POLITECNICO DI MILANO
Corso di Laurea MAGISTRALE in Ingegneria Informatica

ECoWare
Coordinamento di attuatori in cicli di controllo multi-livello in applicazioni cloud

Dipartimento di Elettronica, Informazione
e Bioingegneria

Relatore: Prof. Sam Guinea
Correlatore: Dott. Ing. Filippo Seracini

Tesi di Laurea di:
Giovanni Quattrocchi, matricola 782710

Anno Accademico 2013-2014

Grazie a Sam Guinea, relatore di questa tesi, per l'aiuto fondamentale nella stesura, le possibilità che mi ha offerto e la stima che mi ha rivolto. Grazie ai miei genitori, Enia e Gianni, per avermi messo nelle condizioni di studiare con serenità (a parte certi pranzi troppo escandescenti). Grazie a Silvia per le correzioni grammaticali. E l'affetto con cui mi ha sostenuto. Grazie al correlatore Filippo Seracini, Ph.D a UCSD, per avermi fatto sentire a casa in un altro continente (anche se avrei evitato le domeniche notte in compagnia di JBoss1). E grazie al 'tuttologo' Dottor Bonassi per la revisione delle formule.

Giovanni

Sommario

La complessità dei servizi e delle applicazioni web e cloud, misurata in termini quantititativi e qualitativi, è negli anni costantemente aumentata a tal punto che la gestione manuale di queste risulta inefficiente e limitata. Nasce così l'esigenza di un approccio autonomico al ciclo di vita dell'applicazione che deve garantire robustezza rispetto al cambiamento dell'ambiente in cui viene eseguita; ad esempio, un aumento del volume di richieste ricevute deve corrispondere, se necessario, all'allocamento di nuove risorse.
Verrà presentato ECoWare, un framework per il controllo multi-livello autonomico di applicazioni costituito da quattro blocchi: monitoraggio, analisi, pianificazione ed esecuzione. Questo lavoro si concentra sull'ultima componente, l'esecutore, che attraverso il coordinamento distribuito di attuatori, esegue piani d'azione al fine di rendere effettive le strategie pianificate.
Saranno presentati alcuni esperimenti utilizzando un noto benchmark per applicazioni web, RUBiS, che simula un sito di aste online. Mostrerò come l'approccio multi-livello migliora fino al 42.5% lo stato dell'arte e come l'esecutore aggiunga un overhead contenuto rispetto all'esecuzione complessiva di una strategia.

A Enia, mia mamma

Indice

Sommario **3**

1 Introduzione **11**

2 Stato dell'arte **15**
2.1 Stato della ricerca 15
2.2 Lo stato dell'industria 21
2.2.1 Amazon Web Service 22
2.2.2 Google App Engine 26
2.2.3 DevOps: Chef & Puppet 26
2.2.4 Cloudify 28
2.2.5 Altri provider IaaS e PaaS 29

3 Definizione del problema di ricerca **31**
3.1 Runtime Management 31
3.2 ECoWare 32
3.2.1 Monitorragio 34
3.2.2 Analisi 36
3.2.3 Pianificatore 37
3.3 Esecutore 37
3.3.1 Oltre la Dicotomia Infrastruttura - Piattaforma 38
3.3.2 Obbiettivi 39
3.4 Un esempio 40

4 Architettura della Soluzione **43**
4.1 Anatomia di un'applicazione 43
4.2 Tier 45
4.2.1 Capability 46
4.2.2 Action 46
4.2.3 Parametri e precondition 47
4.2.4 Azioni Unsafe e Critical 47

4.3 Riassunto: il Meta-Modello 48
4.4 EADl: ECoWare App Description Language 48
4.4.1 Grammatica 48
4.4.2 EADl: un esempio 51
4.5 EPDl: ECoWare Plan Description Language 54
4.5.1 Grammatica 54
4.5.2 EPDl: un esempio 54
4.6 Processing model 56
4.6.1 L'architettura 57
4.6.2 L'esecuzione del piano sul Main Node 57
4.6.3 Fase 4: la coreografia 58
4.6.4 L'esecuzione del piano sui nodi 61

5 Implementazione **67**
5.1 XText ed i DSL 67
5.2 RabbitMQ ed il bus 69
5.3 L'implementazione del Main Node 71
5.4 L'implementazione dell'agente distribuito 76
5.5 Figura riassuntiva 79

6 Realizzazioni sperimentali e valutazione **83**
6.1 Il caso di studio 83
6.1.1 Il setup iniziale: private cloud 84
6.2 ECoWare on RUBiS 84
6.2.1 Monitoraggio ed Analisi 84
6.2.2 Planner e Perfomance Model 86
6.2.3 Executor 87
6.3 Valutazione del sistema 87
6.3.1 Esperimento 1 88
6.3.2 Esperimento 2 90
6.4 Valutazione del nuovo esecutore 92
6.4.1 Lo stack applicativo di RUBiS 93
6.4.2 Il nuovo setup: hybrid cloud 93
6.4.3 L'implementazione degli Actuator 95
6.4.4 L'implementazione dell'allocatore e deallocatore 97
6.4.5 La valutazione dell'esecutore 98

7 Valutazioni critiche e direzioni future **105**

Capitolo 1

Introduzione

La complessità dei servizi e delle applicazioni web e cloud, misurata in termini quantititativi e qualitativi, è negli anni costantemente aumentata a tal punto che la gestione manuale di queste risulta inefficiente e limitata [20, 13, 14, 6]. Nasce così l'esigenza di un approccio autonomico al ciclo di vita dell'applicazione che deve garantire robustezza rispetto al cambiamento dell'ambiente in cui viene eseguita; ad esempio, un aumento del volume di richieste ricevute deve corrispondere, se necessario, all'allocamento di nuove risorse. La gestione autonomica non si riduce però alla sola infrastruttura, possono essere necessari cambiamenti di parametri a livello applicativo o su un particolare servizio (middleware) installato. In altre parole il controllo dinamico di applicazioni deve puntare all'elasticità [12] considerata come la proprietà di soddisfare requisiti di qualità, di quantità e di costo e alla capacità di adattarsi al cambiamento degli stessi.

Gran parte delle tecniche di adattamento autonomico si fondano su cicli di controllo a blocchi, la cui interpretazione più riconosciuta è l'architettura MAPE [15] che si compone di quattro fasi: monitoraggio, analisi, pianificazione ed esecuzione. La componente di monitoraggio misura i parametri e le prestazioni dell'applicazione da cui l'analizzatore estrae della conoscenza. Il pianificatore, attingendo dall'analisi, sviluppa una strategia affinché i parametri dell'applicazione possano rispettare i requisiti e tendere all'ottimo. Infine l'esecutore, grazie ad attuatori installati sui nodi del sistema, concretizza la strategia con l'esecuzione di azioni. Attraverso questa architettura è quindi possibile valutare ciclicamente il contesto e le prestazioni del sistema e riconfigurarlo automaticamente al fine di ottimizzare le risorse disponibili in base ai requisiti. Come nota [16] tutto ciò deve avvenire in maniera trasparente rispetto alla logica dell'applicazione; sistemi di questo tipo devono

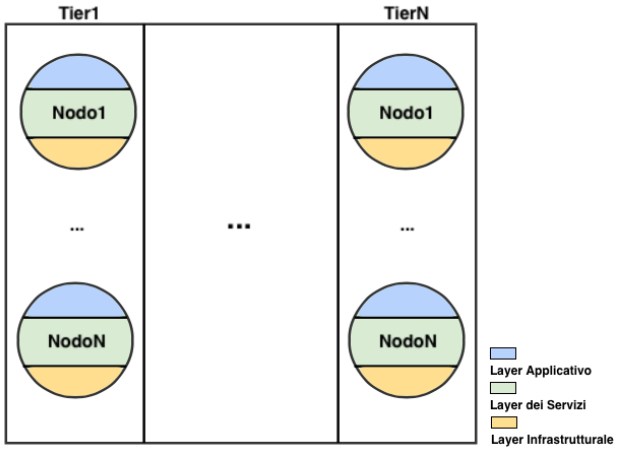


Figura 1.1: La struttura di un'applicazione

utilizzare tecniche di introspezione ed intercettamento, sensori ed attuatori che lo rendano riutilizzabile in contesti diversi.
A partire dall'architettura MAPE verrà presentato ECoWare, un framework per la gestione autonomica multi-livello di applicazioni. Lavori come [25, 26] considerano l'adattamento dipendente univocamente dal volume di richieste, il cui aumento può causare violazioni del SLA[1]; è stato però dimostrato [22, 24] che una visione di questo tipo è limitata ed è necessario considerare diversi aspetti dell'applicazione. Infatti discostamenti dalla qualità attesa del servizio si possono presentare, come sintomi, ad un livello ma in realtà la vera e propria causa del problema potrebbere su un altro; inoltre potrebbe essere più efficiente ed efficace intervenire, in maniera sincronizzata, su più livelli [4] in modo da creare una strategia complessiva che armonizzi l'adattamento in tutte le sue componenti. ECoWare è multi-livello in ciascuna componente poiché considera l'applicazione suddivisa ortogonalmente in layer e tier. Ingegneristicamente infatti un'applicazione si compone di più livelli, chiamati tier, costituiti ciascuno da una o più macchine[2] e ai quali corrisponde una funzionalità implementativa diversa come la gestione del database o la logica di business dell'applicazione. D'altra parte su ogni macchina si possono riconoscere tre diversi layer: il layer applicativo che riguarda il codice o i dati specifici dell'applicazione, il layer dei servizi dedicato ai middleware che caratterizzano il tier di appartenenza (ad esempio MySQL per il data tier) e il layer infrastrutturale (CPU, memoria,

[1]SLA o Service Level Agreement è il contratto tra provider di un servizio ed utente in cui si defiscono le soglie di qualità e prestazionali da rispettare.

[2]In questa tesi si userà il termina *macchina* per indicare indistintamente un server fisico o una macchina virtuale.

network, etc.). In Figura 1.1 si mostra questa suddivisione in tier e layer dell'applicazione.
Per la tesi ho lavorato alle componenti di monitoraggio e analisi, ho collaborato per tre mesi a San Diego con UCSD alla componente di pianificazione e alla valutazione del framework, ma il contributo centrale ed innovativo di questo lavoro riguarda l'esecutore. Diversi sistemi autonomici [8, 9, 10, 16] sono particolarmente sviluppati nelle componenti di monitoraggio, analisi e presentano linguaggi specifici per la pianificazione, ma relegano a dettaglio implementativo l'effettiva esecuzione delle strategie. In realtà il problema di coordinare attuatori distribuiti non è di semplice soluzione, soprattutto se si considera l'applicazione come bipartita in tier e layer, cosa che comporta dipendenze e quindi sincronizzazioni da rispettare tra le azioni da eseguire. Esiste una dipendenza quando i nodi di un tier per funzionare correttamente necessitano di informazioni di altri: ad esempio un loadbalancer deve conoscere quali server sono attivi per smistare le richieste. Infine l'esecutore si cala nel contesto del cloud offrendo un'interfaccia agnostica ed integrata con i provider IaaS.
Saranno presentati alcuni esperimenti utilizzando un noto benchmark per applicazioni web, RUBiS, che simula un sito di aste online. Mostrerò come l'approccio multi-livello migliora fino al 42.5% lo stato dell'arte e come l'esecutore aggiunga un overhead contenuto rispetto all'esecuzione complessiva di una strategia.

La tesi è strutturata nel modo seguente.
Nel capitolo due si illustra lo stato dell'arte del mondo accademico ed industriale. Nel capitolo tre si descrive il problema di ricerca e le motivazioni ed obbiettivi fondativi della costruzione dell'esecutore. Nel capitolo quattro si mostra l'architettura della soluzione e il modello dell'esecutore. Nel capitolo cinque si propone una implementazione della soluzione. Nel capitolo sei si esibiscono gli esperimenti effettuati sul sistema. Nelle conclusioni si espongono alcune valutazioni critiche e le prospettive future della ricerca.

Una pubblicazione basata su parte di questo lavoro è attualmente in attesa di approvazione in una conferenza internazionale.

Capitolo 2

Stato dell'arte

Il mio genio è nelle mie narici.

Friedrich Nietzsche

In questa sezione si analizzeranno alcuni lavori accademici incentrati sul *runtime management* di applicazioni web nel contesto del cloud. Inoltre, vista l'attualità degli argomenti e il grosso interesse pratico, si studieranno alcuni prodotti commerciali che aiuteranno a comprendere il lavoro svolto.

2.1 Stato della ricerca

L'esigenza di un approccio autonomico alla gestione di infrastrutture e piattaforme cloud emerge da diversi fattori tra cui i costi di mantenimento causati dalla gestione manuale [20, 13, 14], l'alta percentuale d'errore degli amministratori di sistema (20-50%) [6] e la crescente complessità quantitativa e qualitativa delle applicazione web e cloud moderne. In generale gli obbiettivi del controllo autonomico sono due: migliorare l'efficienza e il controllo della complessità. Runtime management significa automatizzazione del gestione dell'applicazione in tutto il suo ciclo di vita: il deploy iniziale delle risorse, l'installazione, la configurazione e l'aggiornamento di software, la scalabilità orizzontale e verticale ed il rilevamento di situazioni critiche e di fallimenti di sistema. Il fondemento teorico di questa visione è l'architettura MAPE [15] composta da un ciclo di controllo a blocchi che consiste nello spezzare la gestione autonomica in quattro fasi: il monitoraggio delle risorse, l'analisi dei dati ottenuti, la pianificazione di strategie e l'esecuzione di azioni che modificano concretamente i parametri e la configurazione del sistema. A volte si estende questa struttura con un quinto blocco: una base di conoscenza

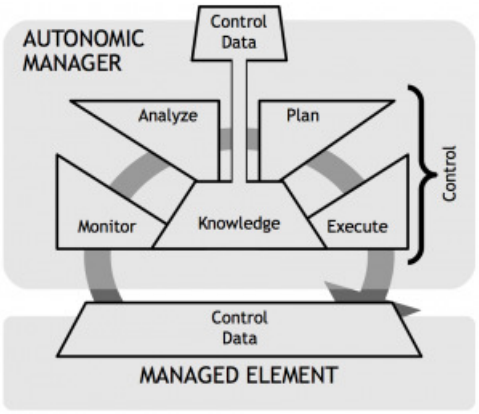


Figura 2.1: L'architettura MAPE o MAPE-K

(knowledge-base) da cui gli altri blocchi attingono informazioni (Figura 2.1). Attraverso questa architettura è possibile analizzare il contesto e l'evoluzione del sistema e, attraverso un modello, riconfigurarlo e riadattarlo in base ai cambiamenti; si parla anche di adattamento automatico o self-adaptation. I contributi della ricerca in questo campo sono molteplici. In [8] sfruttando l'architettura MAPE si propone la creazione di un linguaggio chiamato Stitch grazie al quale l'utente può creare sofisticate strategie di adattamento. Vengono definite tre classi di funzionalità: gli operatori, le tattiche e le strategie. Un operatore è l'unità più piccola di esecuzione ovvero una azione che modifica lo stato di un servizio o configura una risorsa. Una tattica (Figura 2.2a) è un insieme di operatori e l'elemento atomico di una strategia. Ogni tattica comprende:

- un insieme di operatori;
- una precondizione basata su i dati monitorati ed analizzati che ne determina l'applicabilità;
- la definizione degli effetti che si cercano di ottenere;
- un vettore contentente l'impatto sulle dimensioni del sistema.

Le strategie (Figura 2.2b) sono le vere e proprie unità di adattamento, contengono un insieme di tattiche precedute da delle precondizioni sull'intera strategia o sulla singola tattica da eseguire.
Il framework ASF (Adaptive Server Framework) [16], anch'esso con architettura MAPE, si fonda su alcuni principi che, generalizzando, possono valere per tutti i sistemi autonomici di adattamento:

```
module znn.tactics;
import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
import op "znn.operators.ArchOps" { ArchOps as Sys };

tactic switchToTextualMode () {
  condition {
    exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;
  }
  action {
    svrs = { select s : T.ServerT | !s.isTextualMode };
    for (T.ServerT s : svrs) {
      Sys.setTextualMode(s, true);
    }
  }
  effect {
    forall c:T.ClientT in M.components | c.expRspTime ≤ M.MAX_RSPTIME;
    forall s:T.ServerT in M.components | s.isTextualMode;
  }
}
```

(a) Una tattica

```
module znn.strategies;
import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
import model "ZnnEnv.acme" { ZnnEnv as E };
import lib "znn.tactics";
import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util

define boolean styleApplies = Model.hasType(M,"ClientT")//.."ServerT";
define boolean cViolation =
    exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;

strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
  define boolean hiLatency =
    exists k:T.HttpConnT in M.connectors | k.latency > M.MAX_LATENCY;
  define boolean hiLoad =
    exists s:T.ServerT in M.components | s.load > M.MAX_UTIL;

  t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
    t1a: (success) -> done ;
  }
  t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t2a: (!hiLoad) -> done ;
    t2b: (!success) -> do [1] t1 ;
  }
  t3: (default) -> fail;
}
```

(b) Una strategia

Figura 2.2: Stitch, un linguaggio per l'adattamento automatico

- il sistema deve possedere un modello che conoscendo le misure dell'infrastruttura deve produrre parametri ottimi per la configurazione dell'applicazione;
- il sistema autonomico deve essere separato dala logica di business dei componenti dell'applicazione, si deve quindi fare uso di sistemi di introspezione, intercettamento e sensori;
- l'overhead generato dal sistema deve essere valutato rispetto alla velocità con cui l'ambiente cambia;
- in tutti i sistemi di adattamento il transitorio è parte del problema.

Il quarto punto è di fondamentale importanza nell'implementazione di sistemi autonomici. Esiste infatti un intervallo di tempo in cui, dopo un ciclo di adattamento, i parametri monitorati non saranno o saranno solo parzialmente influenzati dalla strategia applicata. Ciò è causato dal fatto che durante

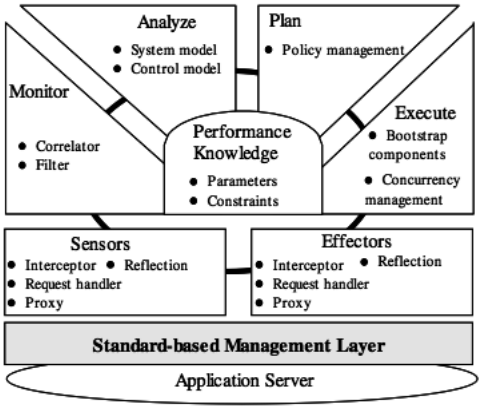


Figura 2.3: L'architettura MAPE-K del framework ASF

l'esecuzione di una strategia, i sensori continuano a monitorare il sistema che è in evoluzione e, inoltre, ciascuna delle azioni può presentare un periodo di transitorio prima che effettivamente vengano applicate le modifiche. La Figura 2.3 mostra l'instanza di architettura MAPE-K utilizzata in questo lavoro e svela alcuni dettagli di un framework di questo tipo.

SYBL (Simple Yet Beautiful Language) [9, 10] è un liguaggio per il controllo dell'elasticità nelle applicazioni cloud. Come nel caso di Stitch il linguaggio accede ai dati di monitoraggio e ne permette l'aggregazione al fine di costruire strategie. Un aspetto di interesse è la nozione di elasticità fornita dagli autori: non si intende infatti solo l'aspetto infrastrutturale della scalabità delle risorse [7, 21, 17], ma è considerata la misura della capacità di adattarsi al cambiamento. È quindi una proprietà multi-dimensionale (Figura 2.4) che non si limita solamente all'aspetto quantitativo ma anche al costo e alla qualità del servizio [12]. Se i requisiti coinvolgono diversi aspetti, anche l'adattamento è multi-livello. [9, 10] considerano l'applicazione suddivisa in componenti implementativi ciascuno dei quali, come si mostra in Figura 2.5, è associato ad uno o più servizi. Ad esempio il componente che fa riferimento alla gestione dei dati utilizza diverse tecnologie come Cassandra e HBase. Ciascun componente è quindi eseguito su più processi i quali sono distribuiti disomogeneamente nell'infrastruttura. I requisiti e l'adattamento sono definibili a diverse granularità: livello di applicazione, di componente o di codice. I primo è quello complessivo di tutta l'applicazione e quindi i dati sono aggregati a partire dalle varie componenti, il livello componente si riferisce ad una funzionalità logica dell'applicazione ed è il risultato dell'aggregazione dei processi che lo compongono. Il terzo livello di granularità permette di associare ad una porzione di codice requisiti e

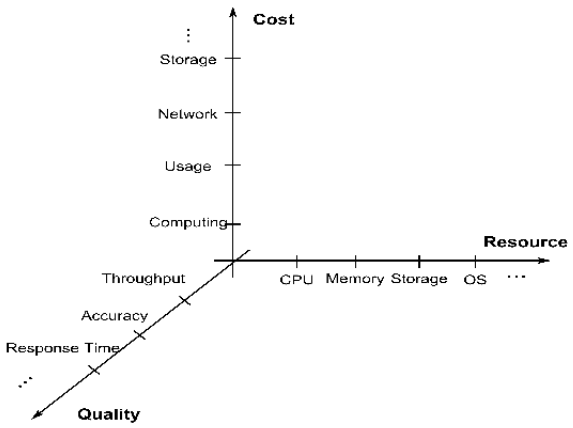


Figura 2.4: La multi-dimensionalità dell'elasticità

relative strategie in termini prestazionali e di costo. Il linguaggio SYBL permette quindi di esplicitare il livello di granularità a cui si vuole operare e inoltre dichiarare, attraverso opportune direttive, quali KPI monitorare. Tali KPI fanno riferimento a quanto monstrato in Figura 2.4, ad esempio *Quality.ResponseTime*. Inoltre, sempre nel contesto del livello di granularità dichiarato, attraverso direttive di analisi (o *constraint*) è possibile porre vincoli a tali KPI. Si può quindi imporre che a livello del componente dati il response time debba essere inferiore ad una determinata soglia e che il costo non sfori il budget previsto. Infine è possibile dichiarare strategie: in caso si verifichino determinate condizioni (ad esempio il response time supera la soglia prevista) si reagisce eseguendo azioni. Le azioni possono essere sia di tipo infrastrutturale (scalabilità verticale) che di tipo applicativo o di piattaforma (scalabilità orizzontale).

MELA [18] è un sistema di monitoraggio ed analisi per applicazioni cloud. Prendendo le mosse da [12] il contributo più interessante del lavoro è l'intuizione della necessità di effettuare l'analisi ed il monitoraggio multi-livello concentrandosi non solo sulla parte infrastrutturale, che, essendo orientato al cloud, si compone di macchine virtuali, ma anche sulla parte di piattaforma o del servizio. L'amministratore di sistema deve fornire al framework due documenti XML: il primo (Figura 2.6a) per la descrizione della topologia della applicazione finalizzato ad evidenziare la struttura e le dipendenze dell'infrastruttura, il secondo (Figura 2.6b)) volto alla definizione delle metriche in cui bisogna esplicitare: a quale livello bisogna effettuare la misura, il tipo della metrica e le operazione da effettuare. È importante notare, al fine della comprensione dei capitoli successivi, come da questi lavori emerga

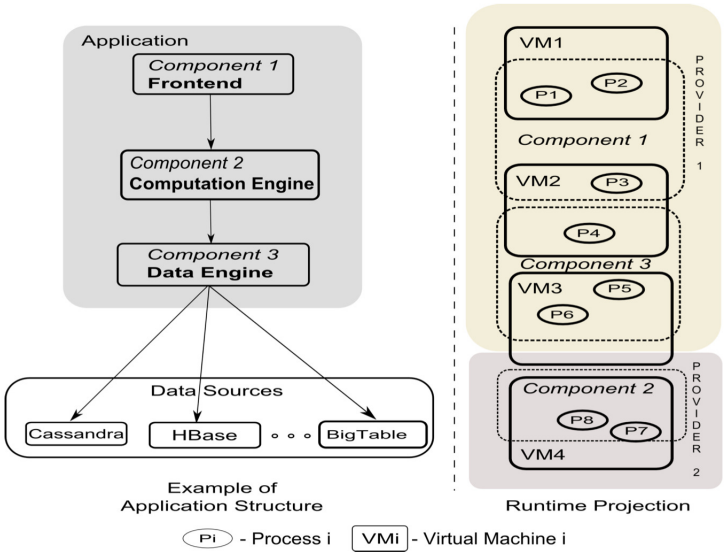


Figura 2.5: La visione dell'applicazione in SYBL

sempre di più la necessità di esplicitare la topologia dell'applicazione al fine di mettere in luce le dipendenze tra le varie componenti che la compongono. MELA è stato integrato in SYBL per la parte di monitoraggio ed analisi [11].
Sull'adattamento multi-livello all'interno dell'architettura MAPE si sono espressi favorevolmente anche [3] e [19] sottolineando come con un approccio legato a diversi aspetti dell'applicazione come la topologia, il livello infrastrutturale o il livello dei servizi sia possibile gestire al meglio, sia qualitativamente che quantitativamente, il sistema. Dagli studi menzionati emorgono alcuni principi fondativi del nostro lavoro:

- è sempre più forte l'esigenza di automatizzare il cloud e le applicazioni web;
- cloud non significa solo infrastruttura, il sistema autonomico deve essere multi-livello;
- l'architettura MAPE è uno standard de facto;
- la ricerca si concentra soprattutto sulle componenti di monitoraggio, analisi e pianificazione, ma l'esecutore è un problema non banale se si pensa alla eterogeneità del mondo cloud e alla sua natura distribuita;
- il sistema autonomico deve integrarsi nell'applicazione in maniera trasperente, senza modificarne la logica di business.

```
<CloudService id="S" name="M2MDaaS">
    <ServiceTopology id="ST1" name="EventProcessing">
        <ServiceUnit id="ST1_SU1" name="LoadBalancer"/>
        <ServiceUnit id="ST1_SU2" name="EventProcessing"/>
    </ServiceTopology>
    <ServiceTopology id="ST2" name="DataEnd">
        <ServiceUnit id="ST2_SU1" name="DataController"/>
        <ServiceUnit id="ST2_SU2" name="DataNode"/>
    </ServiceTopology>
</CloudService>
```

(a) La descrizione della topologia

```
<CompositionRules TargetLevel="ServiceTopology" >
   <CompositionRule TargetID="ST1">
       <SourceMetric name="dataOut" unit="GB/s"
              level="ServiceUnit"/>
       <Operations>
           <Operation name="SUM"/>
           <Operation name="DIV" unit="no/s"
                      sourceMetric="numberOfClients"
                      level="ServiceTopology"/>
       </Operations>
       <ResultingMetric name="dataCostPerClient" unit="$"
           type="COST"/>
   </CompositionRule>
</CompositionRules>
```

(b) La definizione delle metriche

Figura 2.6: Mela, un framework per il monitoraggio e l'analisi multi-livello

2.2 Lo stato dell'industria

Il cloud è certamente uno dei campi in cui si è più concetrata l'innovazione tecnologica: applicazioni e servizi web sempre più complessi e di grandi dimensioni richiedono una gestione che risulta poco sostenibile sia economicamente che quantitativamente se effettuata privatamente dalla singola azienda. Nascono quindi provider che mettono a disposizione servizi e risorse, a cui si usufruisce tramite web e su richiesta, che semplificano e automatizzano notevolmente alcuni dei problemi sopracitati. In particolare due categorie di prodotti si collocano al centro del mercato: le soluzioni IaaS (Infrastructure As A Service) e quelle denominate PaaS (Platform As A Service). Nelle prime il provider offre risorse infrastrutturali (virtual machine, storage, networking, etc.) on demand alle aziende che, in questo modo, non devono preoccuparsi di costruire un parco macchine privato e possono così evitare un forte investimento iniziale; nel secondo caso il provider offre una serie di servizi software che compongono un stack applicativo (un application server, un database, etc.) ed un ambiente di sviluppo in cui lo sviluppatore può scrivere la sua applicazione. Il grado di automazione di questi prodotti dipende da diversi fattori: le soluzioni PaaS tendono a trattare l'infrastruttura come una black-box, questo significa che, in un certo senso, l'automazione è totale ma la personalizzazione del servizio è praticamente nulla; le piattafor-

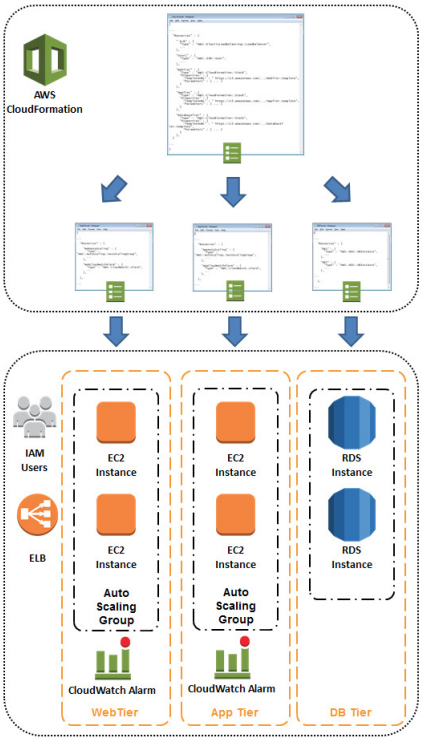


Figura 2.7: Cloud Formation

me IaaS, al contrario, offrono un automazione limitata, nella gran parte dei casi, alla scalabità e al monitoraggio automatico ma garantiscono un buon grado di personalizzazione. Di seguito verranno descritte le funzionalità di automazione dei player più importanti nel campo del cloud.

2.2.1 Amazon Web Service

La piattaforma IaaS più importante sul mercato è Amazon Elastic Compute Cloud (EC2) che si combina, per quanto riguarda lo storage, con il servizio Simple Storage Service (S3). Offre un ambiente integrato, accessibile attraverso un'interfaccia web o via codice, in cui lanciare virtual machine con diversi sistemi operativi, configuare le impostazioni di rete e gestire i dischi. L'automazione fornita consiste in prima analisi nella componente Auto Scaling che grazie all'integrazione con Amazon Cloudwatch, un sistema di monitoraggio, permette di schedulare l'allocazione di nuove risorse quando soglie predefinite vengono violate. Amazon offre inoltre tre prodotti volti soprattutto a semplificare il deploy di risorse AWS: Amazon Cloud Formation, Amazon OpsWork e Amazon Beanstalk.

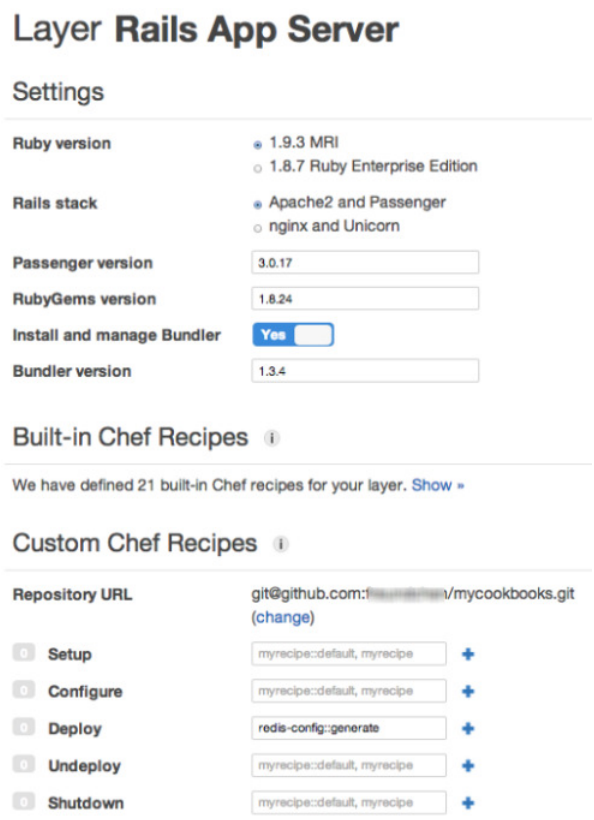


Figura 2.8: La creazione di un application server in OpsWork

Con Cloud Formation è possibile descrivere l'applicazione attraverso template scritti in JSON. In Figura 2.9 si mostra un esempio in cui si definisce una semplice architettura formata da un web server che si appoggia ad un database. Il template contiene parametri da ottenere durante il caricamento del file (vedi campo 'Parameters' nell'esempio) come elementi di login o di configurazione. Si dichiarano inoltre le risorse AWS da inizializzare. Non eslicitando i parametri e descrivendo la topologia dell'applicazione si garantisce la riutilizzabilità del modello non lengandolo ad una particolare caso d'uso. Le risorse possono dipendere da altre risorse utilizzando la clausola 'WaitCondition', viene così definito un ordine in cui eseguire le inizializzazioni. Cloud Formation si integra con CloudWatch e permette di definire regole per la scalabilità automatica. L'utente quindi attraverso una web console carica il template ed inserisce i parametri (se presenti); di conseguenza il sistema elabora il file, lo suddivide in parti (una per ciascuna risorsa) ed esegue, orchestrando il processo, il deploy dell'infrastruttura (Figura 2.7).

OpsWork è il tentativo di integrare nativamente il mondo DevOps (Sezione

Figura 2.9: Un estratto di un file Cloud Formation

2.2.3) in Amazon Web Service. In OpsWork l'applicazione è descritta da uno stack che si crea attraverso l'interfaccia web. Lo stack è un insieme di servizi software ciascuno rappresentante un tier[1]. Ciascun tier è estendibile con funzionalità dell'ecosistema Amazon: si può aggiungere un 'Elastic Loadbalancer', impostare i gruppi di sicurezza, la scalabilità automatica o la tipologia di storage. La configurazione di ciascuna instanza del tier avviene attraverso l'integrazione con Chef (Sezione 2.2.3). Amazon propone una scelta di tier predefiniti:

- loadbalancer tier
  - Elastic Load Balancing (AWS)
  - HAProxy
- application e web server tier
  - Java App Server (Tomcat)
  - Node.js App Server
  - PHP App Server
  - Rails App Server
  - Nginx Web Server
- MySQL tier
- Ganglia tier
- Memcached tier

[1]Nella nostra terminologia tier rappresenta un livello all'interno dello stack (application server tier, data tier, web tier) mentre layer rappresenta un livello all'interno della singola macchina (layer applicativo, layer dei servizi o middleware, layer infrastrutturale). Amazon per indicare i tier utilizza la parola layer.

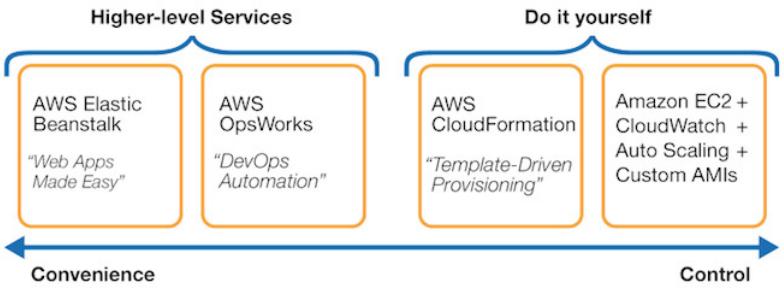


Figura 2.10: Amazon Web Service

oppure è possibile crearne uno personalizzato perdendo però parte delle funzionalità offerte.
Beanstalk è a tutti gli effetti la soluzione PaaS di Amazon. Offre uno stack standard a tre tier: HTTP Server (Apache), application server (Node.js, PHP, Python, Ruby, etc.) e il data tier (Amazon SimpleDB, Oracle, IBM DB2, etc.) su cui fare il deploy della propria applicazione. Si inserisce all'interno dell'ecosistema AWS appoggiandosi ad EC2 per la computazione, su S3 per lo storage e supporta Auto Scaling. Inoltre, al contrario della maggioranza di prodotti PaaS, è concesso l'accesso a queste risorse per eventuali configurazioni ma, tuttavia, non è possibile modificare lo stack applicativo. In conclusione, come si mostra in Figura 2.10, il tradeoff più significativo nell'offerta Amazon è quello tra facilità d'uso e comodità rispetto al controllo delle risorse; spostandosi verso il primo aspetto si tende ad un approccio PaaS in cui l'automazione è completa ma non controllabile dall'utente, al contrario un approccio più rivolto alla personalizzazione del servizio è più vicino alla filosofia IaaS a scapito però della portabilità, riusabilità e, in particolare, della gestione autonomica.

2.2.2 Google App Engine

Una dei principali prodotti PaaS è Google App Engine. GAE permette il deploy e la gestione di applicazioni web all'interno dell'ecosistema Google. È infatti un servizio completo che mette a disposizione una piattaforma di sviluppo in diversi linguaggi di programmazione (Java, PHP, Python e Go) supportata da servizi come storage, database, ricerca, log, mail, manipolazioni di immagini e scalabilità automatica. A differenza di AWS, Google App Engine possiede molte più funzionalità lato sviluppatori rendendo più semplice la scrittura e la gestione di nuove applicazioni. Nonostante supporti vari framework di terze parti come Spring, il rischio di creare applicazioni su misura e quindi di lock-in è elevato. Infatti alcune delle tecnologie e dei servizi disponibili come Datastore, un database che utilizza Google Query Language, sono proprietarie e rendono praticamente inusabile l'applicazione in altri ambienti. L'infrastruttura è trattata come una black-box, inaccessibile all'utente se non attraverso delle minori personalizzazioni. È il tradeoff che segna il mondo commerciale del cloud: una semplificazione ed un approccio autonomico alla piattaforma preclude l'accesso alla infrastruttura.

2.2.3 DevOps: Chef & Puppet

La filosofia DevOps nasce come risposta alla sempre più crescente consapevolezza del conflitto che esiste tra la fase di sviluppo e quella delle operazioni.

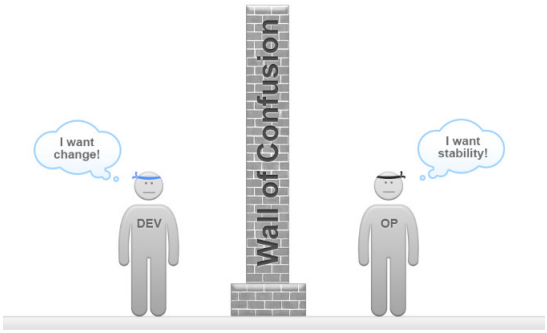


Figura 2.11: DevOps, il conflitto tra operazioni e sviluppo

Gli sviluppatori si occupano della scrittura dell'applicazione e dell'implementazione di nuove funzionalità mentre gli operatori di fare il deploy definitivo delle nuove versioni e della manutenzione. La mancanza di comunicazione nasce dalla differenza di motivazioni, processi e strumenti. Gli sviluppatori tendono ad avere una mentalità orientata al cambiamento sia per la natura del lavoro sia per gli obbiettivi imposti dal lato commerciale che cerca di dare all'azienda una voce originale ed innovativa in un mondo che cambia rapidamente ed è ricco di competizione. Dall'altra parte gli adetti alle operazioni hanno come primo obbiettivo la stabilità, l'affidabilità e la qualità del servizio cosa che è, almeno in parte, in contraddizione con il cambiamento. Per questo motivo spesso il reparto delle operazioni e quello dello sviluppo lavorano con manager diversi, in luoghi diversi e con strumenti diversi creando una barriera alla comunicazione che rallenta i processi aziendali e vanifica in parte l'approccio agile allo sviluppo. Nasce quindi l'esigenza, tra le altre cose, di creare strumenti software per l'automazione ed unificazione di questi processi. Puppet e Chef sono dei *configuration tool* che automatizzano la configurazione ed il deploy dell'infrastruttura. Si appoggiano sulla definizioni di 'ricette' riusabili che contengono un insieme di azioni da compiere come pacchetti da installare, modifiche di parametri in un file di configurazione o lo scaricamento di codice da una repository. Le ricette contengono anche le relazioni e dipendenze che occorrono tra le varie azioni. L'orchestrazione avviene quindi sulla singola macchina e riguarda soprattutto il livello applicativo e dei servizi. La parte infrastrutturale è una premessa al loro funzionamento, non si occupano quindi di lanciare nuove instanze o configurare la rete. Possono esere eseguiti in due modalità: parzialmente centralizzata (client-server) o completamente distribuita. Nella versione centralizzata c'è un server centrale che contiene metadata sui nodi a cui client possono attingere. Ciò nonostante non è possibile definire di-

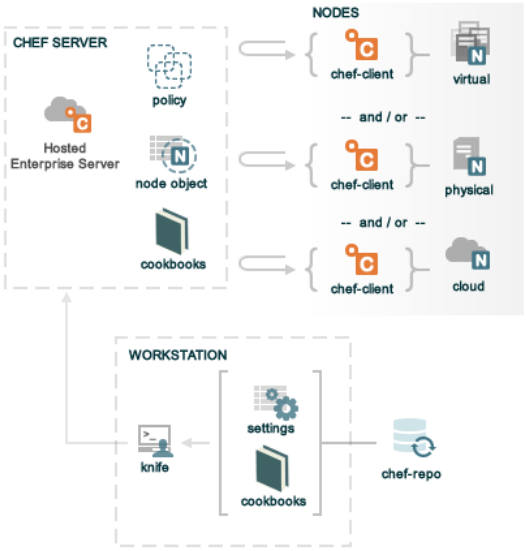


Figura 2.12: L'architettura parzialmente centralizzata di Chef

pendenze tra macchine, in altre parole l'orchestrazione non tiene conto della struttura a tier dell'applicazione con dipendenze di alto livello: non si può impostare un ordine con il quale eseguire la configurazione delle macchine facendo precedere, ad esempio, le azioni da compiere sugli application server a quelle del loadbalancer e comunicare dei dati quando necessario.

2.2.4 Cloudify

Cloudify è il prodotto che più raccoglie l'essenza di questo lavoro. Cloudify ha un approccio olistico al cloud: non si propone come un provider IaaS o PaaS bensì affronta il problema del deploy e la gestione di un'applicazione in tutte le sue parti. Cloudify si occupa:

- dell'approvigionamento delle risorse infrastrutturali in maniera agnostica, è quindi possibile scegliere il provider preferito (EC2, Azure, etc.);
- monitoraggio e configurazione di soglie (SLA);
- scalabilità automatica;
- l'installazione di servizi e la loro configurazione: integrazione con Chef;
- orchestrazione a livello di applicazione: è possibile definire dipendenze tra servizi che comporta il rispetto di un ordine per l'installazione e l'attivazione di questi.

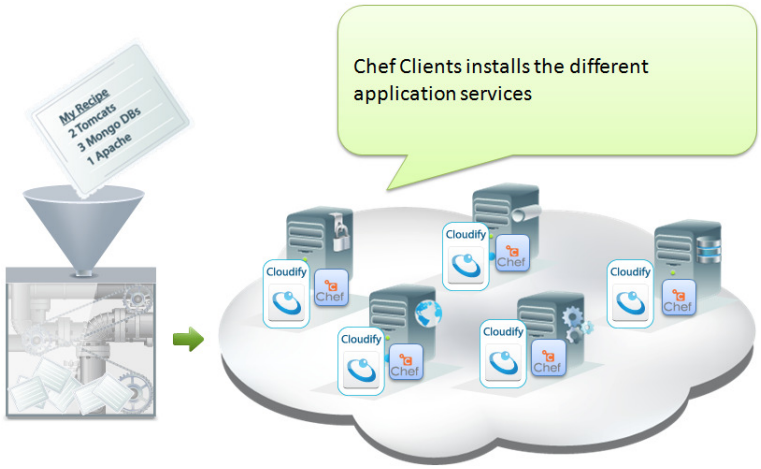


Figura 2.13: Cloudify e Chef

Cloudify quindi, attraverso delle ricette, permette la definizione di quale provider utilizzare a livello infrastrutturale, esplicitare qual'è il servizio da utilizzare su ciascuna macchina, le dipendenze tra diversi servizi (attraverso l'attributo `dependsOn`) e come configurare la macchina ovvero quali ricette Chef vanno eseguite su questa. Un orchestratore esamina la ricetta (Figura 2.13) e comunica con il provider IaaS scelto allocando le macchine necessarie su cui viene installato un agente proprietario e Chef, successivamente viene eseguita la configurazioni delle macchine e l'avvio dei servizi corrispondenti rispettando l'ordine esplicitato. Il limite di questo approccio è che si esaurisce al deploy dell'applicazione, non è possibile per esempio modificare parametri dinamici o statici di servizi e comunicare le modifiche alle altre macchine (se interessate). In generale manca il concetto di control loop e di pianificazione se non per quanto riguarda il monitoraggio e la scalabilità automatica.

2.2.5 Altri provider IaaS e PaaS

Esistono molti altri provider IaaS e PaaS sul mercato. In Figura 2.14 viene proposta una tabella riassuntiva che li confronta negli aspetti di interesse per questo lavoro.

	Monitoraggio ed Analisi	Pianificazione	Automazione ed orchestrazione
AWS	*CloudWatch*: monitoraggio di alcune metriche standard e individuazione di violazioni e picchi. Personalizzabile	Reagisce alle violazioni	Auto Scaling CloudFormation OpsWork Beanstalk
Azure	Monitoraggio di alcune metriche standard e individuazione di violazioni e picchi. Personalizzabile	Reagisce alle violazioni	Scalabilità pianificabile e altamente configurabile
OpenStack	Celiometer: monitoraggio multi livello dall'infrastruttura all'energia	Reagisce alle violazioni	*Heat*: deploy automatico a partire da un template (supporta CloudFormation). Scalabilità automatica ed integrata con Celiometer
Google Compute Engine	Informazioni generiche sulle VM	Nessuna	Come tutte le soluzioni PAAS offre un alto grado di automazione nascondendo la configurazione all'utente
System Center	*Operation Manager*: monitoraggio multi-livello	Reagisce alle violazioni	*Orchestrator Runbooks:* creazione di workflow, attraverso una interfaccia grafica, per l'automatizzazione di azioni. Integrato con Operation Manager
Rightscale	Monitoraggio via *collectd* e *RDDtool*	Basato su script	Automatizzazione del deploy dell'applicazione attraverso script
vCloud	*vCenter Operation*: monitoraggio e analisi dinamici. Impostazioni di soglie che considerano la storia del sistema e ne prevedono l'evoluzione.	Reattivo e proattivo (considera l'evoluzione del sistema)	*vCenter Orchestrator*: scalabilità automatica e cloud bursting (hybrid cloud).

Figura 2.14: Panoramica dei maggiori provider IaaS o PaaS

Capitolo 3

Definizione del problema di ricerca

In questo capitolo si esporrà il problema di ricerca mettendo in evidenza il contesto in cui si inserisce e gli obbiettivi che si vogliono raggiungere; si introdurrà anche la soluzione proposta, descritta nel dettaglio nei capitoli successivi. Infine si fornirà un esempio che metterà in luce alcune difficoltà e concetti essenziali per la compresione del lavoro.

3.1 Runtime Management

I sistemi moderni sono soggetti a continui cambiamenti nei requisiti e nel software, e anche l'ambiente in cui vengono eseguiti è protagonista di frequenti variazioni che influiscono sulla qualità del servizio. Affinché il sistema rispetti le funzionalità e gli standard di qualità definiti e quindi riesca ad evolvere di conseguenza alle variabili esterne, è necessario ridefinire le tecniche e gli strumenti di gestione dinamica (o runtime management) dell'applicazione. Nel campo delle applicazioni e servizi web e cloud, su cui questo lavoro si focalizza, uno dei parametri che più determina incertezza e cambiamento è la quantità di carico (arrival rate) che riceve il sistema; in piccola parte aumenti e picchi improvvisi possono essere previsti e ripetersi ciclicamente (un servizio può essere utilizzato più in un certo periodo della giornata o dell'anno) ma in larga misura sono causati dalla complessità dell'ambiente e quindi risultano pressoché imprevedibili e, spesso, inaspettati. A seguito di queste variazioni nella quantità di carico ci si può trovare in due situazioni: se la quantità è superiore a quella prevista si verificherrano violazioni del SLA e, più in generale, riscontreremo un discostamento dalla

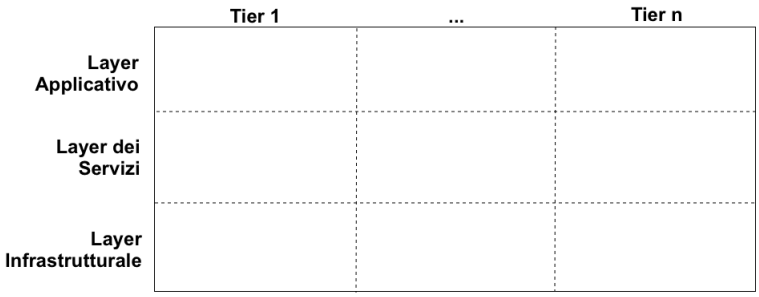


Figura 3.1: La suddivisione in layer e tier delle applicazioni

qualità del servizio desiderata (tempi di risposta lunghi, crash del sistema, etc.); se invece è inferiore alle aspettative si vanifica l'efficienza nella gestione delle risorse utilizzandone più del necessario. Essendo l'SLA spesso molto stringente, i provider di servizi preferiscono trovarsi nel secondo scenario ovvero sovra-allocare il sistema affinché anche in casi di alto traffico sia possibile rispettare standard di qualità. La conseguenza di questo fenomeno è che l'utilizzazione media dei server in un tipico data center sia circa del 30-40% [5] (altri studi [23] dicono addirittura 18%).
La diffusione di tecniche di controllo dinamico delle risorse ha permesso la realizzazione di sistemi capaci di adattare la propria configurazione a fronte dei cambiamenti e dell'evoluzione dell'ambiente d'esecuzione. Come analizzato nel Capitolo 2, l'architettetura MAPE costituisce il meta-modello di gran parte dei sistemi autonomici esistenti: attraverso un ciclo di controllo costituito dal monitoraggio delle risorse, l'analisi dei dati monitorati, la pianificazione di strategie e l'esecuzioni di azioni è possibile anticipare o reagire dinamicamente ai cambiamenti.

3.2 ECoWare

Molti dei modelli presentati [25, 26] tengono conto di un solo aspetto dell'applicazione: il volume delle richieste; è stato però dimostrato che non è sufficiente per la gestione efficiente delle risorse [22, 24]. Altri lavori considerano anche la tipologia di richieste ricevute (workload-mix aware) in quanto a tipi di richiesta differenti possono corrispondere strategia di adattamento differenti. È stato mostrato inoltre nel Capitolo 2 che emerge l'esigenza di considare l'applicazioni in tutti i suoi aspetti attraverso sistemi multi-livello. Infatti discostamenti dalla qualità attesa del servizio si possono presentare, come sintomi, ad un livello ma in realtà la vera e propria causa del problema potrebbere su un altro; inoltre potrebbe essere più efficiente ed efficace

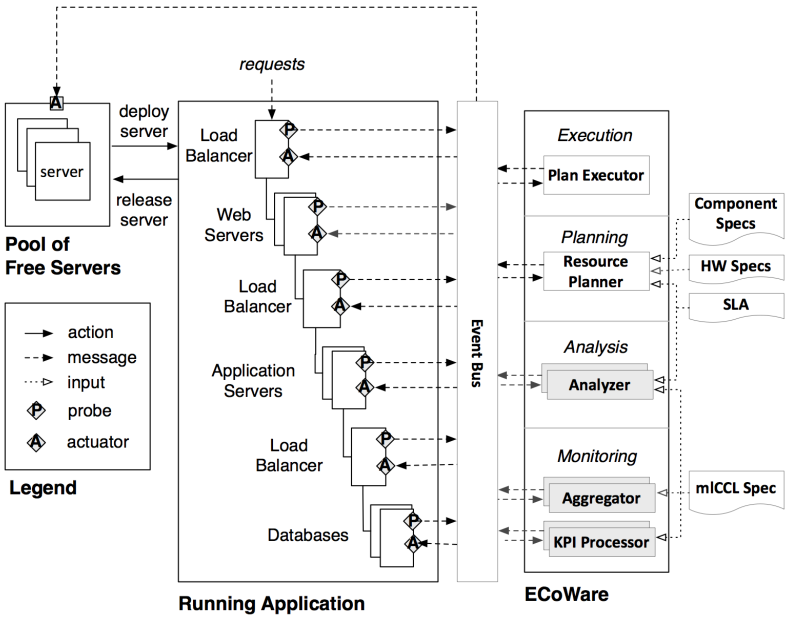


Figura 3.2: ECoWare all'interno di un'applicazione web a sei tier

intervenire, in maniera sincronizzata, su più livelli [4] in modo da creare una strategia complessiva che armonizzi l'adattamento in tutte le sue componenti.

Il progetto ECoWare, sviluppato dal Politecnico di Milano, è un framework multi-tier e multi-layer per la gestione autonomica di applicazioni web e cloud. Nell'ingegneria del software un'applicazione è suddivisa in più tier, ciascuno avente una funzionalità diversa e composto da una o più macchine. Esempi di tier possono essere quello per la gestione dei dati, della business logic o dell'interfaccia web. Inoltre in ciascuna macchina di ciascun tier si possono riconoscere tre layer: il layer applicativo (o application layer) che fa riferimento al codice o ai dati propri della particolare applicazione, il layer dei servizi (o service layer) che riguarda i middleware software a supporto della stessa applicazione e quello infrastrutturale (o infrastructure layer) che corrisponde alle risorse hardware (CPU, memoria, network, etc.) su cui vengono eseguiti i software dei layer superiori. Come si mostra in Figura 3.1 si propone quindi una suddivisione ortogonale dell'applicazione che tende a coinvolgerne tutti gli aspetti: orizzontalmente con i layer e verticalmente con i tier. Questo approccio deve essere valido in ciascuna delle componenti dell'architettura MAPE come sarà esposto di seguito. ECoWare è stato realizzato in collaborazione con la *Univesity of Calfifornia San Diego*, il cui contributo più importante si concetra sul pianificatore. In Figura 3.2

si mostra l'architettura di ECoWare a supporto di un'applicazione web composta da sei tier:

1. loadbalancer tier, per la gestione delle richieste provenienti dai client;
2. web server tier;
3. loadbalancer tier, per la gestione delle richieste verso gli application server;
4. application server tier;
5. loadbalancer tier, per la gestione delle richieste verso i database;
6. database tier.

Il framework utilizza un bus ad eventi per la comunicazione tra i componenti.

3.2.1 Monitorragio

La componente di monitoraggio permette la correlazione e l'aggregazione multi-livello di dati. Il sistema è composto da tre parti:

- sonde e sensori distribuiti sulle macchine costituenti l'applicazione;
- un nodo centrale che raccoglie i dati, li manipola e ne permette la visualizzazione;
- un linguaggio, mlCCL (Multi-Level Collection and Constraint Language[1]), che permette di dichiarare i parametri da monitorare e le soglie di violazione; è la componente attraverso cui l'utente si interfaccia al sistema.

Le sonde ed i sensori che misurano i dati sono dei piccoli applicativi che in maniera trasparente si innestano nel sistema; all'interno di una singola macchina possono essere molteplici, uno o più per layer. Sono presenti sonde che si interfacciano con strumenti già esistenti, come *Ganglia* o *Collectd*, che misurano alcuni parametri standard o con strumenti creati ad hoc utilizzando tecniche come AOP, filtri o proxy che permettono di acquisire i dati senza intaccare la logica dell'applicazione. Le misure raccolte vengono pubblicate sul bus di sistema come eventi e raccolte dal nodo centrale dove sono attivi i *KPI Proccesor* e gli *Aggregator*.
I KPI Processor sono implementati utilizzando *Esper*, uno strumento per processare flussi di eventi complessi. Con Esper è possibile eseguire query

[1] Originariamente Multi-Layer Collection and Constraint Language

```
O = (getUserWeather, TwitterWeather, Orchestrator);
DCA = (convert, Converter, DCA);
T = (getUserLocation, Twitter, Orchestrator);
W = (getWeatherByLocation, Weather, Orchestrator);

avgrtO = collect(avgRT, O, 20 seconds,
   2 minutes, null);
rateO = collect(rate, O, 20 seconds,
   1 hour, null);
avgrtDCA = collect(avgRT, DCA, 20 seconds,
   2 minutes, null);
avgrtT = collect(avgRT, T, 20 seconds,
   2 minutes, null);
avgrtW = collect(avgRT, W, 20 seconds,
   2 minutes, null);

violation = evaluate(avgrtO, avgrtO.get(value) > 5500
            || avgrtO.get(value) < 3000);

agg = aggregate(violation, rateO.window(2 minutes),
         avgrtDCA.window(2 minutes),
         avgrtT.window(2 minutes),
         avgrtW.window(2 minutes));
```

Figura 3.3: Un blocco di codice mlCCL

complesse su uno stream di eventi utilizzando un linguaggio chiamato EPL, dalla sintassi simile a SQL. La differenza principale tra SQL ed EPL è che al posto di eseguire query su dati salvati in un database, EPL salva le query e le esegue continuamente sui dati. Nelle query è possibile utilizzare funzioni matematiche e statistiche come media e percentile, e si definisce una finestra temporale, una sorta di periodo, sulla quale eseguire il calcolo che viene poi pubblicato anch'esso sul bus. Attraverso le query è anche possibile raggruppare i dati, in questo modo è possibile impostare a quale granularità pubblicare l'output: è possibile, per esempio, ottenere dati per macchina, cluster, tier o sistema oppure dati riguardanti il singolo thread, un insieme di thread o un processo.

Gli Aggregator aggregano eventi generati dai KPI Processor al fine di creare uno snapshot comprensivo di più metriche. Gli Aggregator possiedono un evento principale, almeno un evento secondario ed una finestra temporale. Non appena viene ricevuto un dato appartente all'evento principale si raccolgono tutti gli eventi secondari che si verificano nella finestra indicata; in questo modo si genera un evento aggregato che viene anch'esso pubblicato sul bus.

Il linguaggio mlCCL (Figura 3.3) è l'interfaccia attraverso la quale l'utente può determinare le metriche e le componenti da monitorare. È possibile definire delle *location* che descrivono un punto del sistema che si vuole montiorare. Il monitoraggio e la computazione dei dati è affidata alla funzione `collect` a cui vengono passati una location, un KPI e due finestre tempo-

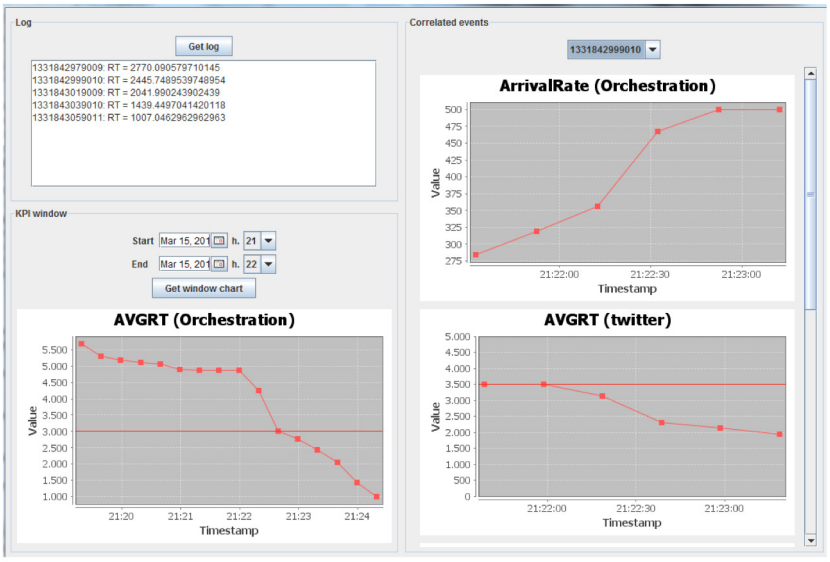


Figura 3.4: La dashboard della componente di monitoragigo

rali: una rappresenta il tempo su cui raccogliere dati, l'altra ogni quanto calcolare e pubblicare il risultato. Per l'aggregazione di dati si utilizza la funzione `aggregate` a cui si passa l'evento primario ed una lista di eventi addizionali. La funzione `evaluate` riguarda l'analisi dei dati che si tratterà di seguito.
La visualizzazione dei dati avviene attraverso una dashboard mostrata in Figura 3.4. Essendo nato come progetto indipendente la componente di monitoraggio offre anche questa funzionalità che però non rientra nella gestione autonomica dei dati, bensì è dedicata agli amministratori di sistema per controllare lo stato dell'infrastruttura, soprattutto in caso di violazioni.

3.2.2 Analisi

Il modulo dedicato all'analisi, l'Analyzer, ha il compito di raccogliere i dati aggregati e processati forniti dalla componente di monitoraggio ed estrarne conoscenza. L'analisi riguarda soprattutto la scoperta di anomalie o il superamento di soglie critiche definite dall'utente attraverso il linguaggio mlCCL o la violazione del SLA. L'utente può definire l'analisi con mlCCL grazie alla funzione `evaluate` che possiede come parametri un evento precedentemente definito attraverso la funzione `collect` ed una espressione. L'espressione può contenere i classici operatori logici e di comparazione e i quantificatori *exists* e *forall*. L'analisi può, così come per la componente di monitoraggio,

possedere granularità diversa ed essere valuata per tier. L'Analyzer pubblica sul bus un evento contente lo stato del sistema, le soglie superate e le violazioni avvenute.

3.2.3 Pianificatore

Il pianificatore o Planner, curato in collaborazione con UCSD, si occupa della stesura di una strategia che prende le mosse dagli eventi generati dall'Analyzer. Anche questa componente è multi-tier e multi-layer infatti non solo può decidere di allocare o deallocare macchine su un particolare tier ma anche di riconfigurare un servizio come, ad esempio, aumentare il numero di thread ad un application server. Le decisioni sono prese, puntando alla massima efficienza nel rispetto dei vincoli posti dall'utente, attraverso un modello basato sulla teoria delle code. Il contributo innovativo del Planner è racchiuso in tre funzionalità che combinate fra loro costituiscono un ulteriore valore aggiunto:

- mix-aware: attraverso tecniche di clusterizzazione le richieste vengono raggruppate per tipologia (ad esempio CPU-intensive o memory-intensive) per essere gestite più accuratamente.

- hardware-aware: nel modello si considera anche la tipologia di macchine disponibili e costituenti l'infrastruttura, in questo modo se si presenta un picco di richieste che stressano particolarmente la CPU il Planner può creare una strategia in cui allocare, se disponibile, una macchina con spiccate capacità computazionali.

- tier-aware: il Planner possiede un modello basato sulla teoria delle code per ciascun tier presente nell'applicazione.

L'output del pianificatore è una lista di azioni da eseguire, il piano, pubblicata anch'essa sul bus.

3.3 Esecutore

Se il lavoro svolto è stato diffuso in tutte le componenti dell'architettura MAPE, il principale contributo innovativo di questa tesi è rappresentato dalla costruzione dell'esecutore o Executor; infatti ECoWare inizialmente possedeva un esecutore creato ad hoc, molto semplice e senza un'interfaccia né un modello a supporto. Di seguito verranno analizzate le motivazioni e gli obbiettivi del lavoro e verrà presentato un esempio introduttivo al problema.

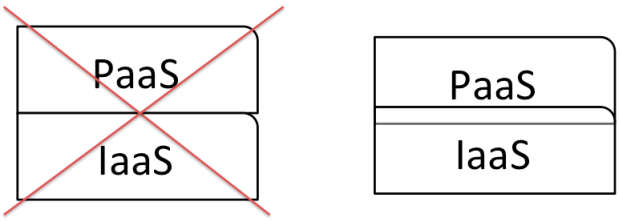


Figura 3.5: PaaS vs IaaS

3.3.1 Oltre la Dicotomia Infrastruttura - Piattaforma

Con l'avvento del cloud sono state affrontate e risolte diverse problematiche: da una parte quelle riguardanti l'infrastruttura (IaaS) volte all'approvigionamento di risorse con un servizio fruibile on-demand dalle aziende, dall'altra quelle relative alla piattaforma (PaaS) che permette la scrittura di applicazioni attraverso un ambiente di svlippo integrato ed una serie di middleware a supporto che nell'insieme astraggono l'infrastruttura. Come analizzato in Sezione 2.2 queste due realtà sono rimaste completamente o parzialmente inconciliabili: i prodotti industriali IaaS offrono una completa personalizzazione del servizio mentre quelli PaaS si concentrano sulla facilità d'uso e nel creare un ecosistema chiuso e completo per lo sviluppo di applicazioni nascondendo all'utente la parte infrastrutturale.

In realtà il problema da risolvere è uno ed uno solo: la gestione autonomica delle applicazioni web. Non esiste piattaforma senza infrastruttura e viceversa. Si pensi solo al deploy iniziale: è necessario allocare delle macchine, configurare la rete, installare e configuare i middleware a seconda della funzionalità richiesta e caricare su ciascuna il codice applicativo; gli aspetti toccati dalle azioni da compiere sono molteplici e non si riducono solamente all'infrastruttura o alla piattaforma. La convinzione di fondo di questo lavoro è che nel mondo del cloud e delle applicazioni web è sbagliato considerare la parte infrastrutturale e la piattaforma come compartimenti stagni; è necessario creare uno strumento che coniughi i due aspetti senza rinunciare alla personalizzazione su entrambi.

I lavori accademici presentati in Sezione 2.1, architetturalmente basati su il control-loop MAPE, concentrano gli sforzi sulle altre componenti: monitoraggio, analisi e pianificazione; il modello o l'implementazione dell'esecutore è assente o relegata a poche righe. In realtà nell'esecutore di sistemi autonomici multi-livello risiedono diverse problematiche non trascurabili che diventano sostanziali quando si passa alla fase implementiva e di valuta-

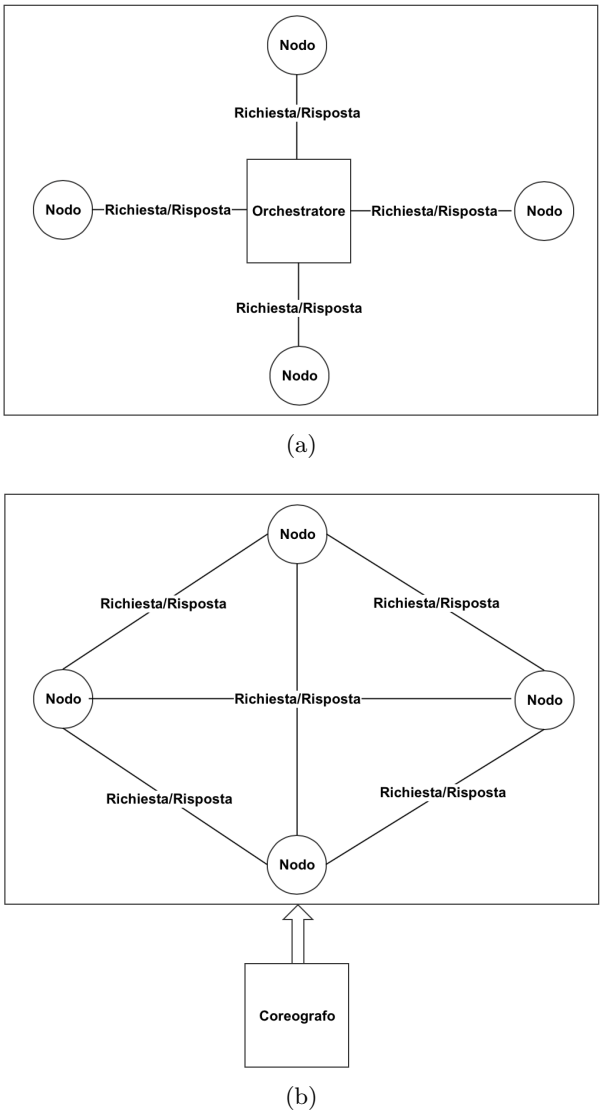


Figura 3.6: Orchestrazione vs Coreografia

zione. Da queste motivazioni emerge la necessità di creare un modello e successivamente l'implementazione di un esecutore che risolva i problemi esposti.

3.3.2 Obbiettivi

L'esecutore che si vuole costruire dovrà basarsi su un modello ben definito a cui qualunque pianificatore potrà agganciarsi. Dovrà gestire l'intero ciclo di vita di un'applicazione: il deploy, la configurazione e la manutenzione a

runtime. Attraverso la compilazione di semplici piani, il pianificatore potrà eseguire azioni sia infrastrutturali come l'allocamento di nuove macchine sia di piattaforma come la configurazione del numero massimo di connessioni ad un database; in altre parole dovrà essere multi-layer. L'esecutore dovrà conoscere la suddivisione in tier dell'applicazione e tenere conto delle dipendenze tra questi. L'esecuzione delle azioni dovrà essere deterministica e robusta ad eventuali guasti o malfunzionamenti.
Essendo la scalabilità un aspetto di fondamentale importanza a fronte di sistemi composti potenzialmente da migliaia di server è cruciale la scelta della topologia dell'architettura. L'esecutore infatti dovrà dirigere le comunicazioni tra i nodi del sistema, rispettando dipendenze e sincronizzando le operazioni. Di fronte a questi problemi l'approccio autonomico può essere di due tipi mostrati in Figura 3.6: orchestrazione o coreografia. Nell'orchestrazione un nodo centrale, l'orchestratore, gestisce le comunicazioni e le sincronizzazioni tra nodi che non possono comunicare direttamente tra loro. Nella coreografia un nodo centrale distribuisce ad ogni attore (nodo) la sua parte e successivamente ciascuno di questi si autogestisce coordinandosi direttamente con gli altri nodi. In questo lavoro si è scelto un approccio orientato alla coreografia poiché il numero indefinito e potenzialmente elevato di nodi renderebbe il nodo centrale il collo di bottiglia del sistema.

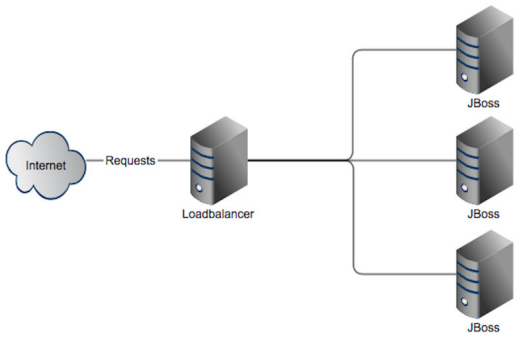
3.4 Un esempio

Per comprendere meglio il problema e per sottolineare l'importanza di un approccio multi-livello in un sistema autonomico si propone un esempio. Ipotizziamo per semplicità un'applicazione, installata su virtual machine nel cloud, composta da un loadbalancer che riceve le richieste dai client e alcuni application server che utilizzano JBoss: vedi Figura 3.7a. Quando si alloca una nuova macchina (per esempio a causa di un picco di traffico) è necessario compiere i seguenti passaggi:

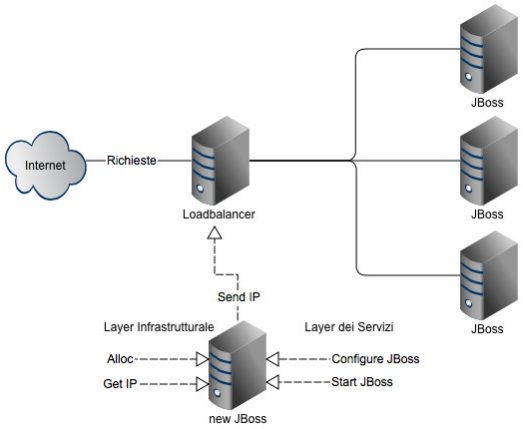
- richiedere al provider IaaS una nuova VM contente JBoss;
- ottenere l'indirizzo IP della VM;
- configurare alcuni parametri di JBoss come il massimo numero di thread o la porta su cui ricevere richieste;
- attivare JBoss;

- comunicare al load balancer l'indirizzo IP del JBoss appena attivato affinché incominci ad indirizzare, secondo un certo algoritmo, anche alla nuova macchina una parte delle richieste.

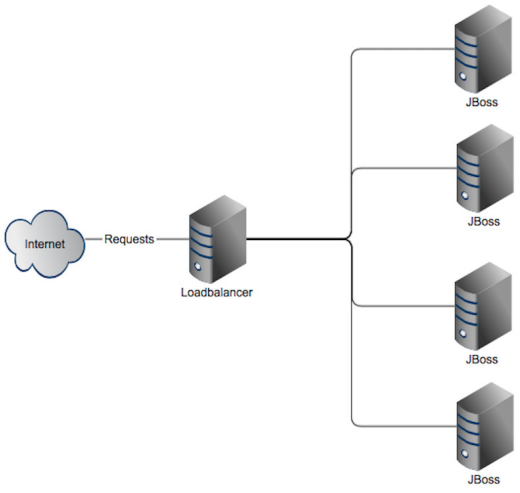
Come si può notare in Figura 3.7b, per ottenere la nuova topologia le azioni di allocamento della VM e di networking sono solo parte problema: è necessario infatti gestire simultaneamente la parte infrastrutturale e di piattaforma rispettando un ordine da determinare e coinvolgendo le altre macchine interessate. Questi ed altri problemi stanno alla base della costruzione di un esecutore distribuito e verranno risolti nei capitoli successivi.



(a)



(b)



(c)

Figura 3.7: L'inserimento di un nuovo application server in una semplice web application

Capitolo 4

Architettura della Soluzione

I discovered that if one looks a little closer at this beautiful world, there are always red ants underneath.

David Lynch

In questo capitolo verrà descritto il modello della soluzione al problema posto in Capitolo 3, in particolare verrà introdotto il concetto di application stack, di tier e di action che insieme compongono il meta-modello dell'esecutore. Si mostrerà un'instanza del meta-modello attraverso la definizione di due linguaggi: uno per la descrizione dell'applicazione e l'altro per la scrittura del piano. Infine si descriverà l'architettura e il processing model.

4.1 Anatomia di un'applicazione

Come si è esposto in Sezione 3.3, affinché l'esecutore possa coordinare correttamente l'esecuzione di un piano è necessario che sia a conoscenza della struttura dell'applicazione. Un'applicazione web è normalmente suddivisa in più tier ognuno con una funzionalità logica e software differente. Chiamiamo *application stack* o stack applicativo l'insieme dei tier che compongono l'applicazione. Ad esempio, la più classica delle web application è quella a tre tier:

- loadbalancer tier, a fronte dell'applicazione;
- application server tier;
- database tier;

Lo stack applicativo determina una gerarchia tra tier in quanto il flusso di informazioni è monodirezionale e fluisce dal basso verso l'alto; in altre

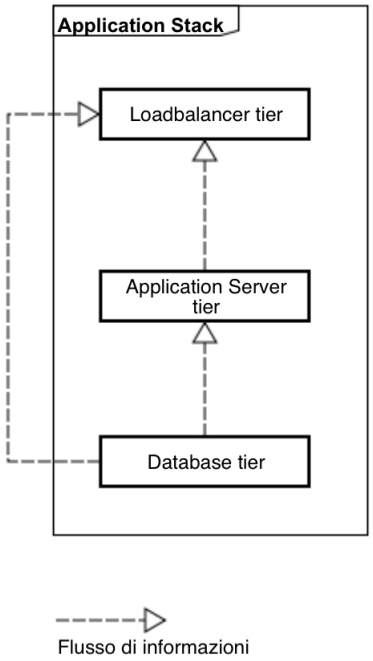


Figura 4.1: Un'applicazione a tre tier. Il flusso di informazioni si sposta dal basso verso l'alto.

parole un tier può dipendere da un altro solo se risulta più in alto nello stack. Di conseguenza nell'esempio precedente il loadbalancer può richiedere informazioni e quindi avere una dipendenza con gli application server ma non viceversa (Figura 4.1)[1]. Definiamo due predicati: $upper(tier_i, tier_j, stack)$ che determina se il tier $tier_i$ è più in alto nello stack del tier $tier_j$ ed il predicato infisso $tier_j \rightarrow tier_i$ che determina se $tier_i$ dipende da $tier_j$.

$$\forall tier_1, tier_2 \in stack, tier_1 \neq tier_2 \ \wedge \ tier_2 \rightarrow tier_1 \\ \Leftrightarrow upper(tier_1, tier_2, stack)$$

Nell'applicazione ogni tier (Figura 4.2) è composto da uno o più nodi (o macchine). All'interno di un nodo si indentifica la seguente suddivisione in tre layer:

- application layer: riguarda il codice e i dati dell'utente;

[1]Questa scelta permette di ottenere una soluzione elegante ed un processing model chiaro e deterministico. È però vero che si pone una limitazione che potrebbe precludere la gestione di alcune applicazioni strutturate in maniera non convenzionale. Per questo motivo negli sviluppi futuri questa scelta potrebbe essere rivista.

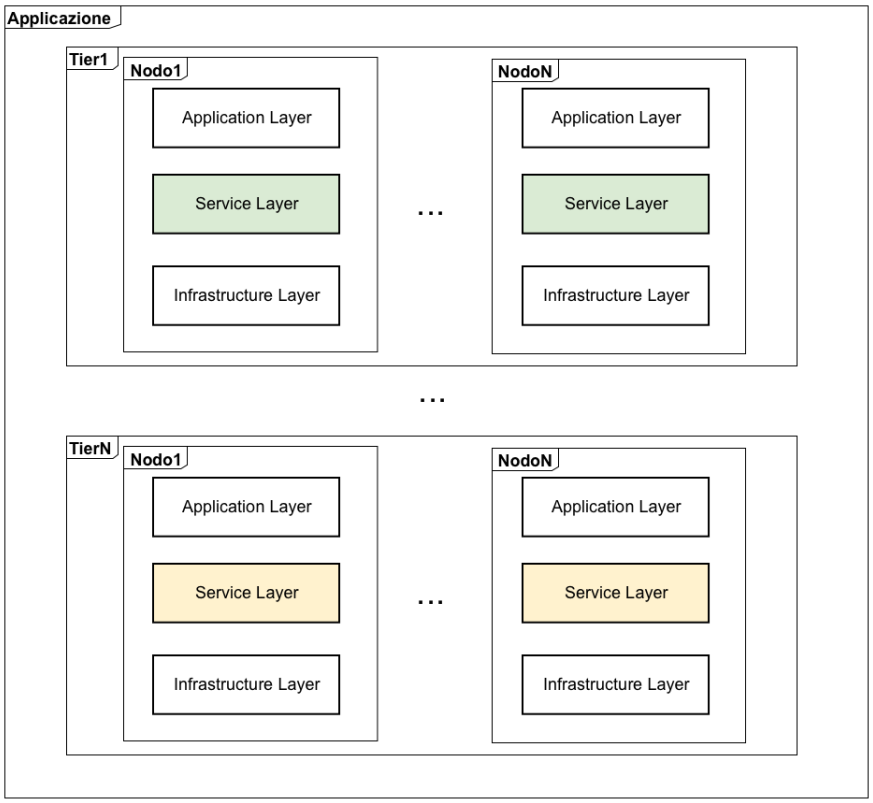


Figura 4.2: Anatomia di un'applicazione

- service layer: il livello dei middleware a supporto dell'applicazione; ogni tier è associato ad un particolare servizio (JBoss, MySQL, etc.);
- infrastrucure layer: tutto ciò che concerne l'infrastruttura come CPU, memoria e network.

Infine si definisce *topologia* l'insieme dei nodi attivi nell'applicazione. Si dice che un nodo o un tier è topologicamente superiore ad un altro se è posto più in alto nello stack.

4.2 Tier

L'applicazione è ingegneristicamente suddivisa in tier, ognuno avente una funzionalità diversa ovvero un diverso servizio associato. L'esecutore deve conoscere diverse informazioni di ciascun tier:

- la quantità minima di macchine accettata;
- la quantità massima di macchine accettata;

- le *capability*;
- le *action*.

Si noti come il modello di tier che verrà illustrato di seguito vale per tutti i nodi che lo compongono, questo non significa che ogni nodo di un tier sia uguale agli altri: il modello è in comune mentre la configurazione e i valori assegnati alle capability (spiegate successivamente) e ai vari parametri possono cambiare.

4.2.1 Capability

Le *capability* rappresentano tutti quei dati pubblici che ciascun nodo di un tier rende disponibili agli altri, in particolare a quelli topologicamente superiori nello stack (vedi Sezione 4.1). Ogni nodo contiene quattro capability predefinite:

- `id`: un identificativo univoco all'interno della topologia.
- `ip`: l'indirizzo IP della macchina;
- `started`: un booleano che descrive lo stato del servizio installato;
- `configuration`: la descrizione della configurazione hardware della macchina.

Le capability di un nodo possono essere lette dal nodo stesso e dai nodi topologicamente superiori (attraverso azioni parametri) mentre le modifiche possono avvenire solamente all'interno del nodo. Inoltre, come verrà descritto in seguito, possono essere utilizzate all'interno di espressioni per formulare le precondizioni delle azioni.

4.2.2 Action

Le *action* sono le azioni pubbliche dichiarate nel tier ed invocabili ed eseguibili su un nodo se inserite in un piano. Possono modificare le capability del nodo stesso ma non degli altri, nemmeno di quelli appartenenti allo stesso tier. Le azioni possono riguardare l'infrastruttura, il servizio installato o il layer applicativo. Le action possono utilizzare le capability di altri nodi attraverso argomenti espliciti ed impliciti che verrano descritti successivamente. Ogni tier possiede cinque azioni predefinite:

- `alloc`: comporta la richiesta di un nuovo nodo al provider IaaS, l'impostazione delle capability di default, in particolare viene assegnato un indirizzo IP e imposta la capability `started` a `false`;

- `start`: accensione del servizio, imposta `started` a `vero`;
- `new`: esecuzione di `alloc` seguita da `start`;
- `stop`: spegnimento del servizio, imposta `started` a `falso`;
- `remove`: esecuzione di `stop` (se il servizio è acceso) seguita dalla distruzione del nodo via provider IaaS.

È possibile inoltre possibile definire azioni personalizzate.

4.2.3 Parametri e precondition

Le azioni, ad eccezione di `start` e `stop` (scelta giustificata in Sezione 4.2.4), possono essere parametriche. I parametri possono essere di due tipi:

- primitivi: stringhe, numeri e booleani;
- nodo: è possibile indicare come argomento di una funzione un particolare nodo del piano (solo lettura).

Questi parametri sono espliciti ovvero se si vuole invocare una determinata action bisogna inserirli come argomenti nel piano.
È possibile prevedere un input implicito che non compare nell'invocazione dell'action. Chiamiamo questo genere di input *precondition*. Le precondition sono parametri di tipo nodo che non si riferiscono ad una macchina in particolare ma a quelli appartenenti ad un tier con eventualmente una espressione sulle capability da valutare. Le precondizioni possono essere di due tipi:

- *All*: sono precondizioni del tipo *tutti i nodi del tier* $tier_i$ che soddisfano una eventuale espressione. Possono riguardare zero o più nodi.
- *Any*: sono precondizioni del tipo *uno ed un solo nodo del tier* $tier_i$ che soddisfa una eventuale espressione. Notare che se il nodo non esiste l'azione non può essere eseguita.

Ogni azione può contenere sia parametri esplitici che precondizioni.

4.2.4 Azioni Unsafe e Critical

Come è stato descritto in precedenza la comunicazione tra nodi è essenziale al fine dell'esecuzione di un piano. Come verrà descritto nel dettaglio nella Sezione 4.6 i nodi interessati all'esecuzione di azioni nei tier topologicamente sottostanti vengono notificati a procedura terminata.

Talvolta è necessario dare un preavviso prima di eseguire una determinata azione su un nodo; tali azioni sono definite *unsafe*. Un esempio: un loadbalancer a fronte di più application server deve essere avvisato dello spegnimento di uno di questi *prima* che questo sia di fatto spento, in quanto durante lo spegnimento e, in generale, durante il transitorio possono essere perse alcune richieste. Per questo motivo, l'azione `stop` è definita come unsafe. Un nodo viene dichiarato unsafe fintanto che su di esso devono essere eseguite azioni di questo tipo.
Viene definita *critical* una azione che non solo è unsafe ma che necessita per la corretta esecuzione che il servizio sia spento. In altre parole esistono alcune azioni che devono essere precedute da uno `stop` (se il servizio è acceso), quindi eseguite, e poi seguite da una `start` se il servizio era acceso. Esempi di questo tipo di azioni sono tutte quelle che modificano parametri non dinamici del servizio come, nel caso di un application server, il numero di thread disponibili. Queste riconfigurazioni per essere effettive necessitano di un riavvio del servizio. Poiché l'esecuzione delle azioni `start` e `stop` è vincolata alle azioni critical e quindi possono essere eseguite senza essere esplicitamente invocate nel piano, non è possibile che queste siano parametriche.

4.3 Riassunto: il Meta-Modello

Si propone in Figura 4.3 un diagramma UML che riassume quanto descritto finora.

4.4 EADl: ECoWare App Description Language

In questa sezione verrà presentato EADl, un linguaggio per la descrizione dello stack applicativo. Attraverso questo DSL l'amministratore di sistema può dichiarare la struttura dell'applicazione e dei tier. Si mostrerà la grammatica e verrà fornito un esempio basato su una classica 3-tier application.

4.4.1 Grammatica

Si riporta in Listato 4.1 la grammatica del linguaggio EADl. In prima instanza si dichiara si definisce il costrutto `app` che contiene un identificativo rappresentante il nome dell'applicazione ed una lista non vuota di tier. Si parla di lista e non di insieme poichè a livello semantico l'ordine con cui vengono dichiarati i tier rappresenta la gerarchia dello stack applicativo.

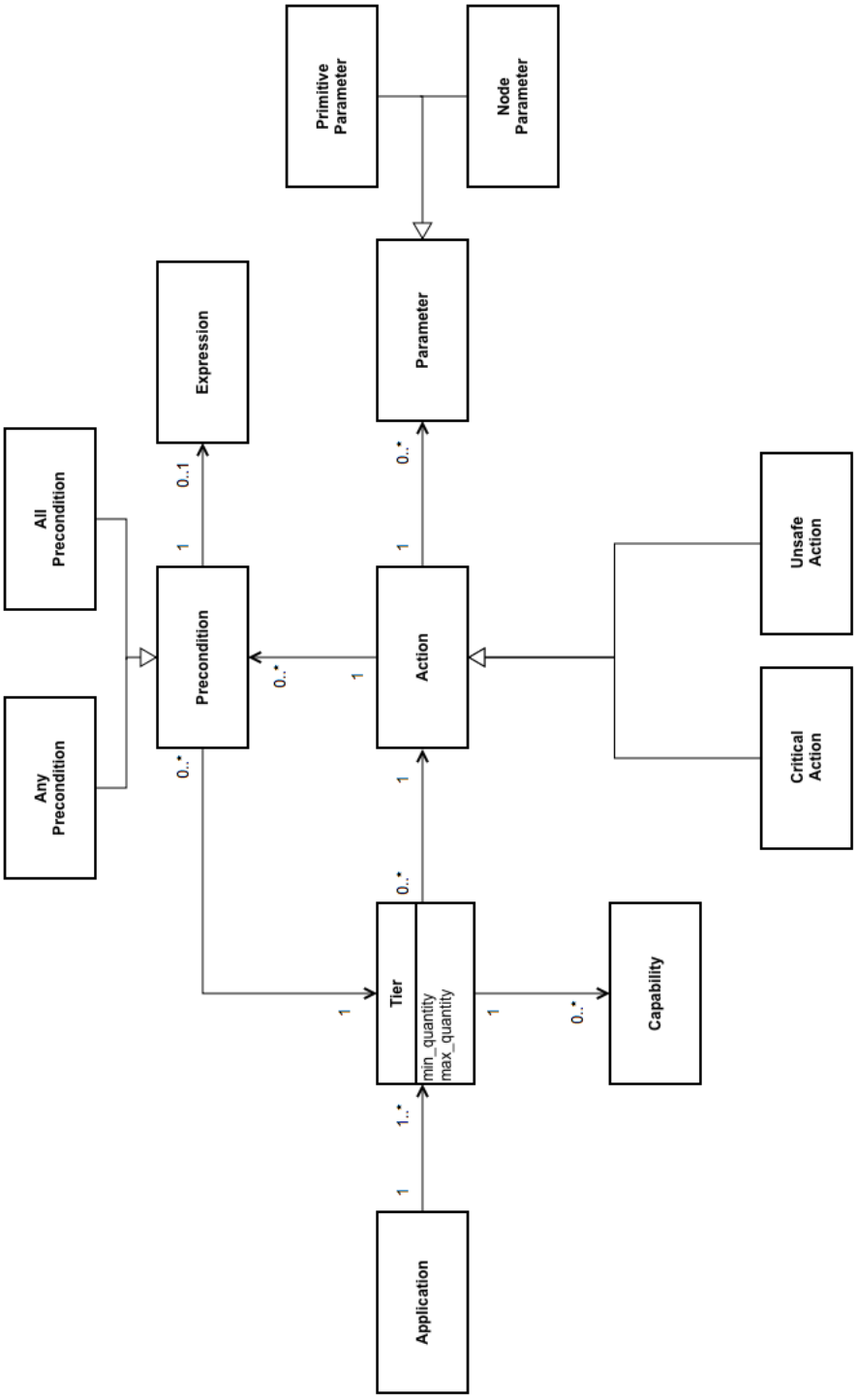


Figura 4.3: Il meta-modello

Listing 4.1: La grammatica di EADl

```
App -> 'app' ID ':' Tier+

Tier ->
        'tier' ID ':'
                ('capabilities:' ID (',' ID)*)?
                'minq:' INT
                'maxq:' INT
                ('actions:' Action+)?

Action ->
        ID '(' (Parameter (',' Parameter)* ')'  |
        ID '(' (Parameter (',' Parameter)* '):'
                ('preconditions:' Precondition+)?
                ('modifies:' ID (',' ID)*)?
                (critical | unsafe)?

Precondition -> (any | all) ID ('where' Expression)?

Parameter -> 'String' | 'Number' | 'Boolean' | ID

Expression -> Comp (LogicOp Expression)?

Comp ->
        ID CompOp Val |
        ('(' Expression ')')

Val -> STRING | BOOLEAN | INT | DOUBLE

CompOp -> '<' | '>' | '==' | '<=' | '>=' | '!='

LogicOp -> 'and' | 'or'

BOOLEAN -> 'true' |  'false'

DOUBLE ->  INT '.' INT

INT -> [0-9]+

ID -> [a-zA-z]+ (INT | [a-zA-z]+)*
```

Un tier dichiarato più in alto può dipendere da quelli sottostanti e non viceversa. Il costrutto `tier` è caratterizzato dal nome del tier e contiene i campi: `capabilities` facoltativo, `minq`, `maxq` e `actions` facoltativo. Con il campo `capabilities` vengono dichiarate le capability personalizzate che si aggiungono alle quattro predefinite (Sezione 4.2.1). I campi obbligatori `minq` e `maxq` determinano la quantità minima e massima di macchine accettate nel tier; `minq` inoltre viene utilizzato per il deploy iniziale: ricevuta la descrizione, il sistema, se necessario, allocherà un numero di macchine pari alla quantitià minima.
Il costrutto `actions` permette di dichiarare un insieme di azioni personalizzate o modificare quelle predefinite aggiungendo, ove possibile, parametri e informazioni aggiuntivi. Per ciascuna azione si dichiara la signature: un nome e un insieme (anche vuoto) di parametri espliciti separati da virgola e racchiuso tra parentesi tonde. I parametri possono essere di tipo Number, String o Boolean oppure può essere indicato il nome di un tier per i parametri di tipo nodo. La signature dichiarata deve essere rispetta dall'invocazione nel piano. Le precondizioni sono descritte dalla keyword `any` o `all` seguite dal nome del tier e dal campo opzionale `where` contente una espressione. Le espressioni sono costituite da comparazioni tra capability e valori costanti (ad esempio `x==true` o `y>10`, dove x e y sono capability), possono essere concatenate da operatori logici e annidate con parentesi. Se una azione modifica o può modificare una capability è buona pratica inserire questa nella lista `modifies`[2]. L'azione `alloc` modifica l'ip, la configurazione e l'id mentre le azioni `start` e `stop` modificano la capability `started`. Una azione può essere inoltre dichiarata `critical` o `unsafe`, la action `stop` è unsafe per definizione.

4.4.2 EADl: un esempio

Si propone in Listato 4.2 un esempio di un application stack composto da tre tier: loadbalancer, application server e data tier. Come già premesso in Sezione 4.4.1 le capability e le action predefinite, quest'ultime se non modificate, sono implicite; non è quindi necessario doverle dichiarare.
Il primo tier, chiamato `LOADBALANCER`, possiede un'unica istanza e dipende dal tier `APP_SERVER` poiché deve tener traccia degli application server accesi a cui poter smistare le richieste. Per far ciò il loadbalancer espone tre azioni: `updateServerList` che richiede, tramite precondizione, tutti gli application

[2]Il campo `modifies` non è utilizzato dal processing model presentato in questa tesi ma permette una maggiore comprensione di ciò che ciascuna azione compie. Questo costrutto sarà centrale per il fault detection e per le reazioni automatiche che verrano aggiunte al lavoro per la pubblicazione di un nuovo paper.

Listing 4.2: Un esempio di app

```
app Example:
  tier LOADBALANCER:
    minq: 1
    maxq: 1
    actions:
      alloc():
        preconditions:
          all APP_SERVER
            where started==true
      updateServerList():
        preconditions:
          all APP_SERVER
            where started==true
      addServer(APP_SERVER)
      removeServer(APP_SERVER)
      setAlgorithm(String):
        unsafe
  tier APP_SERVER:
    minq: 1
    maxq: 10
    actions:
      alloc():
        preconditions:
            any DB
              where started==true
      changeDatabase(DB):
        critical
      setMemoryAllocation(Number, Number):
        critical
      setMaxThreads(Number):
        critical
      setQueueLength(Number):
        critical
      setDBPoolSize(Number, Number):
        critical
      clearLogFile()
      deleteLogFile()
  tier DB:
    capabilities: port, username, password
    minq: 1
    maxq: 1
    actions:
      alloc():
        modifies: port, username, password
      setMaxConnection(Number)
      setMemLock(Boolean):
        critical
```

server accesi, `addServer` e `removeServer` per l'aggiunta o la rimozione manuale di un particolare nodo. Le ultime due azioni sono ridondanti rispetto a `updateServerList` ma sono riportate a titolo esemplificativo. È disponibile inoltre l'azione `setAlgorithm` che permette di modificare l'algoritmo di smistamento delle richieste (round robin, hardware aware[3]) ed è definita come unsafe in quanto, nel transitorio, il loadbalancer può risultare instabile. Viene ridefinita anche l'azione `alloc` con l'aggiunta di una precondizione di tipo `all`: in questo modo all'avvio della macchina vengono trovati tutti gli application server attivi.

Il secondo tier, chiamato `APP_SERVER`, è quello dedicato agli application server costituito da un numero di macchine compreso tra 1 e 10. Il tier presenta una dipendenza con il tier `DB` poichè ciascun application server necessita dei dati di un database per funzionare correttamente. Per questa ragione l'azione `alloc` contiene una precondizione di tipo `any`: durante l'allocazione di un application server è necessario ricevere l'indirizzo ip, la porta, e le credenziali di un macchina del tier `DB`. A ciò si aggiunge l'azione `changeDatabase` che possiede un parametro di tipo nodo per la modifica a runtime del database a cui collegarsi. Si espongono inoltre quattro azioni critical che modificano parametri non dinamici (non modificabili a servizio acceso): `setMemoryAllocation`, `setMaxThreads`, `setQueueLength` e `setDBPoolSize`. Sono presenti anche due azioni per la gestione del file di log: `clearLogFile` e `deleteLogFile`.

Il terzo ed ultimo tier, `DB`, è quello relativo al database, per semplicità composto da una sola macchina, che oltre alle capability predefinite aggiunge: `port`, `username` e `password`, necessarie agli application server. Oltre all'azione `alloc` che inizializza tutte le capability sono esposte due azioni per la configurazione della macchina: `setMaxConnection` e `setMemLock`, quest'ultima critical.

Attraverso questo semplice esempio sono stati messi in luce alcuni dei meccanismi di funzionamento dell'executor che, attraverso un semplice linguaggio, definisce un'astrazione alla gestione di un'applicazione cloud sia a livello infrastrutturale che di piattaforma senza considerarli compartimenti stagni bensì due aspetti di un unico problema.

[3]Un loadbalancer che utilizza una strategia hardware aware smista le richieste tenendo conto delle caratteristiche prestazionali e tecniche delle macchine a disposizione.

Listing 4.3: La grammatica di EPDl

```
Plan -> PlanAction+

PlanAction ->
      ID actionName=ID '(' PlanArgument (',' PlanArgument)* ')'

PlanArgument -> PlanVal | ID

PlanVal -> STRING | BOOLEAN | INT | DOUBLE

BOOLEAN -> 'true' |  'false'

DOUBLE ->  INT '.' INT

INT -> [0-9]+

ID -> [a-zA-z]+ (INT | [a-zA-z]+)*
```

4.5 EPDl: ECoWare Plan Description Language

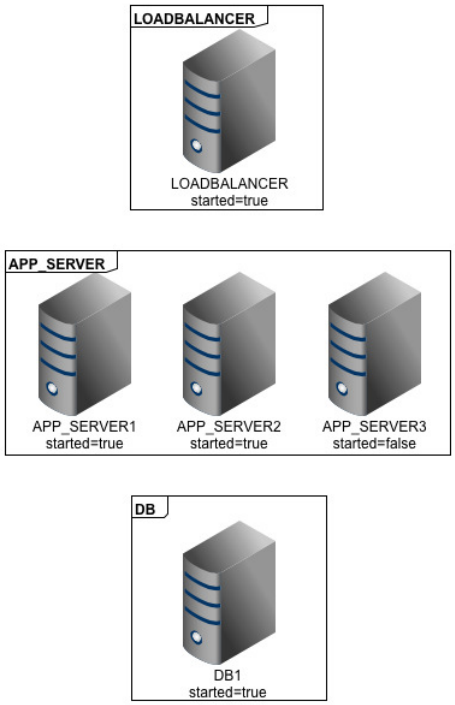
Il pianificatore comunica la strategia di adattamento all'esecutore attraverso piani d'esecuzione. Un piano è una lista di action da eseguire sui nodi della topologia. In questa sezione verrà presentato EPDl, un linguaggio per la descrizione del piano d'esecuzione. Si mostrerà la grammatica e verrà fornito un esempio che continua il caso di studio di Sezione 4.4.2.

4.5.1 Grammatica

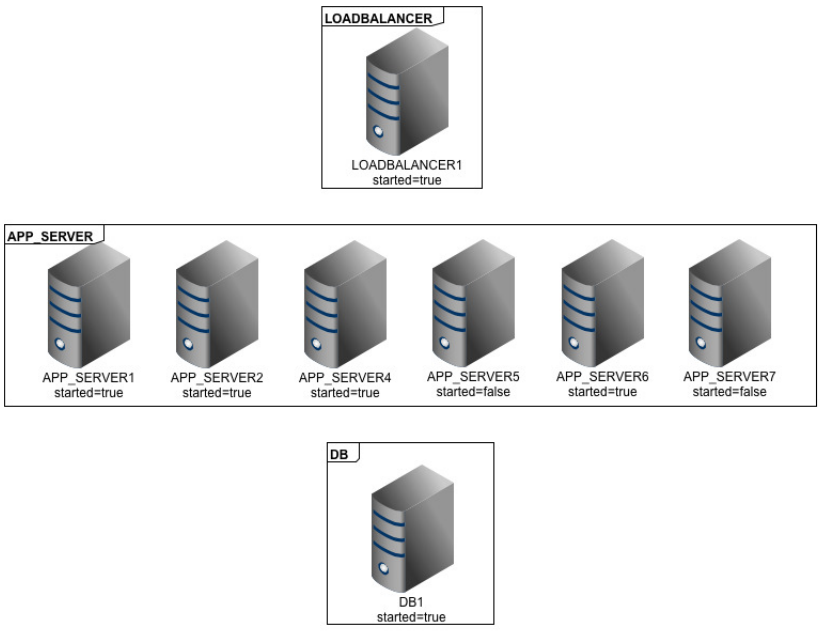
Ciascuna azione si riferisce ad un nodo, indicato attraverso il suo id (ad esempio `APP_SERVER1 stop()`). Fanno eccezione `alloc` e `new` in quanto il nodo che si vuole creare non esiste prima di eseguire l'azione e quindi non possiede un identificatore. Per invocare queste due azioni ci si riferisce generalemente al tier in cui si vuole aggiungere una macchina (ad esempio `APP_SERVER new()`). Le action del piano devono rispettare la signature dichiarata in EADl. L'ordine delle azioni non è importante in quanto, conoscendo l'application stack, sarà l'esecutore a deciderlo. Si riporta in Listato 4.3 la grammatica relativa alla stesura del piano.

4.5.2 EPDl: un esempio

Come si è mostrato, la grammatica del piano consiste in una lista, non vuota, di azioni con i relativi argomenti. Un argomento può essere di tipo primitivo oppure è possibile indicare l'id di un nodo. In Listato 4.4 viene riportato un esempio, la cui esecuzione porta dalla topologia in Figura 4.4a a quella in



(a) La topologia iniziale



(b) La topologia dopo l'esecuzione del piano

Figura 4.4: Un esempio di esecuzione di piano

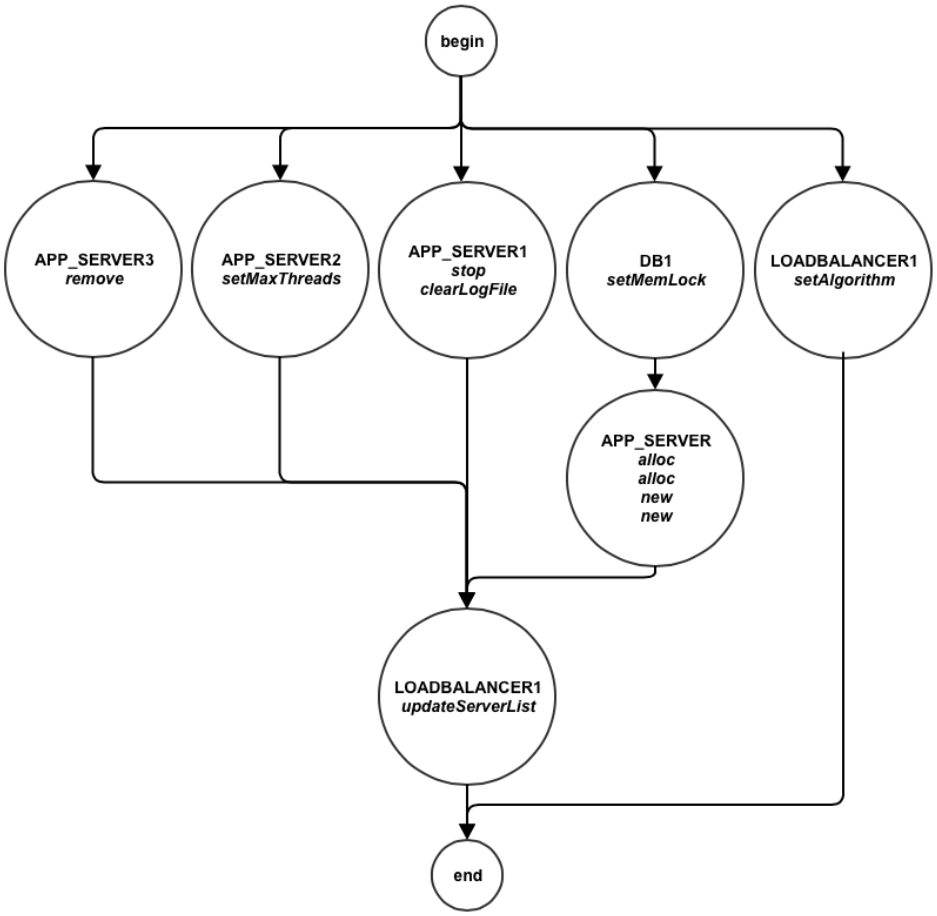


Figura 4.5: L'ordine di esecuzione

Figura 4.4b. In Figura 4.5 si mostra l'ordine di esecuzione delle azioni del piano. Inialmente vengono eseguite parallelamente tutte le azioni che non presentano dipendenze. Le action `alloc` e di conseguenza le `new` possiedono una precondizione `any` sul tier `DB` perciò devono aspettare il completamento dell'azione sul database. Infine l'azione `updateServerList` poichè possiede una precondizione `all` sul tier `APP_SERVER` deve aspettare che tutte le azioni su questo tier vengano eseguite.

4.6 Processing model

L'input dell'executor è un piano di cui si è mostrata la sintassi e la semantica, l'output è la topologia che deriva dall'esecuzione. L'esecuzione si basa sul concetto di coreografia: un nodo centrale riceve il piano, lo elabora, lo suddivide e invia a ciascun nodo la sua parte da eseguire. Come si è visto

Listing 4.4: Un esempio di piano

```
APP_SERVER new()
APP_SERVER alloc()
APP_SERVER new()
LOADBALANCER1 setAlgorithm("round_robin")
DB1 setMemLock(true)
LOADBALANCER1 updateServerList()
APP_SERVER2 setMaxThreads(25)
APP_SERVER1 stop()
APP_SERVER1 clearLogFile()
APP_SERVER3 remove()
APP_SERVER alloc()
```

ad un nodo di un tier possono servire dati di altri nodi ciò implica che è necessario considerare le dipendenze tra azioni.

4.6.1 L'architettura

Per comprendere come viene eseguito il piano è necessario definire l'architettura del sistema. L'executor è composto da un nodo centrale (Main Node) che riceve il piano e lo elabora. Su ogni macchina di ciascun tier invece è presente un agente composto da due parti: il Dispatcher e l'Actuator. Il Main Node comunica con l'agente di ciascuna macchina attraverso un bus (vedi Figura 4.6). Il Dispatcher è uguale per tutte le macchine, si occupa di gestire lo scambio di dati con il server, di gestire le capability e di invocare le azioni implementate sul Actuator che invece è tier-specific. Si è scelta una architettura e una modalità d'esecuzione distribuita in quanto può essere necessario gestire centinaia di macchine e azioni.

4.6.2 L'esecuzione del piano sul Main Node

Per l'elaborazione del piano sul nodo centrale si usa l'Algoritmo 1. L'algoritmo è diviso in cinque fasi:

- Fase 1: viene effettuato una ricerca degli agenti attualmente attivi.
- Fase 2: viene effettuato un controllo sui vincoli di quantità a partire dalla topologia corrente e dal numero di new, alloc e remove presenti nel piano.
- Fase 3: avviene la comunicazione con i provider IaaS per l'allocamento e la rimozione delle risorse. Ciò è propedeutico alle fasi successive. Non è possibile infatti eseguire azioni su un nodo se questo ancora non esiste o non è possibile calcolare correttamente le dipendenze se

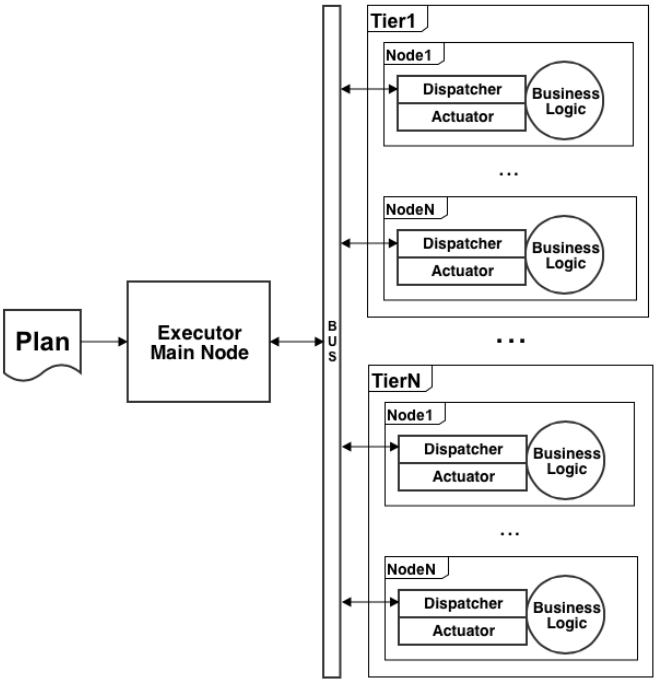


Figura 4.6: Schema dell'architettura distribuita

nella topologia compaiono nodi che saranno rimossi. Per ogni `alloc` e `new` viene allocato un nuovo nodo, mentre avviene la rimozione di quelli che compaiono in una `remove`. Per ciascuna azione di tipo `new` vengono aggiunte al piano una azione `alloc` ed una `start`. Si noti che la azione `alloc` comporta sia la creazione 'fisica' del nodo che, nelle fasi successive, l'esecuzione di una action omonima. Un procedimento simile avviene con l'azione `remove`: prima di effetture la richiesta di deallocazione viene notificato l'agente che può eseguire una azione di preparazione alla rimozione.

- Fase 4: viene creata la coreografia. Data la complessità della fase è necessario analizzarla in una sezione a parte (vedi Sezione 4.6.3).

- Fase 5: viene inviato sul bus un messaggio broadcast di inizio piano e si attendono i messaggi di completamento da parte dei nodi.

4.6.3 Fase 4: la coreografia

La fase che riguarda la creazione della coreografia è la più complessa delle cinque; in questa fase il piano viene suddiviso in parti da inviare ciascuna ai

Algorithm 1 elaboratePlan(plan)

```
// Phase 1
top ← getCurrentTopology()
// Phase 2
checkQuantityLimit(top, plan)
// Phase 3
for all new or alloc ∈ plan do
  createNewNode()
end for
for all remove ∈ plan do
  id ← getId(remove)
  destroyNode(id)
end for
for all new ∈ plan do
  decomposeNewAction(new, plan)
end for
// Phase 4
removeActionWithName('new', plan)
removeActionWithName('remove', plan)
started ← getStartedNode(top)
for all nodeId ∈ getInvolvedNode(plan) do
  actionsData ← new map()
  numOfCritical ← getCriticalCount(nodeId, plan)
  restartAfterCritical ← (numOfCritical ≥ 0) ∧ in(nodeId, started)
  for all action ∈ getActions(nodeId, plan) do
    if type(action, 'start') ∧ numOfCritical ≥ 0 then
      restartAfterCritical ← true
      continue
    end if
    if type(action, 'stop') ∧ numOfCritical ≥ 0 then
      restartAfterCritical ← false
      continue
    end if
    dep ← calculateDependency(action, plan)
    args ← parseArgs(action)
    put(actionsData, action, < dep, args >)
  end for
  send(nodeId, 'prepare', actionsData, restartAfterCritical, numOfCritical)
end for
// Phase 5
send('broadcast', 'start plan')
wait until completition
return updateTopology()
```

nodi coinvolti. Innanzitutto vengono rimosse dal piano le azioni già eseguite o quelle superflue: `new` e `remove`; inoltre vengono ottentuti, a partire dalla topologia corrente, i nodi con il servizio acceso (variabile `started`). L'insieme `started` serve per determinare come comportarsi in caso di azioni `critical`. Infatti si è scelto di minimizzare al massimo il numero di `start` e `stop` all'interno di una esecuzione: ciò deve avvinere al più una volta. Per far ciò vengono contante le azioni critical (variabile `numOfCritical`) e impostato un booleano, `restartAfterCritical` che determina se alla fine dell'esecuzione delle azioni critical è necessario far ripartire il servizio. Ciò è vero se:

- il nodo è acceso e il piano non contiene `stop`
- il nodo è spento e il piano contiene `start`

Inoltre in questa situazione non bisogna aggiungere al piano dedicato ad un nodo le azioni di start e stop in quanto attraverso la variabile `numCritical` il nodo saprà che alla prima azione critica dovrà eseguire uno `stop` e all'ultima, se `restartAfterCritical` è vera dovrà eseguire una `start`. Per ogni azione da eseguire bisogna inoltre calcolare gli argomenti e le dipendenze. Per quanto riguarda gli argomenti se sono primitivi vengono estratti dal piano altrimenti viene estratto l'id del nodo e si controlla che esiste. Più complessa è la questione delle dipendenze (vedi Algoritmo 2): si dice che esiste una dipendenza tra una action a ed un nodo n ovvero $(n \rightarrow a)$ se per eseguire a è necessario attendere il completamento dell'esecuzione delle azioni sul nodo n. Data una azione a con un lista di argomenti $args$ da eseguire su un nodo n_0:

- per ogni precondizione `any` p_1 su il tier t_1 possono verificarsi tre sottocasi:

$$\exists n_1 \in t_1 \mid satisfy(n_1, p_1) \wedge !\, isInvolved(n_1, plan) \implies \emptyset$$

$$((\nexists n_1 \in t_1 \mid satisfy(n_1, p_1) \wedge !\, isInvolved(n_1, plan)) \wedge \\ involvedNodeCount(t_1, plan) > 0) \implies \\ (\forall n_i \in t_1,\ isInvolved(n_i, plan) \implies n_i \rightarrow a)$$

$$((\nexists n_1 \in t_1 \mid satisfy(n_1, p_1) \wedge !\, isInvolved(n_1, plan)) \wedge \\ involvedNodeCount(t_1, plan) = 0) \implies cannot\ execute\ action$$

- per ogni precondizione `all` p_1 su il tier t_1 possono verificarsi due sottocasi:

$$involvedNodeCount(t_1, plan) = 0 \implies \emptyset$$

$$involvedNodeCount(t_1, plan) > 0 \implies$$
$$(\forall n_i \in t_1,\ isInvolved(n_i, plan) \implies n_i \to a)$$

- l'azione presenta argomenti non primitivi

$$\forall n \in args,\ isNode(n) \wedge isInvolved(n, plan) \implies n \to a$$

Si noti che i tre casi principali non sono esclusivi in quanto una azione può possedere sia precondizioni `all`, sia precondizioni `any` che argomenti non primitivi. Nel caso di precondizioni di tipo `any` se esiste un nodo che soddisfa la precondizione (ovvero soddisfa l'eventuale espressione) e tale nodo non è coinvolto nel piano allora l'azione può essere eseguita senza attendere, in caso invece ciò non sia vero allora esisterà una dipendenza con tutti i nodi coinvolti nel piano del tier specificato: ciò significa che bisogna 'sperare' che dopo l'esecuzione delle azioni la precondizione sarà soddisfatta. Se invece la precondizione non è soddisfatta e non esistono nodi del tier specificato coinvolti nel piano sicuramente l'azione non potrà essere eseguita. In caso di precondizione `all` esiste una dipendenza con tutti i nodi del tier in questione coinvolti nel piano. Il terzo caso dice che per ogni argomento di tipo nodo se tale nodo viene modificato durante l'esecuzione (esiste un azione da eseguire sul nodo) allora esiste una dipendenza.

4.6.4 L'esecuzione del piano sui nodi

L'agente distrubito, come abbiamo visto, entra in gioco a partire dalla fase 4 dell'esecuzione del piano, in quel momento riceve tutti i dati necessari per eseguire le azioni. In realtà non appena l'agente viene installato (Figura 4.7) esegue una richiesta per ottenere informazioni al nodo centrale con un evento di tipo 'tier info' che contiene: la lista di action dichiarate, le lista di action critiche ed unsafe e le precondizioni di ciascun action. L'agente inoltre può

Algorithm 2 calculateDependency(action, plan)

```
deps ← new list
for all precondition ∈ getPreconditions(action) do
  tierName ← getTier(precondition)
  nodeCount ← involvedNodeCount(tierName, plan)
  if isAny(precondition) then
    expr ← getExpr(precondition)
    nodes ← send(tierName, 'get', expr)
    statisfied ← false
    for all nodeId ∈ nodes do
      if !isInvolved(nodeId, plan) then
        statisfied ← true
      end if
    end for
    if !statisfied then
      if nodeCount ≤ 0 then error
      end if
      add(deps, createDeps(action, tierName, nodeCount))
    end if
  else
    if nodeCount > 0 then
      add(deps, createDeps(action, tierName, nodeCount))
    end if
  end if
end for
for all arg ∈ getArguments(action) do
  if !isPrimitive(arg) then
    nodeId ← getNodeId(arg)
    if isInvolved(nodeId) then
      add(deps, createDep(action, nodeId))
    end if
  end if
end for
```

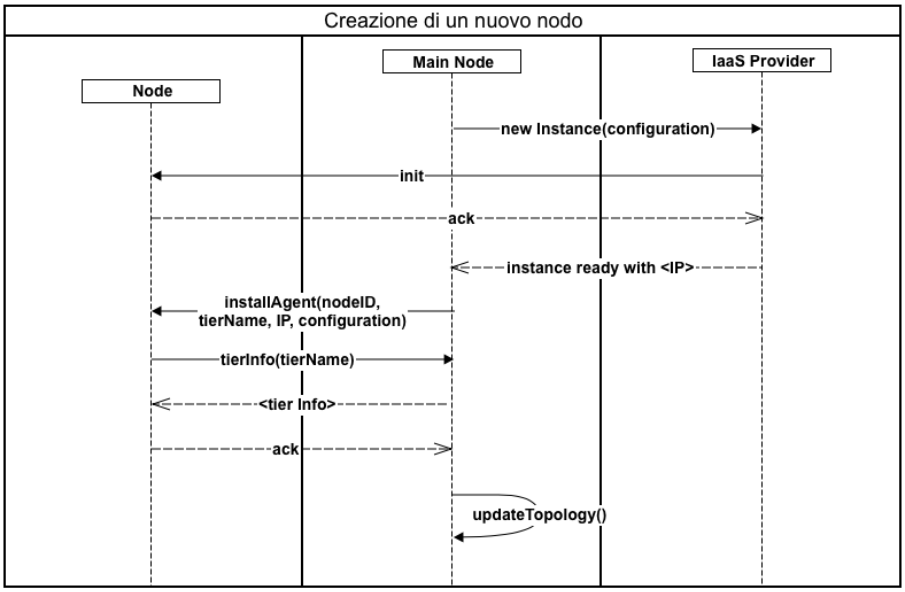


Figura 4.7: La creazione di un nuovo nodo

ricevere cinque tipi di eventi dal bus che coincidono con le più importanti funzionalità:

- Evento 'get': rappresenta la richiesta di un nodo di ottenere una o più capability. Esso viene usato per ottenere argomenti di tipo nodo oppure per soddisfare le precondition di una azione. Può contenere una espressione da verificare.

- Evento 'prepare': come si mostra in Algoritmo 1 questo evento rappresenta l'invio della parte di piano che esso deve eseguire. Vengono comunicate le azioni da eseguire, gli argomenti associati, le eventuali dipendenze presenti, il numero di azioni critiche e, se presenti, se è necessario un riavvio dopo averle eseguite.

- Evento 'start plan': è l'avviso inviato dal nodo centrale che si può iniziare il piano, verrà descritto meglio successivamente.

- Evento 'safe warning': è il messaggio di un nodo che entra o che esce da uno stato di instabilità (vedi Sezione 4.2.4).

- Evento 'completion': viene ricevuto quando un nodo ha completato la propria esecuzione, può servire a eliminare dipendenze tra le azioni in attesa di essere eseguite.

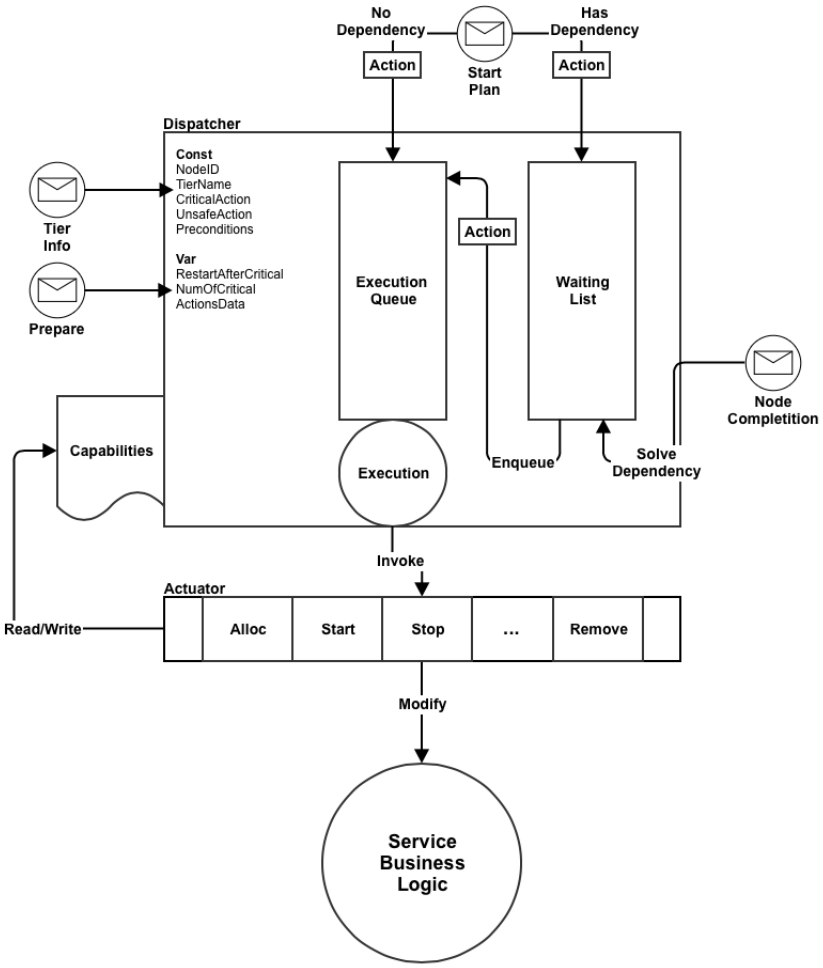


Figura 4.8: Il nodo distribuito

In Figura 4.8 viene mostrata l'architettura del nodo distribuito ponendo l'accento sul dispatcher. I messaggi 'tier info' e 'prepare' assegnano le proprietà del dispatcher. Esso inoltre contiene tre componenti principali:

- una lista chiamata *Waiting List* contente tutte le azioni che possiedono dipendenze da risolvere e quindi non ancora pronte per la fase di esecuzione;
- una coda di azioni pronte per l'esecuzione chiamata *Execution Queue*;
- un'unità di esecuzione rappresentata nel dettaglio in Figura 4.9.

Non appena viene ricevuto il messaggio 'start plan' vengono smistate le azioni: quelle senza dipendenze vengo aggiunte alla Execution Queue, le

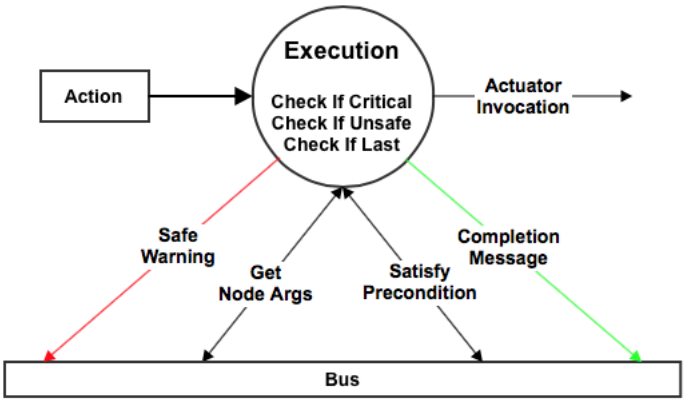


Figura 4.9: L'unità d'esecuzione

altre alla Waiting List. L'unità di esecuzione esegue una azione alla volta affinché le modifiche alle capability siano consistenti.
L'unita di esecuzione invoca le vere e proprie azioni sull'Actuator che contiene l'implementazione fornita dall'utente. L'actuator può accedere in lettura e scrittura alle capability e alla logica di business del servizio associato al nodo (e al tier). L'unità di azione possiede le seguenti funzionalità:

- Attraverso il bus ottiene eventuali nodi di tipo argomento (utilizzando eventi 'get').
- Prova a soddisfare le precondizioni dell'azione; in caso di azioni di tipo `any` può non essere possibile e ne consegue il fallimento dell'esecuzione del piano.
- Controlla se l'azione è di tipo `unsafe` e che lo stato del nodo non sia già stato impostato come instabile. In tal caso viene inviato un messaggio 'safe warning' sul bus e si attende un tempo determinato in fase implementativa[4] prima di eseguire l'azione.
- Controlla se l'azione è `critical`, in tal caso se il servizio è accesso viene eseguita l'azione `stop` e poi si esegue l'azione.
- Al termine dell'esecuzione di una azione vengono effettuali i seguenti controlli:
  - Se l'azione eseguita è l'ultima `critical` (si controlla la proprietà `NumOfCritical` che viene adeguatamente decrementata) e la

[4] È stato scelto 5 secondi.

proprietà `RestartAfterCritical` è vera allora si esegue l'azione `start`.

– Se l'azione è l'ultima in assoluto viene inviato l'evento 'completion' per notificare l'avvenuta esecuzione del piano. In caso lo stato del nodo sia instabile viene inviato sul bus un evento 'safe warning' per avvertire della stabilizzazione del nodo.

Le azioni rimangono nella Waiting List fintanto che le dipendenze non sono risolte. Ciò avviene attraverso la ricezioni di eventi 'completition' provenienti da altri nodi; quando questo avviene si scorrono tutte le azioni e nel caso sia presente una dipendenza con il nodo da cui si è ricevuto il messaggio si elimina la dipendenza. Le azioni che non presentano più dipendenze vengono aggiunte alla Execution Queue.

Capitolo 5

Implementazione

In questo capito si mostrano le tecnologie e le tecniche utilizzate per l'implementazione della soluzione. L'ambiente di sviluppo utilizzato è Eclipse e la scelta del linguaggio di programmazione è naturalmente ricaduta su Java.

5.1 XText ed i DSL

Dovo aver progettato le grammatiche sono stati implementati i linguaggi per la descrizione dell'application stack e del piano: *EADl* e *EPDl*. A questo proposito si è fatto uso di XText, un framework per lo sviluppo di linguaggi che si integra nativamente in Eclipse. Grazie a questa tecnologia, dopo la scrittura della grammatica con un apposito meta-linguaggio, viene non solo generato un parsificatore ma anche un insieme di classi Java grazie alla quali è possibile facilmente navigare l'albero sintattico. Se da una parte XText include un validatore della sintassi, è stato necessario aggiungere due elementi: un validatore semantico ed un decoratore (vedi Figura 5.1). Il validatore semantico controlla che il significato del testo ricevuto sia consistente e che rispetti tutti i vincoli del modello: per esempio controlla che all'interno delle espressioni siano riportate solo capability dichiarate in un determinato tier oppure che un certo tier dipenda solamente da quelli inferiori. Il decoratore serve per aggiungere alla semantica dell'input ricevuto il significato implicito, ad esempio aggiunge le capability e le action predefinite. I parser sono utilizzati esclusivamente dal nodo centrale; i nodi distributi invece, nonostante siano a conoscenza di parte dell'application stack e della sua descrizione, non utilizzano direttamente le classi generate dal linguaggio ma richiedono i dati al nodo centrale che li restituisce sottoforma di

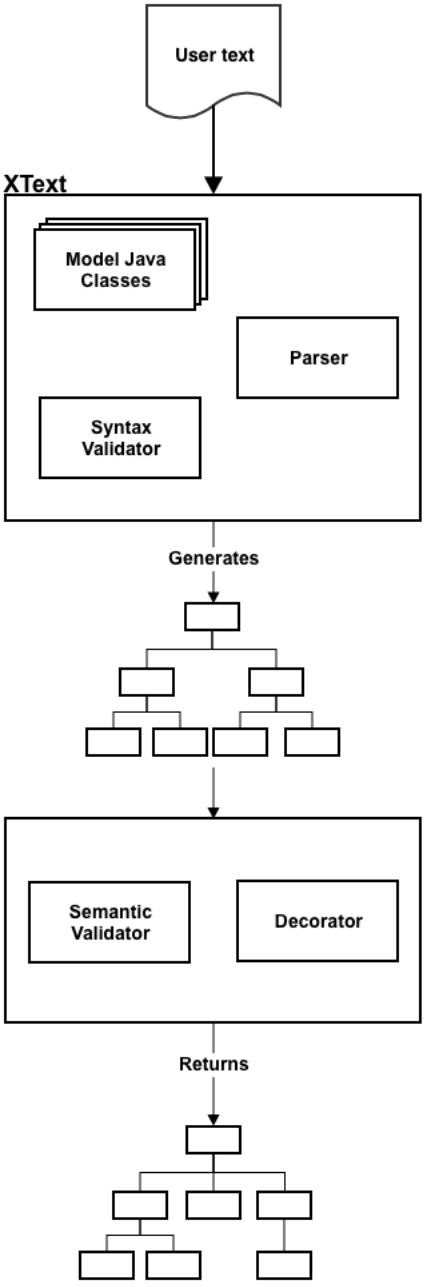


Figura 5.1: Gestione dei DSL

dati primitivi (vedi Sezione 4.6.4). In questo modo viene mantenuta una separazione logica tra le componenti.

5.2 RabbitMQ ed il bus

Il bus è una delle componenti più importanti dell'intero progetto, per questo è stato utilizzato RabbitMQ un software ormai consolidato a livello industriale. RabbitMQ è un middleware open source per lo scambio di messaggi basato sul protollo AMQP. La sua architettura è costituita da un server centrale, di cui è necessario conoscere l'indirizzo ip, e nodi distribuiti su cui deve essere installato un client. Questa tecnologia è stata utilizzata, non solo nell'executor ma in tutta l'architettura MAPE. Di RabbitMQ è disponibile un'implementazione Java che ha facilitato molto il lavoro. Tuttavia sono state apportate diverse modifiche per un miglior funzionamento tanto da creare un insieme di classi che creano un livello d'astrazione superiore che chiameremo ECoWare Bus Wrapper.
L'ECoWare Bus Wrapper, di cui si riporta l'UML in Figura 5.2, oltre a rendere più sintentico l'invio e la ricezione di messaggi fornisce gli strumenti per una comunicazione sincrona o asincrona multi-agente. È costituito da quattro classi principali: ECoWareMessage rappresenta il messaggio da inviare il cui body è una mappa stringa-oggetto, ECoWareRequestSender è una richiesta asincrona ad uno o più nodi del sistema su cui si specifica il tipo di evento, il mittente, una chiave di pubblicazione (per l'esempio l'id del nodo destinatario) e un numero atteso di risposte, ECoWareResponseSender è la risposta ad una precedente richiesta ed ECoWareConversation a cui è possibile aggiungere una o più richieste e rendere sincrona la conversazione (per ciascuna si aspetta il numero di risposte atteso fino al timeout specificato). Questa implementazione è stata di fondamentale importanza per l'intero sistema che si basa su dipendenze e sincronizzazioni da rispettare. Iscrivendosi ad una o più chiave di pubblicazione e facendo uso dell'ECoWareMessageReciver è possibile ricevere i messaggi sul bus. In particolare ciascun agente si iscrive a tre chiavi di pubblicazione:

- l'id del nodo: per essere contattato direttamente;
- il nome del tier: per le comunicazione riguardanti l'intero tier;
- 'broadcast': per le comunicazioni a tutta l'infrastruttura.

Gli eventi utilizzati sono numerosi e si riportano di seguito solo i più importanti:

- GET: è l'evento inviato ad un nodo distribuito per la richieste di dati. Il body della richiesta contiene il nome della capablility desiderata o la keyword `all` se servono tutte ed una espressione opzionale da verificare, in Listato 5.1 si mostra un esempio di richiesta;

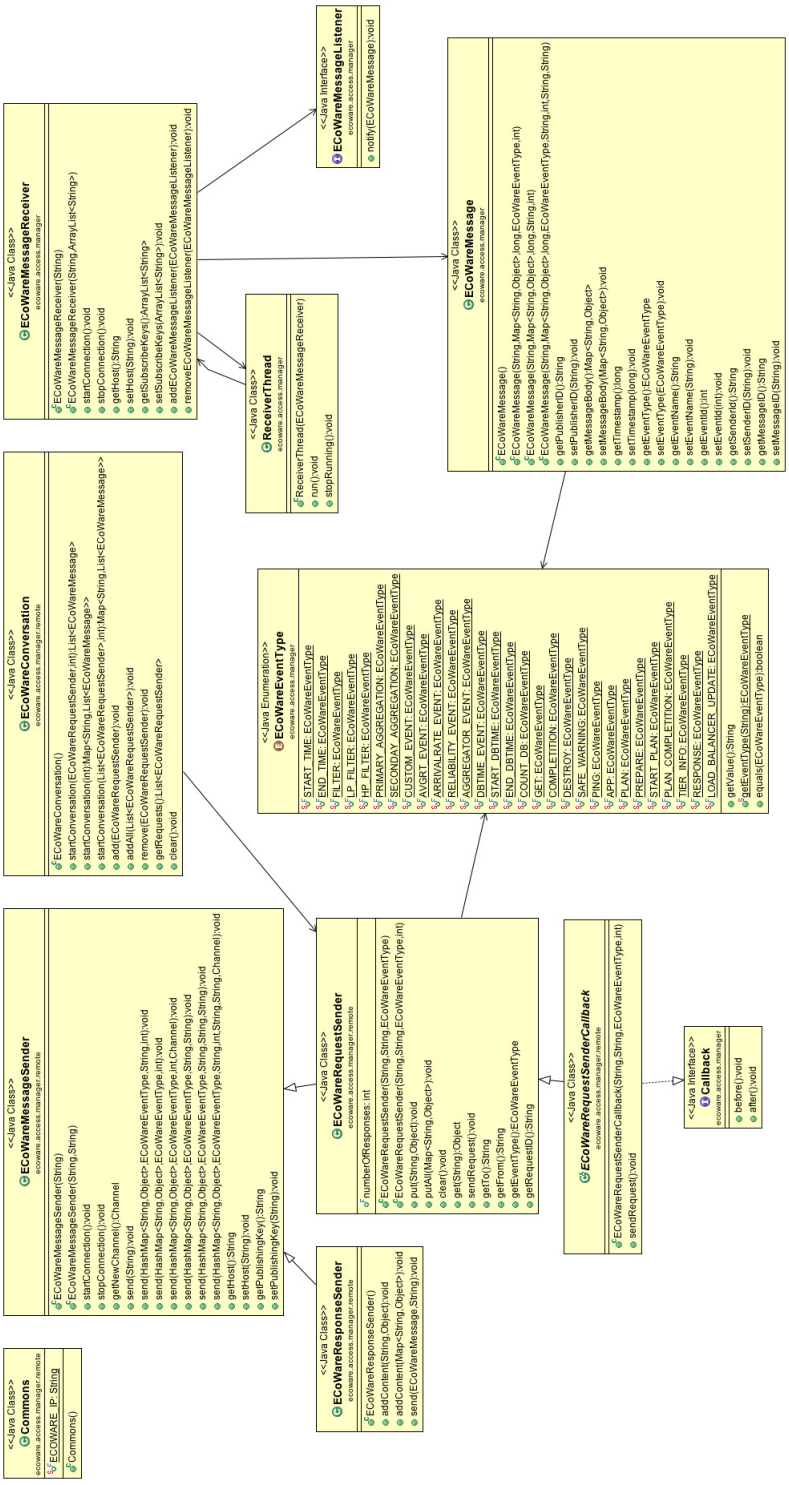


Figura 5.2: L'UML dell'ECoWare Bus Wrapper

Listing 5.1: Un esempio di richesta GET verso il nodo JBOSS3

```
ECoWareRequestSender request = new ECoWareRequestSender(this.nodeID, "
    JBOSS3", ECoWareEventType.GET, 1);
request.put("capability", "all");
reqiest.put("expr", expr1);
ECoWareConversation c= new ECoWareConversation();
ECoWareMessage message = c.startConversation(request, 15000).get(0);
Map<String, Object> node=message.getMessageBody().get("value");
System.out.println("Node id: "+node.get("id")+" started: "+node.get("
    started"));
```

- PING: è un generico evento valido per tutti i nodi usato per il fault detection e controlli di vario genere;
- PREPARE: è l'evento, già descritto nel dettaglio in Sezione 4.6.3, in cui vengono inviati ai nodi distribuiti i dati necessari per l'esecuzione del piano;
- START_PLAN: per l'avvio distribuito del piano;
- SAFE_WARNING: è l'evento che consegue all'entrata o all'uscita di un nodo da un periodo di instabilità. Contiene un booleano che defisce di che caso si tratti e tutte le capability del nodo;
- COMPLETITION: è l'evento scaturito dalla terminazione della propria parte di piano da parte di un nodo distribuito. Attraverso questo evento vengono gestite le dipendenze tra nodi (Sezione 4.6.4);
- DESTROY: è il messaggio inviato dal nodo centrale ad un nodo distribuito affinché venga eseguita l'eliminazione del nodo stesso;
- PLAN_COMPLETITION: è l'evento di conclusione inviato dal nodo centrale che sancisce la fine dell'esecuzione del piano, contiene la topologia aggiornata.

5.3 L'implementazione del Main Node

In Figura 5.3 si mostra l'implementazione del nodo centrale. Esso è costituito da cinque componenti: EWXServer, PlanExecutor, TopologyManager AllocatorManager e DeallocatorManager.
L'EWXServer (EWX sta per ECoWare Executor) si occupa di ricevere, tramite bus, la descrizione dell'applicazione che viene navigata, validata e ne viene salvato l'albero sintattico. In secondo luogo riceve i piani dal Planner

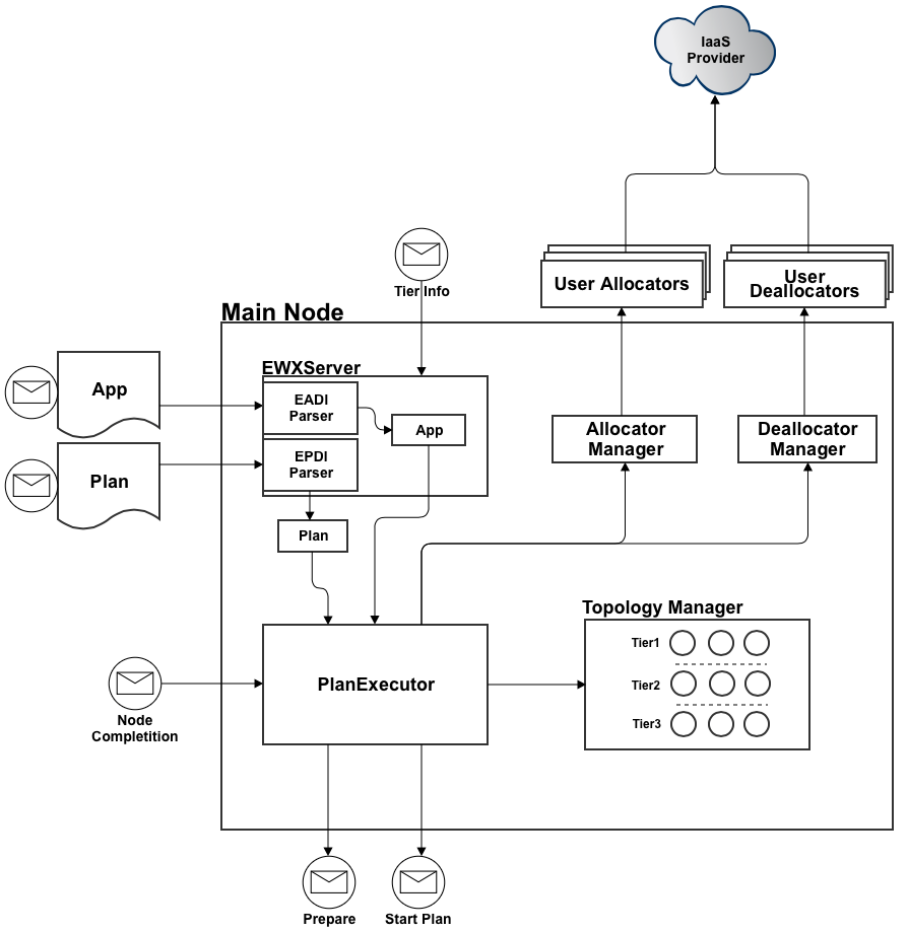


Figura 5.3: L'implementazione del Main Node

su cui viene effettuato il parsing, analogalmente all'applicazione. Dopo aver terminato l'elaborazione, il piano e l'applicazione vengono passati al Plan Executor per l'esecuzione. Inoltre è la componente che risponde agli eventi 'tier info' (Sezione 4.6.4) inviati degli agenti distribuiti.

Il PlanExecutor è la componente che si occupa di creare una coreografia (Sezione 3.3.2) per la vera e propria esecuzione. Implementa gli Algoritmi 1 e 2 utilizzando tutti i meccanismi di comunicazione menzionati in Sezione 5.2. In Listato 5.2 si mostra l'implementazione della fase 4 dell'algoritmo di esecuzione (Sezione 4.6.3). Oltre a quanto già descritto, si occupa di tener traccia dei messaggi 'completition' inviati dagli agenti, in questo modo attende la terminazione di tutti ed eventualmente produce un messaggio di errore se il tempo d'esecuzione supera una soglia. Il PlanExecutor utilizza il TopologyManager per la gestione della topologia e l'AllocatorManager e

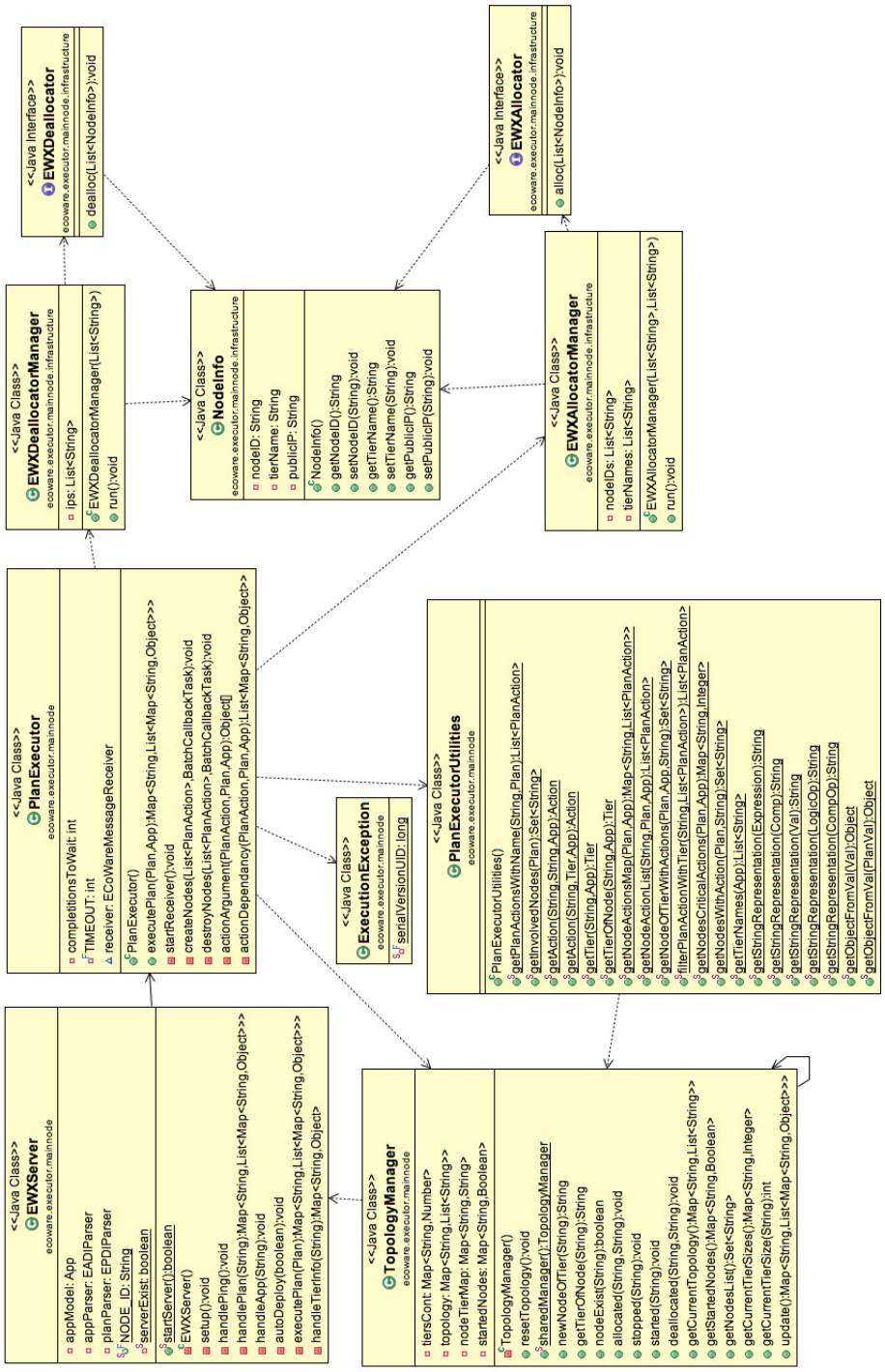


Figura 5.4: L'UML del nodo centrale

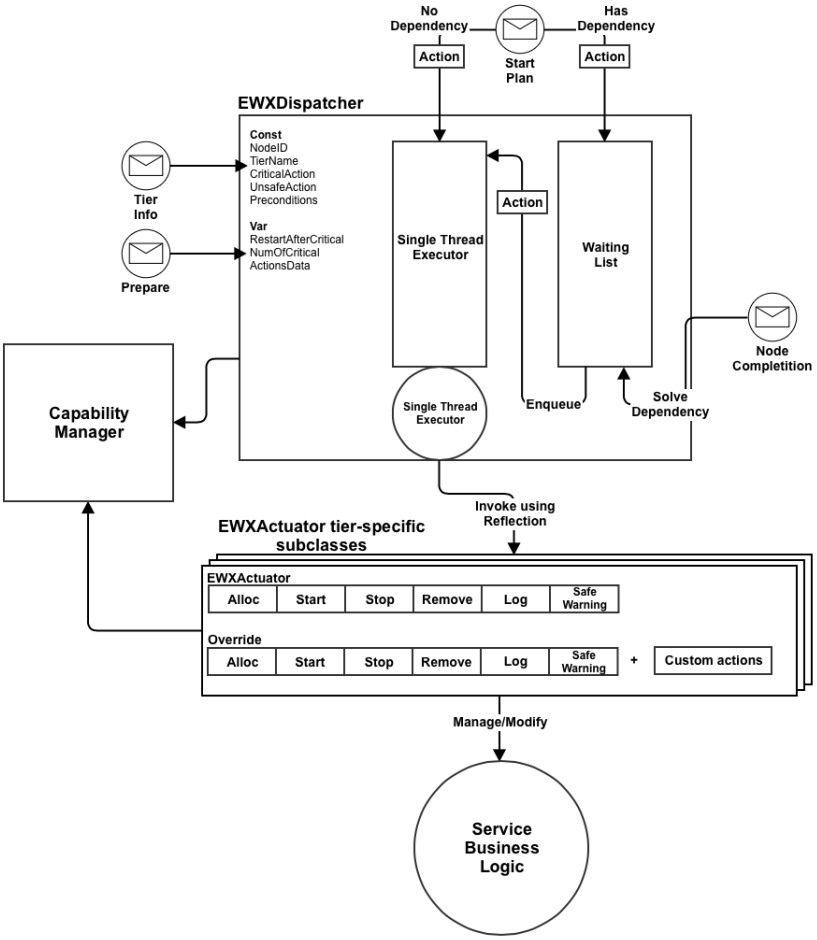


Figura 5.5: L'implementazione dell'agente

il DeallocatorManager per l'interfacciamento ai provider IaaS.

TopologyManager gestisce la topologia dei nodi, tiene traccia di quelli esistenti e di quelli con servizio acceso, produce nuovi identificatori per i nodi da allocare e, quando necessario (ad esempio non appena viene creato), controlla l'esistenza di nodi già attivi.

AllocatorManager e DeallocatorManager sono le due componenti che si interfacciano ai provider IaaS invocando gli allocatori e deallocatori implementati dall'utente. Sono presenti infatti due interfacce: EWXAllocator e EWXDeallocator che l'utente implementa per interfacciarsi via codice al provider preferito.

In Figura 5.4 è riportato l'UML del nodo centrale.

Listing 5.2: La fase di coreagrafia nel nodo centrale

```
completitionsToWait=0;
Map<String, Boolean> started=TopologyManager.sharedManager().
    getStartedNodes();
Map<String, Integer> criticals=PlanExecutorUtilities.
    getNodesCriticalActions(plan, appStack);
Set<String> nodes=TopologyManager.sharedManager().getNodesList();
Map<String, List<PlanAction>> actionsMap = PlanExecutorUtilities.
    getNodeActionsMap(plan, appStack);
Map<String, Integer> tierSizes=TopologyManager.sharedManager().
    getCurrentTierSizes();
ECoWareConversation conversation = new ECoWareConversation();
for(String nodeId : nodes){
  List<PlanAction> nodeActions=actionsMap.get(nodeId);
  boolean isCritical=criticals.containsKey(nodeId);
  boolean restartAfterCritical=started.get(nodeId) && isCritical;
  boolean firstCritical=true;
  Map<String, List<Map<String, Object>>> actionDeps=new HashMap<String
      ,List<Map<String, Object>>>();
  Map<String, Object[]> actionArgs=new HashMap<String, Object[]>();
  List<String> actionList=new ArrayList<String>();
  if(nodeActions!=null)
    for(PlanAction action : nodeActions){
      if(action.getActionName().equals("start")  && isCritical){
        if(firstCritical)
          num_starts++;
        restartAfterCritical=true;
        firstCritical=false;
        continue;
      }
      if(action.getActionName().equals("stop")  && isCritical){
        if(firstCritical)
          num_stops++;
        restartAfterCritical=false;
        firstCritical=false;
        continue;
      }
      actionDeps.put(action.getActionName(), actionDependancy(action,
          plan, app));
      actionArgs.put(action.getActionName(), actionArgument(action,
          plan, app));
      actionList.add(action.getActionName());
    }
  if(actionList.isEmpty())
    continue;
  else
    completitionsToWait++;
  ECoWareRequestSender request = new ECoWareRequestSender(EWXServer.
      NODE_ID, nodeId, ECoWareEventType.PREPARE, 1);
  request.put("action_list", actionList);
  request.put("action_deps", actionDeps);
  request.put("action_args", actionArgs);
  request.put("restart_after_critical", restartAfterCritical);
  request.put("tier_sizes", tierSizes);
  Integer i=criticals.get(nodeId);
  request.put("critical", i);
  conversation.add(request);
}
conversation.startConversation(10000); //sends all the 'PREPARE'
    events
ECoWareRequestSender request = new ECoWareRequestSender(EWXServer.
    NODE_ID, "broadcast", ECoWareEventType.START_PLAN);
request.sendRequest(); // starts the distributed execution
```

5.4 L'implementazione dell'agente distribuito

L'implementazione dell'agente, che viene mostrata in Figura 5.5, si compone di tre componenti principali: EWXDispatcher, EWXActuator e CapabilityManager.
EWXDispatcher è l'implementazione del modello illustrato in Sezione 4.6.4; questo oggetto è identico per tutti i nodi e si occupa di ricevere il piano dal MainNode, di rispondere alle richieste di altri agenti riguardanti le proprie capability e partecipa all'effettiva esecuzione del piano. Se per l'implementazione della WaitingList si utilizzano strutture-dati comuni in Java, come *Java List* o *Java Map*, per l'Execution Queue e l'unità di esecuzione si fa uso del package *java.util.concurrent*. In particolare ci si avvale di un *ExecutorService* dotato di un singolo thread in esecuzione ed una coda di oggetti Runnable in attesa, in questo modo le azioni vengono eseguite in maniera sequenziale. Ciascuna azione, non appena è scevra di dipendenze, viene innestata all'interno di un oggetto Runnable ed aggiunta alla coda dell'Executor Service. Prima di eseguire la vera e propria azione viene eseguito il parsing degli argomenti e la risoluzione delle precondizioni. Per quest'ultime, e in caso di argomenti di tipo nodo, si ottengono i dati necessari attraverso l'invio di eventi 'GET' sul bus verso altri agenti. I nodi sono rappresentati come mappe stringa-oggetto contenenti le capability. Si noti che il parsing degli argomenti complessi viene fatto in maniera distribuita per due motivi: migliora la scalibilità del sistema e permette di rispettare le dipendenze tra azioni; infatti è possibile che il nodo passato come argomento sia anch'esso modificato durante il piano (ovvero esiste una dipendenza) e quindi sarebbe sbagliato inviarlo in fase di preparazione. In Listato 5.4 viene mostrato il metodo *checkForAvaliableActions* che viene chiamato dopo ogni qual volta vengono rimosse delle dipendenze dalle azioni nella WaitingList. È importante notare come tutte le azioni sono dipendenti dalla azione `alloc`; se questa è in attesa (perché presenta ancora delle dipendenze) nessun'altra azione può essere eseguita.
In Listato 5.3 si mostra l'implementazione del metodo *exec* in cui si possono notare tutte le funzionalità espresse in fase di modello in Sezione 4.6.4.
L'esecuzione effettiva delle azioni avviene nell'Actuator. Come mostrato in Figura 5.5 l'utente deve implementare sottoclassi della classe EWXActuator in cui è necessario fare l'ovveride delle azioni predefinite e implementare le azioni personalizzate. Sarà presente un attuatore per ogni tier. Il Dispatcher possiede un oggetto EWXActuator che viene inizializzato utilizzando la segunte convezione: EWXActuator<nome del tier>. Aldilà di quanto esposto in Sezione 4.2.2 sono state aggiunte due ulteriori action predefinite:

Listing 5.3: L'implementazione del metodo exec

```
private void exec(final String actionName, final Object[] args)
{
    if(deallocated)
      return;
    singleThreadExecutor.execute(new Runnable(){
      @Override
      public void run() {
        Object[] actionParams = processArgs(args);
        actionParams=checkPreconditions(actionName, actionParams);
        if(actionParams==null)
          try {
            throw new Exception("Preconditions not satisfied");
          } catch (Exception e1) {
            e1.printStackTrace();
          }
        checkUnsafe(actionName);
        try {
          Method m=getMethod(actionName, actionParams);
          if(m==null)
            throw new Exception("Method not valid");
          if(!actionName.equals("stop") && criticalActions.contains(
              actionName)
              && ((Boolean) getCapability("started", null))==true){
            checkUnsafe("stop");
            actuator.stop();
          }
          m.invoke(actuator, actionParams);
          if(actionName.equals("alloc")){
            allocated=true;
          }
          if(criticalActions.contains(actionName))
            criticalToDo--;
          if(criticalToDo==0 && restartAfterCritical && ((Boolean)
              getCapability("started", null))==false){
            actuator.start();
          }
          actionToDo--;
          if(actionToDo==0)
            sendCompletitionNotification();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
}
```

Listing 5.4: L'implementazione del metodo checkForAvaliableActions

```
private void checkForAvaliableActions(){
  if(actionToDo==0 || waitingList==null || waitingList.isEmpty())
    return;
  List<String> actionListCloned = new ArrayList<String>(waitingList);
  for(String action: actionListCloned){
    List<Map<String, Object>> dep=actionDeps.get(action);
    if((allocated==true || action.equals("alloc")) && (dep==null ||
        dep.isEmpty())){
      Object[] args=actionArgs.get(action);
      if(args==null)
        args=new Object[0];
      waitingList.remove(action);
      exec(action, args);
      if(action.equals("alloc")){
        allocated=true;
        checkForAvaliableActions();
        return;
      }
    }
  }
}
```

- safeWarning: per supportare il meccanismo di notifica da parte di nodi instabili
- log: attraverso la quale ogni nodo può stampare dati riguardanti la sua logica di business o le capability.

Oltre alle azioni predefinite è necessario implementare un metodo omonimo per ciascuna azione dichiarata nella descrizione dell'applicazione. La signature del metodo deve seguire questa convenzione:

- se sono presenti precondizioni, esse rappresentano, in ordine, i primi parametri del metodo. In particolare:
  - le precondizioni `any` sono rappresentate da un oggetto *HashMap<String, Object>*
  - le precondizioni `all` sono rappresentate da un oggetto *ArrayList<HashMap<String, Object>>*
- gli argomenti, se presenti, sono inseriti anche'essi nell'ordine di dichiarazione seguendo la seguente metodologia:
  - tipo Number: *Java Number*
  - tipo Boolean: *Java Boolean*
  - tipo String: *Java String*

Listing 5.5: La convenzione per gli argomenti delle action: un esempio

```
---App Description---

[...]
tier TIER1:
  [...]
  actions:
      foo(Number, String):
        preconditions:
          any TIER2
              where x==1
          all TIER3
[...]

---Plan---

//NODE25 ∈ TIER1
[...]
NODE25 foo(25, "test")
[...]

---EWXActuatorTIER1.class---

public void foo(HashMap<String, Object> tier2node, ArrayList<HashMap<
    String, Object>> tier3nodes, Number arg2, String arg3)
{
[...]
}
```

– tipo nodo: *HashMap<String, Object>*

A tal proposito in Listato 5.5 si riporta un esempio.
Il CapabilityManager si occupa di allocare, restituire e impostare le capability dichiarate nel tier e inoltre di verificare le espressioni condizionali su queste. Per la valutazione è stato usato l'engine Javascript. Questo oggetto è reperibile sia dal Dispatcher sia dall'Actuator grazie all'uso del patter Singleton.
È riportato in Figura 5.6 l'UML dell'agente.

5.5 Figura riassuntiva

Si mostra in Figura 5.7 un diagramma riassuntivo che mostra tutti i componenti del progetto.

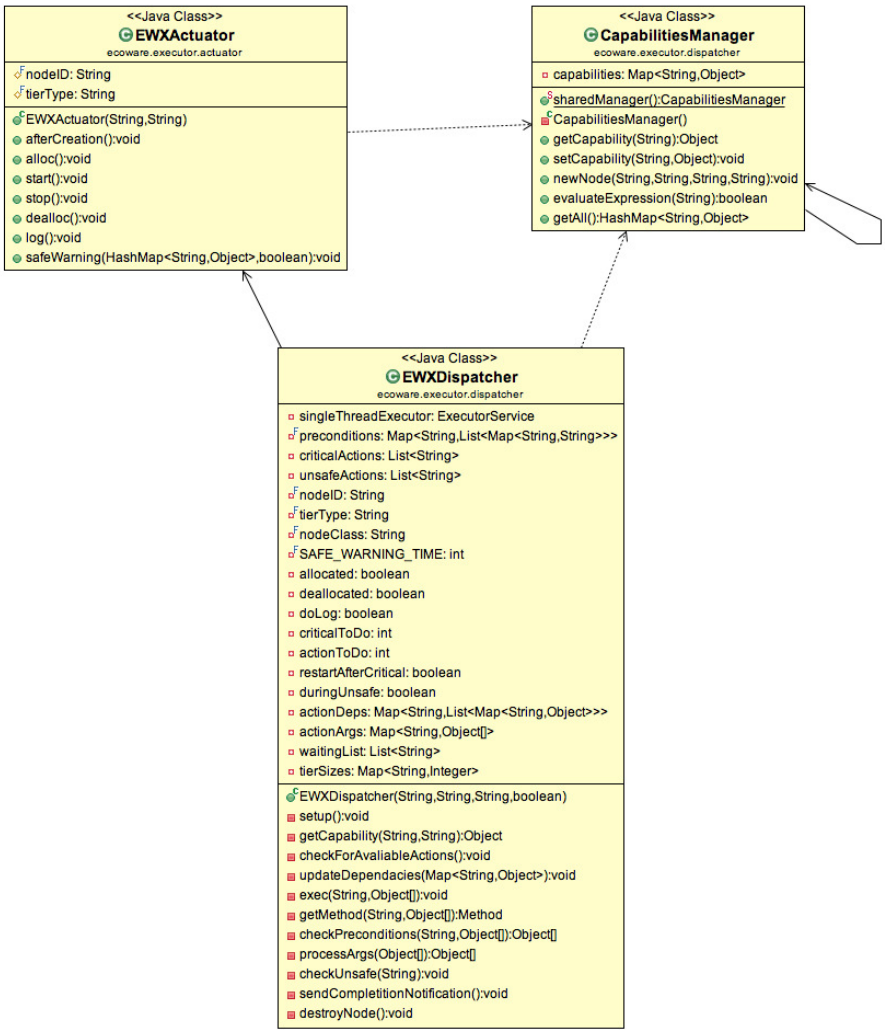


Figura 5.6: L'UML dell'agente distribuito

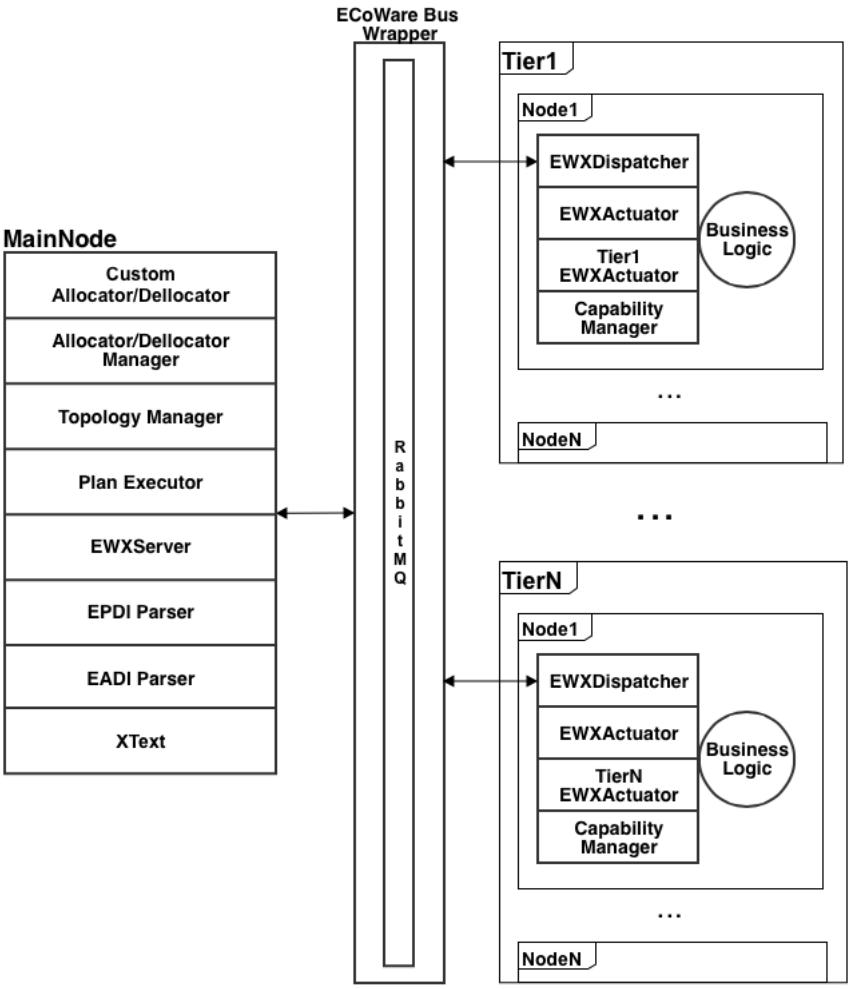


Figura 5.7: Uno schema riassuntivo dell'implementazione

Capitolo 6

Realizzazioni sperimentali e valutazione

> "Credo nelle idee che diventano azioni"
>
> Ezra Pound

In questo capitolo si illustra la valutazione del sistema attraverso RUBiS, un'applicazione di e-commerce simile ad Ebay utilizzata come benchmark in letteratura. Si mostrerà come è stato installato il framework ECoWare e che risultati sono stati ottenuti. Il lavoro svolto riguarda tutti i blocchi dell'architettura MAPE e verranno quindi riportati i contributi più rilevanti con un approfondimento per l'esecutore. Le valutazioni sperimentali riguardano in un primo approccio l'architettura completa ma con un esecutore rudimentale creato ad hoc, successivamente verrano riportate alcune misurazioni fatte sull'Executor descritto in questa tesi.

6.1 Il caso di studio

Per valutare il lavoro si è utilizzata una applicazione benchmark conosciuta con il nome di RUBiS [2]. RUBiS implementa alcune funzionalità base di un sito di aste online: navigazione, vendita, offerta. È stata sviluppata utilizzando tecnologia standard come HTML, Java Servlet e SQL. RUBiS è un'applicazione a tre tier: il web tier utilizza un server Apache, il business tier usa JBoss Application Server e il data tier è supportato da MySQL. Per questo lavoro si è deciso di distinguere la gestione del contenuto statico (home page, immagini, etc.) da quello dinamico (i prodotti disponibili in una regione, la cronologia delle offerte di un utente, etc.).

6.1.1 Il setup iniziale: private cloud

Per il deploy di RUBiS si è utilizzato un parco macchine composto da:

- un server dedicato al loadbalancer a fronte dell'applicazione, riconosce la tipologia del contenuto e la smista verso Apache o JBoss;
- tre server per il tier JBoss che si occupa del contenuto dinamico;
- un server per Apache che gestisce il contenuto statico;
- un server per il data tier;
- un server in cui è installato RabbitMQ e il framework ECoWare.

Le sette macchine utilizzate possiedono un processore Intel Xeon da 2.66GHz composto da due core ciascuno con 6MB di cache L2 e 32GB di memoria. Ad eccezione della macchina dedicata al loadbalancer che utilizza Microsoft Windows Server 2008 R2 le altre usano Ubuntu 12.04 LTS.

6.2 ECoWare on RUBiS

Nonostante ECoWare possa gestire più tipi di SLAs diversi (throughputs, costi, etc.) per semplificare la discussione ci si è concentrati sulla gestione delle risorse a runtime facendo riferimento ad una SLA del tipo il "$X°$ percercentile del tempo di risposta calcolato su un determinato periodo di tempo". Negli esperimenti calcoleremo il percentile su una finestra di due minuti e mezzo.

6.2.1 Monitoraggio ed Analisi

Come già espresso in Sezione 3.2.1 e 3.2.2 la componente di monitoraggio utilizza KPI Processor e Aggregator che calcolano, aggregano e producono nuovi eventi raccogliendo i dati inviati sul bus dai sensori e sonde innestate in ciascun layer di ogni macchina; la componente di analisi confronta quanto monitorato con le soglie preimpostate e pubblica sul bus un resoconto sulle violazioni eventualmente occorse.
Per gli esperimenti che verranno mostrati nel seguito sono state create diverse sonde. È stato sviluppato un Servlet Probing Filter con precisione al nanosecondo che calcola il *service time* e il *response time* (service time + attesa) di ciascuna servlet; ogni qualvolta una richiesta viene eseguita viene generato un evento contente il valore misurato. Al fine di misurare il layer infrastrutturale è stato creato e installato su tutte le macchine un sensore

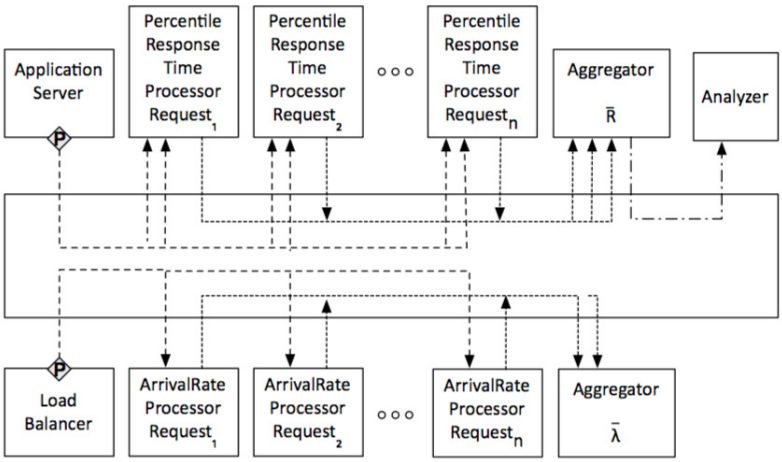


Figura 6.1: Monitoraggio ed Analisi in RUBiS

basato su Ganglia [1] che ogni minuto invia un evento con dati riguardanti CPU, memoria, networking e I/O. Per il tuning del Performance Model, che verrà spiegato successivamente, è stata creata una sonda per la misura dell'*arrival rate* nel loadbalancer, e un'altra nel tier JBoss per la misura del tempo speso nelle interazioni con il database. Inoltre si è anche voluto identificare il contributo di ciascuna invocazione di servlet su CPU, cache e RAM. Linux offre la raccolta di informazioni per thread attraverso l'utility `perf` ma non è stato possibile usarla a causa dell'overhead che crea rispetto alla granutilità scelta. Siccome si è voluto misurare ogni chiamata servlet e le servlet sono eseguite in thread abbiamo esteso il kernel Linux[1] affinché renda disponibili i dati con granularità thread direttamente all'interno dei file `sched` del sistema operativo. In Figura 6.1 si mostra come ECoWare produce $\overline{R}$ e $\overline{\lambda}$. Il primo è il vettore contente l'X° percentile dei tempi di risposta per servlet, il secondo è il vettore dell'arrival rate per servlet. Non aggreghiamo i questi valori ma li diviano per servlet poiché, come espresso in Sezione 3.2.3, il nostro approccio è mix-aware ovvero ECoWare è a conoscenza del tipo di richieste ricevute e dell'impatto che hanno sul sistema (CPU-intensive, memory-intensive, etc.). Dalla figura si notano anche le componenti che contribuiscono al monitoraggio: sul loadbalancer e sugli application server sono installate le sonde (probe) menzionate precedentemente. Due serie di processori (uno per servlet) calcolano il percentile del

[1]Il kernel modificato per eseguire i nostri esperimenti è disponibile all'indirizzo https://sosa.ucsd.edu/confluence/x/tAD0.

Algorithm 3 computeNewConfiguration($\overline{\lambda}$, R_{SLA}, $SysConf_{Current}$)

if $freeServerPool = \emptyset$ **then**
  return $SysConf_{Current}$
else
  select server $S \in freeServerPool$
  $SysConf_{New} \leftarrow SysConf_{Current} \cup S$
  $\overline{R_{New}} \leftarrow AnalyzePerf(\overline{\lambda}, SysConf_{New})$
  if $\overline{R_{New}}$ $satisfies$ R_{SLA} **then**
    return $SysConf_{New}$
  else
    return $computeNewConfiguration(\overline{\lambda}, R_{SLA}, SysConf_{New})$
  end if
end if

response time e l'arrival rate e gli aggregatori generano i vettori $\overline{R}$ e $\overline{\lambda}$. In linea con l'SLA i Processor calcolano i dati su una finestra di due minuti e mezzo e sempre ogni due minuti mezzo generano un evento sul bus.

L'Analyzer produce un evento contente i dati raccolti e le violazione se un valore in $\overline{R}$ è superiore a R_{SLA}.

6.2.2 Planner e Perfomance Model

Tratteremo questa componente in maniera sintetica in quanto è stata curata in gran parte da UCSD e quindi non rientra nelle competenze del lavoro svolto; è necessario però una panoramica per capire meglio i risultati della valutazione.

Come descritto in Sezione 3.2.3 il pianificatore ha il ruolo di creare, a partire dalle violazioni rilevate dall'Analyzer, una strategia affinché i parametri del sistema tendano sempre all'ottimo e all'efficenza. Si è integrato nel framework un pianificatore coeerente con il nostro approccio multi-layer e multi-tier e a questo proposito presenta diversi tipi di conoscenza: è mix-aware ovvero tiene conto della differenze tra diverse tipologie di richieste), è tier-aware ovvero possiede potenzialmente un modello per ogni tier ed è hardware-aware in quanto tiene conto delle caratteristiche della macchine disponibili. Questi aspetti non sono indipendenti tra loro in quanto, ad esempio, a fronte di un picco di richieste computazionalmente onerose il modello può scegliere, tra le macchine disponibili, quella più adeguata a supportare tale carico (ciò è facilmente estendibile ad un contesto di public cloud, con l'aggiunta, forse necessaria, dei costi all'interno del modello).

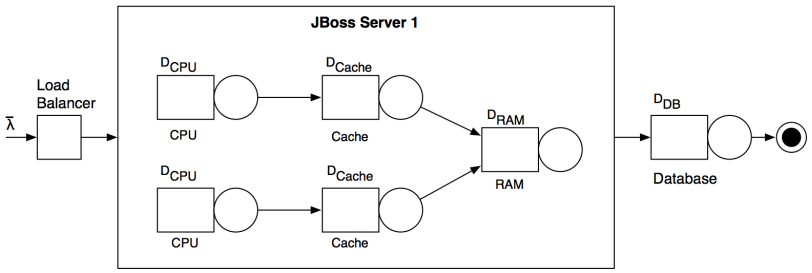


Figura 6.2: Performance Model

Il pianificatore utilizzato in questo esperimento fa uso dell'Algoritmo 3. L'algoritmo attraverso una procedura trial-and-error cerca la configurazione con il numero minimo di server che rispetti l'SLA. Per simulare i tempi di risposta con una nuova configurazione si utilizza la funzione *AnalyzePerf* che risolve un modello basato sulla teoria delle code che chiamiamo Performance Model. In Figura 6.2 si mostra una parte del Performance Model (per semplicità si mostra un unico nodo degli application server) e risulta evidente come emergano i principi precedentemente esposti: il modello conosce lo stack applicativo, il layer infrastrutturale (ad esempio si noti la modellizzazione dei due core di JBOSS1) e informazioni riguardanti il layer applicativo ovvero l'arrival rate di ciascuna servlet ($\overline{\lambda}$).

6.2.3 Executor

Come precedentemente anticipato per questo primo esperimento abbiamo fatto uso di un esecutore molte unicamente in grado di accendere e spegnere macchine fisiche. È proprio dopo questa esperienza che è stato deciso di costruire un esecutore più sofisticato che abbiamo presentato nei precedenti capitoli e che verrà anch'esso valutato.

6.3 Valutazione del sistema

Negli esperimenti che si mostreranno ci si è concentrati sull'efficientamento dell'allocazione delle risorse rispettando SLA; confrontandoci con lo stato dell'arte [22] si è riscontrato un miglioramento del 42.5%. Grazie alla flessibilità ed alla modularità di ECoWare attraverso piccole modifiche all'infrastruttura è stato possibile eseguire diversi test.
Durante la prima valutazione di RUBiS si è caricato il sistema per misurare i profili delle servlet. Questi dati sono successivamente stati utilizzati per calibrare il Performance Model. Poiché il tempo di esecuzione delle servlet è in gran parte determinato dal numero di query ci si è concentrati su due con

un numero fisso di richieste al database: `ViewUserInfo` e `ViewBidHistory`. È stato misurato il comportamento del sistema a diversi carichi e lo si è comparato con il modello proposto: la parcentuale d'errore per il response time si è verificata vicina all'8%. Posto che le servlet possedevano un tempo medio di risposta molto simile e spesso sotto il millisecondo è stato scelto, per rendere lo scenario più realistico, di aumentarne il numero di offerte nel database in modo tale da portare `ViewBidHistory` ad un tempo d'esecuzione medio di 88ms. Inoltre abbiamo aggiunto complessità a `ViewUserInfo` aggiungendo una procedura di validazione dei commenti lasciati dagli utenti (pratica tipica dei siti di social networking) portandola ad un tempo medio di 160ms.
Gli esperimenti che sono stati eseguiti riguardano in larga misura l'application tier, per questo motivo è stato assicurato che gli altri tier fossero dimensionati in modo da non essere mai il collo di bottiglia. L'approccio di questo lavoro è stato confrontato con due soluzione basate anch'esse sulla teoria delle code: una baseline e l'approccio mix-aware di Singh et al. [22][2]. Per questa valutazione sono stati utilizzati principi molto simili a quelli usati da [22]:

- è stata fissata l'SLA al *95esimo* percentile del response time con una soglia di 0.5 secondi;
- è stato usato un benchmark pubblico (RUBiS nel nostro caso);
- sono state selezionate alcune servlet;
- sono stati creati due differenti carichi di lavoro;
- è stato riconfigurato il loadbalancer in modo tale da distribuire ugualmente le richieste ai server disponibili.

Il primo carico di lavoro si riferisce al primo esperimento illustrato in Sezione 6.3.1, il secondo il quello di sezione 6.3.2.

6.3.1 Esperimento 1

Il primo esperimento è stato effettuato con un carico di lavoro a volume costante ma con mix di richieste variabile. Il carico di lavoro presenta una durata pari a 110 minuti e un volume costante di 15 richieste al secondo ($\lambda = 15 req/s$), con la modifica del mix tra le due richieste ogni 10 minuti. Il valore di λ è stato scelto in modo tale da non saturare mai completamente le

[2]Per il confronto sono state implementate le formule presentate nel paper e costruito vari Processor a supporto per fornire i dati necessari

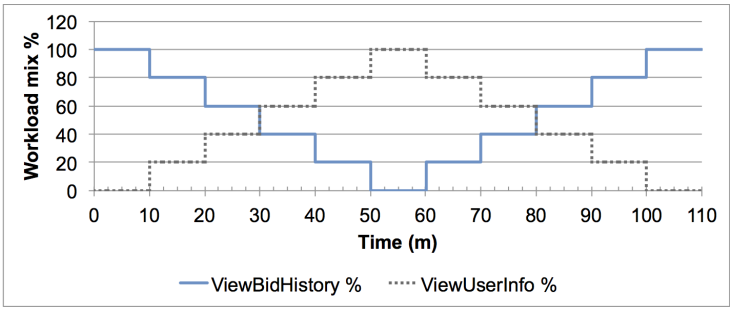


Figura 6.3: La variazione del mix delle richieste nel primo esperimento

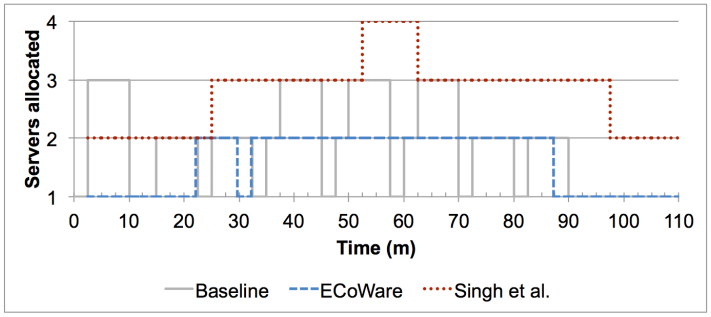


Figura 6.4: Il numero di server allocati dai tre approcci nel primo esperimento

risorse disponibili. In Figura 6.3 si mostra come il mix delle richieste varia durante il corso dell'esperimento. Inizialmente il 100% delle richieste è di tipo `ViewBidHistory` ($\lambda_{ViewBidHistory} = 15req/s$) e nei successivi 50 minuti, ogni 10 minuti, viene decrementato di 20% il numero di `ViewBidHistory` e viene aumentato sempre del 20% quello di `ViewUserInfo` cosa che causa un aumento graduale del response time in quanto `ViewUserInfo` è più pesante. ALl'inizio della seconda ora `ViewUserInfo` viene decrementata del 20% e viene aumentata `ViewBidHistory`.

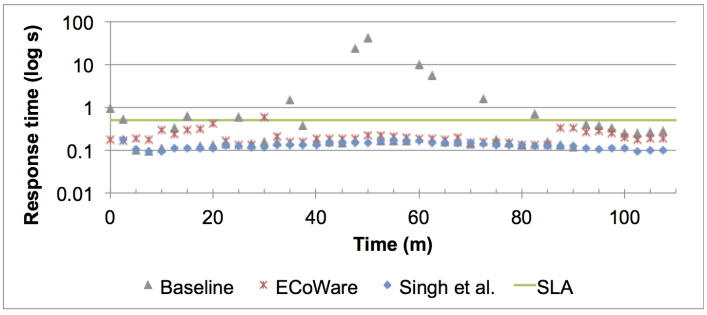


Figura 6.5: Il 95esimo percentile del response time nel primo esperimento

La logica di approvvigionamento delle risorse dei tre approcci viene invocata ogni 150 secondi. Figura 6.4 si mostra come i tre approcci rispondono al carico di lavoro mentre in Figura 6.5 si evidenzia la variazione del response time con riferimento alla soglia SLA di 0.5 secondi. Dalla Figura 6.5 si osserva come l'approccio baseline tenda a sotto-allocare risorse riportando di conseguenze diverse violazioni all'SLA. L'approccio di Singh et al. non incorre in violazioni di SLA ma questo deriva da un numero alto di server allocati ed è notare come in Figura 6.5 l'andamento del response time per questo approccio sia praticamente piatto, segno di sovra-allocazione. Tra $t = 52.5et = 62.5$ l'approccio richiede quattro application server; poiché ne erano solamente tre a disposizione l'esecutore non ha potuto soddisfare la richiesta ma ciò non intruduce alcuna imparzialità nell'esperimento in quanto l'argoritmo in questione non utilizza alcuna informazione su precedenti decisioni ma solamente sullo stato corrente dell'infrastruttura.
ECoWare, invece, presenta due minori violazioni che sono però prontamente risolte con un aumento delle risorse. Il doppio gradino di deallocamento-allocamento nell'intervallo da $t = 30$ a $t = 32.5$ è previsto. Infatti ECoWare correttamente dealloca al minuto $t = 30$ non potendo sapere che proprio in quel momento sarebbe cambiamo il mix aumentando il tempo di risposta cosa risolta alla chiamata successiva aggiungendo una macchina. ECoWare inoltre mostra una corretta simmetria tra l'andamento del numero delle risorse allocate e la variazione nel mix.
Per definire quantitativamente le differenze tra gli approcci è stato moltiplicato il numero di server per il tempo che sono stati allocati:

- baseline: 200 $server * minuti$ con un alto numero di violazioni (SLA non rispettata);
- approccio di Singh et al.: 300 $server * minuti$ con nessuna violazione;
- ECoWare: 172.5 $server * minuti$ con solo due violazioni trascurabili in un approccio reattivo non sovra-allocato.

ECoWare utilizza quindi 13.75% $server*minuti$ in meno rispetto al baseline e 42.5% in meno rispetto a [22].

6.3.2 Esperimento 2

Al fine di valutare ECoWare in una situazione più realistica in cui il volume di richieste cambia nel tempo, abbiamo randomizzato sia l'entità del volume sia l'intervallo temporale ogni quanto questo cambia. In Figura 6.6 si mostra come cambia in questo esperimento il mix delle richieste. Si noti che in

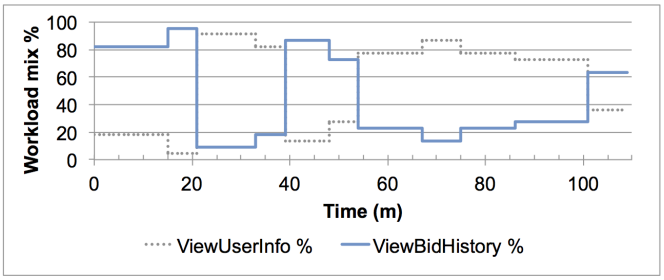


Figura 6.6: La variazione del mix delle richieste nel secondo esperimento

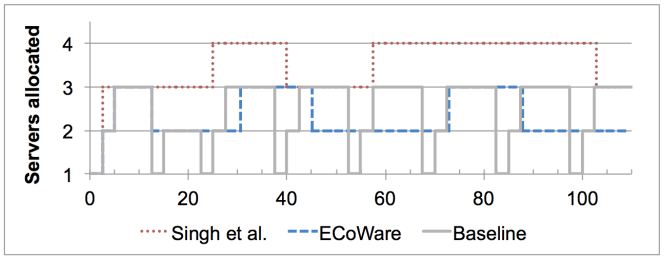


Figura 6.7: Il numero di server allocati dai tre approcci nel secondo esperimento

questo caso il 100% del volume cambia ad ogni step (precedentemente era fissato a $\lambda = 15req/s$); è stato posto un limite superiore di $\lambda = 22.5req/s$ affinché il carico di lavoro sia gestibile dal nostro parco macchine. Come è ovvio il carico randomizzato è stato identico in tutte i tre approcci.

Come nel precedente esperimento l'approccio baseline presenta forte instabilità. In Figura 6.7 si mostra come gli approcci gestiscono le risorse. Il baseline alloca due o più server ogni volta che il sistema si trova in uno stato di transitorio e, non appena questa fase è terminata, riduce il numero di server incorrendo in multiple violazioni del SLA. L'approccio baseline quindi conferma la tendenza a sottoallocare risorse. La soluzione [22] invece, oltre alle due iniziali violazione del SLA inevitabili in quanto il carico di lavoro inizia repentinamente con un alto numero di richieste, si conferma corretta non incorrendo in altri superementi della soglia. In quanto a numero di server allocati l'approccio di Singh et al. si conferma però troppo generoso, richiedendo fino a quattro server per lunghi periodi di tempo e, nonostante solo tre dei quattro server richiesti vengono realmente allocati, il response time si mantiene molto al di sotto del SLA.

ECoWare incorre nelle due fisiologiche violazioni iniziali come gli altri approcci e non incorre in altre per tutto il corso del workload. ECoWare adatta correttamente l'allocazione delle risorse ai cambiamenti di carico richiedendo fino a tre server in tre diversi momenti. In Figura 6.8 si mostra l'andamento

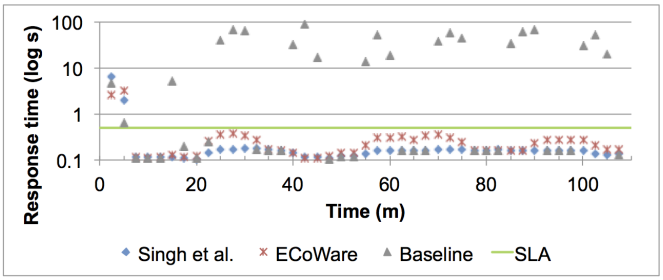


Figura 6.8: Il 95esimo percentile del response time nel secondo esperimento

del response time nei tre approcci; si vede come ECoWare nonostante le fluttuazioni riesce a mantenersi sempre al di sotto della soglia. Anche per questo esperimento è stato calcolato quantitativamente il risultato dei tre approcci:

- baseline: 302.5 $server * minuti$ con un alto numero di violazioni (SLA non rispettata);
- approccio di Singh et al.: 385 $server * minuti$ con nessuna violazione fatta eccezione per le due iniziali;
- ECoWare: 255 $server * minuti$ presentando anch'esso solamente le due violazioni inevitabili.

ECoWare, presentando un response-time più fluttuante, mostra come i server siano molto più utilizzati rispetto all'approccio [22] senza violare significativamente l'SLA. Infine ECoWare utilizza 16% $server * minuti$ in meno rispetto al baseline con un numero inferiore di violazioni e 33% in meno rispetto all'approccio di Singh et al. con lo stesso numero di violazioni.

6.4 Valutazione del nuovo esecutore

Dopo aver valutato il sistema complessivo è emersa l'esigenza di concentrarsi sulla componente d'esecuzione che fino a quel momento era stata costruita ad hoc per la realizzazione degli esperimenti. Nei capitoli precedenti si è descritto il modello e l'implementazione di un nuovo esecutore multi-livello che si integri e rispetti i principi dell'infrastruttura. In questa sezione si espone l'application stack di RUBiS descritto con il linguaggio EADl, come è stato cambiato il parco server e si mostreranno alcuni dettagli implementativi alla base della valutazione effettuata. Infine verranno riportati i risultati degli esperimenti e alcune osservazioni finali.

6.4.1 Lo stack applicativo di RUBiS

Come abbiamo visto per gli esperimenti utilizziamo una versione di RUBiS a quattro tier composta da: un loadbalancer a fronte dell'applicazione, un server Apache per il contenuto statico, diversi JBoss per il tier relativo alla logica di business e un database che utilizza MySQL. Questo stack è stato descritto con EADl in Listato 6.1.
Il tier in testa allo stack è quello del loadbalancer, chiamato `FRONT_LB`. Possiede una singola instanza e ridefinisce l'azione `alloc` affinché vengano passati, attraverso precondizioni, tutti gli application server ed il server Apache accesi: in questo modo può essere correttamente inizializzato per lo smistamento delle richieste. Si noti che essendo unico il server Apache la precondizione `any` su questo tier poteva essere espressa anche come `all`; la differenza tra le due possibilità sta nel fatto che se una condizione di tipo `any` non viene soddisfatta genera un messaggio d'errore e fallisce l'esecuzione del piano. Il tier presenta inoltre altre due azioni: `updateJBossList` e `setAlgorithm`. La prima serve per l'aggiornamento della lista degli application server in quanto l'Actuator del loadbancer, come vedremo, tiene traccia, attraverso una lista, dei JBoss attivi. La seconda permette il cambiamento a runtime dell'algoritmo di smistamento delle richieste il cui identificativo viene passato come stringa. Il tier `APACHE` ridefinisce `alloc` per dichiarare la modifica della capability aggiuntiva `port`, inoltre presenta una semplice azione di gestione `clearLogFile`. Il tier `JBOSS` può contenere al più 300 istanze, scelta fatta per contenere i costi. L'azione `alloc` necessita le informazioni di un database per la configurazione del connettore a MySQL. Il tier presenta anche un'azione critica che modifica un parametro dinamico del servizio: il numero di thread disponibile per l'esecuzione delle richieste (servlet). Il fatto che questo tier sia stato dichiarato successivamente al tier `APACHE` non è rilevante in quanto non vi è dipendenza tra i due e logicamente si pongono allo stesso livello: tra il loadbalancer e il db. Infine il tier `DB` espone tre capability personalizzate: `username`, `password` e `port`, necessarie per il funzionamento delle instanze del tier JBOSS.

6.4.2 Il nuovo setup: hybrid cloud

Al fine di rendere più realistica l'architettura è stata aggiunta la possibilità per gli application server di poter scalare su Amazon AWS. Mantenendo le sette macchine fisiche costruiamo una vera e propria hybrid cloud ovvero una server farm privata con la possibilità, in caso di necessità, di poter scalare su un provider IaaS pubblico. Su Amazon Web Service abbiamo fatto uso di macchine relativamente poco potenti per ridurre i costi; in particolare

Listing 6.1: L'application stack di RUBiS

```
app RUBiS:
  tier FRONT_LB:
    minq: 1
    maxq: 1
    actions:
      alloc():
        preconditions:
          all JBOSS
            where started==true
          any APACHE
            where started==true
      updateJBossList():
        preconditions:
          all JBOSS
            where started==true
      setAlgorithm(String)
  tier APACHE:
    capabilities: port
    minq: 1
    maxq: 1
    actions:
      alloc():
        modifies: port
      clearLogFile()
  tier JBOSS:
    minq: 1
    maxq: 300
    actions:
      alloc():
        preconditions:
            any DB
      setMaxThreads(Number):
        critical
      clearLogFile()
  tier DB:
    capabilities: port, username, password
    minq: 1
    maxq: 1
    actions:
      alloc():
        modifies: port, username, password
      setMaxConnection(Number)
      setMemLock(Boolean):
        critical
```

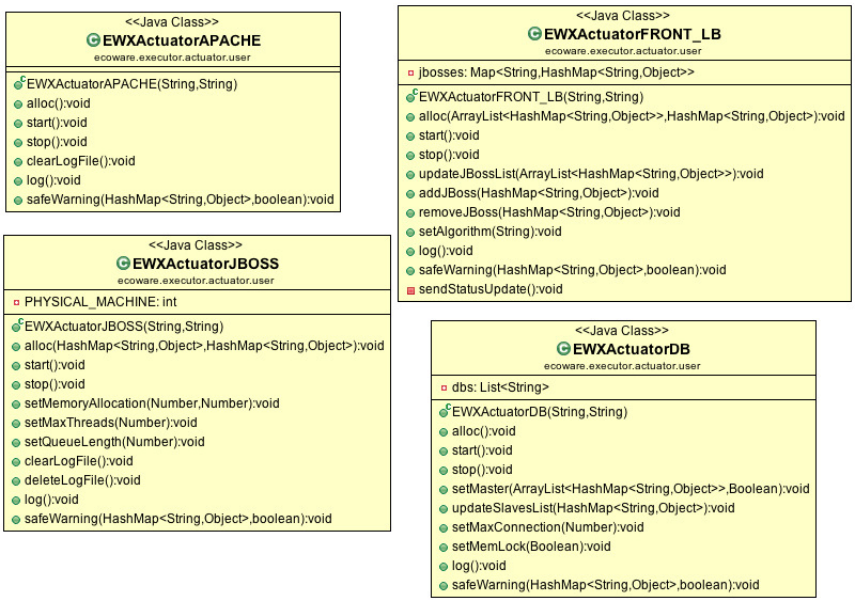


Figura 6.9: UML degli Actuator usati nell'esempio

si è fatto uso di macchine *General Purpose - Previous Generation* di tipo *m1.small* con una singola CPU virtuale, 1.7GB di memoria e 160GB di disco dal costo di 0.060$/ora. Sulle macchine fisiche sono stati installati i vari componenti (applicativi jar) per il funzionamento dell'esecutore: sulla macchina dedicata ad ECoWare e RabbitMQ è stato fatto il deploy del Main Node, sul loadbalancer è stato attivato manualmente l'agente con id `FRONT_LB1`, così come sui tre appliation server `JBOSS1`, `JBOSS2` e `JBOSS3`, sul server Apache `APACHE1` e sul database con id `DB1`. Poiché il Main Node, ogni qual volta si voglia eseguire un piano, lancia un messaggio sul bus e raccoglie i nodi attivi, l'accensione manuale sulle macchine fisiche non comporta alcun problema.

6.4.3 L'implementazione degli Actuator

La costruzione degli attuatori consiste nella definizione di quattro classi che forniscono l'implementazione delle action definite nei relativi tier. In Figura 6.9. Gli attuatori, a cui vengono passati i nodi necessari, modificano le capability del nodo e gestiscono il servizio installato sulla macchina.
In Listato 6.2 si mostrano i metodi `start()` e `stop()` dell'Actuator installato nei nodi del tier `JBOSS`. L'implementazione fa uso di script per l'avvio e lo spegnimento del servizio, e al termine delle operazioni si modifica la capability `started`. In Listato 6.3 si mostra invece l'implementazione del

metodo `updateJBossList` nell'attuatore del tier `FRONT_LB` in cui si aggiorna la propria struttura dati `jbosses` e la si invia, sotto forma di lista, sul bus. Il servizio potrà in questo modo ricevere l'evento e aggiornare la sua business logic.

Listing 6.2: Start e Stop action di JBoss

```
private int PHYSICAL_MACHINE=3;

public void start(){
  super.start();
  String[] command = {"./jbossrun.sh", "rubis_"+nodeID.toLowerCase(),
      "</dev/null &>/dev/null &"};
  ProcessBuilder pb = new ProcessBuilder(command);
  int i=Integer.parseInt(nodeID.substring(5));
  String path="opt";
  if(i<=PHYSICAL_MACHINE)
    path="root";
  pb.directory(new File("/"+path+"/rubis/rubis-cvs-2008-02-25/
      Servlets_Hibernate/"));
  Process p=pb.start();
  Thread.sleep(10000);
  capabilityManager.sharedManager().set("started", true);
}

public void stop(){
  super.stop();
  String[] command = {"/bin/bash", "-c", "ps -ef | grep -i jboss |
      grep -i native | grep -v grep | awk '{print $2}' | xargs kill"};
  Runtime.getRuntime().exec(command);
  Thread.sleep(10000);
  capabilityManager.sharedManager().set("started", false);
}
```

Listing 6.3: Il metodo 'updateJBossList' nell'attuatore del loadbalancer

```
public void updateJBossList(ArrayList<HashMap<String, Object>> nodes){
  Commons.p("Executing action updateJBossList", nodeID);
  jbosses.clear();
  for(HashMap<String, Object> node : nodes)
      jbosses.put((String) node.get("id"), node);

  ECoWareRequestSender request = new ECoWareRequestSender(nodeID, "
      LOAD_BALANCER_UPDATE", ECoWareEventType.LOAD_BALANCER_UPDATE);
  request.put("value", Arrays.asList(jbosses.values().toArray()));
  request.sendRequest();
}
```

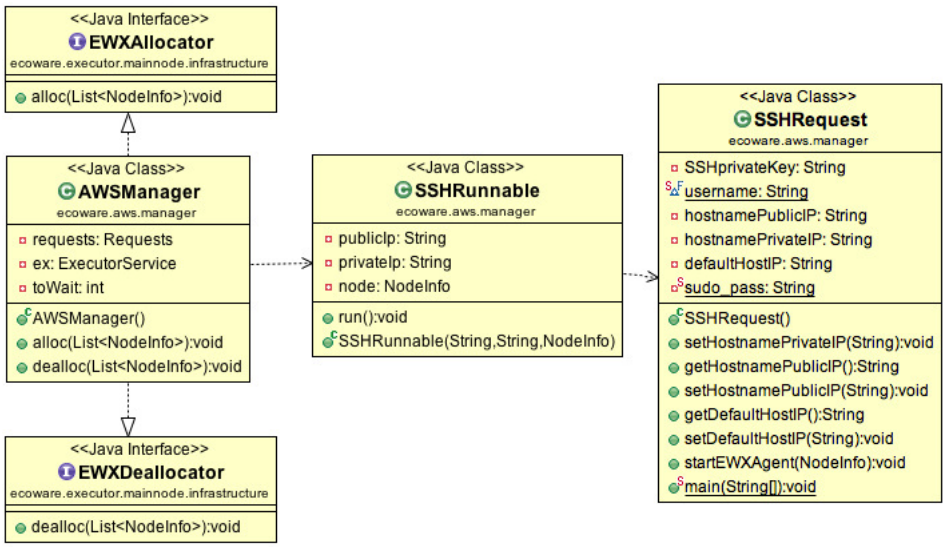


Figura 6.10: UML dell'interfaccia ad Amazon AWS

6.4.4 L'implementazione dell'allocatore e deallocatore

Come è stato anticipato il nuovo setup comprende la possibilità per gli application server di scalare su Amazon Web Service. Per il deploy del progetto abbiamo creato una AMI (Amazon Machine Image) contente RUBiS e l'eseguibile dell'agente. Come esplicitato in Sezione 5.3 per gestire l'allocazione e deallocazione su provider IaaS è necessario implementare un allocatore e un deallocatore conformi ad una interfaccia standard; si riporta in Figura 6.10 l'UML di questa componente. La classe `AWSManager` implementa sia l'interfaccia `EWXAllocator` che `EWXDeallocator`. Nell'attuale implementazione l'allocatore deve garantire al Main Node che dopo l'esecuzione del metodo `alloc` i nodi siano attivi con l'agente installato. L'AWSManager, che utilizza un apposito SDK, effettua una richiesta verso Amazon di allocazione di un certo numero di macchine e aspetta finché non riesce ad ottenere l'indirizzo IP di tutte. Quando tutte le macchine sono pronte invia in parallelo una richiesta SSH per l'accensione dell'agente; ciò avviene attraverso un `ExecutorService` e una classe wrapper `SSHRunnable` che permette l'uso parallelizzato della classe che effettivamente esegue la richiesta: `SSHRequest`. Quest'ultima classe attiva l'agente che invia sul bus un evento di compiuta accensione; questi eventi vengono raccolti dall'AWSManager che attende fintanto che non riceve il numero atteso di messaggi.

6.4.5 La valutazione dell'esecutore

La valutazione di un esecutore presenta diverse problematiche. Non è facile infatti trovare delle misure adeguate che possano rappresentare completamente il valore di un lavoro di questo tipo. Fattori come la facilità d'uso o l'espressività del linguaggio utilizzato non possono essere misurati scientificamente con semplici esperimenti di laboratorio. Si è quindi misurato l'esecutore in tutte le fasi e componenti ove possibile. Per prima cosa è stato investigato l'impatto sulla CPU e sulla memoria del nodo centrale e dell'agente attraverso l'uso delle utility `top` e `ps`. È stato rilevato un impatto pressoché nullo sulla CPU sia in attesa che durante l'esecuzione di un piano; il Main Node occupa costantemente circa il 4% della memoria mentre, anche in questo caso, il peso dell'agente è trascurabile. Per quanto riguarda il bus, la grandezza dei messaggi va dalle centinaia di byte alle decine di kilobyte nel caso di messaggi contenenti tutte le capability. Il numero dei messaggi per piano è pari al risultato delle seguenti equazioni:

$$topology\ search = 1 + \#nodes$$

$$IAAS\ actions = 3 * \#created + \#destroyed$$

$$coreogrphy = \#preconditions + \sum_{i=0}^{\#preconditions} \#tier_i + prepare + \#involved + start$$

$$execution = 2 * \#node\_args + \#preconditions + \sum_{i=0}^{\#preconditions} \#tier_i + 2 * \#unsafe + \#involved$$

$$\#message = topology\ search + IAAS\ actions + coreogrphy + execution$$

La prima equazione indica il numero di messaggi causati da una ricerca dei nodi attivi: un messagio per la richiesta broadcast e un numero di risposte pari al numero dei nodi. La seconda equazione riguarda la fase delle azioni infrastrutturali: tre messaggi per ciascun nodo creato (uno inviato dall'agente appena installato e due per la richiesta e la risposta dell'evento 'tier info'). La terza equazione calcola il numero di messaggi nella preparazione al piano. Il nodo centrale, controllando le dipendenze, esegue richieste sui nodi per verificare se esistono precondizioni soddisfatte a priori, ciò comporta un

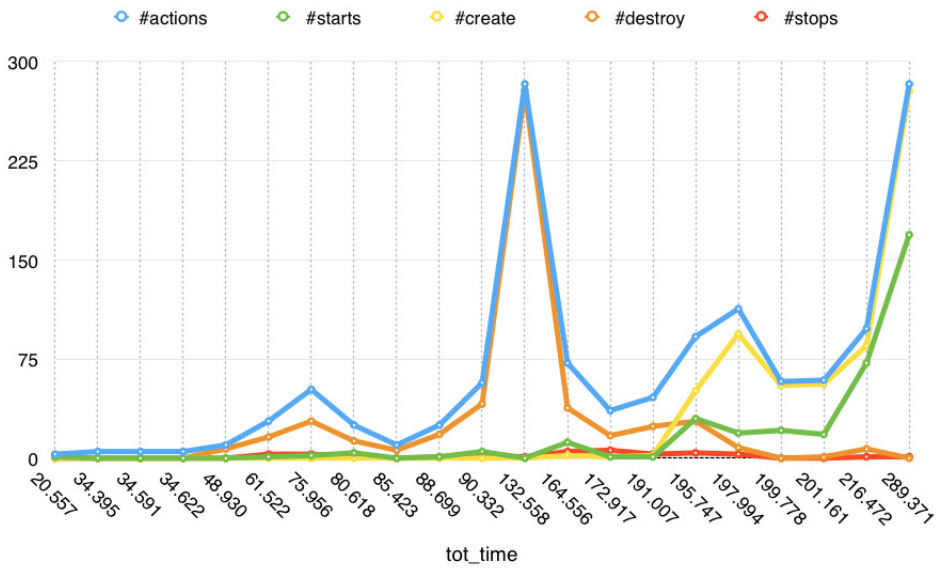


Figura 6.11: La variazione del numero di azioni rispetto al tempo d'esecuzione

numero di messaggi pari alle precondizioni più, per ciascuna, un numero di risposte pari ai nodi presenti nel relativo tier. Inoltre viene inviata una richiesta broadcast di preparazione al piano a cui seguono un numero di risposte pari ai nodi coinvolti nel piano e una richiesta che sancisce l'avvio dell'esecuzione. La quarta equazione si riferisce all'esecuzione distribuita sui nodi: per ciascun argomento di tipo nodo vengono inviati due messaggi (richiesta e risposta), per le precondizioni vale quanto detto per la terza equazione ma il numero di queste può essere minore se qualcuna è stata soddisfatta in fase di coreografia; a ciò si aggiungono due messaggi per ciascun nodo che contiene almeno un'azione `unsafe` e un numero di messaggi di completamento pari al numero di nodi coinvolti nel piano. Infine la quinta equazione mostra la somma finale.

In Figura 6.14 si riporta la tabella relativa alle misurazioni effettuate sull'esecutore durante l'esecuzione di diversi piani. La prima colonna rappresenta il numero di azioni nel piano, la seconda il numero di creazioni di nodi (`alloc`+`new`), la terza il numero di `remove`, la quarta e la quinta rispettivamente il numero di accensioni e spegnimenti (compresi quelli causati da azioni `critical`) che si riportano separatamente in quanto sono le azioni più onerose (in particolare la `start`), la sesta e la settima il numero di nodi presenti nella topologia all'inizio e al termine dell'esecuzione, la colonna otto è il tempo totale dell'esecuzione e le successive colonne rappresentano i tempi di ciascuna fase: la ricerca dei nodi nella topologia, il controllo sui

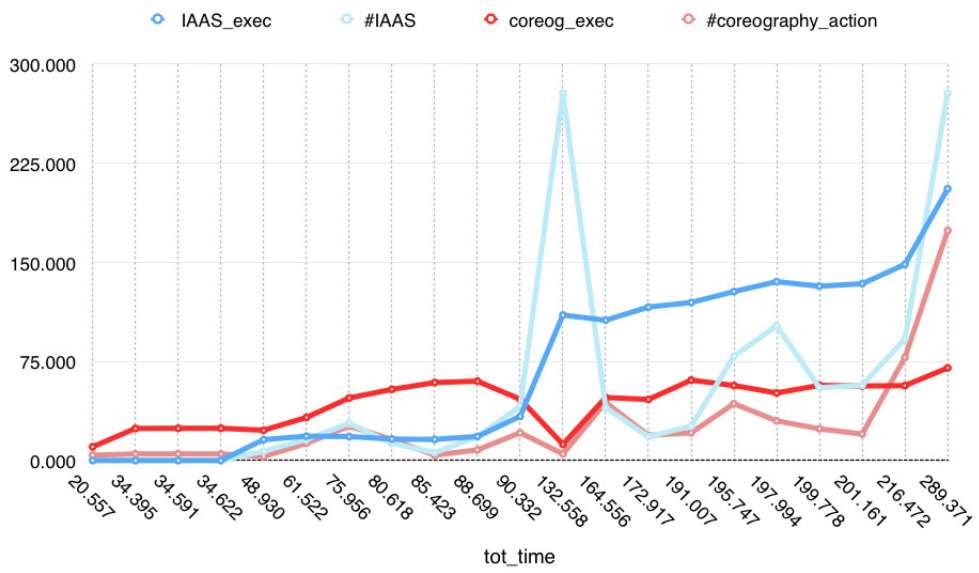


Figura 6.12: Confronto delle due fasi principali di esecuzione

limiti del numero delle istanze, la preparazione alle azioni infrastrutturali, l'esecuzione delle azioni infrastrutturali, la creazione della coreografia e l'esecuzione distribuita. Si noti che per ogni `new` viene eseguita una `start` (il valore in tabelle include tutto). Dalla tabella abbiamo creato tre grafici che mettono in evidenza alcuni aspetti interessanti.

In Figura 6.11 mostriamo le dipendenza tra il numero di azioni, totali o divise per tipologia, e il tempo totale di esecuzione. Come si può notare non emerge una chiara dipendenza tra le varie misure in quanto i parametri da valutare non si limitano al numero delle azioni ma anche al numero di nodi all'interno della topologia, al numero di dipendenze all'interno del piano, di precondizioni da soddisfare e anche al tipo di azioni che vengono eseguite. Come si può notare il piano con il tempo d'esecuzione maggiore è quello in cui sono state allocate e accesse quasi trecento macchine in un volta: queste due azioni sono quelle che incidono di più sul tempo d'esecuzione. Il picco centrale si riferisce alla rimozione di quasi trecento macchine contemporaneamente, la deallocazione è meno dispendiosa dall'allocazione in quanto è meno vincolante dal punto di vista del sistema: per completare l'allocazione è necessario che la macchina faccia il boot, restituisca l'indirizzo ip, che venga acceso l'agente tramite richiesta ssh e che questo si configuri correttamente; al contrario per la deallocazione è necessario solo notificare l'agente affinché possa prepararsi alla rimozione e procedere con la richiesta al provider IaaS.

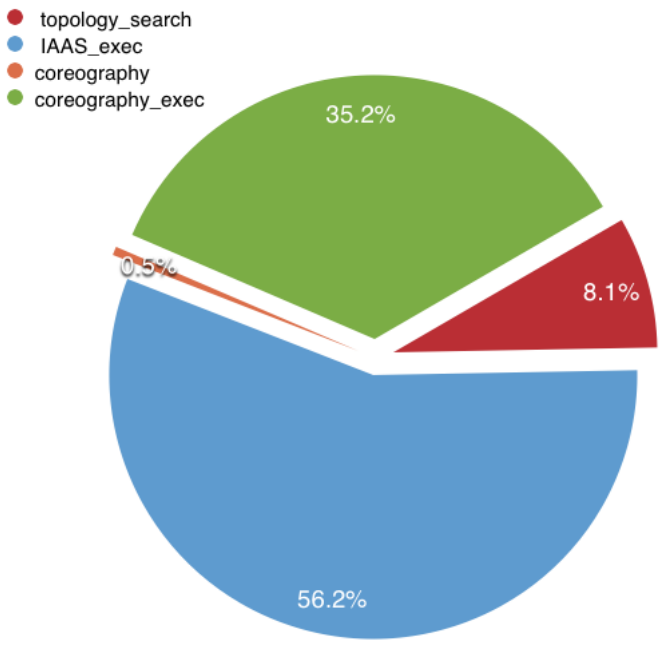


Figura 6.13: Overhead ed esecuzione

In Figura 6.12 si mostra l'andamento e il numero di azioni dell'esecuzione infrastrutturale e di piattaforma. Per quanto riguarda la fase IaaS si può vedere come l'aumento del numero di azioni non coincida necessariamente con un aumento del tempo d'esecuzione totale in quanto la deallocazione è meno onerosa dell'allocazione ed inoltre il tempo di risposta e di esecuzione lato provider può variare indistintamente. La fase di coreografia e di piattaforma invece mostra un andamento più costante nel tempo, all'aumentare del numero di azioni di questo tipo il tempo di esecuzione presenta qualche fluttuazione ma rimane nella grande maggioranza dei casi nell'intervallo tra i 50 e i 75 secondi. Inoltre come si può notare dagli ultimi due punti del grafico (ultime due righe di Figura 6.11) passando da 79 azioni all'interno della coreografia a 175 ovvero un incremento del 121% il tempo d'esecuzione di questa fase aumenta solamente del 25%.

In Figura 6.13 si illustra come le varie fasi, escludendo quelle trascurabili (pochi millisecondi), incidano sul tempo totale. La ricerca dei nodi attivi nella tipologia e la costruzione della topologia rappresentano l'overhead dell'esecutore che rimane al di sotto del 10%. Inoltre la ricerca nella topologia è costante in tutti i piani in quanto si lancia un messaggio sul bus e si aspettano per 10 secondi le risposte dai nodi, la creazione della coreografia aumenta all'aumentare delle dipendenze e soprattutto delle precondizioni

da valutare, ma risulta inferiore ai 3 secondi anche nel caso di centinaia di azioni con dipendenze. Più del 90% del tempo è dedicato alla vera e propria esecuzione con un peso maggiore delle azioni di infrastrutturali.
In conclusione l'esecutore non rappresenta un overhead rilevante: è praticamente ininfluente dal punto di vista computazionale e della memoria e sul tempo totale incide meno del 10%. Un piano complesso composto da quasi 300 allocazioni (pressoché irrealistico) simultanee con centinaia di dipendenze e sincronizzazioni da rispettare comporta una durata di circa cinque minuti di cui tre e mezzo dedicati alle comunicazioni, indipendenti dal sistema, con il provider IaaS. Inoltre si è mostrato come l'incidenza del tempo d'esecuzione della coreografia sia al più sottolineare rispetto al numero di azioni.

#actions	#create	#destroy	#starts	#stops	#nodes_i	#nodes_f	tot_time	topology_search	limit_check	IAAS_prep	IAAS_exec	coreography	coreog_exec
3	0	0	1	0	5	5	20.557	10.039	0.000	0.000	0.001	0.133	10.384
5	0	0	0	0	6	6	34.395	10.037	0.001	0.000	0.001	0.106	24.250
5	0	0	0	0	61	61	34.591	10.041	0.000	0.000	0.001	0.164	24.385
5	0	0	0	0	61	61	34.622	10.069	0.000	0.000	0.000	0.162	24.391
10	0	7	0	0	12	5	48.930	10.055	0.001	0.000	15.820	0.142	22.912
28	0	16	1	3	34	18	61.522	10.059	0.000	0.001	18.392	0.481	32.589
52	0	28	2	3	61	33	75.956	10.050	0.000	0.000	18.124	0.469	47.313
25	0	13	4	1	33	20	80.618	10.040	0.001	0.000	16.253	0.493	53.831
10	0	6	0	0	18	12	85.423	10.042	0.001	0.000	15.940	0.451	58.989
25	0	18	1	1	35	17	88.699	10.045	0.000	0.001	18.175	0.461	60.017
57	0	41	5	2	83	42	90.332	10.053	0.000	0.001	33.433	0.464	46.381
283	0	278	0	1	283	5	132.558	10.085	0.001	0.000	110.079	0.159	12.234
72	2	38	12	5	106	70	164.556	10.062	0.000	0.001	106.184	0.637	47.672
36	1	17	1	6	51	35	172.917	10.050	0.000	0.000	116.119	0.548	46.200
46	2	24	1	3	56	34	191.007	10.054	0.001	0.000	119.586	0.549	60.817
92	51	28	30	4	60	83	195.747	10.058	0.001	0.002	127.844	1.089	56.753
113	94	8	19	3	20	106	197.994	10.049	0.001	0.003	135.336	1.511	51.094
58	55	0	21	0	5	60	199.778	10.039	0.000	0.004	131.914	0.960	56.861
59	56	1	18	0	6	61	201.161	10.056	0.001	0.002	133.764	0.940	56.398
98	85	7	72	1	17	95	216.472	10.041	0.001	0.005	148.436	1.316	56.673
283	278	0	169	1	5	283	289.371	10.054	0.002	0.016	205.670	2.557	70.072

Figura 6.14: Rilevazioni durante l'esecuzione di diversi piani

Capitolo 7

Valutazioni critiche e direzioni future

Nei precedenti capitoli è stato presentato un approccio multi-livello alla gestione autonomica di applicazioni, focalizzato sulla gestione di attuatori nell'architettura MAPE distribuita nel cloud. Il lavoro, frutto di un anno di sforzi che mi hanno arricchito umanamente e scientificamente grazie anche alla colaborazione con UCSD che si è concretizzata in un viaggio di tre mesi, valorizza il framework ECoWare con la creazione di un innovativo esecutore multi-livello che permette il passaggio naturale da private cloud ad hybrid cloud. Quanto presentato non è però esente da difetti.

L'approccio multi-tier e multi-layer è sbilanciato sul tier degli application server: un passo importante potrebbe essere di affiancare a RUBiS un altro benchmark con uno stack applicativo diverso, magari più complesso, che permetta una valutazione più complessiva. Il Performance Model su cui ci basiamo è limitato all'allocazione e alla deallocazione di risorse: sarebbe interessante renderlo più intelligente e integrato con il nuovo esecutore e con la possibilità di cambiare anche parametri riguardanti il servizio e l'applicazione; inoltre la nostra idea iniziale comprendeva un approccio pro-attivo che superasse quello reattivo attigendo ad una conoscenza pregressa e a dati statistici rigurdanti il dominio dell'applicazione.

L'esecutore necessita di una maturazione che non è stata possibile durante questo periodo. Non è stata trattata nella tesi la scoperta e la risoluzione di malfunzionamenti in quanto è presente una semplice implementazione che non si basa però su un modello definito. Nonostante ciò sono presenti tutti gli strumenti per estendere il lavoro e verrà fatto in vista della pubblicazione di un paper tra Aprile e Maggio 2014. Nel paper sarà presente anche una

valutazione complessiva del framework in quanto, come già detto, il nuovo esecutore è stato testato in isolamento. Potrebbe essere utile, nonostante sia un dettaglio implementativo, rendere l'esecutore ancora più strutturato: un'idea potrebbe essere di generare automaticamente il codice degli attuatori a partire da un file EADl ed inoltre rappresentare ogni nodo non più con delle generiche mappe ma con delle classi aventi come proprietà le capability. La sicurezza del sistema non è stata considerata, tutte le capability vengono inviate in chiaro cosa che potrebbe creare diversi problemi (si pensi solo ai dati del database). A tal proposito alcuni valori da assegnare alle capability, prendendo spunto da Amazon Cloud Formation, potrebbe essere richiesti all'utente prima di effetture il primo deploy. Le prestazioni dell'esecutore potrebbero essere migliorate attraverso l'uso di notifiche automatiche che segnalano ai nodi interessati (topologicamente superiori) l'esecuzione di azioni e la conseguente modifica delle capability. In questo modo, nel nostro esempio, il Planner potrebbe evitare di invocare un'azione di aggiornamento del loadbalancer a fronte di modifiche di application server in quanto avverrebbe tutto automaticamente. A tal fine sarebbe necessaria un'estensione del modello. Infine per la gestione del deploy di nuove risorse potrebbe essere interessante un'integrazione con configuration tool come Chef e Puppet.

Bibliografia

[1] Ganglia – Ganglia Monitoring System – http://ganglia.sourceforge.net.

[2] RUBiS – Rice University Bidding System – http://rubis.ow2.org/.

[3] Software services and systems network — s-cube - european network of excellence.

[4] Luciano Baresi and Sam Guinea. Event-based multi-level service monitoring. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 83–90. IEEE, 2013.

[5] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[6] Aaron B Brown and David A Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2003.

[7] Clovis Chapman, Wolfgang Emmerich, Fermın Galán Marquez, Stuart Clayman, and Alex Galis. Elastic service definition in computational clouds. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 327–334. IEEE, 2010.

[8] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, 2012.

[9] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *Service-Oriented Computing*, pages 429–436. Springer, 2013.

[10] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Sybl: an extensible language for controlling elasticity in cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013*

13th IEEE/ACM International Symposium on, pages 112–119. IEEE, 2013.

[11] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Sybl+ mela: Specifying, monitoring, and controlling elasticity of cloud services. In *Service-Oriented Computing*, pages 679–682. Springer, 2013.

[12] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5), 2011.

[13] C. Frye. Self-healing systems. *Application Development Trends*, 2003.

[14] A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. *IBM System Journal 42*, 2003.

[15] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[16] Yan Liu and Ian Gorton. Implementing adaptive performance management in server applications. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 12. IEEE Computer Society, 2007.

[17] Patrick Martin, Andrew Brown, Wendy Powley, and Jose Luis Vazquez-Poletti. Autonomic management of elastic services in the cloud. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 135–140. IEEE, 2011.

[18] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Mela: Monitoring and analyzing elasticity of cloud services. In *International Conference on Cloud Computing Technology and Science, ser. CloudCom*, 2013.

[19] CELAR project. Automatic, multi-grained elasticity provisioning for the cloud. *http://www.celarcloud.eu/wp-content/uploads/2014/02/CELAR-Project-Presentation-Feb14.pdf*, 2014.

[20] Karyl Scott. Computer, heal thyself. *Information Week, April*, 1:1–1, 2002.

[21] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559–570. IEEE, 2011.

[22] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*, pages 21–30, 2010.

[23] Bill Snyder. Server Virtualization has Stalled, Despite the Hype. *InfoWorld*, 2010.

[24] Sebastiano Spicuglia, Mathias Björkqvist, Lydia Y. Chen, Giuseppe Serazzi, Walter Binder, and Evgenia Smirni. On Load Balancing: a Mix-aware Algorithm for Heterogeneous Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 71–76, 2013.

[25] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC)*, pages 217–228, 2005.

[26] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning Servers in the Application Tier for e-commerce Systems. *ACM Transactions on Internet Technology*, 7(1), 2007.

Elenco delle figure

1.1 La struttura di un'applicazione 12

2.1 L'architettura MAPE o MAPE-K 16
2.2 Stitch, un linguaggio per l'adattamento automatico 17
2.3 L'architettura MAPE-K del framework ASF 18
2.4 La multi-dimensionalità dell'elasticità 19
2.5 La visione dell'applicazione in SYBL 20
2.6 Mela, un framework per il monitoraggio e l'analisi multi-livello 21
2.7 Cloud Formation 22
2.8 La creazione di un application server in OpsWork 23
2.9 Un estratto di un file Cloud Formation 24
2.10 Amazon Web Service 25
2.11 DevOps, il conflitto tra operazioni e sviluppo 27
2.12 L'architettura parzialmente centralizzata di Chef 28
2.13 Cloudify e Chef 29
2.14 Panoramica dei maggiori provider IaaS o PaaS 30

3.1 La suddivisione in layer e tier delle applicazioni 32
3.2 ECoWare all'interno di un'applicazione web a sei tier 33
3.3 Un blocco di codice mlCCL 35
3.4 La dashboard della componente di monitoragigo 36
3.5 PaaS vs IaaS 38
3.6 Orchestrazione vs Coreografia 39
3.7 L'inserimento di un nuovo application server in una semplice web application 42

4.1 Un'applicazione a tre tier. Il flusso di informazioni si sposta dal basso verso l'alto. 44
4.2 Anatomia di un'applicazione 45
4.3 Il meta-modello 49
4.4 Un esempio di esecuzione di piano 55

4.5 L'ordine di esecuzione 56
4.6 Schema dell'architettura distribuita 58
4.7 La creazione di un nuovo nodo 63
4.8 Il nodo distribuito 64
4.9 L'unità d'esecuzione 65

5.1 Gestione dei DSL 68
5.2 L'UML dell'ECoWare Bus Wrapper 70
5.3 L'implementazione del Main Node 72
5.4 L'UML del nodo centrale 73
5.5 L'implementazione dell'agente 74
5.6 L'UML dell'agente distribuito 80
5.7 Uno schema riassuntivo dell'implementazione 81

6.1 Monitoraggio ed Analisi in RUBiS 85
6.2 Performance Model 87
6.3 La variazione del mix delle richieste nel primo esperimento . . 89
6.4 Il numero di server allocati dai tre approcci nel primo esperimento 89
6.5 Il 95esimo percentile del response time nel primo esperimento 89
6.6 La variazione del mix delle richieste nel secondo esperimento 91
6.7 Il numero di server allocati dai tre approcci nel secondo esperimento 91
6.8 Il 95esimo percentile del response time nel secondo esperimento 92
6.9 UML degli Actuator usati nell'esempio 95
6.10 UML dell'interfaccia ad Amazon AWS 97
6.11 La variazione del numero di azioni rispetto al tempo d'esecuzione 99
6.12 Confronto delle due fasi principali di esecuzione 100
6.13 Overhead ed esecuzione 101
6.14 Rilevazioni durante l'esecuzione di diversi piani 103