveWithin`.

Un'altra strategia per mantenere l'attore asincrono è l'utilizzo di `futures`.

`futures` è un oggetto segnaposto per un valore che non è ancora stato restituito da un processo asincrono. È possibile inviare un messaggio ad un attore con il metodo `!!`; una variante consente di passare il messaggio ad una partial function (si veda prossimo paragrafo per il dettaglio) che viene applicata al valore futuro. `futures` è un'operazione bloccante. Come si può vedere dal seguente esempio, il recupero di un valore da un `futures` è simile all'invocazione del metodo `apply` (discusso più avanti).

Listing 5.3: Esempio di utilizzo di Futures

```
import scala.actors.Futures._
...
    val eventually = future(5*4)
    println(eventually())           >res: 20
```

Gli attori non sono l'unico modo di applicare la concorrenza di Scala. Dal momento che è interoperabile con Java, possono esser usati anche Thread. È possibile usare il pacchetto `java.util.concurrent` per usare *Executors* per creare un Thread pool.

## Partial Function

Sono espressioni in cui non tutti gli argomenti definiti in una funzione vengono forniti come parametri alla funzione. In Scala, le funzioni parziali sono utilizzate per costruire una funzione, tra cui i suoi parametri e il tipo restituito, e assegnare tale funzione ad una variabile o passarla come argomento ad un'altra funzione.

Di seguito viene riportato un esempio di utilizzo delle Partial Function:

Listing 5.4: Esempio di Partial Function

```
...
def concatUpper(s1: String, s2: String): String = (s1 + " " +
    s2).toUpperCase

val c = concatUpper _
println(c("Hello", "world"))

val c2 = concatUpper("Hello", _: String)
println(c2("world"))
```

Chiamando `concatUpper` con un underscore (`_`) si trasforma il metodo in un valore della funzione. Nella prima parte, ad esempio, viene assegnato una

versione parzialmente applicata di `concatUpper` al valore di `c`. Quando viene invocato *apply*, implicitamente si richiama tale metodo `c` passando direttamente i parametri. Il valore restituito viene quindi stampato.

Nella seconda parte, è specificato il primo parametro di *concatUpper* ma non il secondo, anche se viene specificato il tipo del secondo parametro. Viene assegnata questa variante ad un secondo valore, `c2`. Per produrre lo stesso output del precedente, bisogna passare solo un unico valore quando si applica `c2`.



# Capitolo 6

## Il modello ad Eventi

Thread ed attori non sono gli unici modelli di concorrenza. la concorrenza basata sugli eventi é un particolare approccio asincrono o NIO (non-blocking I/O) per scrivere, ad esempio, un server scalabile a migliaia di client contemporaneamente.

Questo modello di concorrenza espone eventi che si verificano quando particolari condizioni sono soddisfatte (per esempio, quando i dati vengono ricevuti da un client su un socket di rete). In genere, il programmatore associa una callback con ogni evento che é rilevante per il suo programma.

Il pacchetto `java.nio` offre una varietà di primitive per il NIO ma sono ancora troppo semplici per creare eventi funzionali.

### 6.1 React vs. Receive (Event-based vs. Thread-based)

Il problema dell'approccio a Thread non è rappresentato dal Thread in sé, bensì il vero collo di bottiglia è il modello a memoria condivisa. Con un modello `shared-nothing` i thread assumono molto più di quello che sente un "processo". Linguaggi come Erlang e Termite implementano un modello basato su processi leggeri `shared-nothing` al di sopra del livello del sistema operativo. I vantaggi della condivisione di processi leggeri all'interno di uno spazio unico d'indirizzamento, prende il meglio dai due modelli (thread-based, event-based).

- **Thread-based:** il comportamento di un processo concorrente é definito mediante l'implementazione di un metodo `thread-specific`. Lo stato di esecuzione é gestito da un Thread stack associato.

- **Event-based**: il comportamento è definito da una serie di (non-nested) gestori di eventi che vengono chiamati dall'interno di un ciclo. Lo stato di esecuzione di un processo concorrente è mantenuto da un record associato o un oggetto.

L'idea di base è quella di usare un **Thread Pool** per eseguire gli attori, *ridimensionando* il pool ogni volta che è necessario. Se un attore utilizza solo operazioni del modello ad eventi, la dimensione del pool può essere fissa (non nel caso che utilizzi operazioni bloccanti come `receive` o `system I/O`).

Nel caso in cui ogni Thread attivo sia occupato per un attore bloccante e ci siano task in coda, il pool cresce. La situazione di *deadlock* viene evitata aumentando la dimensione del pool quando è necessario. Bisogna aggiungere un ulteriore Thread ogni volta che ci sono task in coda e tutti i Thread sono bloccati. In questo caso, i task in coda sono gli unici che potrebbero sbloccare uno dei Thread bloccati. Per far ciò, il **Thread Scheduler** (separato dai Thread del Thread Pool) controlla periodicamente la presenza di task in coda e se tutti i Thread sono bloccati. In questo caso viene creato un nuovo Thread nel pool che processerà i task rimanenti.

Quando si dice "Thread leggero", si intende il fatto che ciascun attore non è mappato in un singolo Thread.

La JVM offre Thread con shared memory, con lock come forma primaria d'astrazione di concorrenza. Ma i Thread con shared memory sono piuttosto pesanti e incorrono in severe penalizzazioni a causa di overhead da context switch. Per un'implementazione basata sugli attori sulla base di un mapping uno-a-uno con i Thread della JVM, il payload per un attore Scala sarà leggero come generare un milione di attori per un calcolo specifico<sup>1</sup>. Quindi gli attori Scala sono stati progettati come oggetti ad eventi leggeri, che vengono schedulati ed eseguiti su un Thread Pool che viene automaticamente ridimensionato quando tutti i Thread si bloccano a lungo. In realtà, Scala implementa un modello unificato del modello basato su attori-Thread e del modello basato sugli eventi.

Gli attori Scala offrono due forme di meccanismo di sospensione:

- Una sospensione *full frame stack* (implementata con `receive`);
- Una sospensione sulla basata su una *closure continuation* (implementata con `react`).

In caso di eventi basati sugli attori, una *wait* su `react` è rappresentata da una closure continuation, per esempio, una closure che cattura il resto di una computazione di un attore.

---

<sup>1</sup>[www.scala-lang.org/node/25](http://www.scala-lang.org/node/25)

Quando un attore sospeso riceve un messaggio che corrisponde ad uno dei modelli di cui l'attore riconosce, la continuation è eseguita schedulando il task ad uno dei Thread in esecuzione nel Thread Pool.

### 6.1.1 React

Di seguito viene riportata l'implementazione della funzione `react`

Listing 6.1: Codice sorgente `react`

```
def react(f: PartialFunction[Any, Unit]): Nothing =
  synchronized {
    mailbox.dequeueFirst(f.isDefinedAt) match
    {
      case Some(msg) => schedule(new Task({ () =>
        f(msg) })))
      case None => continuation = f
                    isDetached = true
                    waitingFor = f.isDefinedAt
    }
    throw new SuspendActorException
  }
```

Il tipo di ritorno `Nothing`, indica che una funzione non restituirà normalmente mai nulla ma terminerà sempre con un'eccezione.

Quando si invoca `start` su di un attore, il metodo organizza le cose in modo che alcuni Thread invochino `act` su quell'attore. Se `act` invoca `react`, `react` cercherà nella mailbox dell'attore un messaggio, passato dalla funzione parziale, in grado di gestire. (Si comporta come `receive`, passando i messaggi al metodo `isDefinedAt`). Se trova un messaggio che può essere gestito, `react` programmerà il trattamento di questo messaggio per l'esecuzione successiva e lancerà un'eccezione. Se non ne trova, porrà l'attore in “*cold storage*”, per poi risvegliarlo se e quando arriveranno più messaggi nella sua mailbox, e genera un'eccezione.

In entrambi i casi, `react` termina la sua esecuzione con questa eccezione, e quindi continua con l'esecuzione di `act()`.

Il Thread che ha invocato `act` cattura l'eccezione, dimenticando l'attore, e passa ad altri compiti. Questo è il motivo per cui se si invoca `react` per gestire più messaggi oltre al primo, bisogna invocare nuovamente `act` da dentro la funzione parziale, o usare altri metodi per invocare `react`.

Di seguito, un esempio di Produttore/Consumatore con buffer che utilizza il metodo `react`.

**Algorithm 6.1** Pseudocodice P-C con buffer che usa react

---

```

1: loop
2:   react
3:   repeat
4:     if (msg == true) then
5:       f(msg) and throw exception // processa msg
6:     else
7:       congela attore and throw exception
8:     end if
9:   until Match nel Buffer
10: end loop

```

---

**6.1.2 Receive**

Se c'è un messaggio nella mailbox dell'attore corrente che corrisponde a uno dei casi specificati nella funzione parziale *f*, viene restituito il risultato dell'applicazione di *f* a quel messaggio.

In caso contrario, il thread corrente viene sospeso; questo consente all'attore ricevente di riprendere l'esecuzione normalmente quando riceve un messaggio corretto. Si noti che viene mantenuto lo stack delle chiamate del ricevente; il comportamento dell'attore è dunque un programma sequenziale che corrisponde alla programmazione thread-based.

Di seguito viene riportato l'implementazione della funzione `receive`

Listing 6.2: Codice sorgente receive

```

def receive[R](f: PartialFunction[Msg, R]): R = {
  ...
  while (!done) {
    val qel = mailbox.extractFirst((m: Any) => f.isDefinedAt(m))
    if (null eq qel) {
      synchronized {
        ...
        waitingFor = f.isDefinedAt
        suspendActor()
        done = true
      }
    }
    else {
      received = Some(qel.msg)
      senders = qel.session :: senders
      done = true
    }
  }
}

```

```
    }  
  }  
  val result = f(received.get)  
  received = None  
  senders = senders.tail  
  result  
}
```

Di seguito, un esempio di Produttore/Consumatore con buffer che utilizza il metodo `receive`.

---

**Algorithm 6.2** Pseudocodice P-C con buffer che usa receive

---

```
1: loop  
2:   receive  
3:   repeat  
4:     if (msg == true) then  
5:       f(msg) and throw exception // elabora msg  
6:     else  
7:       wait()  
8:     end if  
9:   until Match nel Buffer  
10: end loop
```

---

## 6.2 Test sulla funzione react

Sono stati svolti dei test sulla funzione `react` di cui di seguito sono riportati codice, risultati ottenuti e considerazioni, utilizzando il modello Produttore/Consumatore.

Con questi test, si vuole andare a valutare il comportamento della funzione `react` in diversi ambiti, per comprendere il funzionamento dei task nelle situazioni di seguito illustrate.

I vari test sono stati effettuati su una macchina con un processore Intel core 2 Duo 2.31GHz a 64 bit e 4GB di RAM e i test di performance sono stati valutati con il tool *JVisualVM* (Java Virtual Machine Monitoring) di Java. Per tutti i test sono state fatte delle ipotesi sulle relazioni tra i seguenti tempi:

Ipotesi iniziali:  $T_1 < T_2 < T_3 < T_4$

### 6.2.1 Test1

Consumer

Producer

```
loop{
  react{
    case MSG1 => print("START MSG1")
                sleep T2
                print("END MSG1")
    case MSG2 => print("MSG2")
  }
}
```

```
c ! MSG1
sleep T1
c ! MSG2
sleep T3
```

#### Result

- 1: **SEND:** MSG1
- 2: **REC:** START MSG1
- 3: **SEND:** MSG2
- 4: **REC:** END MSG1
- 5: **REC:** MSG2

**Obiettivo:** Verifica di non-blocking di react.

**Svolgimento:** In questo test, nella funzione react sono inseriti due case match su due messaggi diversi. Quando il produttore invia al consumatore MSG1 (punto 1) si mette in sleep per un tempo T1 breve. Il consumatore, quando entra nel corpo del react si aspetta di ricevere o un messaggio di tipo MSG1 o MSG2. Nel ricevere MSG1, stampa un avviso di avvenuta ricezione (punto 2) ed esegue uno sleep T2 di durata superiore a T1 ed infine stampa un avviso di fine esecuzione (punto 4) ritornando così all'inizio del `loop` e rimanendo in attesa di ulteriori messaggi. Il consumatore alla fine del suo tempo di sleep T1, si risveglia e invia un MSG2 (punto 3) e ritorna in sleep con un tempo T3 superiore al precedente per poi ripetere all'infinito dall'inizio il ciclo. Quando il consumatore riceve un messaggio di tipo MSG2 stampa solo l'avviso di ricezione (punto 5) e ritorna all'inizio del loop.

**Conclusione:** Dal test si evince che l'invio di un MSG2 mentre il consumatore sta ancora processando il MSG1 (perché il tempo T1 di attesa del produttore è più breve del tempo di elaborazione T2, del consumatore) esso verrà posto in coda in attesa che il MSG1 abbia finito l'esecuzione, quindi non c'è preemption del task.

### 6.2.2 Test2

Consumer

Producer

```
loop{
  react{
    case MSG1 => print("START MSG1")
                sleep T2
    react{
      case MSG3 => print("MSG3")
                print("END
                      MSG1")
    }
    case MSG2 => print("MSG2")
  }
}
```

```
c ! MSG1
sleep T1
c ! MSG2
sleep T3
c ! MSG3
sleep T4
```

#### Result

- 1: **SEND:** MSG1
- 2: **REC:** START MSG1
- 3: **SEND:** MSG2
- 4: **SEND:** MSG3
- 5: **REC:** MSG3
- 6: **REC:** END MSG1
- 7: **REC:** MSG2

Testare il funzionamento di react annidate

Obiettivo:

Rispetto al *Test1*, in questa prova si è pensato di testare il comportamento del metodo react all'interno di un altro react. Come si può osservare dal codice, l'annidamento avviene all'interno di un case/match. Il produttore invia il MSG1 (punto 1) al consumatore e si mette in sleep per un tempo T1 breve. Il consumatore, come nel caso precedente, all'interno della react presenta due case su messaggi di tipo MSG1 e MSG2. Alla ricezione del MSG1, il consumatore stampa la conferma di ricezione (punto 2) e si mette in sleep per un tempo T2 superiore a T1 del produttore. Trascorso il tempo T2, il consumatore entra nella react annidata e rimane in attesa di ricevere un terzo tipo di messaggio, MSG3. Il produttore, trascorso il tempo T1 di attesa, invia il messaggio MSG2 (punto 3) e si mette in sleep per un tempo T3 maggiore del tempo T2 del consumatore. Trascorso questo tempo, il produttore invia in terzo messaggio (punto 4) MSG3 e torna in sleep per un tempo T4 maggiore del tempo T3 di attesa precedente. Ricevendo il terzo messaggio MSG3, il consumatore può entrare nel case stampando la ricezione del messaggio MSG3

Svolgimento:

(punto 5); uscendo dal case esegue il codice residuo della react annidata e stampa la fine (punto 6) del case principale MSG1. Infine, entra nel secondo case per la precedente ricezione di MSG2 e ne stampa l'avviso di ricezione (punto 7).

Conclusione: Da questo esperimento, si può osservare che in presenza di react annidati, alla ricezione del MSG1, il produttore rimane in attesa sulla react annidata aspettandosi di ricevere il MSG3. Ulteriori messaggi ricevuti prima del MSG3, vengono accodati in un buffer. Solo dopo aver ricevuto il MSG3 viene conclusa l'esecuzione del restante codice del case MSG1 (corpo contenuto nella react innestata) e prelevando successivamente i messaggi accodati, come il MSG2, per processarli.

6.2.3 Test3

| Consumer                                                                                                                                                                                                                                                                | Producer                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| <pre>loop{   react{     case MSG1 =&gt; print("START MSG1")                   sleep T2                   react{                     case MSG3 =&gt; print("MSG3")                   }                   print("END MSG1")     case MSG2 =&gt; print("MSG2")   } }</pre> | <pre>c ! MSG1 sleep T1 c ! MSG2 sleep T3 c ! MSG3 sleep T4</pre> |

| Result                    |
|---------------------------|
| 1: <b>SEND:</b> MSG1      |
| 2: <b>REC:</b> START MSG1 |
| 3: <b>SEND:</b> MSG2      |
| 4: <b>SEND:</b> MSG3      |
| 5: <b>REC:</b> MSG3       |
| 6: <b>REC:</b> MSG2       |

Testare il funzionamento di react annidate

Obiettivo:

Il test è del tutto identico al TEST2 se non per la porzione di codice che segue la react annidata. In questa prova si è studiato cosa succede se il codice segue la react anziché essere posto all'interno della stessa. Dopo la ricezione del MSG3 (punto 5), ci si aspetterebbe di vedere stampato il messaggio che avvisa la fine del case MSG1. Invece, viene ignorato tutto il codice successivo alla react annidata e prosegue con la ricezione di MSG2 (punto 6) come nel TEST2.

Svolgimento:

Dal comportamento del programma e dall'analisi del codice di `react`, può dedurre che tutto il codice dopo il corpo di una react non verrà mai eseguito perché il tipo di dato restituito della funzione react è Nothing, quindi non arriverà mai in quel punto del codice.

Conclusione:

### 6.2.4 Test4

| Consumer                                                                                                                                                                                                                                                                                                      | Producer                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <pre> loop{   react{     case MSG1 =&gt; print("START MSG1")                 sleep T2                 react{                   case MSG3 =&gt; print("MSG3")                               print("END                                 MSG1")                 }     case MSG2 =&gt; print("MSG2")   } } </pre> | <pre> c ! MSG3 sleep T1 c ! MSG2 sleep T3 c ! MSG1 sleep T4 </pre> |

#### Result

- 1: **SEND:** MSG3
- 2: **SEND:** MSG2
- 3: **REC:** MSG2
- 4: **SEND:** MSG1
- 5: **REC:** START MSG1
- 6: **REC:** MSG3
- 7: **REC:** END MSG1

**Obiettivo:** Testare il funzionamento di react annidate

**Svolgimento:** Anche questo test è del tutto identico al TEST2 ma solo dal lato del consumatore. Ciò che cambia, invece, è il lato del consumatore in quanto viene inviato per primo il MSG3 (punto 1) seguito dal MSG2 (punto 2) e infine il MSG1 (punto 4). L'aver cambiato l'ordine del primo e ultimo messaggio ha comportato l'accodamento del buffer in quanto il consumatore, trovandosi nella react principale, non aveva trovato riscontro nei suoi case. Solo ricevendo il MSG1 (punto 5) ha potuto entrare nella react annidata e il consumatore ha prelevato dalla coda del buffer il MSG3 (punto 6), concludendo poi la sua esecuzione (punto 7).

**Conclusione:** Da questo test è possibile vedere che l'invio preventivo di un messaggio, non presente nei suoi case/match, viene accodato nel buffer in attesa di un eventuale processing. Solo all'arrivo del messaggio, che innesci la react annidata, il consumatore ha potuto riconoscere con successo il MSG3 presente nel buffer così da poter esser elaborato.



### 6.2.6 Test6

| Consumer                                                                                                                                                                                                                     | Producer                                                           |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <pre> loop{   react{     case MSG1 =&gt; print("MSG1")                   sleep T2                   funct()     case MSG2 =&gt; print("MSG2")   } }  def funct() = {   react{     case MSG4 =&gt; print("MSG4")   } } </pre> | <pre> c ! MSG1 sleep T1 c ! MSG2 sleep T3 c ! MSG4 sleep T4 </pre> |

#### Result

- 1: **SEND:** MSG1
- 2: **REC:** MSG1
- 3: **SEND:** MSG2
- 4: **SEND:** MSG4
- 5: **REC:** MSG4
- 6: **REC:** MSG2

**Obiettivo:** Testare il funzionamento di react come chiamata a funzione

**Svolgimento:** In questo test, il produttore invia un MSG1 (punto 1) e si mette in sleep per un tempo T1. Il consumatore alla ricezione di tale messaggio, notifica la l'evento (punto 2) e si mette in sleep per un tempo T2 superiore a T1. Trascorso questo tempo, esegue una chiamata a funzione FUNCT() il cui corpo è costituito da una react su un messaggio di tipo MSG4. Il produttore, trascorso il tempo T1, riprende la sua esecuzione inviando al consumatore un messaggio MSG2 (punto 3) per poi tornare in sleep per un tempo T3 superiore al tempo T2 di attesa del consumatore. Il consumatore, aspettandosi un messaggio di tipo MSG4, accoda nel buffer il messaggio MSG2 appena ricevuto. Dopo il tempo T3 di sleep, il produttore invia un messaggio MSG4 (punto 4) e si mette in sleep per un tempo T4 più lungo del tempo T3 precedente. Il consumatore, riceve correttamente il MSG4 e lo notifica (punto 5) prelevando, successivamente, dal buffer il MSG2 (punto 6) per il match con il secondo case

incontrato al ritorno dalla chiamata a funzione.

Si può vedere che la `react` è supportata anche da una chiamata a funzione. Conclusione:

Da questi test si evince che il comportamento della `react` non è una vera e propria continuation come ci si aspettava che fosse, bensì, l'attuale framework sugli attori di Scala, si basa sul concetto di closure. Per osservare il comportamento della vera continuation, bisogna imporlo esplicitamente sia nel codice che in fase di compilazione.

Esistono dei progetti che utilizzano il vero concetto di continuation quali **Javaflow**<sup>2</sup> e **Kilim**<sup>3</sup>.

---

<sup>2</sup>[commons.apache.org/sandbox/javaflow](https://commons.apache.org/sandbox/javaflow)

<sup>3</sup><http://www.malhar.net/sriram/kilim/>



# Capitolo 7

## Scala Scheduler

In questo capitolo si valuterà il comportamento dello scheduler in Scala. Se tra i punti di forza di Scala si può citare il modello Threadless per la gestione della concorrenza, si deve dare merito al tipo di Scheduler sottostante. Il contributo allo sviluppo dello Scheduler lo si deve attribuire alla “State University of New York at Oswego” e nello specifico a Doug Lea. Come si può leggere in [19] e [20], l'autore parte da una critica all'inefficienza del modello a Thread di Java in quanto troppo pesante da eseguire. Esso infatti mappa ogni Task su un Thread per CPU ed agisce ad un livello sottostante il Sistema Operativo. Lea, al contrario, adotta una strategia che cerca di alleggerire il carico con un doppio mapping. Deriva il suo lavoro da Cilk<sup>1</sup>, un framework per la programmazione concorrente che si posiziona su un layer soprastante il Sistema Operativo (al contrario di Java) mappando i Task in LWP o Kernel Thread che a loro volta sono mappati sulla CPU. Il vantaggio principale di questa configurazione è di permettere a programmi a cui è possibile applicare il meccanismo di Divide & Conquer, una migliore portabilità, semplicità e regolarità. Lea, infatti, prendendo spunto da Cilk, ha creato uno Scheduler di tipo Fork/Join che permette una migliore parallelizzazione dell'esecuzione dei Task. In generale, un algoritmo Fork/Join, quando incontra una *fork*, divide i task in sottoTask dalla risoluzione più semplice per poi essere ricongiunti con il *join* per il raggiungimento della soluzione al Task padre. Un vantaggio che segue da questo tipo di algoritmo è la capacità di non bloccarsi mai se non per l'esecuzione dei sottoTask.

L'algoritmo crea un Pool di Worker Thread (WT), ognuno dei quali è un normale Thread. Normalmente ci sono in esecuzione molti Thread e, per questa ragione, l'algoritmo li mappa su dei LWP, come spiegato in precedenza, così come ogni Task da eseguire (con la possibilità di cambiare unicamente il

---

<sup>1</sup>[supertech.csail.mit.edu/cilk/](http://supertech.csail.mit.edu/cilk/)

numero di WT in esecuzione, che di solito sono pari al numero di CPU disponibili, e modificare la soglia del caso base dell'algoritmo D&C sopra la quale viene invocata la fork). La gestione dei Task è il cuore dell'algoritmo di Scheduling e si affida ad una strategia di work-stealing, derivata da Cilk. Ogni WT ha associata una particolare coda (head-tail linked list o double-ended queue) gestita sia con strategia LIFO sia FIFO, da cui è possibile inserire ed estrarre sia dalla testa che dalla coda. I Task vengono accodati e prelevati dal WT in modalità LIFO (prelevando quindi quello che da minor tempo è arrivato in coda); anche i sottoTask generati sono accodati nella stessa coda e gestiti con la medesima policy. Nel momento in cui un WT svuota la sua coda e non avendo così altri task da eseguire, seleziona casualmente un WT tra quelli disponibili e “ruba” (da qui il termine *stealing*) un Task (o sottoTask che sia) dalla coda del WT selezionato con modalità FIFO (ovvero prelevando quello che da più tempo è accodato) e lo esegue. Il WT che incontra un join alla fine di un sottoTask non prelevato dalla propria coda, continua a prelevare dalla stessa coda altri sottoTask (sempre in modalità FIFO) fino al completamento della fork che li ha generati. Nel caso in cui un WT fallisca nel reperire un Task da un'altra coda, si mette in uno stato di Idle per un certo periodo di tempo per riprovarci successivamente. Di seguito viene presentato uno schema riassuntivo della strategia di work-stealing:

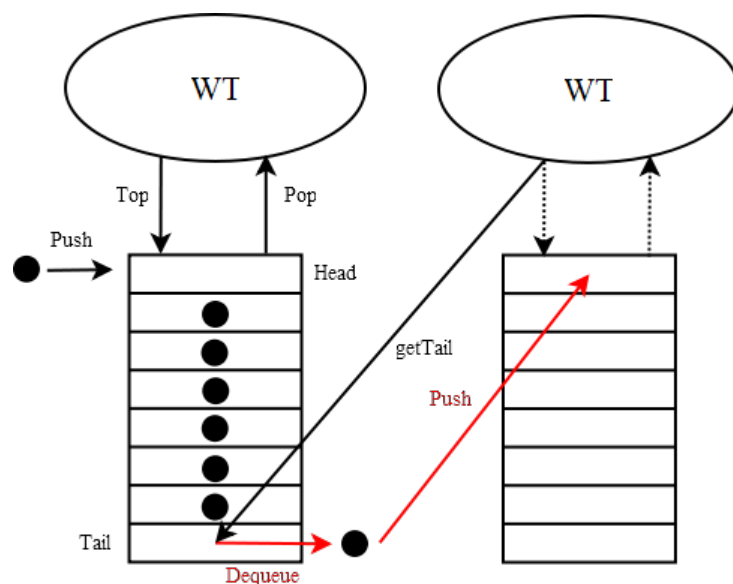


Figura 7.1: Algoritmo di work-stealing

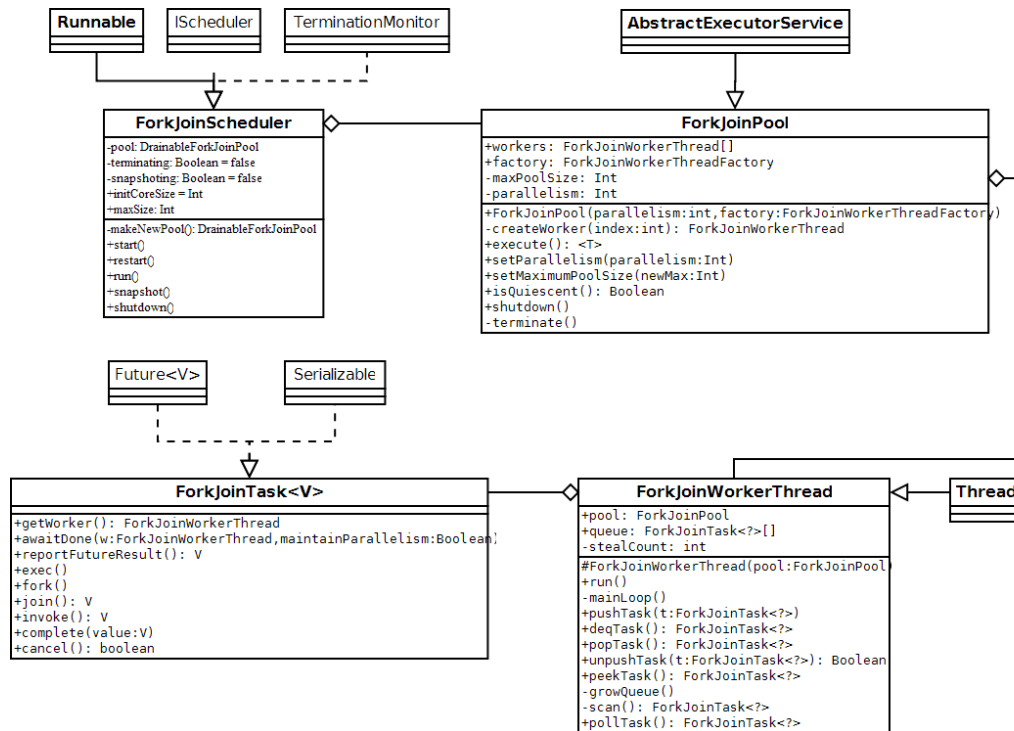


Figura 7.2: Diagramma UML delle Classi

Come da Diagramma UML di Figura 7.2, le classi principali che intervengono nella gestione dello Scheduling sono 4. **ForkJoinScheduler** fa parte della libreria Scala e la sua funzione è quella di invocare e gestire il Thread Pool invocando la Garbage Collector quando necessario. L'invocazione di FJ-Scheduler innesca l'esecuzione di **FJPool**, classe factory per la creazione e la gestione dei WT e fa parte della libreria scritta in Java così come **FJWorkerThread** (implementazione e gestione dei WT) e **FJThread** (implementazione dei Task e sottoTask). Analizzando il comportamento degli attori di Scala, si può affermare che (come descritto anche in [19]) una callback viene sempre mappata in un meccanismo di Fork/Join. Questa situazione, applicata allo scambio di messaggi tra attori, porta a dire che un attore è mappato in un **FJThread** e schedulato all'interno della coda di un WT. Quando questo attore entra in un meccanismo di receive/react, innesca una procedura di Fork creando dei sottoTask che consistono nel messaggio inviato/ricevuto. Alla ricezione del messaggio, (quindi al ritorno dalla callback invocata dal react) si avvia il processo di Join del sottoTask concludendo il passo ricorsivo.

Per testare lo scheduler, è stata costruita una classe *PingPong* fatta come segue:

Listing 7.1: Codice per **Ping**

```
var count = 0
loop{
  react{
    case StopPong(ID) => print("Ping: Pong " + ID + " Fine")
                        count += 1
                        pong(ID) ! StopPing
                        if(count == 100)
                        {
                          println("Stop Ping")
                          exit('stop')
                        }
  }
}
```

Listing 7.2: Codice per **Pong**

```
loop{
  react{
    case Start => print(Start Pong ID)
                 funct() //computazione
                 ping ! StopPong(ID)
    case StopPing => print(Stop Pong ID)
                   exit('stop')
  }
}
```

Listing 7.3: Codice per il **Main**

```
val a: Array[Pong] = new Array(num)
val ping = new Ping(a)
for(i <- 0 to num - 1)
{
  a(i) = new Pong(i, ping)
  a(i).start
}
ping.start
for(i <- 0 to num - 1)
  a(i) ! Start
```

Il programma si comporta nel seguente modo:

- Vengono creati e avviati dal manager (main) un attore di tipo Ping e **num** attori di tipo Pong in un array.
- Il manager invia a tutti i Pong il messaggio Start.
- Appena un Pong riceve uno Start, inizia la sua computazione. Alla fine di questa, invia al Ping un messaggio per informarlo d'aver finito l'elaborazione.
- Ping appena riceve un messaggio di fine elaborazione si appresta a conteggiare il numero di Pong che hanno finito e risponde al Pong con un messaggio di Stop.
- Pong nel ricevere il messaggio di Stop forza la sua uscita e termina.
- Ping, appena riceve il **num**-esimo messaggio di Stop, sa che tutti i Pong hanno finito l'elaborazione e quindi forza anch'esso la sua uscita.

Il motivo per cui “ResizableThreadPoolScheduler” non è lo scheduler predefinito è che è significativamente meno efficiente rispetto allo scheduler basato sul Fork/Join. Inoltre, può essere meno robusto sotto carichi pesanti, se non configurato correttamente.

Le alternative sono due. Utilizzare lo scheduler basato sul Fork/Join oppure forzare l'utilizzo del Thread Pool resizable in gestione automatica o manuale. Dalle prove viste in precedenza, si sarebbe tentati di fare il `maxPoolSize` molto grande. Osservando `scala.actors.Scheduler.ThreadPoolConfig` si è scoperto che il `maxPoolSize` è impostato a **256**, valore abbastanza grande, e quindi ci si è chiesto come mai, nonostante la sovrastima dell'impostazione, avesse delle performance piuttosto deludenti.

Il punto focale sta nel *ForkJoinScheduler* di Java. Esso, infatti, utilizza *ForkJoinPool* che tenta di mantenere fisso il numero di Thread concorrenti impostato dal grado di parallelismo *setParallelism*.

*ForkJoinPool* utilizza il parametro *corePoolSize* per settare *setParallelism* che di default è impostato pari al doppio del numero di CPU. Ad esempio, in un dualcore, di default, non ci saranno mai più di  $2 * \#CPU = 4$  Thread concorrenti attivi.

Per task con lunga elaborazione, questa è una cosa positiva, la CPU non verrà sovraccaricata da troppi *context switch* e *cache miss*. Tuttavia se il Thread è di tipo **blocking I/O**, *ForkJoinScheduler* non ha idea se il Thread stia effettivamente lavorando o è bloccato, quindi tutti i task devono necessariamente aspettare la fine della computazione. Solo se usassi il metodo “!” (vedi tabella 4.2 a pagina 28) (che implementa lo `stop&wait`) lo scheduler è a conoscenza dei Thread effettivamente bloccati ed agirebbe creando un nuovo

Thread nel Thread Pool.

È possibile, nonostante tutto, specificare manualmente che tipo di scheduler debba usare Scala direttamente nel codice:

Listing 7.4: Esempio di dichiarazione di uno scheduler ridimensionabile dinamicamente

```
import scala.actors.Scheduler
import scala.actors.scheduler.ResizableThreadPoolScheduler

...

Scheduler.impl = {
    val s = new ResizableThreadPoolScheduler(false);
    s.start
    s
}
```

In questo caso si agisce direttamente sulle specifiche dell'implementazione dello scheduler andando a dichiarare al suo interno una variabile locale che riferisce all'oggetto *ResizableThreadPoolScheduler* a cui viene passato il valore *false* per utilizzare il demone nel sistema. Successivamente si predispone l'avvio e quindi l'effettiva esecuzione nel sistema.

Alternativamente si può specificare l'utilizzo dell'algoritmo Fork/Join (che doveva essere implementato anche nella nuova versione di Java 7<sup>2</sup> ma posticipato alla 8<sup>3</sup>) e/o modificando i parametri *corePoolSize* e *maxPoolSize*:

Listing 7.5: Esempio di utilizzo di uno scheduler Fork/Join con *corePoolSize* di 5 e *maxPoolSize* di 10

```
...
System.setProperty("actors.corePoolSize", "5")
System.setProperty("actors.maxPoolSize", "10")
System.setProperty("actors.enableForkJoin", "false") //se voglio
    anche il Fork/Join
Scheduler.impl = Scheduler.makeNewScheduler
...
```

La conclusione di questa parte di analisi è che se si utilizza il *ForkJoinScheduler* è meglio non effettuare chiamate bloccanti o utilizzare attori che hanno un lungo carico di lavoro.

<sup>2</sup>[www.oracle.com/technetwork/articles/java/fork-join-422606.html](http://www.oracle.com/technetwork/articles/java/fork-join-422606.html)

<sup>3</sup>[openjdk.java.net/projects/jdk7/features](http://openjdk.java.net/projects/jdk7/features)

# Parte III

## Asynchronous I/O



## Capitolo 8

# Introduzione all'Asynchronous I/O

Una delle motivazioni a vantaggio della programmazione concorrente mediante Thread è l'ottenimento di servizi non bloccanti per Input/Output. Quando l'applicazione ha necessità di effettuare un I/O, è opportuno che non si blocchi, in modo da consentire che altre operazioni non dipendenti da quell'I/O possano essere effettuate. Una tecnica di gestione dell'I/O non bloccante, utilizzata ad esempio nei microcontrollori per il physical computing, è il polling, cioè la verifica ciclica dell'accadimento di eventi su un insieme di dispositivi di input. La ciclicità del polling è gestita da un timer. L'I/O non bloccante si può realizzare mediante i Thread. La lettura di un certo dispositivo si può assegnare ad un certo Thread il quale si blocca in attesa dei dati. Gli altri Thread, che non dipendono dalla lettura del dato, possono però procedere in maniera concorrente.

Nel caso di programmazione con tecniche bloccanti, un server effettua una chiamata `wait()` (primitiva bloccante) sul socket in attesa di una nuova connessione e, una volta stabilita, esegue una chiamata `read()` anch'essa bloccante, tenendo impegnati slot temporali per lunghi tempi di inattività.

Per evitare questa situazione è stata introdotta la tecnica di programmazione asincrona (o non bloccante) che mette a disposizione delle primitive e una metodologia di programmazione diversa (rispetto a quella bloccante) che, supportata da tecniche di basso livello come il *polling* e il *Direct Memory Access* (DMA), permette di migliorare la gestione dei tempi di inattività implementando l'asincronismo nelle chiamate.

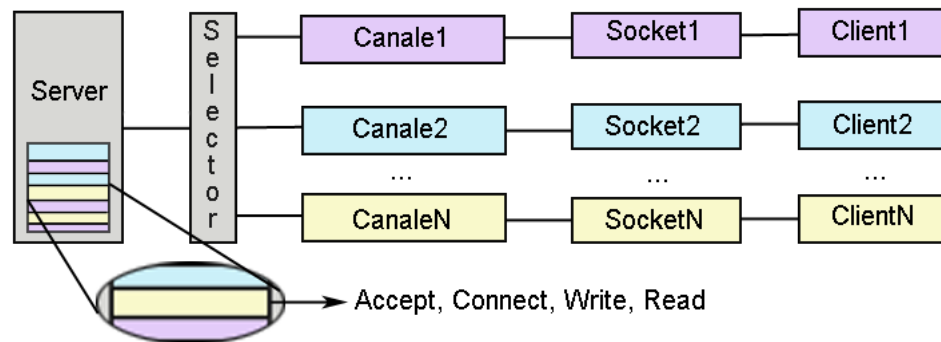


Figura 8.1: Architettura client/server del modello NIO di Java

## 8.1 Java Non-Blocking I/O, il pacchetto `java.nio`

In questa architettura, le componenti principali sono (vedi Figura 8.1):

- **Server:** L'applicazione che riceve le richieste;
- **Client:** L'insieme di richieste delle applicazioni inviate al Server;
- **Socket channel:** Il canale di comunicazione tra il Client e il Server. È identificato dall'IP del Server e il numero della porta. I dati passano attraverso il canale del socket da oggetti di tipo `Buffer` (vedi più avanti per dettagli);
- **Selector:** L'oggetto principale di tutta la tecnologia non bloccante. Monitora i socket channel registrati e serializza le richieste che il server deve soddisfare;
- **Keys:** Gli oggetti usati dal Selector per ordinare le richieste. Ogni chiave rappresenta una singola sotto-richiesta del client e contiene informazioni per identificare il client e il tipo di richiesta.

Come si può notare, le applicazioni client effettuano verso il server richieste contemporanee. Il Selector le raccoglie, crea le chiavi e le invia al server. Questo può sembrare un sistema bloccante perché le richieste vengono elaborate una alla volta, in realtà, non è così. Ogni chiave non rappresenta l'intero flusso di informazioni che un client invia ad un server, ma solo una parte (Il Selector divide i dati del client in sotto-richieste identificate dalle chiavi). Di conseguenza, se più client inviano continuamente dati al server, il Selector creerà più chiavi che saranno trattate in base ad una politica di condivisione del tempo.

Un singolo Thread viene utilizzato per gestire un numero arbitrario di socket.

Per implementare ciò, Java mette a disposizione il pacchetto NIO formato da un Selector e logica non-blocking I/O. In teoria, con il multiplexing dell'I/O, è possibile avere un unico Thread che svolga tutto il lavoro necessario in un'applicazione server. In pratica, ci sono alcuni problemi in questo approccio. Quando si utilizza un singolo Thread, non è possibile nascondere la latenza del disco I/O (Java NIO non supporta operazioni di non-blocking sui file) o sfruttare i sistemi multi-core. Un'applicazione server deve avere almeno  $2 * N$  Thread, dove  $N$  rappresenta il numero di unità funzionali (CPU) in esecuzione. Pertanto, esistono tre modi di dividere il lavoro tra i Thread.

Modello di threading:

- M dispatcher/no-worker;
- 1 dispatcher/N worker;
- M dispatcher/N worker.

In tutti e tre i casi, le connessioni in entrata sono assegnate ad un event-dispatch Thread (un Selector nella fattispecie, vedi più avanti nel testo) per tutta la durata del loro ciclo vita. Nella prima caso, gli eventi di I/O sono elaborati per intero dall'event-dispatch Thread (il dispatcher si occupa dell'invio degli eventi, della lettura, dell'esecuzione delle richieste e della scrittura). Negli altri due casi invece, la fase di elaborazione è delegata al worker Thread. Un Selector, con le relative chiavi e i canali presso lui registrati, non devono mai essere accessibili da più di un Thread per motivi legati ai meccanismi di concorrenza. Generalmente è preferibile scegliere un modello con M dispatcher e nessun worker in quanto questo problema non sussiste. L'architettura I/O dei router si sono ispirate al modello Swing event-dispatch<sup>1</sup> di Java. Nell'architettura di Swing, la JVM riceve gli eventi generati dall'interfaccia utente e vengono salvati in una coda degli eventi. Un event dispatch Thread, implementato nella classe `java.awt.EventQueue`, controlla gli eventi presenti nella coda e invia quelli in arrivo ai relativi listener registrati per quel tipo di evento. Nel router è presente un ulteriore event dispatch Thread che viene implementato nella classe `SelectorThread`. Questa classe incapsula un oggetto di tipo `Selector` e uno di tipo `Thread`. Il Thread controlla il Selector attendendo l'arrivo di eventi I/O inviandoli agli handler appropriati. Nella programmazione di socket non bloccanti, e in generale in tutti i contesti in cui è implementata la logica NIO, è molto importante l'oggetto *Buffer*, perché i canali sul nuovo socket utilizzano questi oggetti per trasferire i dati attraverso la rete.

La classe `SelectorThread` genera quattro tipi di eventi, corrispondenti alle

---

<sup>1</sup>[download.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html](http://download.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html)

operazioni definite dal pacchetto `java.nio.channels.SelectionKey`: `ACCEPT`, `CONNECT`, `READ`, `WRITE` (Figura 8.1). Gli Handler assieme alla classe `SelectorThread` ricevono gli eventi. A seconda del tipo di eventi a cui sono interessati, devono implementare una delle seguenti interfacce:

- `ConnectSelectorHandler`: Per stabilire le connessioni in uscita;
- `AcceptSelectorHandler`: Per ricevere le connessioni in entrata;
- `ReadWriteSelectorHandler`: Per leggere e scrivere i dati in una connessione.

Di seguito viene esposto un esempio di server implementato con il pacchetto `java.nio`:

Listing 8.1: Esempio Server in Java NIO

```
1 import java.io.IOException;
2 import java.net.InetSocketAddress;
3 import java.nio.ByteBuffer;
4 import java.nio.CharBuffer;
5 import java.nio.channels.SelectionKey;
6 import java.nio.channels.Selector;
7 import java.nio.channels.ServerSocketChannel;
8 import java.nio.channels.SocketChannel;
9 import java.nio.charset.Charset;
10 import java.nio.charset.CharsetDecoder;
11 import java.nio.charset.CharsetEncoder;
12 import java.util.Iterator;
13 import java.util.Set;
14
15 public class Asyn {
16     private static byte[] data = new byte[255];
17     public static void main(String[] args) {
18         int cont=0;
19         System.out.println(System.nanoTime());
20         try {
21             ServerSocketChannel server = ServerSocketChannel.open();
22             server.configureBlocking(false);
23             Charset charset = Charset.forName("ISO-8859-1");
24             CharsetDecoder decoder = charset.newDecoder();
25             CharsetEncoder encoder = charset.newEncoder();
26             ByteBuffer buffer = ByteBuffer.allocateDirect(1024); //
27                 Allocate buffers
28             CharBuffer charBuffer = CharBuffer.allocate(1024);
```

```
28     server.socket().bind(new InetSocketAddress(9000));
29     Selector selector = Selector.open();
30     server.register(selector, SelectionKey.OP_ACCEPT);
31     while (true) {
32         selector.select();
33         Set readyKeys = selector.selectedKeys();
34         Iterator iterator = readyKeys.iterator();
35         while (iterator.hasNext()) {
36             SelectionKey key = (SelectionKey) iterator.next();
37             iterator.remove();
38             if (key.isAcceptable()) {
39                 if (cont==0)
40                     System.out.println(System.nanoTime());
41                 SocketChannel client = server.accept();
42                 client.configureBlocking(false);
43                 SelectionKey key2 = client.register(selector,
44                     SelectionKey.OP_READ);
45             }
46             else if (key.isWritable()) {
47                 SocketChannel client = (SocketChannel)
48                     key.channel();
49                 ByteBuffer output = (ByteBuffer) key.attachment();
50                 if(!output.hasRemaining()) {
51                     output.rewind();
52                 }
53                 client.write(output);
54             }
55             else if (key.isReadable()) {
56                 System.out.println("Message is Readable");
57                 SocketChannel client = (SocketChannel)
58                     key.channel();
59                 client.read(buffer);
60                 buffer.flip();
61                 decoder.decode(buffer, charBuffer, false);
62                 charBuffer.flip();
63                 buffer.clear();
64                 charBuffer.clear();
65                 client.close();
66             }
67         }
68         if(cont++==1000){
69             System.out.println(System.nanoTime());
70             System.exit(0);
71         }
72     }
```

```
68         }  
69     }}  
70     catch(Exception e){e.printStackTrace(); }  
71 }  
72 }
```

Il server sopra citato, presenta un ciclo infinito il cui corpo è rappresentato da una serie di if/else che rappresentano i quattro casi citati in questo paragrafo (accept, connect, write e read). Per ogni caso, il server invoca il register a cui sottoscrive il comportamento da tenere. Ad esempio, se incontra un accept, si sottoscrive alla ricezione di una futura read.

Più in dettaglio:

- I. Dalla riga 21-28, il server instaura una nuova connessione e il buffer per i dati;
- II. Alle righe 29-30, il server istanzia un nuovo selector e vi si registra ponendo come preferenza per le operazioni desiderate ACCEPT;
- III. Dalla riga 35 il programma entra in un ciclo infinito nella quale itera in attesa di uno dei quattro casi previsti, scorrendo per mezzo di un iteratore le SelectionKeys;
- IV. Nel primo if, alla riga 38, il programma controlla se la chiave che sta osservando è di tipo ACCEPT. Nel caso lo fosse, il server si registra ad accettare il client e attende di trovarsi come prossima chiave una READ (riga 43);
- V. Alla riga 45, nel caso che la chiave non fosse una ACCEPT, viene controllato se è una WRITE e, nel caso, viene scritto al client il contenuto del buffer di output;
- VI. Nel caso che anche il controllo precedente fallisca, alla riga 53 viene controllato se la chiave è una READ.

### Critica al framework Java NIO

C'è un dibattito che vede contrapposta la libreria Java IO alla Java NIO. Nel confronto, si possono trarre alcune considerazioni in merito alle performance. Considerando un solo server e un solo client, e testando la velocità di trasferimento dei dati<sup>2</sup>, una parte sostiene che il modello di I/O bloccante è considerevolmente più veloce di circa 25%-35% rispetto alla stessa applicazione scritta con la libreria NIO non trovando un miglioramento al modello thread-per-connection della versione bloccante. In contrapposizione, però, altri affermano che il modello NIO permette un approccio multi-thread più veloce rispetto alla versione sincrona in quanto i frequenti context switch di quest'ultima incidono pesantemente sul tempo di processore e sulla memoria occupata. A vantaggio del NIO è possibile affermare che le strutture dati utilizzate sono maggiormente scalabili (`ConcurrentLinkedQueue` e `ConcurrentHashMap`) e il sincronismo tra i Thread è semplificato in quanto si deve gestire la sincronizzazione di due Thread, il Selector e il Thread Pool (rispetto alla versione thread-per-connection di Java IO).

---

<sup>2</sup>[www.mailinator.com/tymaPaulMultithreaded.pdf](http://www.mailinator.com/tymaPaulMultithreaded.pdf)

## 8.2 Scala Non-Blocking I/O, Scala Continuation

In questo paragrafo viene presa in considerazione la continuation in Scala. Per capire il funzionamento della continuation, bisogna prima introdurre il concetto di Continuation Passing Style (CPS). CPS è uno stile di programmazione nella quale il controllo del flusso è passato esplicitamente nella forma di continuation.

In un programma in CPS, invece di restituire i valori come col solito metodo diretto, viene scritta una funzione in stile continuation, passando esplicitamente come argomento una “continuazione”, cioè è una funzione che è destinata a ricevere il risultato del calcolo eseguito all'interno della funzione originaria. Allo stesso modo, quando una subroutine viene richiamata all'interno di una funzione CPS, la funzione chiamante è tenuta a fornire una procedura da invocare con i valori di ritorno della subroutine.

Esempio di funzione CPS in Scala:

Listing 8.2: Esempio codice senza CPS

```
def main(m:M):Z = {  
    val x:X = precomp(m)  
    val y:Y = subrout(m, x)  
    val z:Z = postcomp(m, x, y)  
    return z  
}  
def subrout(m:M,x:X):Y {  
    val y:Y = substuff(m, x)  
    return y  
}
```

Questo esempio di codice CPS presenta un metodo main e una subroutine. Il main riceve un parametro m di tipo M e restituisce una variabile di tipo Z. Viene inizializzata una variabile x di tipo X che assume il valore di ritorno di un ipotetica funzione pre-computazionale che riceve m in ingresso. Una variabile y di tipo Y riceve come valore il risultato della chiamata a subroutine passando alla funzione m e x. La subroutine richiama una qualche funzione substuff che elabora m e x salvandone il risultato in una variabile y di tipo Y, restituendola. Infine, l'ultimo passo che il main esegue è una chiamata ad una funzione di post-computing che riceve m, x, y salvandone il risultato in z di tipo Z restituendolo in conclusione del programma.

Listing 8.3: Esempio codice con CPS

```
def main(m:M) {  
  val x:X = precomp(m)  
  val z:Z = subrout(m, x, { postcomp(m, x, _) } )  
  return z  
}  
def subrout(m:M, x:X, subCont: (Y) => Z) {  
  val y:Y = substuff(m, x)  
  val z:Z = subCont(y)  
  return z  
}
```

Con questo codice, al posto di continuare l'esecuzione di `postcomp` subito dopo la fine di `subrout` in modo automatico come nel Listing precedente, con il codice del Listing qui sopra, si indica esplicitamente il punto da cui si dovrà riprendere, passando il pezzo di codice che invoca `postcomp` come ulteriore argomento della funzione `subrout`. Si può quindi modificare la chiamata a `subrout` in modo che, invece che in conclusione della sua fase di calcolo e relativa generazione di variabili ritorni al `main` come `y`, chiami invece `postcomp` con i relativi argomenti e restituisca come proprio valore di ritorno quello di ritorno di `postcomp` che nel `main` è rappresentato da `z`.

Con questa tecnica di programmazione, in Scala, è stata introdotta la *continuation*. Essa è caratterizzata principalmente da due funzioni:

- **Reset**: Demarca i limiti della *continuation*. All'interno del corpo del `reset`, c'è il codice CPS; Precisazione: il valore restituito da `reset` non è CPS;
- **Shift**: Indica quanto in profondità può arrivare il percorso CPS. Il corpo dello `shift` non è codice CPS, ma il valore restituito è CPS. La chiamata `shift` viene passata come argomento alla *continuation*, raccolta da tutte le chiamate all'interno del `reset`.

Tutto il codice presente tra `reset` e `shift` è CPS mentre ogni metodo che include `shift`, deve essere specificato come CPS e ogni metodo che chiama codice CPS a sua volta deve essere dichiarato come CPS fino al raggiungimento della chiamata `reset` che ripristina il contesto. Quando vengono usati `reset/shift`, il compilatore trasforma (non implicitamente, bisogna specificare al compilatore che nel codice è presente codice CPS) il codice come appena detto.

Listing 8.4: Esempio di utilizzo di reset/shift

```
...
reset
{
    shift { k: (Int=>Int) => k(7) } + 1 // risultato: 8
}
```

Questo pezzo di codice rappresenta la continuation. Il metodo shift che somma il valore 1 a 7, può essere sostituito da codice non CPS rappresentato col metodo `def f(x:Int)= x+1; f(7);`

Ci sono quindi tre tipi di dato associati a shift:

- Il tipo di argomento da passare alla continuation, che è lo stesso tipo di ritorno dello shift;
- Il tipo di ritorno dalla continuation, che è lo stesso del tipo di ritorno di tutto il codice che segue il blocco di shift. Questo è chiamato tipo di ritorno non trasformato;
- Il tipo del valore ultimo shift del blocco, che diventa il tipo del valore di ritorno della funzione di inclusione o return del blocco. Questo è chiamato il tipo di ritorno trasformato.

Nella firma di shift, compaiono tre tipi di dato, A, B e C, rispettivamente:

```
def shift[A, B, C](fun: ((A) => B) => C): A
    @scala.util.continuations.cpsParam[B,C]
```

I due tipi cpsParam nell'annotazione [B,C], rappresentano sempre il tipo non trasformato e trasforma i tipi restituiti, rispettivamente.

La firma di reset utilizza solo due tipi: il primo tipo è il tipo non trasformato del blocco di codice passato a reset che corrisponde al tipo B di shift, il secondo, è il tipo del blocco di codice trasformato, che corrisponde al tipo C di shift, ed è anche il tipo realmente restituito dal blocco di reset al suo chiamante. La firma di reset è la seguente:

```
def reset[B, C](ctx: => B @scala.util.continuations.cpsParam[B,C]):
    C
```

Di seguito viene esposto un esempio di server che utilizza la continuation di Scala

```
import java.net.InetSocketAddress
import java.nio.ByteBuffer
import java.nio.channels.{SelectionKey, Selector,
    ServerSocketChannel, SocketChannel}

val selector = Selector.open()

def asRunnable(body: => Unit) = new Runnable{ def run() { body() }}

def accept(s: ServerSocketChannel): SocketChannel @cps[Unit, Unit] =
  shift { cont =>
    selector.register(s, SelectionKey.OP_ACCEPT, asRunnable {
      val c = s.accept()
      c.configureBlocking(false)
      cont(c)
    })
  }

def read(s: SocketChannel, b: ByteBuffer): Int @cps[Unit, Unit] =
  shift { cont =>
    selector.register(s, SelectionKey.OP_READ, asRunnable {
      cont(s.read(b)) })
  }

def write(s: SocketChannel, b: ByteBuffer): Int @cps[Unit, Unit] =
  shift { cont =>
    selector.register(s, SelectionKey.OP_WRITE, asRunnable {
      cont(s.write(b)) })
  }

val server = ServerSocketChannel.open()
server.socket().bind(new InetSocketAddress(12345))
server.configureBlocking(false)

reset {
  while (true) {
    val socket = accept(server)
    reset {
      val buffer = ByteBuffer.allocateDirect(1)
      def iter(): Unit @cps[Unit, Unit] = {
        buffer.clear()
        val count = read(socket, buffer)
      }
    }
  }
}
```

```
        if (count < 1)
            socket.close()
        else {
            buffer.flip()
            write(socket, buffer)
            iter()
        }
    }
    iter()
}}
}

while (true) {
    selector.select()
    val l = selector.selectedKeys().toList
    selector.selectedKeys().clear()
    for (k <- l) {
        k.interestOps(0)
        k.attachment().asInstanceOf[Runnable].run()
    }
}
```

Come si evince dal codice, il server cicla all'infinito in una porzione di codice entro la quale vi è innestata la funzione `reset` che aspetta in modo asincrono le richieste. Il corpo di ogni richiesta rappresenta lo shift nella quale si va a specificare il comportamento del server agendo sul register.

## 8.3 Un approccio ibrido, Java NIO e Scala Delimited Continuation

Una semplice implementazione Java di un server potrebbe archiviare i propri stati per-client in un Thread per ogni client attivo e bloccarsi in attesa di una richiesta su ogni Thread [24]. Lo scheduler di Java si prende poi l'onere di notificare quando l'input è disponibile presso una di quelle letture bloccanti e continuare l'esecuzione sul Thread associato. Come visto nel paragrafo 8.1, è possibile, per mezzo del pacchetto NIO di Java, rendere queste chiamate asincrone ed evitare quindi il blocco. Tuttavia, i Thread sono alquanto dispendiosi di memoria e ne consumano più del necessario. Senza però perdere i vantaggi dell'architettura, è possibile sfruttare l'implementazione più compatta, che salva manualmente il contesto, della continuation in Scala vista nel paragrafo 8.2 ed è quindi in grado di gestire molti più client della versione NIO, in quanto, combinare il non-blocking I/O di JAVA con la delimited continuation di Scala per implementare un server multi-client non richiede un thread dedicato per ogni client attivo.

Come visto in precedenza, il modello di I/O di JAVA NIO si basa sui Channels e sui Selectors. Un Channel rappresenta la connessione ai file, socket, ecc... mentre un Selector permette invece di raccogliere insieme una serie di Channel e aspettare finché uno di questi sia pronto per l'I/O.

Per far ciò, è necessario registrare (*register*) il Channel (*SelectableChannel*) con il Selector. Quando si registra un canale, si deve esplicitamente specificare quali sono le operazioni di interesse, come READ o WRITE. Queste operazioni sono rappresentate da bit che possono essere combinati insieme da una OR per creare un valore che indichi più operazioni.

Registrare un canale con un selettore crea un *SelectionKey*. Quando successivamente si esegue il metodo di selezione del selettore, viene restituito un insieme di *SelectionKeys* da cui è possibile estrarre le informazioni passate al register delle chiamate quando il *SelectionKey* è stato creato.

### 8.3.1 NioSelector

La chiamata *register* di *java.nio.channels.SelectableChannel* include un parametro che permette all'applicazione di allegare un oggetto arbitrario a quella registrazione e di recuperarlo in un secondo momento quando il canale associato diventa selezionato. Nell'esempio seguente, viene applicata una funzione di callback quando il canale diventa selezionato.

Listing 8.5: Codice Scala di NioSelector che usa Java NIO

```
1 import scala.collection.JavaConversions
2 import java.nio.channels.SelectableChannel
3 import java.nio.channels.SelectionKey
4 import java.nio.channels.spi.SelectorProvider
5
6 class NioSelector extends Runnable
7 {
8     val selector = SelectorProvider.provider.openSelector()
9     def register(channel:SelectableChannel, op:Int, body: => Unit)
10     {
11         val callback:Function0[Unit] = { () => { body }}
12         channel.register(selector, op, callback)
13     }
14     def run()
15     {
16         selectLoop(true)
17     }
18     def selectLoop(continueProcessing: => Boolean)
19     {
20         while (continueProcessing)
21         {
22             selectOnce(0)
23         }
24     }
25     def selectOnce(timeout:Long)
26     {
27         selector.select(timeout)
28         val jKeys:java.util.Set[SelectionKey] =
29             selector.selectedKeys
30         val keys = JavaConversions.asScalaSet(jKeys).toList
31         selector.selectedKeys.clear()
32         keys foreach { _.interestOps(0) }
33         val callbacks = keys map {
34             _.attachment.asInstanceOf[()=>Unit] }
35         executeCallbacks(callbacks) //Execute callbacks for all
36                                     selected keys
37     }
38     def executeCallbacks(callbacks:List[()=>Unit])
39     {
40         callbacks foreach { _() }
41     }
42 }
```

Come detto in precedenza, la funzione *register* ha come argomenti il canale selezionato e una lista di operazioni permesse. Nella funzione, viene definito il parametro callback come una funzione continuation come illustrato nella sezione 8.2 e tale funzione viene passata come parametro alla funzione *register* della classe Java NIO.

Il metodo *SelectOnce* attende che uno o più dei canali registrati sia selezionato, o finché sia trascorso un tempo specificato se non ci sono canali selezionati durante quel periodo. La chiamata *selectedKeys* di *java.nio.channels.Selector* restituisce un oggetto *java.util.Set* contenente le informazioni in merito a quali canali siano selezionati. Prima di elaborare ogni canale, bisogna cancellare lo stato attuale in modo da ri-registrarsi su quel canale durante quella elaborazione. Viene effettuata una copia dello stato di tutti i canali selezionati e poi si cancella quegli stati prima di elaborare qualsiasi canale.

Il metodo *executeCallbacks* è responsabile dell'esecuzione della callback per il canale selezionato. Ipotezzando che almeno una di queste callback possa richiedere un tempo relativamente lungo per completarsi, allora in un server reale si potrebbe pensare di inserire tutte queste callback in un Thread Pool o farle creare all'esterno da degli attori in modo che si possa evitare di avere un'unica lunga callback in esecuzione che blocca il resto dei canali. Nel codice precedente, viene proposta una semplice implementazione che esegue ogni callback sequenzialmente nello stesso thread.

Il metodo *selectLoop* è stato inserito per continuare a invocare *selectOnce* finché non si è pronti ad interrompere l'esecuzione.

### 8.3.2 NioListener

Listing 8.6: Codice Scala di NioListener che usa Java NIO

```
1 import java.net.{InetAddress, InetSocketAddress}
2 import java.nio.channels.ServerSocketChannel
3 import java.nio.channels.SelectionKey
4 import java.nio.channels.SocketChannel
5 import scala.util.continuations._
6
7 class NioListener(selector:NioSelector, hostAddr:InetAddress,
8   port:Int)
9 {
10   val serverChannel = ServerSocketChannel.open()
11   serverChannel.configureBlocking(false);
12   val isa = new InetSocketAddress(hostAddr, port)
13   serverChannel.socket.bind(isa)
```

```
13     def start(continueListening: =>Boolean):Unit =
14     {
15         reset
16         {
17             while(continueListening)
18             {
19                 val socket = accept()
20                 NioConnection.newConnection(selector, socket)
21             }
22         }
23     }
24     private def accept():SocketChannel @suspendable =
25     {
26         shift
27         { k => selector.register(serverChannel,
28                                 SelectionKey.OP_ACCEPT, {
29                 val conn = serverChannel.accept()
30                 conn.configureBlocking(false)
31                 k(conn)
32             })
33     }
34 }
```

La classe *NioSelector* fornisce il main del server, ma per comprendere come gli oggetti ad esso collegati inizino a generare eventi, bisogna prima comprendere quali passi sono necessari affinché il flusso di dati in lettura passi per il socket in ascolto del server. Il server opera nei seguenti tre passi:

- Rimane in ascolto per una nuova connessione sul socket associato;
- Quando viene creata una nuova connessione, essa viene inizializzata e il server rimane in attesa dei dati in ingresso su quella connessione;
- Quando i dati in ingresso vengono ricevuti, essi vengono elaborati;
- Eventualmente, la connessione viene chiusa.

La classe *NioListener* si occupa di creare socket su cui il server rimane in ascolto e accettare nuove connessioni. Dal momento che verrà passato questo socket al selector, viene creato un *ServerSocketChannel* anziché un socket normale. Non si vuole che sul socket ci siano chiamate bloccanti, quindi viene impostato *configureBlocking(false)*.

Dalla riga 9 alla 12 del codice di implementazione della classe, viene creato

il socket che dovrà essere registrato presso il selector (riga 27-31). In quella chiamata, verrà detto al selector che si è interessati a ricevere, sul canale del socket, operazioni di ACCEPT. Il terzo argomento del metodo register è la callback, ovvero il punto di ingresso della continuation.

Il codice bloccante è generalmente più semplice da scrivere e capire rispetto al codice asincrono (non bloccante) ma usando la continuation, viene salvato lo stato corrente del codice, salvandolo esplicitamente per l'esecuzione futura per poi riprendere l'esecuzione del Thread in un punto diverso. Da notare che il codice della delimited continuation è bloccante ma è il Thread a non esser bloccato.

Piuttosto che una creazione e un passaggio esplicito nella callback dello stato, il codice è stato organizzato in modo tale che la callback, passata al metodo register, includa una chiamata alla continuation in modo tale che, una chiamata alla callback esegua la continuation e continui il codice da dove era stato sospeso. Il metodo accept è stato quindi implementato con la chiamata bloccante *ServerSocket.accept* ma che utilizza la continuation appena descritta. Il metodo accept() viene invocato da un blocco di codice interno ad un reset (riga 19) che delimita la continuation e determina la porzione di codice da salvare (interna al blocco reset) e il punto da cui riprendere (subito dopo il reset) quando nel Thread vengono accettate chiamate bloccanti.

Quando viene invocato il metodo start, si entra nel blocco reset e viene invocato accept() che registra l'intenzione di voler operazioni ACCEPT, viene salvata la continuation come parte della callback nel metodo register e restituisce immediatamente il controllo alla porzione di codice successiva al blocco reset (il metodo start si conclude rapidamente). Quando arriva una nuova connessione, il codice in NioSelector invoca la callback dal suo metodo *executeCallbacks*. La callback accetta la nuova connessione e la imposta come non bloccante, quindi esegue la continuation che "ritornerà" dal blocco del metodo accept nel loop del metodo start. Il codice processa la nuova connessione e, cicla e invoca accept nuovamente, che registrerà l'intenzione di voler operazioni ACCEPT sul socket e restituisce l'esecuzione al chiamante della continuation, che restituirà il controllo al metodo *executeCallback* della classe *NioSelector*.

### 8.3.3 NioConnection

Listing 8.7: Codice Scala di NioConnection che usa Java NIO

```
1 import java.nio.ByteBuffer
2 import java.nio.channels.SelectionKey
3 import java.nio.channels.SocketChannel
4 import scala.util.continuations._
5
6 object NioConnection
7 {
8     def newConnection(selector:NioSelector, socket:SocketChannel)
9     {
10         val conn = new NioConnection(selector,socket)
11         conn.start()
12     }
13 }
14
15 class NioConnection(selector:NioSelector, socket:SocketChannel)
16 {
17     private val buffer = ByteBuffer.allocateDirect(2000)
18     def start():Unit =
19     {
20         reset
21         {
22             while(socket.isOpen)
23                 readWait
24         }
25     }
26     private def readWait =
27     {
28         buffer.clear()
29         val count = read(buffer)
30         if(count < 1)
31             socket.close()
32         else
33             readAction(buffer)
34     }
35     private def read(b:ByteBuffer):Int @suspendable =
36     {
37         if(!socket.isOpen)
38             -1 //indica EOF
39         else
40             shift
```

```
41         { k => selector.register(socket, SelectionKey.OP_READ,
42             {
43                 val n = socket.read(b)
44                 k(n)
45             })
46     }
47     private def readAction(b:ByteBuffer)
48     {
49         b.flip()
50         socket.write(b)
51         b.clear()
52     }
53 }
```

La classe *NioConnection* è la responsabile della creazione della connessione, della registrazione con il selettore, della lettura dei dati di input quando disponibili e del processing degli stessi. La parte di processing implementata in questa applicazione è un echo dei dati inviati dal client. Per semplicità, nel lato della scrittura, non viene applicato nessun meccanismo di NIO o di continuation ma solo la chiamata del metodo *write* per scrivere direttamente sul socket i dati.

Come nella classe *NioListener* è stato ridefinito il metodo *accept*, in *NioConnection* viene ridefinito il metodo *read* in cui è stata inserita la logica della continuation. Come per *NioListener*, il metodo *start* che invoca il metodo *read* contiene il blocco *reset* che delimita la continuation.

Quando viene invocato il metodo *start*, il programma entra nel blocco *reset* e, attraverso la chiamata *readWait*, viene invocato *read* che registra l'interesse per le operazioni READ, salva la continuation come parte della callback nel metodo *register* e immediatamente restituisce il controllo alla porzione di codice successiva al blocco *reset* (come per *NioListener*, anche l'esecuzione del *start* del *NioConnection* è un'operazione che termina velocemente). Quando i dati sono disponibili per la lettura dal socket, il metodo *select* della classe *NioSelector* richiama la callback dal suo metodo *executeCallbacks*. La callback legge i dati dalla connessione ed esegue la continuation, la quale "ritornerà" dal metodo *read* invocato all'interno del metodo *readWait*. Il programma elabora i dati in ingresso nel metodo *readAction* (il quale rimanda indietro al client gli stessi dati), quindi, se il socket è ancora aperto, cicla e finisce per chiamare nuovamente il metodo *read* che registrerà sul socket l'interesse per operazioni READ e restituirà l'esecuzione al chiamante della continuation che restituirà il controllo al metodo *executeCallbacks* della

classe NioSelector.

### 8.3.4 NioServer

Listing 8.8: Codice Scala di NioServer che usa Java NIO

```
1 import java.net.InetAddress
2
3 object NioServer
4 {
5     def main(args:Array[String])
6     {
7         val hostAddr:InetAddress = null //localhost
8         val port = 1234
9         val server = new NioServer(hostAddr,port)
10        server.start()
11    }
12 }
13
14 class NioServer(hostAddr:InetAddress, port:Int)
15 {
16     val selector = new NioSelector()
17     val listener = new NioListener(selector, hostAddr, port)
18     def start()
19     {
20         listener.start(true)
21         selector.run()
22     }
23 }
```

La classe NioServer è il main che testa i moduli sopra esposti. Come detto nella sezione 8.2, bisogna esplicitamente informare il compilatore che nel codice è presente la continuation. Per far ciò, i comandi da inviare sono i seguenti:

```
scalac -P:continuations:enable *.scala
scala NioServer
```

Dopo l'avvio del server, è possibile collegarsi via telnet all'indirizzo associato (telnet localhost 1234) e scrivere sul buffer i dati per il server.

Di seguito viene presentato il diagramma UML sequenziale del flusso di informazioni scambiato dalle quattro classi appena descritte

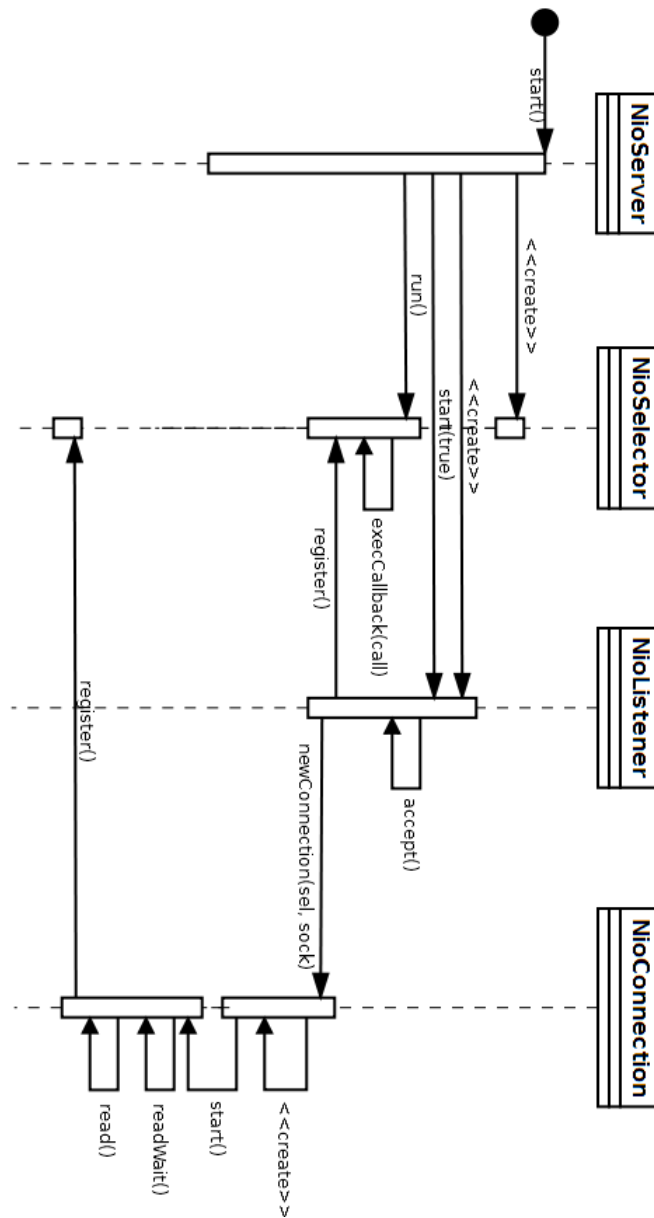


Figura 8.2: diagramma sequenziale UML



## Parte IV

### Netty



## Capitolo 9

# L'architettura Netty

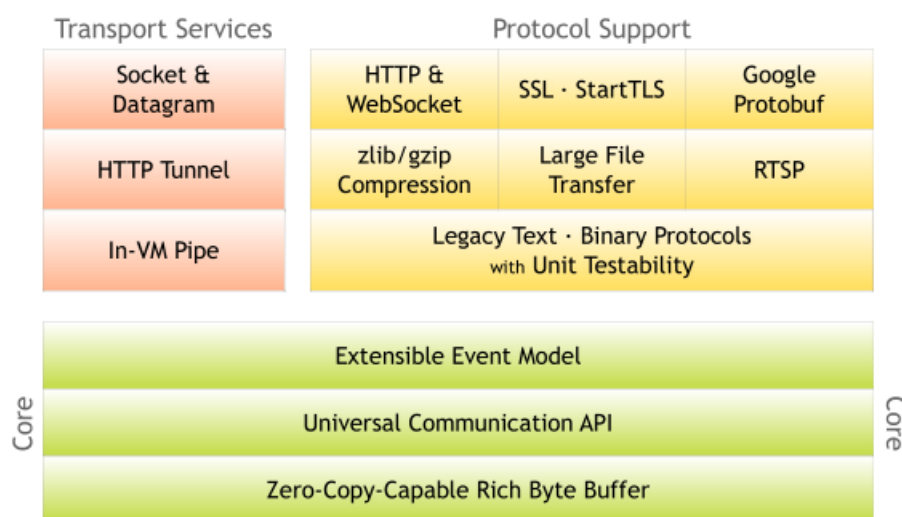


Figura 9.1: Architettura del framework Netty di Java

Il progetto Netty<sup>1</sup> è un tentativo di fornire un framework applicativo di rete asincrono basato su eventi e un tool per il rapido sviluppo di un manutenibile e performante protocollo client/server. In altre parole, Netty è un framework NIO client/server che permette lo sviluppo di applicazioni di rete come protocolli server e client.

In figura 9.1 viene presentata l'architettura del framework Netty, di seguito descritto.

Le principali differenze che si possono citare tra Netty e il NIO asincrono della libreria Java sono:

---

<sup>1</sup>[www.jboss.org/netty](http://www.jboss.org/netty)

- Una migliore struttura **data buffer** (*ChannelBuffer*) in quanto Netty usa una propria implementazione di *ByteBuffer* della libreria NIO per rappresentare le sequenze di byte. Questo approccio ha dei vantaggi significativi rispetto all'utilizzo di *ByteBuffer* in quanto la struttura è stata scritta correggendo i problemi che presentava *ByteBuffer*:
  - *ByteBuffer* ha un approccio di accesso ai dati tipo RAM;
  - La dimensione del buffer non è espandibile; non presenta meccanismi circolari di accesso;
  - Non ci sono code dove accumulare dati in presenza di grandi stream di dati;
  - Presenta un algoritmo lookahead sincrono;
  - È una struttura di basso livello dove il programmatore deve specificare manualmente lo svuotamento, la lettura e la scrittura e l'implementazione di meccanismi di controllo per evitare overflow;

*ChannelBuffer* permette di:

- Definire il tipo di buffer desiderato;
  - Il buffer è espandibile dinamicamente;
  - Non è necessario invocare il metodo *flip()* (in *ByteBuffer* è necessario invocare *flip()* per convertire da un buffer riempito da un I/O fisico ad un buffer svuotato attraverso il metodo *get()* e viceversa da un buffer riempito con il metodo *put()* ad un buffer svuotato per un I/O fisico);
  - Velocizzare i trasferimenti.
- Mentre in NIO, per mancanza di un super tipo comune, la libreria mette a disposizione tipi e metodi diversi per diversi tipi di dato trasportato (es: *java.net.Socket* e *java.net.DatagramSocket*) rendendo difficoltoso il porting di una applicazione, Netty fornisce un'interfaccia comune per l'I/O asincrono chiamata **Channel**.
  - Un **modello ad eventi estensibile** costituito da una rigida gerarchia che permette di modificare il codice esistente senza dover riscrivere l'applicazione consentendo l'aggiunta di eventi personalizzati o l'estensione degli stessi.

I Web Service, o più in generale gli oggetti distribuiti, hanno una struttura di base comune costituita dai seguenti passi (passi che differiscono per natura e costo in base al servizio) in cui ogni handler può essere avviato sul suo Thread:

- Request
  - Richiesta di lettura (**Read**);
  - Richiesta di decodifica (**Decode**);
- Service
  - Servizio di processing (**Process**);
- Reply
  - Risposta di codifica (**Encode**);
  - Risposta di invio (**Send**);

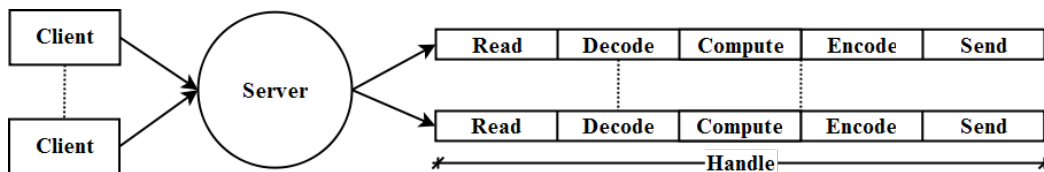


Figura 9.2: Design classico dei servizi

Questo pattern soffre di alcuni problemi come il degrado della scalabilità e delle risorse (CPU, memoria, banda, latenza, ecc...) all'aumentare del numero di client interfacciati con il server. Per evitare ciò, Netty ha applicato una strategia Divide-and-Conquer per migliorare i requisiti di scalabilità.

La strategia consta in una suddivisione dei processi in task più piccoli (task che esegue un'azione che risulta essere non bloccante) che vengono eseguiti solo quando sono abilitati dividendo così le fasi di Non-blocking (Read, Send) da quelle di Dispatch (Decode, Process, Encode).

Per questa strategia si possono applicare tre modelli:

- **Single Thread Reactor;**
- **Worker Thread Pool;**
- **Multiple Reactor Threads.**

## 9.1 Single Thread Pattern - Reactor

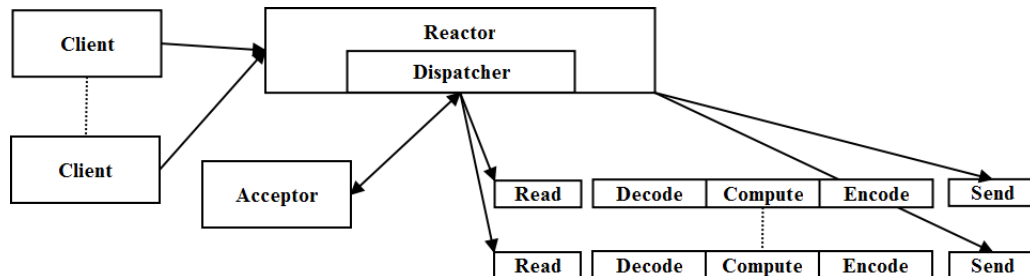


Figura 9.3: Design con Reactor a Single Thread

Il Reactor (figura 9.3) risponde ad un evento di I/O di un client inviando la richiesta (dispatching) all'handler appropriato. Gli handler eseguono in un singolo Thread i task non bloccanti gestendo gli eventi sugli handler appropriati.

Con il supporto di *java.nio*, vengono costruite le classi che sono comuni anche con gli altri pattern:

- **Channel** che si occupa di instaurare una connessione con i file, socket, ecc... che supportano il meccanismo di lettura non bloccante;
- **Buffer** che è un oggetto che assomiglia ad un array che può essere direttamente letto e scritto dai Channel;
- **Selector** tiene traccia di quali Channel ha eventi di I/O;
- **SelectionKeys** mantiene lo stato degli eventi I/O e le loro associazioni.

Esempio di implementazione Java del pattern Reactor:

Listing 9.1: Esempio di Pattern Reactor in Java

```

class Reactor implements Runnable
{
    final Selector selector;
    final ServerSocketChannel serverSocket;

    Reactor(int port) throws IOException
    {
        selector = Selector.open();
        serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(port));
    }
}

```

```
serverSocket.configureBlocking(false);
SelectionKey sk = serverSocket.register(selector,
    SelectionKey.OP_ACCEPT);
sk.attach(new Acceptor());
}

public void run()
{
    try
    {
        while(!Thread.interrupted())
        {
            selector.select();
            Set selected = selector.selectedKeys();
            Iterator it = selected.iterator();
            while(it.hasNext())
                dispatch((SelectionKey)it.next());
            selected.clear();
        }
    } catch(IOException ex) { ... }
}

void dispatch(SelectionKey k)
{
    Runnable r = (Runnable)(k.attachment());
    if(r != null)
        r.run();
}

class Acceptor implements Runnable //classe interna
{
    public void run()
    {
        try
        {
            SocketChannel c = serverSocket.accept();
            if(c != null)
                new Handler(selector, c);
        } catch(IOException ex) { ... }
    }
}
}
```

Listing 9.2: Esempio di Handler per Pattern Reactor in Java

```
final class Handler implements Runnable
{
    final SocketChannel socket;
    final SelectionKey sk;
    ByteBuffer input = ByteBuffer.allocate(MAXIN);
    ByteBuffer output = ByteBuffer.allocate(MAXOUT);
    static final int READING = 0, SENDING = 1;
    int state = READING;
    Handler(Selector sel, SocketChannel c) throws IOException
    {
        socket = c; c.configureBlocking(false);
        sk = socket.register(sel, 0); sk.attach(this);
        sk.interestOps(SelectionKey.OP_READ);
        sel.wakeup();
    }
    boolean inputIsComplete() { ... }
    boolean outputIsComplete() { ... }
    void process() { ... }
    public void run()
    {
        try
        {
            if(state == READING) read();
            else if (state == SENDING) send();
        } catch(IOException ex) { ... }
    }
    void read() throws IOException
    {
        socket.read(input);
        if(inputIsComplete())
        {
            process();
            state = SENDING;
            sk.interestOps(SelectionKey.OP_WRITE);
        }
    }
    void send() throws IOException
    {
        socket.write(output);
        if(outputIsComplete()) sk.cancel();
    }
}
```

## 9.2 Multi-Threaded Pattern - Worker Thread Pool

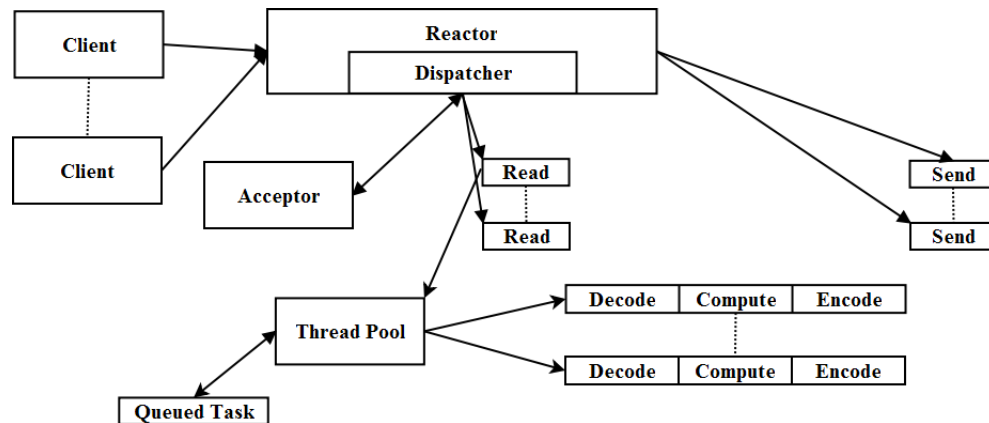


Figura 9.4: Design Multi-Thread con Worker Thread

L'idea è quella di utilizzare come base l'approccio precedente unita alla strategia del multithreading per migliorare la scalabilità. Vengono infatti creati due tipi di Thread:

- **Worker Thread** in modo che il Reactor attivi rapidamente gli handler (l'elaborazione degli handler rallentano il reactor) con il vantaggio di poter assegnare il processing che non è I/O ad altri Thread per velocizzare il Thread Reactor;
  - **Thread Pool** associato ai Worker Thread per ottimizzare e controllare i task che vengono abilitati, attivati o invocati dal task successivo. L'accoppiamento con un Thread Pool velocizza l'esecuzione anche se introduce una serie di problematiche legate all'accesso esclusivo a risorse condivise come il problema del race condition<sup>2</sup>;
- **Multiple Reactor Thread** per distribuire il carico su altri reactor (Bilanciamento del carico per soddisfare i picchi di CPU e I/O) dal momento che possono saturarsi durante l'I/O.

Di seguito viene presentato un esempio di codice scritto in Java. Il codice della classe Reactor rimane invariato, cambia solo l'handler per l'introduzione del Thread Pool:

<sup>2</sup>vedi paragrafo 3.1

Listing 9.3: Esempio di Handler per Worker Thread Pool in Java

```
class Handler implements Runnable
{
    static PooledExecutor pool = new PooledExecutor(...);
    static final int PROCESSING = 3;
    ...
    synchronized void read()
    {
        ...
        socket.read(input);
        if(inputIsComplete())
        {
            state = PROCESSING;
            pool.execute(new Processer());
        }
    }
    synchronized void processAndHandOff()
    {
        process();
        state = SENDING;
        sk.interest(SelectionKey.OP_WRITE);
    }
    class Processer implements Runnable
    {
        public void run() { processAndHandOff(); }
    }
}
```

## 9.3 Multi-Threaded Pattern - Multiple Reactor Thread

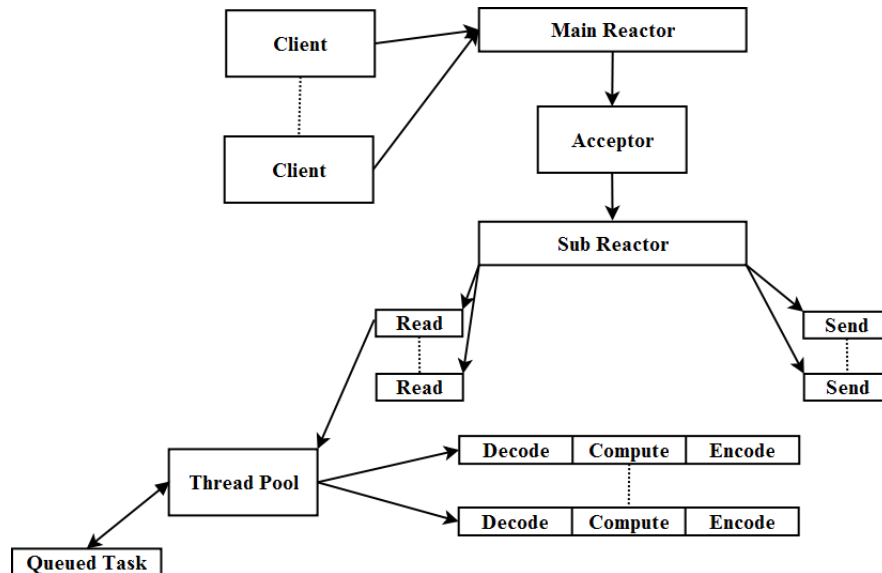


Figura 9.5: Design Multi-Thread con Multiple Reactor Thread

Per questo modello si sfruttano i vantaggi del precedente inserendo un Pool di Reactor utilizzato per incrementare il rendimento durante i picchi di CPU e I/O. Può essere costruito staticamente o dinamicamente ed ognuno ha il proprio Selector, Thread e un ciclo di dispatching, con un Acceptor principale che distribuisce i task agli altri Reactor.

Di seguito viene esposto un esempio di codice Java che implementa il modello; rispetto alla versione precedente, cambia solo la classe interna Acceptor della classe Reactor, mantenendo invariata la restante logica del Reactor e degli Handler.

Listing 9.4: Esempio di Multiple Reactor Thread in Java

```

class Reactor implements Runnable
{
    final Selector[] selectors;
    int next = 0;
    ...
    class Acceptor implements Runnable //classe interna
    {

```

```

        public synchronized void run()
        {
            try
            {
                Socket connection = serverSocket.accept();
                if(connection != null)
                    new Handler(selectors[next], connection);
                if (++next == selectors.length) next = 0;
            } catch(IOException ex) { ... }
        }
    }
}

```

## 9.4 Esempi di applicazione in Netty

Per contrapposizione all'esempio del paragrafo 8.1 verrà esposto un Echo Server (con annesso client).

Listing 9.5: Esempio di Echo Client in Netty

```

1 import java.net.InetSocketAddress;
2 import java.util.concurrent.Executors;
3 import org.jboss.netty.bootstrap.ClientBootstrap;
4 import org.jboss.netty.channel.ChannelFuture;
5 import org.jboss.netty.channel.ChannelPipeline;
6 import org.jboss.netty.channel.ChannelPipelineFactory;
7 import org.jboss.netty.channel.Channels;
8 import
    org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;
9
10 public class EchoClient
11 {
12     public static void main(String[] args) throws Exception
13     {
14         if(args.length < 2 || args.length > 3)
15         {
16             System.err.println(
17                 "Usage: " + EchoClient.class.getSimpleName() +
18                 " <host> <port> [<first message size>]");
19             return;
20         }
21         final String host = args[0];

```

```
22     final int port = Integer.parseInt(args[1]);
23     final int firstMessageSize;
24     if(args.length == 3)
25         firstMessageSize = Integer.parseInt(args[2]);
26     else
27         firstMessageSize = 256;
28     ClientBootstrap bootstrap = new ClientBootstrap(
29         new NioClientSocketChannelFactory(
30             Executors.newCachedThreadPool(),
31             Executors.newCachedThreadPool()));
32     bootstrap.setPipelineFactory(new ChannelPipelineFactory()
33     {
34         public ChannelPipeline getPipeline() throws Exception
35         {
36             return Channels.pipeline(new
37                 EchoClientHandler(firstMessageSize));
38         }
39     });
40     ChannelFuture future = bootstrap.connect(new
41         InetSocketAddress(host, port));
42     future.getChannel().getCloseFuture().awaitUninterruptibly();
43     bootstrap.releaseExternalResources();
44 }
```

Il client invia un messaggio quando una connessione viene aperta e rispedisce indietro ogni dato ricevuto al server. In poche parole, il client avvia un traffico ping-pong tra il client e l'eco server con l'invio del primo messaggio al server. Dopo una prima fase di fetch dei parametri quali indirizzo e porta (righe 14-27), alla riga 28 inizia la configurazione del client creando un nuovo Channel Factory sul socket del client e impostando un nuovo Pool per il gestore dei Reactor (riga 30) e uno per i Worker Thread (riga 31). Dalla riga 32 a 38 viene creata la pipeline su cui viene depositato l'handler del client (riga 36) (per i dettagli dell'handler si veda più avanti). Nella riga 39 inizia il tentativo di connessione all'indirizzo/porta precedentemente specificato; alla riga 40 si attende che la connessione venga chiusa o che il tentativo di connessione fallisca; infine alla riga 41 vengono rilasciate le risorse (chiusura del Thread Poll) per uscire dal programma.

Di seguito viene presentato il codice dell'handler del client:

---

Listing 9.6: Esempio di Handler per Echo Client in Netty

---

```
1 import java.util.concurrent.atomic.AtomicLong;
2 import java.util.logging.Level;
3 import java.util.logging.Logger;
4 import org.jboss.netty.buffer.ChannelBuffer;
5 import org.jboss.netty.buffer.ChannelBuffers;
6 import org.jboss.netty.channel.ChannelHandlerContext;
7 import org.jboss.netty.channel.ChannelStateEvent;
8 import org.jboss.netty.channel.ExceptionEvent;
9 import org.jboss.netty.channel.MessageEvent;
10 import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
11
12 public class EchoClientHandler extends SimpleChannelUpstreamHandler
13 {
14     private static final Logger logger =
15         Logger.getLogger(EchoClientHandler.class.getName());
16     private final ChannelBuffer firstMessage;
17     private final AtomicLong transferredBytes = new AtomicLong();
18     public EchoClientHandler(int firstMessageSize)
19     {
20         if(firstMessageSize <= 0)
21         {
22             throw new IllegalArgumentException("firstMessageSize:
23                 " + firstMessageSize);
24         }
25         firstMessage = ChannelBuffers.buffer(firstMessageSize);
26         for (int i = 0; i < firstMessage.capacity(); i++)
27         {
28             firstMessage.writeByte((byte) i);
29         }
30     }
31     public long getTransferredBytes()
32     {
33         return transferredBytes.get();
34     }
35     @Override
36     public void channelConnected(ChannelHandlerContext ctx,
37         ChannelStateEvent e)
38     {
39         e.getChannel().write(firstMessage);
40     }
41     @Override
42     public void messageReceived(ChannelHandlerContext ctx,
43         MessageEvent e)
```

```
40     {
41         transferredBytes.addAndGet(((ChannelBuffer)
42             e.getMessage()).readableBytes());
43         e.getChannel().write(e.getMessage());
44     }
45     @Override
46     public void exceptionCaught(ChannelHandlerContext ctx,
47         ExceptionEvent e)
48     {
49         logger.log(Level.WARNING, "Unexpected exception from
50             downstream.", e.getCause());
51         e.getChannel().close();
52     }
53 }
```

L'Handler dell'echo client viene avviato quando inizia il traffico tra il client e il server, ovvero quando il client invia il primo messaggio al server. Dalla riga 17 inizia la creazione dell'handler del client, dove il costruttore riceve come parametro la dimensione del primo messaggio ricevuto in modo da inizializzare correttamente il buffer. Alla riga 34 viene descritto il comportamento del client alla connessione, cioè invia un messaggio al server il quale non risponderà lì perché la capacità di quel primo messaggio è 0. Alla riga 39 viene descritto il comportamento alla ricezione del messaggio cioè rispedisce indietro il messaggio ricevuto al peer remoto. Infine, alla riga 45, in caso di errore, viene chiusa la connessione e stampato il log con l'errore.

Di seguito viene proposto un esempio di implementazione della classe Echo Server:

Listing 9.7: Esempio di Echo Client in Netty

```
1  import java.net.InetSocketAddress;
2  import java.util.concurrent.Executors;
3  import org.jboss.netty.bootstrap.ServerBootstrap;
4  import org.jboss.netty.channel.ChannelPipeline;
5  import org.jboss.netty.channel.ChannelPipelineFactory;
6  import org.jboss.netty.channel.Channels;
7  import
8      org.jboss.netty.channel.socket.nio.NioServerSocketChannelFactory;
9
10 public class EchoServer
11 {
12     public static void main(String[] args) throws Exception
```

```
12     {
13         ServerBootstrap bootstrap = new ServerBootstrap(
14             new NioServerSocketChannelFactory(
15                 Executors.newCachedThreadPool(),
16                 Executors.newCachedThreadPool());
17         bootstrap.setPipelineFactory(new ChannelPipelineFactory()
18             {
19             public ChannelPipeline getPipeline() throws Exception
20             {
21                 return Channels.pipeline(new EchoServerHandler());
22             }
23             });
24         bootstrap.bind(new InetSocketAddress(8080));
25     }
26 }
```

Il comportamento del server è quello di rispedire indietro gli stessi dati ricevuti da un client remoto. Analizzando il codice, dalla riga 13, come per il client, avviene la fase di configurazione del server e la creazione di due Thread per la gestione dei Reactor e dei Worker rispettivamente. Dalla riga 17 avviene la configurazione della pipeline e la creazione dell'handler sul canale, mentre alla riga 24 avviene il bind del server alla porta 8080.

Infine, viene ora proposta l'implementazione dell'handler del server:

Listing 9.8: Esempio di Echo Client in Netty

```
1 import java.util.concurrent.atomic.AtomicLong;
2 import java.util.logging.Level;
3 import java.util.logging.Logger;
4 import org.jboss.netty.buffer.ChannelBuffer;
5 import org.jboss.netty.channel.ChannelHandlerContext;
6 import org.jboss.netty.channel.ExceptionEvent;
7 import org.jboss.netty.channel.MessageEvent;
8 import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
9
10 public class EchoServerHandler extends SimpleChannelUpstreamHandler
11 {
12     private static final Logger logger =
13         Logger.getLogger(EchoServerHandler.class.getName());
14     private final AtomicLong transferredBytes = new AtomicLong();
15     public long getTransferredBytes()
16     {
```

```
16         return transferredBytes.get();
17     }
18     @Override
19     public void messageReceived(ChannelHandlerContext ctx,
20         MessageEvent e)
21     {
22         transferredBytes.addAndGet(((ChannelBuffer)
23             e.getMessage()).readableBytes());
24         e.getChannel().write(e.getMessage());
25     }
26     @Override
27     public void exceptionCaught(ChannelHandlerContext ctx,
28         ExceptionEvent e)
29     {
30         logger.log(Level.WARNING, "Unexpected exception from
31             downstream.", e.getCause());
32         e.getChannel().close();
33     }
34 }
```

Alla riga 19, in caso di ricezione del messaggio, il server lo preleva dal canale per poi depositarlo nuovamente; in caso di errore (riga 25) il server stampa l'errore segnalato e termina la connessione.



## Parte V

# Prototipo di Orchestratore in Scala



# Capitolo 10

## L'orchestratore di servizi basato su eventi in Scala

In questa sezione, si vuole descrivere l'organizzazione lato server di una piattaforma che supporti l'esecuzione di Mashup<sup>1</sup>. Questo tipo di piattaforme dovrebbero supportare un numero elevato di esecuzioni di Mashup concorrenti quindi, è importante sviluppare un sistema il più efficiente possibile sfruttando la potenza di elaborazione delle architetture multi-processore. Per fare ciò, in questo capitolo viene descritta un'implementazione che sfrutta Scala, il linguaggio descritto nel capitolo 4, che fornisce un modello efficiente di concorrenza basata sul modello degli attori che è adatto ad essere eseguito su sistemi multi-processore. La piattaforma descritta in questa sezione è stata precedentemente sviluppata in Java. La trattazione vuole anche mostrare la portabilità di Scala, grazie alla compatibilità esistente tra questi due linguaggi di programmazione.

### 10.1 Mashup basato sugli eventi

Negli ultimi anni molti fornitori di servizi nonché operatori di telecomunicazioni, hanno deciso di esporre le funzionalità di base dei loro servizi agli sviluppatori di terze parti attraverso le Application Programmable Interfaces (API). Tali API, che di solito sono basate su tecnologie Web Service come SOAP e REST, possono essere combinate insieme per creare servizi innovativi chiamati Mashup. I mashup possono essere sviluppati sia da programmatori esperti che sfruttano il Software Development Kit (SDK) fornito insieme alle API così come da programmatori non esperti che possono usufruire di stru-

---

<sup>1</sup>Con Mash-up si intende un'applicazione web ibrida tale da includere dinamicamente informazioni o contenuti provenienti da più fonti



Figura 10.1: Esempio di Mashup basato sugli eventi

menti grafici come Yahoo! Pipes, ecc... Una volta creati, i Mashup possono essere eseguiti sul lato client e sul lato server. Quest'ultima soluzione è preferibile quando devono essere soddisfatti requisiti di sicurezza, affidabilità, prestazioni, ecc...

Le piattaforme lato server devono essere scalabili per gestire un gran numero di esecuzioni concorrenti di Mashup. Inoltre una Mashup Execution Platform (MEP) dovrebbe essere progettata tenendo conto della disponibilità di architetture multi-processore, che può essere sfruttata per eseguire servizi composti in modo efficiente. Da un punto di vista di progettazione, questo tipo di piattaforme presentano molte analogie con i Web Server che si sviluppano per servire grandi volumi di richieste; in tal modo è possibile riutilizzare alcuni degli approcci già conosciuti per questo tipo di piattaforme (in particolare l'approccio Event Driven e quello basato sui Thread esposto nel capitolo 6.1).

Un Event Driven Mashup è un'applicazione sviluppata componendo diversi servizi di base che comunicano attraverso degli eventi. Ogni servizio di base è definito attraverso una serie di azioni (cioè le funzionalità specifiche fornite da un servizio), una serie di eventi (es: cambiamenti di stato che possono verificarsi in un servizio), e una insieme di proprietà che possono essere di input o di output. Per esempio, il servizio Monitor Google Calendar Service illustrato nella figura 10.1 fornisce l'azione "start monitoring GCalendar", l'evento "when-new-item-matching-filters-found", la proprietà di input "user credentials", "date range", "event-name-keyword" e la proprietà di output "new-item-info". Tale servizio di base continua a monitorare il profilo di Google Calendar specificato dalla proprietà "user credentials" e lancia un nuovo evento "when-new-item-matching-filters-found" ogni volta che un nuovo elemento, aggiunto al calendario, corrisponde ai filtri in input "date range"/"event-name-keyword". L'evento ha la proprietà di output "new-item-info". La figura 10.1 mostra il Mashup Event Driven appena descritto per il servizio Monitor Google Calendar, dove l'evento "when-new-item-matching-filters-found" attiva la valutazione di una specifica condizione nel secondo blocco (es: controllo dell'ora) e, infine, se il blocco Conditional Service emette l'evento "if-condition-is-true" vengono mostrati alcuni dati (es: la posizione

del nuovo oggetto recuperato dal Monitor Google Calendar Service).

## 10.2 Una piattaforma Mashup basata su Scala

In questa sezione viene descritto il modo in cui è stato re-implementato il MEP la cui architettura è stata mostrata in Figura 10.2 sfruttando il linguaggio di programmazione Scala.

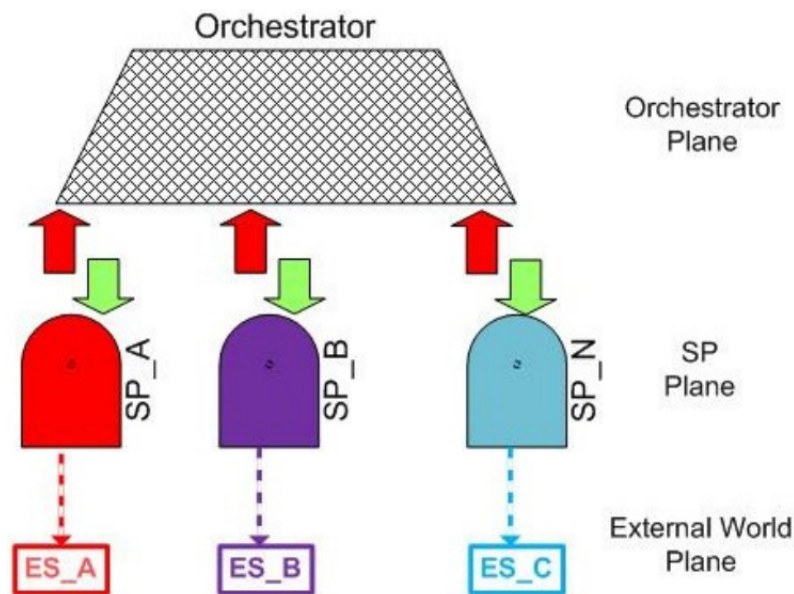


Figura 10.2: Architettura d'alto livello per la piattaforma Mashup lato server

### 10.2.1 Mapping dei componenti MEP negli attori Scala

Il lavoro sviluppato è consistito in una completa migrazione dal vecchio MEP basato sul linguaggio Java ad un MEP basato su Scala, riutilizzando le funzionalità di base dell'Orchestratore già sviluppate in Java, sfruttando la compatibilità tra la Scala e Java. Nella versione dell'Orchestratore basata su Java, è stata disaccoppiata l'attuale business logic (es: un metodo per l'implementazione di nuovi Mashup, un metodo per il routing dei nuovi eventi Mashup in arrivo, ecc...) dal modello sottostante di threading. Questo approccio modulare ha permesso un rapido porting della componente Orchestrator al linguaggio di programmazione Scala, come mostrato nel Listing 10.1 dove sono state sfruttate le seguenti caratteristiche distintive di Scala:

- L'implementazione del modello ad attori senza Thread (es: con l'utilizzo della primitiva `react`) che permette di minimizzare il numero di Thread attivi, migliorando così l'utilizzo della memoria riducendo così l'overhead da context switch;
- La compatibilità tra Java e Scala permette di importare la libreria Java che contiene l'attuale logica dell'Orchestrator (riga 3 nel Listing 10.1) e di riutilizzarla come in un normale programma Java (riga 32 del Listing 10.1).

Listing 10.1: Codice Scala dell'Orchestratore

```

1 import scala.actors.Actor
2 import Actor._
3 import mep.orchestrator._
4
5 abstract class Message
6 case class MessageDo(val actionName: String,
7                       val baseServiceID: String,
8                       val baseServiceInstanceID: String,
9                       val sessionID: String,
10                      val sessionProperties: Array[SessionProperty],
11                      val orch: Orchestrator) extends Message
12 case class MessageEvent(val eventName: String,
13                         val baseServiceID: String,
14                         val baseServiceInstanceID: String,
15                         val sessionID: String,
16                         val sessionProperties: Array[SessionProperty],
17                         val orch: Orchestrator) extends Message
18 case class MessageDeployment(val mashupName: String,
19                             val gsdFile: mashupXMLdescription) extends Message
20
21 class Orchestrator extends Actor
22 {
23   def act =
24   {
25     val orchJava = new OrchestratorImpl()
26     loop
27     {
28       react
29       {
30         case MessageEvent(eventName, baseServiceID,
31                           baseServiceInstanceID, sessionID, sessionProperties, orch) =>
32         {
33           val invokeActions = orchJava.notifyEvent(eventName,
34             baseServiceID, baseServiceInstanceID, sessionID,
35             sessionProperties)
36           invokeActions foreach(j => var sp = new ServiceProxy()
37                                 sp.start
38                                 sp ! messageDo(
39                                   j.getNextAction(),
40                                   j.getBsId(),
41                                   j.getBsInstanceId(),
42                                   sessionID, j.getSp(),
43                                   orchJava)
44                                 )
45         }
46         case Deployment(scName, gsdFile) => orche.deployService(scName,
47                           gsdFile)
48       }
49     }
50   }
51 }

```

```

40     }
41   }
42 }

```

Listing 10.2: Codice Scala di un generico Service Proxy

```

1 import scala.actors.Actor
2 import Actor._
3
4 class ServiceProxy extends Actor
5 {
6   def act =
7     {
8       loop
9       {
10        react
11        {
12          case messageDo(eventName, baseServiceID, baseServiceInstanceID,
13            sessionID, sessionProperties, orch) =>
14            {
15              //logica del SP
16              orch ! MessageEvent("newEventName", baseServiceID,
17                baseServiceInstanceID, sessionID, sessionProperties, orch)
18            }
19          }
20        }
21      }
22    }
23 }

```

Non appena un nuovo evento iniziale (es: un evento che attiva l'esecuzione di un nuovo Mashup) viene ricevuto dal MEP, una nuova istanza dell'attore Orchestrator viene creata da una componente chiamato Orchestration Management System (OMS) che è incaricata di avviare le nuove sessioni. Da notare che ogni volta saranno presenti un certo numero di istanze di attori pari al numero di sessioni attive, ma grazie all'uso della primitiva `react`, si è in grado di evitare l'associazione di un thread ad ogni istanza di attore. Ne consegue che è stata progettata una soluzione scalabile in grado di gestire fino a milioni di attori.

L'orchestratore gestisce tre tipi di messaggi:

- **MessageDeployment** è ricevuto dall'Orchestratore quando dovrebbe essere installato un nuovo mashup nel MEP. Si tratta di una sorta di operazione di "gestione";
- **MessageDo** viene inviato agli attori Service Proxy per invocare un'azione che rappresenta la componente di un Mashup (es: l'account di

Monitor GCalendar). L'ultimo parametro di questo messaggio, chiamato `orch`, è un riferimento all'istanza dell'attore Orchestrator che sta gestendo l'esecuzione della sessione corrente;

- **MessageEvent** viene ricevuto dall'Orchestrator quando un si verifica un cambiamento di stato in un Service Proxy. Questo messaggio attiva l'esecuzione dell'algoritmo di routing (riga 32 del Listing 10.1) che restituisce l'elenco degli SP da richiamare. L'algoritmo di routing è provvisto di una variabile chiamata `orchJava` che è un'istanza della classe `OrchestratorImpl` contenuta nel pacchetto `mep.orchestrator._`, libreria Java importata alla riga 3 nel Listing 10.1.

Per quanto riguarda gli SP, è necessario fornire un'implementazione ad attori per ogni servizio di base disponibile nel SCP (vedi paragrafo 2.1). La struttura generale di un attore SP è mostrato nel Listing 10.2. Quando si riceve un messaggio `MessageDo` l'attore esegue la sua specifica logica e poi, quando si verifica un cambiamento di stato, invia un messaggio `MessageEvent` all'Orchestratore<sup>2</sup>. Dal codice si può notare che il corpo della primitiva `react` è eseguita molto velocemente per evitare problemi di starvation tra i Thread, così alcuni SP più complessi (es: polling delle risorse esterne) dovrebbero essere attuati mediante meccanismi più complessi come la continuation di Scala (paragrafo 8.2) e/o non-blocking I/O (paragrafo 8.1).

### 10.2.2 L'algoritmo di orchestrazione

Viene ora descritto più in dettaglio il flusso esecutivo dell'algoritmo di orchestrazione finora esposto.

- I. La componente OMS riceve una richiesta di esecuzione di un nuovo Mashup (es: da un portale web), genera un nuovo `sessionId` e crea una nuova istanza dell'attore Orchestrato;
- II. La nuova Orchestrator Actor Instance (OAI) esegue le istruzioni di inizializzazione e si blocca sulla primitiva `react` (riga 28 nel Listing 10.1), in attesa di nuovi messaggi in arrivo. Il framework Scala gestisce questa operazione potenzialmente bloccante senza l'assegnazione di un Thread bloccato per il solo protocollo OAI;
- III. L'OMS manda il messaggio `MessageEvent` al nuovo OAI creato;

---

<sup>2</sup>Il parametro `orch` nel messaggio `MessageDo` contiene un riferimento all'istanza dell'orchestratore che sta gestendo la sessione

- IV. Il framework Scala gestisce l'arrivo dei nuovi messaggi riattivando l'OAI corrispondente (es: uno Scheduler invia il messaggio all'istanza attore corretta);
- V. L'OAI entra nel corpo del react e si accorge che il messaggio ricevuto corrisponde al caso `MessageEvent`, quindi esegue le linee 30-37 del Listing 10.1. I messaggi vengono inviati al SP successivo. In questo caso l'attore SP `GMailMonitor` riceverà il messaggio;
- VI. L'OAI ritorna dallo stato "bloccato" alla riga 28 del Listing 10.1. Si noti che nuovamente non è associato alcun `Thread` a questo OAI finché non riceve un nuovo messaggio;
- VII. L'attore SP `GMailMonitor` entra nel corpo della `react` e scopre che il messaggio ricevuto corrisponde caso `MessageDo` quindi esegue la sua business logic (possibilmente con operazioni di non-blocking I/O);
- VIII. Non appena un nuovo evento viene recuperato nel calendario, un nuovo messaggio `MessageEvent` viene inviato all'OAI il cui riferimento era contenuto nel messaggio `MessageDo`;
- IX. L'esecuzione prosegue con un scambio di messaggi tra gli OAI e le istanze SP come descritto nei passaggi IV-VIII.

### 10.2.3 Conclusioni

La possibilità di importare librerie Java nel codice Scala e l'intelligente implementazione modulare del MEP ha avuto come obiettivo la separazione della business logic dal modello di Threading, semplificando la portabilità della realizzazione del MEP precedente verso l'implementazione in Scala, sfruttando il modello ad attori. L'implementazione senza `Thread` del modello ad attori attraverso l'uso della primitiva `react`, supporta la realizzazione di un MEP scalabile in quanto, consente di ridurre il numero di `Thread` attivi nel sistema, ridurre l'effetto di memory pressure<sup>3</sup> causato dalla gestione degli stack dei `Thread`, e l'overhead da context switch. Inoltre, il modello ad attori di Scala permette anche di ridurre l'overhead dovuto alla sincronizzazione in quanto non è basato sul modello a memoria condivisa. Nell'implementazione basata su Scala, il numero di `Thread` attivi nel sistema è dato dal numero di

---

<sup>3</sup>La memoria dinamica determina la quantità di memoria necessaria ad una macchina virtuale (VM) calcolando la memory pressure. Essa è data dal rapporto della quantità di memoria che richiede la VM con quella che già ha in uso.

notifiche di eventi e le invocazioni di azioni che vengono effettivamente eseguite in un certo momento. Questa soluzione supera il modello tradizionale basato sul modello thread-per-session (paragrafo 3.2) in cui viene assegnato un Thread per ogni sessione anche se le notifiche degli eventi e le invocazioni di azioni non sono realmente accadute. Questo approccio è stato utilizzato per risolvere problemi simili a quelli che si accadono nei Web Server quando la piattaforma gestisce le sessioni HTTP.



# Capitolo 11

## Prestazioni dell'Orchestratore

### 11.1 L'Orchestratore in Scala

In questo capitolo vengono presentati i risultati dei test svolti sull'orchestratore in quattro situazioni diverse:

- Client, Orchestratore e Service Proxy sono avviati in locale sulla stessa macchina;
- Orchestratore e Service Proxy sulla stessa macchina ma separati rispetto al Client che funziona in remoto sulla stessa rete;
- Ognuno dei tre componenti funziona su macchine distinte nella stessa rete;
- Client remoto che comunica con un Server che utilizza più Orchestratori e Service Proxy in locale.

#### 11.1.1 Funzionamento in locale

In questa situazione, i tre attori sono eseguiti all'interno della stessa macchina. La stessa utilizza anche jconsole/jvisualvm per rilevare le prestazioni. Lo schema architetturale è riportato nella figura seguente:

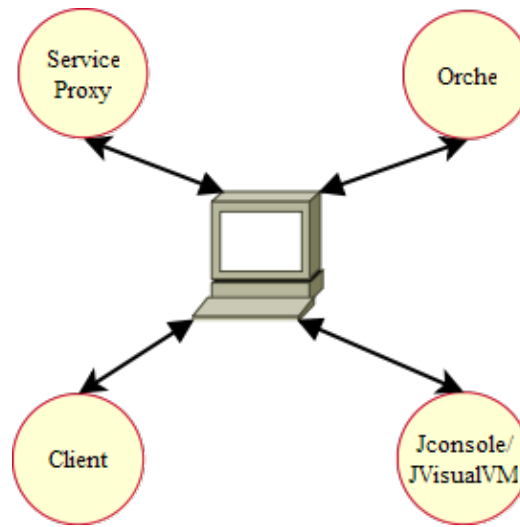


Figura 11.1: Architettura per l'esecuzione in locale

Il codice dell'Orchestratore e del Service Proxy sono riportati rispettivamente nei Listing 10.1 a pagina 111 e 11.1 mentre per il client viene riportato di seguito:

Listing 11.1: Codice Scala di un generico Service Proxy

```

1 import see.orchestrator._
2
3 object Client extends Application
4 {
5     var i = 1
6     ...
7     while(true)
8     {
9         i += 1
10        var orch = new Orchestrator()
11        orch.start
12        if (i == 2) orch ! Deployment("Prova",
13                                     "Deployment.xml");
14        orch ! Evento("when-start-request-received",
15                     "Start", "0", "SP.1." + i, vectSessionProp, orch)
16        Thread sleep xXx //un certo tempo
17    }
18 }

```

Come da codice, il corpo del client è costituito da un ciclo while nella quale viene creata e avviata una nuova istanza dell'orchestratore ad ogni ciclo. Solo per il primo ciclo il client invia all'orchestratore un messaggio di Deployment

contenente il file XML con la struttura dei Service Proxy. Viene poi inviato un messaggio di tipo Evento che contiene il nome del Mashup e dell'evento iniziale "when-start-request-received", un ID del Service Proxy, un ID dell'istanza del SP, l'ID della sessione, un'array contenente le Session Property e l'istanza dell'orchestratore stesso.

Questi tre attori funzionano localmente sulla stessa macchina. Le prove effettuate si differenziano per i tempi con cui vengono generate nuove richieste da parte del client e per i tempi di elaborazione dei singoli Service Proxy. Di seguito viene riportata una tabella riassuntiva con i diversi tempi sperimentati e la media calcolata su un'esecuzione di circa 300 secondi.

Tabella 11.1: Risultati relativi alle prove in locale

|                | Tempo tra nuove<br>richieste [ms] | Tempo esecuzione SP<br>[ms] | Media [ms] |
|----------------|-----------------------------------|-----------------------------|------------|
| <b>Prova 1</b> | 100                               | 100                         | 201,8358   |
| <b>Prova 2</b> | 50                                | 0                           | 1,29031    |
| <b>Prova 3</b> | 100                               | 50                          | 101,6028   |
| <b>Prova 4</b> | 1000                              | 2000                        | 4923,9     |

Tabella 11.2: Percentuale metodi occupazione CPU

|                | Scambio<br>messaggi | Thread Pool<br>(sync) | Thread Pool<br>(F/J) | notifyEvent() | Altro |
|----------------|---------------------|-----------------------|----------------------|---------------|-------|
| <b>Prova 1</b> | 36%                 | 33%                   | 14%                  | 0,2%          | 16,8% |
| <b>Prova 2</b> | 35%                 | 34%                   | 17%                  | 0,7%          | 13,3% |
| <b>Prova 3</b> | 20%                 | 50%                   | 18,6%                | 0,3%          | 11,1% |
| <b>Prova 4</b> | 77,7%               | /                     | 18,9%                | 0,1%          | 3,3%  |

In Figura 11.2 vengono riportati i grafici che mettono in risalto l'andamento dei tempi di esecuzione di tutti i Service Proxy per tutta la durata dell'esecuzione.

Nelle Figure 11.3 e 11.4 vengono presentati i grafici relativi alle risorse di sistema impiegate (CPU, Memoria, Classi caricate e attività della GC) nelle 4 prove.

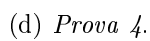
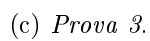
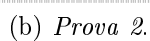
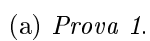
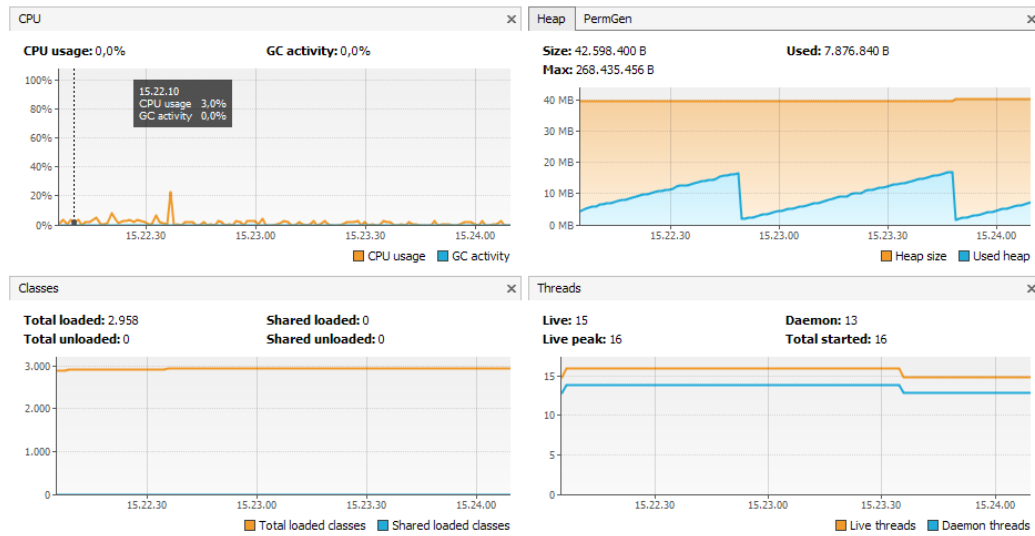
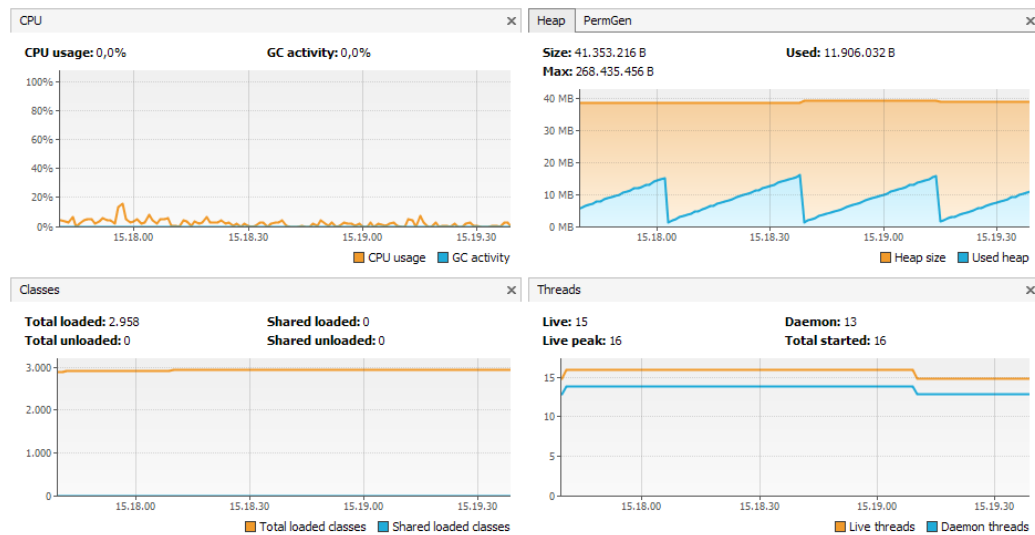


Figura 11.2: Andamenti temporali per l'esecuzione in locale

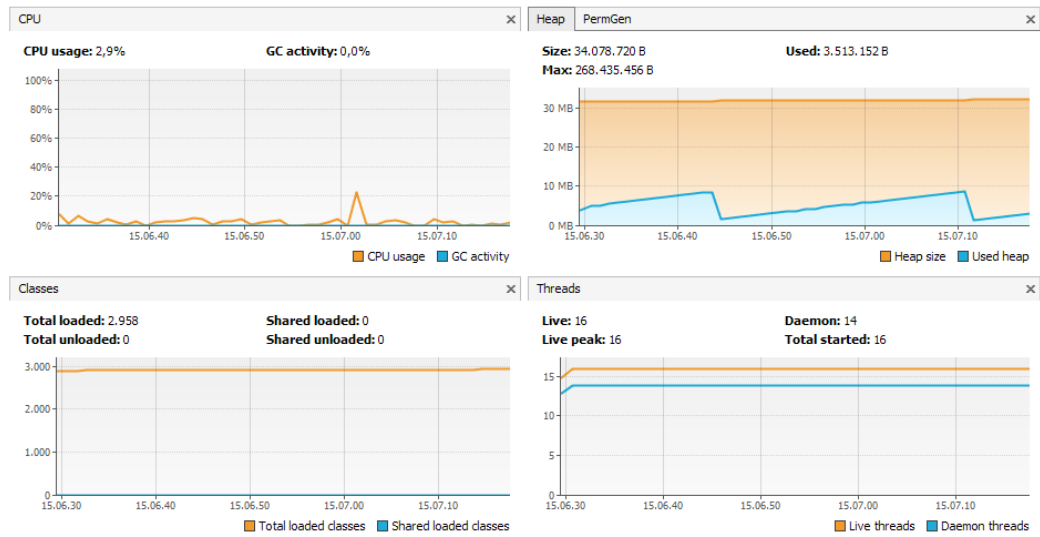


(a) Prova 1.

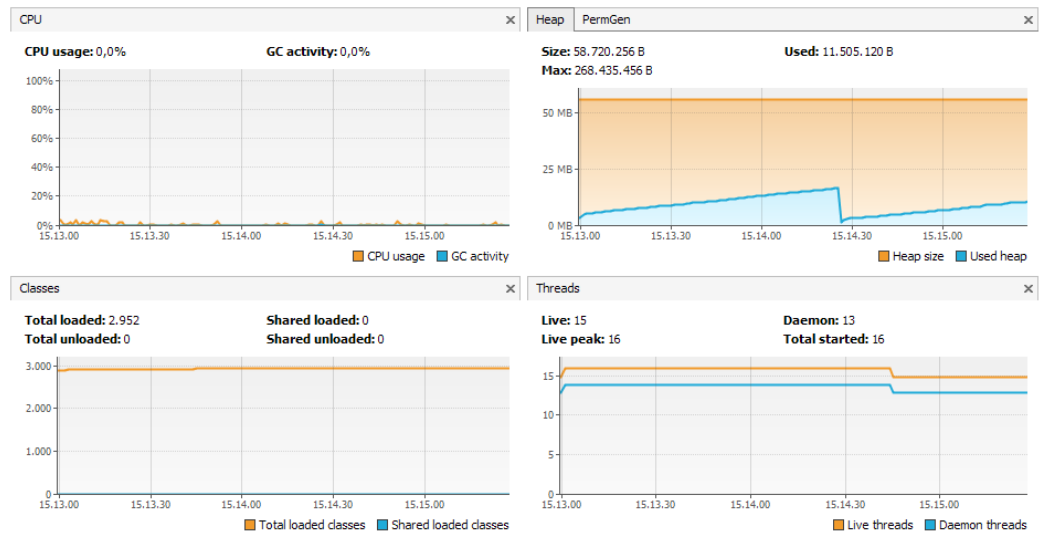


(b) Prova 2.

Figura 11.3: Risorse impiegate per l'esecuzione in locale - Parte I



(a) Prova 3.



(b) Prova 4.

Figura 11.4: Risorse impiegate per l'esecuzione in locale - Parte II

Nella Prova 1, i tempi utilizzati sono uguali e relativamente brevi che portano ad una media di esecuzione totale del singolo SP pari a poco più la somma dei due tempi e ad un andamento che è omogeneo con qualche picco più lungo nella fase iniziale. Discorsi analoghi si possono fare anche per la Prova 3. Diversa è la Prova 2 in quanto viene utilizzato un tempo basso per la generazione delle richieste e un tempo nullo per l'esecuzione del SP che rappresenta la situazione di lavoro ideale. L'andamento che ne conse-

gue mantiene una media di esecuzione molto bassa sugli 1,2 ms. La Prova 4 invece rappresenta la situazione sfavorevole in quanto a fronte di richieste lunghe nel tempo, il SP ha tempi di esecuzione molto lunghi ottenendo una media di circa 5 secondi per SP. Questa situazione è anche quella sconsigliata dagli autori di Scala, come si può leggere anche in [15] e come analizzato nel Capitolo 7 a pagina 57, quando si utilizza il metodo `react` per ricevere un messaggio.

Per quanto riguarda le risorse impiegate, in tutti i casi si possono osservare caratteristiche simili come un'occupazione media del processore di circa il 20%, una dimensione impiegata dell'Heap in memoria che varia tra i 10 e i 20MB e un numero costante di Thread presenti nel sistema che conferma i presupposti iniziali di Scala nel mantenere basso e invariato il numero di Thread.

### 11.1.2 Orchestratore e Service Proxy locali e Client remoto

Per questa prova si è messo in esecuzione il Client e l'Orchestratore/Service Proxy su due macchine distinte, collegate per mezzo di una rete locale attraverso uno switch, a cui è stato collegato una terza macchina per effettuare i rilevamenti delle prestazioni per non alterare eccessivamente le prove svolte. Lo schema architetturale è riportato nella figura seguente:

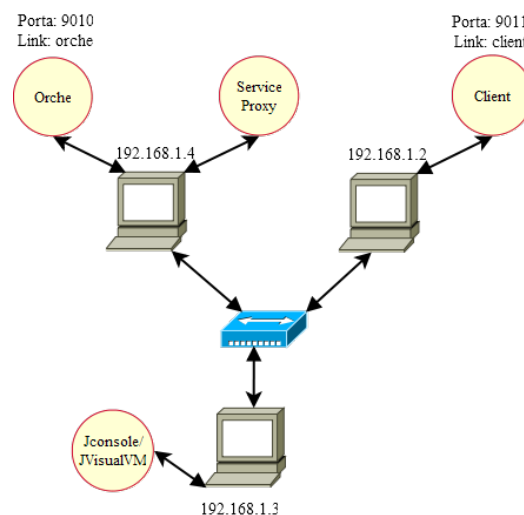


Figura 11.5: Architettura per l'esecuzione con Orchestratore e Client remoto

In questa situazione il codice dell'Orchestratore e del Client devono essere modificati come segue per poter introdurre il concetto di attori remoti.

Listing 11.2: Codice Scala di un Orchestratore remoto

```
1 import scala.actors.Actor
2 import Actor._
3 import scala.actors.remote.{Locator,Node,RemoteActor}
4 import scala.actors.remote.RemoteActor.{alive,register}
5 import scala.actors.AbstractActor
6 import see.orchestrator._
7
8 object OrchestratoreApp
9 {
10   def main(args: Array[String])
11   {
12     val orche = Locator(Node("127.0.0.1", 9010), 'orche)
13     new Orchestratore(orche)
14   }
15 }
16
17 class Orchestratore(me : Locator)
18 {
19   RemoteActor.classLoader = getClass().getClassLoader()
20   actor
21   {
22     alive (me.node.port)
23     register (me.name, self)
24     var i = 1
25     val orche = new OrchestratorImpl()
26     loop
27     {
28       react
29       {
30         ... come Orch locale ...
31       }
32     }
33   }
34 }
```

Nella versione remota, l'orchestratore possiede due componenti, un object e una classe. L'object implementa il main e rappresenta la componente che permette l'avvio dell'applicazione. Al suo interno vengono creati i riferimenti "simbolici" all'orchestratore, ovvero viene creata una variabile orche, che rappresenta il riferimento all'orchestratore remoto, e assume il valore assegnato dalla chiamata Locator che a sua volta accetta una componente di tipo Node (cioè la coppia di valori "indirizzo orchestratore" e "porta in ascolto") e un

riferimento simbolico. Così facendo, ci si riferisce all'istanza di orchestratore per mezzo della terna (indirizzo, porta, link simbolico). Infine, la classe `OrchestratoreApp` crea una nuova istanza di `Orchestratore` che contiene a tutti gli effetti la logica implementativa già vista in precedenza. Dal momento che gli attori remoti in Scala utilizzano la serializzazione degli oggetti di Java per l'invio di messaggi da e verso un socket, nella classe `Orchestratore` viene invocato, per prima cosa, il metodo `getClass()` perché il `ClassLoader`<sup>1</sup> che Scala utilizza quando de/serializza un attore è quello che si trova in `RemoteActor` il cui valore di default è `null`.

Il blocco `actor` implementa il comportamento dell'`Orchestratore`. Nel precedente object era stato solo creato il riferimento simbolico dell'orchestratore, qui invece viene istanziato realmente l'orchestratore sulla base dei parametri simbolici specificati in precedenza attraverso l'invocazione dei metodi `alive` (per la porta) e `register` (che registra la coppia indirizzo e riferimento a sé stesso).

---

<sup>1</sup>Il `ClassLoader` è l'oggetto responsabile del caricamento delle classi. Dato il nome di una classe, un `ClassLoader` cerca di individuare o generare dati che costituiscono una definizione per la classe.

Listing 11.3: Codice Scala di un Client remoto

```
1 import scala.actors.Actor
2 import Actor._
3 import scala.actors.remote.{Locator,Node,RemoteActor}
4 import scala.actors.remote.RemoteActor.{alive,register}
5 import scala.actors.AbstractActor
6 import see.orchestrator._
7
8 object ClientApp
9 {
10     def main(args: Array[String])
11     {
12         val client = Locator(Node("127.0.0.1", 9011), 'client)
13         new Client(client)
14     }
15 }
16
17 class Client(val client : Locator)
18 {
19     RemoteActor.classLoader = getClass().getClassLoader()
20     actor
21     {
22         alive(client.node.port)
23         register(client.name, self)
24         val orche = Locator(Node("192.168.1.4", 9010), 'orche)
25         val orcScala = select(orche.node, orche.name)
26
27         ... come il client locale ...
28     }
29 }
```

Tabella 11.3: Risultati relativi alle prove con Orchestratore e Client remoto

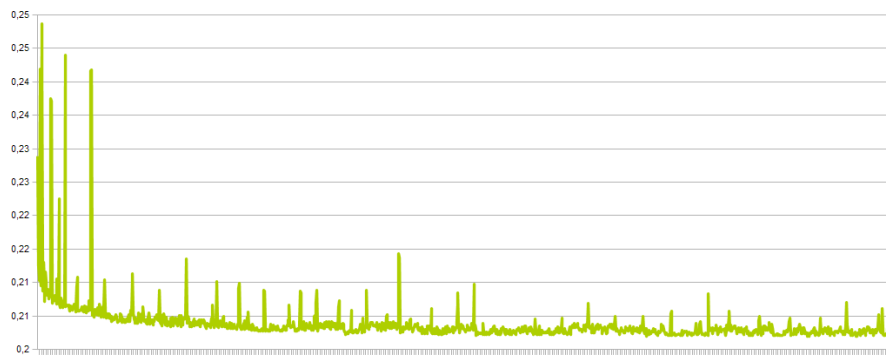
|                | Tempo tra nuove<br>richieste [ms] | Tempo esecuzione SP<br>[ms] | Media [ms] |
|----------------|-----------------------------------|-----------------------------|------------|
| <b>Prova 1</b> | 100                               | 100                         | 203,71581  |
| <b>Prova 2</b> | 50                                | 0                           | 3,41701    |
| <b>Prova 3</b> | 100                               | 50                          | 102,67951  |
| <b>Prova 4</b> | 1000                              | 2000                        | 4005,268   |

Tabella 11.4: Percentuale metodi occupazione CPU

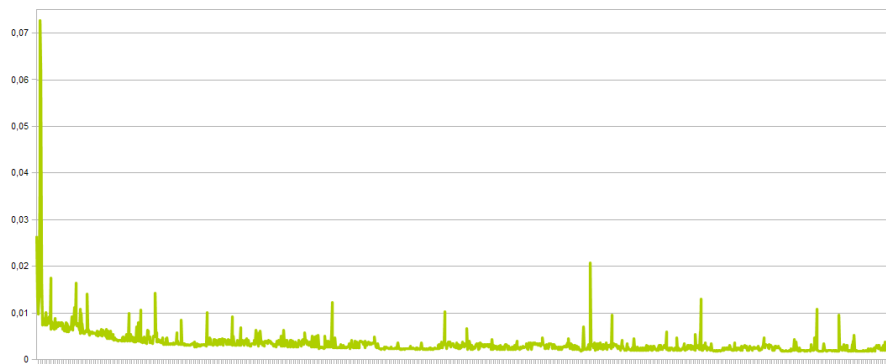
|                | Scambio<br>messaggi<br>(TCP) | Thread Pool<br>(sync) | Thread Pool<br>(F/J) | notifyEvent() | Altro |
|----------------|------------------------------|-----------------------|----------------------|---------------|-------|
| <b>Prova 1</b> | 11,3%                        | 63,8%                 | 1,4%                 | 0,1%          | 23,4% |
| <b>Prova 2</b> | 17,7%                        | 80,5%                 | 0,2%                 | 0,1%          | 1,5%  |
| <b>Prova 3</b> | 17,8%                        | 72,5%                 | 0,1%                 | 0,2%          | 9,4%  |
| <b>Prova 4</b> | 1,8%                         | 52,4%                 | 18,7%                | 0,3%          | 26,8% |

In Figura 11.6 vengono riportati i grafici che mettono in risalto l'andamento dei tempi di esecuzione di tutti i Service Proxy per tutta la durata dell'esecuzione.

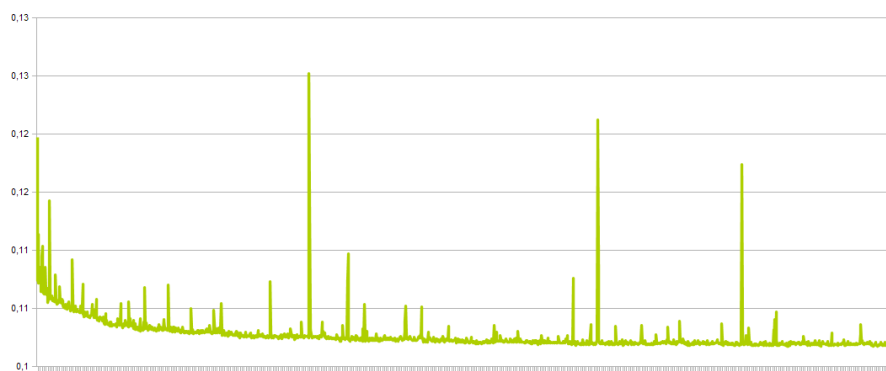
Nelle Figure 11.7 e 11.8 vengono presentati i grafici relativi alle risorse di sistema impiegate (CPU, Memoria, Classi caricate e attività della GC) nelle 4 prove.



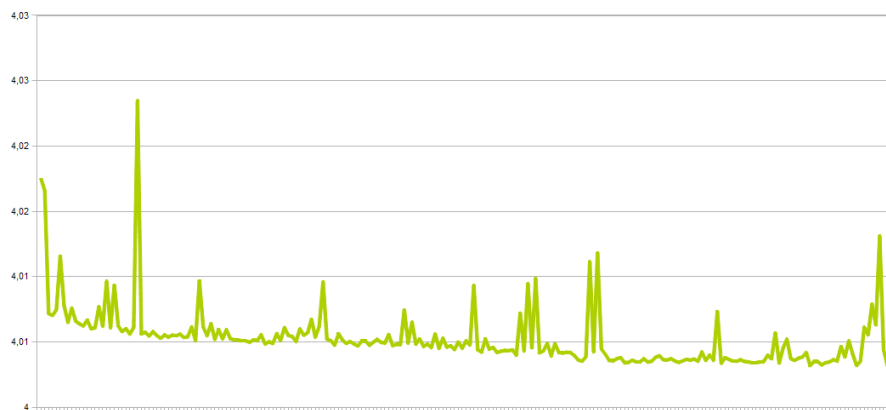
(a) Prova 1.



(b) Prova 2.

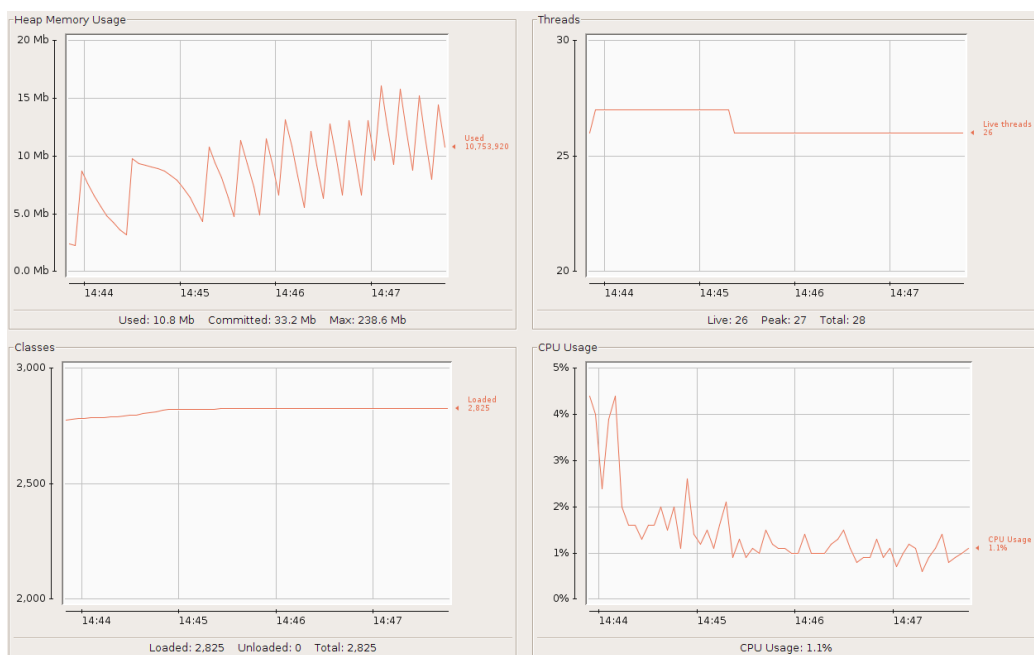


(c) Prova 3.

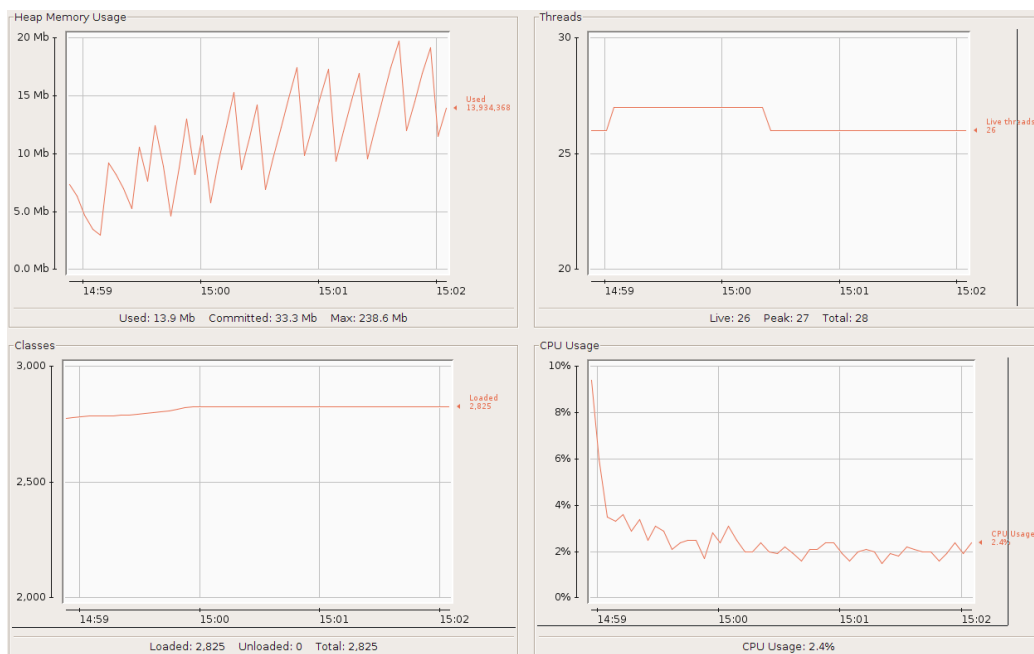


(d) Prova 4.

Figura 11.6: Andamenti temporali per l'esecuzione con Orchestratore e Client remoto

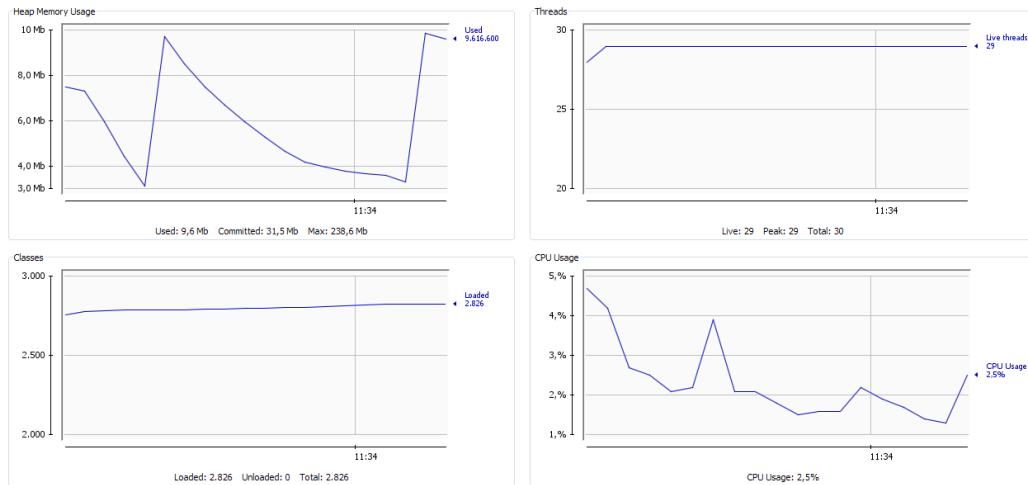


(a) Prova 1.

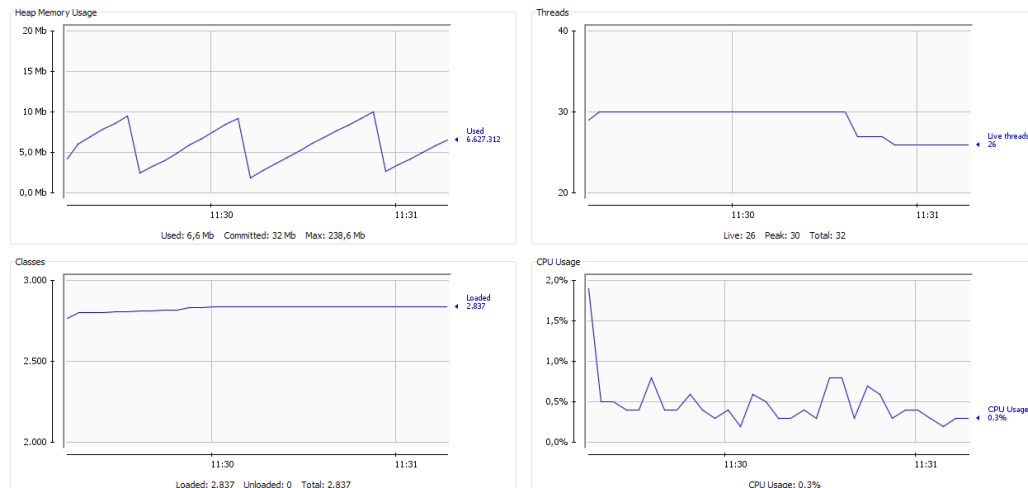


(b) Prova 2.

Figura 11.7: Risorse impiegate per l'esecuzione con Orchestratore e Client remoto - Parte I



(a) Prova 3.



(b) Prova 4.

Figura 11.8: Risorse impiegate per l'esecuzione con Orchestratore e Client remoto - Parte II

Utilizzando le stesse impostazioni della prova precedente (Vedi Tabella 11.1 a pagina 119), dalla Tabella 11.6 si può notare come i tempi medi sono leggermente più lunghi a causa dell'introduzione di un canale TCP/IP nella comunicazione tra il Client e l'Orchestratore.

Confrontando, invece, i diagrammi temporali con quelli del precedente esperimento (Figura 11.2 a pagina 120) si può notare come nel caso remoto la situazione iniziale di startup del sistema incida maggiormente nella durata dell'esecuzione dei Service Proxy, presentando un andamento parabolico e una successiva stabilità nel medio/lungo periodo.

Osservando i grafici sulle risorse impiegate nei due casi, si evince una similitudine nel consumo di tempo di CPU e di memoria per l'Heap con un apparente aumento del Thread. Da un'analisi approfondita, confrontando i Thread effettivi in esecuzione nella versione locale e nella versione remota, si è potuto constatare che, a causa del programma di analisi JVisualVM, vengono creati ulteriori Thread dovuti alla connessione RMI tra la macchina che esegue l'Orchestratore e la macchina che esegue i rilevamenti. Quindi, il numero dei Thread in esecuzione nel sistema è uguale per tutte le versioni testate.

### 11.1.3 Orchestratore, Service Proxy e Client remoti

In questa situazione, vi sono 3 macchine collegate nella rete sulle quali sono in esecuzione i 3 attori. Anche in questo caso vi è stata aggiunta un'ulteriore macchina alla rete, per poter effettuare i rilevamenti prestazionali. Lo schema architetturale è riportato nella figura seguente:

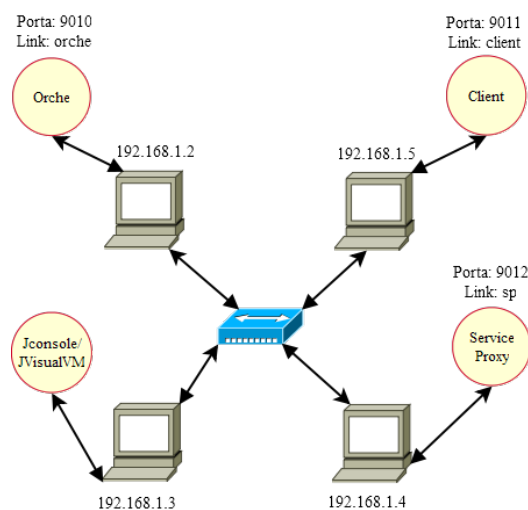


Figura 11.9: Architettura per l'esecuzione remota

Il codice del Client, per questo caso, risulta inalterato rispetto alla prova precedente (Listing 11.3 a pagina 126). All'orchestratore deve essere aggiunto il riferimento al service proxy cui si è registrato nel modo seguente:

Listing 11.4: Codice Scala di un Orchestratore remoto con Service Proxy remoto

```
1 import scala.actors.Actor
2 import Actor._
3 import scala.actors.remote.{Locator,Node,RemoteActor}
4 import scala.actors.remote.RemoteActor.{alive,register}
5 import scala.actors.AbstractActor
6 import see.orchestrator._
7
8 object OrchestratoreApp
9 {
10     def main(args: Array[String])
11     {
12         val orche = Locator(Node("127.0.0.1", 9010), 'orche)
13         new Orchestratore(orche)
14     }
15 }
16
17 class Orchestratore(me : Locator)
18 {
19     RemoteActor.classLoader = getClass().getClassLoader()
20     actor
21     {
22         alive (me.node.port)
23         register (me.name, self)
24
25         val spScala = Locator(Node("192.168.1.4", 9012), 'sp)
26         val sp = select (spScala.node, spScala.name)
27
28         var i = 1
29         val orche = new OrchestratorImpl()
30         loop
31         {
32             react
33             {
34                 ... come Orch locale ...
35             }
36         }
37     }
38 }
```

Rispetto al codice precedente, sono state aggiunte le righe 24-25 che rappresentano la registrazione del Service Proxy da parte dell'Orchestratore. Il Service Proxy, invece, viene modificato come segue:

Listing 11.5: Codice Scala di un Service Proxy remoto

```
1 import scala.actors.Actor
2 import Actor._
3 import scala.actors.remote.{Locator,Node,RemoteActor}
4 import scala.actors.remote.RemoteActor.{alive,register}
5 import scala.actors.AbstractActor
6 import see.orchestrator._
7
8 object SPApp
9 {
10   def main(args: Array[String])
11   {
12     val sp = Locator(Node("127.0.0.1", 9012), 'sp)
13     new ServiceProxy(orchestra)
14   }
15 }
16
17 class ServiceProxy(sp : Locator)
18 {
19   RemoteActor.classLoader = getClass().getClassLoader()
20   actor
21   {
22     alive(sp.node.port)
23     register(sp.name, self)
24     val orchScala = Locator(Node("192.168.1.2", 9010), 'orchScala)
25     val orchestra = select(orchScala.node, orchScala.name)
26     loop
27     {
28       react
29       {
30         case messageDo(eventName, baseServiceID,
31           baseServiceInstanceID, compositionSessionID,
32           sessionProperties) =>
33         {
34           Thread sleep xXx //un certo tempo...
35           reply(Evento("when-sleep-time-expires", baseServiceID,
36             baseServiceInstanceID, compositionSessionID,
37             sessionProperties))
38         }
39       }
40     }
41   }
42 }
```

Il Service Proxy alle righe 24-25 registra i riferimenti dell'Orchestratore come visto in precedenza, mentre alla riga 33 viene inviato il messaggio per mezzo del costrutto *reply* che sostituisce implicitamente l'invio del messaggio in modo sincrono (con il metodo "!!") in quanto non è prevista nella progettazione

di Scala l'invio asincrono per attori remoti.

Di seguito viene riportata la tabella con i dati relativi al test in questione:

Tabella 11.5: Risultati relativi alle prove con attori remoti

|                | Tempo tra nuove<br>richieste [ms] | Tempo esecuzione SP<br>[ms] | Media [ms] |
|----------------|-----------------------------------|-----------------------------|------------|
| <b>Prova 1</b> | 100                               | 100                         | 119763     |
| <b>Prova 2</b> | 50                                | 0                           | 53,26      |
| <b>Prova 3</b> | 100                               | 50                          | 545,241    |
| <b>Prova 4</b> | 1000                              | 2000                        | 364950     |

Tabella 11.6: Percentuale metodi occupazione CPU

|                | Scambio<br>messaggi<br>(TCP) | Thread Pool<br>(sync) | Thread Pool<br>(F/J) | notifyEvent() | Altro |
|----------------|------------------------------|-----------------------|----------------------|---------------|-------|
| <b>Prova 1</b> | 19,9%                        | 74,2%                 | 2,3%                 | 0,1%          | 3,5%  |
| <b>Prova 2</b> | 36,7%                        | 62,4%                 | 0,2%                 | 0,2%          | 0,5%  |
| <b>Prova 3</b> | 30,2%                        | 69,4%                 | 0,2%                 | 0,1%          | 0,1%  |
| <b>Prova 4</b> | 30,1%                        | 68,7%                 | 0,1%                 | 0,1%          | 1%    |

In Figura 11.10 vengono riportati i grafici che mettono in risalto l'andamento dei tempi di esecuzione di tutti i Service Proxy per tutta la durata dell'esecuzione.

Nelle Figure 11.11, 11.12, 11.14 e 11.14 vengono presentati i grafici relativi alle risorse di sistema impiegate (CPU, Memoria, Classi caricate e attività della GC) nelle 4 prove dall'Orchestratore (Figure 11.11, 11.12) e del Service Proxy (Figure 11.13, 11.14).

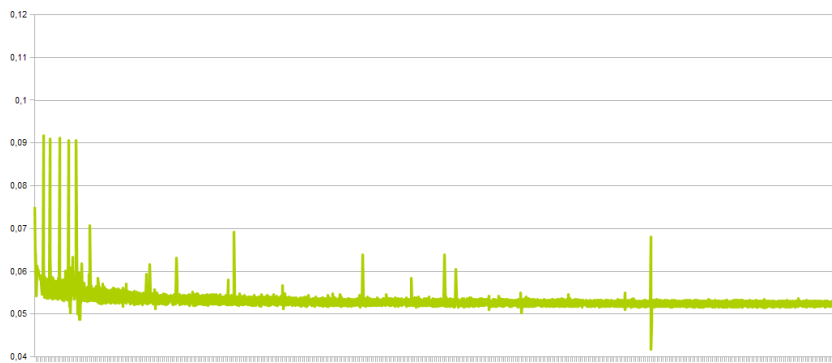
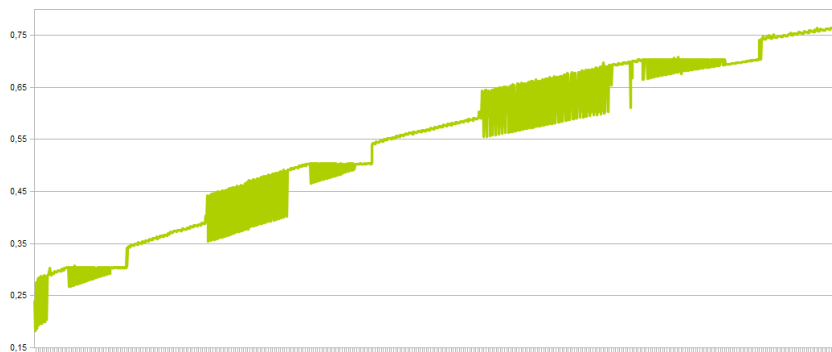
(a) *Prova 1.*(b) *Prova 2.*(c) *Prova 3.*(d) *Prova 4.*

Figura 11.10: Andamenti temporali per l'esecuzione remota

Come si può notare, la remotizzazione del Service Proxy ha peggiorato sensibilmente le prestazioni del sistema introducendo ulteriori ritardi dovuti alla comunicazione remota tra l'Orchestratore e il Service Proxy. La prestazione migliore, come ci si aspettava, deriva dalla situazione ideale della Prova 2 e Prova 3, mentre la situazione peggiore comporta dei ritardi non ragionevoli per la corretta esecuzione del sistema.

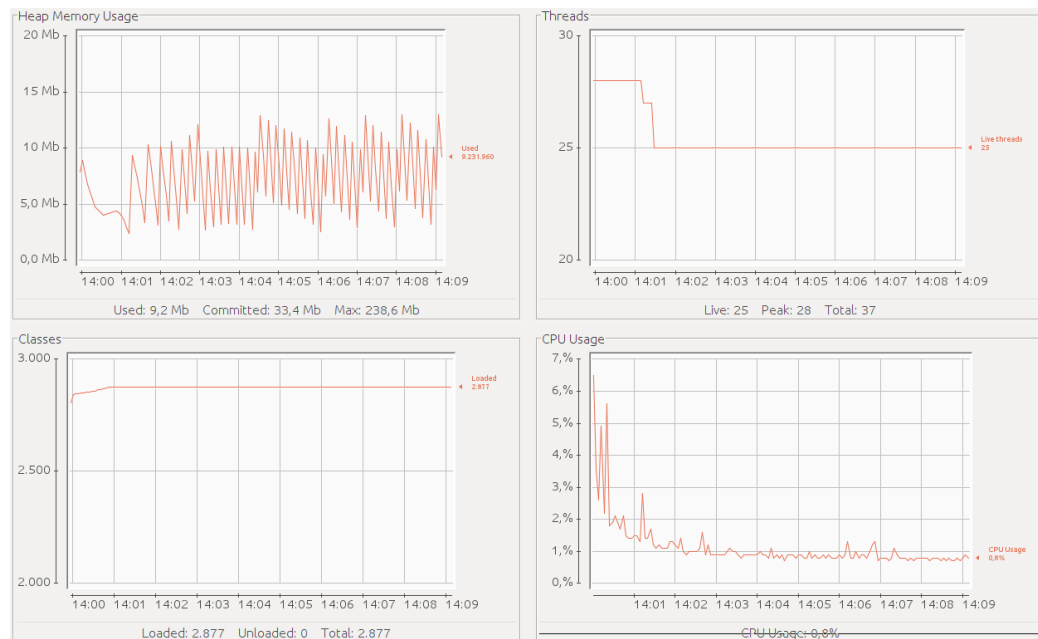


(a) Prova 1.

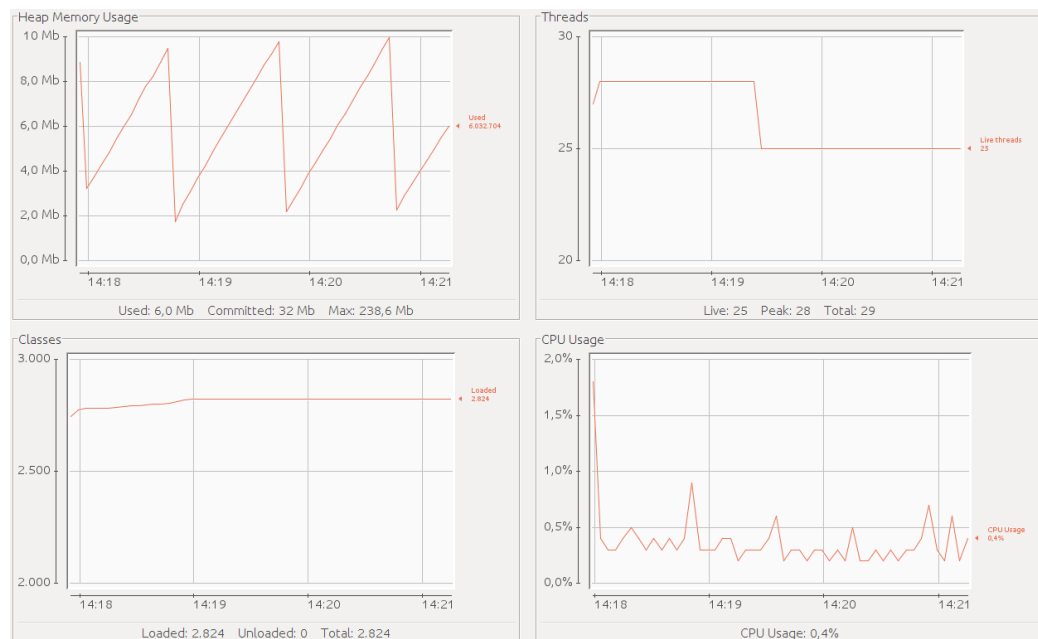


(b) Prova 2.

Figura 11.11: Risorse impiegate per l'esecuzione remota dell'Orchestratore - Parte I

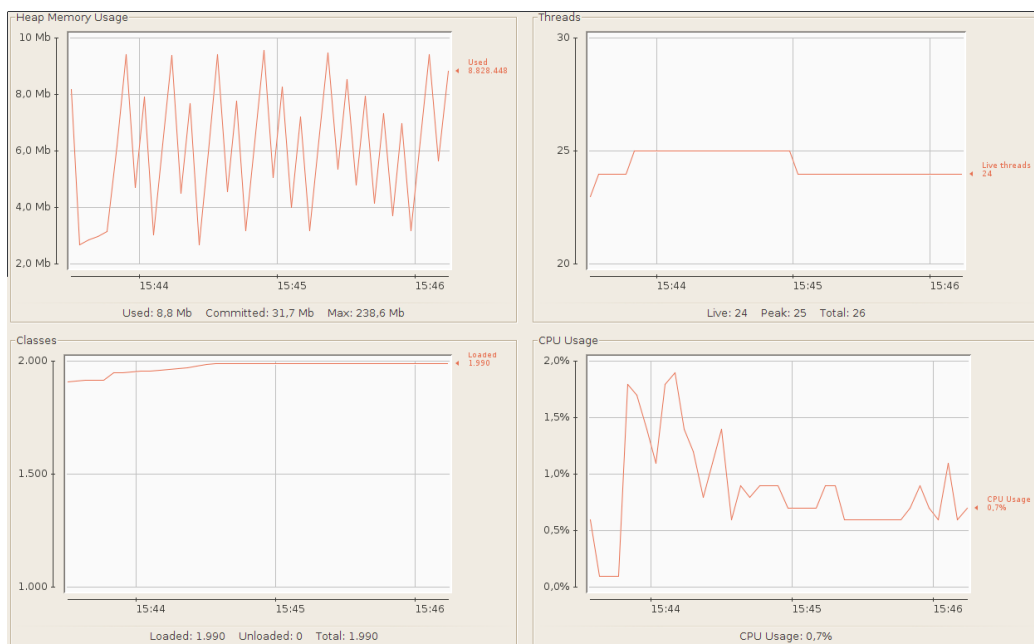


(a) Prova 3.

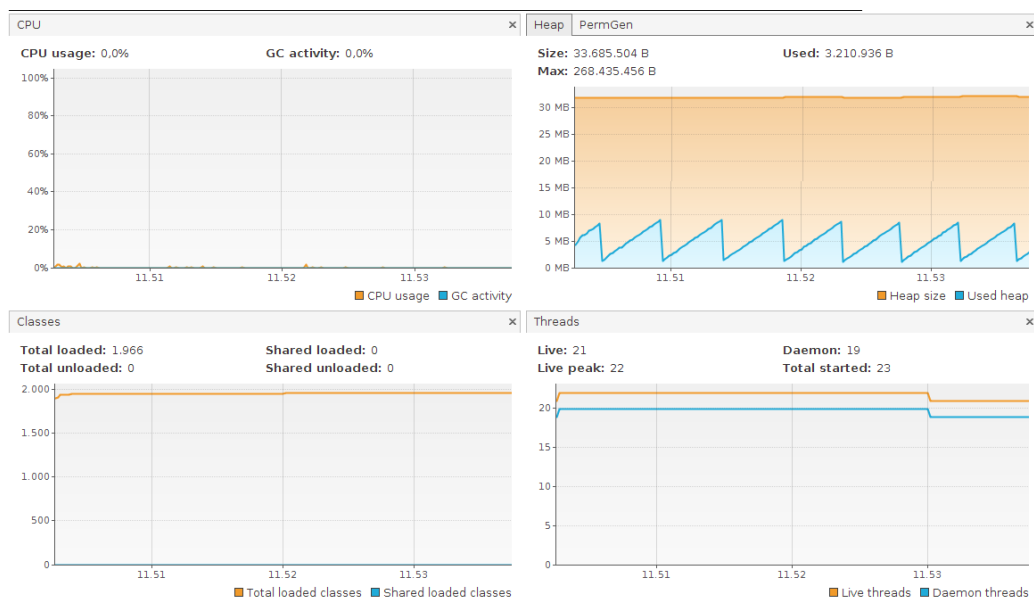


(b) Prova 4.

Figura 11.12: Risorse impiegate per l'esecuzione remota dell'Orchestratore - Parte II

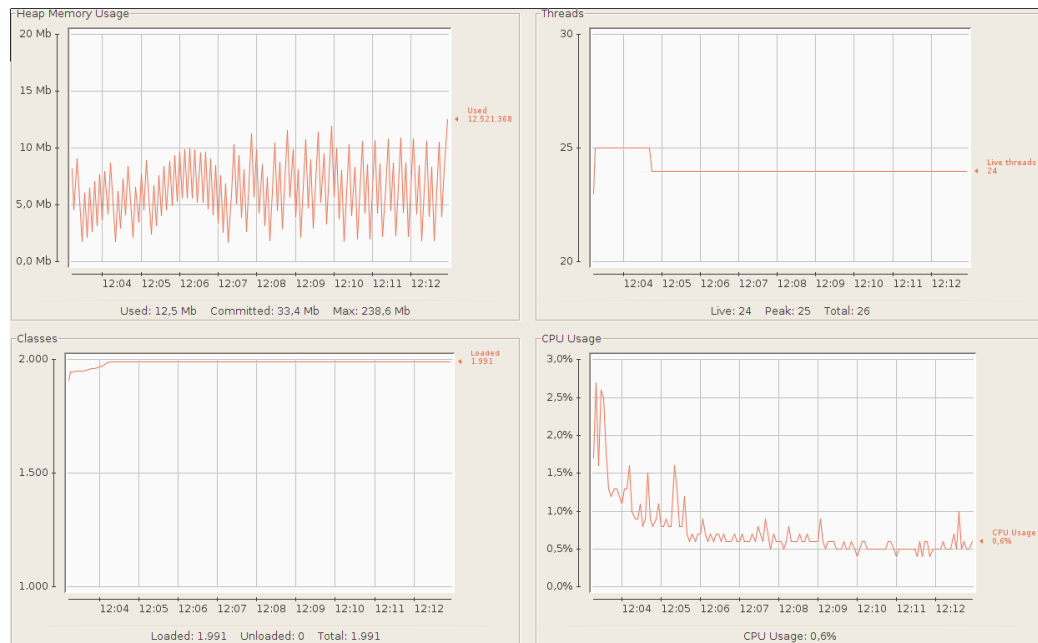


(a) Prova 1.

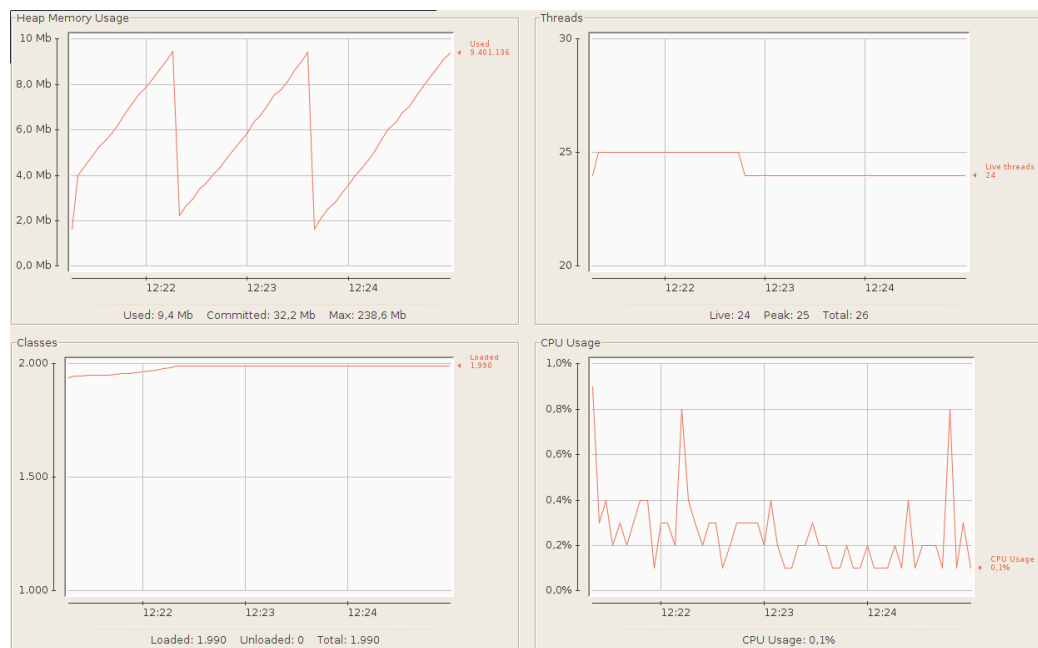


(b) Prova 2.

Figura 11.13: Risorse impiegate per l'esecuzione remota del Service Proxy - Parte I



(a) Prova 3.



(b) Prova 4.

Figura 11.14: Risorse impiegate per l'esecuzione remota del Service Proxy - Parte II

Come nel caso precedente, abbiamo un consumo contenuto del tempo della CPU del circa 2%, un consumo della memoria per l'utilizzo dell'Heap di circa 10/20MB e un numero di Thread medio in esecuzione pari a 24 sia per l'Orchestratore che per il Service Proxy.

#### 11.1.4 Client e Server Service remoti con Orchestratore e Service Proxy locali

Questa situazione, mostrata in Figura 11.15, presenta un'architettura più simile a quella sviluppata nella versione Java di Figura 11.19 a pagina 149, presentata nella prossima sezione. La situazione presenta un Client remoto che comunica con un Server Service remoto che si occupa di ricevere tutte le richieste da parte dei Client e inoltrarle a degli orchestratori generati localmente che comunicano con dei Service Proxy anch'essi eseguiti localmente.

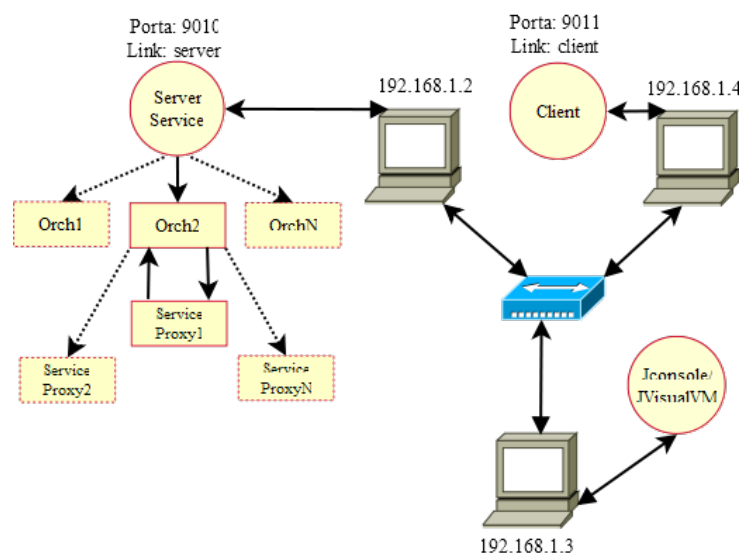


Figura 11.15: Architettura per l'esecuzione con Server Service

Viene ora presentato di seguito il codice del Server Service.

Listing 11.6: Codice Scala di un Orchestratore remoto con Service Server Service

```

1 import scala.actors.Actor
2 import Actor._
3 import scala.actors.remote.{Locator,Node,RemoteActor}
4 import scala.actors.remote.RemoteActor.{alive,register}
5 import scala.actors.AbstractActor
6 import see.orchestrator._
7
8 abstract class Message
9 case class EventoTCP(val eventName: String,
10                     val baseServiceID: String,
11                     val baseServiceInstanceID: String,
12                     val compositionSessionID: String,
13                     val sessionProperties: String) extends Message
14 case class Evento(val eventName: String,
15                 val baseServiceID: String,
16                 val baseServiceInstanceID: String,
17                 val compositionSessionID: String,
18                 val sessionProperties: Array[SessionProperty],
19                 val orch: Orchestratore) extends Message
20 ... //messageDo e Deploy rimangono uguali
21
22 object ServerServiceApp
23 {
24   def main(args: Array[String])
25   {
26     val server = Locator(Node("127.0.0.1", 9010), 'server)
27     new ServerService(server)
28   }
29 }
30
31 class ServerService(me : Locator)
32 {
33   RemoteActor.classLoader = getClass().getClassLoader()
34   def parser (...) //parsing delle SessionProperty per gli Orchestratori locali
35   { ... }
36   actor
37   {
38     alive (me.node.port)
39     register (me.name, self)
40     var i = 1
41     loop
42     {
43       react
44       {

```

```
45         case EventoTCP(...) =>
46         {
47             var vettSessionProperty = new Array[SessionProperty](3)
48             ...
49             parser (...)
50             var orch = new Orchestratore()
51             orch.start
52             orch ! Evento(...)
53         }
54         case Deployment(scName, gsdFile) =>
55         {
56             val orche = new OrchestratorImpl()
57             orche.deployService(scName, gsdFile)
58         }
59     }
60 }
61 }
62 }
```

Il codice del Client rimane inalterato se non per il tipo di messaggio che viene inviato, ovvero EventoTCP. Il messaggio EventoTCP è identico a Evento ma cambia il tipo di SessionProperty che invia per problemi di serializzazione. EventoTCP le invia sotto forma di stringa che vengono tradotte in Array di SessionProperty da una funzione parser alla riga 34. Dopo la conversione, il ServerService inizializza, avvia e spedisce il messaggio Evento all'Orchestratore locale.

Il comportamento dell'Orchestratore alla ricezione del messaggio Evento è lo stesso della versione locale presentato nel Listing 10.1 a pagina 111. Lo stesso discorso vale per il Service Proxy che è stato presentato nel Listing 11.1 a pagina 118. Chiaramente, il comportamento del Client è lo stesso della versione remota del Listing 11.3 a pagina 126 tranne per il fatto che viene inviato un messaggio di tipo EventoTCP anziché Evento.

Di seguito vengono presentate le prestazioni e i tempi impiegati da questa versione.

Tabella 11.7: Risultati relativi alle prove con Server Service

|                | Tempo tra nuove<br>richieste [ms] | Tempo esecuzione SP<br>[ms] | Media [ms] |
|----------------|-----------------------------------|-----------------------------|------------|
| <b>Prova 1</b> | 100                               | 100                         | 200,76675  |
| <b>Prova 2</b> | 50                                | 0                           | 0,51730    |
| <b>Prova 3</b> | 100                               | 50                          | 100,71181  |
| <b>Prova 4</b> | 1000                              | 2000                        | 4001,260   |

Tabella 11.8: Percentuale metodi occupazione CPU

|                | Scambio<br>messaggi<br>(locale) | Scambio<br>messaggi<br>(TCP) | Thread Pool<br>(sync) | Thread Pool<br>(F/J) | notifyEvent() | Altro |
|----------------|---------------------------------|------------------------------|-----------------------|----------------------|---------------|-------|
| <b>Prova 1</b> | 19,2%                           | 8,6%                         | 72%                   | /                    | /             | 0,2%  |
| <b>Prova 2</b> | 0,1%                            | 22,6%                        | 76,5%                 | 0,2%                 | 0,1%          | 0,5%  |
| <b>Prova 3</b> | 10,9%                           | 6,7%                         | 81,3%                 | 0,1%                 | 0,1%          | 0,9%  |
| <b>Prova 4</b> | 34,9%                           | 10,4%                        | 54,5%                 | /                    | 0,1%          | 0,1%  |

In Figura 11.16 vengono riportati i grafici che mettono in risalto l'andamento dei tempi di esecuzione nella versione con Server Service e Client remoto.

Nelle Figure 11.17 e 11.18 vengono presentati i grafici relativi alle risorse di sistema impiegate (CPU, Memoria, Classi caricate e attività della GC) nelle 4 prove.

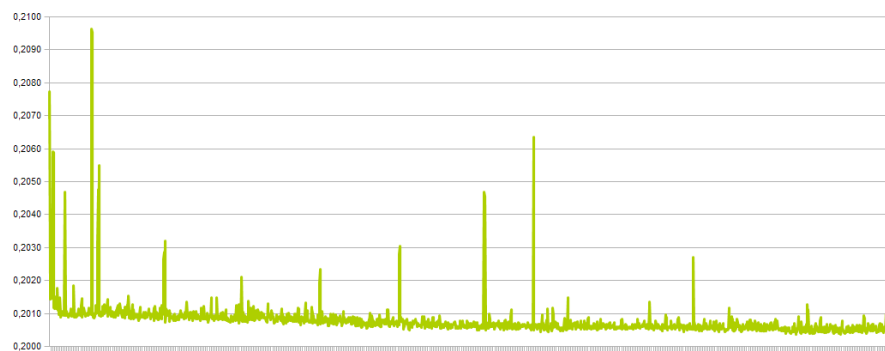
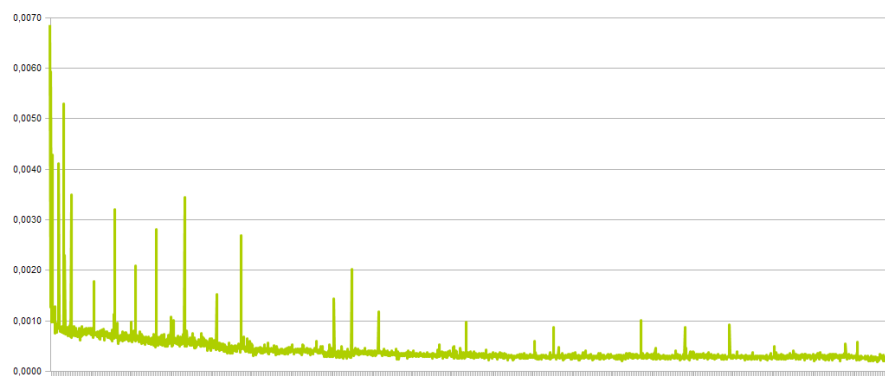
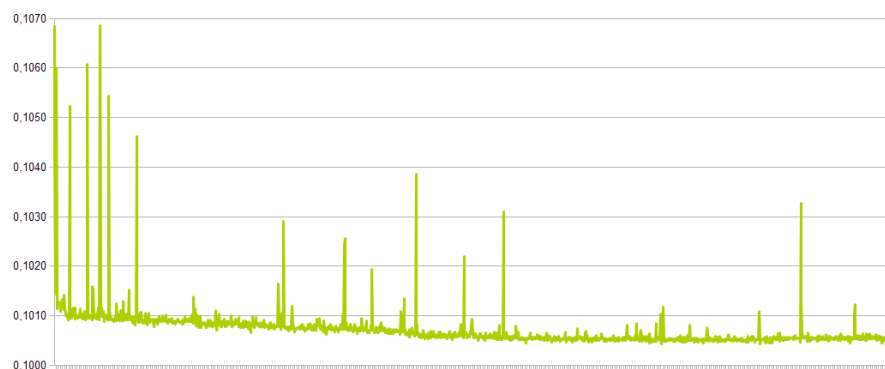
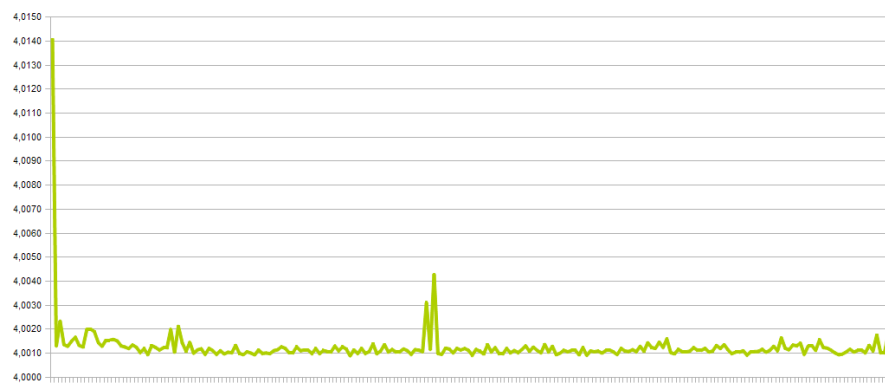
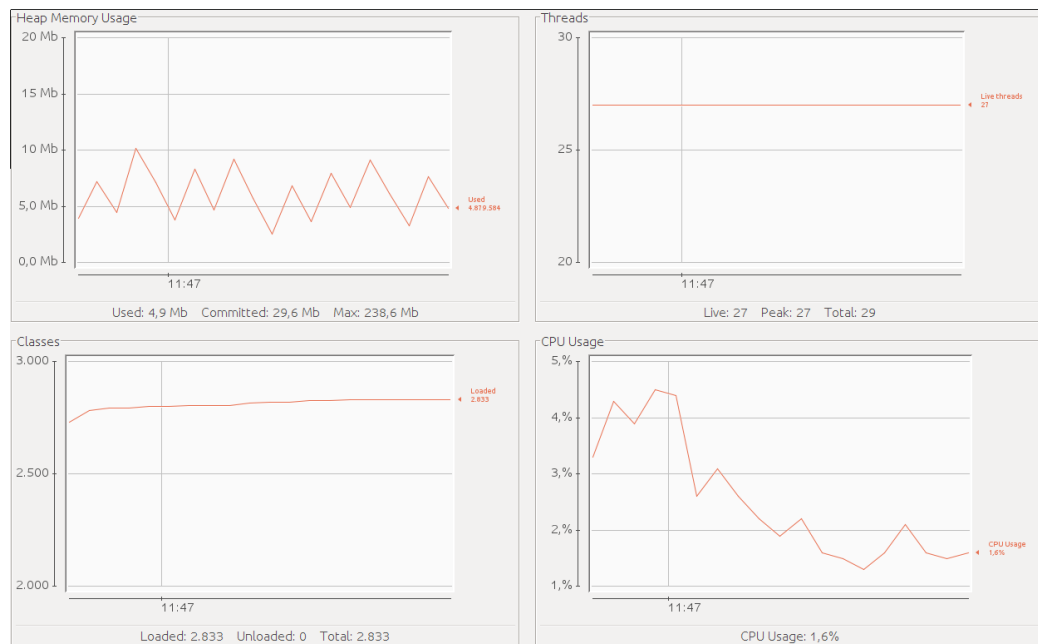
(a) *Prova 1.*(b) *Prova 2.*(c) *Prova 3.*(d) *Prova 4.*

Figura 11.16: Andamenti temporali per l'esecuzione con Server Service remoto

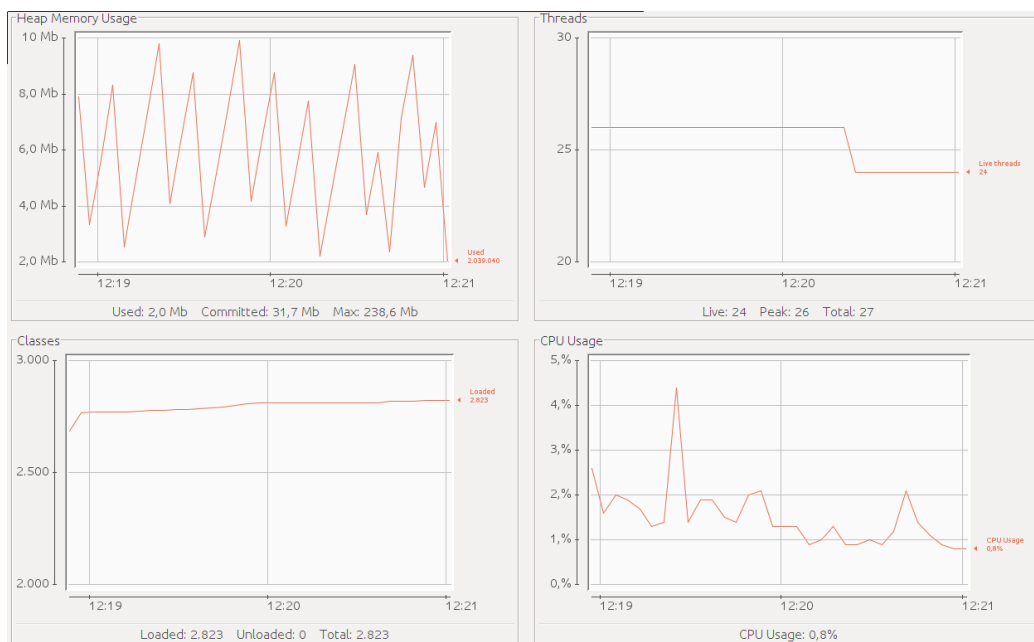


(a) Prova 1.

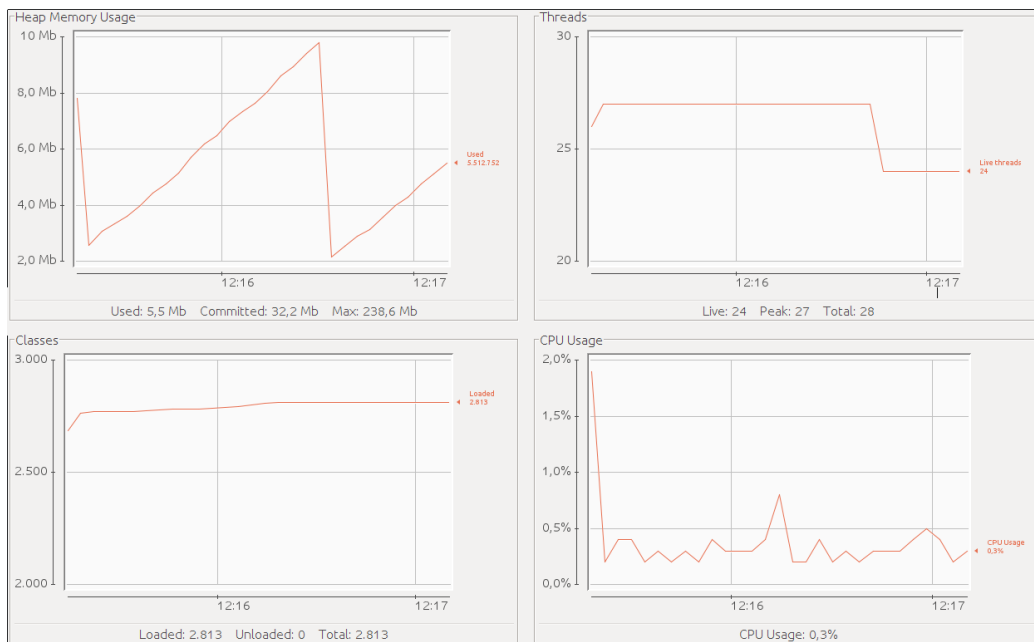


(b) Prova 2.

Figura 11.17: Risorse impiegate per l'esecuzione con Server Service remoto - Parte I



(a) Prova 3.



(b) Prova 4.

Figura 11.18: Risorse impiegate per l'esecuzione con Server Service - Parte II

Dai grafici temporali si notano dei tempi più lunghi dovuti alla fase di startup fino al raggiungimento a regime del programma. I tempi ottenuti con questa configurazione sono inferiori rispetto alla media delle 3 prove precedenti. Per quanto riguarda le risorse impiegate, come si può notare, sono congruenti con le precedenti prove.

## Conclusioni

Con questi quattro casi si è voluto verificare il funzionamento del sistema considerando le possibilità di collocamento degli attori. Il terzo caso rappresenta la situazione più vicina ad un'applicazione reale configurata con un Orchestratore (o una gerarchia) a cui si interfacciano più Client e più Service Proxy dislocati in macchine diverse (più Service Proxy possono stare anche su una stessa macchina a patto che sia assegnato loro una porta diversa per ciascuno). Il quarto caso invece è più simile alla configurazione remota del sistema in Java (come sarà presentato nella prossima sezione). Come da test effettuati, il sistema scala bene anche a fronte di richieste molto vicine temporalmente tra loro da parte di più Client, purché il Service Proxy che ospita il servizio richiesto non svolga calcoli che possano impiegare un elevato tempo di CPU e operazioni bloccanti.

## 11.2 L'orchestratore in Java

L'architettura dell'orchestratore, precedentemente sviluppato in Java, è mostrata in Figura 11.19.

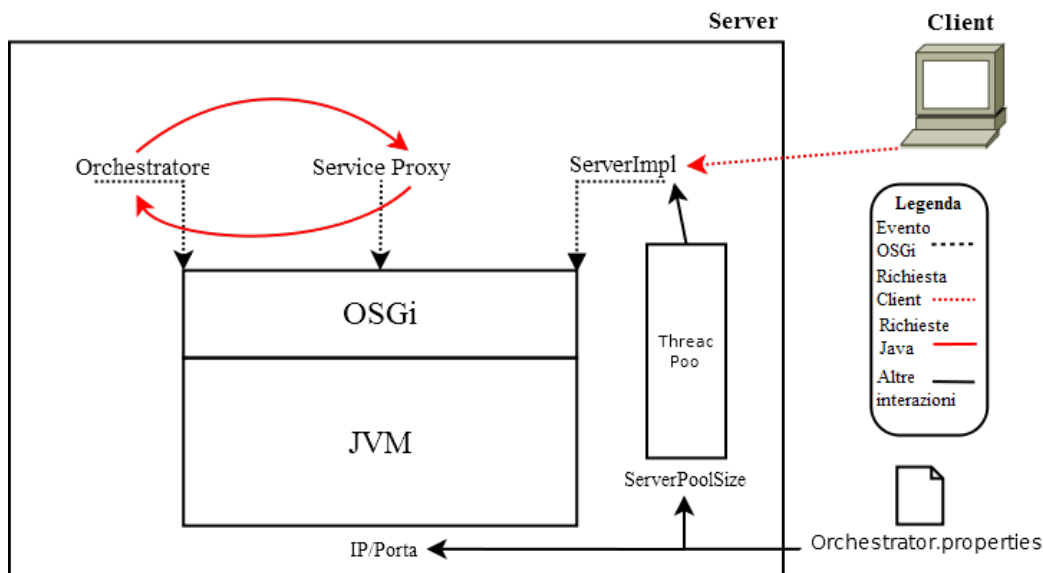


Figura 11.19: Architettura Orchestratore in Java

I servizi dell'orchestratore vengono posti sopra ad una specifica chiamata OSGi<sup>2</sup>. OSGi permette di sviluppare codice secondo un modello a componenti, caratteristica che manca a Java (o mal implementata). Sopperendo alle mancanze del linguaggio Java, OSGi permette di:

- Creare una piattaforma orientata ai servizi;
- Definire il concetto di “modulo” o bundle;
- Gestire in modo automatico le dipendenze dei moduli;
- Creare un sistema dinamico che consente la creazione, l'inserimento, l'avvio, l'arresto e la rimozione dei moduli a runtime;

Il framework, che implementa la specifica OSGi, che è stata usata per lo sviluppo di questa versione dell'orchestratore è Equinox<sup>3</sup>, sviluppato per Eclipse secondo le specifiche OSGi R4<sup>4</sup>.

La logica architetturale mostrata in Figura 11.19 si articola nei seguenti passi:

<sup>2</sup>[www.osgi.org](http://www.osgi.org)

<sup>3</sup>[www.eclipse.org/equinox](http://www.eclipse.org/equinox)

<sup>4</sup>[www.osgi.org/Specifications/HomePage?section=2#Release4](http://www.osgi.org/Specifications/HomePage?section=2#Release4)

- Il client esegue una richiesta al server (arco rosso tratteggiato);
- Il server, per mezzo della classe `ServerImpl`, preleva dal canale TCP unicamente la prima richiesta del client (che contiene il messaggio “when-start-request-received”) passandola all’orchestratore (dopo averlo avviato) attraverso il framework OSGi (archi nero tratteggiati). La fase di Deploy è l’unico momento in cui il sistema utilizza il framework;
- L’orchestratore inizia lo scambio di messaggi con i Service Proxy come già visto in precedenza nella versione Scala unicamente attraverso chiamate chiamate Java senza più coinvolgere il framework (archi rossi tratto continuo).

Il codice dei moduli principali vengono di seguito riportati:

Listing 11.7: Codice del client

```
public class Client
{
    public static void main(String[] args)
    {
        ...
        UtilTCP a, b; //Classe che implementa la comunicazione TCP
                     per il client
        a = new UtilTCP("DEPLOY;Esempio;C:/ServiceProxy.xml\n",
                       8000); //Invio messaggio DEPLOY lista Service Proxy
        a.sendTCP();
        ...
        b = new UtilTCP("RUN;SP.1;C\n", 8001); //Invio messaggio
        RUN su porta 8001
        b.sendTCP();
        ...
        for(int i = 0; i < #Thread; i++)
        {
            (new myThread(false, i)).start();
            Thread.sleep(xXx) //Pausa di xXx ms tra una richiesta e
                             la seguente
        }
    }

    private static class myThread extends Thread
    {
        UtilTCP c;
        static int port = 9100;
```

```
...
public void run()
{
    c = new
        UtilTCP("EVENTNAME='when-start-request-received';
                BSNAME='Start';BSINSTANCE='0';COMPOSITIONSESSIONID
                ='Esempio.1'###userId,xXx;serviceId,SP1\n",
                port++);
    ...
    c.sendTCP();
}
}
```

Il Client crea due nuove istanze di UtilTCP, classe che implementa la comunicazione TCP sul canale, depositando il messaggio. La prima istanza è utilizzata per inviare al Server il messaggio di Deployment, la seconda per avviare il Server. Un ciclo for avvia #Thread Thread che inviano al Server ogni xXx ms il messaggio con la richiesta iniziale “when-start-request-received”.

Di seguito viene presentata la classe ServerImpl:

Listing 11.8: Codice del ServerImpl

```
public class ServerImpl extends Thread
{
    Orchestrator orch;
    String ipAddr;
    int port;
    static ThreadPoolExecutor pool;
    ...
    public void run()
    {
        ...
        try
        {
            while(!this.isInterrupted())
            {
                clientSocket = socket.accept();
                pool.execute(new
                    TCPConnectionHandler(clientSocket, orch));
            }
            socket.close();
        }
    }
}
```

```
    }  
}
```

La classe `ServerImpl`, come detto precedentemente, si occupa di prelevare dal canale la prima richiesta del client per poi inviarla all'orchestratore. Data l'istanza dell'orchestratore, l'indirizzo IP, la porta e il numero di Thread da mandare in esecuzione nel Thread Pool, il Server richiama la classe `TCPConnectionHandler`, passando il riferimento al socket del client e l'istanza dell'orchestratore, che fa il parsing del messaggio ricevuto e si comporta di conseguenza.

Di seguito viene presentata la classe `TCPConnectionHandler`:

Listing 11.9: Codice del `TCPConnectionHandler`

```
public class TCPConnectionHandler implements Runnable  
{  
    ...  
    public void run()  
    {  
        ...  
        if(msg.startsWith("EVENT"))  
        {  
            clientSocket.close();  
            notifyEventParser(msg);  
            orch.notifyEvent(eventId, bsName, bsInstance,  
                             compositionSessionID, sp);  
        }  
  
        else if(msg.startsWith("DEPLOY"))  
        {  
            clientSocket.close();  
            deployParser(msg);  
            orch.deployService(deployServiceName,  
                              deployServiceFile);  
        }  
  
        else if(msg.startsWith("RUN"))  
        {  
            clientSocket.close();  
            runParser(msg);  
            orch.run_SC_in_O(rscId, scType);  
        }  
    }  
}
```

```
        ...  
    }  
}
```

Questa classe si occupa di fare il parsing del messaggio passato dal Server per poter invocare il giusto metodo nell'orchestratore demandandone l'esecuzione. In caso di DEPLOY invoca il metodo `deployService` dell'orchestratore, in caso di EVENT, viene richiamato il metodo `notifyEvent`.

Di seguito viene presentato il codice dell'Orchestratore:

Listing 11.10: Codice dell'Orchestratore

```
public class OrchestratorImpl implements Orchestrator  
{  
    ...  
    public OrchestratorImpl() { ... }  
  
    public int deployService(String scName, String gsdFile) { ... }  
  
    public int run_SC_in_O(String rscId, String type) { ... }  
  
    public int notifyEvent(String eventName, String baseServiceID,  
        String baseServiceInstanceID, String compositionSessionID,  
        SessionProperty[] sessionProperties)  
    {  
        (new MyNotifyEvent(eventName, baseServiceID + "." +  
            baseServiceInstanceID, compositionSessionID,  
            sessionProperties)).start();  
        return 0;  
    }  
    ...  
    private class MyNotifyEvent extends Thread  
    {  
        private String eventId, senderBsId, rscId, scName;  
        private SessionProperty[] sp;  
        private String tmpBSName=null;  
  
        MyNotifyEvent(String myeventId, String mysenderBsId,  
            String myrscId, SessionProperty[] mysp)  
        {  
            this.eventId = myeventId;  
            this.senderBsId = mysenderBsId;  
            this.rscId = myrscId;
```

```

        this.sp = mysp;
    }
}

public void run()
{
    ...
    if(eventId.equals("when-start-request-received"))
    {
        for(int i = 0; i < sp.length; i++)
        {
            sp[i].setPropertyName(sp[i].getPropertyName().
                replace('_', '.'));
        }
        System.out.println(this.rscId + ";Start;" +
            System.nanoTime());
    }
    if(tmpEvent.getArc(0).getNextAction().
        equals("finalEvent"))
    {
        ...
        System.out.println(this.rscId + ";Final;" +
            System.nanoTime());
        ...
    }
    else
    {
        Arc tmpArc= tmpEvent.getArc(i); //Prossimo elemento
            nella catena di servizi
        nextBsId=tmpArc.getNextBsId(); //Prossimo SP ID
        nextAction=tmpArc.getNextAction(); //Prossima Azione
        ... //Aggiornamento Properties
        tmpBSName=nextBsId.substring(0, nextBsId.indexOf("."));
            //Prossimo endpoint SP
        ...
        nextBS.invokeAction(nextAction, rscId,
            Integer.valueOf(nextBsId.substring(nextBsId.
                indexOf(".") + 1, nextBsId.length())), newSp);
    }
}
}

```

Il metodo `notifyEvent` istanzia e avvia un nuovo Thread della classe `My-`

NotifyEvent passando i parametri ricevuti attraverso il messaggio inviato. Questo Thread imposta le variabili locali che rappresentano le componenti del messaggio. Il metodo run specifica il comportamento da tenere nei tre casi previsti che possono essere contenuti in un messaggio di tipo EVENT, cioè il caso iniziale “when-start-request-received”, il caso finale “finalEvent” o il caso in cui non sia uno dei precedenti. Nel caso di messaggio iniziale, viene sistemata la sintassi delle properties e viene stampato il messaggio di avvio del SP. Nel caso di messaggio finale, viene stampato il messaggio di fine dell'esecuzione. Nel caso che non sia né un messaggio iniziale né uno finale, viene prelevato il prossimo elemento presente nella catena dei servizi, l'ID del SP, la prossima azione che deve compiere. Vengono poi aggiornate le Properties e prelevato il prossimo endpoint disponibile; infine viene inviato il messaggio con invokeAction contenente i nuovi parametri.

Per completezza viene ora presentato il codice di un generico Service Proxy:

Listing 11.11: Codice del TCPConnectionHandler

```
public class GMailMonitor implements BSInterface
{
    ...
    public int invokeAction(String actionName, String
        CompositionSessionID, int baseServiceInstance,
        SessionProperty[] properties)
    {
        (new GMail(actionName,
            CompositionSessionID, baseServiceInstance, properties)).
            start();
        return 0;
    }

    private class GMail extends Thread
    {
        public void run()
        {
            ...
            Thread.sleep(xXx);
            orch.notifyEvent("targetMailReceived", "GMailMonitor",
                ""+bsInstance, "Esempio.1", props);
        }
    }
}
```

Il Service Proxy definisce un proprio metodo `invokeAction` andando ad avviare un Thread che tiene nel proprio metodo `run` uno `sleep` che simula un certo tempo di processing e l'invio del `notifyEvent` all'orchestratore con il relativo messaggio.

Ora vengono proposti i risultati dei test con gli stessi parametri utilizzati per la versione Scala per un paragone qualitativo.

Tabella 11.9: Risultati relativi alle prove in locale

|                | Tempo tra<br>nuove<br>richieste [ms] | Tempo<br>esecuzione SP<br>[ms] | Media<br>Java[ms] | Media<br>Scala[ms] | Differenza[ms] |
|----------------|--------------------------------------|--------------------------------|-------------------|--------------------|----------------|
| <b>Prova 1</b> | 100                                  | 100                            | 984,3744          | 201,8358           | 782,5386       |
| <b>Prova 2</b> | 50                                   | 0                              | 9,87153           | 001,29031          | 8,58122        |
| <b>Prova 3</b> | 100                                  | 50                             | 521,9915          | 101,6028           | 420,3887       |
| <b>Prova 4</b> | 1000                                 | 2000                           | 20011,6           | 4923,9             | 15087,7        |

La Tabella 11.9 presenta i risultati relativi alle prove svolte con l'Orchestratore in locale. Mettendo a confronto i tempi medi della versione Java e Scala, si può osservare il netto vantaggio nei tempi di esecuzione di Scala, compresa la Prova 4 (che è il caso sfavorevole per un applicazione Scala come discusso in precedenza).

Di seguito vengono presentati i grafici con l'andamento dei tempi e delle risorse impiegate relative alle 4 prove.

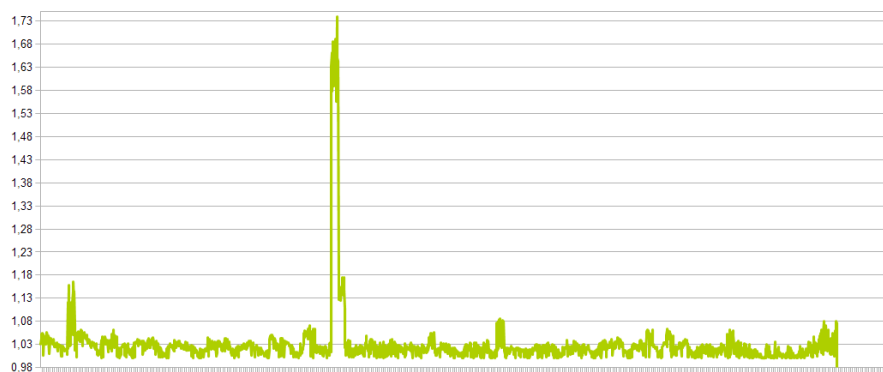
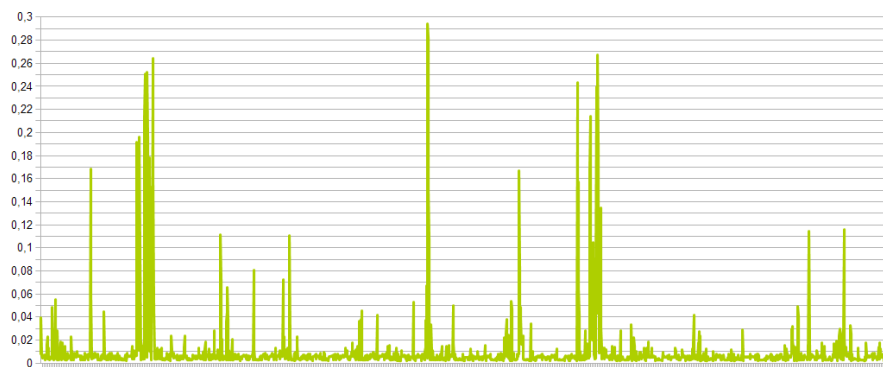
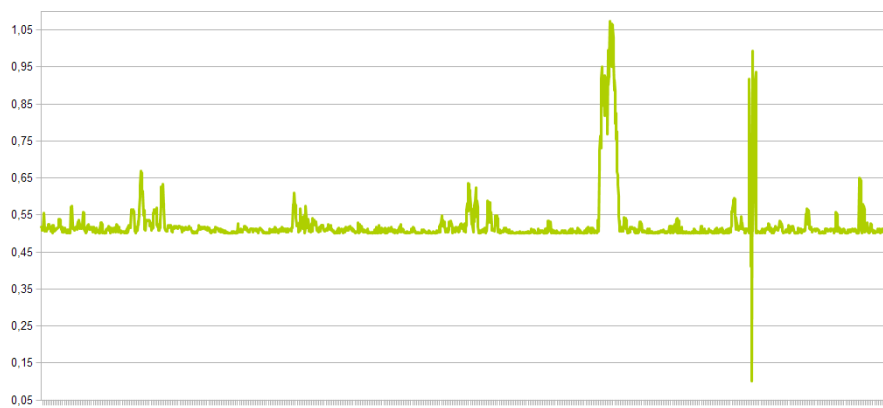
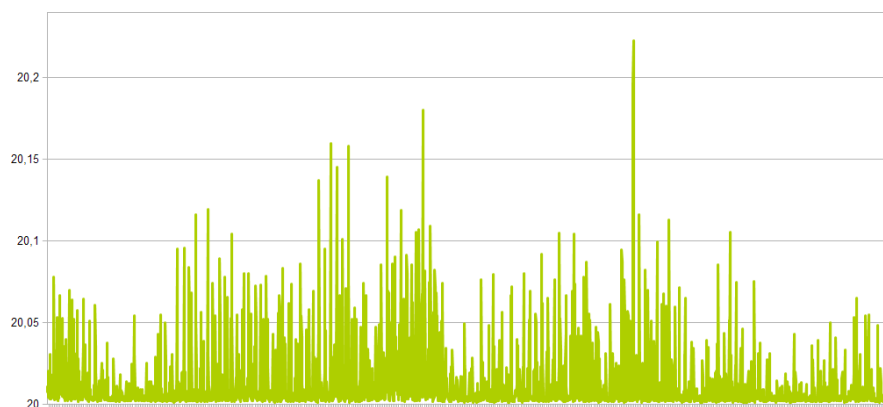
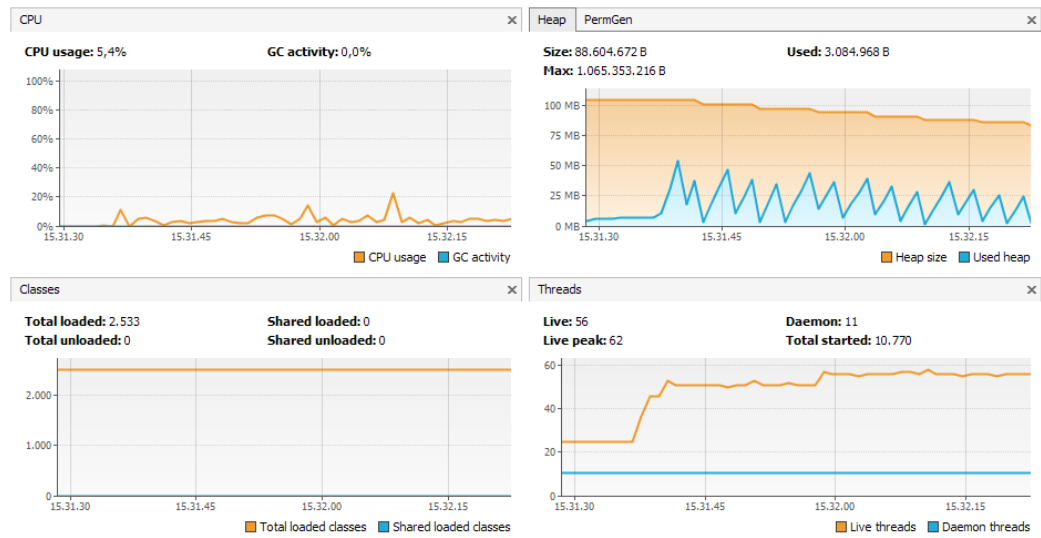
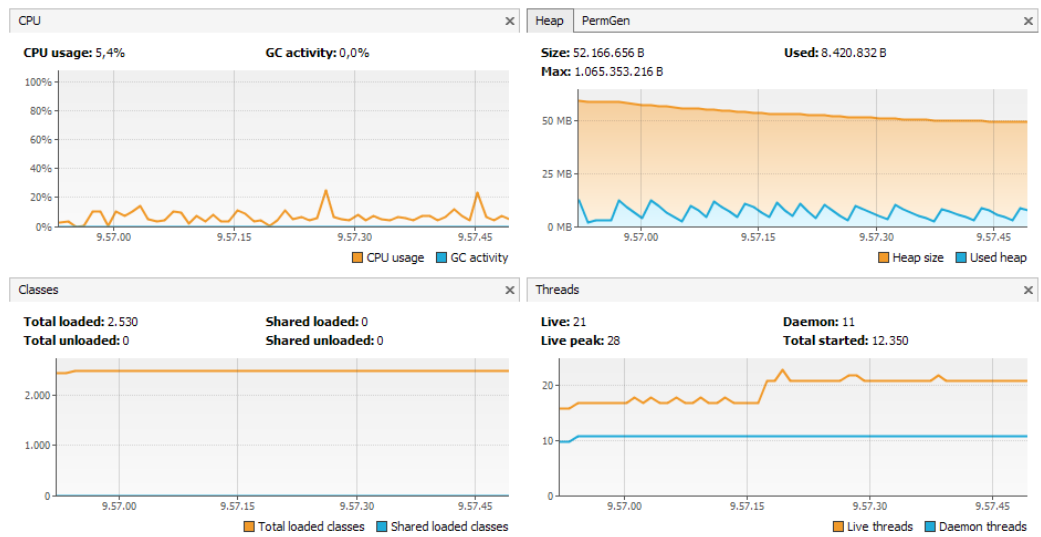
(a) *Prova 1.*(b) *Prova 2.*(c) *Prova 3.*(d) *Prova 4.*

Figura 11.20: Andamenti temporali per l'esecuzione in locale



(a) Prova 1.



(b) Prova 2.

Figura 11.21: Risorse impiegate per l'esecuzione in locale - Parte I

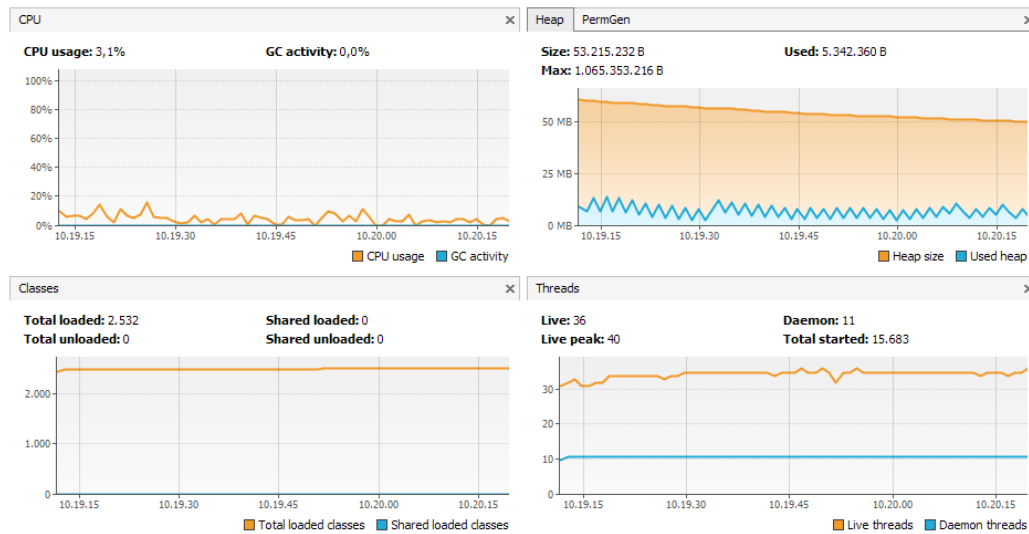
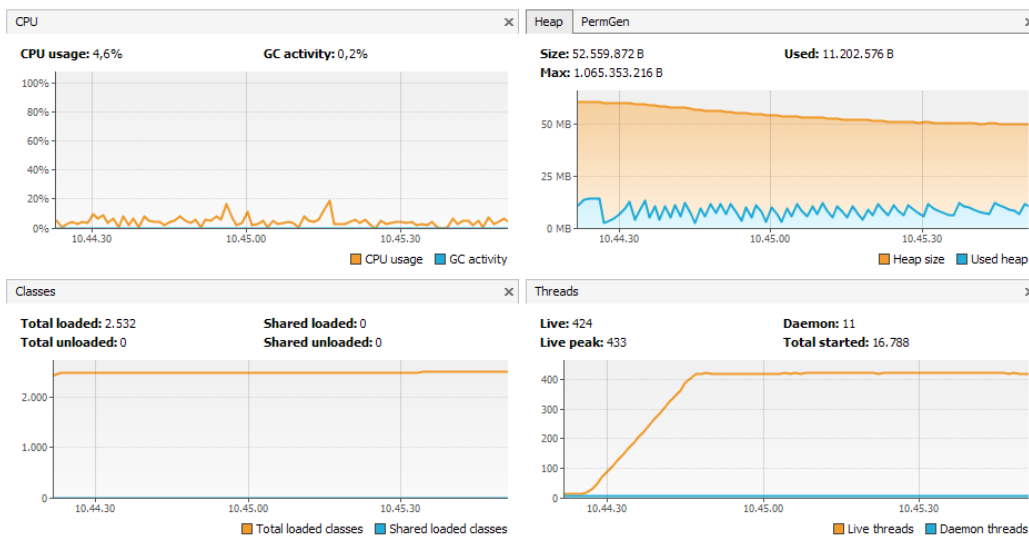
(a) *Prova 3.*(b) *Prova 4.*

Figura 11.22: Risorse impiegate per l'esecuzione in locale - Parte II

Confrontando le performance dell'Orchestratore Java con quello della versione Scala di Figure 11.3 e 11.4 a pagina 122 si può notare come il consumo di CPU rimanga pressoché invariato mentre la dimensione dell'Heap e il suo utilizzo è quasi raddoppiata. Per quanto riguarda il numero di Thread in esecuzione si può confermare l'alto numero presente nel sistema a favore dell'architettura proposta da Scala. Bisogna far notare però che l'esecuzione del Client sulla stessa macchina in cui gira il Server comporta un ovvio aumento

del numero di Thread e delle risorse impiegate, quindi per un confronto più realistico, si vedano i seguenti risultati:

Vengono ora proposti i risultati relativi all'esecuzione remota del Client e dell'Orchestratore.

Tabella 11.10: Risultati relativi alle prove in remoto

|                | Tempo tra<br>nuove<br>richieste [ms] | Tempo<br>esecuzione SP<br>[ms] | Media<br>Java[ms] | Media<br>Scala[ms] | Differenza[ms] |
|----------------|--------------------------------------|--------------------------------|-------------------|--------------------|----------------|
| <b>Prova 1</b> | 100                                  | 100                            | 1001,6754         | 200,76675          | 800,90865      |
| <b>Prova 2</b> | 50                                   | 0                              | 0,97274           | 0,51730            | 0,45544        |
| <b>Prova 3</b> | 100                                  | 50                             | 501,6776          | 100,71181          | 400,96579      |
| <b>Prova 4</b> | 1000                                 | 2000                           | 20001,774         | 4001,260           | 16000,514      |

Dai risultati emerge, anche in questa situazione, la superiorità dell'applicazione in Scala tranne nel secondo caso il quale perde qualche ms rispetto alla versione Java. La piccola differenza di tempo tra le due versioni è dovuta al fatto che, considerando un tempo di esecuzione nullo per un Service Proxy, Java esegue una comunicazione diretta e quindi è più rapido nel servire la prossima richiesta mentre la comunicazione in Scala deve passare attraverso l'infrastruttura che ne permette l'esecuzione ritardando di pochi ms il tempo di risposta.

Di seguito vengono presentati i grafici delle risorse impiegate dal sistema.

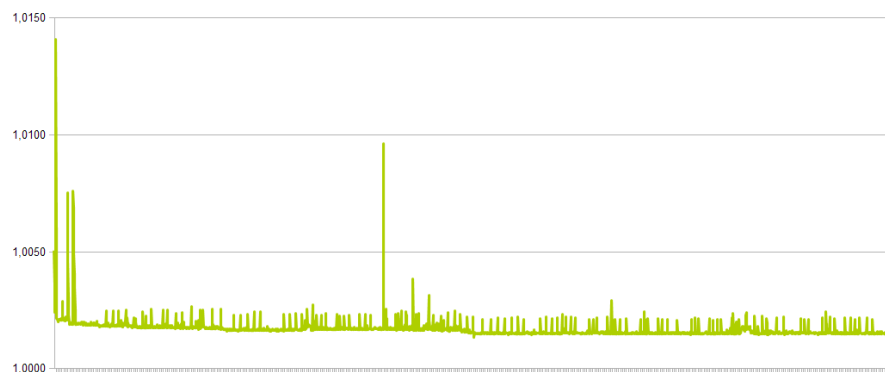
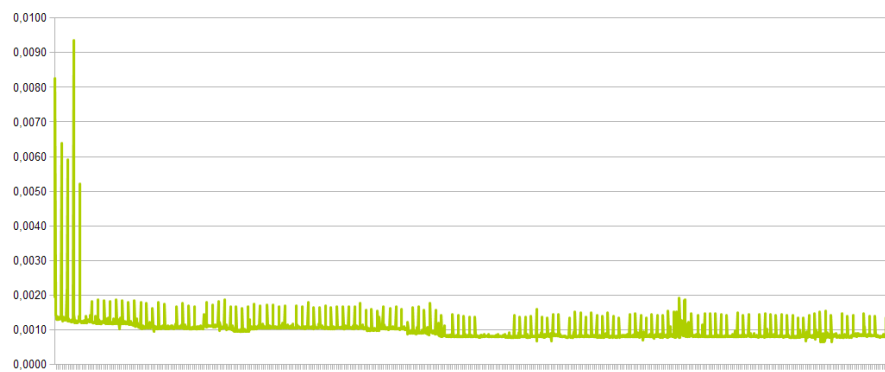
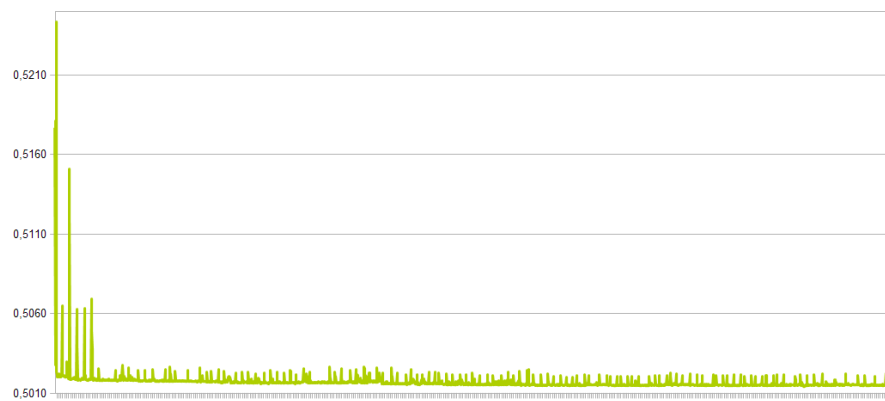
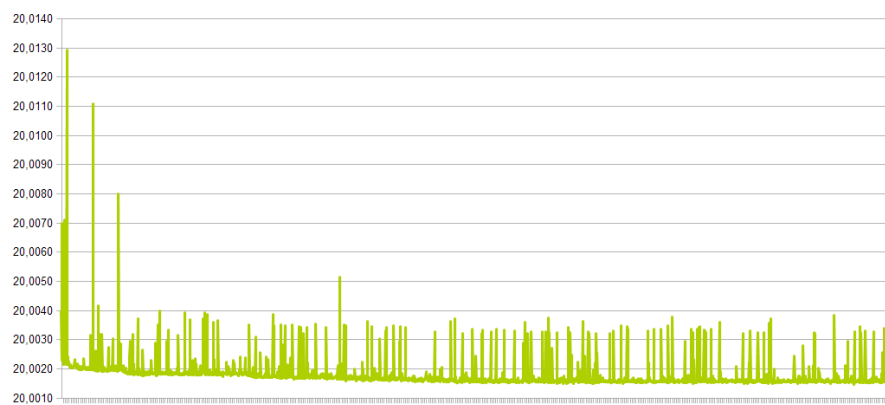
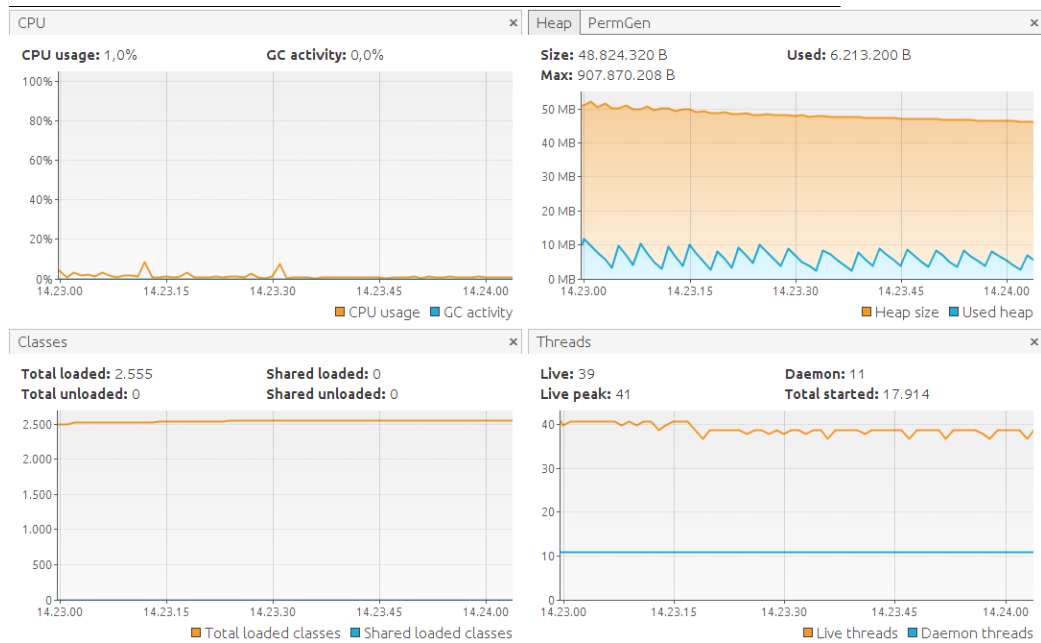
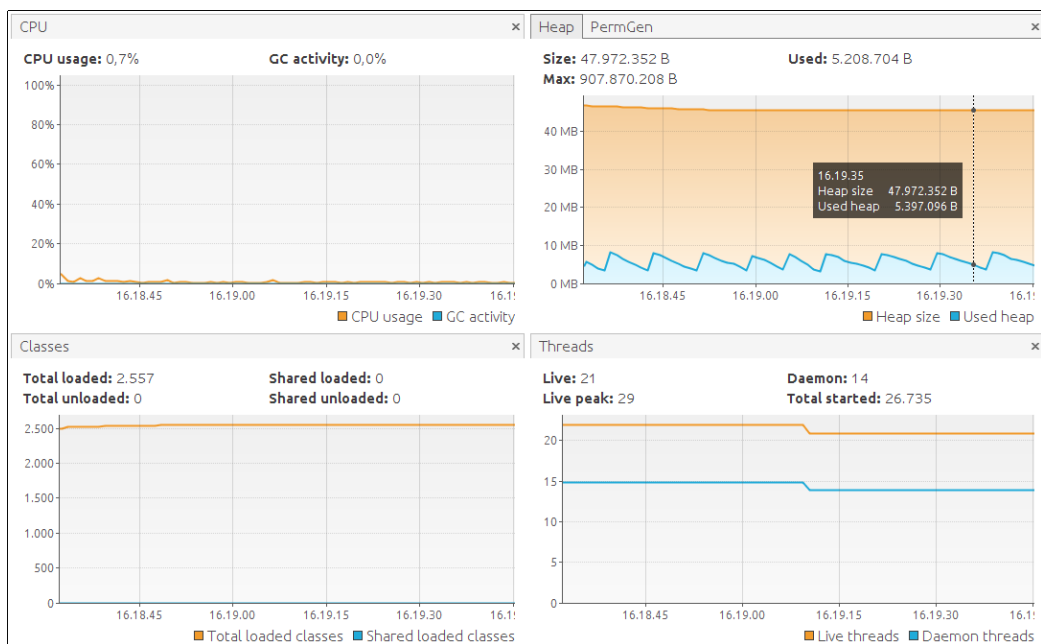
(a) *Prova 1.*(b) *Prova 2.*(c) *Prova 3.*(d) *Prova 4.*

Figura 11.23: Andamenti temporali per l'esecuzione remota

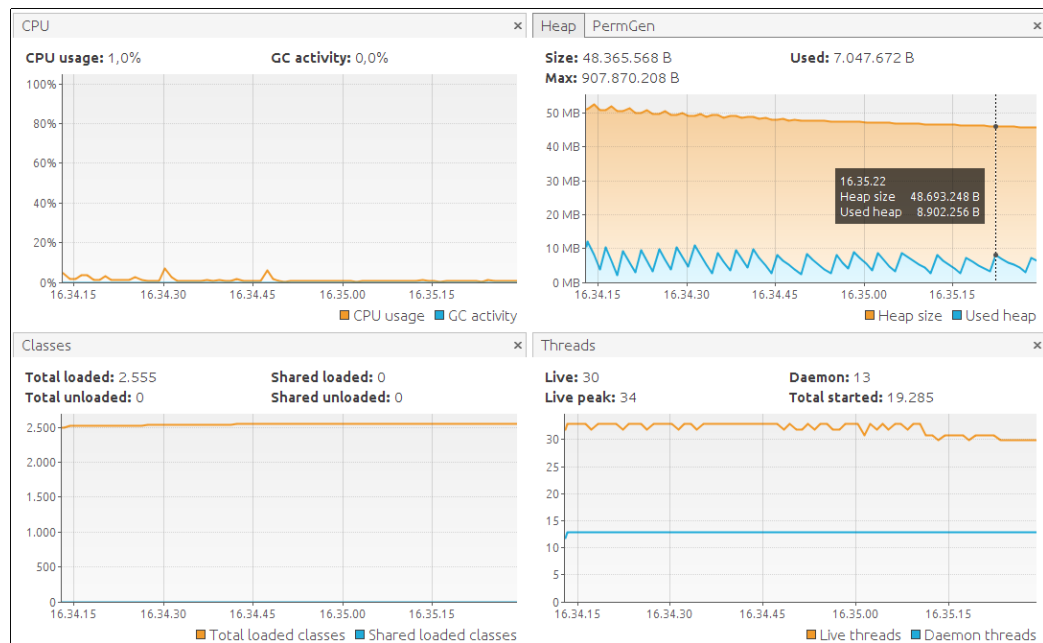


(a) Prova 1.

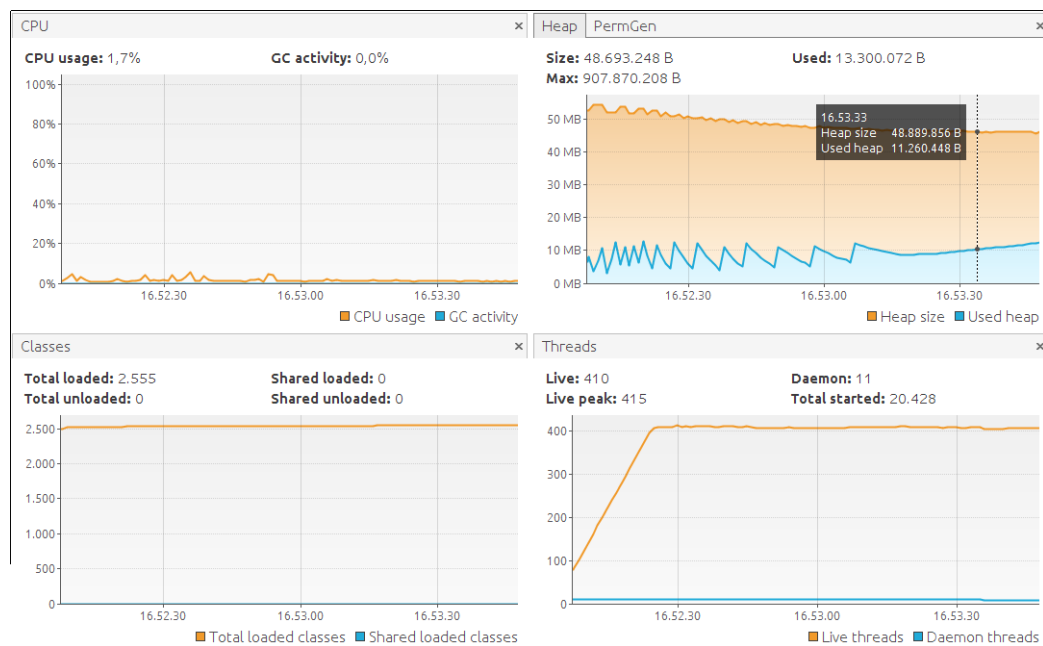


(b) Prova 2.

Figura 11.24: Risorse impiegate per l'esecuzione remota - Parte I



(a) Prova 3.



(b) Prova 4.

Figura 11.25: Risorse impiegate per l'esecuzione remota - Parte II

Confrontando i grafici di Scala di Figure 11.7 e 11.8 a pagina 130 si vede una netta differenza nell'uso dei Thread. Per quanto riguarda l'Heap, a

fronte di una dimensione maggiore da parte di Java, la quantità occupata è paragonabile a quella di Scala. Nella Prova 2, dove la versione Java ha fatto meglio dal punto di vista dei tempi, il numero di Thread impiegati da entrambe le versioni si equivalgono.

Un problema che si può notare dalle occupazioni della CPU dei vari metodi, è che il tempo impiegato dall'orchestratore è dovuto per la maggior parte dal lavoro che Thread Pool impiega nella gestione delle code.

# Parte VI

## Conclusioni



# Capitolo 12

## Conclusioni e Sviluppi futuri

In questa Tesi sono state presentate le caratteristiche e le funzionalità dei servizi compositi, in particolare, sono stati affrontati i problemi relativi all'inefficienza del modello a Thread che ha condotto a questo studio sull'impiego degli Attori del linguaggio Scala nello scambio di messaggi fra applicazioni, evidenziando i vantaggi che ne derivano, e le debolezze scaturite dai limiti in determinate situazioni operative.

Nella prima parte si è visto cos'è un servizio composito, quali sono le sue componenti principali e come sono messe in relazione tra di loro al fine di creare un servizio più complesso. Sono stati evidenziati i vantaggi e le problematiche dell'impiego dei servizi compositi, dal modello multi-Threading ai requisiti di un Web Server, dai diversi modelli di Thread alle architetture di un Server.

Nella seconda parte di questo lavoro, è stato affrontato il problema di mantenere basso il numero di Thread in esecuzione in un'applicazione client/server. Cercando di sopperire alle mancanze del modello Thread di Java, è stato analizzato il linguaggio di programmazione Scala che permette un modello Threadless basato sull'impiego di entità più leggere quali sono gli Attori. In particolare è stato analizzato il concetto di continuation, che ancora manca nella versione 7 di Java da poco rilasciata, con la quale è possibile salvare il contesto di un attore per poter operare in modo asincrono in una chiamata che, diversamente, risulterebbe bloccante. Si è visto che un programma che entra in un blocco di codice CPS per effettuare una chiamata, il contesto del processo che invoca viene salvato in una struttura dati e inviato nella chiamata stessa. Questo ha permesso di poter continuare la normale esecuzione dell'applicazione senza attendere nella chiamata, altrimenti bloccante, il risultato dell'invocazione, creando così un sistema di comunicazione fra processi asincrono.

Sono stati presi in considerazione anche framework e architetture in Java,

come Netty e il pacchetto NIO, per implementare un diverso sistema di comunicazione asincrono che non contribuiscono, comunque, allo scopo di mantenere controllato l'utilizzo di Thread, ma rappresentano una buona soluzione per applicazioni che vogliono avere solo l'asincronismo come requisito di progettazione.

In questo lavoro di tesi è stato sviluppato un porting di un sistema di orchestrazione precedentemente sviluppato in Java. Il porting è risultato semplice e veloce una volta appreso il linguaggio, grazie anche al supporto e la possibilità di importare librerie Java in un programma Scala. Lo sforzo non indifferente che richiede l'avvicinarsi ad un linguaggio che, per buona parte del suo codice, è di natura funzionale, è presto ripagato dai risultati ottenuti dai test effettuati. A parità di condizioni di lavoro, i risultati ottenuti dall'orchestratore in Scala (come da analisi prestazionale), presenta un miglior tempo di risposta e dei consumi relativamente più bassi rispetto alla versione in Java. Esistono, tuttavia, ampi margini di miglioramento nelle prestazioni se si sfrutta un diverso algoritmo di scheduling del Thread Pool rispetto al work-stealing implementato nella JSR166Y utilizzato nella libreria standard di Scala.

In futuro si può pensare di testare il comportamento di AKKA<sup>1</sup>, framework per Scala che sfrutta una diversa implementazione del modello ad eventi basato sugli attori presente nella libreria di Scala.

Un ulteriore margine di sviluppo futuro potrebbe portare ad eseguire una continuation manuale più "mirata" piuttosto che lasciata in gestione "automatica" alla funzione di react. Si può pensare anche di unire questa funzionalità alle tecniche di Asynchronous I/O viste nel Capitolo 8.

---

<sup>1</sup>akka.io

# Bibliografia

- [1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, “Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming,” in *In Proceedings of the 2002 Usenix ATC*, 2002.
- [2] G. A. Agha, “Actors: A model of concurrent computation in distributed systems (parallel processing, semantics, open, programming languages, artificial intelligence),” Ph.D. dissertation, University of Michigan, 1985.
- [3] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG*, 1st ed. Prentice Hall, 1996.
- [4] G. Banga and P. Drusche, “Resource containers: A new facility for resource management in server systems - typical models for high-performance servers,” OSDI’99, 1999. [Online]. Available: [http://www.usenix.org/events/osdi99/full\\_papers/banga/banga\\_html/node3.html](http://www.usenix.org/events/osdi99/full_papers/banga/banga_html/node3.html)
- [5] A. Begel, J. Macdonald, and M. Shilman, “Picothreads: Lightweight threads in java,” Tech. Rep., 2000.
- [6] R. von Behren, J. Condit, and E. Brewer, “Why events are a bad idea,” in *HotOS*, 2003.
- [7] V. Beltran, D. Carrera, J. Torres, and E. Ayguade, “Evaluating the scalability of java event-driven web servers,” in *Proc. Int. Conf. Parallel Processing ICPP 2004*, 2004, pp. 134–142.
- [8] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” in *Proc. th Annual Symp Foundations of Computer Science*, 1994, pp. 356–368.
- [9] D. Debasish, “Threadless concurrency on the jvm - aka scala actors,” 2006. [Online]. Available: <http://debasishg.blogspot.com/2006/11/threadless-concurrency-on-jvm-aka-scala.html>

- [10] M. Drewniok, M. Maresca, S. Rego, J. Sienel, and M. Stecca, “Experiments and performance evaluation of event driven mashups,” in *Computers and Communications, ISCC 2009. IEEE Symposium*, 2009, pp. 19–22.
- [11] D. Ghosh and S. Vinoski, “Scala and lift functional recipes for the web,” vol. 13, no. 3, pp. 88–92, 2009.
- [12] R. Gurram and et al, “A web based mashup platform for enterprise 2.0,” in *Proceedings of the 2008 international workshops on Web Information Systems Engineering*, 2008, pp. 144–151.
- [13] N. Hagen, “Scalable nio servers,” 2009. [Online]. Available: <http://www.znetdevelopment.com/blogs/2009/04/13/scalable-nio-servers-part-3-features/>
- [14] P. Haller and M. Odersky, “Scala actors: Unifying thread-based and event-based programming,” in *Theoretical Computer Science*, 2008, pp. 01–36.
- [15] ———, “Event-based programming without inversion of control,” in *Modular Programming Languages*, 2006, pp. 04–22.
- [16] P. Haller, M. Odersky, J. Vitek, and A. L. Murphy, “Actors that unify threads and events,” in *Proceedings of the 9th International Conference on Coordination Models and Languages*, 2007, pp. 171–190.
- [17] A. e. a. Koschmider, “Elucidating the mashup hype: Definition, challenges, methodical guide and tools for mashups,” in *2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, 2009.
- [18] C. Lauer and R. M. Needham, “On the duality of operating system structures.” in *Operating Systems Review*, 1979, pp. 03–19.
- [19] D. Lea, “A java fork/join framework,” 2000.
- [20] ———, *Concurrent Programming in Java Design Principles and Patterns*. Addison Wesley, 1999.
- [21] J.-S. Légaré, “Providing continuations in java via source code transformations,” in *Department of Computer Science, University of British Columbia*, 2006.

- [22] E. Maximilien and et al, “An online platform for web apis and service mashups,” in *Internet Computing*, 2008, pp. 32–43.
- [23] J. McBeath, “Java nio and scala continuations,” 2011. [Online]. Available: <http://jim-mcbeath.blogspot.com/2011/03/java-nio-and-scala-continuations.html>
- [24] —, “Delimited continuations,” 2011. [Online]. Available: <http://jim-mcbeath.blogspot.com/2010/08/delimited-continuations.html>
- [25] B. Newport, “Multi-core may be bad for java,” 2006. [Online]. Available: [http://www.devwebsphere.com/devwebsphere/2006/11/multicore\\_may\\_b.html](http://www.devwebsphere.com/devwebsphere/2006/11/multicore_may_b.html)
- [26] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 1st ed. Artima Inc, 2008.
- [27] T. Ogasawara, “Scalability limitations when running a java web server on a chip multiprocessor,” in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10)*. ACM, New York, NY, USA, 2010, p. 10.
- [28] J. Ousterhout, “Why threads are a bad idea (for most purposes),” in *USENIX Technical Conference (Invited Talk)*, Austin, TX, 2003.
- [29] S. Pellegrini, “Scala and erlang, the actor based concurrency model in the multi-core era,” 2008, pp. 01–23.
- [30] S. Pietschmann and et al, “Application composition at the presentation layer: Alternatives and open issues,” in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services (iiWAS 2010)*, ACM, 2010.
- [31] D. Pollak, “Implementing web sites with scala and lift,” in *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, 2010.
- [32] G. Roth, “Architecture of a highly scalable nio-based server,” 2007. [Online]. Available: <http://today.java.net/pub/a/today/2007/02/13/architecture-of-highly-scalable-nio-server.html>
- [33] S. Run, “More java actor frameworks compared,” 2009. [Online]. Available: <http://sujitpal.blogspot.com/2009/01/more-java-actor-frameworks-compared.html>

- [34] D. Schmidt, “Reactor: An object behavioral pattern for demultiplexing and dispatching handles for synchronous events,” 1994, pp. 01–11.
- [35] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Vol2: Patterns for Concurrent and Networked Objects*, 1st ed. Wiley, 2001.
- [36] M. Stecca and M. Maresca, “A comparison between web service and java message service technologies for event-driven mashup execution,” *International Journal of Web and Grid Services*, Inderscience Publishers, pp. 269–288, 2010.
- [37] ———, “Mashup patterns applied to e-government,” *Journal of Internet Business*, pp. 81–108, 2010.
- [38] S. Tilkov and S. Vinoski, “Node.js: Using javascript to build high-performance network programs,” in *IEEE Internet Computing*, 2010, pp. 80–83.
- [39] D. Wampler and A. Payne, *Programming Scala: Scalability = Functional Programming + Objects*, 1st ed. O’Reilly Media, 2009.
- [40] M. Welsh, D. Culler, and E. Brewer, “Seda: An architecture for well-conditioned, scalable internet services,” 2001.
- [41] Wikipedia, “Garbage collection,” 2009. [Online]. Available: [http://en.wikipedia.org/wiki/Garbage\\_collection\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science))
- [42] “Scala actor api.” [Online]. Available: [http://www.scala-lang.org/docu/files/actors-api/actors\\_api\\_guide.html](http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html)
- [43] “Scala 2.8.1 api library.” [Online]. Available: <http://www.scala-lang.org/api/current/index.html>