Feedback architetturale basato su sistematica interpretazione di software performance analysis

20 luglio 2006

Indice

1 Introduzione **1**

2 Software Process e Performance Analysis **4**

3 Related Work **9**

3.1 The PASA Approach 10

3.2 The Parsons' Approach 12

3.3 The Sancho's Approach 14

3.4 The Barber's Approach 15

3.5 The Dobrzanski's Approach 16

4 Interpretazione dei Risultati di Performance e Generazione di Feedback Architetturale **18**

4.1 Matrici d'interpretazione 24

4.2 Antipattern 31

4.2.1 Blob 32

4.2.2 Unbalanced Processing: "Pipe and Filter" Architecture 36

4.2.3 Unbalanced Processing: Extensive processing 38

4.2.4 Circuitous Teasure Hunt 41

4.3 LQN 43
4.4 Antipattern modellati come LQN 46

5 Studio dell'Automatizzazione del Processo: un Primo Tool di Supporto 55

6 Un Caso di Studio: Crazy-Robot 72
6.1 Esecuzione della metodologia 76
6.2 Esempio di utilizzo del tool di supporto 99

7 Conclusioni 105

A Sorgenti Modelli LQN 108

B Sorgenti Tool di Supporto 122

Ringraziamenti 134

Bibliografia 136

Elenco delle figure

4.1 *Esempio di sotto-sistemi di tipologia 1* 22

4.2 *Esempio di sotto-sistemi di tipologia 2* 23

4.3 *Matrice d'interpretazione - Granularità Sistema* 26

4.4 *Matrice d'interpretazione - Granularità Sotto-Sistema tipo 1* . 27

4.5 *Matrice d'interpretazione - Granularità Sotto-Sistema tipo 2* . 28

4.6 *Matrice d'interpretazione - Granularità Risorsa Software* . . . 29

4.7 *Matrice d'interpretazione - Granularità Risorsa Hardware* . . 30

4.8 *Blob* 33

4.9 *Blob, una variante* 34

4.10 *Blob* - ristrutturato 35

4.11 *Blob, una variante* - ristrutturato 35

4.12 *Unbalanced Processing: "Pipe and Filter"* 36

4.13 *Unbalanced Processing: "Pipe and Filter"* - ristrutturato 37

4.14 *Unbalanced Processing: Extensive processing* 39

4.15 *Unbalanced Processing: Extensive processing* - ristrutturato . . 40

4.16 *Circuitous Teasure Hunt* 42

4.17 *Circuitous Teasure Hunt - una variante* 42

4.18 *LQN Blob* 47
4.19 *LQN Blob - Indici di interesse* 48
4.20 *LQN "Pipe and Filter" Architecture* 49
4.21 *LQN "Pipe and Filter" Architecture - Indici di interesse* 50
4.22 *LQN Extensive processing* 51
4.23 *LQN Extensive processing - Indici di interesse* 52
4.24 *LQN Circuitous Teasure Hunt* 53
4.25 *LQN Circuitous Teasure Hunt - Indici di interesse* 54

5.1 *Activity Diagram di alto livello* 56
5.2 *Activity Diagram - Livello Sistema* 58
5.3 *Activity Diagram - Livello Sotto-Sistema* 59
5.4 *Activity Diagram - Livello Risorsa* 60
5.5 *Tool - Step0, modello LQN in input* 64
5.6 *Tool - Step1 (sx) e Step2 (dx) con risoluzione antipattern* . . . 67
5.7 *Tool - Indici di interesse per il caso con risoluzione antipattern* 68
5.8 *Tool - Step1 (sx) e Step2 (dx), caso con rollback* 70
5.9 *Tool - Indici di interesse per il caso con rollback* 71

6.1 *Sequence Diagram, gestione evento pericoloso* 74
6.2 *Sequence Diagram, gestione evento regolare* 75
6.3 *Modello LQN iniziale - Iteration0* 79
6.4 *Indici di interesse - Iteration0* 80
6.5 *Modello LQN - Iteration1* 84
6.6 *Indici di interesse - Iteration1* 85
6.7 *Modello LQN - Iteration2* 89

6.8 *Indici di interesse - Iteration2* 90

6.9 *Utilizzazione risorse SubS_ regular - Iteration2a* 92

6.10 *Utilizzazione risorse SubS_ regular - Iteration3a* 93

6.11 *Modello LQN - Iteration3* 96

6.12 *Indici di interesse - Iteration3* 97

6.13 *Riepilogo performance da Iteration0 a Iteration3* 98

6.14 *Riepilogo Tempo Medio di Risposta su getEvent* 99

6.15 *Modello LQN generato dal tool per il caso di studio - Step1* . . 101

6.16 *Modello LQN generato dal tool per il caso di studio - Step2* . . 103

6.17 *Indici di interesse, applicazione del tool di supporto* 104

Elenco delle tabelle

4.1 *Profilo Blob* 36

4.2 *Profilo Unbalanced Processing: "Pipe and Filter" Architecture* . 38

4.3 *Profilo Unbalanced Processing: Extensive processing* 41

4.4 *Profilo Circuitous Teasure Hunt* 43

6.1 *Parametri iniziali del modello LQN considerato* 77

6.2 *Riepilogo requisiti - Iteration0* 81

6.3 *SottoSistemi - Iteration1* 83

6.4 *Riepilogo requisiti - Iteration1* 86

6.5 *SottoSistemi - Iteration2a* 87

6.6 *SottoSistemi - Iteration2b* 88

6.7 *Riepilogo requisiti - Iteration2* 91

6.8 *Riepilogo requisiti - Iteration3* 95

6.9 *Riepilogo Performance, risoluzione manuale ed automatica* . . 102

Capitolo 1

Introduzione

Nei moderni sistemi software le performance rappresentano ormai un aspetto dal quale non si può prescindere ed è per questo consigliabile prenderlo in considerazione già nelle prime fasi del ciclo di vita del software stesso. In genere accade però che una combinazione di vari fattori, tra i quali una data di rilascio troppo vicina, la mancanza di competenze specifiche all'interno del team di sviluppo o la scarsa conoscenza del contesto applicativo, portino a sottovalutare i problemi prestazionali con conseguente aumento della difficoltà, che peraltro cresce al susseguirsi delle varie fasi previste dal processo software utilizzato [1], nella loro successiva risoluzione, specie nel caso di sistemi ad elevata complessità.

L'obiettivo di questa tesi è delineare una possibile metodologia di analisi e risoluzione di problemi a carattere prestazionale che operi a livello architetturale, in modo da poter essere impiegata già dalle prime fasi del ciclo di vita del software, che sia model-based, ma che non vincoli i designer all'adozione di particolari modelli di performance o processi software e che offra strumen-

ti di supporto decisionale utili nella progettazione di architetture software, senza la necessità di possedere particolari competenze nel campo dell'analisi qualitativa del software e quindi degli aspetti non funzionali dello stesso, nel tentativo di colmare il divario che spesso si rileva tra sviluppatori ed analisti.

A tal fine è stato quindi realizzato un approccio per l'interpretazione sistematica delle informazioni di carattere prestazionale derivanti dalla risoluzione di modelli di performance. L'approccio può essere sintetizzato in due fasi fondamentali: la prima, *identificativa*, nella quale si cerca di determinare la tipologia e le principali caratteristiche del problema che causa anomalie prestazionali nel sistema e la seconda, *propositiva*, in cui vengono suggerite possibili soluzioni allo stesso; queste ultime, insieme ad informazioni riguardanti lo stato del sistema esaminato, formano il *feedback architetturale*, frutto dell'interpretazione di particolari indici d'interesse.

L'architettura software viene considerata a vari livelli di dettaglio ai quali sono associati diversi indici prestazionali d'interesse e matrici d'interpretazione, all'interno di queste ultime sono contenute informazioni sulle possibili cause riguardanti il problema eventualmente rilevato ed una serie di tecniche che suggeriscono potenziali strategie risolutive: come ad esempio la ristrutturazione di antipattern, particolari strutture presenti nel sistema software che ne influenzano negativamente le performance, e la clonazione di risorse software e hardware presenti nell'architettura in esame [16].

La struttura della tesi è la seguente: il capitolo 2 contiene un'introduzione riguardante i modelli di processo software e come si colloca la performance analysis rispetto ad essi; nel capitolo 3 sono trattati alcuni degli approcci esistenti riguardanti il feedback architetturale e le loro principali differenze con

l'approccio proposto in questo lavoro di tesi; quest'ultimo è invece descritto in maniera dettagliata nel capitolo 4 partendo dai concetti sui quali esso si fonda, i diversi livelli di dettaglio considerati, le schematizzazioni utilizzate, fino ad arrivare all'iteratività del processo di raffinamento architetturale; nel capitolo 5 è illustrata una possibile automazione dell'approccio descritto in questa tesi attraverso l'implementazione di un tool di supporto in grado di risolvere alcuni dei principali problemi prestazionali identificati e di fornire una rappresentazione grafica dei risultati ottenuti; il capitolo 6 contiene un caso di studio nel quale l'approccio viene utilizzato in tutti i suoi passi su un'architettura software impiegata in un robot con capacità di interazione ambientale; infine nel capitolo 7 sono illustrate, oltre alle conclusioni e ai risultati raggiunti, le maggiori difficoltà incontrate ed alcuni degli sviluppi futuri di particolare interesse.

Capitolo 2

Software Process e Performance Analysis

Fanno parte di un processo software una serie di attività, più o meno complesse ed articolate, strettamente connesse con lo sviluppo di un sistema software, la cui definizione e caratterizzazione viene guidata principalmente dalla creatività e dall'esperienza di coloro che su di esso lavorano. Esistono però degli aspetti che accomunano le varie tecniche: un buon punto di inizio consiste in genere nel collezionare accuratamente le specifiche del software che si intende realizzare e cioè definire cosa esso dovrà essere in grado di fare; successivamente i requisiti così raccolti ed eventualmente formalizzati dovranno essere interpretati e tradotti nell'applicazione vera e propria che, una volta realizzata, sarà validata rispetto agli stessi. La fase successiva è in genere descritta come l'evoluzione del software, quest'ultimo potrà infatti essere ulteriormente plasmato, in base alle necessità che si manifesteranno durante la sua messa in opera, mediante il raffinamento o l'introduzione di nuovi requisiti.

Trattandosi di approcci molto legati ad aspetti principalmente umani, quindi di difficile definizione ed influenzabili da una moltitudine di fattori non predicibili, non esiste attualmente un processo software ideale ma solo delle astrazioni, dette *software process model* o modelli di processo software, che prendono in considerazione gli aspetti fondamentali di un processo software fornendo una sequenziazione degli stessi, è poi compito degli sviluppatori scegliere ed istanziare il modello di processo sugli strumenti a loro disposizione. Tra i modelli più classici, accuratamente trattati in letteratura [5], troviamo il *waterfall model* e l'*evolutionary development* (sicuramente tra i più utilizzati nella pratica), il *formal system development* ed il *reuse-based development* che, tra quelli elencati, è il più attuale. Quest'ultimo si basa sulla possibilità di riutilizzare porzioni di software già esistenti in modo tale da integrarle tra loro nel sistema software che si intende realizzare, con il principale vantaggio di un più rapido sviluppo, o mero assemblamento nei casi più estremi. Tendenzialmente gli sviluppatori cercano di riutilizzare del lavoro già svolto in precedenza come forma di ottimizzazione di quello in corso, ma nel caso delle componenti software si è assistito alla delineazione, che prosegue tutt'oggi, di un vero e proprio mercato grazie alle loro peculiari caratteristiche, quali la capacità di astrarre da dettagli implementativi che possono essere quindi nascosti all'utilizzatore e alla possibilità di essere trattate come sistemi stand-alone: si pensi ad esempio ai *web service*, di recente introduzione, come a "componenti" software remote.

Un'altro processo, anch'esso ispirato dal riuso ma applicabile solo a software accomunati dallo stesso dominio applicativo, è quello delle *product line* [5]: l'idea alla base di tale approccio consiste nell'utilizzo di un insieme di

componenti software tra le quali c'è un riutilizzo sistematico di una porzione del sistema denominata "core" alla quale andranno affiancate una serie di altre componenti a seconda delle esigenze espresse nei requisiti.

A differenza dei processi software nei quali il sistema viene interamente realizzato da zero, nei modelli fortemente incentrati sul concetto di riuso può accadere, specie al diminuire del livello di dettaglio riguardante le componenti utilizzate, che nell'evoluzione del software siano le caratteristiche di queste ultime ad influenzare in qualche modo i requisiti e non viceversa; qualora tale adattamento non fosse però accettabile, potrebbe essere necessario sostenere dei costi aggiuntivi per la sostituzione di una o più componenti divenute non compatibili, infatti la non disponibilità del codice sorgente, tipica dei tradizionali sistemi basati su componenti *Commercial-Off-The-Shelf* o *COTS*, rende impossibile la loro modifica e quindi il pieno soddisfacimento dei nuovi requisiti.

Un processo software abbastanza recente ma soprattutto rivoluzionario rispetto alle soluzioni e alle tendenze del passato è rappresentato dall'approccio Open Source[1]; esso è caratterizzato principalmente dal coinvolgimento, nelle varie fasi dello sviluppo, di quelli che saranno gli utilizzatori finali del software stesso, specialmente per quanto concerne le fasi di testing e la definizione dei requisiti. Inoltre, grazie a questa collaborazione, il processo software affronta, spesso con successo, le problematiche di tipo manageriale legate ai vincoli temporali di rilascio o ad aspetti economici [6].

[1]Si è scelto di utilizzare tale termine piuttosto che Software Libero, ispiratore e arzillo progenitore dell'Open Source, con l'intento di astrarre volutamente dalle sue basi filosofiche, concentrandosi esclusivamente sugli aspetti ingegneristici e metodologici che lo caratterizzano

Essenzialmente per la sua estrema dinamicità esso può quindi difficilmente essere descritto da modelli tradizionali e rigidi quali il "waterfall", mentre ben si sposa, ad esempio, con approcci più dinamici, che prevedano cioè la presenza di feedback da una fase verso le precedenti ed orientati al riuso, all'interno dei quali l'Open Source potrebbe giocare un ruolo fondamentale specie nella riduzione dei costi legati alla manutenzione di sistemi software e alla loro evoluzione, descritti in precedenza.

Nei moderni sistemi software le performance rappresentano ormai un aspetto essenziale, il che ha portato all'introduzione di quella che viene definita *performance analysis*; con tale termine si intende la misurazione, la stima e l'interpretazione di un insieme di indici prestazionali di interesse relativi ad un dato sistema software. Il numero e la tipologia di tali indici variano in funzione di quest'ultimo ed in particolar modo in base alla sua natura.

In letteratura si trovano diversi approcci, alcuni dei quali saranno trattati nel capitolo 3, relativi all'analisi prestazionale di architetture software; una possible classificazione di tali metodologie può essere fatta considerando la fonte utilizzata dalle stesse per il calcolo degli indici prestazionali d'interesse: ad esempio alcune fanno uso di modelli di performance, siano essi analitici o simulativi, altre invece, nel caso di sistemi già implementati, impiegano dati derivanti dal monitoraggio, oppure ottenuti dall'esecuzione di benchmark.

Attualmente sembra non esistere un modello perfetto che si presti cioè ad un utilizzo universale [5]: ogni modello può essere più o meno indicato a seconda del sistema software oggetto di studio e la loro adozione non è esclusiva. Possono infatti essere adottati approcci ibridi, ad esempio i dati raccolti dal monitoraggio di un sistema software esistente possono alimen-

tare la costruzione di un profilo operazionale adeguato, da utilizzare con un modello di performance per lo studio di una nuova architettura software.

Lo studio delle performance dovrebbe essere integrato quindi con i vari processi software, a partire dalle prime fasi della progettazione fino alla sua messa in opera, ad esempio partendo dalla modellazione di alto livello fino ad arrivare al monitoraggio del sistema implementato. Disporre di informazioni di carattere prestazionale durante tutto il ciclo di vita del software aiuta nei processi decisionali e può evitare onerose, in termini economici e di tempo, ristrutturazioni dei sistemi in fase avanzata di realizzazione [1, 2].

A tal fine è importante disporre di strumenti di supporto, utili nella progettazione di architetture software, che siano utilizzabili dai designer senza la necessità, per questi ultimi, di disporre di particolari competenze nel campo dell'analisi qualitativa del software e quindi degli aspetti non funzionali dello stesso, nel tentativo di colmare il divario che spesso si rileva tra sviluppatori[2] ed analisti di performance.

Nello scenario appena descritto gioca un ruolo importante quello che si definisce *feedback architetturale* e cioè il fornire, principalmente ad architetti ed ingegneri del software e seguendo diverse metodologie che variano a seconda dell'approccio utilizzato, una qualche tipologia di analisi degli indici di performance che possa essere d'aiuto nell'individuare e risolvere eventuali problemi presenti in una data architettura software in esame.

[2] È stato utilizzato il termine generico sviluppatori, piuttosto che ingegneri o architetti del software, a sottolineare il fatto che tutti i soggetti coinvolti durante il ciclo di vita del software potrebbero trarre benefici dallo studio degli aspetti prestazionali dello stesso

Capitolo 3

Related Work

Esistono attualmente in letteratura diversi approcci riguardanti il feedback architetturale di alto livello, di seguito sono trattati alcuni tra quelli giudicati maggiormente affini, rispetto agli scopi e agli aspetti trattati, a questo lavoro di tesi. Una delle principali differenze tra le varie tecniche esaminate consiste nell'utilizzo, da parte di alcune di esse, di approcci model-based, che fanno quindi uso di modelli di performance per lo studio di architetture software, mentre altre, di tipo monitoring-based, incentrano il calcolo delle prestazioni sul collezionamento di informazioni di questo tipo dall'esecuzione del sistema software in esame. Di conseguenza le prime saranno utilizzabili già dalle prime fasi del ciclo di vita del software per lo studio sia di architetture in fase di progettazione, ma non ancora realizzate, che di sistemi già implementati, mentre quelle basate sul monitoraggio possono essere utilizzate solo su questi ultimi e la loro applicazione, nell'intero ciclo di vita del software, è limitata ai processi di sviluppo che prevedono il rilascio di implementazioni parziali ed eseguibili del sistema considerato.

3.1 The PASA Approach

L'approccio descritto da Williams&Smith [8] si basa sul principio che, per il raggiungimento di determinati obiettivi di performance, un buon punto di partenza consiste nella comprensione dell'architettura software del sistema in esame; questo infatti permette di raggiungere i livelli prestazionali desiderati in maniera più veloce e ad un più basso costo. Il PASA si fonda inoltre sui principi e le tecniche definite, dagli stessi autori, nel "Software Performance Engineering (SPE)" [9] utilizzato per stabilire se un'architettura è in grado di raggiungere determinati livelli di performance.

Il processo PASA consiste essenzialmente nella collaborazione, che si protrae in genere per qualche settimana, tra il team coinvolto nello sviluppo di un determinato software ed un team di analisti specializzati nella verifica e validazione di architetture software; a tal fine è fondamentale che tutti i soggetti coinvolti conoscano: la logica alla base di una verifica architetturale, gli obiettivi ed i dettagli di interesse dell'SPE, gli step previsti dal processo PASA, le informazioni necessarie alla realizzazione di tale verifica ed i tradeoff tra le performance e gli altri attributi qualitativi di un sistema software.

I primi incontri sono orientati alla comprensione dell'architettura realizzata o semplicemente progettata dal team di sviluppo, anche attraverso lo studio approfondito di *use case* di particolare interesse e l'individuazione di *performance scenario*, insieme all'identificazione di precisi obiettivi prestazionali e alla caratterizzazione del carico di lavoro stimato.

Collezionate le informazioni necessarie, si può passare all'analisi vera e propria dell'architettura: si cerca di individuare lo stile architetturale utiliz-

zato nella progettazione al fine di stabilire se questo può essere considerato appropriato o meno rispetto ai requisiti indicati in precedenza.

Il passo successivo consiste nel rilevare la presenza di antipattern, che verranno trattati in dettaglio più avanti in questa tesi, all'interno dell'architettura in esame ed alla loro risoluzione attraverso il refactoring, trasformazione correctness-preserving che migliora la qualità del software, nell'eventualità ne venisse accertata la presenza. Inoltre, nel caso in cui si ritenesse necessario, si procederà ad un'analisi model-based utilizzando "software execution model" e "system execution model", i quali forniranno informazioni che dovranno essere studiate ed interpretate dagli analisti.

Qualora nei passi precedenti venissero evidenziati problemi di carattere prestazionale, sarà necessario identificare delle possibili soluzioni che possono essere rappresentate, a seconda dei casi, da deviazioni rispetto allo stile architetturale adottato, revisione delle interazioni tra le componenti che formano il sistema ed il refactoring degli eventuali antipattern presenti, considerando che la soluzione dipenderà dalle caratteristiche del sistema.

Tutto lo studio svolto dagli analisti sull'architettura software ed i risultati ottenuti dovranno poi essere illustrati al team di sviluppo, in modo tale che possa prendere coscienza delle verifiche effettuate e trarne beneficio; verrà inoltre stilata un'analisi di tipo economico per stimare i costi, sia in termini di denaro che di tempo, da sostenere per apportare le modifiche proposte all'architettura esaminata.

Gli autori stanno attualmente lavorando sul progetto di codificare i risultati delle loro esperienze con il PASA, in osservazioni e suggerimenti più generici riguardanti l'applicabilità dei vari stili architetturali su particolari tipologie di applicazioni.

La differenza principale con l'approccio descritto in questo lavoro di tesi riguarda il modo di affrontare il problema: quest'ultimo ha come obiettivo offrire uno strumento di supporto decisionale che fornisca, agli sviluppatori stessi ed in particolare ad architetti del software e designer, informazioni e suggerimenti sotto forma di feedback architetturale, il più possibile automatizzato, tali da aiutare nella risoluzione sistematica di problemi qualitativi legati alle performance del sistema software che intendono realizzare; mentre il PASA si basa sulla stretta collaborazione tra sviluppatori ed analisti.

3.2 The Parsons' Approach

Consiste in un framework che ha come obiettivo automatizzare la rilevazione di design e deployment antipattern all'interno di sistemi software a componenti [10]. Si tratta di un approccio monitoring-based, in quanto attraverso il monitoraggio di un sistema in esecuzione si collezionano informazioni riguardanti le sue performance; tale caratteristica lo rende di conseguenza applicabile solo all'interno di un processo software che preveda il rilascio di un'implementazione parziale del software da eseguire ed analizzare alla fine di ogni "ciclo" di sviluppo.

L'approccio è di tipo sequenziale e si compone di cinque fasi: il *monitoraggio*, che consiste nel collezionare informazioni sulle prestazioni del sistema in esecuzione e la sua composizione, classificandole in diverse categorie a seconda della loro natura; l'*analisi*, in questa fase le informazioni raccolte nella precedente vengono esaminate, ricavando la struttura del sistema in esame e informazioni sulle performance delle singole componenti che lo formano;

la *rilevazione*, nella quale viene utilizzato un approccio rule-based per verificare la presenza di antipattern, opportunamente codificati sotto forma di regole, partendo dalle informazioni fornite dall'analisi. Qualora venisse individuato un antipattern si passa alla fase di *valutazione*, all'interno della quale dovrà essere considerato il modello di performance che lo rappresenta, creato in qualche modo separatamente, al fine di stimare il suo impatto sulle prestazioni del sistema; nell'ultima fase, quella di *presentazione*, si provvede a confezionare un feedback, composto essenzialmente dagli antipattern individuati nella fase precedente con relativa soluzione sotto forma di diagrammi UML, che gli sviluppatori potranno utilizzare per comprendere gli errori presenti e porne rimedio.

Gli autori stanno lavorando alla creazione di un tool, attualmente solo parzialmente realizzato, in grado di automatizzare tale tecnica.

L'automazione del procedimento appena illustrato riguarda solo il calcolo e la presentazione dei risultati. L'approccio descritto in questa tesi invece, operando a livello architetturale, può essere automatizzato in modo che un determinato tool sia in grado di apportare delle modifiche sull'architettura software, senza quindi l'intervento dei designer, valutandone opportunamente gli effetti; in tal caso si lavora in direzione di un processo "closed-loop" dall'architettura al modello di performance. Inoltre l'approccio presentato in questo lavoro di tesi non si limita alla rilevazione degli antipattern ma propone possibili soluzioni anche nel caso in cui questi non fossero presenti nel sistema o non siano, per qualche motivo, riconoscibili.

3.3 The Sancho's Approach

In quest'approccio [7], attualmente ancora oggetto di studio, gli autori si pongono come obiettivo quello di effettuare una diagnosi a livello architetturale di un sistema software e fornire suggerimenti su come eventualmente migliorare le sue performance o comunque offrire agli sviluppatori, in particolar modo ai designer, uno strumento utile nell'individuazione delle componenti che presentino problemi di carattere prestazionale all'interno dell'architettura in esame.

Assumendo di considerare sistemi sufficientemente complessi, tali da essere difficili da analizzare analiticamente, gli autori basano la loro tecnica su un approccio di tipo simulativo; attraverso quest'ultimo vengono collezionate informazioni e memorizzato, nel dettaglio, lo stato del sistema in determinate condizioni di carico particolarmente interessanti che verranno poi utilizzate per analizzare il sistema, comprenderne la struttura e validarne gli aspetti qualitativi legati alle performance.

Il feedback architetturale consiste nel fornire suggerimenti utili a rendere stabile un sistema che non lo è oppure, in caso contrario, nel calcolo delle massime performance raggiungibili dal sistema considerato.

L'approccio descritto in questo lavoro di tesi condivide con quello appena illustrato i principi e le idee di base, in special modo quelle riguardanti le caratteristiche del feedback architetturale, mentre si differenzia nella sistematicità dell'analisi e sulla formalizzazione dei criteri utilizzati, che non si evincono invece dalla descrizione fornita nella pubblicazione esaminata.

3.4 The Barber's Approach

Consiste in una suite di processi e tool di supporto alla derivazione di un'architettura software basata sull'acquisizione iterativa di informazioni riguardanti i requisiti del sistema, che non riguardano solo le performance ma anche altri aspetti non funzionali come maintainability, reusability e safety, e la valutazione delle stesse, realizzando un graduale raffinamento dell'architettura in esame [11]. L'approccio può essere distinto in due parti fondamentali: un'analisi di tipo statico implementata, attraverso delle euristiche, nel tool denominato RARE ed un'altra dinamica, di tipo simulativo per quanto riguarda le performance, eseguita dal tool ARCADE.

Considerando nello specifico le performance come aspetto di interesse, entrambe le tipologie di analisi possono fornire informazioni utili: quelle derivanti da un'analisi statica del sistema sono correlate con il numero di componenti che una richiesta dovrà attraversare per essere servita, mentre un'analisi dinamica può mettere in evidenza la presenza di bottleneck.

Le euristiche sono rappresentate da suggerimenti definiti da esperti in base alle loro competenze ed esperienze passate, queste vengono applicate nel sistema in esame attraverso delle strategie che guidano gli architetti del software nelle modifiche da effettuare; una singola euristica può coinvolgere anche più di una strategia e ognuna di queste appartenere a più di una euristica. Per valutare quanto l'architettura rispetti i principi definiti in queste ultime e gli effetti delle eventuali modifiche apportate, vengono utilizzate delle apposite metriche per il calcolo delle varie caratteristiche dell'architettura stessa.

Le euristiche vengono valutate da RARE considerando i feedback provenienti dalle analisi svolte da ARCADE come metriche aggiuntive; in pratica il primo tool esegue una valutazione più grezza dell'architettura che sarà poi raffinata utilizzando le informazioni fornite dal secondo. Inoltre RARE non nasce con l'intento di essere una soluzione completamente automatica, ma dalla volontà di collaborare con gli architetti del software attraverso la comunicazione delle esperienze codificate nelle euristiche, le quali consistono nella formalizzazione di suggerimenti del tipo "all'interno di un sistema Object Oriented è bene mantenere le componenti debolmente accoppiate tra loro".

L'approccio appena descritto abbraccia diverse tipologie di requisiti non funzionali oltre alle performance ed utilizza inoltre un formalismo ben definito per quanto riguarda i modelli utilizzati per l'analisi degli aspetti dinamici di un'architettura software, mentre quello descritto in questa tesi utilizza una tecnica che astrae dal formalismo utilizzato per il calcolo delle performance, che rappresentano inoltre l'unico aspetto non funzionale considerato.

3.5 The Dobrzanski's Approach

Pur non essendo strettamente correlato con le performance in quanto focalizzato sulla maintainability di un sistema software ed in particolare al fenomeno del *design erosion*, l'approccio descritto in [23] risulta comunque interessante sia per il fatto di essere sistematico ed orientato all'automazione sia per l'utilizzo di quelli che vengono definiti "bad smell". Questi ultimi rappresentano determinate strutture che indicano la possibilità di un refactoring; la

loro presenza è infatti una sorta di avvertimento riguardante la possibilità di incorrere in particolari problematiche riguardanti aspetti non funzionali del software in esame. Inoltre, almeno teoricamente, il miglioramento della qualità di quest'ultimo è legato alla riduzione dei bad smell in esso presenti.

La tecnica prevede, per ogni trasformazione supportata, la definizione di una specifica informale contenente informazioni di vario genere sulla natura del problema rilevato, gli obiettivi del refactoring, una descrizione del bad smell considerato ed una serie di precondizioni e postcondizioni; allo stesso tempo la trasformazione sarà descritta in maniera formale in un linguaggio OCL-like, che ne facilita l'implementazione.

L'approccio appena descritto e quello oggetto di questa tesi, pur operando su aspetti qualitativi del software diversi tra loro e a differenti livelli di dettaglio, sono accomunati da concetti base quali il refactoring, la trasformazione orizzontale di modelli software e dall'uso di particolari strutture di riferimento come gli antipattern ed i bad smell concettualmente simili tra loro.

Capitolo 4

Interpretazione dei Risultati di Performance e Generazione di Feedback Architetturale

L'approccio descritto in questo lavoro di tesi opera ad un livello sufficientemente astratto, quello architetturale, che permette di potrerlo utilizzare già nelle prime fasi del ciclo di vita del software, in particolar modo durante la progettazione, ma può anche essere eventualmente utilizzato per esaminare un sistema software già implementato che presenti problemi prestazionali. E' di tipo model-based ma, considerando che non esiste un modello di performance universale [4], si è scelto di fare in modo che non fosse vincolante né rispetto al formalismo utilizzato per la descrizione di quest'ultimo né per quanto riguarda la tipologia di analisi adottata, che può quindi essere analitica o simulativa a seconda della complessità dell'architettura in esame e degli strumenti offerti per il formalismo adottato.

Tale tecnica può essere suddivisa in due fasi fondamentali:

- la prima, di tipo *identificativo*, nella quale, analizzando gli indici prestazionali ottenuti dalla soluzione di un modello di performance, vengono individuate delle particolari situazioni sulle quali è necessario intervenire al fine di migliorare le performance dell'architettura software in esame, qualora questa non rispetti i requisiti non funzionali indicati;
- la seconda, invece di tipo *propositivo*, consiste nell'individuazione di un ventaglio di possibili soluzioni migliorative partendo dalle osservazioni sviluppate nella fase precedente.

Il sistema software viene considerato a diversi livelli di dettaglio in maniera tale da ottimizzare l'analisi dello stesso concentrandosi solo sulle parti che presentano i problemi più gravi e rilevare l'eventuale assenza di problemi prestazionali il prima possibile.

Per architetture software sufficientemente complesse è lecito immaginare che queste possano essere suddivise logicamente in vari sotto-sistemi, ad esempio all'interno di un software di video streaming potrebbero essere identificati almeno due diversi sotto-sistemi: uno che gestisce l'autenticazione degli utenti e l'altro che si occupa dello streaming vero e proprio. Le cattive performance del sistema quindi potrebbero derivare dal non soddisfacente funzionamento solo di alcuni dei sotto-sistemi che lo compongono ed in questo caso sarà conveniente dedicarsi ad un'analisi di più basso livello solo su questi ultimi, ottenendo un vantaggio in termini di tempo computazionale rispetto ad un'analisi approfondita del sistema nella sua interezza. Si pensi ad esempio ad un'architettura a plug-in: le cattive performance della stessa

potrebbero derivare solo da una piccola parte dei plug-in considerati ed in questo caso sarebbe vantaggioso concentrarsi su questi ultimi senza quindi preoccuparsi del "core" e degli altri sotto-sistemi, per i quali non si rilevano problemi di carattere non funzionale.

I tre differenti *livelli di granularità* individuati sono:

- *sistema* è il livello più alto tra i tre, quindi il più astratto, in quanto l'architettura software viene considerata nella sua interezza; si esegue una prima validazione, in base agli obiettivi di performance indicati nei requisiti, al fine di stabilire se è necessario intervenire sull'architettura software in esame apportando modifiche migliorative oppure no;

- *sotto-sistema* rappresenta un livello di dettaglio intermedio, che sarà necessario considerare qualora la precedente verifica mettesse in evidenza problemi prestazionali, nel quale possono essere analizzate le componenti che formano il sistema e le loro interazioni; l'individuazione dei vari sotto-sistemi consiste in una suddivisione logica dell'architettura, eseguita in genere in base alla tipologia di requisiti disponibili a livello di sistema:

  - disponendo di requisiti di tipo *flat*, cioé che non si riferiscono a determinati servizi offerti ma che hanno un carattere più generale, è possibile partizionare logicamente l'architettura in sotto-sistemi distinti in base alle funzionalità da loro offerte; tale approccio risulta particolarmente intuitivo qualora il sistema fosse stato creato assemblando, ad esempio, delle componenti software preesistenti o comunque progettato in modo tale che alcuni particolari insiemi

di componenti ("cluster") risultino debolmente accoppiati tra loro. In questo modo si considerano sotto-sistemi *cross-path* rispetto ai servizi offerti dal sistema e cioé rispetto al percorso che ogni richiesta segue, per essere servita, all'interno di quest'ultimo. Di seguito si farà riferimento a sotto-sistemi di questo genere definendoli di *tipologia 1*. Una rappresentazione grafica di tale suddivisione è riportata in figura 4.1 in cui vengono individuati, nell'architettura di esempio, tre diversi sotto-sistemi, dove le risorse contenute in quello delimitato dal tratteggio di colore *rosso* implementano la parte intelligente del sistema considerato ed effettuano richieste agli altri due, in *blu* e in *arancione*, che invece rappresentano rispettivamente un insieme di servomeccanismi fisici pilotati da un controller e gli apparati di memorizzazione utilizzabili;

– se invece si dispone di requisiti specifici per i vari servizi individuati a livello di sistema, è possibile identificare dei sotto-insiemi di risorse facendo in modo che quelle utilizzate da un determinato servizio appartengano ad uno stesso sotto-sistema, percorrendo quindi il *path* seguito dalle richieste per tale servizio; il vantaggio principale di questa soluzione consiste nel fatto che i requisiti non funzionali, indicati a livello sistema per un determinato servizio, possono essere facilmente ricondotti al sotto-sistema che lo implementa. E' inoltre utile osservare che in questo modo, a differenza del caso precedente, può accadere che i vari sotto-sistemi, così individuati, possano intersecarsi tra loro considerando che una stessa risorsa può essere verosimilmente utilizzata da più di un servizio.

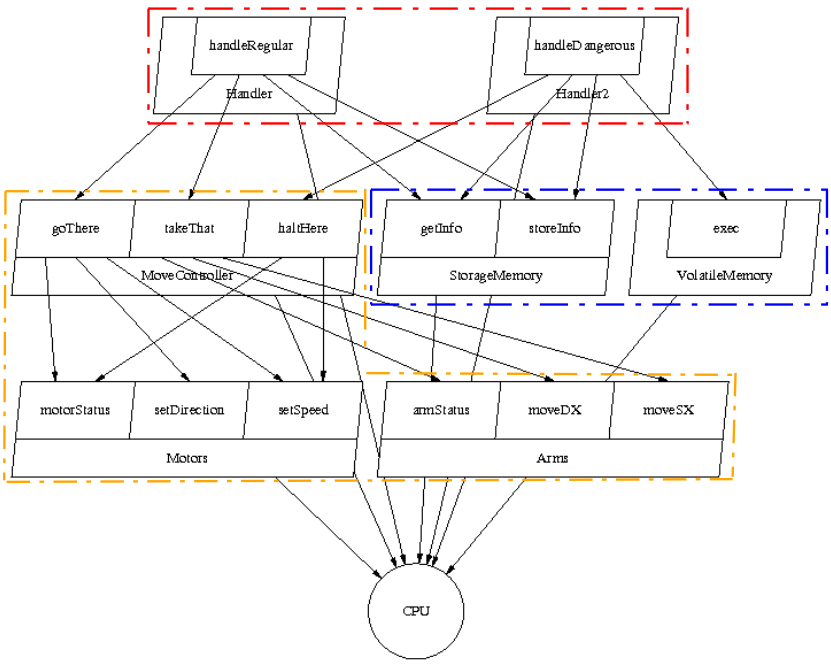


Figura 4.1: *Esempio di sotto-sistemi di tipologia 1*

Di seguito ci si riferirà a sotto-sistemi di questo tipo definendoli di *tipologia 2*. Nell'illustrazione in figura 4.2 è rappresentato graficamente quanto appena descritto, in questo caso vengono infatti individuati due sotto-sistemi, uno delimitato dal tratteggio in *blu* e l'altro da quello in *arancione*, ognuno dei quali contiene le risorse necessarie al soddisfacimento di una particolare classe di richieste, che in questo caso di esempio vengono generate dalle due risorse posizionate più in alto nel disegno.

Possono però anche essere seguiti approcci ibridi nel senso che, se il progettista lo ritiene opportuno, è possibile utilizzare un approccio di "tipologia 2" anche quando i requisiti sono di tipo flat andando ad indi-

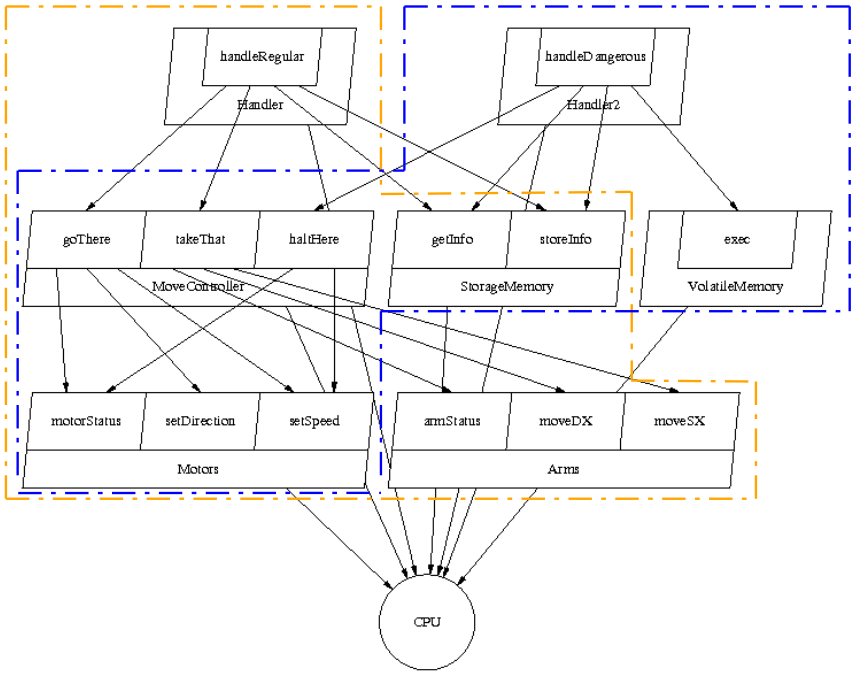


Figura 4.2: *Esempio di sotto-sistemi di tipologia 2*

viduare, ad esempio, il servizio più critico tra quelli offerti dal sistema; un discorso analogo può essere fatto per l'uso di sotto-sistemi cross-path, quindi di "tipologia 1", anche quando si dispone di requisiti specifici per i vari servizi. In quest'ultimo caso però la scelta potrebbe non essere sempre conveniente infatti, a meno che non vengano individuati dei requisiti per tutti i servizi, l'altra suddivisione offre il vantaggio di non dover considerare tutte le risorse presenti nel sistema ma solo quelle utilizzate dai servizi classificati come di particolare interesse, per i quali cioè sono stati individuati dei requisiti di tipo non funzionale. Inoltre, considerando sotto-sistemi cross-path, alcune anomalie architetturali, che coinvolgono risorse contenute in sotto-sistemi differenti, potrebbero non essere evidenziate dall'analisi.

- *risorsa* rappresenta il livello di dettaglio massimo, all'interno del quale è possibile individuare ed analizzare le due diverse tipologie di risorse che compongono il sistema: *software* e *hardware*. La differenza principale tra le due risiede nelle possibilità di intervento che nel primo caso sono maggiori, come verrà illustrato nel dettaglio in seguito, mentre per una risorsa hardware si può ricorrere ad un potenziamento o ad una sostituzione con una versione della stessa più performante a seconda dei casi, ma per la quale non si può, ad esempio, agire sull'organizzazione dei servizi offerti. Va aggiunto che il numero di particolari tipologie di risorse, presenti all'interno di un dato sistema, non può sempre essere modificato in fase di ristrutturazione preservando nel contempo le caratteristiche funzionali del sistema stesso; infatti, come sarà illustrato con un caso di studio nel capitolo 6, una risorsa hardware potrebbe rappresentare uno o più device fisici che non possono esistere in numero maggiore rispetto a quello stabilito; allo stesso modo clonare una risorsa software che svolge particolari compiti, come ad esempio un controller, potrebbe non avere riscontro nella realtà o sconvolgere il funzionamento dell'architettura software in esame.

4.1 Matrici d'interpretazione

Per ragioni di semplicità e chiarezza, le fasi identificativa e propositiva, per ogni livello di dettaglio identificato, sono state sintetizzate in quelle che sono state definite "matrici d'interpretazione", riportate nelle figure 4.3, 4.4, 4.5, 4.6 e 4.7.

In questo modo è sufficiente confrontare i valori ottenuti dalla soluzione del modello di performance per gli indici considerati d'interesse con la tipologia di requisiti a disposizione al livello di dettaglio considerato oppure rispetto a soglie predefinite.

Nello specifico, per quanto concerne l'indice di *utilizzazione* si è scelto di confrontare i suoi valori con soglie pseudo-obiettive piuttosto che prevedere la possibilità di indicare dei requisiti non funzionali su di esso. Infatti specificare tali vincoli potrebbe eventualmente essere utile a mantenere empiricamente un certo margine di scalabilità del sistema ma tale risultato è ottenibile, in maniera più precisa e senza indicare requisiti non funzionali sull'utilizzazione, definendo un opportuno profilo operazionale che sia elaborato in modo tale da considerare carichi per il sistema anche superiori a quelli previsti, al fine di apprezzarne la scalabilità.

Il *throughput* viene invece confrontato, a seconda dei casi, con eventuali requisiti non funzionali per esso definiti oppure con il suo valore massimo o comunque con un suo upper-bound; a seconda degli strumenti utilizzati per la soluzione del modello di performance, quest'ultimo valore può essere fornito automaticamente dal risolutore oppure essere calcolato, in generale, come $X_{max} = \frac{1}{S}$ dove X_{max} è il massimo valore di throughput raggiungibile ed S il tempo medio di servizio [12]. Per quanto riguarda invece il *tempo medio di risposta*, questo può essere confrontato con i requisiti non funzionali espressi in merito oppure, qualora questi non fossero disponibili, con informazioni più generiche come ad esempio, nel caso di un sotto-sistema, la sua prossimità al valore considerato accettabile per il sistema nella sua interezza.

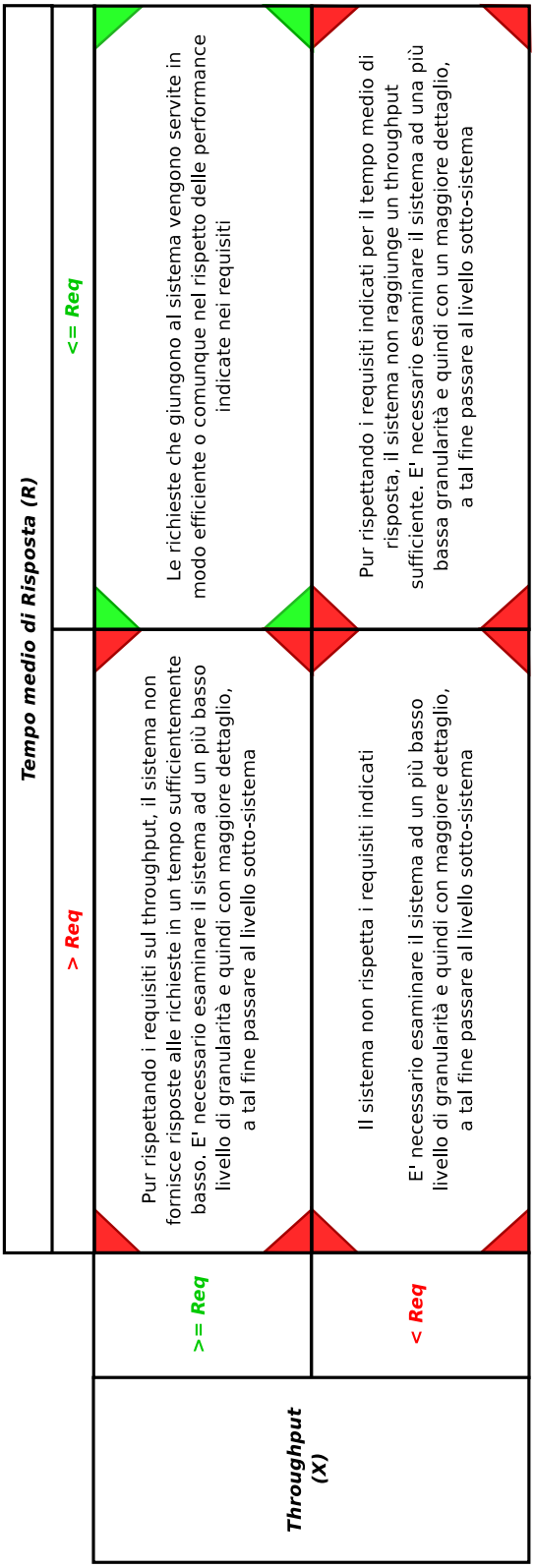


Figura 4.3: *Matrice d'interpretazione - Granularità Sistema*

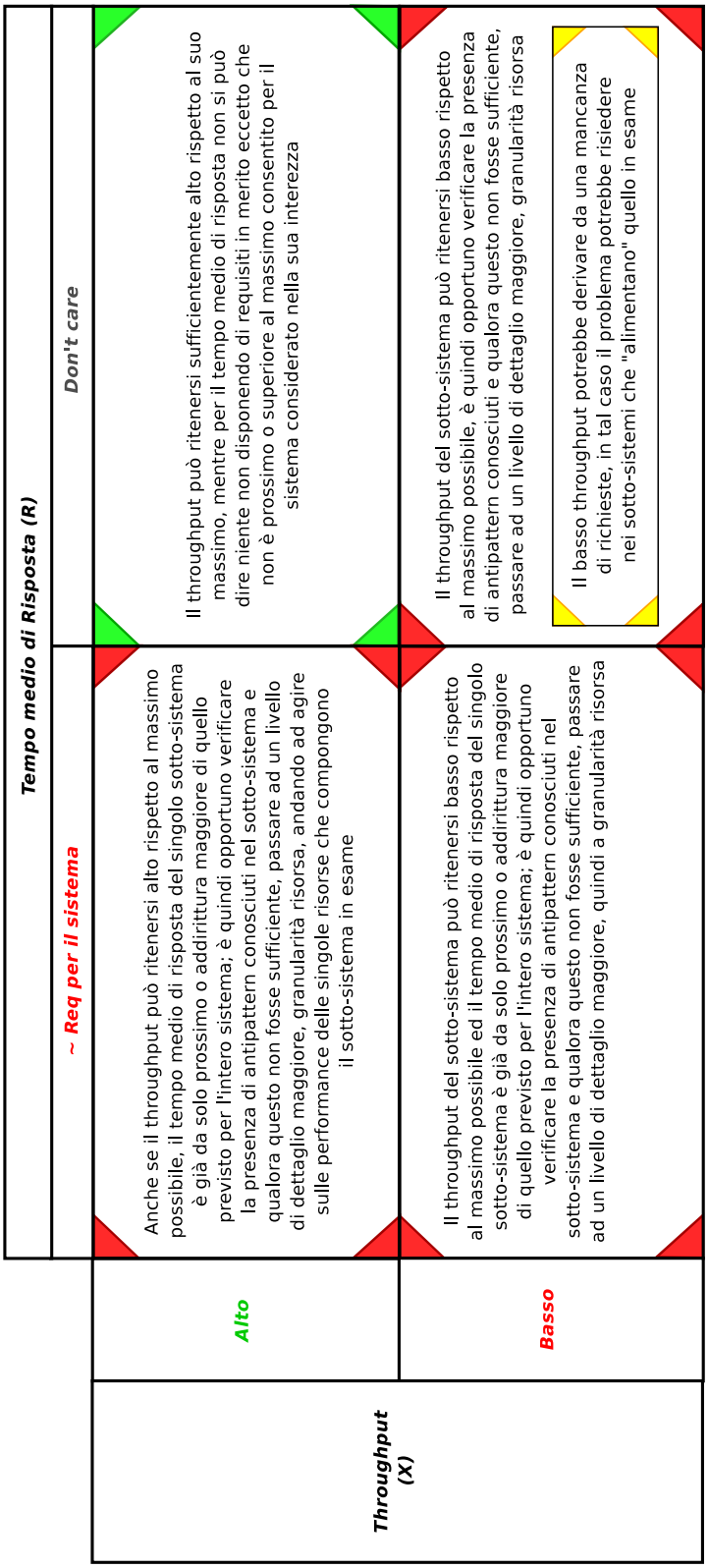


Figura 4.4: *Matrice d'interpretazione - Granularità Sotto-Sistema tipo 1*

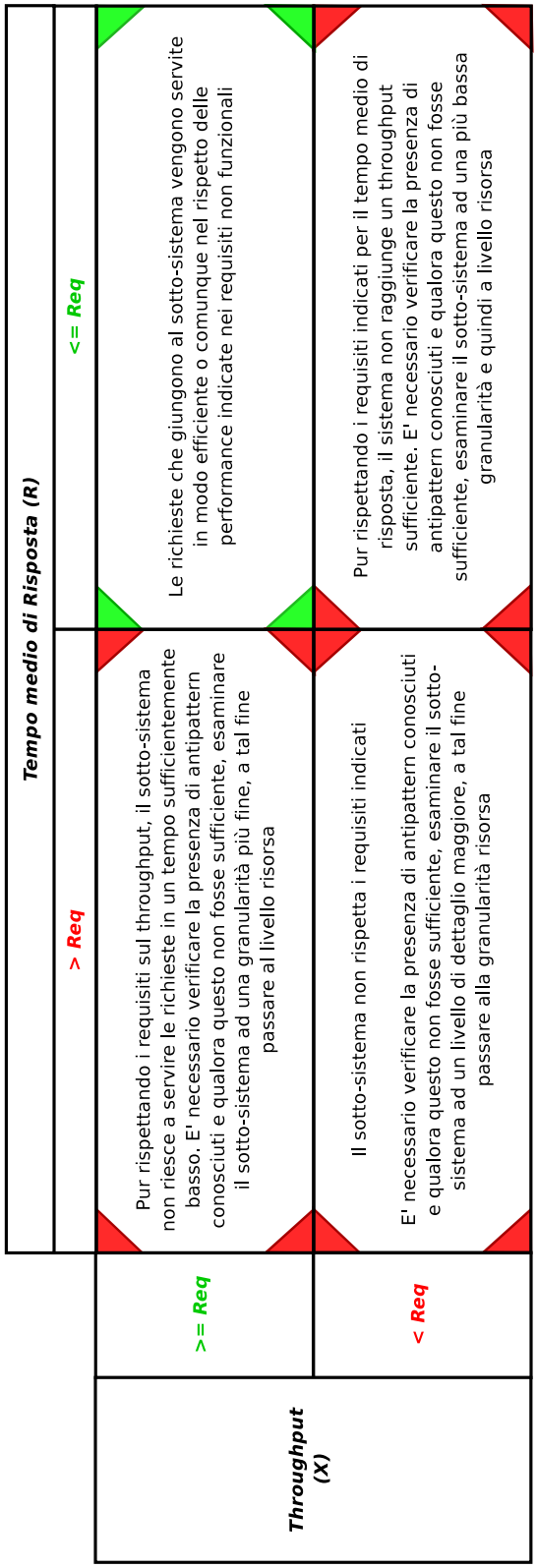


Figura 4.5: *Matrice d'interpretazione - Granularità Sotto-Sistema tipo 2*

		Tempo medio di Risposta (R)	
		~ Req per il sotto-sistema che la contiene	***Don't care***
Utilizzazione (U)	***Alto***	I servizi offerti dalla risorsa sono molto richiesti o comunque ogni richiesta occupa molto la risorsa considerata *Risorsa insufficiente* . clonazione della risorsa software . se possibile liberare la risorsa attraverso lo splitting di alcuni dei servizi ed il trasferimento degli stessi su una risorsa meno utilizzata . verificare lo stato delle risorse hardware utilizzate da quella in esame (granularità risorsa hardware)	I servizi offerti dalla risorsa sono molto richiesti o comunque ogni richiesta occupa molto la risorsa considerata *Risorsa insufficiente* . clonazione della risorsa software . se possibile liberare la risorsa attraverso lo splitting di alcuni dei servizi ed il trasferimento degli stessi su una risorsa meno utilizzata . verificare lo stato delle risorse hardware utilizzate da quella in esame (granularità risorsa hardware) Il problema potrebbe anche non risiedere nella risorsa in esame: non avendo sufficienti informazioni su R ad avvalorare l'ipotesi, è infatti difficile stabilirlo con certezza. Comunque la risorsa si trova ad operare ai limiti della scalabilità
	Basso	Il numero di richieste che giungono alla risorsa non è eccessivo ma la loro elaborazione richiede un tempo prossimo o superiore a quello dell'intero sotto-sistema *Cattiva progettazione della risorsa software in esame* . ricerca di antipattern nell'implementazione della risorsa software condiderata (necessaria la disponibilità del codice sorgente . rivisitazione degli obiettivi non funzionali a livello di sistema	I servizi offerti dalla risorsa sono poco richiesti oppure molto efficienti, nel senso che ogni richiesta necessita di "poca" risorsa per essere servita La risorsa potrebbe essere sovradimensionata rispetto al numero di richieste che riceve; tale osservazione è però valida solo quando non sono presenti problemi prestazionali nel sistema considerato, in caso contrario infatti l'eliminazione di un bottleneck "a monte" potrebbe cambiare le condizioni della risorsa in esame

Figura 4.6: *Matrice d'interpretazione - Granularità Risorsa Software*

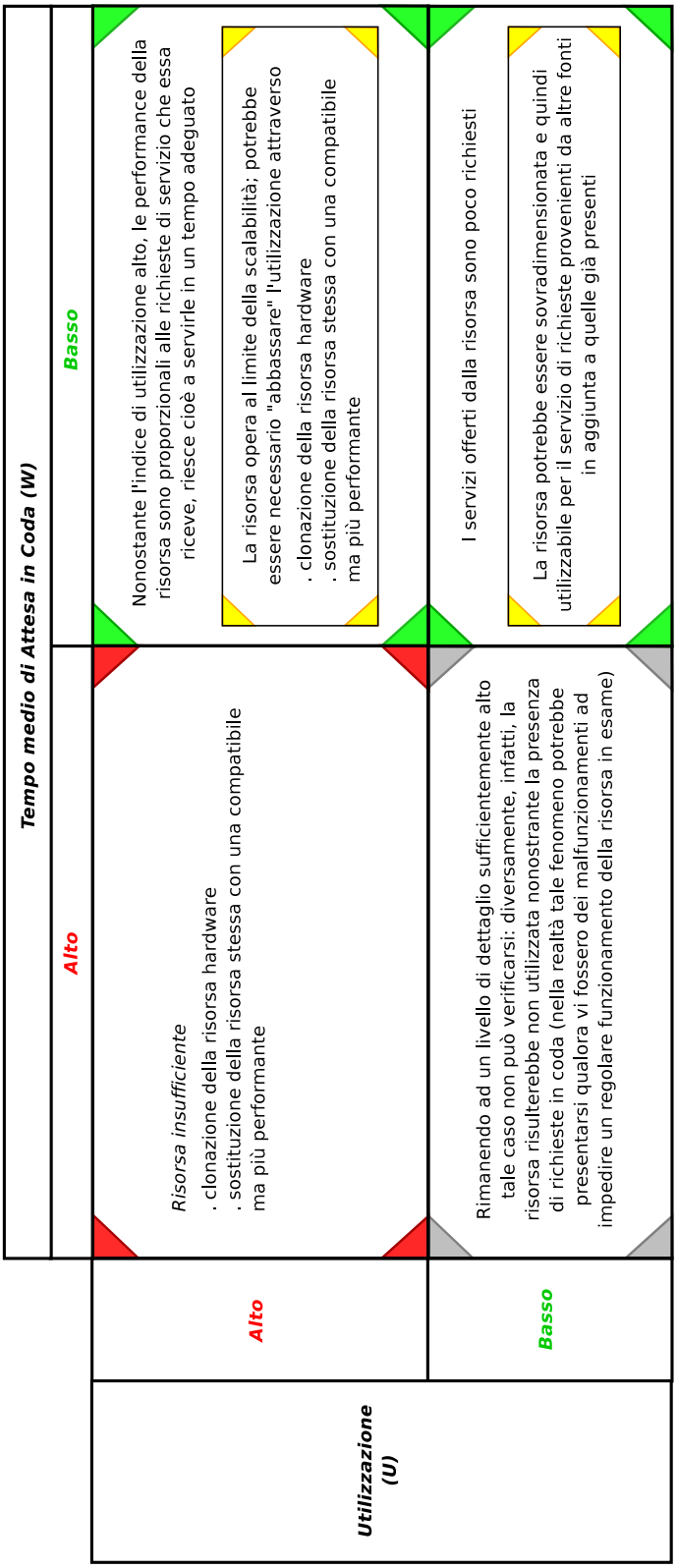


Figura 4.7: *Matrice d'interpretazione - Granularità Risorsa Hardware*

Operando tali confronti, ogni matrice d'interpretazione offre informazioni di carattere identificativo ed informativo riguardanti le possibili cause del problema eventualmente presente nell'architettura e propone una serie di possibili soluzioni come ad esempio, per il livello sotto-sistema, la ricerca di antipattern conosciuti.

4.2 Antipattern

Così come un *design pattern* rappresenta una soluzione generica ad un determinato problema, che va opportunamente contestualizzata a seconda dei casi e che nasce dall'esperienza maturata nella progettazione di architetture software, gli *antipattern*, o per meglio dire i "software performance antipattern" [13] in questo caso, sono concettualmente simili ad essi ma la loro presenza in un design produce, al contrario, conseguenze negative che nel caso specifico sono rappresentate da problemi di carattere prestazionale.

Gli antipattern suggeriscono quindi cosa è meglio evitare nel design e come risolvere un particolare problema, qualora venisse individuato nel sistema, attraverso ristrutturazioni o riorganizzazioni che preservino la correttezza del sistema stesso migliorandone nel contempo le prestazioni [13]. Tali operazioni, che compongono quello che viene definito "refactoring", consistono in *trasformazioni orizzontali* del modello architetturale originale in quanto viene mantenuto il medesimo livello di astrazione tra questo ed il modello risultante [23].

Considerata la natura del lavoro svolto in questa tesi, in seguito ci si riferirà ai software performance antipattern semplicemente con il termine

antipattern; senza tale premessa il termine potrebbe risultare ambiguo visto che esistono altre categorie riguardanti aspetti non funzionali del software diversi dalle performance, come reusability e maintainability ad esempio. Sono inoltre stati presi in considerazione solo alcuni tra i diversi antipattern presenti in letteratura [13, 14, 15] individuando quelli che meglio si adattano, apportando le dovute modifiche, ad essere applicati a livello architetturale.

Inoltre, nella trattazione dei singoli antipattern, ci si riferirà con il termine risorsa, salvo diversa indicazione, ad entrambe le tipologie hardware e software identificate in precedenza e per ogni antipattern sarà fornita una descrizione, basata sull'osservazione di indici prestazionali, che rappresenta una sorta di *profilo*, inteso come una descrizione sintetica dei tratti essenziali dell'antipattern stesso.

4.2.1 Blob

Si è in presenza di questo antipattern quando all'interno di un'architettura software una risorsa svolge la maggior parte del lavoro relegando le altre a ruoli di supporto minori. Questo genere di situazione è spesso facile da riconoscere in quanto la risorsa "blob"[1], che ingloba in sé la maggior parte delle funzionalità offerte dalla porzione di architettura considerata o che ne assume il quasi totale controllo, presenta un alto indice di utilizzazione, a differenza delle risorse che la circondano e che vengono da essa utilizzate per un corretto smaltimento delle richieste, per le quali si riscontrano invece valori sufficientemente bassi o, nel caso limite, quasi nulli come ad esempio può

[1] In letteratura è conosciuta anche con il nome di *god class* o *god object*; si è preferito il termine *blob* in quanto non riconducibile ad un particolare paradigma

avvenire se una risorsa viene utilizzata solo come contenitore di informazioni senza che questa svolga alcun tipo di elaborazione su di esse.

L'antipattern appena descritto è rappresentato graficamente in figura 4.8, dove i livelli di utilizzazione delle risorse sono rappresentati cromaticamente dal verde al rosso, rispettivamente dai valori più bassi a quelli più alti.

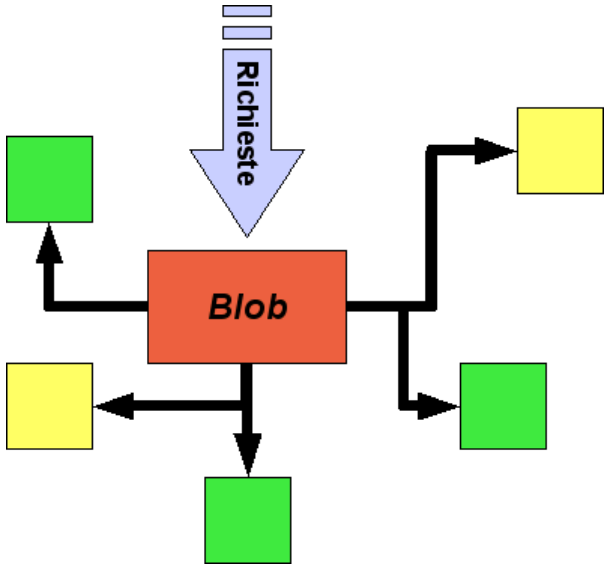


Figura 4.8: *Blob*

In figura 4.9 è invece illustrata una variante dell'antipattern in esame, nella quale la risorsa "blob" non serve monopolisticamente le richieste di servizio ma, contrariamente al caso precedente, contiene la maggior parte delle informazioni necessarie alle altre che la circondano per la gestione delle richieste che ricevono, questa volta, direttamente. Sarà quindi necessario, per le singole risorse, recuperare le informazioni in questione ed eventualmente immagazzinarne delle altre su quella "blob".

L'aspetto che accomuna entrambe le versioni di quest'antipattern è la scarsa distribuzione dell'intelligenza all'interno del sistema.

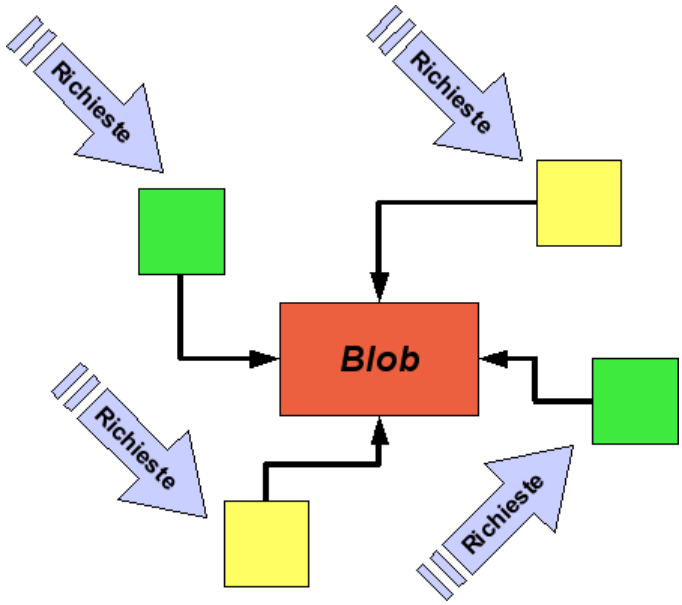


Figura 4.9: *Blob, una variante*

In generale è consigliabile mantenere le informazioni ed i servizi che ne fanno uso nello stesso luogo, di conseguenza ogni risorsa dovrebbe essere in grado, per quanto possibile, di servire le richieste che riceve minimizzando l'interazione con le altre, specialmente se queste sono dislocate su nodi remoti; in quest'ultimo caso infatti, specie se vengono utilizzati canali di comunicazione non molto veloci, le performance del sistema potrebbero degradare drammaticamente.

Applicando tale principio alle due istanze dell'antipattern considerato, si ottiene un maggiore utilizzo delle risorse "periferiche" rispetto a quella "blob", per la quale invece il livello di utilizzazione tende a scendere in funzione di una corretta distribuzione dell'intelligenza o delle informazioni all'interno dell'architettura software; quanto appena descritto è illustrato graficamente nelle figure 4.10 e 4.11.

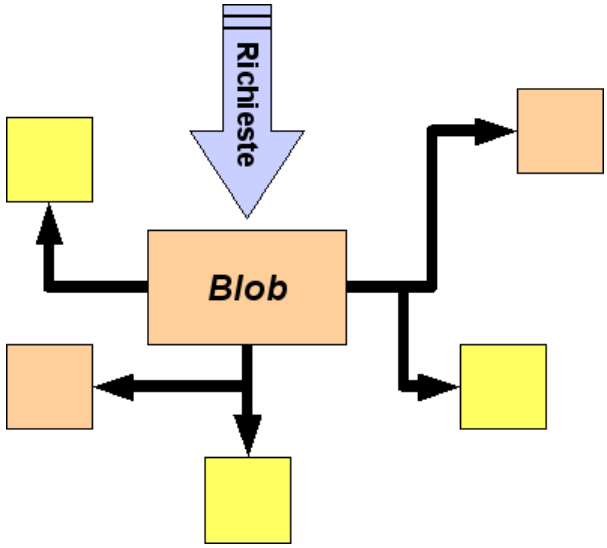


Figura 4.10: *Blob* - ristrutturato

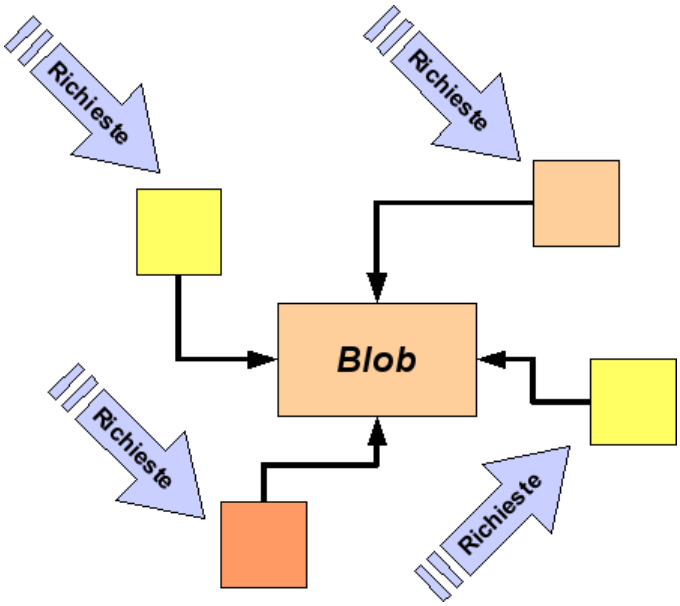


Figura 4.11: *Blob, una variante* - ristrutturato

Nella tabella 4.1 sono riportati gli aspetti caratterizzanti in termini di tempo medio di risposta, throughput ed utilizzazione, dell'antipattern appena descritto.

Tempo medio di Risposta	Throughput	Utilizzazione
alto	*basso*	*alto* per la risorsa "blob" e *medio-basso* per le altre che la circondano

Tabella 4.1: *Profilo Blob*

4.2.2 Unbalanced Processing: "Pipe and Filter" Architecture

Identificando all'interno del sistema considerato un'insieme di risorse del tipo in figura 4.12, si può osservare che il suo throughput è determinato da quello della risorsa, in esso contenuta, più "lenta" nel servire le richieste ricevute. L'antipattern in esame si focalizza proprio su quest'aspetto nel caso in cui tutte le richieste, che giungono alla porzione di architettura considerata, debbano necessariamente percorrere un determinato path all'interno della stessa e quindi essere servite dalla medesima "catena" di risorse.

La risoluzione di quest'antipattern risulta utile qualora i vincoli non funzionali riguardanti il throughput non fossero rispettati; essa è infatti orientata

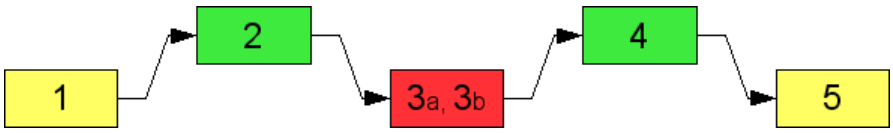


Figura 4.12: *Unbalanced Processing: "Pipe and Filter"*

principalmente ad un aumento dei livelli di quest'ultimo per l'insieme di risorse considerato. È inoltre particolarmente efficace quando si è in presenza di una catena di risorse all'interno della quale gli indici di utilizzazione variano da valori molto bassi a valori molto alti; nel qual caso, l'eliminazione dell'antipattern dovrebbe portare ad un bilanciamento di tali livelli per le risorse considerate.

La soluzione a questo problema, illustrata in figura 4.13, si basa sull'approccio "pipe-and-filter" che consiste nel dividere le risorse maggiormente utilizzate creando delle unità di elaborazione, chiamate *filter*, tra loro indipendenti e connesse, attraverso i *pipe*, in modo tale che possano operare in maniera concorrente o, preferibilmente, parallela. Affinché tale soluzione risulti efficace, sarà necessario tener conto anche dello stato delle risorse hardware che saranno coinvolte dalla ristrutturazione architetturale.

L'applicazione ottimale della soluzione appena esposta prevede la possibilità di agire sull'implementazione della risorsa oggetto di ristrutturazione; qualora questo non fosse possibile, ad esempio perché non conveniente in termini economici oppure nel caso di componenti COTS per le quali non si dispone del codice sorgente, potrebbe essere comunque possibile trovare una soluzione alternativa riconfigurando opportunamente il sistema in modo tale

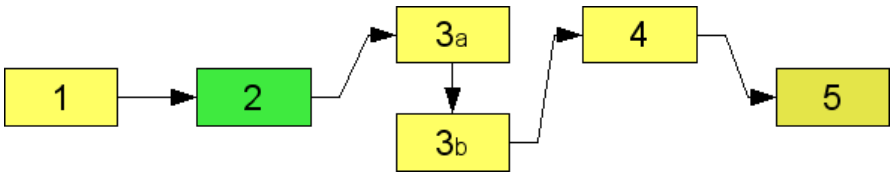


Figura 4.13: *Unbalanced Processing: "Pipe and Filter"* - ristrutturato

che, ad esempio, i servizi offerti da una data risorsa software vengano invocati ognuno su una copia distinta di quest'ultima, evitando quindi di utilizzare tutti i servizi offerti dalla stessa.

In tabella 4.2 è riportato il profilo dell'antipattern *Unbalanced Processing: "Pipe and Filter" Architecture* descritto.

Tempo medio di Risposta	Throughput	Utilizzazione
non caratterizzante	*basso*	*alto* per alcune risorse

Tabella 4.2: *Profilo Unbalanced Processing: "Pipe and Filter" Architecture*

4.2.3 Unbalanced Processing: Extensive processing

Processare le richieste in maniera concorrente offre diversi vantaggi in termini prestazionali e di scalabilità del sistema; queste potenzialità vengono però vanificate se per servire una data richiesta si deve attendere il completamento di altre più onerose, in termini di tempo di servizio, rispetto alla prima. L'Extensive Processing si presenta quando una certa tipologia di richieste tende a monopolizzare l'uso di una particolare risorsa, o di un'insieme di risorse, a scapito del tempo medio di risposta per le altre tipologie di richieste e del throughput dell'intero sistema.

In questo caso, a differenza del precedente "pipe-and-filter", è possibile discriminare diverse tipologie di richieste, all'interno del sistema, che si differenziano tra loro nell'uso più o meno intenso di una o più risorse software. Nell'esempio in figura 4.14 entrambe le tipologie di richieste, identificate come

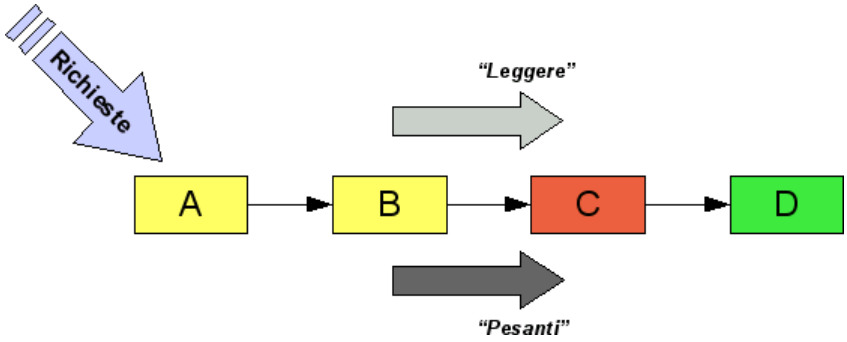


Figura 4.14: *Unbalanced Processing: Extensive processing*

leggere e pesanti, pur utilizzando la stessa risorsa C, la occupano in maniera differente.

L'idea che guida la risoluzione del problema è la realizzazione di percorsi alternativi all'interno del sistema, come rappresentato in figura 4.15, ed in particolare di un "fast path" per le richieste che non impegnano eccessivamente le risorse, o per le quali si è particolarmente interessati ad un rapido servizio.

Potrebbe infatti accadere che solo una piccola parte delle richieste totali che giungono al sistema siano particolarmente onerose da servire, ma che queste influenzino in maniera eccessivamente negativa le performance dell'intero sistema del caso medio; realizzando una sorta di routing all'interno dello stesso, in modo tale da indirizzare le richieste "più leggere" verso un percorso preferenziale, sarà possibile realizzare una loro elaborazione di tipo concorrente e/o parallela evitando che le prime si accodino, in attesa di servizio, su risorse particolarmente impegnate nella gestione di richieste per le quali non si hanno particolari esigenze in termini di tempi di risposta. Un esempio tipico di tale situazione è rappresentato da sistemi che gestiscono richieste

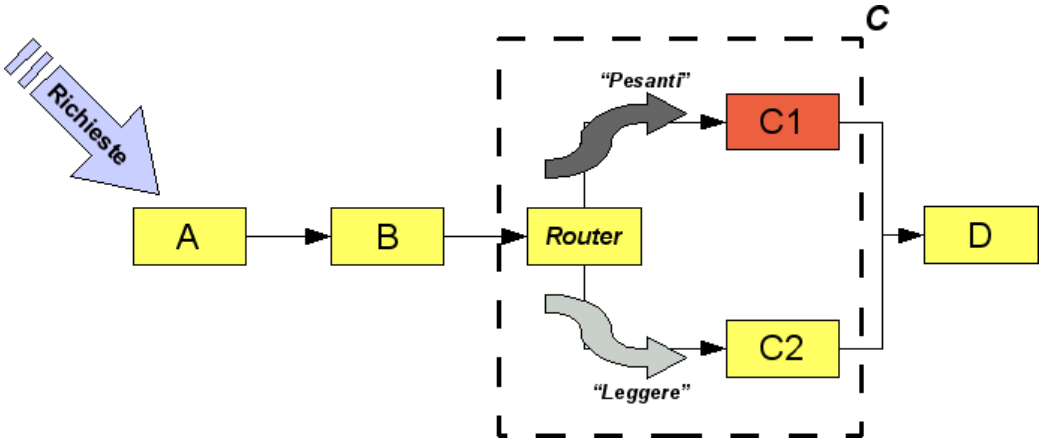


Figura 4.15: *Unbalanced Processing: Extensive processing* - ristrutturato

sia di tipo real-time che batch: per le prime si avranno in genere requisiti più stringenti rispetto a quelli eventualmente indicati per la seconda tipologia.

Anche in questo caso, come per il precedente antipattern trattato e affinché tale soluzione risulti efficace, sarà necessario tener conto anche dello stato delle risorse hardware che saranno coinvolte dalla ristrutturazione architetturale.

Gli effetti positivi di tale ristrutturazione, rappresentati da un tempo medio di risposta più basso ed un throughput maggiore rispetto ai valori ottenuti per l'architettura originale, risulteranno più marcati per la tipologia di richieste considerate privilegiate; mentre gli effetti globali sul sistema saranno essenzialmente legati alla percentuale di richieste di questo tipo rispetto al totale di quelle servite.

Nella tabella 4.3 è riportata una breve descrizione dei tratti caratterizzanti dell'antipattern appena trattato.

Tempo medio di Risposta	Throughput	Utilizzazione
alto	*basso*	*alto* per alcune risorse*: i momenti di inattività (stato idle) per alcune delle risorse saranno rari a causa dell'accumularsi in coda delle richieste durante il servizio di quelle più onerose*

Tabella 4.3: *Profilo Unbalanced Processing: Extensive processing*

4.2.4 Circuitous Teasure Hunt

Quest'antipattern è caratterizzato dal fatto che, per soddisfare le singole richieste, una particolare risorsa software deve recuperare informazioni da una seconda risorsa ed utilizzarle su una terza e così via, fino ad ottenere l'informazione necessaria, come schematizzato in figura 4.16; inoltre, qualora più risorse fossero disponibili su nodi remoti, l'effetto negativo sulle performance del sistema sarebbe amplificato. Tale antipattern può essere considerato come concettualmente opposto rispetto al "blob": in questo caso, infatti, siamo in presenza di un'eccessiva distribuzione dell'informazione del sistema, in contrapposizione all'eccessiva centralizzazione della stessa descritta in precedenza.

Una possibile soluzione consiste nel riorganizzare la distribuzione della conoscenza all'interno del sistema cercando di limitare le "scatole" all'interno delle quali è necessario recuperare sequenzialmente le informazioni necessarie; è doveroso però osservare che ottimizzare il sistema per un determinato scenario potrebbe penalizzare le performance di altri.

Una variante dell'antipattern in esame si presenta invece quando una richiesta fatta ad una risorsa ne scatena una serie in cascata, tra questa ed altre

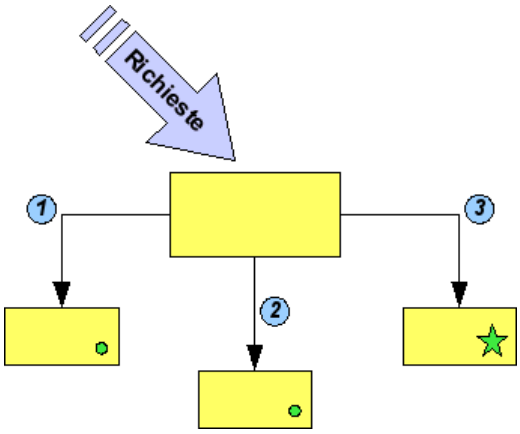


Figura 4.16: *Circuitous Teasure Hunt*

risorse software all'interno del sistema, fino ad ottenere l'informazione necessaria per soddisfare la richiesta iniziale; come rappresentato graficamente in figura 4.17.

Per tale variante considerata potrebbe invece essere utile creare un percorso per raggiungere direttamente, o quantomeno seguendo un percorso più breve, l'informazione necessaria al soddisfacimento delle richieste.

In tabella 4.4 è riportato il profilo individuato per l'antipattern in esame.

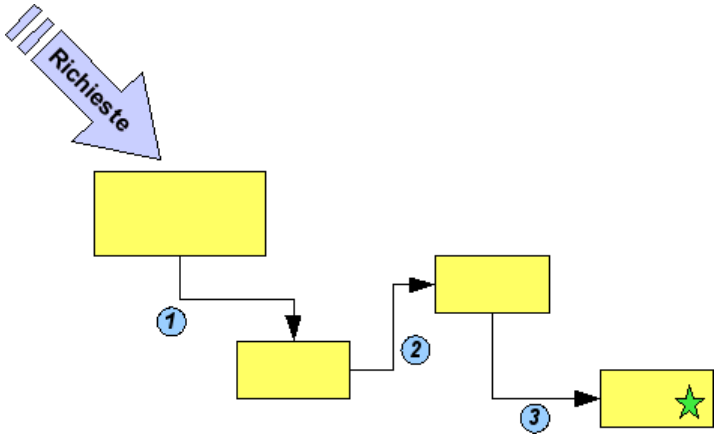


Figura 4.17: *Circuitous Teasure Hunt - una variante*

Tempo medio di Risposta	Throughput	Utilizzazione
alto	*basso*	*non caratterizzante*

Tabella 4.4: *Profilo Circuitous Teasure Hunt*

4.3 LQN

Come illustrato in precedenza, l'approccio descritto in questo lavoro di tesi prescinde dal formalismo utilizzato nella modellazione software, è stato però necessario sceglierne uno per poter realizzare il caso di studio, esposto nel capitolo 6, e per fare esperimenti di vario genere. Si è scelto di adottare le Layered Queued Network essenzialmente per la qualità della documentazione disponibile [17, 18], del solver, gradevolmente efficiente, ed in generale della suite di tool a disposizione che, ad esempio, si è rivelata molto utile nella generazione della rappresentazione grafica dei modelli LQN utilizzati e nel loro studio.

Inoltre nel formato dei file, previsto per la definizione dei modelli LQN, è particolarmente semplice spostare un servizio da una risorsa all'altra considerando che le "chiamate" si riferiscono direttamente ai primi, che dovranno di conseguenza avere un nome univoco all'interno del modello, senza alcun riferimento alle risorse che li contengono; un discorso analogo può essere fatto per la variazione dei valori di molteplicità ed in generale per la parametrizzazione delle componenti che formano il modello. Inoltre la maggior parte dei tool a disposizione supporta il formato XML, sia per la definizione del modello che per l'output dell'analisi: la disponibilità di informazioni ben strutturate risulta particolarmente utile nella realizzazione di automatismi

che siano in grado di interpretarle ed eventualmente di operare delle modifiche su di esse.

Nell'uso di tale formalismo sono state considerate le seguenti corrispondenze tra esso e l'approccio descritto in questa tesi:

- *sistema.* Si riferisce al modello LQN nella sua interezza e rappresenta l'architettura software;
- *sotto-sistema.* Insieme di risorse definito, a seconda delle due tipologie illustrate in precedenza, in base alla loro natura o ai servizi da esse offerti. A seconda dei casi, i sotto-sistemi identificati all'interno di un'architettura possono intersecarsi tra loro così come pure, per sistemi non particolarmente complessi, un sotto-sistema può coincidere con il sistema stesso;
- *risorsa software*, il livello di dettaglio più basso previsto nell'approccio, è rappresentata nei modelli LQN dal *task* che conterrà a sua volta varie *entry* che esprimono i servizi offerti dalla risorsa considerata;
- *risorsa hardware.* E' possibile identificare due tipologie di risorse hardware: una è rappresentata dai processori, utilizzati dai task presenti per le eventuali elaborazioni, mentre l'altra dalla modellazione, attraverso l'uso dei task, di device fisici come, ad esempio, supporti di memorizzazione.

Per quanto riguarda la clonazione delle risorse, prevista nell'approccio, in LQN si è scelto di implementarla variando il valore di molteplicità delle stesse.

La differenza tra questo e la replica vera e propria di una risorsa consiste che nella soluzione utilizzata, conosciuta con il nome di multi-server, tutte le copie di una data risorsa servono la stessa coda di attesa mentre nel secondo caso, di più difficile gestione rispetto al precedente, ognuna di esse servirà una propria coda.

La presenza di un'unica coda di attesa per tutte le copie della risorsa può essere interpretata come la modellazione di un load-balancer estremamente "light", cioè con dei tempi di servizio trascurabili, che distribuisce uniformemente il carico di lavoro ad un dato insieme di risorse identiche tra loro. Nel caso della replica, invece, sarebbe necessario realizzare un qualche tipo di indirizzamento delle richieste, o quantomeno modificarlo, ad ogni clonazione delle risorse in esame.

Una possibile soluzione per il calcolo del tempo medio di servizio per un generico sotto-sistema, considerando che gli attuali tool per LQN non forniscono tale informazione in maniera automatica non contemplando nativamente la suddivisione di un'architettura in sotto-sistemi, consiste nell'aggiunta di due task fittizi con tempo di servizio nullo: uno contenente una entry per ogni servizio offerto dal sotto-sistema considerato T_{IN} e l'altro con una entry per ogni richiesta effettuata al di fuori dello stesso T_{OUT}. La differenza tra il tempo medio di servizio misurato sul task T_{IN} e quello sul task T_{OUT} e cioè $S_{SubSys} = S_{IN} - S_{OUT}$ rappresenta il tempo medio di servizio calcolato per il sotto-sistema in esame.

Alcuni esempi di modelli LQN sono riportati nella sezione che segue.

4.4 Antipattern modellati come LQN

Scelto il formalismo utilizzato per la modellazione, si sono realizzati dei modelli LQN per ogni antipattern identificato in precedenza, al fine di studiarne le caratteristiche e la bontà delle soluzioni proposte. I sorgenti di tali modelli sono riportati nell'appendice A.

Trattandosi di modelli di performance, sono stati calcolati per ognuno i principali indici di interesse: tempo medio di servizio, throughput e utilizzazione ed in particolare per i task in essi contenuti che, a seconda dei casi, sono stati considerati più rilevanti; le misurazioni sono state eseguite al variare del carico di lavoro, rappresentato in questo caso dal numero delle sorgenti di richieste che alimentano i modelli considerati.

Nelle figure 4.18, 4.20, 4.22 e 4.24 sono rappresentati i modelli appena descritti insieme alla loro versione "refactored", mentre nelle figure 4.19, 4.21, 4.23 e 4.25 sono riportate le curve degli indici di performance opportunamente calcolati per entrambi i casi. Questi ultimi grafici possono riferirsi ad una sola risorsa di particolare interesse, il cui nome è riportato nella descrizione dell'asse delle ordinate, oppure a più di una ed in questo caso nella legenda vengono riportati i nomi delle risorse a cui le curve si riferiscono e tra parentesi la loro molteplicità, che si assume pari a 1 se non indicata.

Gli effetti sulle performance delle ristrutturazioni in esame sono fortemente influenzati dal profilo operazionale impiegato nello studio dei singoli modelli LQN, ma in generale si è cercato, nella definizione degli stessi, di riprodurre quanto descritto precedentemente nella sezione 4.2 di questa tesi, sia topologicamente che per quanto concerne gli aspetti dinamici dei singoli antipattern considerati.

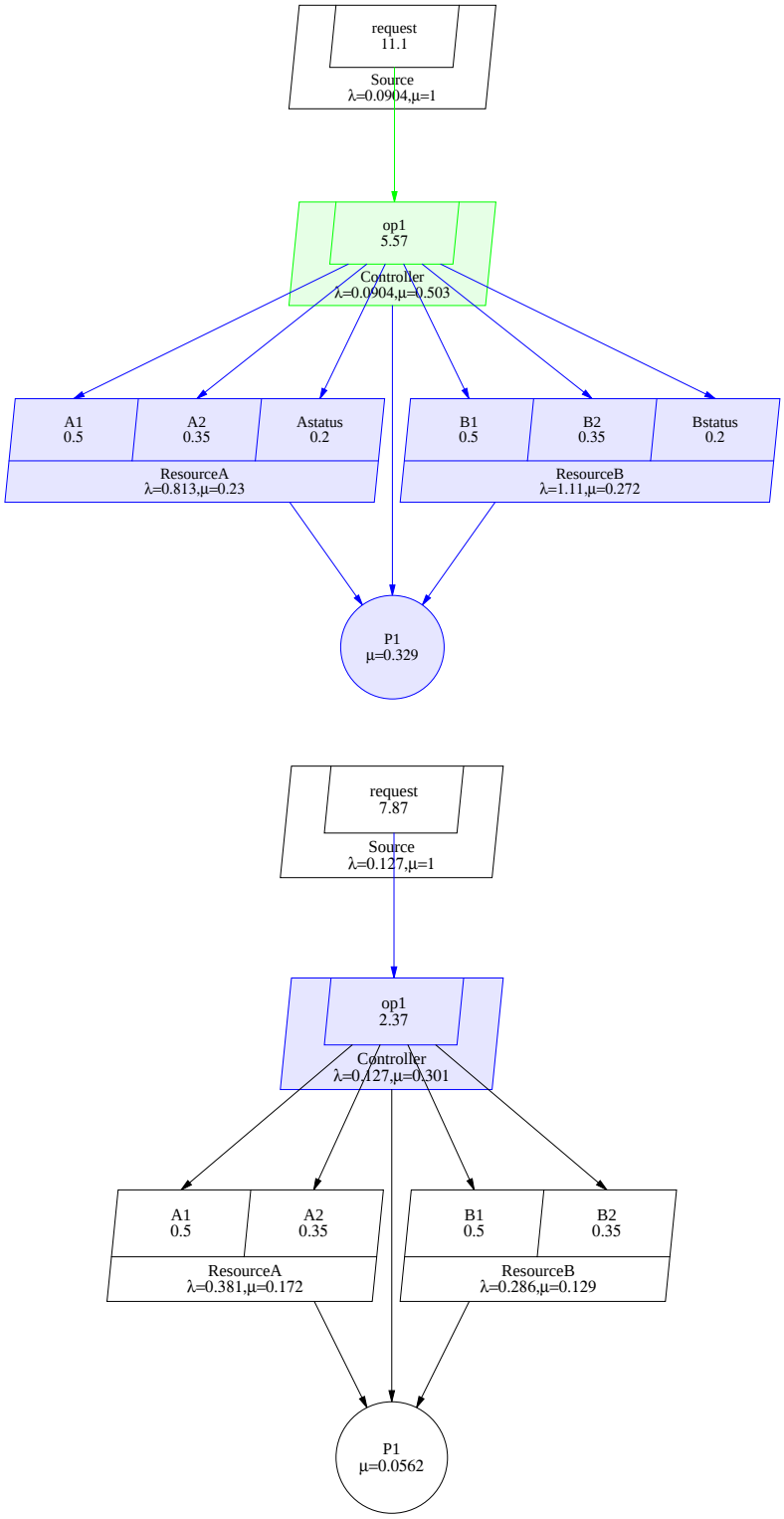


Figura 4.18: *LQN Blob - in alto originale, in basso ristrutturato*

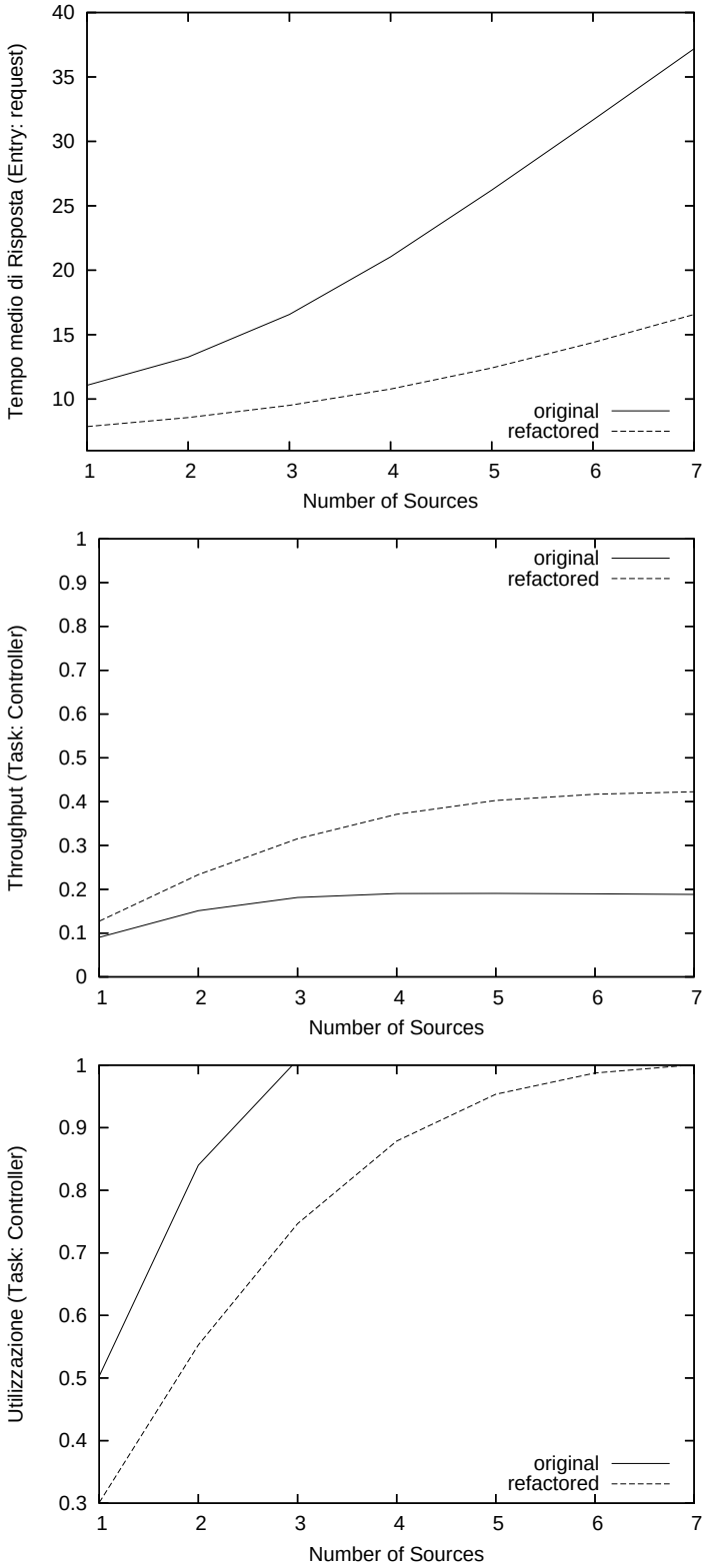


Figura 4.19: *LQN Blob - Indici di interesse*

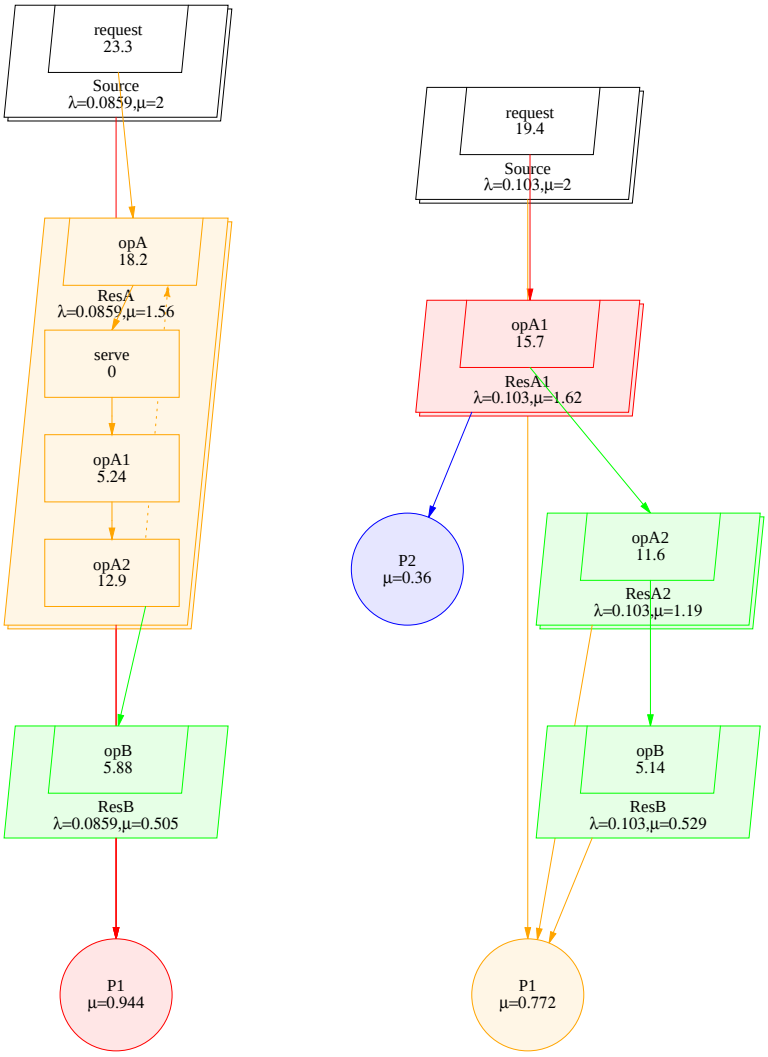


Figura 4.20: *LQN "Pipe and Filter" Architecture - a sinistra il modello originale, a destra quello ristrutturato*

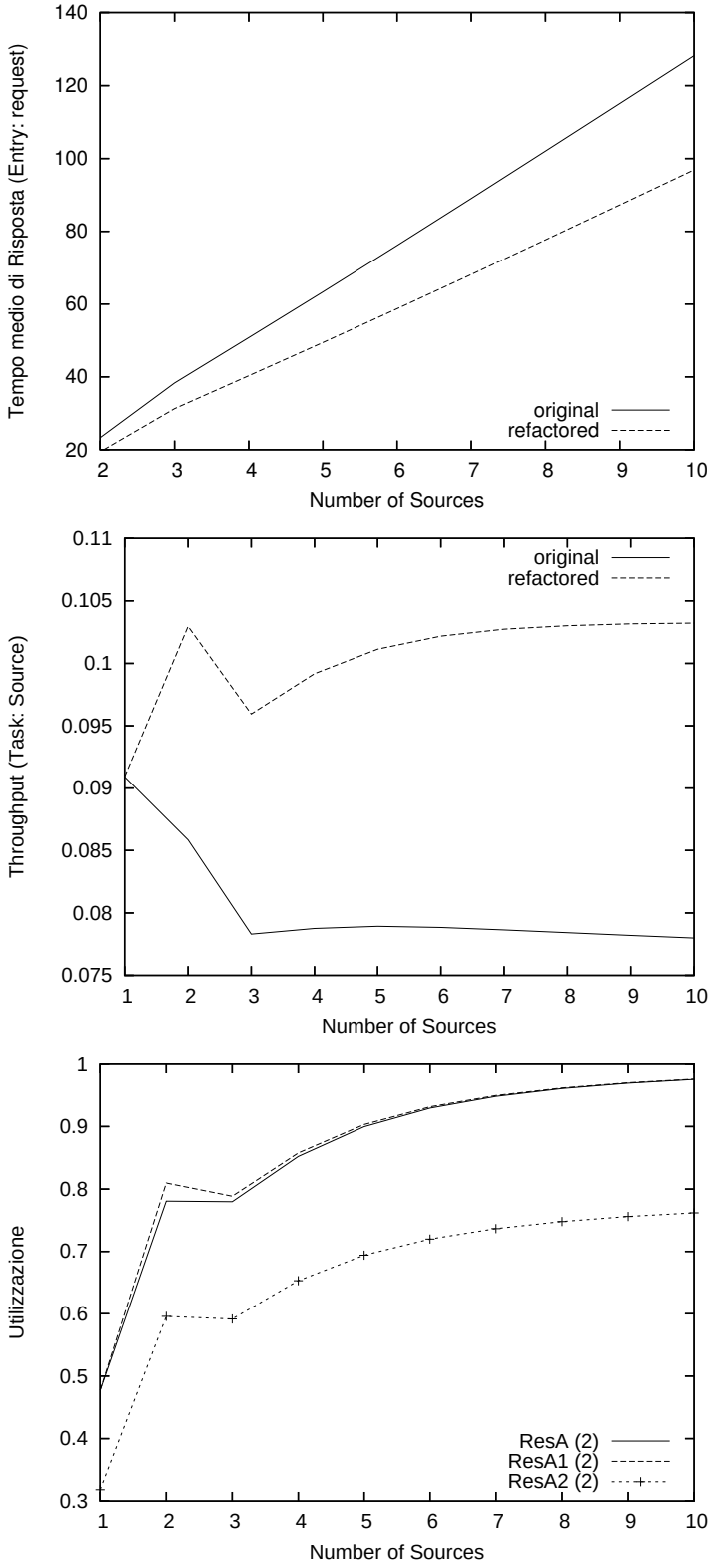


Figura 4.21: *LQN "Pipe and Filter" Architecture - Indici di interesse*

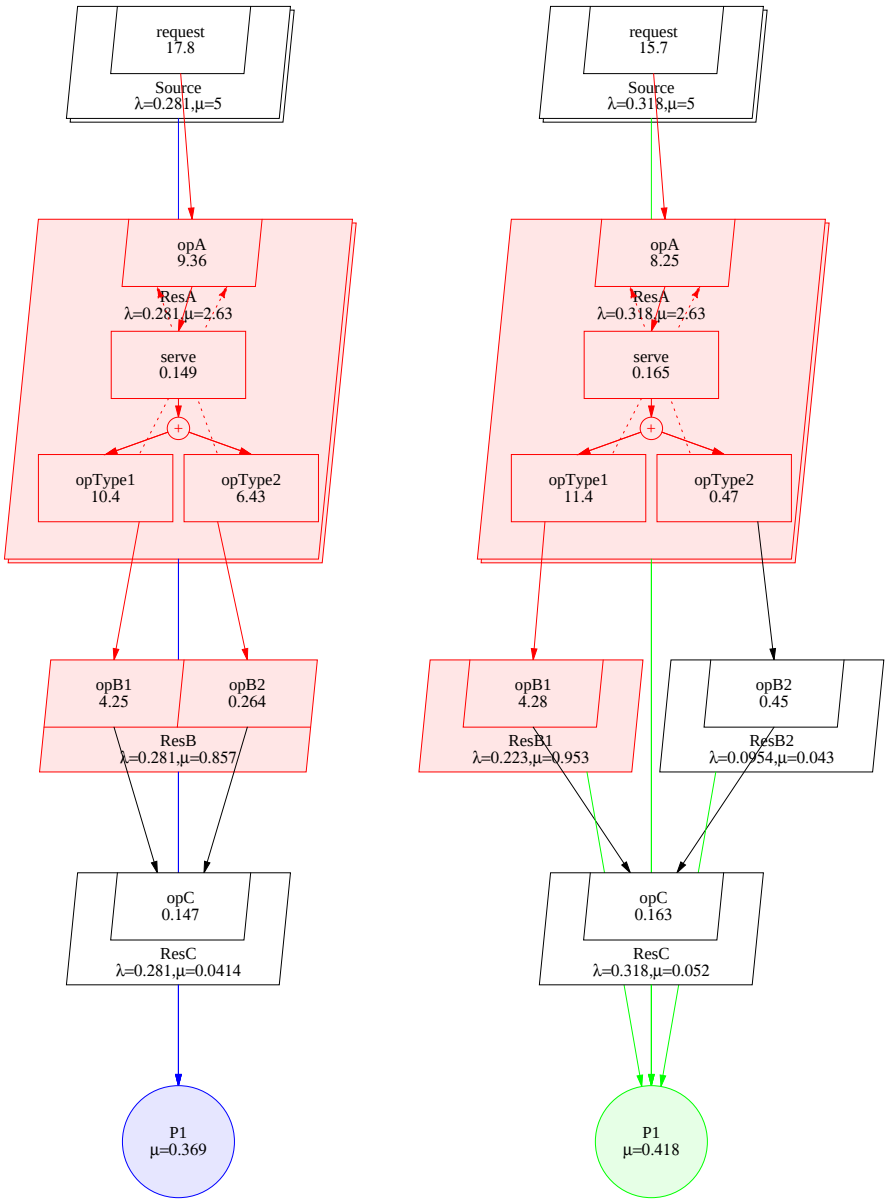


Figura 4.22: *LQN Extensive processing - a sinistra il modello originale, a destra quello ristrutturato*

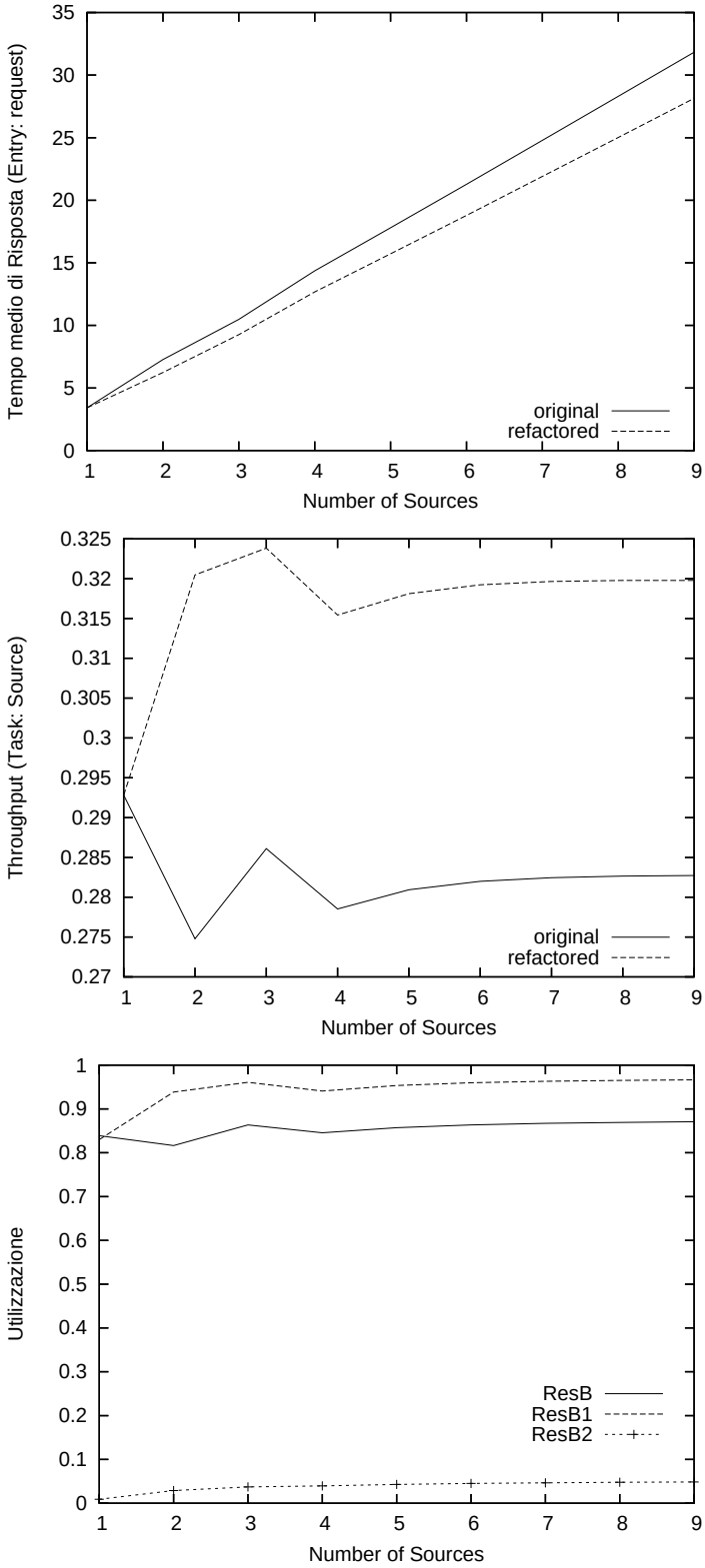


Figura 4.23: *LQN Extensive processing - Indici di interesse*

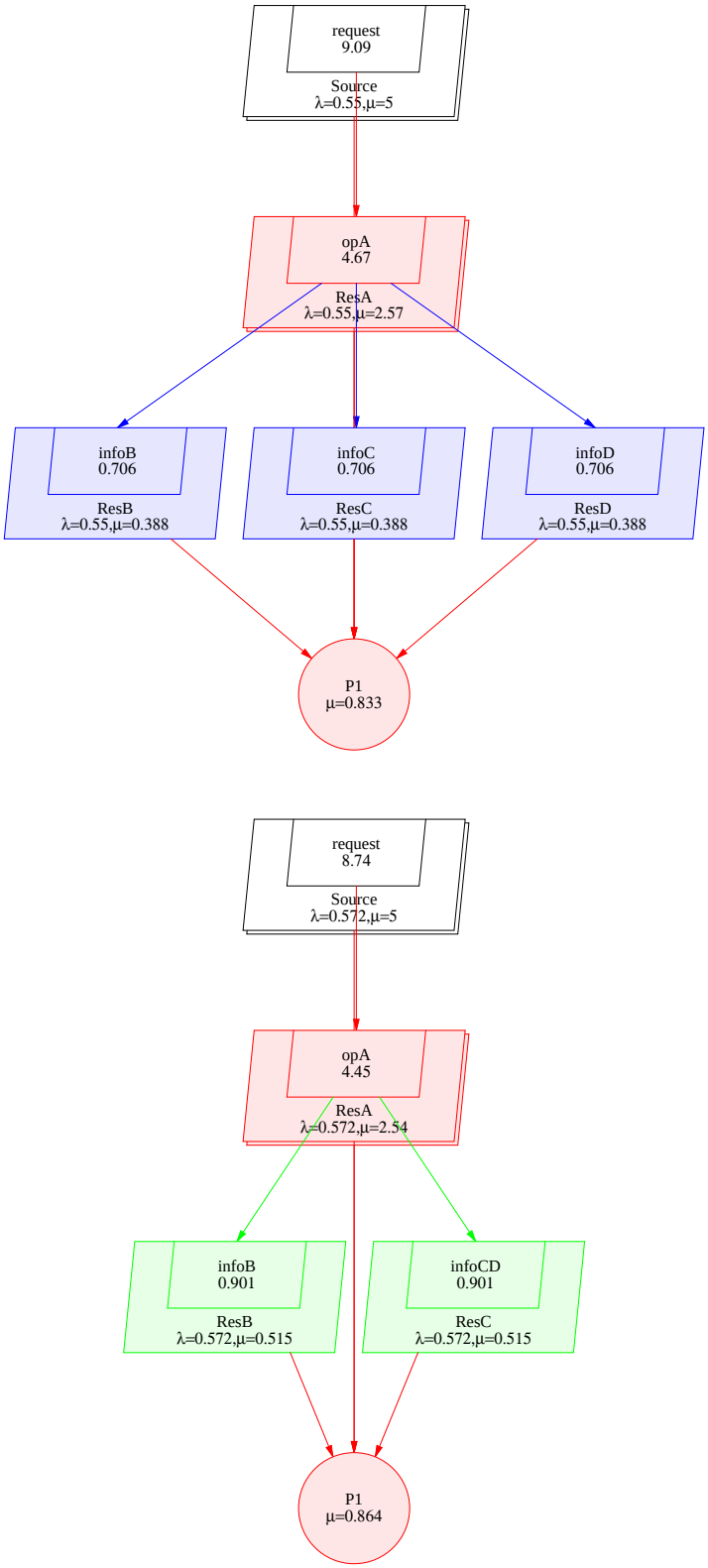


Figura 4.24: *LQN Circuitous Teasure Hunt - a sinistra il modello originale, a destra quello ristrutturato*

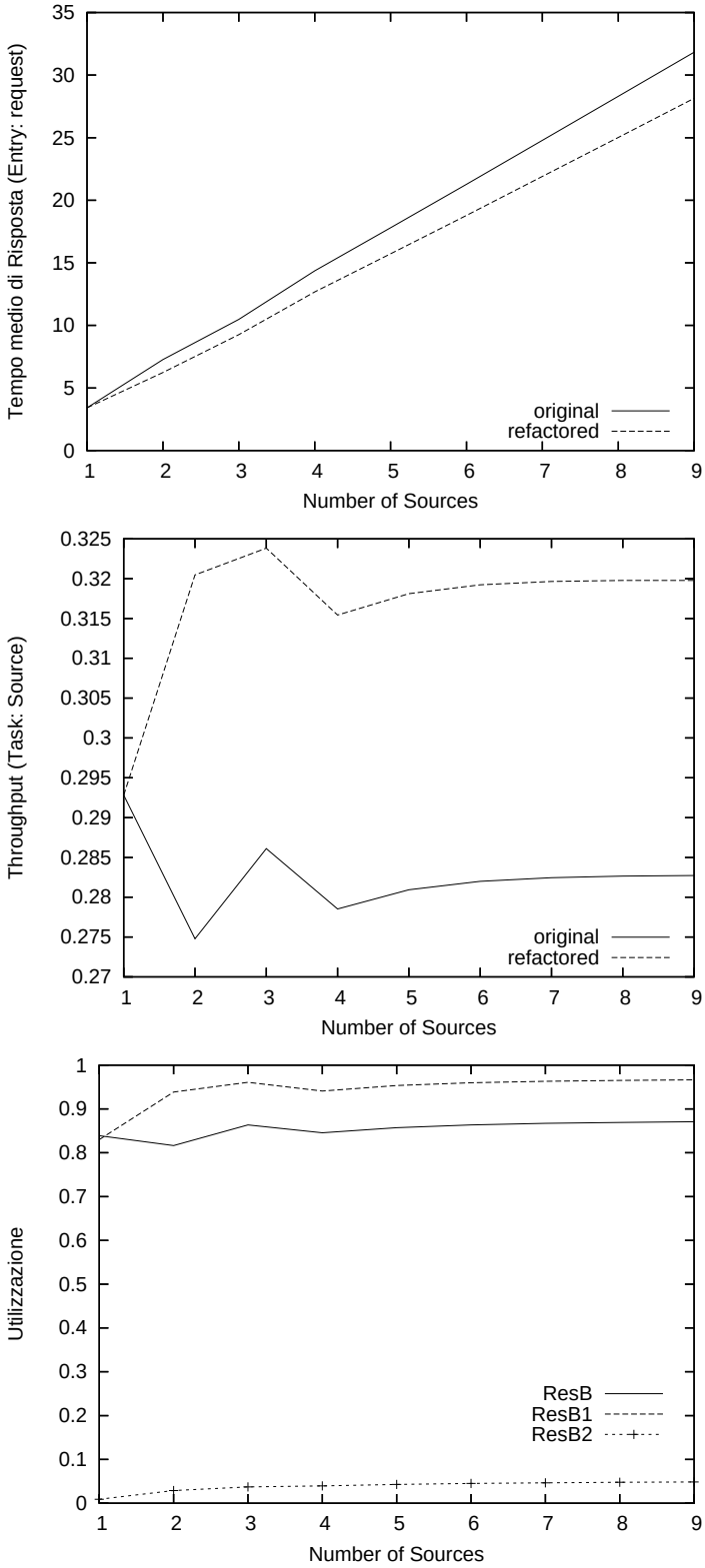


Figura 4.25: *LQN Circuitous Teasure Hunt - Indici di interesse*

Capitolo 5

Studio dell'Automatizzazione del Processo: un Primo Tool di Supporto

Con l'intento di automatizzare l'approccio descritto in questo lavoro di tesi, tenendo comunque conto che si tratta di un processo di estrazione di conoscenza ed in quanto tale difficilmente automatizzabile, si è scelto di sperimentare l'applicazione dei principali suggerimenti offerti dalla tecnica, come la ricerca di antipattern e la clonazione delle risorse software, attraverso l'implementazione di un tool di supporto, denominato *GARFIELD* (Generator of Architectural Feedback through Performance Antipatterns Revealed) e i cui sorgenti sono riportati nell'appendice B.

L'*activity diagram* riportato in figura 5.1 contiene una possibile descrizione algoritmica di alto livello della tecnica descritta nei capitoli precedenti; da tale schematizzazione si evince come il tool risolva sistematicamente il modello di performance risultante da ogni modifica architetturale introdotta nelle varie fasi di raffinamento previste nell'approccio.

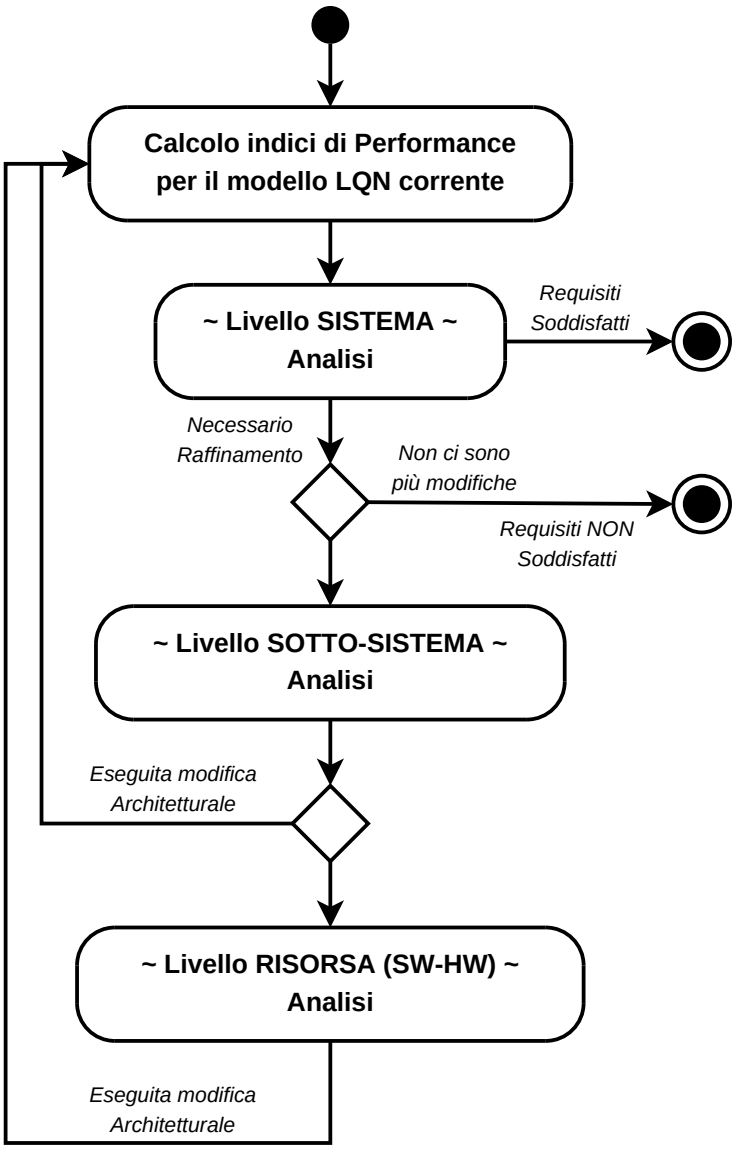


Figura 5.1: *Activity Diagram di alto livello*

Ogni livello di dettaglio ingloba in se sia la componente identificativa che quella propositiva previste dalla metodologia adottata, prevedendo quindi l'utilizzo degli strumenti disponibili a tale livello e l'eventuale applicazione di suggerimenti e strategie risolutive fornite da quest'ultima.

Il tool quindi, dopo aver risolto il modello LQN fornito in ingresso, effettua un'analisi dell'architettura software a livello di sistema e qualora alcuni dei requisiti di performance indicati non fossero soddisfatti, passa al livello di dettaglio immediatamente maggiore, cioè a livello sotto-sistema; se a questo punto la metodologia fornisce una possibile modifica architetturale potenzialmente utile questa viene applicata e gli indici ricalcolati, diversamente sarà necessario ricorrere ad un'analisi a livello risorsa, il massimo livello di dettaglio disponibile. Nel caso in cui l'architettura soddisfi tutti i requisiti prestazionali per essa indicati oppure non fossero più disponibili modifiche architetturali suggerite dall'approccio utilizzato, il programma termina la sua esecuzione.

I singoli macro-passi utilizzati, ognuno dei quali si riferisce ai tre diversi livelli di granularità adottati, vengono descritti nel dettaglio nelle figure 5.2, 5.3, 5.4; dall'unione degli activity diagram in esse contenuti, si ottiene la rappresentazione grafica delle attività svolte dal tool nel tentativo di raggiungere gli obiettivi di performance indicati per una data architettura software.

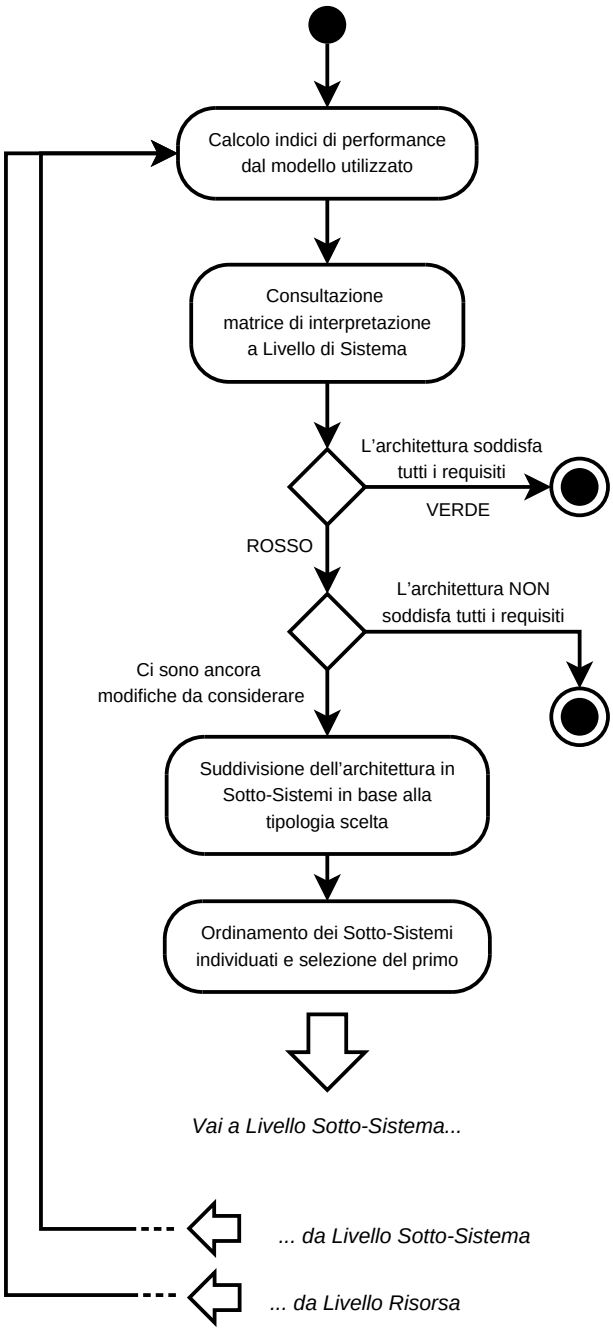


Figura 5.2: *Activity Diagram - Livello Sistema*

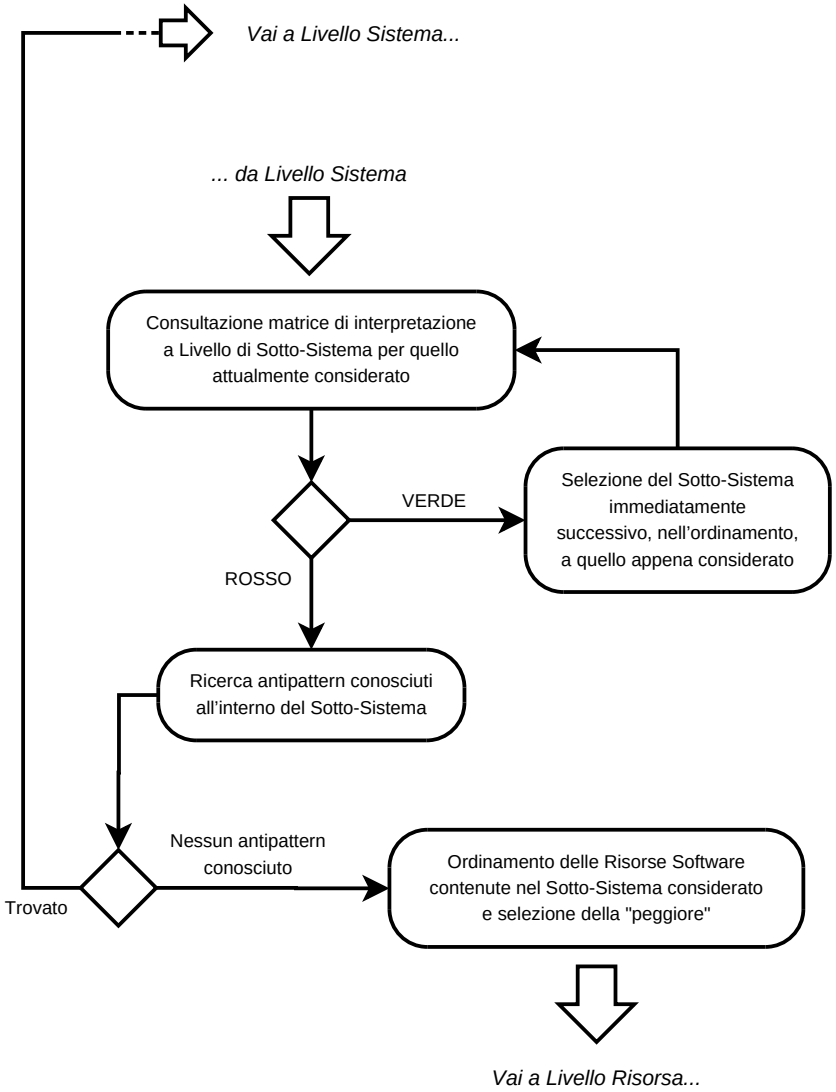


Figura 5.3: *Activity Diagram - Livello Sotto-Sistema*

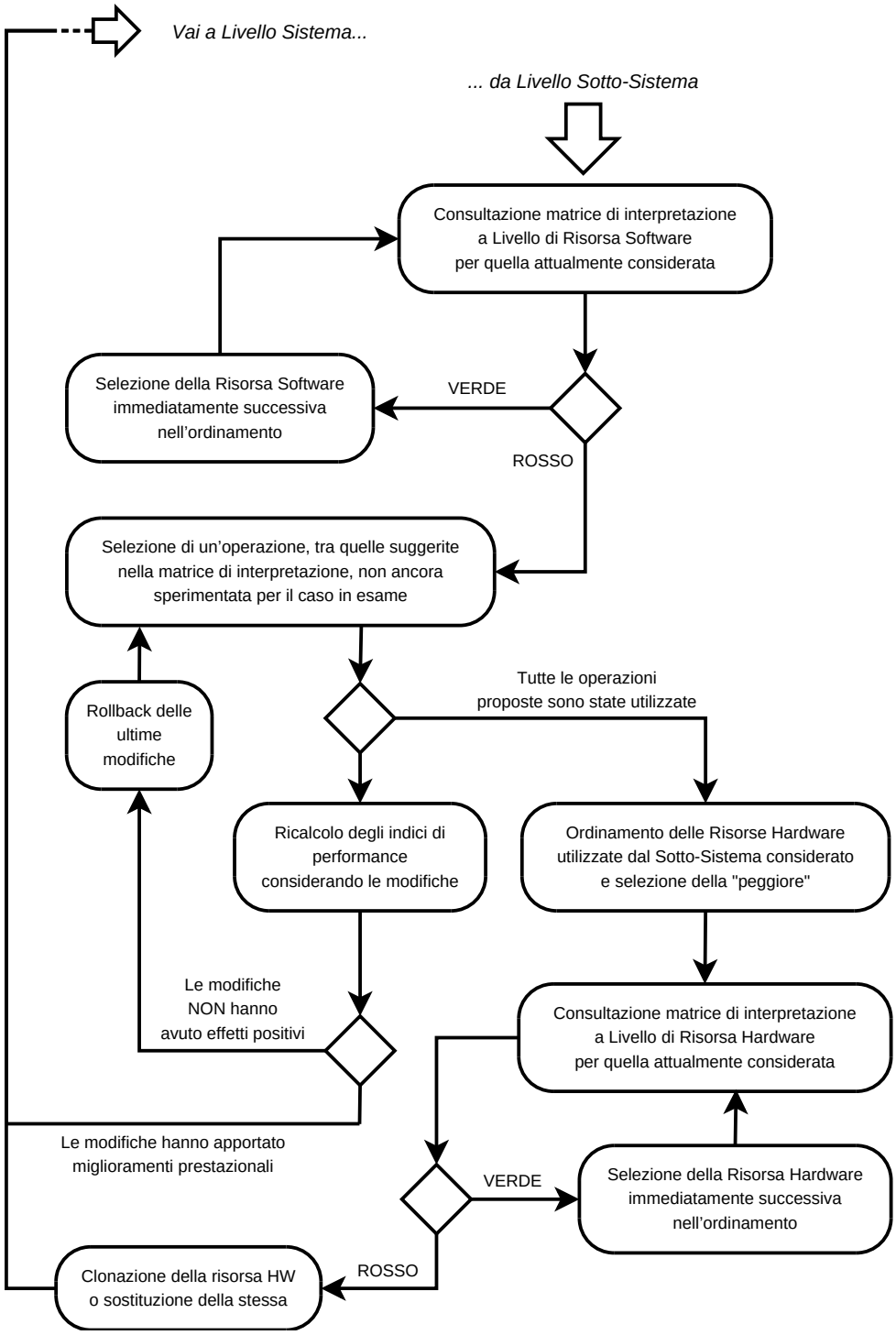


Figura 5.4: *Activity Diagram - Livello Risorsa*

Le rappresentazioni appena descritte si riferiscono in alcuni punti alle matrici d'interpretazione così come definite in precedenza, di conseguenza potrebbe essere necessario revisionarle qualora queste ultime dovessero subire cambiamenti orientati, ad esempio, all'introduzione di nuovi percorsi risolutivi o di raffinamento.

Grazie al supporto del formato XML, offerto dal solver utilizzato per la risoluzione dei modelli LQN, è stato possibile utilizzare il Document Object Model (DOM) per la manipolazione delle informazioni contenute nei file utilizzati, senza la necessità di dover ricorrere alla realizzazione di un parser ad-hoc. Il solver può inoltre aggiungere le informazioni ricavate dalla risoluzione del modello di performance direttamente all'interno del file XML dato in input allo stesso, in questo modo si ha la possibilità di mantenere tutte le informazioni necessarie all'analisi in un'unica struttura dati ben organizzata.

Il tool è scritto in PHP e realizzato per un utilizzo da linea di comando ma, anche grazie alle peculiarità del linguaggio scelto, può essere adattato all'utilizzo via web con poche modifiche.

```
$ php garfield.php Architecture.xml
```

Il programma provvede inizialmente a creare una copia del file, indicato come input, nella directory di lavoro specificata nella configurazione dello stesso e al nome del file di tale copia, contenente quindi l'architettura di partenza, verrà inoltre aggiunto il suffisso *_step0*; in generale, infatti, ad ogni modifica architetturale effettuata il tool provvederà a creare un nuovo file il cui nome conterrà un numero di step progressivo. In questo modo sarà quindi mante-

nuto uno storico delle modifiche apportate, in modo tale da poter valutare gli effetti di ognuna di esse sull'architettura software considerata.

Nel rispetto dell'approccio indicato in questo lavoro di tesi, come si può evincere dalla sua schematizzazione riportata nelle figure precedenti, il tool contiene un'implementazione delle principali tecniche risolutive illustrate nelle varie matrici d'interpretazione. Si è cercato inoltre di dare enfasi alla visione dell'architettura per diversi livelli di dettaglio, così come precedentemente illustrato nel capitolo 4.

Negli activity diagram è inoltre evidenziato come anche a seguito di modifiche che non variano la struttura topologica dell'architettura in esame, come la clonazione di risorse ad esempio, venga comunque rieseguita la ricerca degli antipattern conosciuti in quanto anche una variazione nei livelli di carico delle singole risorse, riconducibile magari ad effetti secondari di una modifica apportata, potrebbe far emergere un antipattern che in precedenza non era ben delineato all'interno dell'architettura stessa.

Nella versione attuale del tool viene richiesto l'intervento dell'utilizzatore per scegliere quale antipattern considerare e quindi risolvere tra quelli conosciuti ed individuati nell'architettura analizzata. Lo strumento è inoltre in grado di valutare se una data modifica ha portato complessivamente miglioramenti nel sistema oppure no: utilizzando la formula $\frac{\Delta Value}{Value_{REQ}} = \frac{Value_{CUR} - Value_{PREV}}{Value_{REQ}}$ (dove $Value_{REQ}$, $Value_{CUR}$ e $Value_{PREV}$ rappresentano rispettivamente il valore richiesto, quello rilevato per il modello attuale e quello per il modello allo step precedente per un dato requisito) vengono attualmente calcolati gli indici utilizzati per la comparazione tra effetti positivi e negativi.

In base alla natura del requisito a cui essi si riferiscono il loro segno viene eventualmente invertito in modo tale da ottenere un valore positivo per variazioni migliorative o negativo qualora non lo fossero[1].

Se le modifiche in esame non dovessero risultare complessivamente vantaggiose in termini prestazionali ma la loro applicazione porta comunque al soddisfacimento dei requisiti definiti per l'architettura, queste saranno giudicate utili dal tool e quindi applicate al sistema software.

Per l'individuazione di eventuali antipattern all'interno dell'architettura analizzata, il software utilizza una serie di profili caratterizzanti definiti, per ogni singolo antipattern conosciuto, sulla base delle informazioni topologiche e relative alle performance del sistema fornite dal risolutore.

Al fine di mostrare le principali funzionalità implementate nel tool, si è scelto di utilizzarlo su una stessa architettura, mostrata in figura 5.5, seguendo due strade distinte: in un caso verrà risolto l'antipattern individuato automaticamente nel sistema mentre nell'altro quest'ultimo verrà ignorato ed il software provvederà ad applicare le altre opzioni implementate, previste dalle opportune matrici di interpretazione.

I requisiti non funzionali per l'architettura considerata sono i seguenti:

- Tempo di servizio per *opType2* al più pari a 5 unità di tempo;
- Throughput dell'intero sistema, quindi misurato sulla sorgente di richieste *Source*, non inferiore a 0.4 richieste servite per unità di tempo.

[1] Le variazioni del tempo medio di risposta non seguono tale criterio, infatti un aumento dello stesso e quindi un valore positivo per l'indice calcolato è di norma non desiderabile, di conseguenza in questo caso il segno viene invertito per uniformare l'indice al criterio appena esposto; le variazioni del throughput seguono invece già intrinsecamente tale andamento di desiderabilità e quindi non è necessario operare modifiche sugli indici calcolati in quest'ambito

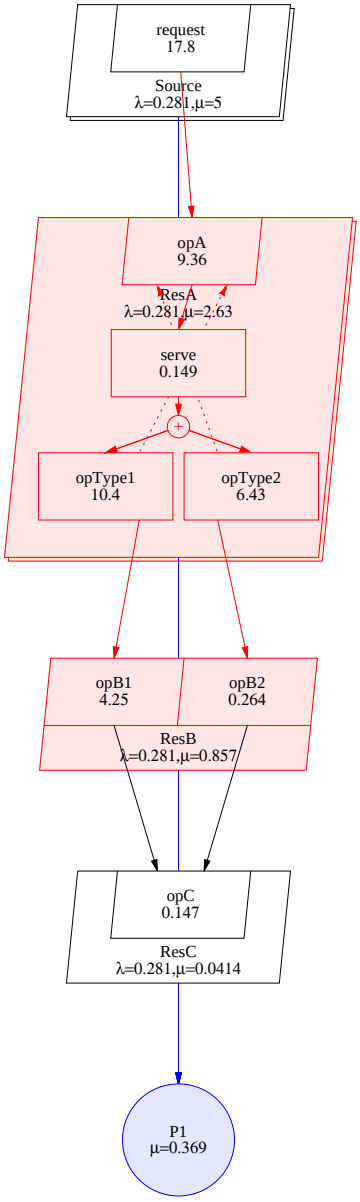


Figura 5.5: *Tool - Step0, modello LQN in input*

Si è scelto inoltre di abilitare, agendo opportunamente sui vari parametri di configurazione disponibili, la modalità verbosa e la conservazione dei file XML contenenti le modifiche ignorate ai fini della realizzazione del modello finale, in quanto giudicate non utili in fase di elaborazione, con l'intento di mostrare graficamente e nel dettaglio le singole azioni e modifiche all'architettura in esame eseguite dal tool. Nel primo caso considerato e per il modello LQN iniziale si ottiene il seguente output:

```
Searching for known antipattern in SubSystem 'sub1'...
  Possible antipatterns in 'sub1' subsystem:
   (0) Extensive Processing
  Selection (-1 to skip): 0
== Some requirements aren't Satisfied...
=== Step 1 created
  Extensive Processing antipattern was removed

Searching for known antipattern in SubSystem 'sub1'...
  No known antipattern found

== Some requirements aren't Satisfied...
=== Step 2 created
Resource ResB2:
  raise multiplicity from 1 to 2
Performance Variations:
  opType2;0.0736036 (service_time)
  Source;0.37191 (throughput)

ALL Requirements are Satisfied!

Creating Summary Graphs (using gnuplot)...
Creating Models Graphs (using lqn2ps)...
```

Al termine dell'elaborazione il tool genera in maniera automatica una rappresentazione grafica dei modelli LQN risultanti dall'esecuzione degli step considerati, riportati per il caso considerato in figura 5.6, e dei diagrammi, in figura 5.7, che illustrano invece i valori assunti dagli indici di interesse, selezionati in base ai requisiti non funzionali definiti per l'archittettura, sui vari

step identificati; in quest'ultima rappresentazione vengono inoltre riportati, utilizzando delle linee verdi tratteggiate, il valore desiderato ed indicato nei requisiti per i vari indici e, nella descrizione dell'asse delle ascisse, quale tra gli step di riferimento contiene il modello finale. Potrebbe infatti capitare, nel caso in cui si scelga di visualizzare graficamente anche gli step giudicati non utili, che il modello finale non coincida con l'ultimo step creato dal tool.

Indicando invece al tool, in fase di analisi del modello, di non risolvere l'antipattern individuato, questo percorre una strada differente che porta ad ottenere un modello finale diverso dal precedente ma, almeno in questo caso, comunque rispettoso dei requisiti non funzionali indicati per l'architettura.

```
Searching for known antipattern in SubSystem 'sub1'...
  Possible antipatterns in 'sub1' subsystem:
   (0) Extensive Processing
  Selection (-1 to skip): -1
Skipping...

== Some requirements aren't Satisfied...
=== Step 1 created
Resource ResA:
  raise multiplicity from 3 to 4
Performance Variations:
  opType2;0.449388 (service_time)
  Source;0.01411 (throughput)
=== Rollback: The last Step was deleted!

== Some requirements aren't Satisfied...
=== Step 2 created
Resource ResB:
  raise multiplicity from 1 to 2
Performance Variations:
  opType2;-0.784648 (service_time)
  Source;0.4543125 (throughput)

ALL Requirements are Satisfied!

Creating Summary Graphs (using gnuplot)...
Creating Models Graphs (using lqn2ps)...
```

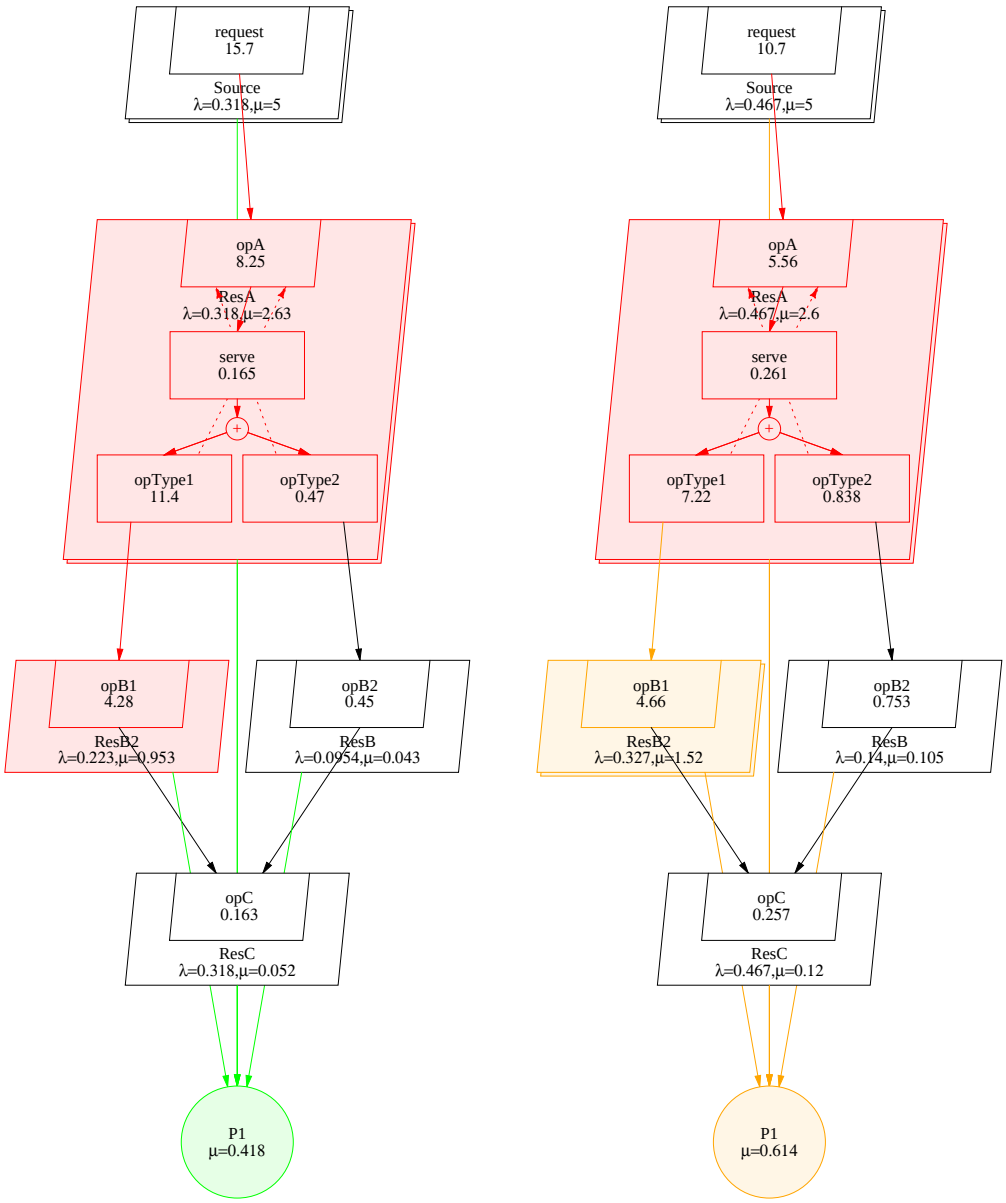


Figura 5.6: *Tool - Step1 (sx) e Step2 (dx) con risoluzione antipattern*

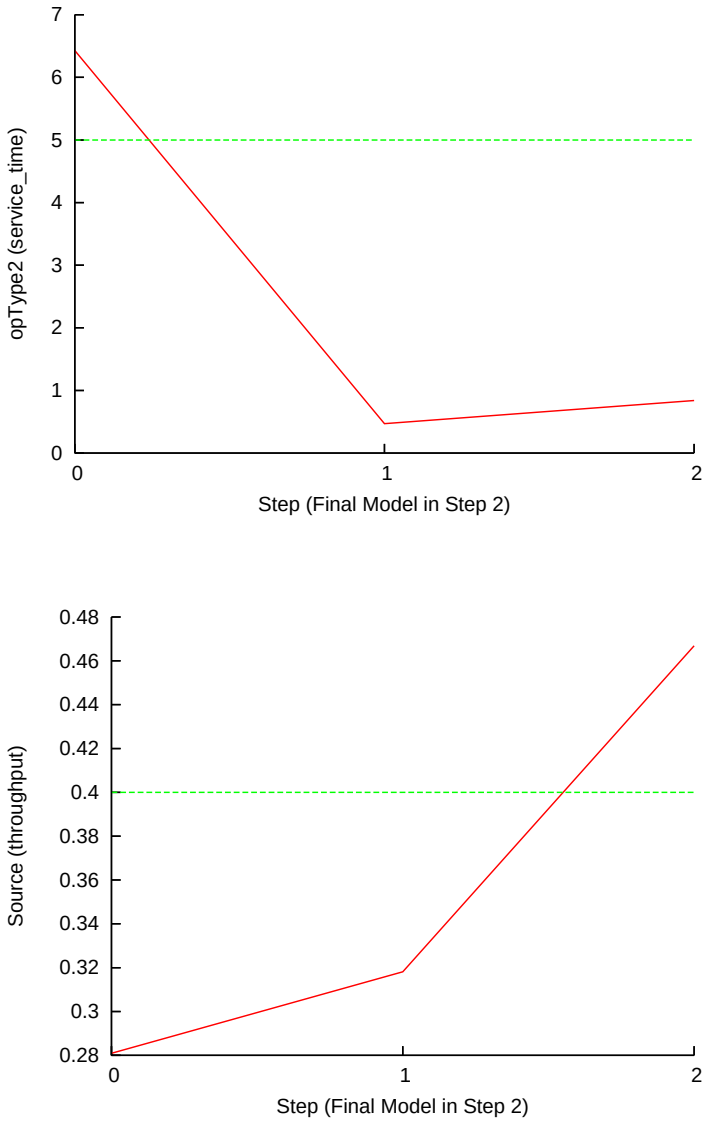


Figura 5.7: *Tool - Indici di interesse per il caso con risoluzione antipattern*

In figura 5.8 sono riportate le rappresentazioni dei modelli LQN per gli step individuati dal tool ma, come si può notare dall'output di quest'ultimo, in questo secondo caso lo *Step1* non concorre alla definizione del modello finale in quanto ad un leggero miglioramento del throughput, misurato sul task *Source* così come indicato nei requisiti e riassunto dal software nei messaggi in output, si affianca un importante aumento del tempo di servizio per *Op-Type2* e questo porta di conseguenza il tool ad optare per il *rollback* delle modifiche. In figura 5.9 sono riportati, come per il caso precedente, l'andamento degli indici di interesse, i requisiti non funzionali definiti e quale tra gli step considerati contiene il modello finale.

Nella versione attuale del tool è necessario indicare manualmente l'appartenenza delle risorse ai vari sottosistemi individuabili nell'architettura in esame ed allo stesso modo indicare quali risorse non possono essere clonate per i motivi illustrati nel capitolo 4; in una versione futura del software potrebbe essere automatizzata la suddivisione del sistema attraverso l'analisi del percorso seguito dalle varie richieste all'interno dello stesso individuando opportunamente i punti di ingresso, rappresentati dai servizi offerti, creando in questo modo sottosistemi di tipologia 2. Per l'individuazione delle risorse non clonabili e per la creazione di sottosistemi di tipologia 1 l'automatizzazione potrebbe invece essere più complessa in quanto subentrano nell'analisi aspetti semantici di difficile schematizzazione, spesso legati a specifiche scelte progettuali dei designer.

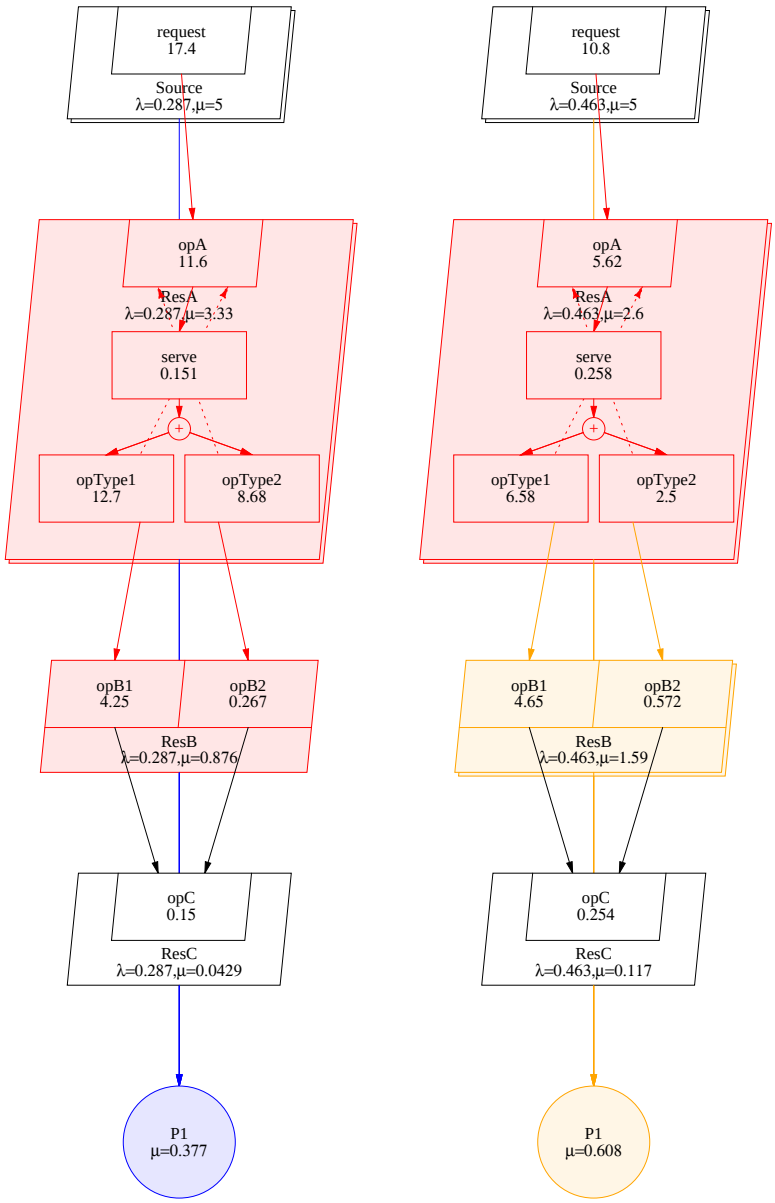


Figura 5.8: *Tool - Step1 (sx) e Step2 (dx), caso con rollback*

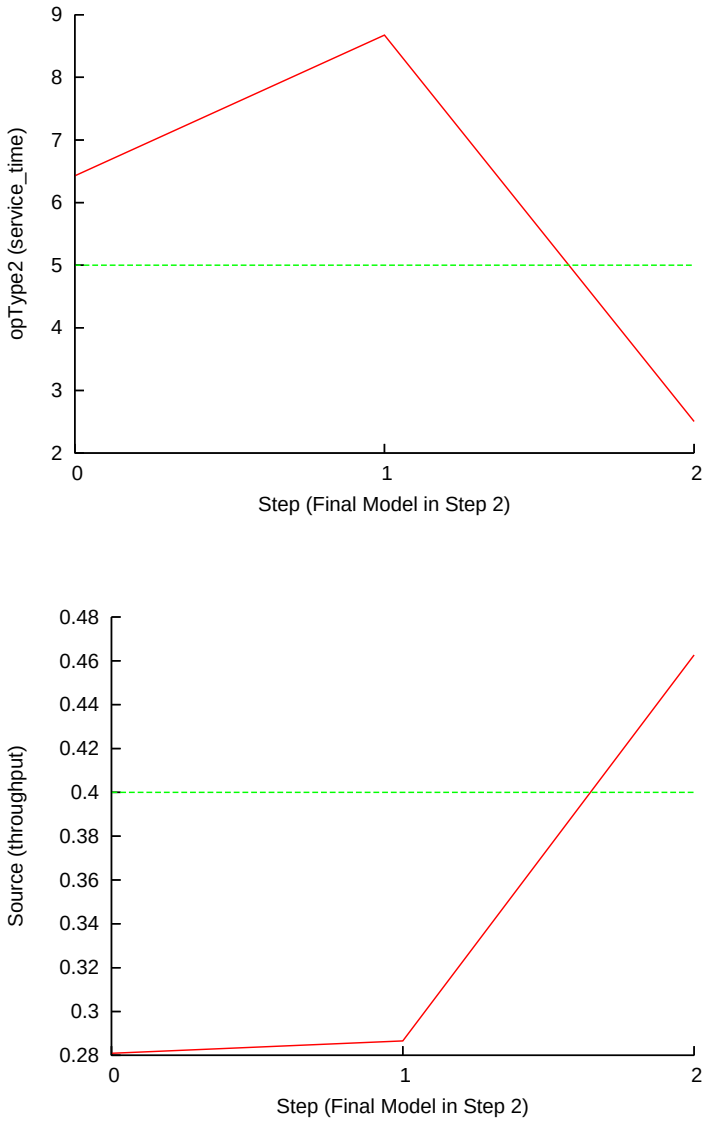


Figura 5.9: *Tool - Indici di interesse per il caso con rollback*

Capitolo 6

Un Caso di Studio: Crazy-Robot

Il modello utilizzato come caso di studio rappresenta un'architettura software utilizzata per un piccolo robot in grado di interagire con l'ambiente che lo circonda e di imparare dalle esperienze passate, in particolare per riuscire a distinguere situazioni potenzialmente pericolose da altre che invece non lo sono.

Il robot in questione è composto da tre parti fondamentali:

- *apparato sensoriale* - utilizzando il quale è in grado di avere una percezione di tipo visivo dell'ambiente in cui si muove e della temperatura di quest'ultimo, un caso particolare è rappresentato dal supporto wireless che gli permette di comunicare con i robot "amici";

- *servomeccanismi* - che permettono al robot di spostarsi in diverse direzioni e di interagire con gli oggetti che incontra nella sua esplorazione;

- *unità di elaborazione centrale* - che comprende la parte intelligente e la parte di reazione agli eventi.

L'attività del robot consiste nel muoversi nell'ambiente, acquisire e condividere conoscenza, comunicando con gli altri robot, in modo tale da riuscire sempre meglio ad individuare condizioni di pericolo da evitare per la propria incolumità e per quella degli altri soggetti, umani e non, che con esso possono interagire.

Al verificarsi di un evento nell'ambiente, questo viene rilevato dall'apparato sensoriale, quindi dai dispositivi che equipaggiano il robot in esame, oppure comunicato dagli altri che con esso collaborano; successivamente l'unità di elaborazione centrale, basandosi sul bagaglio di conoscenze acquisito, stabilisce se si tratta di un evento pericoloso (o potenzialmente tale) oppure di un evento che non desta particolari preoccupazioni ed in quanto tale possa essere analizzato per acquisire nuova conoscenza dell'ambiente all'interno del quale il robot agisce; a tal fine verranno utilizzati i servomeccanismi a disposizione, come l'apparato di gestione degli arti superiori e quello di locomozione, per interagire, ad esempio, con degli oggetti rilevati sul terreno. Nel caso invece di evento pericoloso, il robot dovrà essere in grado di reagire prontamente, fare le opportune valutazioni ed arrestarsi.

Quanto appena descritto è rappresentato graficamente nei *sequence diagram* riportati nelle figure 6.1 e 6.2, dove vengono illustrate le interazioni tra le principali risorse presenti nell'architettura software e coinvolte rispettivamente nella gestione di eventi pericolosi e non.

In figura 6.3, riportata nella sezione successiva, è rappresentato il modello LQN utilizzato per l'analisi del software realizzato (o semplicemente progettato) per la macchina in esame. I task *Environment* e *OtherROBOTS* vengono utilizzati solo come fonte di richieste di servizio quindi sono da

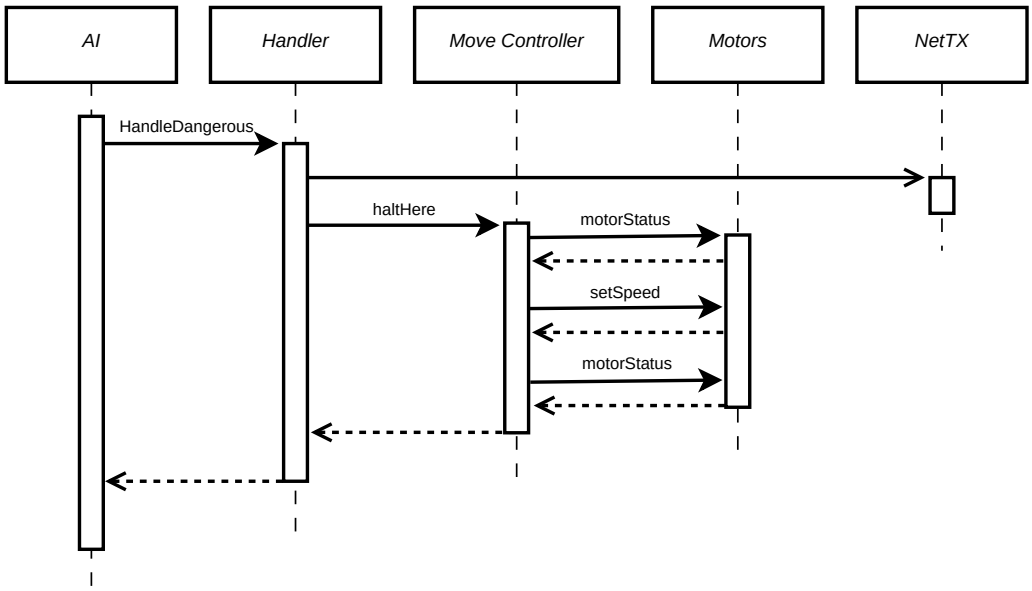


Figura 6.1: *Sequence Diagram, gestione evento pericoloso*

considerarsi esterni al sistema software che si andrà ad analizzare. Inoltre, volendo utilizzare la suddivisione fatta in precedenza, i task possono essere così raggruppati:

- *apparato sensoriale*: Sensors, NetRX, NetTX;
- *servomeccanismi*: Arms, Motors, MoveController;
- *unità di elaborazione centrale*: StorageMemory, VolatileMemory, AI, Handler.

Considerando che il numero di sensori che equipaggiano il robot è fisso e pari a 2, input visivo e temperatura, e che il numero di robot amici invece può variare da un minimo di 1 ad un massimo di 10, ad esempio per ragioni che possono essere prettamente progettuali e/o implementative, sono stati definiti i seguenti requisiti non funzionali:

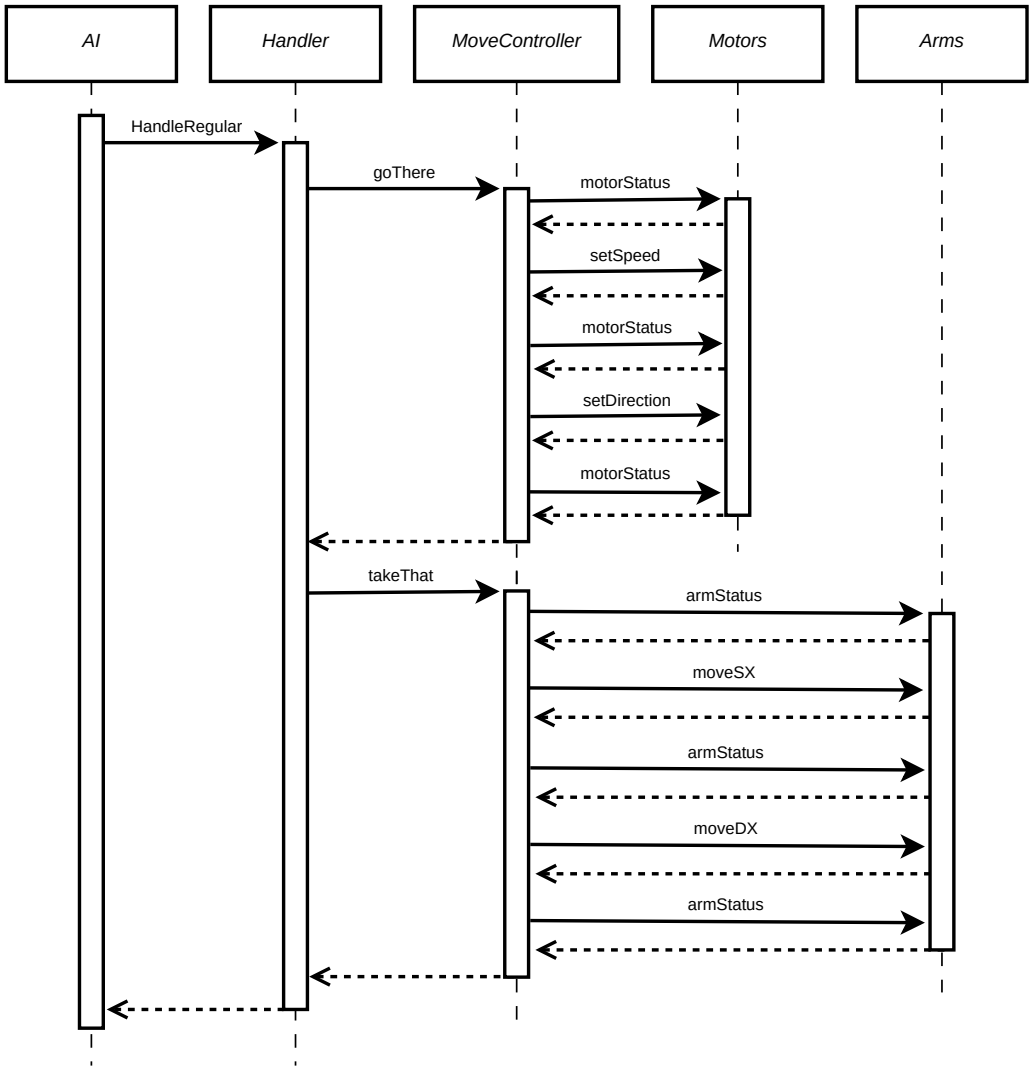


Figura 6.2: *Sequence Diagram, gestione evento regolare*

1. dal momento in cui un dato evento viene classificato come pericoloso o potenzialmente tale, il robot dovrà reagire in non più di 4,5 secondi;

2. in generale il tempo medio di elaborazione e reazione ad un evento, dal momento in cui inizia il suo percorso dall'unità di elaborazione centrale, non dovrà superare gli 11 secondi.

Per il primo requisito andremo a considerare, nel modello, il tempo medio di servizio della entry *isDangerous*, mentre per il secondo quello di *checkEvent*, entrambe facenti parte del task *AI*.

Nello studio di tale architettura software, inoltre, si è deciso di considerare 13 come numero massimo di robot amici, al fine di studiare il comportamento del sistema anche un po' oltre quelli che vengono considerati i massimi livelli di carico.

Nell'appendice A sono riportati i sorgenti in formato LQN dei principali modelli di performance considerati in questo caso di studio.

6.1 Esecuzione della metodologia

Di seguito sono riportati nel dettaglio i risultati ed i modelli ottenuti dell'esecuzione delle singole iterazioni, previste dall'approccio oggetto di questo lavoro di tesi, per il caso di studio in esame.

In tabella 6.1 sono riportati i parametri utilizzati per il modello LQN rappresentante l'architettura software iniziale; inoltre, al termine di ogni iterazione, i parametri per il modello risultate dall'applicazione delle eventuali modifiche proposte saranno derivabili in via naturale dalla tabella stessa.

	Tempo di Servizio (S) o di Attesa (Z)	*Servizi Richiesti (# medio di chiamate)*	CPU
Environment			1
Events	2.0 (Z); 0.005 (S)	getEvent(1.0)	
OtherROBOTS			1
Friends	5.0 (Z); 0.005 (S)	receiveAlert(1.0)	
Sensors			2
getEvent	0.05 (S)	checkEvent(1.0)	
NetRX			2
receiveAlert	0.05 (S)	checkEvent(1.0)	
NetTX			2
sendAlert	0.2 (S); 1.25 (Z)		
AI			2
endElab	0.005 (S)		
memGetElab		getInfo(2.0)	
memPutElab		storeInfo(0.5)	
(70%) isRegular		handleRegular(1.0)	
(30%) isDangerous		handleDangerous(1.0)	
VolatileMemory			2
exec	0.00005 (S)		
StorageMemory			2
getInfo	0.005 (S)		
storeInfo	0.0075 (S)		
Handler			2
handleRegular	2.0 (S)	exec(8.0); getInfo(3.0); storeInfo(1.5); goThere(1.5); takeThat(2.0)	
handleDangerous	0.1 (S)	exec(1.0); getInfo(0.5); storeInfo(0.1); sendAlert(1.0); haltHere(1.0)	
MoveController			2
goThere	0.05 (S)	motorStatus(3.0); setSpeed(1.5); setDirection(0.75)	
takeThat	0.005 (S)	armStatus(4.0); moveSX(1.5); moveDX(1.0)	
haltHere	0.0005 (S)	motorStatus(1.0); setSpeed(1.0)	
Arms			2
armStatus	0.002 (S); 0.05 (Z)		
moveSX	0.000001 (S); 0.5 (Z)		
moveDX	0.000001 (S); 0.5 (Z)		
Motors			2
motorStatus	0.002 (S); 0.05 (Z)		
setSpeed	0.000001 (S); 0.5 (Z)		
setDirection	0.000001 (S); 1.5 (Z)		

Tabella 6.1: *Parametri iniziali del modello LQN considerato*

Quest'ultima riporta le *entry* contenute in ogni *task* presente nel modello e per ognuna di esse il tempo medio di servizio, di attesa pura ove previsto e le chiamate ad altre entry presenti nel sistema, rappresentanti le richieste di servizi offerti da altre risorse.

Le rappresentazioni grafiche dei modelli LQN risultanti da ogni iterazione, e riportati più avanti nel testo, derivano dell'applicazione delle modifiche apportate durante la stessa al modello immediatamente precedente, inoltre contengono informazioni come l'utilizzazione, il throughput ed il tempo medio di servizio per ogni risorsa software ed hardware identificata nell'architettura considerata.

Non appartenendo all'architettura software in esame i task *Environment* e *OtherROBOTS* utilizzano, come si evince dalla tabella, una CPU diversa da quella impiegata dagli altri: in questo modo si evita che le misurazioni vengano falsate dall'uso delle risorse computazionali del robot da parte di entità esterne allo stesso.

Iteration 0

Partendo dal modello in figura 6.3 e osservando i dati riepilogativi riportati nella tabella 6.2, si rileva che i requisiti a livello di sistema non risultano soddisfatti.

In figura 6.4 è inoltre illustrato l'andamento degli indici d'interesse, per i quali cioè sono stati definiti dei requisiti non funzionali oppure il cui valore può fornire informazioni utili, anche di carattere generale, in merito alle prestazioni del sistema software.

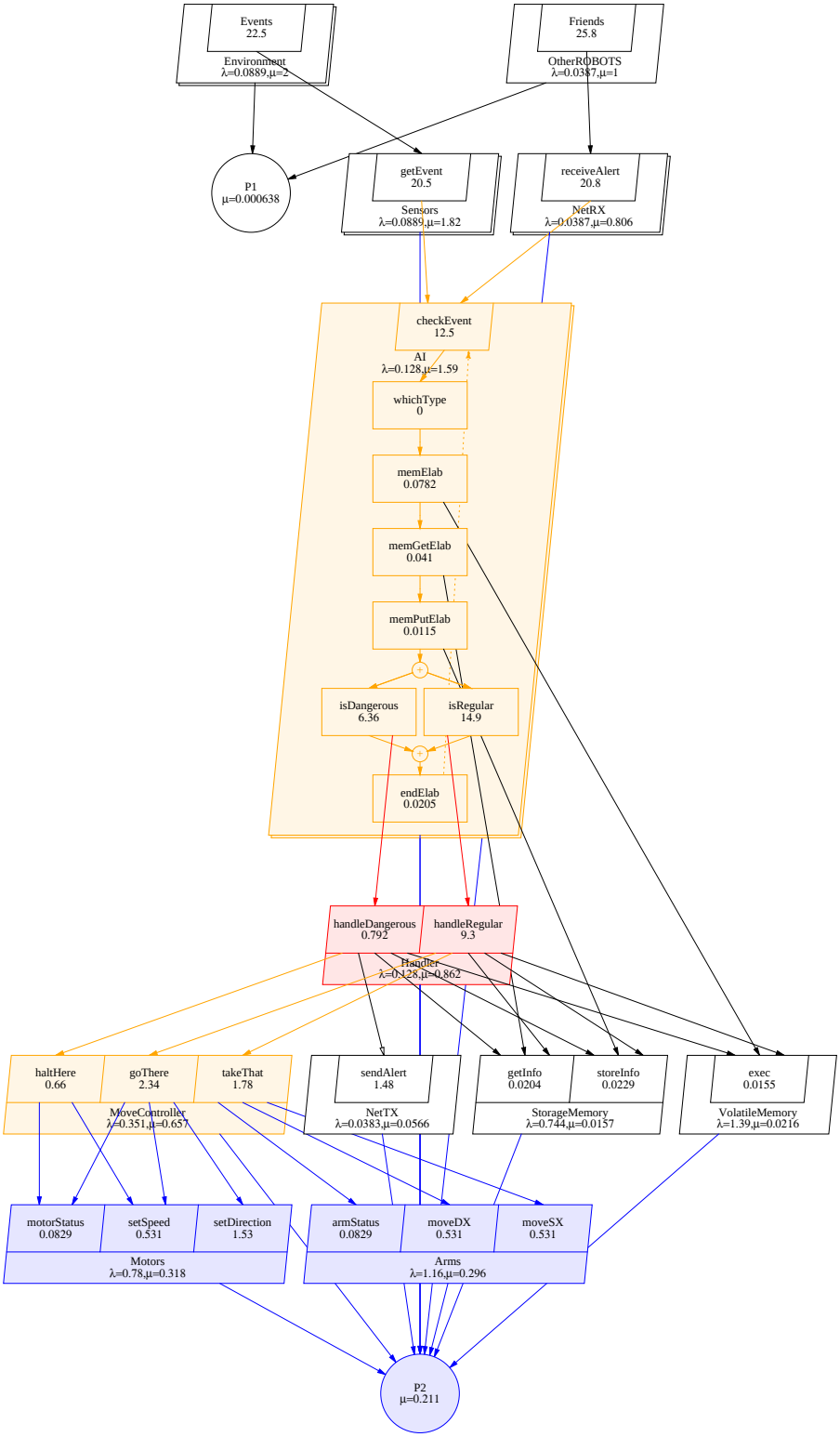


Figura 6.3: *Modello LQN iniziale - Iteration0*

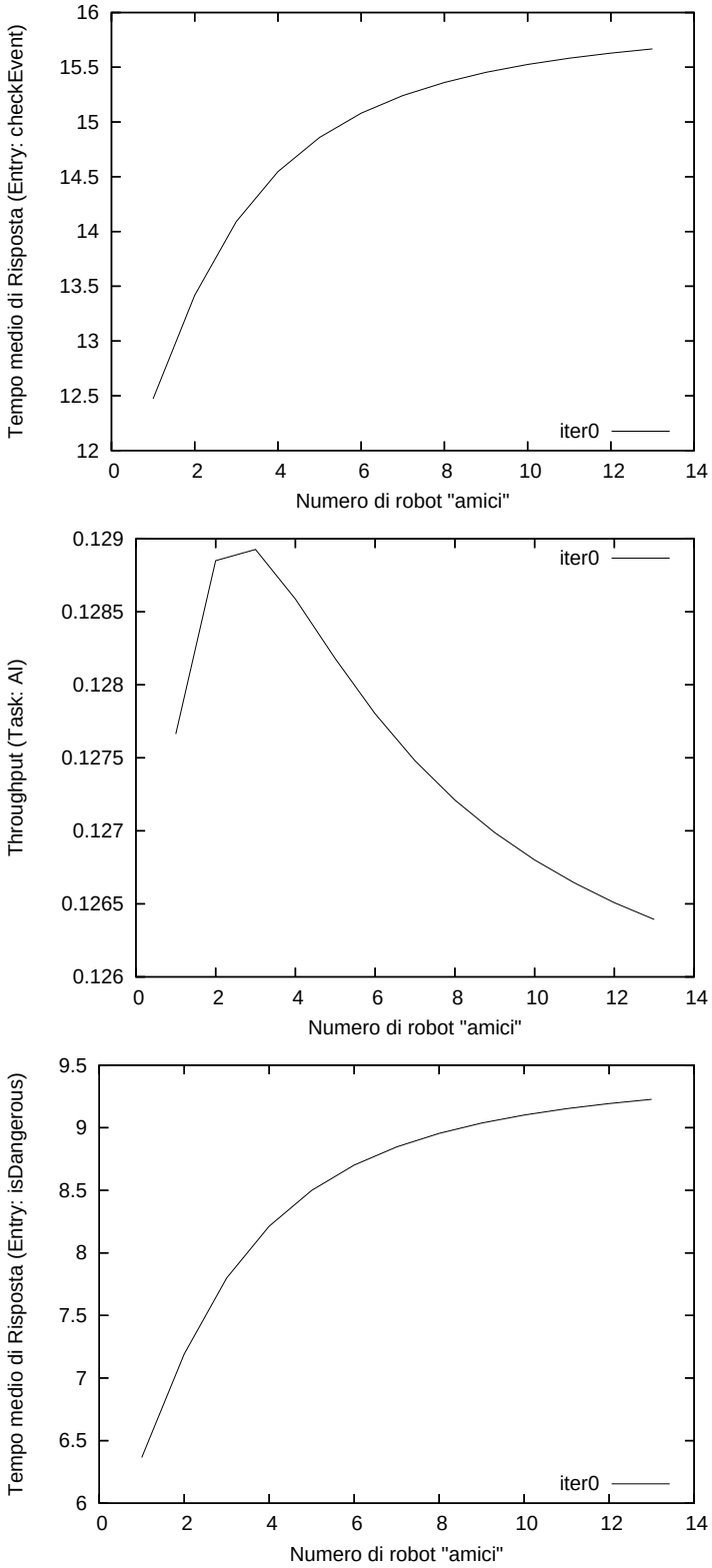


Figura 6.4: *Indici di interesse - Iteration0*

	Obiettivo	*Valore Raggiunto*
isDangerous (req1)	$<= 4{,}5$	$\approx$ **9,25**
checkEvent (req2)	$<= 11$	$\approx$ **15,5**

Tabella 6.2: *Riepilogo requisiti - Iteration0*

L'andamento asintotico, apparentemente anomalo, rappresentato nei grafici riguardanti il tempo medio di risposta è riconducibile al fatto che la misurazione dei valori assunti dagli indici di interesse viene effettuata all'interno di un collo di bottiglia, rappresentato dal task *AI*, e quindi non tiene conto della sua coda esterna. Infatti la crescita del numero di richieste che giungono al sistema, determinata dall'incremento del numero di robot amici riportato sull'asse delle ascisse, porta ad un aumento sempre maggiore del numero di quelle che rimangono in coda a tale task in attesa di servizio. Indicando quindi con $k_{AI_{in}}$ l'aumento di richieste che giungono al task *AI* e con $k_{AI_{out}}$ quello rilevato per le risorse che nella rappresentazione grafica del modello LQN sono al di sotto di tale task, la differenza $\Delta k_{AI} = k_{AI_{in}} - k_{AI_{out}}$ aumenta al crescere del numero di robot amici. Questo comporta quindi una delimitazione superiore del tempo medio di risposta, per le *entry* e le *activity* osservate per le misurazioni, in virtù del ridotto quantitativo di richieste aggiuntive che riescono ad "attraversare" il task *AI*.

Tale comportamento è da attribuirsi alla limitata molteplicità del task che lo porta a raggiungere livelli di utilizzazione prossimi alla saturazione già con un numero di robot amici pari a 6, come sarà visibile nel successivo studio degli indici di utilizzazione delle varie risorse.

Misurando invece il tempo medio di risposta sulla entry *getEvent*, presente nel task *Sensors* e di conseguenza influenzata dal tempo medio di attesa in co-

da per il task *AI*, questo avrà l'andamento tipico ed intuitivamente associabile a tale tipologia di indice; quanto appena descritto, è illustrato graficamente in figura 6.14 nella successiva sezione contenente le informazioni riepilogative sulle modifiche apportate durante l'esecuzione della metodologia.

Iteration 1

Il passo successivo consiste nel suddividere l'architettura software nei sotto-sistemi che saranno analizzati separatamente. Per operare il raggruppamento delle risorse software è necessario prima di tutto scegliere la tipologia dei sotto-sistemi che si intendono creare, quindi se cross-path rispetto ai servizi offerti oppure no. In questo caso particolare si dispone, a livello di sistema, di entrambe le tipologie di requisiti; infatti mentre il primo si riferisce ad un particolare servizio del sistema, il secondo è più generico e non riguarda un servizio in modo specifico. Come primo sotto-sistema sarà analizzato *SubS_ dangerous* di tipologia 2, la cui composizione è illustrata nella tabella 6.3 in funzione delle risorse utilizzate dal sistema per la gestione di un evento che sarà identificato dallo stesso come pericoloso.

Utilizzando la matrice d'interpretazione "sotto-sistema di tipologia 2", riportata in figura 4.5, in questo caso e a questa granularità si è in presenza di: tempo medio di risposta elevato e throughput nel rispetto dei requisiti in quanto, considerando che nessun vincolo riguarda quest'ultimo indice di performance, qualunque valore è da considerarsi accettabile.

Come prima analisi, la tecnica suggerisce di esaminare il sotto-sistema alla ricerca di un antipattern conosciuto: considerando in particolare il task *Handler* e la natura delle richieste che viaggiano all'interno del sistema, si

Risorsa Software	*Utilizzazione*	*Sottosistema* ***SubS_ dangerous***
Sensors	—	√
NetRX	—	√
NetTX	—	√
StorageMemory	1,57 %	√
VolatileMemory	2,16 %	√
AI	79,5 %	√
Handler	86,2 %	√
MoveController	65,7 %	√
Motors	31,8 %	√
Arms	29,6 %	
Obiettivi sottosistema		***R*** $\leq$ ***4,5*** *(su isDangerous)*

Tabella 6.3: *SottoSistemi - Iteration1*

può riconoscere l'antipattern "Unbalanced Processing: Extensive processing". Infatti tale task presenta un indice di utilizzazione sufficientemente alto e riceve due diverse tipologie di richieste: una riguardante la gestione di un evento considerato normale, al quale è associata un'interazione più o meno onerosa con l'ambiente (ed in generale un'elaborazione più impegnativa in termini computazionali e di tempo per il sistema) rispetto a quella prevista per gli eventi giudicati invece pericolosi.

La ristrutturazione architetturale, derivante dall'eliminazione dell'antipattern individuato, porta ad ottenere il modello LQN rappresentato in figura 6.5 con le seguenti variazioni prestazionali riassunte e messe in relazione con i requisiti non funzionali del sistema nella tabella 6.4.

Come si può evincere dall'andamento degli indici d'interesse in figura 6.6, le modifiche applicate al sistema portano ad un notevole miglioramento delle performance dello stesso per quanto concerne il primo requisito non

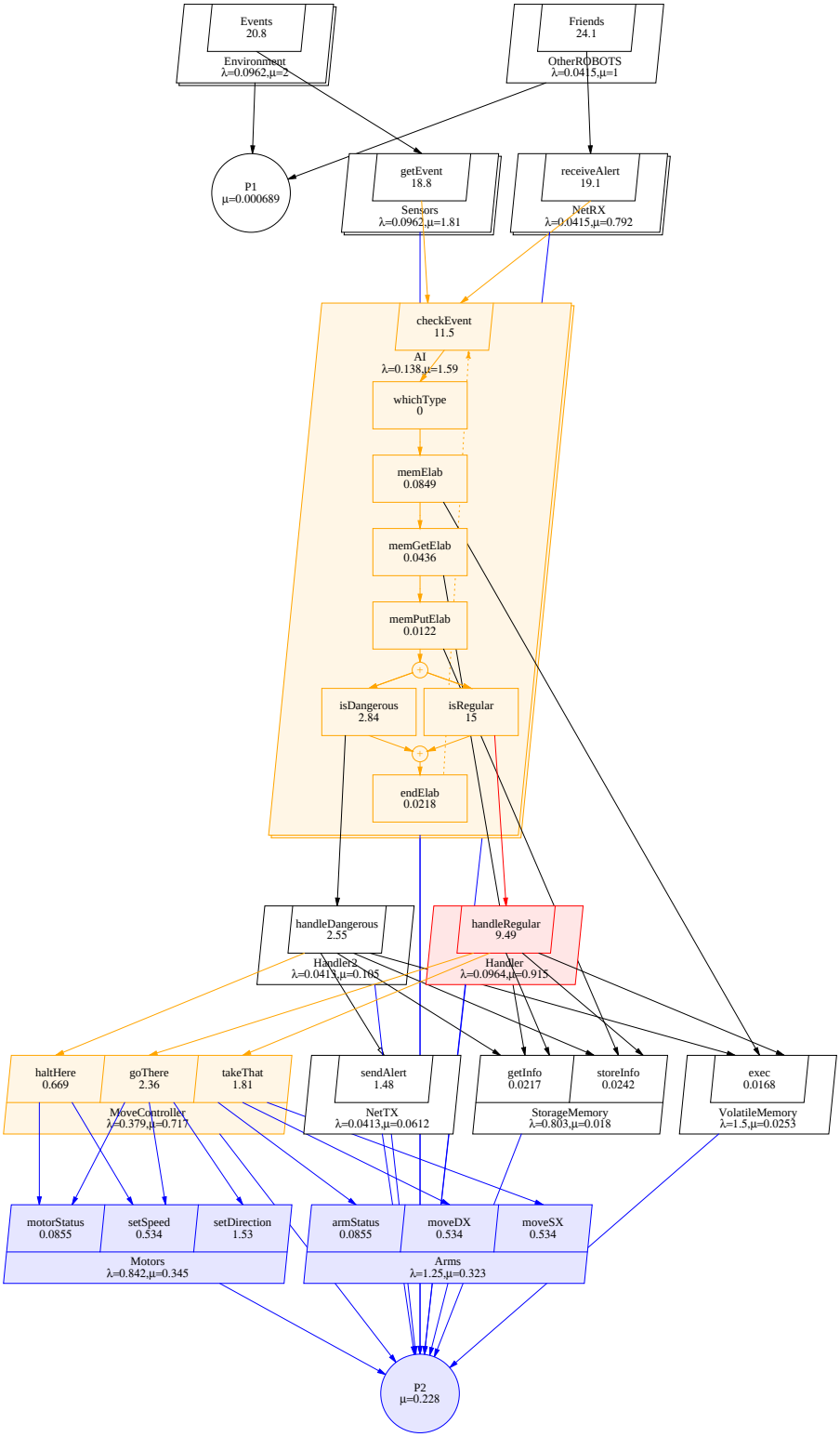


Figura 6.5: *Modello LQN - Iteration1*

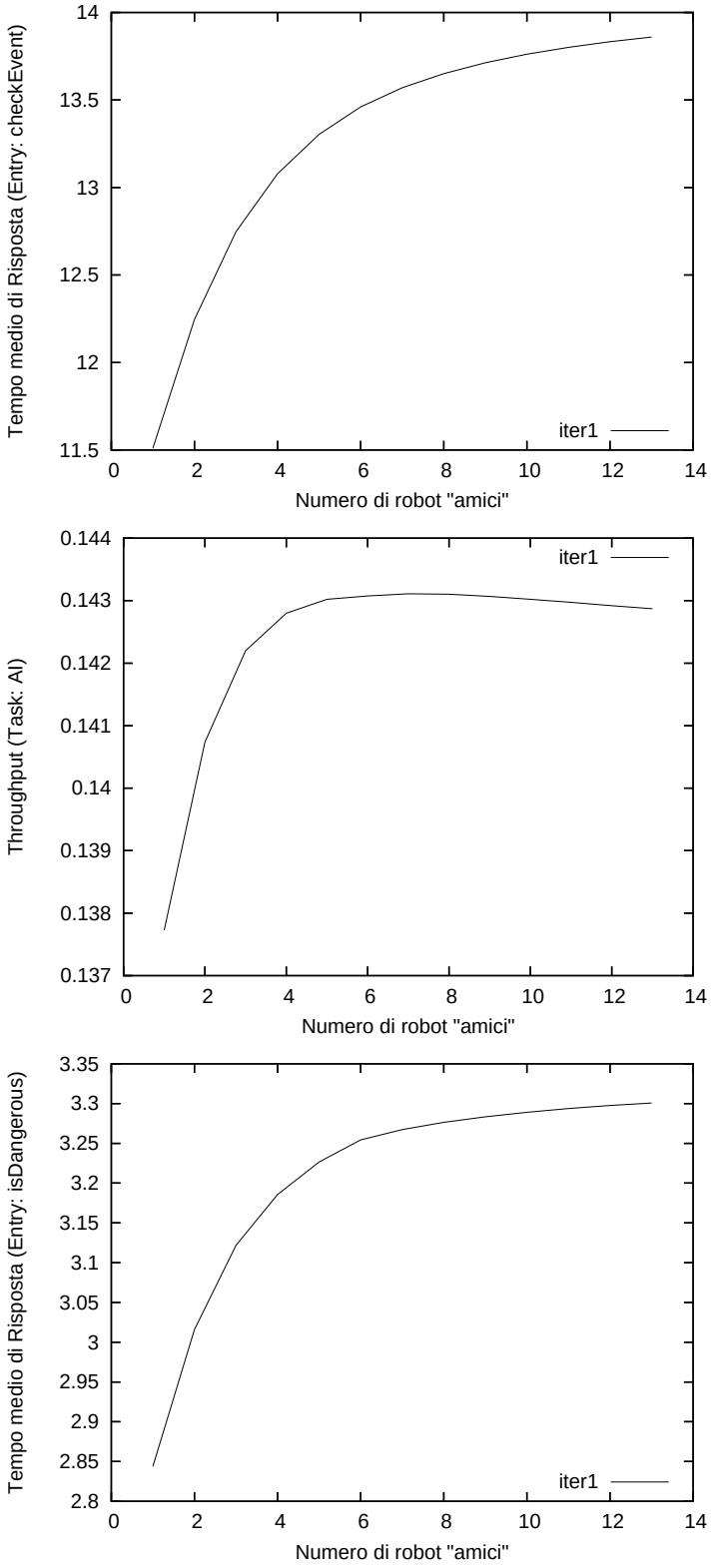


Figura 6.6: *Indici di interesse - Iteration1*

	Obiettivo	*Valore Raggiunto*	*% rispetto al precedente*
isDangerous (req1)	<= 4,5	≈ 3,3	- 64,32 %
checkEvent (req2)	<= 11	≈ **13,75**	- 11,29 %

Tabella 6.4: *Riepilogo requisiti - Iteration1*

funzionale fornito, che risulta peraltro soddisfatto nella nuova architettura; il secondo invece presenta un valore ancora al di sopra del massimo richiesto e sarà quindi necessario proseguire con l'analisi ed il perfezionamento del nuovo sistema.

Dal modello così ottenuto si ricava il nuovo sotto-sistema *SubS_ dangerous*, al quale apparterrà il task *Handler2*, introdotto dalla ristrutturazione, ma non *Handler* che a questo punto non fornisce più servizi necessari alla gestione di eventi pericolosi: aspetto che guida, in questo caso, il raggruppamento delle risorse in sotto-sistemi.

Iteration 2

A titolo dimostrativo e di studio si è scelto, per questa iterazione, di utilizzare entrambe le tipologie previste per la suddivisione dell'architettura in sotto-sistemi; nella reale applicazione dell'approccio descritto in questo lavoro di tesi sarà sufficiente scegliere, in base alle opportune valutazioni del caso, quello che si ritiene sia il più conveniente.

Iteration 2a (sotto-sistemi di tipologia 2)

Avendo raggiunto il soddisfacimento del primo requisito, l'obiettivo diventa quello di abbassare il tempo medio di risposta per un generico evento e

a tal fine, considerando che secondo le stime utilizzate per la modellazione il 70% degli eventi ricevuti dal sistema sono considerati normali e quindi non pericolosi, si passa ad analizzare il sotto-sistema *SubS_ regular*, la cui composizione è illustrata nella tabella 6.5 insieme a quella del sotto-sistema *SubS_ dangerous*. In questo caso non esiste un requisito specifico definito per l'evento "normale", ma viene scelto tale approccio osservando che le prestazioni raggiunte nella gestione di tale classe di eventi influenzano in maniera maggiore quelle del sistema stesso considerato nella sua interezza. Anche in questo caso, come per il precedente, lo schema di granularità "sotto-sistema di tipologia 2" in figura 4.5 suggerisce di tentare l'individuazione di un antipattern nel sotto-sistema considerato ed in questo caso particolare l'interazione tra i task *MoveController*, *Motors* ed *Arms* può essere ricondotta al "Blob".

Risorsa Software	*Utilizzazione*	*Sottosistema* ***SubS_ dangerous***	*Sottosistema* ***SubS_ regular***
Sensors	—	√	√
NetRX	—	√	√
NetTX	—	√	
StorageMemory	1,8 %	√	√
VolatileMemory	2,5 %	√	√
AI	79,5 %	√	√
Handler	91,5 %		√
Handler2	10,5 %	√	
MoveController	65,7 %	√	√
Motors	31,8 %	√	√
Arms	29,6 %		√
Obiettivi sottosistema		***R ≤ 4,5*** *(su isDangerous)*	—
Obiettivi sistema		***R ≤ 11*** *(su checkEvent)*	

Tabella 6.5: *SottoSistemi - Iteration2a*

Iteration 2b (sotto-sistemi di tipologia 1)

Per tutti i sotto-sistemi identificati e riportati in tabella 6.6 la matrice d'interpretazione "sotto-sistema di tipologia 1" in figura 4.4, considerando il caso di throughput basso e tempo medio di risposta "don't care", suggerisce, come prima opzione, di ricercare antipattern conosciuti all'interno dei sotto-sistemi considerati. Nel caso del sotto-sistema *SubS_ move*, l'interazione tra i task *MoveController*, *Motors* ed *Arms* può essere ricondotta al "Blob".

Risorsa Software	*U*	*X*	***SubS sensors***	***SubS elab***	***SubS move***
Sensors	—	0,096	√		
NetRX	—	0,042	√		
NetTX	—	0,041	√		
StorageMemory	1,8 %	0,803		√	
VolatileMemory	2,5 %	1,501		√	
AI	79,5 %	0,138		√	
Handler	91,5 %	0,096		√	
Handler2	10,5 %	0,041		√	
MoveController	65,7 %	0,379			√
Motors	31,8 %	0,842			√
Arms	29,6 %	1,253			√
X sotto-sistema			0,041	0,041	0,379
X_{MAX} sotto-sistema			0,690	1,525	2,973
X/X_{MAX} sotto-sistema			5,94 %	2,69 %	12,75 %

Tabella 6.6: *SottoSistemi - Iteration2b*

Entrambi gli approcci illustrati, tra loro alternativi, portano all'individuazione dello stesso antipattern: dall'eliminazione di quest'ultimo si ottiene una nuova architettura software, rappresentata in figura 6.7, che permette al sistema di raggiungere i livelli di performance riassunti nella tabella 6.7, mentre l'andamento degli indici d'interesse è illustrato in figura 6.8.

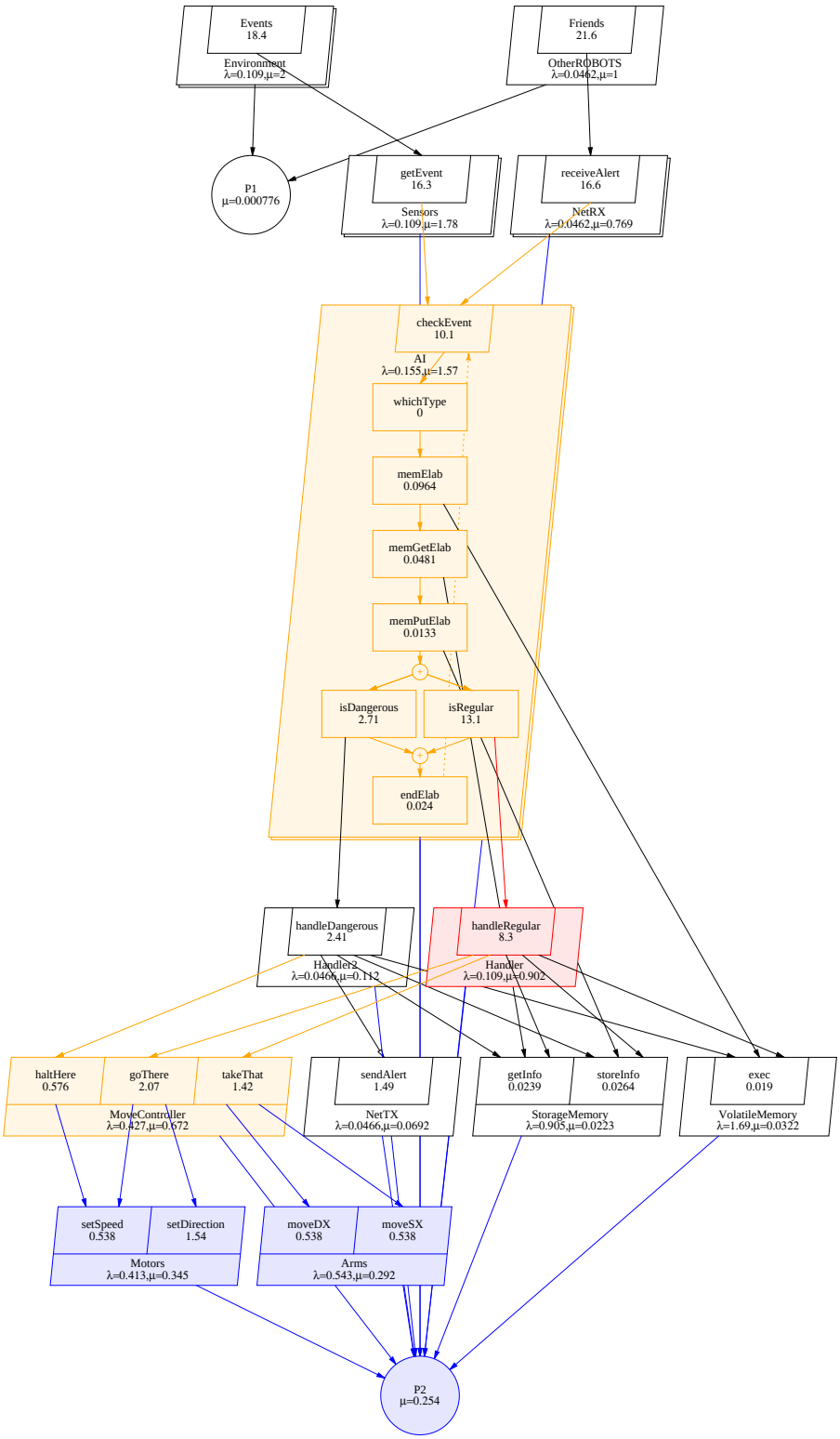


Figura 6.7: *Modello LQN - Iteration2*

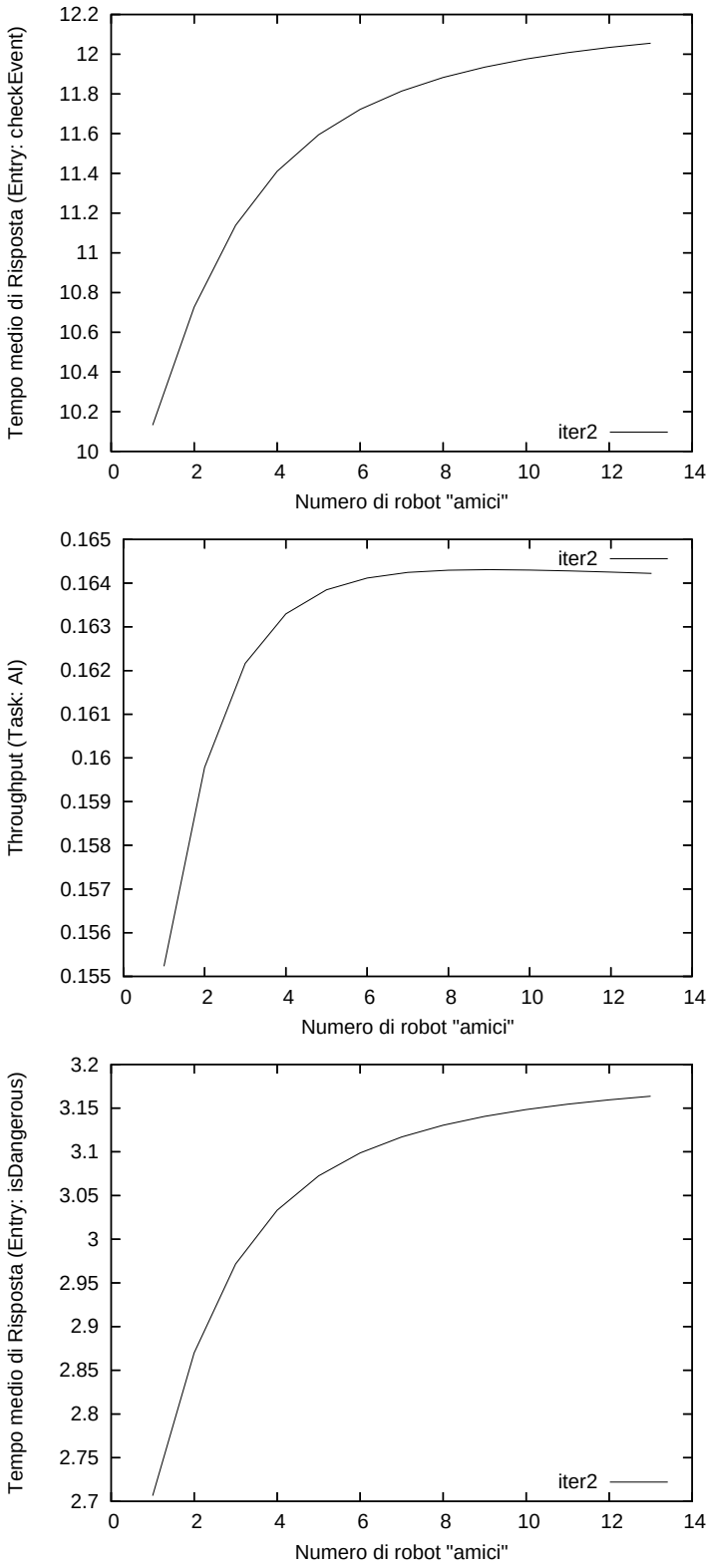


Figura 6.8: *Indici di interesse - Iteration2*

	Obiettivo	*Valore Raggiunto*	*% rispetto al precedente*
isDangerous (req1)	<= 4,5	≈ 3,15	- 4,54 %
checkEvent (req2)	<= 11	≈ **12**	- 12,73 %

Tabella 6.7: *Riepilogo requisiti - Iteration2*

Iteration 3

Considerando che l'eliminazione dell'ultimo antipattern individuato non porta a modifiche architetturali tali da giustificare una differente suddivisione in sotto-sistemi, l'ultima realizzata, ad esempio quella riportata nella tabella 6.6, sarà ancora valida per l'iterazione successiva.

A questo punto, analizzando i sotto-sistemi considerati non si rileva la presenza di antipattern architetturali conosciuti e si passa quindi, come suggerito dalla matrice di interpretazione, a livello "risorsa software": ipotizzando che sia stata seguita l'iterazione 2a illustrata in precedenza e sulla scia dell'osservazione riguardante la maggiore influenza del sotto-sistema *SubS_regular* sulle prestazioni complessive del sistema, si passa ad analizzare l'andamento degli indici di utilizzazione, riportati in figura 6.9, per le risorse software in esso contenute, in modo da iniziare ad esaminare quella che presenta il più alto valore in tale ambito.

Prima di proseguire nell'analisi è necessario considerare che non sarà possibile agire sulla molteplicità delle risorse rappresentate dai task *MoveController*, *Arms* e *Motors*: tale operazione non troverebbe applicazione nella realtà, in quanto è necessario considerare anche la natura dei servizi offerti dalle varie componenti ed in questo caso i task *Arms* e *Motors* non possono essere clonati in quanto ognuno di essi si riferisce ad un singolo insieme di

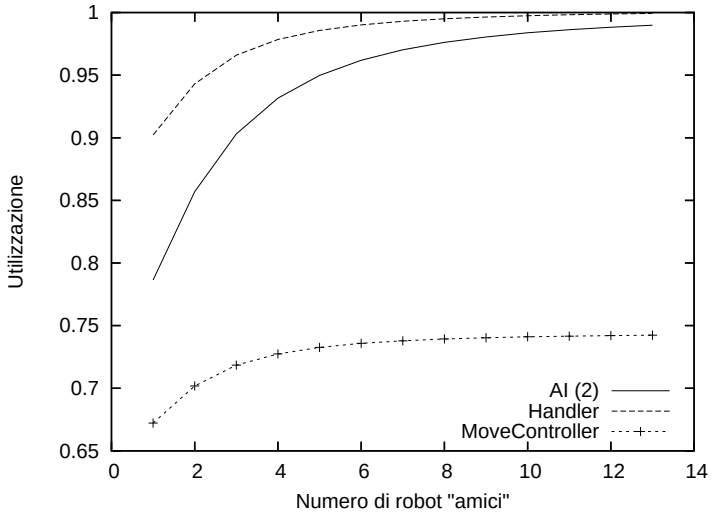


Figura 6.9: *Utilizzazione risorse SubS_regular - Iteration2a*

servomeccanismi fisici; del resto la clonazione di *MoveController* porterebbe alla creazione di più copie della stessa risorsa in concorrenza tra loro ed in questo caso potrebbe accadere che richieste derivanti dalla gestione di un dato evento possano essere interfogliate con quelle di un secondo evento in coda ai task *Arms* e *Motors*, ottenendo un comportamento del robot non corretto e difficilmente prevedibile allo stesso tempo.

Si considera quindi la risorsa *Handler* contenuta nel sotto-sistema in esame e si prova a clonare tale risorsa, come suggerito dalla matrice d'interpretazione "risorsa software" in figura 4.6; in un modello LQN tale operazione viene eseguita aumentando la molteplicità del task considerato, così come illustrato in dettaglio nel capitolo 4.

Per non appesantire e rendere più chiara la trattazione, si è scelto di rappresentare i risultati ottenuti dalle seguenti sotto-iterazioni nei grafici riassuntivi in figura 6.12.

Iteration 3a

Aumentando di una unità la molteplicità della risorsa *Handler*, che da 1 passa quindi a 2, si ottengono evidenti miglioramenti in termini prestazionali per il sistema nella sua interezza, ma nel contempo un marcato peggioramento nel tempo medio di risposta per quanto concerne gli eventi pericolosi; tale fenomeno è da attribuirsi all'aumento di concorrenza, derivante dall'incremento del numero di risorse di tipo *Handler* presenti nel sistema, per *Handler2* che gestisce tale tipologia di eventi.

I nuovi valori assunti da tale indice rispettano comunque i requisiti di performance previsti per questo genere di eventi, di conseguenza la modifica appena descritta può essere considerata valida tenuto conto dei miglioramenti prestazionali globalmente ottenuti dal sistema; in figura 6.10 sono riportate le nuove curve di utilizzazione.

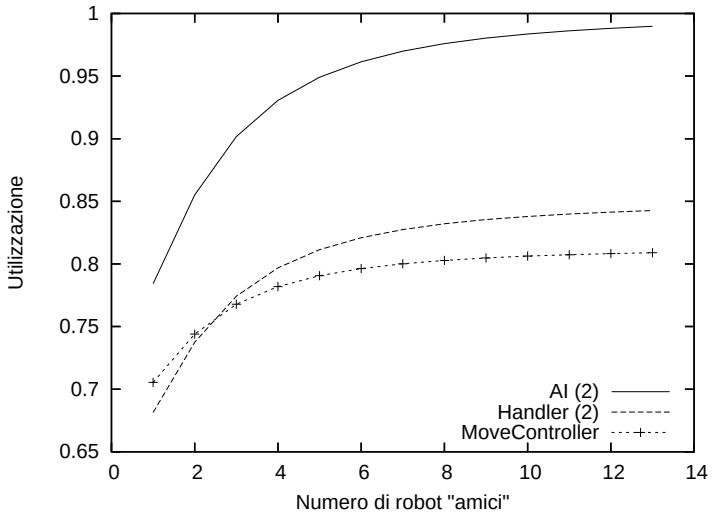


Figura 6.10: *Utilizzazione risorse SubS_regular - Iteration3a*

Iteration 3b

Visti i livelli di utilizzazione raggiunti dalla risorsa *AI*, la matrice d'interpretazione suggerisce anche in questo caso di tentare con la clonazione della stessa, portando la sua molteplicità quindi da 2 a 3. Tale modifica si rivela però svantaggiosa, probabilmente a causa dell'eccessivo numero di richieste concorrenti che viene a determinarsi in coda alle altre risorse presenti nel sistema.

Tale modifica non sarà quindi applicata all'architettura.

Iteration 3c

Operando una sorta di rollback, tornando cioè a considerare lo step3a come situazione di partenza, si prova a considerare la seconda risorsa più utilizzata, *Handler*, e se ne aumenta la molteplicità da 2 a 3. In questo caso la modifica non sortisce alcun effetto: si ottengono infatti valori, per gli indici di interesse, identici a quelli rilevati per il modello allo step3a.

Di conseguenza neanche questa modifica verrà applicata al sistema.

Iteration 3d

A questo punto, considerando il fatto che operare ulteriormente sulla molteplicità delle risorse software non è utile al miglioramento delle performance del sistema e che l'hardware da queste utilizzato, nello specifico CPU e memorie, risulta essere proporzionato al carico presentando valori di utilizzazione medio-bassi, il passo successivo consisterà nell'agire sulla parte hardware re-

lativa ai servomeccanismi che sono di fatto la componente più lenta presente all'interno del sistema.

Apportando un miglioramento pari a 0.1 secondi per ogni attività svolta dai servomeccanismi, nel modello rappresentati dai task *Motors* ed *Arms* i quali introducono nel sistema attesa cosiddetta "pura" (che non deriva cioè da cicli di elaborazione), si ottiene un notevole incremento delle performance dell'intero sistema, come si può notare dai grafici riepilogativi in figura 6.12 riguardanti lo step3.

L'iterazione in esame, considerata nella sua interezza e riportata graficamente in figura 6.11, si traduce quindi nelle seguenti modifiche al sistema:

- Incremento della molteplicità della risorsa *Handler* da 1 a 2;
- Miglioramento dei tempi di attesa legati ai servomeccanismi pari a 0.1 secondi.

Al termine del procedimento il sistema raggiunge livelli di performance, riportati nella tabella 6.8, che soddisfano i requisiti non funzionali indicati.

I grafici riepilogativi in figura 6.13 mostrano invece le variazioni prestazionali del sistema oggetto di studio dall'iterazione 0 all'iterazione 3.

	Obiettivo	*Valore Raggiunto*	*% rispetto al precedente*
isDangerous (req1)	<= 4,5	≈ 4	**+ 26,98 %**
checkEvent (req2)	<= 11	≈ 10,3	- 14,17 %

Tabella 6.8: *Riepilogo requisiti - Iteration3*

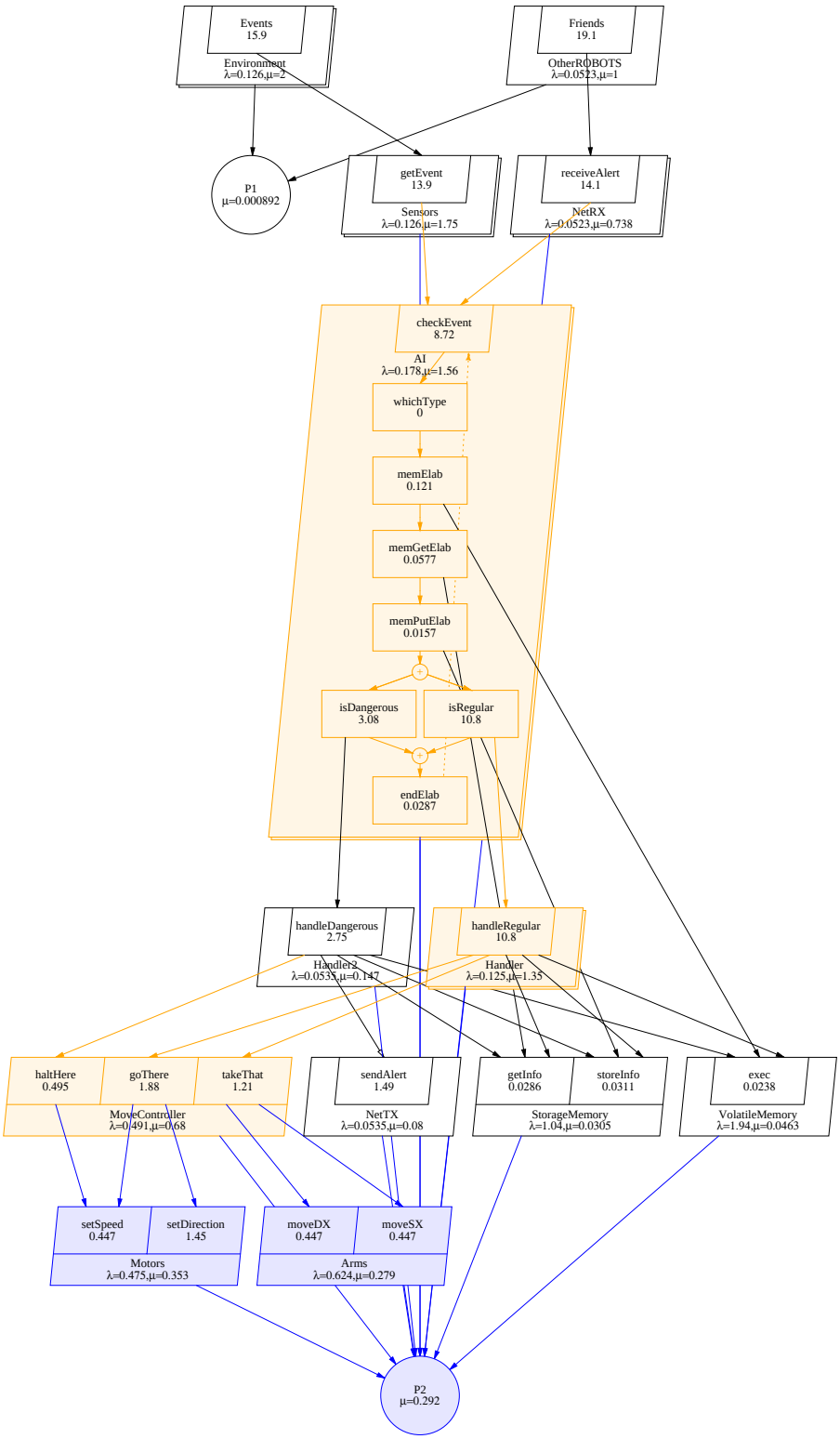


Figura 6.11: *Modello LQN - Iteration3*

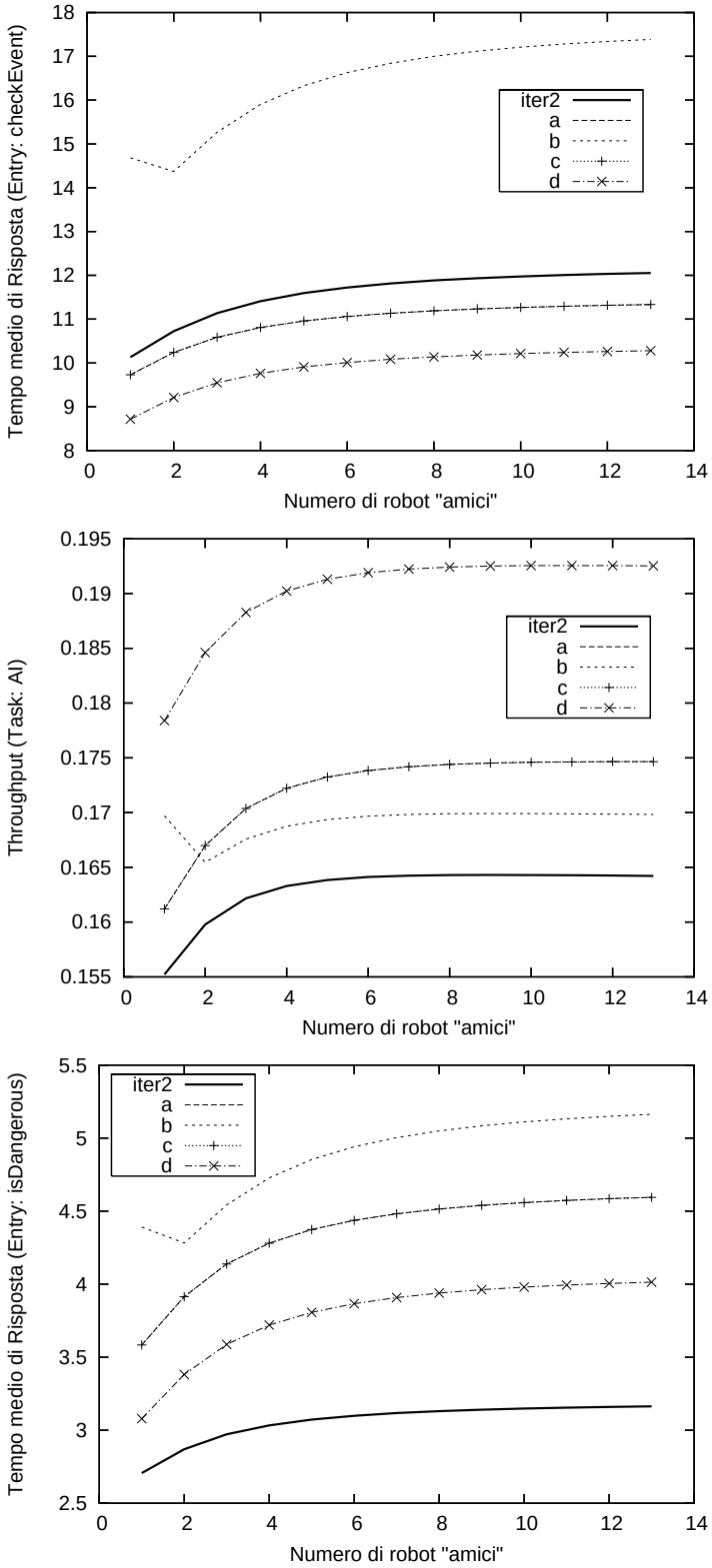


Figura 6.12: *Indici di interesse - Iteration3*

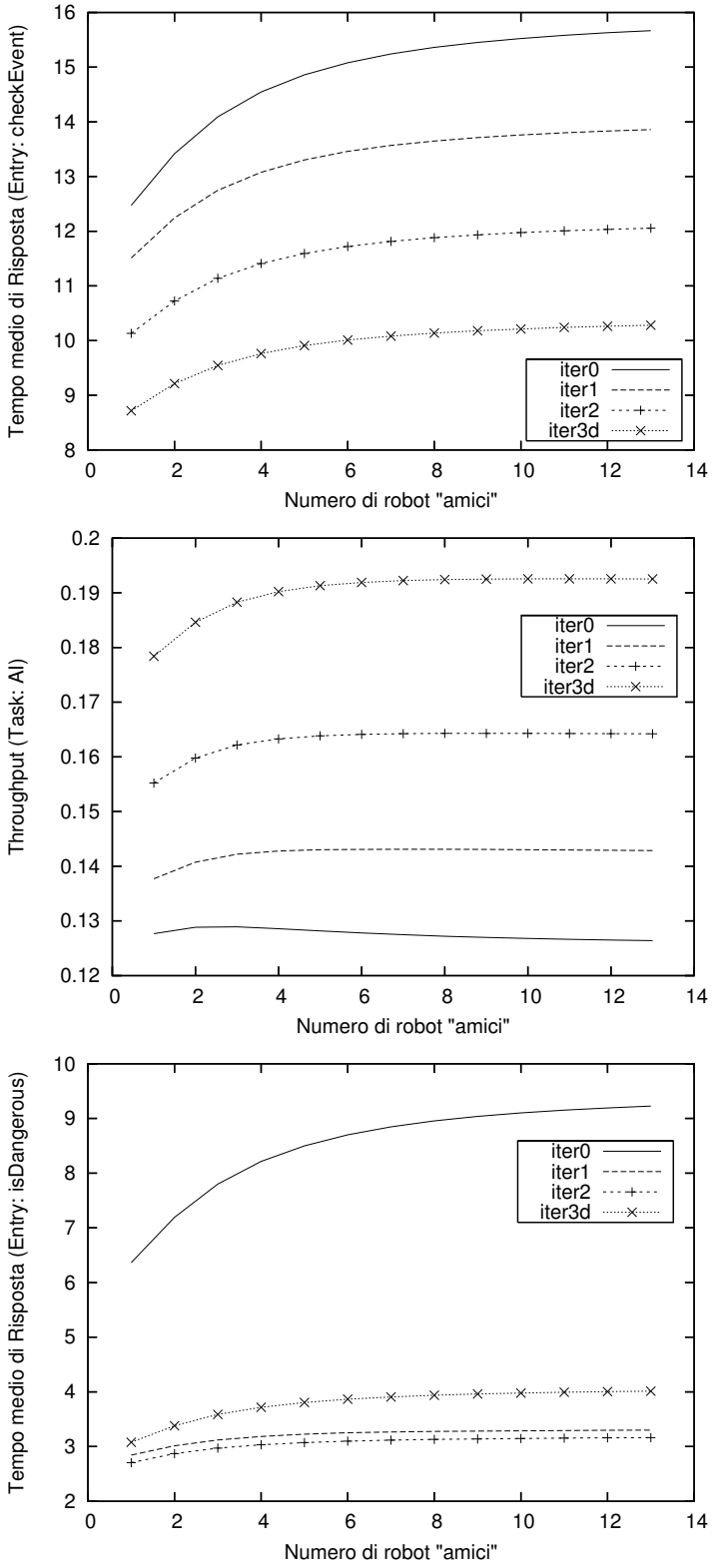


Figura 6.13: *Riepilogo performance da Iteration0 a Iteration3*

In figura 6.14 è inoltre riportato l'andamento del tempo medio di risposta misurato sulla entry *getEvent* (contenuta nel task *Sensors*), da tale rappresentazione si evincono, anche in questo caso, i miglioramenti prestazionali derivanti dalle modifiche architetturali apportate durante l'applicazione della metodologia in esame.

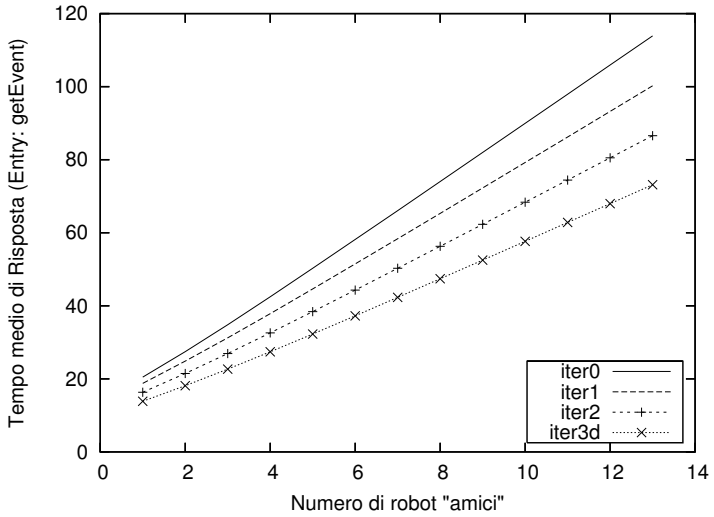


Figura 6.14: *Riepilogo Tempo Medio di Risposta su getEvent*

6.2 Esempio di utilizzo del tool di supporto

Anche se il tool di supporto descritto nel capitolo 5 non è in grado attualmente di riconoscere tutti gli antipattern e applicare tutte le tecniche risolutive definite in questo lavoro di tesi ed utilizzate nelle precedenti sezioni di questo capitolo, è comunque in grado di ristrutturare l'architettura utilizzata come caso di studio, rappresentata sotto forma di modello LQN in figura 6.3, in modo che questa rispetti i requisiti non funzionali per essa definiti e riassunti in tabella 6.2.

L'esecuzione del tool, in modalità verbosa ed utilizzando la suddivisione in sotto-sistemi di tipologia 2 riportata in tabella 6.5, produce il seguente output:

```
Searching for known antipattern in SubSystem 'dangerous'...
   Possible antipatterns in 'dangerous' subsystem:
    (0) Extensive Processing
   Selection (-1 to skip): 0
== Some requirements aren't Satisfied...
=== Step 1 created
   Extensive Processing antipattern was removed
Searching for known antipattern in SubSystem 'regular'...
   Possible antipatterns in 'regular' subsystem:
    (0) Extensive Processing
   Selection (-1 to skip): -1
   Skipping...
== Some requirements aren't Satisfied...
=== Step 2 created
Resource Handler:
  raise multiplicity from 1 to 2
Performance Variations:
  isDangerous;0.2425 (service_time)
  checkEvent;-0.0497272727273 (service_time)

ALL Requirements are Satisfied!

Creating Summary Graphs (using gnuplot)...
Creating Models Graphs (using lqn2ps)...
```

In questo caso il tool provvede quindi alla rimozione dell'antipattern "Unbalanced Processing: Extensive processing" rilevato nel sotto-sistema *dangerous*, eseguendo lo *Step 1* illustrato in figura 6.15; successivamente viene rilevata un'altra istanza dello stesso antipattern, ma questa volta, trattandosi di un falso positivo, si sceglie manualmente di ignorare la modifica proposta.

Il passo successivo consiste nell'aumento della molteplicità per il task *Handler* da 1 a 2, con la conseguente creazione dello *Step 2*, rappresentato in

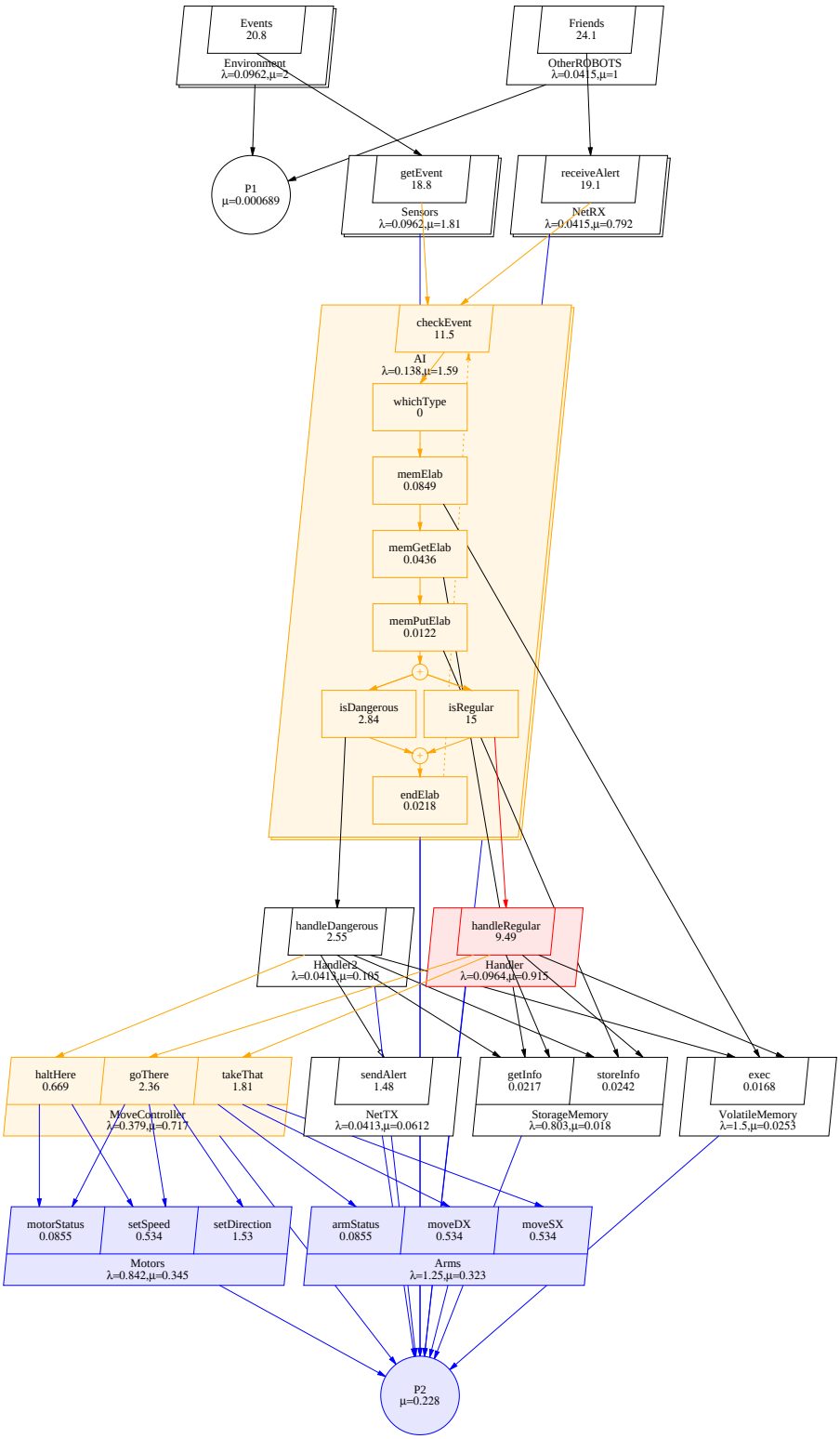


Figura 6.15: *Modello LQN generato dal tool per il caso di studio - Step1*

figura 6.16; quest'ultima modifica all'architettura non porta variazioni prestazionali complessivamente vantaggiose, infatti a fronte di un miglioramento del 5% rispetto al requisito definito per *checkEvent* si riscontra invece un peggioramento del 24,2% per l'altro. In questo caso però la modifica porta al soddisfacimento di tutti i vincoli definiti nei requisiti non funzionali e per questo motivo il tool classifica tale modifica come utile al raggiungimento degli obiettivi di performance. Al termine dell'esecuzione del tool sull'architettura software oggetto di studio si ottiene, per gli indici di interesse, l'andamento illustrato in figura 6.17, dalla quale si evince inoltre che il modello LQN risultante è contenuto nello *Step 2*. Le performance raggiunte dall'architettura ottenuta dall'utilizzo del tool di supporto su quella originale sono riassunte e confrontate con quelle rilevate dall'applicazione manuale dell'approccio definito in questo lavoro di tesi, riportata nelle precedenti sezioni di questo capitolo, in tabella 6.9.

La principale differenza, tra le due architetture software risultanti dai procedimenti considerati, consiste nella mancata risoluzione dell'antipattern *Blob* da parte del tool, operazione che invece viene eseguita nell'appoccio manuale. Esso si comporta però egregiamente per quanto concerne l'individuazione delle risorse critiche e la loro clonazione, discriminando inoltre correttamente, almeno in questo caso, tra modifiche architetturali utili e quelle che invece non lo sono.

	Obiettivo	*Approccio* ***Manuale***	*Approccio* ***Automatico***
isDangerous (req1)	$<= 4{,}5$	≈ 4	≈ 4
checkEvent (req2)	$<= 11$	$\approx 10{,}3$	11

Tabella 6.9: *Riepilogo Performance, risoluzione manuale ed automatica*

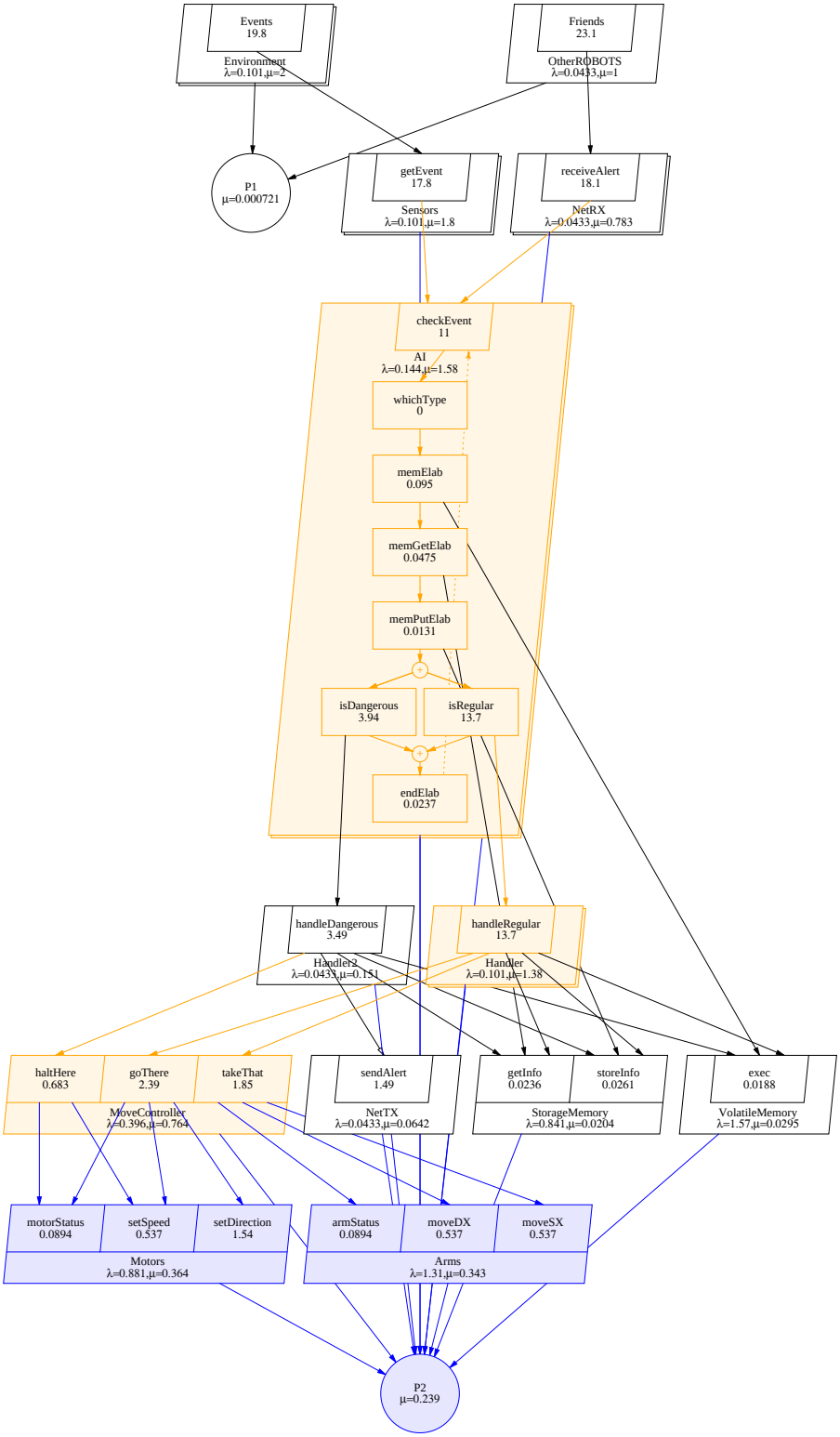
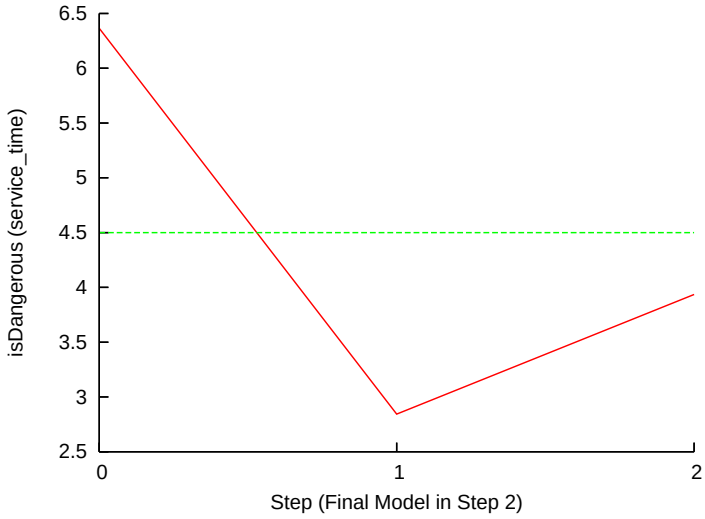


Figura 6.16: *Modello LQN generato dal tool per il caso di studio - Step2*



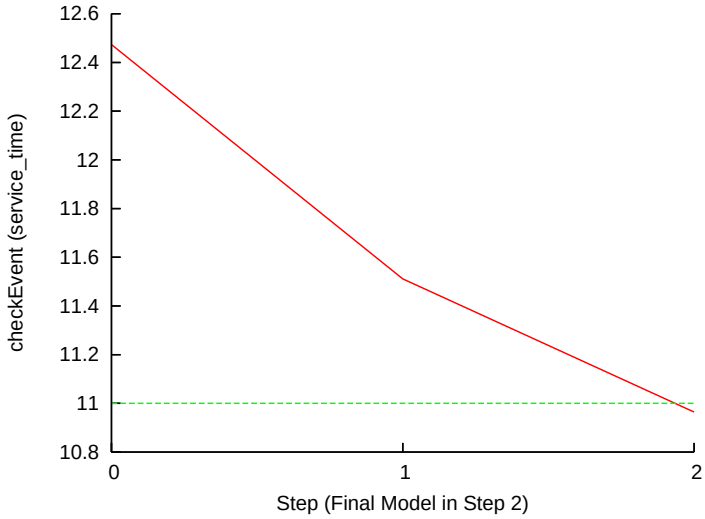


Figura 6.17: *Andamento degli indici di interesse ottenuti dalla risoluzione automatica del caso di studio utilizzando il tool descritto nel capitolo 5*

Capitolo 7

Conclusioni

L'approccio proposto in questo lavoro di tesi rappresenta una possibile soluzione al problema dell'integrazione della software performance analysis, anche attraverso la sua automazione, all'interno dei vari processi di sviluppo software ed in generale nelle fasi del ciclo di vita di quest'ultimo. Attraverso l'uso di schematizzazioni come le matrici d'interpretazione, nonché la possibilità di automatizzare buona parte della tecnica proposta, minimizzando l'interazione con i designer, si è in grado di presentare deduzioni e suggerimenti basati sull'interpretazione sistematica dei risultati ottenuti dalle analisi prestazionali e riguardanti una data architettura software, piuttosto che fornire un insieme di valori che, per i non esperti, potrebbero non essere significativi.

La tecnica illustrata è inoltre estensibile attraverso l'introduzione di nuovi profili di antipattern architetturali, perfezionabile intervenendo sulle matrici d'interpretazione e sull'attuale profilazione degli antipattern stessi.

L'approccio descritto in questa tesi è stato realizzato in modo da non essere vincolato ad un particolare modello di performance ma è necessario, in

tale ambito, osservare che potrebbe essere vantaggioso utilizzare opportuni accorgimenti a seconda del formalismo scelto: nelle LQN non rappresenta un problema ricalcolare gli indici per l'intero sistema in esame ad ogni modifica architetturale grazie anche all'efficienza degli strumenti disponibili; in altri casi, invece, potrebbe essere utile sfruttare le peculiarità offerte da un particolare modello di performance e dai tool per esso disponibili al fine di ottimizzare il calcolo degli indici. Utilizzando, ad esempio, Petri Net con GreatSPN [19] è possibile suddividere il sistema in sotto-reti da poter eventualmente analizzare singolarmente; in questo caso sarà però necessario fare attenzione alle possibili interazioni tra le stesse che, se non opportunamente considerate, potrebbero falsare i risultati delle analisi svolte.

L'input necessario al procedimento consiste nel modello di performance che rappresenta l'architettura software in esame, che però non è sempre disponibile, specie se gli sviluppatori non dispongono di particolari competenze nell'ambito dell'analisi prestazionale. In questo caso possono essere di grande aiuto i tool disponibili per il formalismo scelto: ad esempio per LQN esistono strumenti, anche se ancora in fase di perfezionamento, che permettono di derivare il modello di performance desiderato a partire da alcuni diagrammi UML opportunamente annotati [20, 21, 22].

Le difficoltà maggiori che si incontrano nell'automazione dell'intero procedimento sono essenzialmente legate, in maniera inversamente proporzionale, alla qualità dell'*antipattern profiling*. Migliore sarà la formalizzazione dei profili che permettono una corretta rilevazione degli antipattern conosciuti all'interno delle architetture software esaminate, minimizzando quindi la possibilità di incorrere in falsi positivi e in scelte multiple, minori saranno

le difficoltà che si presenteranno nell'automazione, e nel contempo minore la necessità di interazione con i progettisti. In tale ambito potrebbe essere utile introdurre l'utilizzo di un linguaggio specifico che aiuti a definire con maggiore precisione i vari profili, tenendo conto inoltre che in alcuni casi potrebbe non essere sufficiente analizzare topologicamente un'architettura software alla ricerca di antipattern, mentre potrebbero rendersi necessarie informazioni di carattere semantico per una corretta individuazione ed eventuale risoluzione dello stesso.

Nello sviluppo di un sistema software, inoltre, potrebbe risultare troppo oneroso, in termini di tempo e di denaro, modificare radicalmente un'architettura software esistente; per questo motivo si potrebbero "pesare" le modifiche architetturali proposte dal procedimento con i rispettivi costi stimati, in modo tale da rendere possibile un'ottimizzazione del rapporto costi-benefici.

Al fine di migliorare il riconoscimento degli antipattern, specialmente nel caso in cui vi siano diverse opportunità di scelta possibili, e come strumento per l'elaborazione delle stime riguardanti i costi appena descritti, potrebbe essere utile integrare nel procedimento il supporto di un agente intelligente che, avendo acquisito conoscenza dalle analisi e ristrutturazioni architetturali svolte in precedenza, potrebbe essere d'aiuto nella scelta della modifica più opportuna; in particolare potrebbe essere interessante considerare l'applicazione di reti neurali: considerando infatti le loro potenzialità nel campo del pattern matching [24], potrebbero rappresentare un utile strumento nell'individuazione degli antipattern.

Appendice A

Sorgenti Modelli LQN

Di seguito sono riportati i principali modelli LQN utilizzati in questo lavoro di tesi con i rispettivi riferimenti alle illustrazioni che li rappresentano.

Antipattern Blob, figura 4.18

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
p P2 f
-1

T 0
t Source r request -1 P2 m 1
t Controller n op1 -1 P1
t ResourceA n Astatus A1 A2 -1 P1
t ResourceB n Bstatus B1 B2 -1 P1
-1

E 0
Z request 5.0 -1
s request 0.5 -1
y request op1 1.0 -1
### Task CONTROLLER
# Invoca lo "status" prima e dopo un'operazione
s op1 0.005 -1
y op1 Astatus 6.0 -1
y op1 A1 2.0 -1
y op1 A2 1 -1
y op1 Bstatus 10.0 -1
y op1 B1 1.5 -1
y op1 B2 0.75 -1
### Task ResourceA
s Astatus 0.2 -1
s A1 0.1 -1
Z A1 0.4 -1
s A2 0.05 -1
Z A2 0.3 -1
### Task ResourceB
s Bstatus 0.2 -1
s B1 0.1 -1
Z B1 0.4 -1
s B2 0.05 -1
Z B2 0.3 -1
-1
```

Antipattern Blob (refactored), figura 4.18

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
p P2 f
-1

T 0
t Source r request -1 P2 m 1
t Controller n op1 -1 P1
t ResourceA n A1 A2 -1 P1
t ResourceB n B1 B2 -1 P1
-1

E 0
Z request 5.0 -1
s request 0.5 -1
y request op1 1.0 -1
### Task CONTROLLER
# Invoca lo "status" prima e dopo un'operazione
s op1 0.005 -1
y op1 A1 2.0 -1
y op1 A2 1 -1
y op1 B1 1.5 -1
y op1 B2 0.75 -1
### Task ResourceA
s A1 0.1 -1
Z A1 0.4 -1
s A2 0.05 -1
Z A2 0.3 -1
### Task ResourceB
s B1 0.1 -1
Z B1 0.4 -1
s B2 0.05 -1
Z B2 0.3 -1
-1
```

Antipattern Unbalanced Processing:

"Pipe and Filter" Architecture, figura 4.20

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
-1

T 0
t Source r request -1 P1 m 2
t ResA n opA -1 P1 m 2
t ResB n opB -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA 1.0 -1
### Task ResourceA
A opA serve
### Task ResourceB
s opB 5.0 -1
-1

A ResA
s serve 0.0
s opA1 3.5
s opA2 2.0
y opA2 opB 1.0
:
serve -> opA1;
opA1 -> opA2;
opA2[opA]
-1
```

Antipattern Unbalanced Processing:

"Pipe and Filter" Architecture (refactored), figura 4.20

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
p P2 f
-1

T 0
t Source r request -1 P1 m 2
t ResA1 n opA1 -1 P2 m 2
t ResA2 n opA2 -1 P1 m 2
t ResB n opB -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA1 1.0 -1
### Task ResourceA
s opA1 3.5 -1
y opA1 opA2 1.0 -1
s opA2 2.0 -1
y opA2 opB 1.0 -1
### Task ResourceB
s opB 5.0 -1
-1
```

Antipattern Unbalanced Processing:

Extensive Processing, figura 4.22

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
-1

T 0
t Source r request -1 P1 m 5
t ResA n opA -1 P1 m 3
t ResB n opB1 opB2 -1 P1
t ResC n opC -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA 1.0 -1
### Task ResourceA
A opA serve
### Task ResourceB
s opB1 1.0 -1
Z opB1 3.0 -1
s opB2 0.05 -1
y opB1 opC 1.0 -1
y opB2 opC 1.0 -1
### Task ResourceC
s opC 0.05 -1
-1
A ResA
s serve 0.05
y opType1 opB1 1.0
y opType2 opB2 1.0
:
serve -> (0.7)opType1 + (0.3)opType2;
opType1[opA] + opType2[opA]
-1
```

Antipattern Unbalanced Processing:

Extensive Processing (refactored), figura 4.22

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
-1

T 0
t Source r request -1 P1 m 5
t ResA n opA -1 P1 m 3
t ResB1 n opB1 -1 P1
t ResB2 n opB2 -1 P1
t ResC n opC -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA 1.0 -1
### Task ResourceA
A opA serve
### Task ResourceB
s opB1 1.0 -1
Z opB1 3.0 -1
s opB2 0.05 -1
y opB1 opC 1.0 -1
y opB2 opC 1.0 -1
### Task ResourceC
s opC 0.05 -1
-1
A ResA
s serve 0.05
y opType1 opB1 1.0
y opType2 opB2 1.0
:
serve -> (0.7)opType1 + (0.3)opType2;
opType1[opA] + opType2[opA]
-1
```

Antipattern Circuitous Teasure Hunt, figura 4.24

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
-1

T 0
t Source r request -1 P1 m 5
t ResA n opA -1 P1 m 3
t ResB n infoB -1 P1
t ResC n infoC -1 P1
t ResD n infoD -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA 1.0 -1
### Task ResourceA
y opA infoB 1.0 -1
y opA infoC 1.0 -1
y opA infoD 1.0 -1
s opA 1.0 -1
### Task contenenti le informazioni
s infoB 0.005 -1
Z infoB 0.05 -1
s infoC 0.005 -1
Z infoC 0.05 -1
s infoD 0.005 -1
Z infoD 0.05 -1
-1
```

Antipattern

Circuitous Teasure Hunt (refactored), figura 4.24

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
-1

T 0
t Source r request -1 P1 m 5
t ResA n opA -1 P1 m 3
t ResB n infoB -1 P1
t ResC n infoCD -1 P1
-1

E 0
Z request 0.0 -1
s request 0.5 -1
y request opA 1.0 -1
### Task ResourceA
y opA infoB 1.0 -1
y opA infoCD 1.0 -1
s opA 1.0 -1
### Task contenenti le informazioni
s infoB 0.005 -1
Z infoB 0.05 -1
s infoCD 0.005 -1
Z infoCD 0.05 -1
-1
```

Caso di studio, modello iniziale, figura 6.3

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
p P2 f
-1

T 0
t Environment r Events -1 P1 m 2
t OtherROBOTS r Friends -1 P1 m 1
t Sensors n getEvent  -1 P2 i
t NetRX n receiveAlert -1 P2 i
t NetTX n sendAlert -1 P2
t AI n checkEvent -1 P2 m 2
t VolatileMemory n exec -1 P2
t StorageMemory n getInfo storeInfo -1 P2
t Handler n handleRegular handleDangerous -1 P2
t MoveController n goThere takeThat haltHere -1 P2
t Arms n armStatus moveSX moveDX -1 P2
t Motors n motorStatus setSpeed setDirection -1 P2
-1

E 0
### Artificial Intelligence (AI)
A checkEvent whichType
### Stimoli esterni al sistema (Environment, Robot Amici)
Z Events 2.0 -1
s Events 0.0050 -1
f Events 1 -1 y Events getEvent 1.0 -1
Z Friends 5.0 -1
s Friends 0.0050 -1
f Friends 1 -1
y Friends receiveAlert 1.0 -1
### Sensors
s getEvent 0.05 -1
f getEvent 1 -1
y getEvent checkEvent 1.0 -1
### Net
s sendAlert 0.2 -1      # NetTX
Z sendAlert 1.25 -1
s receiveAlert 0.05 -1  # NetRX
f receiveAlert 1 -1
y receiveAlert checkEvent 1.0 -1
### Storage Memory
s getInfo 0.005 -1
s storeInfo 0.0075 -1
### Volatile Memory (RAM)
s exec 0.00005 -1
```

```
### Handler
s handleRegular 2.0 -1
y handleRegular exec 8.0 -1        # Un po' di elaborazione in memoria
y handleRegular getInfo 3.0 -1     # Utilizzo delle info raccolte
y handleRegular storeInfo 1.5 -1
y handleRegular goThere 1.5 -1
y handleRegular takeThat 2.0 -1
s handleDangerous 0.1 -1           # le elaborazioni in memoria sono
y handleDangerous exec 1.0 -1      # limitate all'indispensabile
y handleDangerous getInfo 0.5 -1
y handleDangerous storeInfo 0.1 -1
z handleDangerous sendAlert 1.0 -1 # send-no-reply: l'invio dell'alert
y handleDangerous haltHere 1.0 -1  # non è assolutamente necessario
### MoveController
s goThere 0.05 -1
y goThere motorStatus 3.0 -1
y goThere setSpeed 1.5 -1
y goThere setDirection 0.75 -1
s takeThat 0.005 -1
y takeThat armStatus 4.0 -1
y takeThat moveSX 1.5 -1  # Robot mancino... perché no ;)
y takeThat moveDX 1.0 -1
s haltHere 0.0005 -1
y haltHere motorStatus 1.0 -1
y haltHere setSpeed 1.0 -1
### Arms
s armStatus 0.002 -1
Z armStatus 0.05 -1
s moveSX 0.000001 -1
Z moveSX 0.5 -1
s moveDX 0.000001 -1
Z moveDX 0.5 -1
### Motors
s motorStatus 0.002 -1
Z motorStatus 0.05 -1
s setSpeed 0.000001 -1
Z setSpeed 0.5 -1
s setDirection 0.000001 -1
Z setDirection 1.5 -1
# I tempi di servizio per moveSX, moveDX, setSpeed e setDirection
# sono fittizi ad indicare che rispetto alle altre elaborazioni
# essi sono trascurabili. Indicare un valore, anche se così basso,
# evita l'emissione di warning da parte del solver.
-1

# Activity per caratterizzare le diverse tipologie di eventi
A AI s whichType 0.0
s endElab 0.005
y memElab exec 5            # Elaborazione in memoria volatile
```

```
y memGetElab getInfo 2      # Recupera e memorizza info da
y memPutElab storeInfo 0.5  #   e verso la memoria non-volatile
f isRegular 1
y isRegular handleRegular 1
f isDangerous 1
y isDangerous handleDangerous 1
:
whichType -> memElab;
memElab -> memGetElab;
memGetElab -> memPutElab;
memPutElab -> (0.7)isRegular + (0.3)isDangerous;
isRegular + isDangerous-> endElab;
endElab[checkEvent]
-1
```

Caso di studio, modello finale, figura 6.11

```
G "" 1.0E-5 50 0 0.9 -1

P 0
p P1 f
p P2 f
-1

T 0
t Environment r Events -1 P1 m 2
t OtherROBOTS r Friends -1 P1 m 1
t Sensors n getEvent  -1 P2 i
t NetRX n receiveAlert -1 P2 i
t NetTX n sendAlert -1 P2
t AI n checkEvent -1 P2 m 2
t VolatileMemory n exec -1 P2
t StorageMemory n getInfo storeInfo -1 P2
t Handler n handleRegular -1 P2 m 2
t Handler2 n handleDangerous -1 P2
t MoveController n goThere takeThat haltHere -1 P2
t Arms n moveSX moveDX -1 P2
t Motors n setSpeed setDirection -1 P2
-1

E 0
### Artificial Intelligence (AI)
A checkEvent whichType
### Stimoli esterni al sistema (Environment, Robot Amici)
Z Events 2.0 -1
s Events 0.0050 -1
f Events 1 -1 y Events getEvent 1.0 -1
Z Friends 5.0 -1
```

```
s Friends 0.0050 -1
f Friends 1 -1
y Friends receiveAlert 1.0 -1
### Sensors
s getEvent 0.05 -1
f getEvent 1 -1
y getEvent checkEvent 1.0 -1
### Net
s sendAlert 0.2 -1      # NetTX
Z sendAlert 1.25 -1
s receiveAlert 0.05 -1  # NetRX
f receiveAlert 1 -1
y receiveAlert checkEvent 1.0 -1
### Storage Memory
s getInfo 0.005 -1
s storeInfo 0.0075 -1
### Volatile Memory (RAM)
s exec 0.00005 -1
### Handler
s handleRegular 2.0 -1
y handleRegular exec 8.0 -1        # Un po' di elaborazione in memoria
y handleRegular getInfo 3.0 -1     # Utilizzo delle info raccolte
y handleRegular storeInfo 1.5 -1
y handleRegular goThere 1.5 -1
y handleRegular takeThat 2.0 -1
s handleDangerous 0.1 -1           # le elaborazioni in memoria sono
y handleDangerous exec 1.0 -1      # limitate all'indispensabile
y handleDangerous getInfo 0.5 -1
y handleDangerous storeInfo 0.1 -1
z handleDangerous sendAlert 1.0 -1 # send-no-reply: l'invio dell'alert
y handleDangerous haltHere 1.0 -1  # non è assolutamente necessario
### MoveController
s goThere 0.05 -1
y goThere setSpeed 1.5 -1
y goThere setDirection 0.75 -1
s takeThat 0.005 -1
y takeThat moveSX 1.5 -1  # Robot mancino... perché no ;)
y takeThat moveDX 1.0 -1
s haltHere 0.0005 -1
y haltHere setSpeed 1.0 -1
### Arms
s moveSX 0.000001 -1
Z moveSX 0.4 -1
s moveDX 0.000001 -1
Z moveDX 0.4 -1
### Motors
s setSpeed 0.000001 -1
Z setSpeed 0.4 -1
s setDirection 0.000001 -1
```

```
Z setDirection 1.4 -1
# I tempi di servizio per moveSX, moveDX, setSpeed e setDirection
# sono fittizi ad indicare che rispetto alle altre elaborazioni
# essi sono trascurabili. Indicare un valore, anche se così basso,
# evita l'emissione di warning da parte del solver.
-1

# Activity per caratterizzare le diverse tipologie di eventi
A AI s whichType 0.0
s endElab 0.005
y memElab exec 5            # Elaborazione in memoria volatile
y memGetElab getInfo 2      # Recupera e memorizza info da
y memPutElab storeInfo 0.5  #   e verso la memoria non-volatile
f isRegular 1
y isRegular handleRegular 1
f isDangerous 1
y isDangerous handleDangerous 1
:
whichType -> memElab;
memElab -> memGetElab;
memGetElab -> memPutElab;
memPutElab -> (0.7)isRegular + (0.3)isDangerous;
isRegular + isDangerous-> endElab;
endElab[checkEvent]
-1
```

Appendice B

Sorgenti Tool di Supporto

Seguono i sorgenti del tool di supporto, nella versione attuale, descritto nel capitolo 5 di questo lavoro di tesi.

garfield.php

```
<?php
### Config
$solver = "/usr/local/bin/lqns";
$workdir = "/home/mrfree/tool_lqnperf/";
### /Config

### Info block (these infos should be in an external file to parse)
$notcloneable_resources = array();
  // "subsys_name" => "array contains the name of resources"
$subsystems = array("sub1" => array("ResA", "ResB", "ResC"));
  // "subsys_name" => ({service_time | throughput} => value;target_element)*
  //   'target_element' is the task or entry where the value will be taken
$reqs = array( "sub1" => array( "service_time" => "5;opType2",
                                "throughput" => "0.4;Source" ) );
### /Info block

require_once("useful_things.php");
require_once("antipatterns_profiles.php");
require_once("graph.php");

define("VERBOSE", true);
define("KEEP_USELESS_STEPS_FILE", true);

# Usage: php mr_tool.php <orig_model.xml>
#
# Note: LQN model MUST be written in XML format!
if($_SERVER["argc"] > 2 || $_SERVER["argc"] <= 1)
   die ("Error: Wrong arguments!\n");
# $_SERVER['argv'][1] should contains model's filename

# Create a copy of the original model in the workdir as step0
$orig_model_filename = basename($_SERVER["argv"][1], ".xml");
workdirCleanUp($orig_model_filename);
$step0_model_fullpath = $workdir.$orig_model_filename."_step0.xml";
if (!copy($_SERVER["argv"][1], $step0_model_fullpath))
   die ("Error: Failed to copy '" . basename($_SERVER["argv"][1])
        . "' file in the workdir\n");

$xml_prev_model = null;
$xml_prev_model_fullpath = null;
$xml_current_model = DOMDocument::load($step0_model_fullpath);
$xml_current_model_fullpath = $step0_model_fullpath;

// First of all...
solveCurrentModel();

if( !lookupTable_sys() ){
   echo "\n\nSome Requirements aren't Satisfied!\n";
   echo "--- I didn't solve the problem :(\n";
} else {
   echo "\n\nALL Requirements are Satisfied!\n";
}

//GnuPlot
createSummaryGraph();
//LQN2PS
createModelGraph();
?>
```

useful_things.php

```
<?php
function workdirCleanUp($model_name){
   global $workdir;

   $command = sprintf("rm -f %s*",$workdir.$model_name);
   exec($command, $output, $exitcode);
   if ($exitcode != 0)
      die (sprintf("Error: %s\n",print_r($output)));
}

function solveCurrentModel() {
   global $solver, $xml_current_model_fullpath, $xml_current_model;

   // Save the actual LQN model version to the original XML file
   if( !$xml_current_model->save($xml_current_model_fullpath) ){
      die ("Error: Failed to save current model to '"
           . basename($xml_current_model_fullpath) . "' file in the workdir\n");
   }

   // Solve the current LQN model
   $command = $solver." ".$xml_current_model_fullpath;
   exec($command, $output, $exitcode);
   if ($exitcode != 0)
      die ("Error: " . print_r($output));

   $xml_current_model = DOMDocument::load($xml_current_model_fullpath);
}

function getLastStepInfos(){
   global $xml_current_model_fullpath, $workdir;
   $output = array();

   if( preg_match("/(.*)_step([0-9]+)/",basename($xml_current_model_fullpath,
       ".xml"), $matches) ){
      # Array(
      #       [0] => Extensive_step0
      #       [1] => Extensive
      #       [2] => 0
      # )
      $step_name = $matches[1];
   } else die ("Error: Something goes wrong!\n");

   $command = sprintf("ls %s_step*.xml",$workdir.$step_name);
   exec($command, $output, $exitcode);
   if ($exitcode != 0)
      die (sprintf("Error: %s\n",print_r($output)));

   $last_used_number = 0;
   foreach($output as $step_filename){
      if( preg_match("/(.*)_step([0-9]+).xml/", $step_filename, $matches) ){
         if($matches[2] > $last_used_number) $last_used_number = $matches[2];
      } else die ("Error: Something goes wrong!\n");
   }

   return array($step_name,$last_used_number);
}

function goToNextStep(){
   global $xml_prev_model_fullpath, $xml_prev_model;
   global $xml_current_model_fullpath, $xml_current_model;

   // Save the actual LQN model version to the original XML file
   if( !$xml_current_model->save($xml_current_model_fullpath) ){
      die ("Error: Failed to save current model to '"
```

```
            . basename($xml_current_model_fullpath) . "' file in the workdir\n");
    }

    list($new_step_name, $new_step_number) = getLastStepInfos();
    // Workaround: the $new_step_number contains the last used number
    $new_step_number++;

    $new_step_fullpath = dirname($xml_current_model_fullpath)."/".$new_step_name
                                 ."_step".$new_step_number.".xml";
    if (!copy($xml_current_model_fullpath, $new_step_fullpath))
       die ("Error: Failed to copy '" . basename($xml_current_model_fullpath)
            . "' file in the workdir\n");

    // Move current to previous
    $xml_prev_model_fullpath = $xml_current_model_fullpath;
    $xml_prev_model = DOMDocument::load($xml_current_model_fullpath);

    $xml_current_model_fullpath = $new_step_fullpath;
    $xml_current_model = DOMDocument::load($new_step_fullpath);

    if(VERBOSE)
       printf("== Some requirements aren't Satisfied...\n");
       printf("=== Step %s created\n",$new_step_number);
}

function undoLastStep(){
    global $xml_prev_model_fullpath, $xml_prev_model;
    global $xml_current_model_fullpath, $xml_current_model;

    // Delete current XML model
    if(!KEEP_USELESS_STEPS_FILE)
       unlink($xml_current_model_fullpath);

    // Rollback...
    $xml_current_model_fullpath = $xml_prev_model_fullpath;
    $xml_current_model = DOMDocument::load($xml_prev_model_fullpath);

    $xml_prev_model_fullpath = null;
    $xml_prev_model = null;

    if(VERBOSE)
       printf("=== Rollback: The last Step was deleted!\n\n");
}

function areChangesUseful(){
    global $reqs, $xml_prev_model_fullpath;
    $perf_variations = array();

    foreach($reqs as $name => $requirements_array){
       $subsys2check[] = $name;
    }

    foreach($subsys2check as $current_subsys_name){
       foreach($reqs[$current_subsys_name] as $req_type => $req){
          list($req_value, $req_where) = explode(";", $req);

          $current_perf = getPerformance($req_where);
          $previous_perf = getPerformance($req_where, $xml_prev_model_fullpath);

          if( $current_perf[$req_type] == -1 || $previous_perf[$req_type] == -1 ||
              !($req_type == "service_time" || $req_type == "throughput")
            )
             die(sprintf("Error: Problems with requirements array.
                          Subsystem '%s'!\n", $current_subsys_name));

          $variation = ($current_perf[$req_type]-$previous_perf[$req_type])/$req_value;
          $perf_variations[$current_subsys_name][$req_type] = $req_where.";".$variation;
       }
    }
```

```
    if(VERBOSE){
       printf("Performance Variations:\n");
       foreach($subsys2check as $current_subsys_name){
          foreach($perf_variations[$current_subsys_name] as $req_type=>$variation){
             printf("  %s (%s)\n",$variation,$req_type);
          }
       }
    }

    // Rule: positive values are good negative ones are bad
    $ago_della_bilancia = 0;
    foreach($subsys2check as $current_subsys_name){
       foreach($perf_variations[$current_subsys_name] as $req_type=>$variation){
          list($req_where, $req_variation) = explode(";", $variation);
          switch ($req_type) {
             case "service_time":
                // In this case a positive value is bad!
                $ago_della_bilancia += -($req_variation);
                break;
             default:
                $ago_della_bilancia += $req_variation;
                break;
          }
       }
    }
    if( $ago_della_bilancia > 0 ){
       return true;
    } else {
       // If changes aren't totally positive, but they help with
       // some requirements... they're ok ;)
       return areRequirementsSatisfied() ? true : false
    }
}

function areRequirementsSatisfied($subsys_name = null){
    global $reqs;
    $subsys2check = array();
    // If no req is defined --> true
    $all_satisfied = true;

    if( $subsys_name == null ){
       foreach($reqs as $name => $requirements_array){
          $subsys2check[] = $name;
       }
    } else {
       $subsys2check[] = $subsys_name;
    }

    foreach($subsys2check as $current_subsys_name){
       foreach($reqs[$current_subsys_name] as $req_type => $req){
          list($req_value, $req_where) = explode(";", $req);

          $current_perf = getPerformance($req_where);

          switch ($req_type){
             case "service_time":
                if( $current_perf["service_time"] != -1 ){
                   if( $current_perf["service_time"] > $req_value )
                      $all_satisfied = false;
                } else die (sprintf("Error: I haven't any service time info for
                                    '%s'\n", $req_where));
                break;
             case "throughput":
                if( $current_perf["throughput"] != -1 ) {
```

```
                    if( $current_perf["throughput"] < $req_value )
                       $all_satisfied = false;
                 } else die (sprintf("Error: I haven't any throughput info for
                                     '%s'\n", $req_where));
                 break;
              default:
                 die ("Error: Unknown requirement type!");
           }
        }
     }

     return $all_satisfied;
  }

  function getPerformance($target_name, $xml_model_fullpath = null){
     global $xml_current_model;
     $output = array();

     if( $xml_model_fullpath == null ){
        // First of all solve current LQN model
        solveCurrentModel();
        $xml_model = $xml_current_model;
     } else {
        // This is an already solved LQN model
        $xml_model = DOMDocument::load($xml_model_fullpath);
     }

     $xpath = new DOMXPath($xml_model);
     $query = "//*[@name='". $target_name ."']";
     $target = $xpath->query($query); // This must return only 1 node (in a NodeList)!

     switch($target->item(0)->tagName){
        case "task":
           $results = $xpath->query("result-task", $target->item(0));
           $output["throughput"] = $results->item(0)->getAttribute("throughput");
           $multiplicity = ($target->item(0)->getAttribute("multiplicity")) ?
                              $target->item(0)->getAttribute("multiplicity") : 1;
           $output["utilization"] = $results->item(0)->getAttribute("utilization")
                                    / $multiplicity;
           // This infos can't be taken directly from here (it should be calculated)
           $output["service_time"] = -1; // it means "don't care"
           $output["throughput_bound"] = ($output["service_time"] != -1) ?
                                          (1/$output["service_time"]) : -1;
           break;
        case "entry":
           switch($target->item(0)->getAttribute("type")){
              case "NONE":
                 $results = $xpath->query("result-entry", $target->item(0));
                 $output["throughput"] = $results->item(0)->getAttribute("throughput");
                 $output["throughput_bound"] =
                    $results->item(0)->getAttribute("throughput-bound");
                 $output["service_time"] =
                    $results->item(0)->getAttribute("phase1-service-time");
                 $output["utilization"] = -1; // Probably not useful
                 break;
              case "PH1PH2":
                 $results = $xpath->query("result-entry", $target->item(0));
                 $output["throughput"] = $results->item(0)->getAttribute("throughput");
                 $output["throughput_bound"] =
                    $results->item(0)->getAttribute("throughput-bound");
                 $output["utilization"] = -1; // Probably not useful
                 // Service Time info is on phase1
                 $results_ph1 = $xpath->query("entry-phase-activities/activity[@name='"
                                 .$target_name."_ph1']/result-activity",$target->item(0));
```

```
                $output["service_time"] =
                   $results_ph1->item(0)->getAttribute("service-time");
                break;
            }
            break;
        case "activity":
            $results = $xpath->query("result-activity", $target->item(0));
            $output["throughput"] = $results->item(0)->getAttribute("throughput");
            $output["service_time"] = $results->item(0)->getAttribute("service-time");
            // This info can't be taken directly from here
            $output["throughput_bound"] = -1;
            $output["utilization"] = -1; // Probably not useful
            break;
    }

    return $output;
}

function lookupTable_sys(){
    global $subsystems;

    if( !areRequirementsSatisfied() ){
        foreach( $subsystems as $subsys_name => $subsys_resources){
            lookupTable_subsys_type2($subsys_name);
        }
    }

    if( areRequirementsSatisfied() )
        return true;
    else
        return false;
}

function lookupTable_subsys_type2($subsys_name){
    $no_more_actions = false;

    while( !areRequirementsSatisfied($subsys_name) && !$no_more_actions ){
        if( !findAndSolveAnAntipattern($subsys_name) ){
            // Any known antipattern were found
            if( !lookupTable_res_sw($subsys_name) ){
                $no_more_actions = true;
            }
        }
    }
}

function resourceCmp($a, $b){
    if ($a["utilization"] == $b["utilization"]) {
        return 0;
    }
    return ($a["utilization"] < $b["utilization"]) ? -1 : 1;
}

function lookupTable_res_sw($subsys_name){
    global $notcloneable_resources, $subsystems;
    $resources = array();
    // Thresholds (1 = 100%)
    define("UTILIZATION_THRESHOLD", 0.7);

    foreach( $subsystems[$subsys_name] as $resource_name ){
        $resources[$resource_name] = getPerformance($resource_name);
    }
    // Sort resources by utilization (reverse order so [0] is the highest)
    uasort($resources, "resourceCmp");
    $resources = array_reverse($resources);
```

```
    reset($resources); // Probably not needed ;)

    foreach( $resources as $resource_name => $resource_perf){
       if( $resource_perf["utilization"] >= UTILIZATION_THRESHOLD ){
          if( !in_array($resource_name, $notcloneable_resources) ){
             // store the actual LQN model in its file and create a new one
             goToNextStep();

             // Clone the most used software resource
             cloneResource($resource_name);

             if( !areChangesUseful() ){
                // undo last model changes
                undoLastStep();
             } else {
                return true;
             }
          } elseif(VERBOSE){
             printf("Resource %s isn't cloneable\n",$resource_name);
          }
       } else {
          // GREEN - Nothing to do
          return false;
       }
    }
    return false;
}

function findAndSolveAnAntipattern($subsys_name){
    $selection = -2;
    $found = array();

    if(VERBOSE)
       printf("Searching for known antipattern in SubSystem '%s'...\n",$subsys_name);

    //Searching phase...
    if( extensive_processing($subsys_name, true) )
       $found[0] = "Extensive Processing";
    //Example:
    //if( some_antipattern($subsys_name, true) )
    //      $found[k] = "Some Antipattern";

    //Interaction with designers...
    if( count($found) > 0 ){
       echo sprintf("   Possible antipatterns in '%s' subsystem:\n",$subsys_name);
       for($i=0; $i < count($found); $i++){
          echo sprintf("    (%d) %s\n", $i, $found[$i]);
       }
       while( $selection < -1 || $selection >= count($found) ){
          echo "   Selection (-1 to skip): ";
          fscanf(STDIN, "%d\n", $selection);
       }

       //Fixing phase...
       switch ($selection){
          case 0:
             // Extensive Processing
             return (extensive_processing($subsys_name));
             break;
          case -1:
             if (VERBOSE)
                printf("   Skipping...\n\n");
             return false;
             break;
       }
```

```
    } elseif (VERBOSE) {
       printf("   No known antipattern found\n\n",$subsys_name);
    }
}

function cloneResource($resource_name){
    global $xml_current_model;

    $xpath = new DOMXPath($xml_current_model);
    $query = "//task[@name='". $resource_name ."']";
    $target = $xpath->query($query); // This must return only 1 node (in a NodeList)!

    // raise task multiplicity
    $resource_task = $target->item(0);
    $current_multiplicity = ($resource_task->getAttribute("multiplicity")) ?
                                $resource_task->getAttribute("multiplicity") : 1;
    $resource_task->setAttribute("multiplicity", $current_multiplicity + 1);
    if(VERBOSE)
       printf("Resource %s:\n  raise multiplicity from %s to %s\n",
               $resource_name,$current_multiplicity,$current_multiplicity+1);
}
?>
```

antipatterns_profiles.php

```
<?php
function getResourceServices($resource_name){
    global $xml_current_model;

    $xpath = new DOMXPath($xml_current_model);
    $query = "//*[@name='". $resource_name ."']/entry";
    $target = $xpath->query($query);

    foreach($target as $services){
       //TODO: Here we can add useful infos for each service
       //      like service time, think time and so on
       $name = $services->getAttribute("name");
       $output[$name] = array();
    }

    return $output;
}

function extensive_processing($subsys_name, $search_only = false){
    global $subsystems, $xml_current_model;
    // Thresholds (1 = 100%)
    define("TARGET_UTILIZATION_THRESHOLD", 0.7);

    foreach( $subsystems[$subsys_name] as $resource_name ){
       $resources[$resource_name] = getPerformance($resource_name);
    }
    // Sort resources by utilization (reverse order so [0] is the highest)
    uasort($resources, "resourceCmp");
    $resources = array_reverse($resources);
    reset($resources); // Probably not needed ;)

    foreach( $resources as $resource_name => $resource_perf){
       if( $resource_perf["utilization"] >= UTILIZATION_THRESHOLD &&
           count($services = getResourceServices($resource_name)) > 1 ){
          if( !$search_only ){
             // Fix
```

```
            // store the actual LQN model in its file and create a new one
            goToNextStep();

            $num = 1;
            $xpath = new DOMXPath($xml_current_model);
            $query = "//*[@name='". $resource_name ."']";
            $current_task_node = $xpath->query($query)->item(0);
            $current_processor_node = $current_task_node->parentNode;
            array_pop($services);
            foreach($services as $service_name => $service_infos){
               $query = "//*[@name='". $service_name ."']";
               $service_node = $xpath->query($query)->item(0);
               $new_task = $xml_current_model->createElement("task", " ");
               $new_task_name = $current_task_node->getAttribute("name").++$num;
               $new_task->setAttribute("name", $new_task_name);
               $new_task->setAttribute("scheduling",
                  $current_task_node->getAttribute("scheduling"));
               $new_task->appendChild($service_node);
               $current_processor_node->appendChild($new_task);
               $subsystems[$subsys_name][] = $new_task_name;
            }
            if(VERBOSE)
               printf("   Extensive Processing antipattern was removed\n\n");
         }
         return true;
      }
   }
   return false;
}
?>
```

graph.php

```
<?php
### Config
$gnuplot = "/usr/bin/gnuplot";
$lqn2ps = "/usr/local/bin/lqn2ps";
$lqn2ps_options = "-i -j -l -q +r +t -w -X 9,120 -Y 54,60";
### /Config

// results.res contains result info for each requirement (1 row per step)
$result_file_fullpath = $workdir."results.res";
$gnuplot_template_file = dirname($_SERVER['PHP_SELF']).'/misc/gnuplot_template.gnu";
$gnuplot_file = $workdir."summary.gnu";

function createSummaryGraph(){
   global $reqs, $gnuplot, $gnuplot_template_file, $gnuplot_file, $workdir;
   global $xml_current_model_fullpath;
   $plot_template = "plot '".$workdir."results.res' using 1:%s title 'measured'
                     lw 2, %s title 'required' lw 2";

   if( VERBOSE )
      printf("\nCreating Summary Graphs (using gnuplot)...\n");

   // First of all...
   createResultFile();

   if (!copy($gnuplot_template_file, $gnuplot_file))
      die (sprintf("Error: Failed to copy '%s' file in the workdir\n",
                   $gnuplot_template_file));
```

```
    if(!$handle = fopen($gnuplot_file, 'a'))
       die(sprintf("Error: Cannot open file '%s'\n",$gnuplot_file));

    // The "final model" is the one linked by $xml_current_model_fullpath
    if( preg_match("/(.*)_step([0-9]+)/", basename($xml_current_model_fullpath, ".xml"),
          $matches) ) {
       $final_step_number = $matches[2];
       if( fwrite($handle, "set xlabel 'Step (Final Model in Step ".$final_step_number
                           .")'\n\n") === FALSE )
          die(sprintf("Error: Cannot write to file '%s'\n",$gnuplot_file));
    } else die ("Error: Something goes wrong!\n");

    $pos = 2;
    foreach($reqs as $subsys_name => $subsys_reqs){
       foreach($subsys_reqs as $req_type => $req){
          list($req_value, $req_where) = explode(";", $req);

          if(fwrite($handle,"!rm graph_".$req_where.".ps &> /dev/null\n") === FALSE ||
             fwrite($handle,"set output '".$workdir."/graph_".$req_where.".ps'\n")===FALSE||
             fwrite($handle,"set ylabel '".$req_where." (".$req_type.")'\n") === FALSE ||
             fwrite($handle,sprintf($plot_template,$pos,$req_value)."\n\n") === FALSE)
                die(sprintf("Error: Cannot write to file '%s'\n",$gnuplot_file));

          $pos++;
       }
    }
    fclose($handle);

    $command = $gnuplot." ".$gnuplot_file;
    exec($command, $output, $exitcode);
    if ($exitcode != 0)
       die ("Error: ".print_r($output));
}

function createResultFile(){
    global $reqs, $workdir;
    global $result_file_fullpath;

    list($last_step_name, $last_step_number) = getLastStepInfos();
    for($i=0; $i <= $last_step_number; $i++){
       $steps_fullpath[$i] = $workdir.$last_step_name."_step".$i.".xml";
    }

    foreach($steps_fullpath as $i => $step_fullpath){
       //Per ogni Step!
       foreach($reqs as $subsys_name => $subsys_reqs){
          foreach($subsys_reqs as $req_type => $req){
             list($req_value, $req_where) = explode(";", $req);
             $perf = getPerformance($req_where, $step_fullpath);
             $step_perf[$i][] = $req_where.";".$req_value.";".$perf[$req_type];
          }
       }
    }
    // $step_perf example structure:
    /* Array(
     *      [0] => Array(
     *                      [0] => opType2;5;8.67504e+00
     *                 )
     *      [1] => Array(
     *                      [0] => opType2;5;3.62585e+00
     *                 )
     * )
     */

    if(!$handle_result = fopen($result_file_fullpath, 'w'))
```

```
        die(sprintf("Error: Cannot open file '%s'\n",$result_file_fullpath));

    foreach($step_perf as $current_step => $current_step_infos){
        $perf_info = "";
        foreach($current_step_infos as $infos){
            list($where, $req_value, $curr_value) = explode(";", $infos);
            $perf_info .= "$curr_value ";
        }
        if(fwrite($handle_result, $current_step." ".$perf_info."\n") === FALSE)
            die(sprintf("Error: Cannot write to file '%s'\n",$result_file_fullpath));
    }

    fclose($handle_result);
}

function createModelGraph(){
    global $lqn2ps, $lqn2ps_options, $workdir;

    if( VERBOSE )
        printf("\nCreating Models Graphs (using lqn2ps)...\n");

    list($last_step_name, $last_step_number) = getLastStepInfos();

    for($i=0; $i <= $last_step_number; $i++){
        $step_fullpath = $workdir.$last_step_name."_step".$i.".xml";
        $output_fullpath = sprintf("%s/%s.ps",dirname($step_fullpath),
                                   basename($step_fullpath,".xml"));
        $command = sprintf("%s %s -O ps -o %s %s",$lqn2ps,$lqn2ps_options,
                           $output_fullpath,$step_fullpath);

        exec($command, $output, $exitcode);
        if ($exitcode != 0)
            die ("Error: ".print_r($output));
     }
}
?>
```

misc/gnuplot_template.gnu

```
set term pos por dashed "Helvetica" 14
set size 1,0.5
set autoscale x
set autoscale y
set data st lines
set nokey

set term postscript color
set border 3
set xtics nomirror 1
set ytics nomirror

# Here tool output commands...
```

Ringraziamenti

É difficile in poche righe ricordare e nel contempo ringraziare tutte le persone che in qualche modo mi hanno supportato, sopportato e che in generale hanno reso la mia esperienza universitaria migliore di come magari poteva essere. Ci provo comunque...

In primo luogo voglio sicuramente ringraziare la mia famiglia, tutta. Devo a loro buona parte di quello che sono, nel bene e nel male ;)

Sono grato in particolare ai miei genitori per avermi dato delle straordinarie lezioni di vita durante questi anni, sono state per me indispensabili ed insostituibili e probabilmente continueranno ad esserlo ancor di più in futuro; a mia sorella Lea per il suo ineguagliabile affetto e per il tentativo, ahimè per ora fallito, di istruirmi nella riproduzione di strane sequenze di movimenti, intuitivamente modellabili e cicliche, che pare vengano definite "balli caraibici".

Gli amici e le amiche... un pensiero va a tutti coloro, universitari e non, con i quali ho condiviso molte piacevoli serate in questi ultimi anni: siano esse trascorse di fronte ad una qualche bevanda moderatamente alcolica a fare quattro chiacchiere o a casa di qualcuno per un'arrostata o un LAN-Party.

Non posso però non rivolgere un ringraziamento speciale agli amici di sempre come Luca “il comp”, Massimo “il maestro”, Francesco e Martina con i quali condivido, da diversi anni, buona parte della porzione “divertente” del mio tempo libero; a Francesco “kireime” e Angelo “lo zio” -senza peraltro dimenticare la sua VOX semiacustica del ’67- con i quali ho il piacere di condividere diversi interessi tra cui l’informatica in senso lato e la musica; all’amico e collega Alessandro, anche lui ex-petrinetter, e Raffaella dimostratasi in più di un’occasione ottima pizzaiola; ai miei colleghi ed ex-colleghi di lavoro Davide, Giovanni “giolod”, Lara “ricciola”, Alessio, Simonetta, Maura, Pina, Vittorio e Ugo che mi hanno confermato che lavorare bene e in maniera piacevole si può; ai sostenitori e sviluppatori di software libero che oltre ad avermi fornito eccellenti strumenti di lavoro e di studio, sia in ambito accademico che personale/professionale, mi hanno offerto interessanti spunti di riflessione in ambito informatico e non; ai blogger che a volte mi intrattengono con improbabili disquisizioni; ai ragazzi e alle ragazze dei forum che frequento abitualmente e con i quali scambio volentieri consigli ed opinioni.

Infine, ma solo per una questione cronologica, rivolgo un grazie al Prof. Cortellessa che con simpatia e grande professionalità mi ha guidato durante tutto il mio lavoro di tesi.

Grazie.

Bibliografia

[1] C. Smith, L. Williams (2001) *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, 1st edition, Addison-Wesley Professional

[2] C. Smith, M. Woodside (1999) *Performance Validation at Early Stages of Software Development*, in E. Gelenbe ed., System Performance Evaluation: Methodologies and Applications, CRC Press

[3] V. Cortellessa, A. Di Marco, P. Inverardi (2003) *Three performance models at work: a software designer perspective*, Proc. of FOCLASA

[4] A. Di Marco (2005) *Model-based Performance Analysis of Software Architectures*, Ph.D. Thesis, Università degli Studi dell'Aquila

[5] I. Sommerville (2000) *Software Engineering*, 6th edition, Addison Wesley

[6] AA.VV. *Open source software development method*, Wikipedia http://en.wikipedia.org/wiki/Open_source_software_development_method

[7] P. Sancho, C. Juiz, R. Puigjaner (2005) *Automatic Performance Evaluation and Feedback for MASCOT designs*, Proc. of the 5th international workshop on Software and performance

[8] L. Williams, C. Smith (2002) *PASA: An Architectural Approach to Fixing Software Performance Problems*, Proc. of CMG international conference

[9] C. Smith, L. Williams (2003) *Software Performance Engineering*, in UML for Real: Design of Embedded Real-Time Systems, Luciano Lavagno, Grant Martin, Bran Selic ed., Kluwer Academic Publishers

[10] T. Parsons (2005) *A Framework for Detecting Performance Design and Deployment Antipatterns in Component Based Enterprise Systems*, Proc. of the 2nd international doctoral symposium on Middleware

[11] S. Barber, T. Graser, J. Holt (2002) *Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation*, Proc. of the 17th IEEE international conference on Automated software engineering

[12] E. Lazowska et al. (1984) *Quantitative System Performance - Computer System Analysis Using Queueing Network Models*, Prentice-Hall Inc.

[13] C. Smith, L. Williams (2000) *Software Performance AntiPatterns*, Proc. 2nd international workshop on Software and Performance

[14] C. Smith, L. Williams (2002) *New Software Performance AntiPatterns: More Way to Shoot Yourself in the Foot*, Proc. of CMG international conference

[15] C. Smith, L. Williams (2003) *More New Software Performance AntiPatterns: Even More Ways to Shoot Yourself in the Foot*, Proc. of CMG international conference

[16] J. Neilson, C. Woodside, D. Petriu, S. Majumdar (1994) *Software Bottlenecking in Client-Server System and Rendezvous Networks*, Tech. Report, Carleton University

[17] M. Woodside, G. Franks (2005) *Tutorial Introduction to Layered Modeling of Software Performance*, Tech. Report, Department of Systems and Computer Engineering, Carleton University, http://www.sce.carleton.ca/rads

[18] G. Franks et al. (2005) *Layered Queueing Network Solver and Simulator User Manual*, Tech. Report, Department of Systems and Computer Engineering, Carleton University, http://www.sce.carleton.ca/rads

[19] Performance Evaluation Group *GreatSPN User's Manual*, Version 2.0.2, Tech. Report, Dipartimento di Informatica, Università di Torino

[20] G. Gu, D. Petriu (2002) *XSLT transformation from UML models to LQN performance models*, Proc. of the 3rd international workshop on Software and performance

[21] OGM - Object Management Group (2003) *UML Profile for Schedulability, Performance, and Time Specification*, version 1.0 - formal/03-09-01

[22] J. Xu, M. Woodside, D. Petriu (2003) *Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time*, Proc. of the 13th international conference on Computer Performance Evaluation, Modelling Techniques and Tools

[23] L. Dobrzanski, L. Kuzniarz (2006) *An Approach to Refactoring of Executable UML Models*, Proc. of ACM symposium on Applied computing

[24] AA.VV. *Artificial neural network*, Wikipedia, http://en.wikipedia.org/wiki/Artificial_neural_network