Politecnico di Milano

Facoltà di Ingegneria dell'informazione

Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Tecniche di resource allocation per sistemi virtualizzati di larga scala

RELATORE: Ing. Danilo Ardagna

CORRELATORE: Dott. Barbara Panicucci

Tesi di Laurea di:

Marco Caldirola, Matr. 733270

Anno Accademico 2009–2010

All'amore silenzioso e incondizionato che solo dei genitori sanno donare ...

Ringraziamenti

Giunto al termine di questo lavoro desidero ringraziare ed esprimere la mia riconoscenza nei confronti di tutte quelle persone che, in modi diversi, mi sono state vicine e hanno permesso e incoraggiato sia i miei studi che la realizzazione e stesura di questa tesi. In particolar modo desidero ringraziare i miei genitori, Sergio e Silvana, a cui sarò per sempre riconoscente sia per i tanti sacrifici affrontati per permettermi di proseguire i miei studi, sia per quell'amore tacito venuto dal loro cuore che mi ha permesso di raggiungere questa meta. Ricordo quei momenti difficili durante i quali con una pacca sulle spalle mi hanno fatto capire che loro erano lì, pronti ad aiutarmi. E' ad essi che dedico questo traguardo.

Desidero ringraziare i miei nonni, Luigi e Antonietta, e mia zia Angela che, anche se ora mi guardano dall'alto, sono certo saranno orgogliosissimi. Un pensiero voglio dedicarlo anche ad Alessandro e mio zio Roberto poiché so che scendendo le scale c'è qualcuno pronto a regalarmi un sorriso.

Ringrazio mio fratello Simone, poiché in tutti questi anni ha sopportato le miei esigenze di studio, i miei nervosismi per gli esami, i turni sul computer, ... ma anche perché con le sue critiche mi ha aiutato a crescere.

Ringrazio il prof. Danilo Ardagna poiché in questi mesi fianco a fianco ha saputo donarmi consigli e chiarimenti che mi hanno permesso di arrivare fino in fondo. Voglio ringraziarlo anche per tutti gli insegnamenti che ha saputo darmi e che hanno permesso di accrescere la stima che già avevo per lui.

Un pensiero ed un grazie vanno anche a Alessandro Busi, Ervis Suljoti, Stefano Vettor, Francesco Nigro, Patrick Ricciolo ed anche a tutti coloro che hanno condiviso con me il sogno di essere ingegnere. In particolare voglio ringraziare Marco Casiero con il quale ho condiviso momenti di studio, ansie e gioie in questi ultimi anni e Gandini Alessandro con il quale mi sono spesso confrontato nel periodo di preparazione della tesi. Voglio dedicare un ringraziamento speciale a Alessandro Caio, con cui ho condiviso questa avventura e con cui ho costruito un legame vero anche al difuori dei momenti di studio. Mi ha supportato, incoraggiato e preso su di sé le mie ansie. Sempre pronto al confronto e a guidarmi nelle scelte da intraprendere.

Un affettuoso ringraziamento va a mio cugino Luca con il quale in tutti questi anni ho sempre avuto momenti di confronto e con cui ho condiviso la mia curiosità verso il mondo IT. Sempre disponibile per due chiacchiere ed una birretta. Voglio ringraziare i miei amici Giovanni Mauri e Daniele Biella poiché, anche se abbiamo intrapreso diverse strade, dopo molto tempo mi fanno capire che per me ci sono sempre. Un pensiero va anche a Luca Xodo e Paolo Candeloro, perché nei momenti di crisi hanno saputo regalarmi sorrisi e conforto.

Un grazie va anche a Massimo Ghezzi e Chiara Brivio che oltre a condividere con me la passione per lo snowboard mi hanno regalato serate spensierate di sano divertimento. Grazie per avermi fatto da "maestri", per avermi ospitato, per avermi supportato senza mai chiedermi nulla in cambio.

Infine, vorrei ringraziare di cuore Irene Sala con cui in questi anni ho vissuto una bellissima avventura. Mi ha sopportato quando ero nervoso, mi ha calmato quando ero agitato, mi ha sorriso quando ero triste, mi ha incoraggiato quando mi sentivo abbattuto ma soprattutto ha saputo amarmi quando avevo bisogno d'amore. Ho passato con lei dei momenti stupendi che non potrò mai dimenticare.

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

L'obiettivo del presente lavoro di tesi è sviluppare algoritmi gerarchici per l'allocazione delle risorse in Service Center di grandi dimensioni. Il problema dell'allocazione delle risorse è un problema NP-difficile. In letteratura sono state proposte delle soluzioni centralizzate per la sua soluzione. Questi algoritmi hanno però dei problemi di scalabilità e consentono di risolvere con tempi di risoluzione accettabili solo piccole istanze che considerano al più 400 server fisici e un migliaio di VM. Tuttavia, i Service Center odierni sono caratterizzati da dimensioni di un ordine di grandezza maggiori. Ad esempio il Service Center Microsoft di Qincy (USA) include circa 10.000 server fisici che supportano a run-time almeno altrettante macchine virtuali. Questi numeri mettono in discussione la scalabilità dei modelli di ottimizzazione proposti in letteratura, rendendoli di fatto non applicabili in contesti reali. Il presente lavoro di tesi ha lo scopo di proporre un'estensione a tecniche di resource allocation proposte in precedenti lavori di ricerca per migliorarne la scalabilità . L'obiettivo è quello di ricercare l'allocazione ottima delle risorse del sistema complessivo in modo da minimizzare il consumo energetico dell'infrastruttura rispettando i livelli di Qualità del Servizio stabiliti nei contratti di Service Level Agreement. E' stato sviluppato un algoritmo gerarchico di allocazione delle risorse basato sul partizionamento logico delle applicazioni e dei server fisici appartenenti al sistema. Il partizionamento delle applicazioni avviene tenendo conto dei modelli di traffico. Ad ogni partizione applicativa viene associato un cluster di server il cui numero e tipologia viene determinato considerando il carico delle richieste da elaborare e le caratteristiche fisiche dei server. Ciascun cluster viene poi controllato da un supervisore locale che stabilisce le performance da garantire alla partizione di applicazioni del cluster. Infine, per valutare l'efficienza degli algoritmi di ottimizzazione proposti è stata effettuata un'analisi sperimentale basata su simulazioni.

Capitolo 1

Introduzione

Negli anni '90, con l'avvento delle applicazioni Internet vennero a formasi grandi strutture contenenti server e apparati di rete per permettere l'esecuzione continua di svariati servizi. I server erano collocati in grosse stanze per consentire una maggiore sicurezza, una facile manutenzione ed una climatizzazione efficace degli ambienti. Queste grandi strutture prendono il nome di Service Center, ossia centri in grado di offrire dei servizi IT ai clienti. Col passare degli anni i prezzi dell'hardware necessario per realizzare questi centri calarono, permettendo alle aziende più piccole di costruire il proprio Service Center ed a coloro che non avevano abbastanza fondi di potersi avvalere di servizi di outsourcing offerti da società più grandi. In quest'epoca nacquero e si svilupparono aziende come Google.

Sopratutto negli ultimi anni, la proliferazione incontrollata dei server ha fatto in modo che non potesse più essere sottovalutata la sostenibilità dei costi, in particolare di quelli energetici. Oggigiorno infatti i Service Center sono responsabili dello 0.5% del consumo energetico mondiale. Negli Stati Uniti l'ammontare di energia spesa in un Service Center raggiunge l'1.5% del consumo nazionale, mentre nel Regno Unito questa percentuale sale a circa il 3%. Oltre al problema energetico non è da sottovalutare anche il problema ambientale. Infatti, i Service Center rappresentano il 2% delle emissioni di CO_2 di tutto il globo, ovvero la metà dei gas serra prodotti dalle compagnie aeree e più dell'emissioni di tutto lo stato argentino (FIGURA 1.1 [1]).

Considerando il consumo delle infrastrutture IT europee, si può notare come le previsioni indichino che nei prossimi anni il consumo energetico dei Service Center sarà in grande ascesa, più di altri settori IT come quello delle telecomunicazioni (FIGURA 1.2) [19].

[1] Per quanto riguarda gli scopi di questo lavoro di tesi si considerano sinonimi Data Center e Service Center.

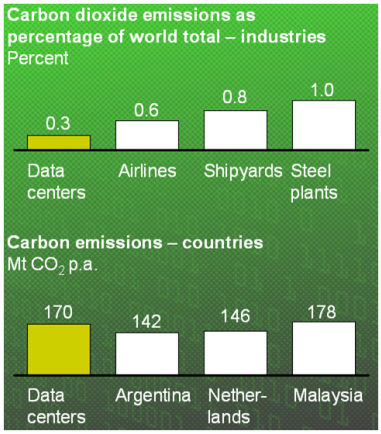

Figura 1.1: Paragoni delle emissioni di gas serra prodotte dai Data Center e grandi industrie o nazioni ([47]).

A livello di costi, Google possiede 36 Service Center sparsi per il mondo che consumano 6 Twh per un costo superiore a 38 milioni di dollari l'anno. Da queste cifre emerge come l'entità del fenomeno non può essere trascurata. Si calcola in fatti che entro il 2020 il consumo dei Service Center comporterà un'emissione di gas serra pari a quella dell'industria aeronautica [6]. Si calcola che negli odierni Service Center ciascun server viene utilizzato in media tra il 5% e il 10% della sua capacità elaborativa, lasciando per gran parte del tempo inutilizzate molte risorse. L'alto livello di sottoutilizzo si contrappone con il paradigma della server consolidation che prevede un approccio di gestione efficiente delle risorse. Il fattore abilitante di questo paradigma è rappresentato dalle tecnologie di virtualizzazione. Grazie ad esse è possibile perseguire obiettivi di ottimizzazione in maniera molto flessibile. Sopratutto negli ultimi anni la comunità scientifica si è mossa nella direzione di progettare sistemi integrati per l'ottimizzazione e l'uso intelligente delle risorse. Tuttavia i Service Center si stanno ampliando così rapidamente da rendere queste tecniche poco scalabili e quindi inefficienti per la gestione dei consumi energetici su larga scala. Ad esempio, il Service Center Microsoft situato a Quincy (Washington) ha raggiunto le dimensioni di 10 campi da football (43600 m^2) con un'estensione di tubature per la climatizzazione degli ambienti di 4.8 Km ([17]). Questo Service Center contiene migliaia di server e può elaborare migliaia di applicazioni. Occorre quindi un sistema di gestione dei costi energetici che sia scalabile e possa permettere di controllare efficientemente queste grosse strutture. L'obiettivo del presente lavoro di tesi è sviluppare tecniche gerarchiche per l'allocazione delle risorse in sistemi di Autonomic Computing con lo scopo di determinare il trade off ottimale tra le prestazioni offerte ai servizi applicativi ed i costi energetici. Tali tecniche

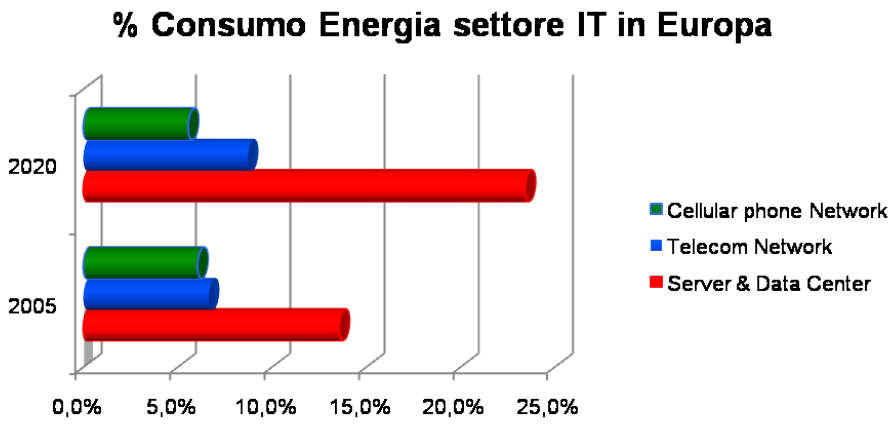

Figura 1.2: Previsioni di consumi energetici europei nel settore IT.

consentiranno di poter gestire in maniera efficiente e scalabile i Service Center odierni di grosse dimensioni.

Il lavori di tesi è organizzato come segue:

- Nel Capitolo 1 viene effettuata una panoramica sulle applicazioni internet, sulla loro struttura multi-tier e sui moderni Service Center. Vengono discussi i contratti (SLA) che regolano i rapporti tra le società proprietarie dei Service Center ed i clienti attraverso vincoli sul livello minimo di qualità del servizio offerto. In seguito saranno descritte le caratteristiche degli Autonomic System e verranno illustrati alcuni concetti sulla Business Continuity. Infine verranno presentati le attuali tecniche utilizzate dalla comunità scientifica per risolvere queste problematiche.

- Nel Capitolo 2 viene descritto un modello di allocazione delle risorse precedentemente sviluppato in altri lavori di tesi ([8, 14]) e che costituisce il punto di partenza per il presente lavoro di tesi.

- Nel Capitolo 3 verranno discusse alcune estensioni apportate al modello di Resource Allocation descritto nel Capitolo 2. Queste estensioni permettono di aggiungere un vincolo al sistema che tiene conto della memoria RAM occupata dalle Virtual Machine e di considerare un modello avanzato per il calcolo del consumo energetico. Nella seconda parte del capitolo verrà illustrato un modello gerarchico di allocazione delle risorse che permette di gestire in maniera flessibile e scalabile Service Center di larga scala.

- Nel Capitolo 4 saranno presentatati alcuni risultati sperimentali il cui

obiettivo è quello di valutare l'efficacia delle soluzioni proposte. In particolare, verrà valutata la scalabilità delle tecniche di resource allocation proposte. Verranno inoltre esposti dei casi di studio per stabilire il comportamento dell'algoritmo a fronte di variazioni dei parametri prestazionali delle applicazioni.

- Infine, nel Capitolo 5 verranno presentate le conclusioni relative al lavoro svolto ed alcune possibili direzioni di ricerca.

Capitolo 2

Stato dell'arte

Il seguente capitolo illustra una panoramica sui moderni Service Center e su tutte le loro principali componenti. Nella Sezione 2.1 *vengono discusse le applicazioni internet. In particolare, viene messa in evidenza una tipica architettura multi-tier dove ciascuna applicazione, per essere elaborata, deve essere superare più stadi.*
La replicazione fisica dei server ha rappresentato per anni l'unico metodo per far fronte alla crescente domanda di carico da soddisfare. Questo processo di espansione ha permesso negli anni un forte ampliamento dei Service Center, con la conseguente esplosione dei costi per il mantenimento di queste infrastrutture. A fronte di questo problema, vengono discusse nella Sezione 2.2 *le tecnologie di virtualizzazione, chiave abilitante della server consolidation. Nella* Sezione 2.3 *si discutono gli accordi contrattuali sui livelli di servizio (SLA) che i Service Center stipulano con i loro clienti mentre nella* Sezione 2.4 *vengono presentate alcune strategie per il risparmio energetico nei centri di servizio, al fine di incrementare i guadagni introdotti dal rispetto degli SLA. Nella* Sezione 2.5 *sono presentate le caratteristiche dei sistemi autonomici (Autonomic System), mentre nella* Sezione *2.6 vengono illustrati due tipici modelli prestazionali basati sulle reti di code. Nella* Sezione 2.7*, vengono illustrati i principi basilari della Business Continuity. Infine, nella* Sezione *2.8 viene presentata una panoramica sulle tecniche di allocazione delle risorse adottate nei Service Center.*

2.1 Le applicazioni Internet

Nell'ultimo decennio, una serie positiva di fattori, ha portato ad una rapida evoluzione delle applicazioni Internet (tra cui ad esempio sono stati realizzati sistemi ed apparati per vendita on line, home banking, aste online, social network, ecc..). Tuttavia, la rapida ascesa ha significato un notevole incremento di complessità degli applicativi. Le applicazioni Internet attuali sono infatti sistemi software complessi che impiegano un'architettura multi-tier. In genere, sono replicate o distribuite su un insieme di server, sia per permettere di mantenere un certo livello di prestazioni a fronte di cambiamenti nel numero delle richieste, sia per poter continuare la fornitura di un servizio anche in presenza di guasti.

In un'architettura multi-tier, ogni strato tier fornisce una certa funzionalità al tier che lo precede ed usa a sua volta la funzionalità fornita dal tier successivo per elaborare il sottoinsieme delle funzionalità che gli sono state assegnate. Questo sistema di suddivisione in tier che compongono un applicativo fornisce un approccio flessibile e modulare per il design delle applicazioni Internet. Per esempio, un'applicazione tipica di *e*-commerce è suddivisa in tre tier (FIGURA 2.1): un tier di presentazione (Web tier), responsabile dell'elaborazione delle richieste HTTP, un application tier intermedio (ad esempio un Java enterprise server), che implementa la logica applicativa, ed infine un tier dati, ossia una base di dati che contiene, ad esempio, i cataloghi dei prodotti e che si occupa di memorizzare gli ordini degli utenti.

Si consideri un'applicazione multi-tier composta da M tier indicati con T_1, T_2 fino a T_M. Nel caso più semplice, ogni richiesta viene eseguita esattamente una volta dal tier T_i e successivamente inoltrata al tier T_{i+1}, e così via. Una volta che il risultato è stato computato dall'ultimo tier T_M, questo viene ritornato al tier T_{M-1} che elabora questo risultato e lo invia a T_{M-2}. In sostanza quindi, il risultato viene elaborato da ogni tier in ordine inverso fino a quando raggiunge T_1 che lo restituisce al client. Sono tuttavia possibili anche elaborazioni più complesse, dove ogni richiesta deve visitare uno stesso tier più volte.

Per poter gestire elevati volumi di carico con un livello prestazionale soddisfacente, occorre molto spesso replicare i vari tier su più macchine in modo da, per mezzo di un componente detto dispatcher, distribuire su più server le richieste in arrivo.

2.2 La virtualizzazione del Service Center

Per virtualizzazione si intende un framework o una tecnologia di gestione virtuale delle risorse reali di una macchina in ambienti di esecuzione multipli (Virtual Machine), che si appoggia su concetti come il partizionamento hardware e software, time-sharing, emulazione e qualità del servizio. In altre parole la virtua-

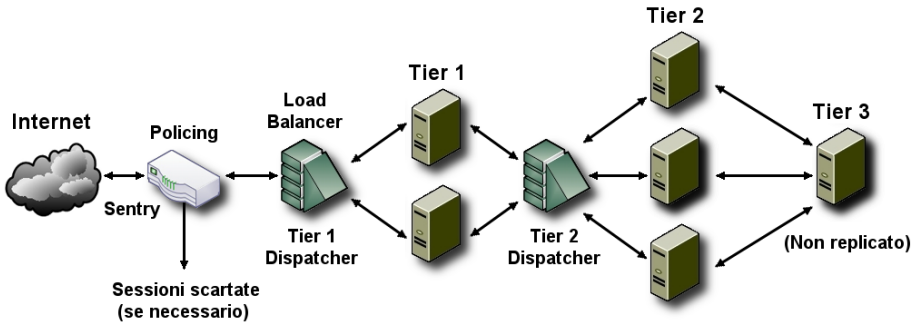

Figura 2.1: Esempio di un'architettura a tre tier.

lizzazione è una tecnica tale da permettere l'esecuzione simultanea di sistemi operativi ed applicazioni differenti sul medesimo server/computer andando a partizionare le risorse del sistema (CPU, memoria, schede di rete, ecc..) in più macchine virtuali (VM). In questo modo, l'esecuzione di più applicazioni su un unico elaboratore aumenta l'efficienza del sistema e riduce il numero di macchine da gestire e mantenere.

Ogni VM si comporta come una singola macchina stand-alone, dedicata all'esecuzione di una particolare applicazione ed è indipendente dall'hardware su cui viene eseguita poiché la VM effettua un'astrazione delle risorse fisiche del calcolatore. In altre parole il software di virtualizzazione "nasconde" le risorse hardware fisiche e modifica il modo con cui il sistema operativo, le applicazioni e gli utenti interagiscono con esse. Per gestire l'accesso alle risorse fisiche condivise dalle VM, il sistema di virtualizzazione utilizza un componente che prende il nome di virtual machine monitor (VMM). Questo elemento crea una o più astrazioni dell'hardware che possono essere utilizzate da "sistemi ospiti" (guest). Vengono create quindi delle repliche virtuali delle componenti hardware che in un sistema non virtualizzato sono invece a diretto contatto con il kernel del sistema operativo.

In un sistema tradizionale (Figura 2.2-A) il kernel si frappone fra la macchina fisica (hardware) e le varie applicazioni. Quando un'applicazione necessita di comunicare con un componente hardware del sistema viene inviato un segnale al microprocessore (interrupt), il quale richiama l'esecuzione del kernel che si preoccupa di gestire opportunamente questi eventi. In un sistema virtualizzato (Figura 2.2-B) il VMM è frapposto tra l'hardware ed i singoli kernel. Ciascun sistema operativo ospitato è a contatto con una macchina virtuale simulata dal VMM che ha associato un proprio contesto di esecuzione. In questo modo i segnali di interrupt richiamano l'esecuzione del VMM, che li gestisce tenendo presente la VM da cui sono stati generati. Va infine considerato che permettere

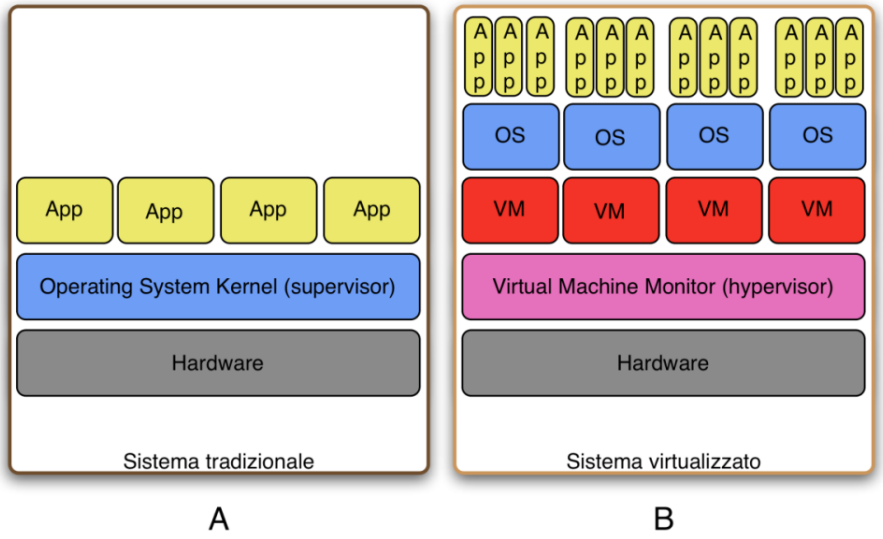

Figura 2.2: Paragone fra un sistema tradizionale (non virtualizzato) ed uno virtualizzato.

ad una VM di modificare autonomamente (senza l'intervento del VMM) lo stato delle risorse hardware costituirebbe un potenziale problema di sicurezza in quanto potrebbe avere effetti imprevedibili sulle altre VM e sul sistema stesso.

Il VMM deve garantire 4 funzionalità:

- **Partizione intelligente delle risorse**
  Deve poter dividere nelle differenti VM tutte le funzionalità del computer/server. Ogni VM ha i suoi processi, i suoi file, i suoi utenti, le sue librerie; tutto ciò che occorre per avere una macchina stand alone.

- **Isolamento completo**
  Le VM devono essere isolate tra loro per quanto riguarda le condizioni di errore e le prestazioni.

- **Allocazione dinamica delle risorse**
  Il VMM deve poter riassegnare in modo dinamico le risorse durante l'esecuzione delle VM.

- **Live migration**
  La virtualizzazione deve fornire la separazione dall'hardware sui cui vengono eseguite le VM. Queste devono poter essere spostare su un altro elaboratore in maniera del tutto trasparente.

2.2.1 Motivazioni a supporto dei sistemi virtualizzati

Inizialmente si credeva che la replicazione delle applicazioni su molti server fosse la soluzione migliore per garantire elevati livelli di servizio e di affidabilità. A causa di questo modo di pensare nel corso degli anni ci fu una proliferazione incontrollata dei server ed il conseguente aumento dei costi di gestione delle infrastrutture. L'aggiunta di una nuova applicazione comportava l'installazione di almeno un nuovo server e non era infrequente avere macchine sottoutilizzate da un unico servizio, che impiegava solo una minima percentuale del tempo CPU e della banda. Lo sfruttamento dei server, infatti, si attestava attorno al 20-30% evidenziando enormi sprechi dovuti al dispendio energetico e al mantenimento di risorse non utilizzate.

E' proprio in questo scenario che nasce quella che viene chiamata "server consolidation", che ha come obiettivo quello di ridurre il numero di server nel sistema, ottimizzando le risorse e la percentuale di utilizzo di quelli già presenti. La server consolidation è resa possibile grazie alla virtualizzazione che permette in maniera molto flessibile di poter eseguire più applicazioni su un unico elaboratore. Oltre agli evidenti risparmi sui costi di mantenimento della struttura IT, la virtualizzazione e la server consolidation stanno conoscendo un periodo di favorevole notorietà poiché favoriscono il risparmio energetico e quella che viene chiamata Green IT. Nel 2008, infatti, VMware (leader nel settore della virtualizzazione) affermava che con ogni server virtualizzato, ogni anno, si può risparmiare circa 7000 KW/h (kilowatt/ora) e circa 4 tonnellate di emissioni di CO2. In fine, in ottica Green, la virtualizzazione consente il riutilizzo della struttura informatica esistenze anziché l'acquisto di nuovi server restando, nel contempo, in linea con il mercato.

La virtualizzazione viene anche definita come la tecnologia abilitante al cloud computing (Figura 2.3). Il termine cloud computing sta ad indicare un' infrastruttura che permette l'utilizzo in remoto di risorse hardware o software distribuite. La cloud utilizza uno schema volto a superare il classico modello di distribuzione client/server. Vengono create delle nuvole che offrono capacità elaborativa e/o di memorizzazione. Queste nuvole sono formate da server fisici che possono essere eterogenei e disposti in spazi geograficamente differenti. Gli utenti possono utilizzare questa infrastruttura in modo totalmente trasparente sia rispetto all'eterogeneità dei server e delle applicazioni, sia alla geolocalizzazione dei server stessi.

Altri vantaggi che si hanno grazie alla virtualizzazione sono:

- L'esecuzione di applicazioni legacy, spesso non pienamente compatibili con l'hardware attuale.

- Le VM forniscono un ambiente sicuro e isolato, ottimo per testare applica-

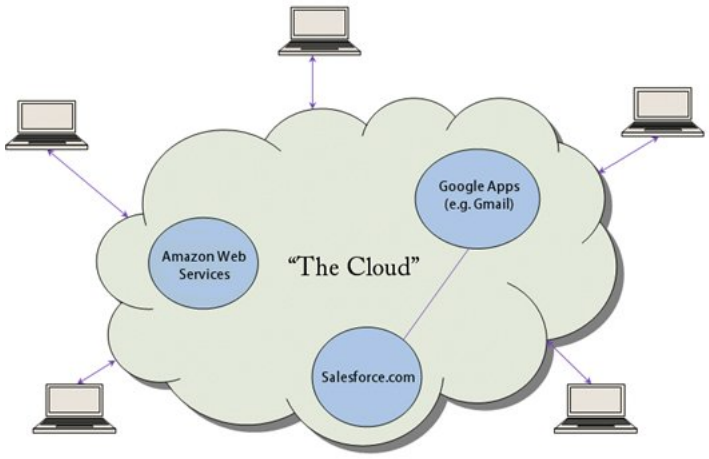

Figura 2.3: Sistema Cloud.

zioni di cui non ci si fida pienamente oppure applicazioni in fase di debug, dato che gli errori restano gestiti all'interno della macchina virtuale.

- Si possono limitare le risorse disponibili ad un'applicazioni e studiarne le performance. Si possono quindi implementare dei meccanismi di quality of service.

- Le VM possono illudere le applicazione utilizzare un hardware che in realtà non si possiede.

- La virtualizzazione incrementa il livello di disponibilità del sistema e la tolleranza ai guasti.

- Le macchine virtuali rendo più facile le migrazioni del software, i backup e il ripristino.

I primi utilizzi pratici della virtualizzazione si sono registrati negli anni '60 quando IBM utilizzò queste tecnologie per partizionare (logicamente) le risorse dei suoi mainframe, grandi sistemi multiutente molto costosi le cui risorse, spesso, venivano sottoutilizzate. Attraverso la virtualizzazione, le partizioni cercavano di rendere questi sistemi multitasking, dando la possibilità di eseguire più programmi contemporaneamente. Questo meccanismo era implementato da un apposito software chiamato supervisore che per ogni applicazione dedicava una certa quantità di memoria, dello spazio su disco e un numero stabilito di cicli CPU facendo, però, credere al programma in esecuzione di avere a disposizione tutte le risorse del sistema.

2.2.2 I requisiti di Popek e Goldberg

Nel 1974, due ricercatori americani (Gerald J. Popek e Robert P. Goldberg) presentarono alla comunità scientifica un'articolo ([35]) che proponeva un insieme di condizioni sufficienti che, se rispettate, permettevano di garantire la virtualizzazione di una data architettura in maniera efficiente e completa.

Questi requisiti sono i seguenti:

- Equivalenza: un'applicazione eseguita in una VM deve comportarsi nello stesso modo di quando essa è eseguita direttamente su una macchina fisica;
- Controllo delle risorse: il Virtual Machine Monitor deve avere pieno controllo delle risorse da esso virtualizzate;
- Efficienza: una frazione di istruzioni statisticamente dominante deve poter essere eseguita senza l'intervento del VMM, in questo modo il VMM verrà eseguito solo quando strettamente necessario.

Inoltre, venne formalizzato che le istruzioni di un'architettura possono essere divise in 3 gruppi:

- Privileged: insieme di istruzioni che, se eseguite da un'applicazione, generano una trap;
- Control sensitive: insieme di istruzioni che possono modificare il contesto di esecuzione del sistema;
- Behaviour sensitive: insieme di istruzioni il cui completamento dipende in qualche modo dal contesto di esecuzione del sistema.

Con queste premesse, può essere formulato il teorema noto come teorema di Popek e Goldberg che dimostra come sia possibile ottenere delle macchine virtuali che rispettano tutte e tre le proprietà riportate sopra solo se le istruzioni control sensitive e behaviour sensitive costituiscono un sottoinsieme di quelle privileged.

Dal teorema segue che un'architettura rispetta le tre proprietà solo se tutte le istruzioni che espongono in qualche modo il contesto di esecuzione del sistema (control sensitive), o il cui risultato è dipendente da questo (behaviour sensitive) generano anche una trap (eccezione software). Queste due insiemi di istruzioni infatti sono considerati problematici dal punto di vista della virtualizzazione poiché, se non si riesce a garantire una corretta emulazione del corrente contesto di esecuzione ogni volta in cui una VM tenta di accedervi si comprometterebbe il corretto funzionamento della VM stessa. Ad ogni modo, su architetture che rispettano i requisiti elencati, la corretta emulazione è garantita dal fatto che il VMM viene richiamato ogni qualvolta un'applicazione debba accedere al proprio contesto di esecuzione o debba ottenere dei risultati che vi dipendano poiché essa procede con la generazione di una trap.

Infine, si può affermare che i requisiti esposti nel teorema di Popek e Goldberg sono da ritenersi solamente sufficienti. Infatti, se questi non sono del tutto soddisfatti è possibile implementare VMM funzionanti ma che in talune condizioni possono non garantire le tre proprietà elencate inizialmente.

Tra le varie architetture hardware dei processori attualmente in commercio, quella che è riuscita a raggiungere una diffusione capillare, sia in ambito workstation che in ambito server, è l'architettura denominata x86, sviluppata e prodotta inizialmente da Intel. Data la diffusione di tale architettura, la virtualizzazione in ambiente x86 ha assunto un ruolo di primo piano nel mercato.

L'architettura x86 tradizionalmente non rispetta le proprietà elencate e di conseguenza è da considerarsi non nativamente virtualizzabile. Tuttavia nel corso degli anni sono state studiate diverse tecniche per consentire una buona virtualizzazione anche su questi sistemi.

2.3 I Service Level Agreement

Si definisce *Service Level Agreement* (SLA) uno strumento contrattuale attraverso il quale vengono definite le metriche di servizio che devono essere rispettate dal fornitore del servizio stesso (*Service Provider*).

Questi contratti rappresentano oggigiorno uno strumento comune per misurare efficacemente i servizi. Questi contratti, infatti, definisco un livello minimo di qualità che deve essere garantito. In caso di mancato adempimento del livello di servizio preposto, vengono predisposte delle penali monetarie a carico del fornitore a seguito del mancato rispetto dei livelli pre-negoziati. In particolare, il livello di servizio da fornire può essere di tipo *guaranteed*, per il quale le violazioni delle specifiche pattuite non sono tollerate, oppure di tipo *predictive*, per il quale si specifica invece una soglia massima di violazioni ammissibili.
La definizione dello SLA è basata sulla determinazione, da parte del cliente, del livello di servizio ideale a garanzia del proprio business. I requisiti contenuti nel contratto possono prevedere l'impiego di più livelli di servizio, in base, ad esempio, all'ora del giorno o alla classe cui appartiene l'utente che invoca il servizio.

Un Service Level Agreement è strutturato su più livelli in funzione dell'ambito di applicazione:

- **Infrastruttura di rete**: riguarda la definizione dei livelli di servizio relativi agli apparati di rete, alla sicurezza di accesso ed alla disponibilità di banda;

- **Ambiente sistemistico**: riguarda la definizione dei livelli di servizio relativi agli apparati hardware, ai sistemi, ai software di base ed ai criteri di monitoraggio;

- **Ambiente applicativo**: riguarda la definizione dei livelli di servizio che sono specifici per l'applicazione del cliente. Possono quindi essere indicati dei parametri relativi ai tempi di risposta, al numero delle transazioni, ecc...;
- **Supporto ai clienti**: riguarda la definizione dei livelli di servizio relativi ai servizi di Help Desk e di supporto agli utenti.

Generalmente, un processo di definizione e monitoraggio di SLA si articola secondo le seguenti fasi:

- Descrizione dettagliata dei servizi che devono essere forniti al cliente;
- Definizione degli indicatori di riferimento (livelli di servizio) e dei relativi algoritmi di calcolo;
- Valutazione dell'impatto di un'eventuale violazione dei livelli di servizio previsti dallo SLA ed indicazione delle penali corrispondenti;
- Realizzazione del sistema di produzione e di reportistica degli indicatori;
- Condivisione dei valori soglia (target) degli SLA;
- Monitoraggio degli indicatori rilevati ed eventuale rinegoziazione periodica dei valori soglia.

Particolare importanza assumono nell'erogazione di servizi sulla rete i cosiddetti *SLA prestazionali* (i quali definisco obiettivi di disponibilità, tempi di risposta, ecc...) che, rispetto agli SLA più generici, offrono un consistente vantaggio competitivo soprattutto grazie alla possibilità di associarvi una sorta di personalizzazione del livello di servizio; il livello viene infatti parametrizzato in base alle esigenze del singolo cliente. Il monitoraggio degli SLA prestazionali, pertanto, consente di verificare sia l'andamento dei livelli di servizio per quanto riguarda i valori target contrattuali, sia la presenza di anomalie tecniche od applicative che potrebbero causare potenziali disservizi all'utente finale. Nella TABELLA 2.1 sono elencate alcune delle categorie e degli elementi specifici che possono essere inclusi nei contratti SLA.

La maggior parte dei lavori, presenti in letteratura, relativi alla *resource allocation* ricercano la migliore allocazione considerando i vincoli di SLA, i quali vengono formalizzati tramite funzioni matematiche. In [33], i vincoli di SLA sono rappresentati da una *cluster utility function*, composta da due tipi di funzioni, entrambe fornite dal service provider:

- per ogni classe di traffico viene definita una *class-specific utility function* di performance;

Categorie	Esempi di elementi dei contratti SLA
Ore di funzionamento	- Supportato: ore di disponibilità del servizio per gli utenti - Garantito: ore di funzionamento minimo di un servizio - Ore riservate all'inattività pianificata (manutenzione, upgrade della rete)
Disponibilità del servizio	- Livello percentuale di funzionamento del servizio (uptime) espresso come percentuale dell'orario di funzionamento garantito rispetto all'orario di funzionamento supportato, al netto dei tempi di manutenzione (ad esempio 99,8%)
Prestazioni del sistema	- Numero di utenti interni che il sistema supporta attualmente - Numero di utenti connessi remotamente che il sistema supporta attualmente - Numero di transazioni supportate per unità di tempo - Livello accettabile di prestazioni come, ad esempio, il tempo di risposta
Ripristino d'emergenza	- Quantità di tempo permessa per il ripristino di ogni tipo di fault del sistema, ad esempio, guasto del database, del server, etc. - Quantità di tempo necessaria per il ripristino dei dati in situazione di malfunzionamento
Servizio di assistenza/ Supporto tecnico	- Metodi specifici e utilizzabili dagli utenti per contattare il servizio di assistenza - Tempo di risposta del servizio di assistenza per varie classi di problemi. - Tempo medio di intervento dalla segnalazione di disservizi

Tabella 2.1: Categorie ed elementi in un tipico contratto SLA

- i valori della class utility sono combinati in un unico valore di cluster utility tramite una *combining function*. Sono proposte due differenti tipi di *combining function*: la prima permette di calcolare la *cluster utility function* come la somma di ogni *class utility function*, al fine di massimizzare l'utility complessiva del sistema. La seconda, invece, massimizza la più piccola *utility function*, con lo scopo di equalizzare l'utility di tutte le classi. Il service provider sceglierà di seguire l'una o l'altra politica basandosi su considerazioni quali l'importanza relativa delle differenti classi di servizio, la soddisfazione degli utenti e la reputazione.

L'*utility function* evidenzia l'importanza di raggiungere gli obiettivi di perfor-

mance per ciascuna classe di servizio. Per ogni servizio possono comunque essere definiti diversi gradi di qualità, ognuno caratterizzato da uno specifico insieme di parametri obiettivo di performance e, usando un tool di configurazione, il service provider può definire il numero ed i parametri per ogni grado di qualità. Inoltre può anche definire insiemi di classi di traffico ed associarvi i rispettivi tempi di risposta *target*. Dal lato utenti, questi, tramite un'interfaccia di sottoscrizione, possono registrarsi al sistema ed abbonarsi ai servizi, selezionando un'offerta specifica ed il grado di qualità associato.

In [46], [10] e [48], il vincolo di SLA consiste nel limitare superiormente il tempo di risposta medio accettabile per l'applicazione. In [28], per ottimizzare le performance del sistema nell'intorno di un tempo di risposta medio desiderato, viene usata una strategia di *pricing* a gradini la quale fornisce al service provider maggiori guadagni per tempi di risposta bassi e che diminuiscono con l'aumentare del tempo di risposta.

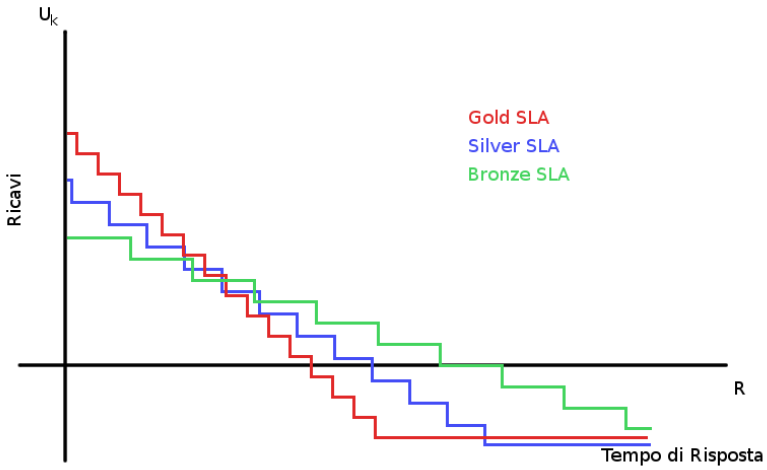

Figura 2.4: Strategia di pricing a gradini. Tre utility function, ciascuna associata ad un diverso SLA e che si differenziano per i diversi vincoli posti sulle prestazioni richieste.

La FIGURA 2.4 esemplifica una strategia di *pricing* comunemente usata, dove i consumatori Gold si aspettano di ricevere il miglior servizio, ossia tempi di risposta bassi, pagando un maggiore corrispettivo, i clienti Silver pagano un prezzo minore per un servizio meno performante mentre i clienti Bronze pagano il minimo per il servizio base. Ogni volta che il tempo di risposta viola lo SLA, ossia è superiore al massimo consentito, il service provider deve pagare delle penali ai propri client.

In [5], si definisce invece una *utility function* lineare per ciascuna classe

di servizio k (FIGURA 2.5). L'*utility function* specifica il ricavo o la penale previsti quando il tempo di risposta medio R_k assume un determinato valore. La pendenza della retta è m_k, assunta positiva ($m_k = \frac{v_k}{z_k} > 0$) e z_k è la soglia che separa la regione di profitto da quella di penale (ossia se $R_k > z_k$ il contratto SLA è violato e si dovranno pagare delle penali). Questa tipologia di *utility function* verrà considerata come tipologia di riferimento nel presente lavoro di tesi.

Figura 2.5: *Utility function* lineare.

2.4 La gestione del consumo energetico

L'espansione dei servizi commerciali basati sulla rete, insieme alla progressiva esternalizzazione dei servizi IT, sta portando alla crescita di Service Center che ospitano differenti servizi/applicazioni. Questi centri generano profitto fornendo migliaia di server e dispositivi di storage per l'esecuzione delle applicazioni dei clienti. Per anni nella progettazione dei Service Center la tendenza è stata quella di sovradimensionare la capacità elaborativa dedicata a ciascun servizio in modo da poter far fronte ai possibili picchi di carico. Col passare del tempo si è capito che questa tecnica non è economicamente conveniente. Infatti, il consumo energetico dei server sta divenendo un aspetto sempre più rilevante per quanto riguarda la progettazione ed il funzionamento di una server farm. La FIGURA 2.6 mostra la ripartizione dei cunsimi all'interno di un server.

Gli effetti di un consumo energetico elevato si ripercuotono sia nei notevoli costi da sostenere per la progettazione di sistemi di raffreddamento efficienti,

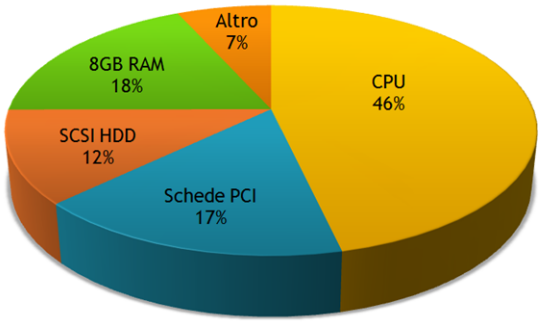

Figura 2.6: Distribuzione dei consumi di un server [30][20].

sia nei costi propri dell'elettricità consumata (si faccia riferimento alla FIGURA 2.7). Diventa cruciale quindi adottare strategie di gestione dell'energia tali da diminuire questi costi, aumentando di conseguenza i profitti.

La comunità scientifica si è focalizzata sulla ricerca della giusta capacità e sulla distribuzione di questa tra le varie applicazioni basandosi sugli SLA. Va comunque considerato che il carico di richieste che giunge al Service Center non è in genere costante, per cui ci potrebbero essere periodi di tempo durante i quali la capacità totale a disposizione è molto più alta di quella effettivamente richiesta, tuttavia durante questi periodi i costi per l'elettricità e per il raffreddamento devono comunque essere sostenuti pienamente.

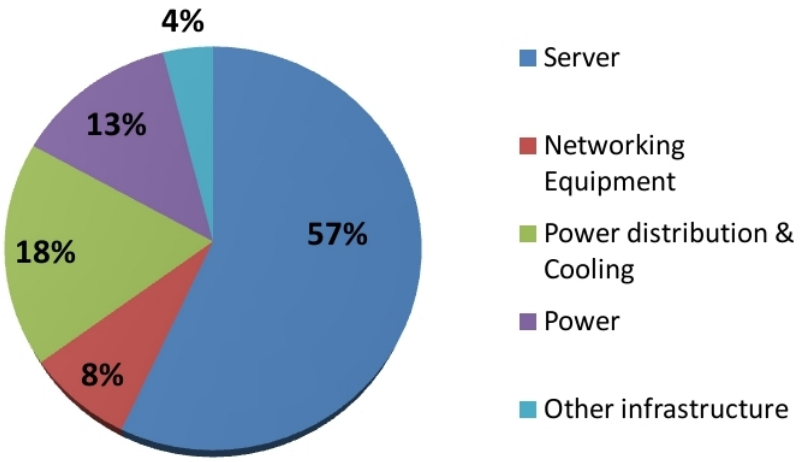

Figura 2.7: Distribuzione mensile dei consumi in un Service Center.

Dimensionare staticamente il numero di server per un certo servizio può por-

tare quindi a scelte eccessivamente conservative od a mancante opportunità di risparmio, poiché il carico associatovi varierà nel tempo. È quindi opportuno gestire dinamicamente il numero dei server che devono essere accesi ed assegnati ad ogni servizio, andando a spegnere i server sottoutilizzati.
Modificare la frequenza operativa di un server per adattare la capacità elaborativa al carico da elaborare, offre ulteriori possibilità di risparmio. A tal fine, oggigiorno è possibile sfruttare il *Dynamic Frequency Scaling*[1] (DFS). Infatti nei moderni processori il voltaggio e la frequenza possono essere ridotti per risparmiare maggiormente sui costi energetici a fronte di perdite di prestazioni ragionevoli. Anche se lo spegnimento di un server porterebbe a risparmi maggiori, con il DFS le richieste possono essere comunque servite, anche se a bassa frequenza, controbilanciando efficacemente i costi.

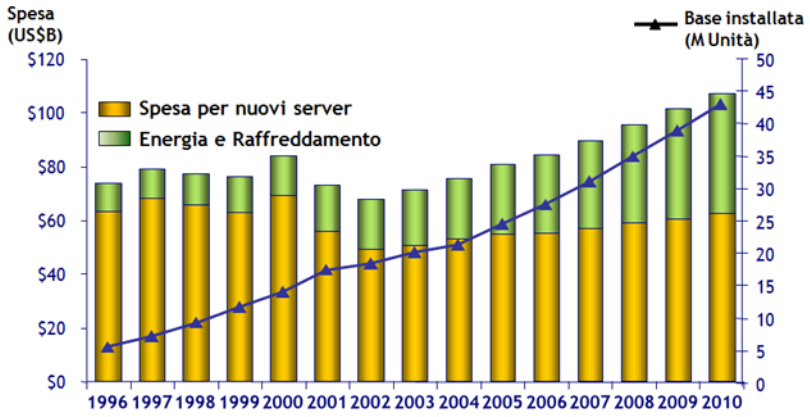

Figura 2.8: Andamento dei costi per il mantenimento di Service Center negli anni. Nel tempo, la potenza dei server aumenta così come l'energia impiegata dai dispositivi di raffreddamento [18][29].

2.4.1 Dynamic Frequency Scaling

Come visto, ridurre dinamicamente la capacità operativa di un server può creare ampi margini di risparmio. Il meccanismo di Dynamic Frequency scaling, presente nei moderni processori, offre la possibilità di operare in diversi stati, denominati *p-state* (*performance state*), a ciascuno dei quali è associato sia un valore di frequenza che un valore di voltaggio. Quando si parla quindi di DFS, o DVS, ci si riferisce alla transizione di un processore da un p-state ad un altro, ossia alla modifica contemporanea della frequenza e del voltaggio del processore. Attualmente vi sono processori in grado di operare a svariate decine di frequenze differenti, tuttavia sono presenti vincoli tecnologici a livello di scheda

[1] Definito anche come *Dynamic Voltage Scaling* (DVS).

madre che limitano il numero di p-state effettivamente raggiungibili (si veda la FIGURA 2.9).

Secondo le specifiche ACPI 3.0[2] [12], i p-state di un processore sono definiti nel modo seguente:

- **P0 Performance State**: quando un processore si trova in questo stato, esso funziona alla sua massima capacità ed ha il massimo consumo di energia;
- **P1 Performance State**: in questo stato, le prestazioni ed i consumi del processore sono al di sotto dei corrispondenti massimi;
- **Pn Performance State**: in questo stato, le prestazioni del processore sono al livello minimo ed esso consuma la minima quantità di energia rimanendo comunque in uno stato attivo. Lo stato n corrisponde quindi al p-state massimo ed è dipendente dal processore.

In genere le specifiche del p-state massimo e del minimo sono immagazzinate nei *Machine Specific Register* (MSR) del processore stesso, mentre le specifiche di p-state addizionali dovranno risiedere nel BIOS e verranno quindi gestiti a livello di scheda madre. Ovviamente definendo un maggior numero di p-state si permette al sistema operativo una maggiore granularità nel selezionare opportunamente la capacità da fornire ed il consumo da sostenere per un determinato carico.

Considerare anche le frequenze operative come variabile di controllo ha, perciò, i seguenti vantaggi:

- Lo scaling della frequenza può essere eseguito quasi istantaneamente con piccoli costi di commutazione;
- Si ha un piccolo impatto nel lungo termine sul logorio dei server;
- La modifica della frequenza operativa permette al cluster di controllare i costi attraverso una relazione cubica tra la frequenza operativa e il consumo energetico [13], mantenendo la risorsa in uno stato attivo in modo da poter comunque gestire picchi imprevisti nel carico di lavoro.

2.5 Autonomic Computing

L'Autonomic Computing è un'iniziativa avviata dall'IBM nel 2001. Il suo scopo è quello di fornire ai computer gli strumenti necessari per garantire determinate proprietà chiamate self-*. Queste proprietà impongono che il sistema sia in grado di auto-configurarsi, auto-gestirsi, auto-ottimizzarsi, auto-verificarsi, auto-proteggersi ed auto-ripararsi (per quanto fattibile), con un intervento umano

[2] Advanced Configuration and Power Interface.

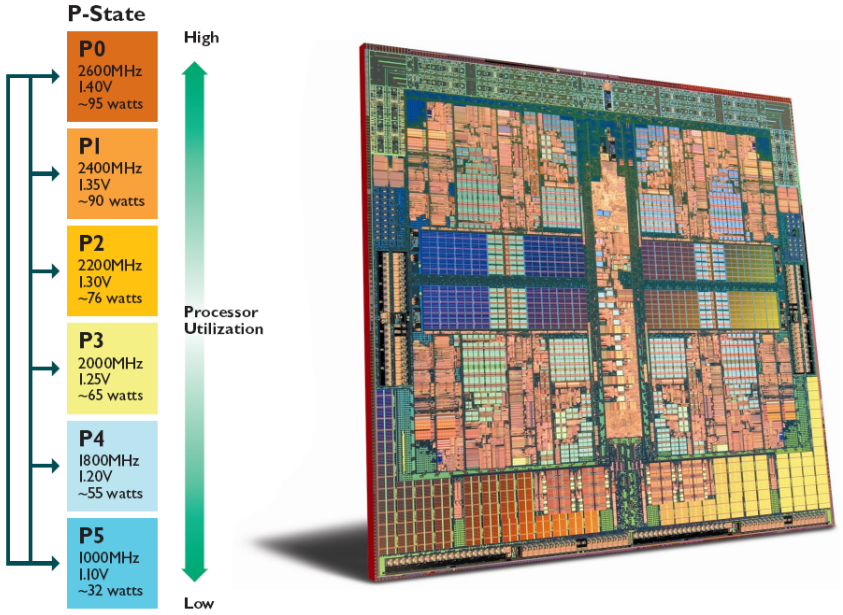

Figura 2.9: Esempio di P-State.

minimo e il più possibile ad alto livello. In altre parole, gli *autonomic computing system* si prefiggono di raggiungere degli obiettivi di QoS (*quality-of-service*) stabiliti negli SLA regolando in modo opportuno i propri parametri operativi chiave e richiedendo il minimo intervento umano possibile. Nella creazione di sistemi di questo tipo ogni componente del sistema deve essere a sua volta *autonomic*, ossia rispettare le medesime caratteristiche del sistema stesso, e deve essere inoltre in grado di interagire in modo autonomo con gli altri componenti che costituiscono il sistema (sempre con un intervento umano nullo o quasi). In particolare, la comunicazione tra i vari componenti deve avvenire unicamente attraverso apposite interfacce, le quali sono di diverse tipologie e possono consentire agli elementi di monitorarsi a vicenda (per verificare le rispettive prestazioni e capacità), di verificare lo stato di un certo elemento, di consentire la modifica delle *policy* dei singoli elementi ed infine di consentire agli elementi di entrare in relazione gli uni con gli altri in modo appropriato. La comunicazione tra i componenti del sistema avviene unicamente attraverso tali interfacce, dato che, per motivi di sicurezza ed autoprotezione del sistema stesso, non devono esistere canali di comunicazione diversi dalle interfacce stesse.

Elementi infrastrutturali caratteristici di un sistema autonomic sono:

- il *registro*, che memorizza l'elenco degli elementi connessi al sistema e

consente quindi agli elementi di riconoscersi e ritrovarsi nel sistema stesso;

- una *sentinella*, in grado di monitorare i servizi offerti dai vari elementi;
- un *aggregatore*, che combina due o più elementi presenti nel sistema per fornire servizi aggiuntivi;
- un *broker*, in grado di facilitare le interazioni tra vari elementi;
- un *negoziatore*, che fornisce assistenza per le transazioni più complesse tra gli elementi.

Come accennato, in sistema autonomico deve essere in grado di autoconfigurarsi con un minimo, preferibilmente nullo, intervento umano. Tale capacità viene garantita dalla presenza del registro, che è in grado di indicare ad ogni nuovo componente quelli già esistenti e viceversa, e da apposite regole che specificano i compiti dei singoli elementi. Ogni componente deve inoltre auto-controllarsi, ossia, verificando i propri input ed i propri output, deve essere in grado di individuare eventuali malfunzionamenti e prendere le necessarie contromisure. Per quanto concerne l'auto-ottimizzazione, l'architettura autonomica prevede la presenza di un *arbitro delle risorse* in grado di richiedere ai vari componenti le rispettive *utility function*, e di analizzarle per derivare la configurazione ottimale del sistema. Il compito dell'ottimizzatore non consta infatti nella ricerca di una configurazione tale da ottenere il funzionamento ottimale dei singoli componenti, bensì quello del sistema nella sua interezza. Inoltre ogni elemento deve essere in grado di offrire vari livelli di servizio e deve essere dotato di apposite interfacce per richiedere e offrire tali livelli durante le interazioni con il resto del sistema. Infine, un sistema di *autonomic computing* deve essere in grado di proteggersi sia dagli attacchi esterni, riconoscendo e riparando automaticamente eventuali malfunzionamenti da essi prodotti, sia da errori di configurazione, guasti interni e simili. Una caratteristica fondamentale di questo tipo di architettura è che l'intero sistema deve continuare a funzionare correttamente anche qualora uno o più componenti si dovessero guastare.

Tipicamente l'operatore umano interagisce con i sistemi autonomici attraverso le cosiddette *policy*, insiemi di regole ed obiettivi di funzionamento, che sono specificate sia per il sistema che per i suoi singoli componenti. Ogni componente deve sempre rispettare le proprie *policy* ed eventualmente può considerare nuove *policy* specificate da altri elementi del sistema, purché dotati dell'opportuna autorizzazione. Esistono tre tipi fondamentali di *policy*:

- le *action policy*: che specificano quali operazioni eseguire al verificarsi di determinate circostanze;
- le *goal policy*: che specificano quali dovrebbero essere gli obiettivi e le condizioni ottimali di funzionamento. Queste sono più potenti delle precedenti

poiché non richiedono un'approfondita conoscenza del funzionamento del sistema;

- le *utility function policy*: che specificano la "desiderabilità" (attraverso una scala numerica o una funzione matematica) degli stati possibili del sistema. Queste sono estremamente potenti perché consentono al sistema di scegliere quale è la condizione ottimale a seconda della situazione.

2.5.1 Autonomic Computing nei Service Center

Per poter rispettare i contratti SLA con i clienti e non incorrere in penali, i service provider hanno la necessità di poter sfruttare in modo ottimale le proprie risorse e di conseguenza massimizzare i guadagni. L'approccio comune prevede che il carico delle richieste venga distribuito tra le risorse disponibili così da fornire un servizio accettabile a tutti gli utenti che ne facciano richiesta.

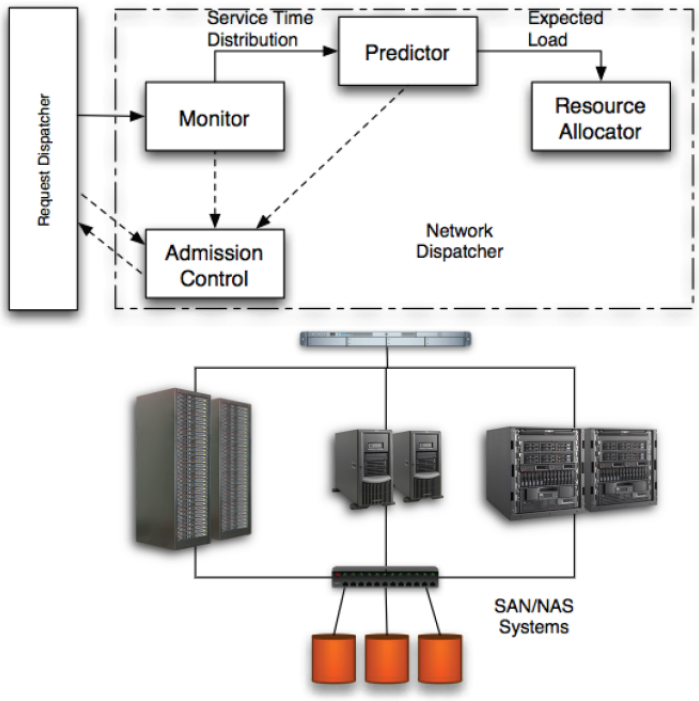

Figura 2.10: Esempio di un Autonomic System.

Data la natura variabile del numero di richieste di servizio è però difficile prevedere correttamente le risorse necessarie, tuttavia al tempo stesso dimensionare il sistema in base al caso peggiore è una strada non percorribile, dato che

comporterebbe un enorme spreco di risorse. Il paradigma di *autonomic computing* sembra quindi adattarsi perfettamente alle necessità della server farm di un service provider, grazie alla possibilità di auto-configurare al meglio i server al fine di servire le richieste accodate in quell'istante. La FIGURA 2.10 mostra i componenti principali di un Service Center che implementa una infrastruttura autonomica, le applicazioni sono allocate e deallocate su diversi server eterogenei dal *network dispatcher*. Ogni server può avviarsi con diversi sistemi operativi ed instanziare le applicazioni in base alle richieste del *network dispatcher*, i sistemi operativi, le applicazioni e tutti i dati sono resi disponibili invece attraverso un sistema di *storage networking*, di tipologia NAS[3] o SAN[4].
I principali componenti del *network dispatcher* sono:

- *admission controller*: che decide se accettare o rifiutare una nuova richiesta in base all'attuale carico di lavoro dei server;
- *monitor*: che misura il carico di lavoro e le metriche di performance di ogni applicazione, identifica le richieste per i diversi servizi e ne stima i tempi di risposta;
- *predictor*: che effettua previsioni sul futuro carico di lavoro del sistema in base allo storico delle richieste, fornitogli dal monitor;
- *resource allocator*: che determina la migliore configurazione del sistema in termini di capacity allocation delle applicazioni sui server ed instradamento delle richieste ai server disponibili.

2.6 Modelli prestazionali per Service Center

Come visto in precedenza due aspetti della gestione di un Service Center, fortemente legati fra loro, sono l'allocazione delle risorse ed il consumo energetico che essa comporta. Il *resource allocator* si occupa di determinare la migliore configurazione del sistema, così come l'assegnamento delle applicazioni ai server. Le sue decisioni dipendono dal carico previsto dal predittore, che fornisce una previsione di breve periodo, e da considerazioni off-line sulla distribuzione del carico nel lungo periodo.

Il gestore del consumo energetico si preoccupa invece di minimizzare la dissipazione di potenza dei server, considerando i vincoli sui tempi di risposta definiti negli SLA pattuiti con i clienti, i costi energetici (sia di alimentazione che di raffreddamento) e di logorio dei server. Per far questo il gestore ha a disposizione principalmente tre operazioni: accensione di nuovi server, spegnimento di server inutilizzati e controllo del *dynamic frequency scaling* (DFS).

[3]Network Attached Storage.
[4]Storage Area Network.

Verranno ora presentati alcuni modelli prestazionali basati su reti di code.

2.6.1 Modelli di prestazioni basati su reti di code

I Resource Allocator dei sistemi di Autonomic Computing si basano molto spesso su di un modello di valutazione delle prestazioni del sistema fisico, basato su *reti di code*, che viene integrato all'interno di un *framework* di ottimizzazione. Le *reti di code* sono un particolare approccio alla modellazione di un impianto informatico per la valutazione della qualità, in termini di prestazioni, del servizio erogato. Il sistema viene considerato come una collezione di centri di servizio, che rappresentano le risorse del sistema, e di clienti, che rappresentano utenti o transazioni, che si spostano da un centro di servizio ad un altro.

Una classificazione per i modelli a reti di code è quella di modelli e *aperti* e *chiusi*. Nel modello aperto, sono ammessi arrivi ed uscite da e verso l'esterno del sistema e non è possibile fare alcuna ipotesi sul numero di utenti presenti nel sistema, che può essere teoricamente infinito. Diversamente, nei modelli chiusi, la popolazione all'interno della rete di code è costante e non vi sono né ingressi né uscite nel sistema.

2.6.1.1 Modello chiuso per applicazioni Multi-Tier session-based

In [46], viene presentato un modello di un'applicazione Internet basato sulle reti di code, dove le code vanno a rappresentare i differenti tier applicativi. Il modello presentato è in grado di gestire applicazioni con un numero arbitrario di tier e con caratteristiche prestazionali significativamente differenti. La modellizzazione dell'applicazione multi-tier, operazione complessa, viene ridotta in questo caso alla modellizzazione del flusso delle richieste che procedono ai diversi tier e del flusso delle richieste attraverso i tier. Inoltre il modello considera e gestisce carichi di lavoro session-based.

Si assume che le applicazioni Internet vengano eseguite su un cluster di server e che ogni tier applicativo (o la replica di ogni tier) venga eseguito su server fisici dedicati. Data una specifica applicazione Internet, si assume che vengano specificati i requisiti di performance desiderati sotto forma di SLA, in particolar modo deve essere definito un limite superiore per il tempo di risposta medio accettabile per l'applicazione. Inoltre, inizialmente si assume che l'applicazione sia composta da M tier non replicati (T_1, ..., T_M), ognuno in esecuzione su un server fisico dedicato. Il sistema viene quindi modellato con una rete di M code (Q_1, ..., Q_M), ciascuna corrispondente ad un tier e, di conseguenza, al server fisico dedicato. Ogni coda deve inoltre adottare il processor sharing, dato che in questo modo è possibile approssimare la politica di scheduling impiegata dai sistemi operativi comunemente usati.

Quando una richiesta arriva al tier T_i, questo genera uno o più richieste al

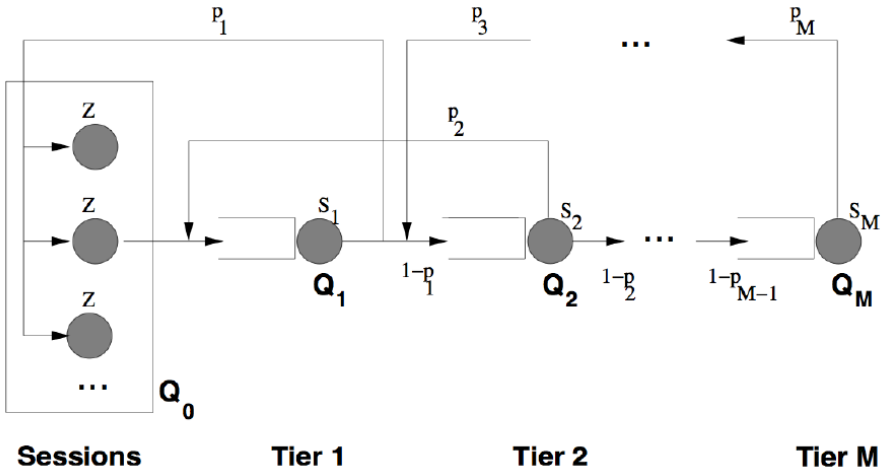

Figura 2.11: Modello chiuso per un'applicazione multi-tier.

tier successivo T_{i+1}, ognuna richiesta può inoltre effettuare molteplici visite ad ogni coda prima di risultare soddisfatta. Per modellare questo comportamento è possibile inserire una transizione che collega ogni coda a quella che la precede, come mostrato in FIGURA 2.11. Una richiesta, dopo un periodo di elaborazione nella coda Q_i, ritorna alla Q_{i-1} con una certa probabilità p_i oppure procede in Q_{i+1} con una probabilità pari a $(1 - p_i)$. Le uniche eccezioni riguardano l'ultima coda Q_M, dove tutte le richieste ritornano alla coda precedente, e la prima coda Q_1, dove una transizione alla coda precedente (Q_0) indica un completamento della richiesta. Va infine notato che il modello può gestire le visite multiple al medesimo tier senza preoccuparsi del fatto che esse avvengano in modo sequenziale o parallelo.

Per quanto riguarda gli effetti del caching, questi sono modellati in modo naturale dal modello. Nel caso il caching sia impiegato sul tier T_i, una cache hit fa immediatamente ritornare la richiesta alla coda precedente Q_{i-1}, senza innescare alcun lavoro sulle code successive a Q_i. L'impatto delle cache hit o delle miss può essere modellato determinando opportunamente la probabilità di transizione p_i ed il tempo di servizio della richiesta alla coda Q_i.

Per la modellazione delle sessioni degli utenti, viene usato un sistema ad accodamento di server infiniti, denotato con Q_0, che alimenta e forma il sistema di code chiuso mostrato in FIGURA 2.11. Si assume che ogni sessione attiva occupi una stazione in Q_0 e che il tempo speso in questa stazione sia pari al *think time* dell'utente, ossia il periodo di tempo che intercorre tra la soddisfazione di una richiesta e la successiva. Il sistema di server infiniti a livello Q_0 permette al modello di esprimere l'indipendenza dei *think time* degli utenti dai tempi di

servizio delle richieste stesse.

Per il calcolo del tempo di risposta medio è stato utilizzato l'algoritmo MVA. I parametri di input di questo, ossia tempi medi di servizio, fattori di visita, think time medi e numero di sessioni concorrenti, possono essere stimati attraverso il monitoraggio dell'esecuzione dell'applicazione. Si suppone infatti che i componenti software base del sistema operativo e delle applicazioni abbiano funzionalità tali da permettere la misurazione esatta di questi parametri.

Il fattore di visita (V_1, ..., V_M) per ogni coda Q_1, ..., Q_M corrisponde al numero medio dei casi in cui la coda è invocata durante l'elaborazione della richiesta, ossia da quando lascia Q_0 a quando vi fa ritorno, ignorando in quale ordine queste vengano effettuate. I fattori di visita possono essere calcolati a partire dalle probabilità di transizione p_1, ..., p_M, andando a calcolare il rapporto tra il numero di richieste servite da uno specifico tier in un periodo T ed il numero di richieste servite da un'intera applicazione nel medesimo orizzonte temporale. Una buona stima può essere ottenuta scegliendo un tempo di osservazione T adeguatamente grande. Il numero di richieste servite da uno specifico tier può invece essere determinato dall'elaborazione in real-time dei log dei tier, ad esempio, per il tier dati, il numero delle interrogazioni e delle transazioni processate durante il periodo T può essere determinato dall'analisi dei log del database. Il numero totale delle richieste servite, invece, può essere monitorato dalla sentry dell'applicazione.

I tempi medi di servizio possono essere stimati prelevando le informazioni sul *residence time* medio della richiesta, ossia il tempo speso dalla richiesta nel tier corrente e nei tier successivi che l'hanno elaborata, ottenute attraverso le funzionalità di *logging* fornite dai componenti dell'applicazione. In condizioni di carico leggero, il tempo di servizio dell'ultimo tier può essere stimato con il corrispondente *residence time*, dato che esso non invoca servizi di nessun altro tier. Per ogni altro tier, il tempo di servizio medio è invece funzione del *residence time* e del fattore di visita del tier in questione e di quello successivo. In condizioni di carico elevato su un tier, gli *overhead* software, come l'attesa per i *lock* ed il *paging* della memoria virtuale, e gli *overhead* del *context switch* diventano fattori non trascurabili nel calcolo del tempo di esecuzione della richiesta. Per cui in questo caso il tempo di servizio medio per il tier è ottenuto utilizzando la legge dell'utilizzo, ossia esso è pari al rapporto tra l'utilizzo della risorsa maggiormente occupata (CPU, disco, interfaccia di rete) ed il *throughput* del tier. Le informazioni sugli utilizzi delle risorse sono fornite dalle funzionalità di *performance monitoring* messe a disposizione dai moderni sistemi operativi, mentre il *throughput* può essere determinato valutando il numero di richieste completate nel periodo T attraverso l'analisi dei *log* oppure ottenendolo dal *dispatcher*.

La stima del *think time* medio può essere ottenuta monitorando nella *sentry*

i tempi di arrivo e di completamento delle singole richieste, non essendo altro che il tempo medio trascorso tra il completamento di una richiesta e l'arrivo, all'interno della medesima sessione, della richiesta successiva.

Attraverso un'ulteriore estensione del modello, possono venir gestiti tier replicati rimpiazzando la singola coda Q_i con r_i code, una per ciascuna replica. Una richiesta in qualunque coda può effettuare una transizione a qualsiasi altra coda del tier precedente o successivo. In questo scenario è necessario un *dispatcher* che permetta la distribuzione delle richieste alle repliche e che decida quale richiesta inoltrare a quale replica, influenzando in questo modo le transizioni effettuate dalla richiesta. Il *dispatcher* deve essere anche responsabile del bilanciamento del carico tra le varie repliche. Si assume che ci sia un perfetto bilanciamento del carico, ossia che ciascuna replica esegua $\frac{1}{r_i}$ del carico di lavoro del tier.

2.6.1.2 Modello aperto per applicazioni Multi-Tier session-based

In [5], il Service Center viene modellizzato come un sistema a reti di code composto da un insieme di server eterogenei con coda a classe multipla e da un *delay center*. I server rappresentano l'insieme di applicazioni atte a supportare le richieste di esecuzione, al contrario il delay center permette di rappresentare i ritardi che si verificano tra il completamento di una richiesta e l'arrivo della richiesta successiva all'interno della medesima sessione. Questi ritardi sono dovuti ai client, ossia ne rappresentano il *think time* (si faccia riferimento alla FIGURA 2.12).

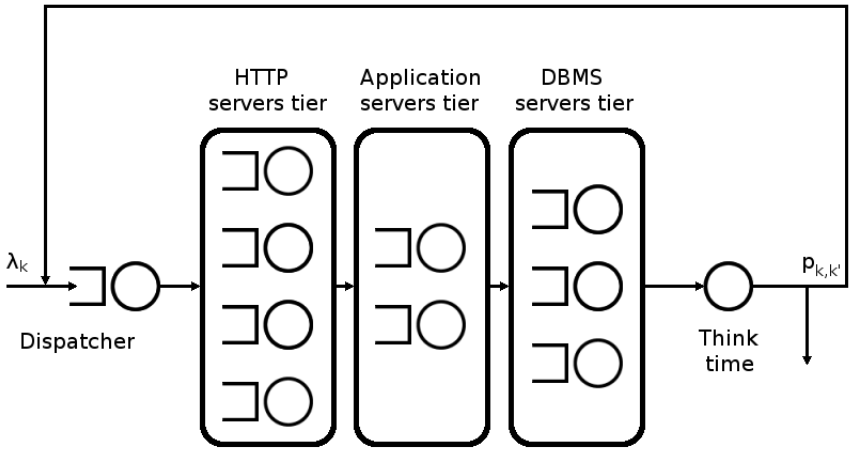

Figura 2.12: Modello aperto applicazione multi-tier.

Le sessioni degli utenti hanno inizio a seguito dell'arrivo di una richiesta di una certa classe k indirizzata al Service Center e proveniente da una sorgente esogena con frequenza λ_k. Una volta elaborata, la richiesta o torna al sistema come richiesta di classe k' con probabilità $p_{k,k'}$ o risulta completata con probabilità $1 - \sum_{l=1}^{K} p_{k,l}$. Con Λ_k viene indicata la frequenza complessiva di arrivo delle richieste di classe k, ossia $\Lambda_k = \sum_{k'=1}^{K} (\Lambda_{k'} p_{k',k}) + \lambda_k$.

2.7 Business Continuity

Per *business continuity* (BC) si intende la capacità di un'azienda di continuare ad esercitare il proprio business a fronte di eventi catastrofici che possono colpirla. La pianificazione della continuità operativa e di servizio si chiama business continuity plan (BCP) (in italiano "piano di continuità del business"). Questo piano identifica i potenziali pericoli e indica una struttura per fronteggiarli in caso di anomalie.

Ogni Service Center per gestire i propri rapporti con il resto del "mondo interconnesso" fa affidamento su uno o più sistemi informatici. Questi sistemi possono essere soggetti a malfunzionamenti come danneggiamenti, perdite di dati o accessi indiscriminati o non consentiti agli stessi. La BC è un processo che permette di fronteggiare tempestivamente ed accuratamente queste situazioni anomale al fine di garantire la continuità del lavoro dell'intera azienda e di supportare i processi di business. La BC si divide in due grandi campi d'azione, totalmente diversi l'uno dall'altro:

- *High Availability* (HA): è ciò che l'azienda si prefigge di ottenere dalla progettazione di un sistema, e dalla sua conseguente implementazione, in modo da assicurare un certo livello di continuità per un periodo prestabilito di tempo rimuovendo, quindi, i single-point-of-failure;
- *Disaster Recovery* (DR): è il processo che permette di riaccedere ai dati, all'hardware ed al software necessari per riprendere operazioni critiche di business in seguito ad un evento disastroso naturale o causato da un errore umano.

2.7.1 High Availability

Assicurare la continuità significa permettere ad un qualsiasi utente di accedere al sistema e di usufruirne senza interruzioni di grossa entità, nel caso l'utente non possa accedere al servizio, allora si dice che tale servizio non è disponibile (*unavailable*). Tuttavia va anche considerato che non tutti i periodi di inattività del sistema sono frutto di casualità, infatti questi periodi possono essere *planned* oppure *unplanned*. Nel primo caso, il periodo di non disponibilità è pianificato, infatti esso può essere dovuto alla necessità di effettuare riconfigurazioni,

aggiornamenti, o attività che richiedono si sospendere l'erogazione del servizio. Viceversa, un *unplanned downtime* è un periodo di inattività non preventivamente pianificato, che dipende eventi straordinari (malfunzionamenti hardware, software, ...)

L'*availability*, ossia la disponibilità di un componente, o di un insieme di componenti, è di solito espressa tramite una percentuale che indica l'intervallo di tempo in cui esso è stato ininterrottamente acceso e correttamente funzionante nell'arco di un anno e rappresenta la probabilità che il sistema stia funzionando in un generico istante di tempo t (a prescindere dal fatto che il componente abbia funzionato continuamente o che abbia subito guasti e sia stato riparato nell'intervallo $[0, t)$). Il modo tipico per rappresentare l'availability è quello di specificare il numero di nove che corrisponde al suo valore. Ad esempio, un availability a tre nove (*three nines*) si riferisce al valore 99.9 % che equivale ad una mancanza di disponibilità per 43.8 minuti/mese o 8.76 ore/anno.

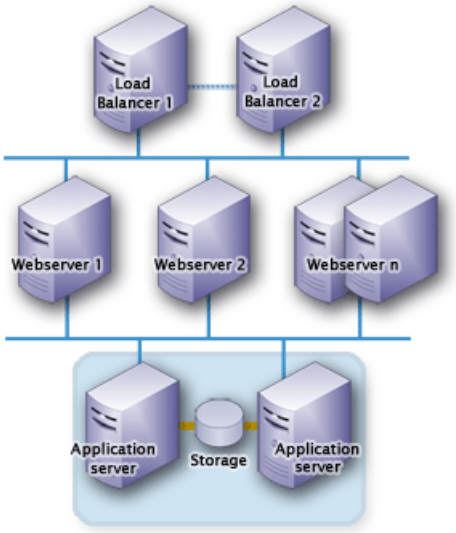

Figura 2.13: Un'architettura per garantire la disponibilità dei servizi, dove ogni componente è ridondato.

Un metodo per incrementare il livello di availability di un sistema è quello di ridondare i componenti che lo compongono, in modo che, se uno di questi smette di operare, gli altri componenti ridondanti possano sopperire alla sua mancanza senza compromettere così l'operatività del sistema. Sulla base di questo principio spesso quindi si parla di *cluster* di server (Figura 2.13), ossia di insiemi di server che mettono a disposizione le medesime funzionalità lavorando in parallelo tra loro. Normalmente, se un server che gestisce una particolare applicazione

si guasta, l'applicazione risulta non disponibile fino a quando qualcuno si prende carico del crash e lo risolve. Utilizzando un cluster di server è invece possibile allocare la medesima applicazione su più server e distribuirne il carico tra questi, in questo modo, a seguito di un malfunzionamento improvviso di uno dei server, i server rimasti possono farsi carico delle richieste di servizio precedentemente indirizzate al server guasto e mantenere l'operatività del sistema, in maniera spesso trasparente all'utente del servizio.
Configurazioni di questo tipo possono essere generalmente suddivise secondo i seguenti modelli:

- *Active/Active*: i vari nodi lavorano in parallelo. In caso di problemi, il traffico destinato al nodo collassato viene rediretto verso i rimanenti nodi. Questa configurazione è possibile solo se si ha una configurazione software omogenea sulle varie macchine;

- *Active/Passive*: per ogni nodo *attivo* (nodo primario) del sistema ne esiste un altro *passivo* in standby (nodo secondario). Se un nodo primario collassa, il corrispondente nodo secondario si attiva e si fa carico del lavoro del nodo collassato. Questa configurazione ovviamente richiede una grande quantità di hardware extra.

- *N+1*: è disponibile un singolo nodo extra che viene attivato nel caso un qualsiasi nodo collassi per sostituirlo. Il nodo extra deve avere caratteristiche tali da essere in grado di sostituire un qualsiasi nodo primario;

- *N+M*: soluzione attuata in cluster con molti servizi attivi contemporaneamente per i quali la soluzione $N + 1$ potrebbe non offrire sufficiente ridondanza. In questi casi si preferisce mantenere M nodi in standby, in modo da poter gestire anche il malfunzionamento contemporaneo di più di un server. Il numero di server extra è un compromesso tra costo architetturale ed esigenze di affidabilità;

- *N-to-1*: questa particolare configurazione è una variazione della *N+1*, infatti si ha sempre un solo nodo in standby che però rimane attivo solo temporaneamente, ossia finché il server collassato non viene ripristinato;

- *N-to-N*: è una combinazione dei modelli *Active/Active* e *N+M*, in cui i servizi della macchina collassata vengono redistribuiti sulle restanti macchine attive del cluster. Questa configurazione elimina la necessità di dover avere a disposizione nodi extra in standby, tuttavia richiede che ogni server del cluster possieda capacità elevata e non sia saturo, per poter gestire anche il carico in eccedenza dei server caduti.

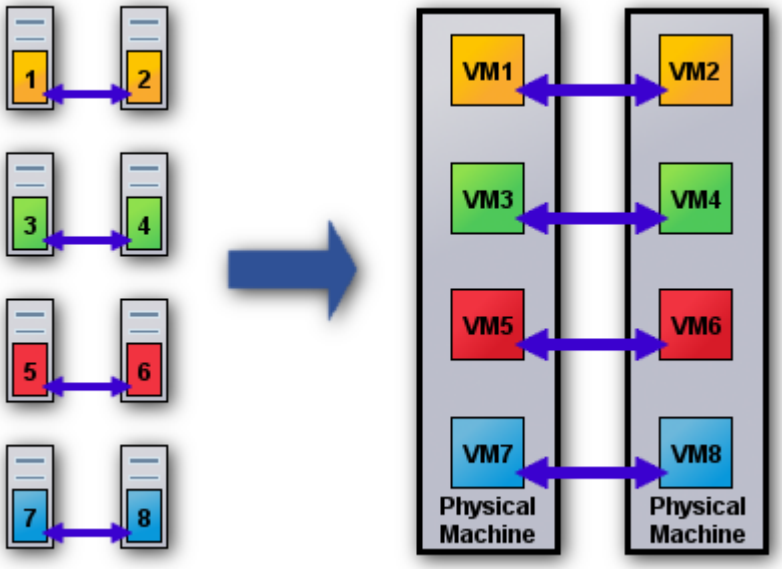

Figura 2.14: Tramite la virtualizzazione è possibile utilizzare solo due server con otto VM, anziché otto elaboratori.

La replicazione dei componenti come metodologia per aumentare il livello di availabiliy è un sistema molto costoso. Oggigiorno, però, la virtualizzazione apre nuove strade per ottenere elevati livelli di disponibilità. Grazie alla proprietà di indipendenza dall'hardware, dal sistema operativo e dalle applicazioni è possibile ridondare gli apparati infrastrutturali non più attraverso insiemi di server fisici, ma utilizzando macchine virtuali VM. In un Service Center non virtualizzato (come accennato nella Sezione 2.1), le applicazioni sono multi-tier ed in genere ad ogni tier è associata una specifica applicazione (*servlet engine*, *web server*, *DBMS*, ...). Spesso, per motivi principalmente di sicurezza, molte di queste applicazioni non possono risiedere sul medesimo server in quanto la compromissione di questo potrebbe permettere l'accesso di malintenzionati a dati riservati. L'allocazione su server diversi risulta quindi una scelta obbligata. Tuttavia, qualora si vogliano avere garanzie di availability, l'obbligo di allocare su server diversi le varie applicazioni comporta un incremento notevole dei costi per l'hardware, in quanto è necessario spesso dedicare un cluster di server a ciascuna applicazione.

L'indipendenza delle macchine virtuali consente di superare il problema dell'allocazione. Associando una sola applicazione a ciascuna virtual machine è possibile allocare sul medesimo server due o più applicazioni che non potrebbero coesistere nel medesimo ambiente operativo. Per incrementare l'availability

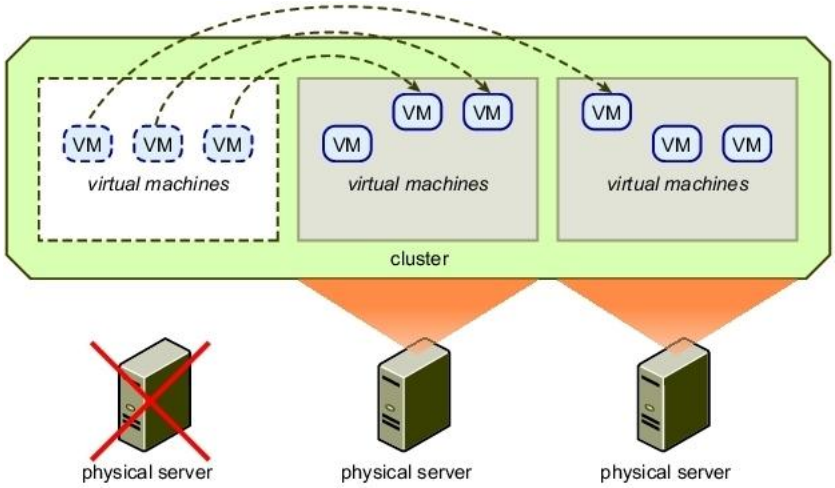

Figura 2.15: Migrazione di una virtual machine su un nuovo server a seguito di un guasto sul server che la ospitava in precedenza.

è quindi possibile replicare le virtual machine ed allocare queste su server differenti, ottenendo così un cluster di virtual machine. Questa nuova tipologia di cluster permette di ridurre i costi facendo in modo che le virtual machine appartenenti ad un cluster siano distribuite ognuna su un server diverso, ma permettendo a virtual machine di cluster diversi di coesistere su di un medesimo server. In sostanza quello che accade è che i cluster di virtual machine "simulano" cluster di server, senza richiederne lo stesso elevato numero di server e dando origine ad un *cluster virtuale* (si faccia riferimento alla FIGURA 2.14).

2.8 Tesniche per la gestione delle risorse in sistemi di Autonomic Computing

La riduzione del consumo energetico nelle strutture IT è un problema attualmente aperto, sono stati proposti in letteratura numerosi articoli scientifici. Inizialmente, molti lavori presentati hanno avuto l'obiettivo di ridurre il consumo in sistemi embedded. Ad esempio, in un sistema mobile di comunicazione, queste tecniche possono essere utilizzate per allungare il periodo di carica della batteria. Con il passare del tempo, è nata la necessità di introdurre modelli di risparmio energetico e di basso consumo anche in ambito server. In questo ambito (Green IT) tutti i più grandi player hanno preso parte e contribuito allo sviluppo di tecnologie (si faccia riferimento ad esempio a IBM's project

Big Green[5] e HP's Green up initiative 2[6]). Inoltre, sono state implementate tecniche di autonomic compunting per controllare le fluttuazioni di carico e per determinare il trade-of ottimo tra performance e costi energetici.

In letteratura, sono state considerate cinque principali categorie di problemi:

- application placement,
- admission control,
- capacity allocation,
- load balancing,
- energy consumption minimization.

Sono stati proposti principalmente tre approcci per risolvere questi problemi:

- anelli di retroazioni basati sulla teoria di controllo,
- tecniche di apprendimento adattivo,
- tecniche di ottimizzazione basate su funzioni di utilità.

Nel seguito si descriveranno ciascuno di questi approcci.

Il principale vantaggio della tecnica basata sulla teoria del controllo, è quello di garantire a design-time la stabilità nel sistema. Inoltre, in presenza di fluttuazioni del traffico, esiste la possibilità di seguire un preciso modello di comportamento durante il transitorio ed è possibile regolare la configurazione del sistema entro dei limiti temporali fissati in fase di progettazione. Tuttavia, queste tecniche sono tipicamente implementate da controllori locali e, conseguentemente, può essere raggiunta solo un'ottimizzazione di tale tipo, [2]. La maggior parte delle strategie di controllo adotta tecniche di identificazione del sistema per costruire modelli lineari ed invarianti nel tempo. Successivamente vengono utilizzati i classici controllori proporzionali integro-differenziali. In [36] è stato presentato un modello lineare a parametri variabili per il controllo delle performance di web server, utilizzando come variabile di controllo lo schema di DVFS. Gli autori di [26, 27] hanno implementato un controllore che effettua delle previsioni (limitate nel tempo) ed attraverso il quale può essere determinato un insieme di server da attivare ad una data frequenza e un'allocazione ottima delle virtual machine da eseguire sui vari elaboratori. Tuttavia, l'implementazione considera l'allocazione delle VM separata dal problema di capacity allocation ed inoltre non è dimostrata la scalabilità di tale approccio.

[5] http://www-03.ibm.com/press/us/en/photo/21514.wss
[6] http://www.hp.com/hpinfo/newsroom/feature-stories/2007/07-360-greenup.html

Recentemente, sono stati adottati i filtri Kalman per il tracciamento ed il controllo dell'utilizzo della CPU in ambienti virtualizzati, al fine di guidare l'allocazione della capacità operativa, [23]. Tuttavia, gli autori hanno considerato solo l'esecuzione di un solo tier applicativo per host. In [38] è stato proposto un controllore capace di coordinare differenti tipi di manager energetici in grado di gestire dalla singola CPU al cluster di server. Uno studio più recente [16] propone una gerarchia composta da due controllori. Il primo si occupa di regolare la frequenza di admission control in modo da evitare sovraccarichi mentre l'altro, che lavora a più basso livello, si preoccupa di allocare le risorse per l'elaborazione delle richieste. La ricerca si prefigge di mantenere un alto utilizzo delle risorse presenti nell'infrastruttura. In [37], viene proposto un sistema distribuito presso ciascun server che si occupa di allocare le VM sui server di un cluster al fine di massimizzare il profitto globale, prevedendo la relazione tra l'utilizzo della CPU e il consumo energetico. Tuttavia l'obiettivo del lavoro è quello di dimostrare la fattibilità del progetto e non quello di implementare un controllo efficacie che operi su di un'infrastruttura reale. Conseguentemente, i modelli energetici e di performance utilizzati risultano troppo semplici per essere accostati a situazioni reali. Gli autori di [22] hanno considerato sistemi a tre tier ed hanno implementato un controllore probabilistico per due dei tre tier che, attraverso l'utilizzo del metodo Robins-Monro per un approssimazione stocastica della tardeness (rapporto tra il response time e la deadline), permette l'elaborazione delle richieste entro una determinata deadline, minimizzando il consumo energetico attraverso il meccanismo di dynamic voltage scheduling. Infine, in [31] si considera l'obiettivo di abbattere i costi energetici dovuti alla climatizzazione degli ambienti contenenti gli elaboratori. In prima analisi viene presentata la progettazione di un modello termodinamico dell'infrastruttura di climatizzazione, in seguito viene sviluppato un controllore per la gestione dei flussi d'aria ed uno scheduler che tiene in considerazione il fattore termico. La limitazione di questo studio risiede nel fatto che non vengono presi in considerazione gli SLA e conseguentemente i profitti derivanti dal rispetto delle deadline di elaborazione.

Le tecniche di apprendimento si basano sul concetto di "esperienza". Tale esperienza viene acquisita monitorando sistemi correntemente in esecuzione, senza aver bisogno di un modello analitico. Per massimizzare i ricavi economici derivanti dagli SLA, in [42], è stata sviluppata una tecnica che permette di allocare le applicazioni su un insieme di elaboratori. In [24] gli autori hanno applicato tecniche di machine learning per coordinare più gestori indipendenti, ciascuno dei quali implementa un obiettivo differente. In particolare, il loro lavoro integra un gestore di performance ed uno energetico. Il fine è quello di minimizzare il consumo di energia, mantenendo un determinato livello di prestazioni. Un

vantaggio riconosciuto del machine learning è quello di poter catturare accuratamente il comportamento del sistema senza un esplicito modello di performance o di traffico, inoltre, non è necessaria un'approfondita conoscenza del sistema complessivo. Tuttavia, le sessioni di apprendimento tendono a durare diverse ore, [24]. Le tecniche proposte considerano, peraltro, un insieme limitato di applicazioni ed il loro controllo avviene applicando i vari meccanismi separatamente.

Gli approcci basati su funzioni di utilità vengono introdotti per ottimizzare il grado di soddisfazione dell'utente esprimendo gli obiettivi in termini di metriche di performance percepite a livello applicativo. Tipicamente, il sistema viene modellato tramite un modello di performance inglobato all'interno di un ottimizzatore. L'ottimizzazione può fornire una soluzione globale ottima o delle soluzione sub-ottime calcolate medianti euristiche, in base alla complessità del problema. La ricerca della soluzione ottima è tipicamente applicata a ciascuno dei cinque problemi considerati separatamente. Alcuni studi cercano di utilizzare un sistema di admission control per prevenire il sovraccarico dei server, [45, 15]. Infatti, il problema di capacity allocation è tipicamente risolto con un'attività di ottimizzazione separata che funziona sotto l'assunzione che i server non siano sovraccarichi. Altre ricerche si focalizzano sulla differenziazione dei servizi offerti dalle server farm attraverso il partizionamento fisico del Service Center in molteplici cluster, ciascuno dei quali serve determinate applicazioni, [44]. Le nuove tecnologie tendono ad evitare di condividere le risorse fisiche tra le richieste di classi differenti [4, 40, 41] supportando la virtualizzazione e la server consolidation. In [40] è stato presentato un framework virtualizzato capace di supportare carichi di richieste eterogenei (applicazioni batch e web interactive), senza però prendere in considerazione il risparmio energetico. Gli autori di [52] hanno descritto un metodo multi-layer e che opera su differenti scale temporali per la gestione di sistemi virtualizzati, senza però porre attenzione al trade-off tra performance e costi. Infine in [34], viene presentato un modello in cui si considerano dei vettori multidimensionali di risorse (CPU, hard disk, ...) ed il cui obiettivo è quello di massimizzare i profitti di ciascuna VM, minimizzando il numero di server usati.

Il lavoro [49] illustra i problemi di admission control e capacity allocation in un ambiente multi-tier virtualizzato. Purtroppo, le soluzioni non sono integrate e viene preso in considerazione solamente un sistema omogeneo. In fine, in [25], viene presentato un framework capace di coordinare gestori di risorse indipendenti attraverso nodi distribuiti. Recentemente, un sistema autonomic management è stato implementato in alcuni prodotti di virtualizzazione come VMWare DRS e Virtuoso VSched [40], con l'obiettivo di equalizzare l'uso del-

le risorse entro determinati trade-off tra costo e performance. Considerando l'ottimizzazione dei consumi energetici, gli autori di [39] hanno presentato il risultato di un progetto di server consolidation basato su un meccanismo di power budgeting. Le frequenze dei processori vengono variate per raggiungere obiettivi di utilizzo della CPU e di risparmio energetico. Una limitazione di questo approccio è quella di non poter garantire il rispetto degli SLA. Similmente, lo studio [11] fornisce un insieme di regole che permettono di considerare un budget energetico in un ambiente virtualizzato, proponendo nel contempo anche un accurato modello di predizione della media dei consumi.

In [9] si integrano i modelli basati su funzioni di utilità con tecniche di controllo per la gestione energetica. Tuttavia, sono considerati solo insiemi di server omogenei, tutti operanti alla stessa frequenza, dedicati a singole operazioni. Le tecniche presentate in [32, 41, 44] sono state implementate in un framework che costituisce parte del middleware IBM Tivoli.

In [44] il modello di riferimento delle performance è stato validato da esperimenti reali effettuati mediante benchmark. Viene illustrato un modello, per sistemi multi-tier, a reti di code chiuso, che consente di valutare le performance attraverso alcune metriche. Vengono valutati sia sistemi single-class, sia multi-class. Nel framework sviluppato dagli autori, il traffico in arrivo alle applicazioni viene bilanciato tra i server fisici. La politica di scheduling applicata a ciascun elaboratore è processor sharing. Il numero di server adottati per ciascun tier viene determinato da un algoritmo di ricerca locale con il quale si spostano server (omogenei) da un tier ad un altro, in modo da migliorare i tempi di risposte degli elaboratori saturi. Il modello di pricing delle richieste di servizio e la politica di admission control (in base alla programmazione dinamica) sono discussi in [32]. In fine, in [41], è proposto un metodo di allocazione delle applicazioni sui server. Il problema è ridotto ad una variante dei problemi multipli di zaino (multiple-knapsack problem). Viene inoltre presentata un'euristica molto efficiente con la quale si è in grado di ridurre il numero delle applicazioni che vengono accese o spente, mentre il carico viene bilanciato sulle macchine fisiche.

Tutti i metodi finora illustrati hanno la limitazione di determinare una soluzione in tempi comparabili con le scale temporali dell'Autonomic Computing solo per piccole istanze di problemi. Nei Service Center attualmente costruiti si arrivano a dover gestire un ampio numero di server (circa 10000) e una molteplicità di applicativi. I modelli appena elencati non permettono risoluzioni in tempi utili di così ampie infrastrutture. Recentemente la comunità scientifica di ricerca sta iniziando a sviluppare tecniche di gestione gerarchica, ma quest'area di ricerca risulta essere ancora acerba. In [50] è stato proposto un algoritmo gerarchico per catturare in modo efficace alcune metriche di consumo energe-

tico e prestazionali. Algoritmi decentralizzati sono stati proposti per sistemi cloud. Gli autori di [3], ad esempio, hanno proposto un'architettura di controllo distribuito dove, attraverso dei local manager, viene deciso se e presso altro quale sito spostare le Virtual Machine attualmente allocate. Infine, in [43] viene presentato un modello gerarchico di ottimizzazione che cerca di garantire tutte le proprietà self-*, caratteristiche dei sistemi autonomici. Il modello raggruppa le varie applicazioni del sistema in application manager (AM), che vengono controllati da un central manager (CM). Il CM ha il compito di assegnare un determinato quantitativo di risorse ad ogni AM. Ciascun AM esegue un algoritmo di ottimizzazione locale e restituisce il valore della sua funzione obiettivo al CM. Quest'ultimo, basandosi sui vari valori delle funzioni obiettivo ricevute dagli application manager, cerca di variare la quantità di risorse agli AM al fine di ottimizzare la funzione obiettivo globale. Il lavoro risulta completo, anche se molto generale ed è alla base di questo lavoro di tesi. Tale modello verrà meglio specificato nel Capitolo 4 per rispettare gli obiettivi di risparmio energetico e di ottimizzazione dei revenue.

Capitolo 3

Tecniche di resource Allocation ottima con garanzie di availability

In questo capitolo vengono descritte le tecniche di Resource Allocation sviluppate in due precedenti lavori di tesi ([8], [14]) che costituiscono il punto di partenza del presente lavoro.

Nella prima parte del capitolo, precisamente nelle Sezioni 3.1 *3.2, viene descritto il modello del sistema. Nella* Sezione 3.3 *il problema di Resource Allocation viene formulato come un modello di programmazione non lineare intera mista. Questo modello prende in considerazione tutte le variabili rilevanti nella gestione dei moderni Service Center. Le procedure euristiche sviluppate sono infine presentate nelle* Sezioni *dalla* 3.4 *alla* 3.6.

Nella seconda parte del capitolo verranno introdotti vincoli di availability relativi alle classi di servizio. L'availability è infatti una dimensione di qualità molto rilevante che assume sempre maggiore importanza con la pratica di server consolidation. Nella Sezione 3.7 *viene proposta una nuova formulazione del problema di ottimizzazione che introduce in modo esplicito i vincoli di availability.*

3.1 Sistema di Resource Allocation

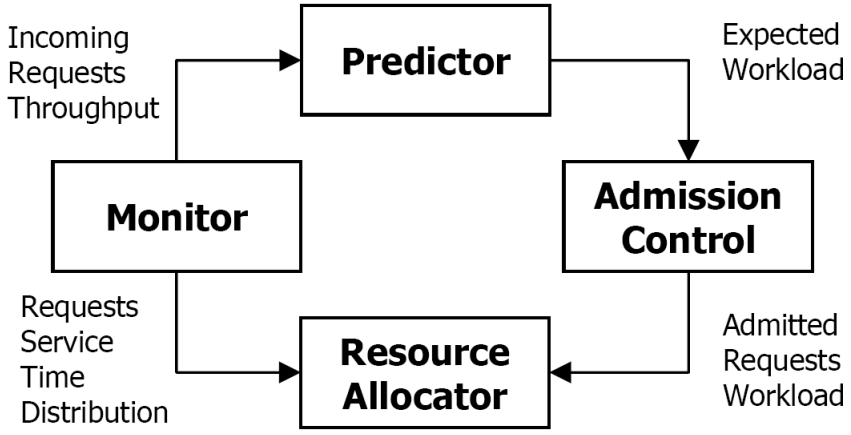

Figura 3.1: Infrastruttura di un Autonomic System.

Un sistema di Resource Allocation (FIGURA 3.1) effettua l'allocazione delle risorse basandosi su delle previsioni di carico, le quali sono ottenute basandosi principalmente sul valore del carico sostenuto dal Service Center nei periodi precedenti. In altre parole, esso riceve queste previsioni, che sono considerate valide per un certo periodo T, le elabora e quindi procede all'allocazione delle risorse, eventualmente applicando politiche di Admission Control[1]. Una volta terminata questa allocazione, il Service Center dovrà gestire l'effettivo carico di richieste che giungeranno al sistema durante il periodo T considerato, carico che potrà o meno discostarsi dal valore previsto. Durante l'elaborazione delle richieste effettive, quindi durante il periodo T, verranno comunque raccolte informazioni (tempi di risposta effettivi, valore reale di carico, ecc...) da parte del componente definito *Monitor*. Tali informazioni andranno ad aggiungersi a quelle raccolte in precedenza così da raffinare ulteriormente le previsioni di carico future. Come si può capire, il sistema di Resource Allocation lavora ciclicamente riferendosi a periodi temporali, ossia riceve previsioni di carico per il periodo successivo, determina l'allocazione più profittevole in base a queste previsioni, alloca le risorse nel periodo corrispondente ed attende le previsioni per il periodo successivo.

[1] Meccanismo che decide se accettare o rifiutare le richieste in base all'attuale carico di lavoro per mantenere un determinato livello di QoS

3.2 Modello di prestazioni del sistema

Viene ora introdotto il modello di prestazioni per applicazioni multi-tier session-based descritto in Sezione 2.6.1.2. Tale modello permette di rappresentare Service Center che sfruttano le tecniche di virtualizzazione per l'allocazione delle applicazioni su ciascun server e che utilizzano la politica di scheduling Generalized Processor Sharing (GPS, Sezione 2.6.1.1). Ad ogni classe di richieste è inoltre associata un'utility function lineare (illustrata in Figura 2.5) che specifica il ricavo (o la penale) previsto in base al tempo di risposta ottenuto.

Nel lavoro di tesi si userà la seguente notazione:

K	:=	numero di classi di richieste;
N_k	:=	numero complessivo di tier applicativi coinvolti nell'esecuzione delle richieste di classe k;
M	:=	numero complessivo di server nel Service Center;
H_i	:=	insieme delle frequenze operative per il server i;
$C_{i,h}$	:=	capacità del server i operante alla frequenza h ($h \in H_i$);
$c_{i,h}$	:=	costo per unità di tempo associato al server i se acceso ed operante alla frequenza h ($h \in H_i$);
$A_{i,k,j}$	:=	termine pari ad 1 se il server i può supportare il tier applicativo j per la classe k, 0 altrimenti;
$\mu_{k,j}$	:=	massima frequenza di elaborazione delle richieste per il tier applicativo j della classe k sostenibile da un server di capacità 1;
$R_{i,k,j}$	:=	tempo medio di risposta del server i a richieste per il tier applicativo j della classe k;
cs_i	:=	overhead causato dall'accensione del server i;
cm	:=	overhead causato dal trasferimento di una virtual machine su un server differente.

La matrice di compatibilità $A_{i,k,j}$, cui ci si riferirà in seguito con il nome di Matrice dei vincoli di tipo $\mathcal{A}$, può essere utilizzata per dedicare server specifici all'esecuzione di particolari classi e tier applicativi. In altre parole specifica se una certa coppia classe-tier applicativo possa risiedere o meno su un determinato server, indipendentemente dagli altri tier applicativi che vi risiedono. Il termine H_i, rappresenta l'insieme delle frequenze operative alle quali ciascun server i può funzionare. Inoltre, dato che, sia il costo sia la capacità di un server variano al

variare della sua frequenza operativa (o meglio, al variare del suo p-state, come spiegato in 2.4.1 i termini utilizzati per indicare capacità e costo di ciascun server, $c_{i,h}$ e $C_{i,h}$, permettono di associare, per ogni server i, un valore di costo ed un valore di capacità a ciascuna frequenza operativa del server, ossia ad ogni frequenza presente in H_i.

I termini cs_i e cm esprimono rispettivamente l'overhead introdotto nel sistema a seguito dell'accensione di un server e quello dovuto al trasferimento di una virtual machine da un server ad un altro.

Come visto, il funzionamento del sistema di Resource Allocation segue un andamento periodico, nel senso che esso produce soluzioni per il periodo successivo basandosi su delle previsioni di carico. Dato che il sistema ha già elaborato l'allocazione ottimale per il periodo corrente, sarebbe controproducente ricercare una soluzione completamente nuova, anzi è spesso invece consigliato "derivare" la nuova soluzione basandosi su quella del periodo precedente. Questo è ancor più evidente quando le previsioni di carico per il periodo successivo non si discostano di molto dai valori di carico corrente. Diventa quindi importante considerare che esistono dei costi e degli overhead associati sia all'accensione di nuovi server sia allo spostamento di applicazioni tra server.

Infine, limitare gli spostamenti delle applicazioni tra server può garantire una maggiore stabilità dell'intero sistema. Nel modello vengono introdotte le seguenti variabili di decisione:

x_i := variabile binaria che vale 1 se il server i è acceso, 0 altrimenti;

$\lambda_{i,k,j}$:= frequenza di arrivo sul server i delle richieste per il tier applicativo j della classe k;

$\phi_{i,k,j}$:= percentuale di allocazione del processore del server i per il tier applicativo j della classe k;

$z_{i,k,j}$:= variabile binaria che vale 1 se il tier applicativo j per la classe k è allocato sul server i;

$f_{i,h}$:= variabile binaria che vale 1 se il server i sta operando alla frequenza h, con $h \in H_i$.

Le variabili $f_{i,h}$ permettono di esprimere la frequenza cui ciascun server sta correntemente operando. Esse sono legate alle variabili x_i dal vincolo $\sum_{h \in H_i} f_{i,h} = x_i$ il quale impone che, nel caso il server i sia acceso ($x_i = 1$), debba essere selezionata una e una sola frequenza operativa tra quelle disponibili per il server. Viceversa, nel caso il server i sia spento ($x_i = 0$), nessuna frequenza

deve essere selezionata.

Data la relazione che lega costo e capacità di un server alla sua frequenza operativa, per rappresentare la capacità messa a disposizione dal server i, andrà utilizzata l'espressione: $\sum_{h \in H_i} C_{i,h} f_{i,h}$, che permette di esprimere l'effettiva capacità fornita del server, determinata dalla frequenza cui sta operando. Allo stesso modo, l'espressione: $\sum_{h \in H_i} c_{i,h} f_{i,h}$ esprime l'effettivo costo del server i.

L'analisi dei sistemi di code multi-classe è notoriamente difficile, per questo motivo si è utilizzata una formula approssimata (sviluppata in [51]) per riuscire ad ottenere un'espressione in forma chiusa dei tempi medi di risposta dello scheduling GPS. Secondo la politica GPS, la capacità di un server i dedicata all'elaborazione di una richiesta di classe k per il tier applicativo j al tempo t è data da:

$$\sum_{h \in H_i} C_{i,h} f_{i,h} \cdot \frac{\phi_{i,k,j}}{\sum_{\hat{k} \in \mathcal{K}(t)} \sum_{j=1}^{N_{\hat{k}}} \phi_{i,j,\hat{k}}},$$

dove con $\mathcal{K}(t)$ si indica l'insieme delle classi per cui sono presenti richieste pendenti sul server i al tempo t. Le richieste ai diversi tier applicativi per ogni classe e per ogni server possono essere elaborate con una politica First In First Out (FIFO) oppure con una politica di Processor Sharing (PS). Nel primo caso si assume che le richieste di classe k indirizzate ad un server i abbiano una distribuzione esponenziale con media $(C_i \cdot \mu_{k,j})^{-1}$, nel secondo caso invece le richieste di classe k per il server i si assumono distribuite secondo una distribuzione generale con media$(C_i \cdot \mu_{k,j})^{-1}$. Nell'approssimazione utilizzata, ogni coda multi-classe su ogni singolo server è scomposta in più code indipendenti di tipo singola-classe singolo-server con capacità maggiore o uguale a $\sum_{h \in H_i} C_{i,h} f_{i,h} \cdot \mu_{k,j} \cdot \phi_{i,k,j}$. La somma dei tempi di risposta valutati nelle singole code per ogni classe fornisce di conseguenza un limite superiore per il corrispondente tempo di risposta della classe nel sistema. Sotto queste ipotesi si ha che $R_{i,k,j}$, ossia il tempo di risposta medio del server i per l'elaborazione di richieste per il tier applicativo j della classe k, può essere valutato come ([51]):

$$R_{i,k,j} = \frac{1}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}}. \tag{3.1}$$

Di conseguenza il tempo di risposta medio aggregato per la classe k può essere valutato come somma dei tempi di risposta medi per ciascun tier applicativo j della classe calcolati su tutti i server, ovvero come:

$$R_k = \frac{1}{\Lambda_k} \cdot \left(\sum_{i=1}^{M} \sum_{j=1}^{N_k} \lambda_{i,k,j} \cdot \frac{1}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}} \right).$$

3.3 Il problema di ottimizzazione

L'obiettivo è massimizzare la differenza tra i ricavi derivanti dagli SLA, utilizzando utility function lineari come quella in FIGURA 2.5, ed i costi dovuti al mantenimento dei server accesi ad una data frequenza. Questa differenza può essere espressa come:

$$(P1) \qquad \max \sum_{k=1}^{K} \left(-m_k \cdot \sum_{i=1}^{M} \sum_{j=1}^{N_k} \frac{\lambda_{i,k,j}}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}} \right) + \\ - \sum_{i=1}^{M} \left(\sum_{h \in H_i} c_{i,h} f_{i,h} \right) - \sum_{i=1}^{M} cs_i \cdot \max\left(0, x_i - \overline{x}_i\right) + \\ - \sum_{i \in M, k \in K, j \in N_k} cm \cdot \max\left(0, z_{i,k,j} - \overline{z}_{i,k,j}\right) \quad (3.2)$$

Gli ultimi due addendi della (3.2) permettono di tenere in considerazione gli overhead introdotti nel sistema rispettivamente a causa dell'accensione di server ed a causa degli spostamenti di virtual machine tra server. In questi due addendi i termini $\overline{x}_i$ e $\overline{z}_{i,k,j}$ rappresentano rispettivamente i valori delle corrispondenti variabili x_i e $z_{i,k,j}$ nel periodo di controllo precedente considerato (SEZIONE 3.1).

Il problema di ottimizzazione può essere formulato come:

$$\max \sum_{k=1}^{K} \left(-m_k \cdot \sum_{i=1}^{M} \sum_{j=1}^{N_k} \frac{\lambda_{i,k,j}}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}} \right) + \\ - \sum_{i=1}^{M} \left(\sum_{h \in H_i} c_{i,h} f_{i,h} \right) - \sum_{i=1}^{M} cs_i \cdot \max\left(0, x_i - \overline{x}_i\right) + \\ - \sum_{i \in M, k \in K, j \in N_k} cm \cdot \max\left(0, z_{i,k,j} - \overline{z}_{i,k,j}\right) \quad (3.3)$$

$$\sum_{i=1}^{M} \lambda_{i,k,j} = \Lambda_k; \qquad \forall k, j \tag{3.4}$$

$$\sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \leq 1; \qquad \forall i \tag{3.5}$$

$$\sum_{j \in \mathcal{B}_k} z_{i,k,j} \leq 1; \qquad \forall i, k \tag{3.6}$$

$$z_{i,k,j} \leq A_{i,k,j} x_i; \qquad \forall i, k, j \tag{3.7}$$

$$\lambda_{i,k,j} \leq \Lambda_k z_{i,k,j}; \qquad \forall i, k, j \tag{3.8}$$

$$\lambda_{i,k,j} < \left(\sum_{h \in H_i} C_{i,h} f_{i,h} \right) \mu_{k,j} \phi_{i,k,j}; \qquad \forall i, k, j \tag{3.9}$$

$$\sum_{h \in H_i} f_{i,h} = x_i; \qquad \forall i \tag{3.10}$$

$$\phi_{i,k,j} \geq 0; \qquad \forall i, k, j$$

$$\lambda_{i,k,j} \geq 0; \qquad \forall i, k, j$$

$$x_i,\ z_{i,k,j},\ f_{i,h} \in [0,1]; \qquad \forall i, k, j;\ \forall h \in H_i$$

L'equazione (3.4) impone che, per ogni tier applicativo, la somma del traffico di classe k assegnato ai server del tier debba essere uguale al carico complessivo di richieste della medesima classe, ossia che ogni richiesta debba essere soddisfatta a livello di ciascun tier. Il vincolo (3.5) stabilisce che è possibile assegnare al più il 100% di capacità di ogni singola macchina. Il vincolo (3.7) indica la possibilità di assegnare una coppia classe-tier applicativo ad un server solo nel caso questo sia acceso e non sia violato alcun vincolo di tipo $\mathcal{A}$. Il vincolo (3.8) impone che del carico associato ad una certa coppia classe-tier applicativo possa essere indirizzato ad un certo server solo se la corrispondente applicazione è stata allocata sul server in questione. Il vincolo (3.9) garantisce che le singole code non siano in saturazione, mentre il vincolo (3.10) esprime il legame già accennato in precedenza tra l'accensione di un server ed il funzionamento una data frequenza. Infine l'espressione (3.6), cui ci si riferisce con il "Vincoli di tipo $\mathcal{B}$" ed in cui $\mathcal{B}_k$ è un sottoinsieme degli indici contenuti in $\{j | 1 \leq j \leq N_k\}$ per la classe k, permette di descrivere le incompatibilità esistenti tra le applicazioni per evitare che applicazioni "incompatibili" vengano allocate sul medesimo server. Per esempio un servlet engine può essere allocato sulla medesima macchina fisica che esegue un web server od un'instanza di application server, tuttavia in genere un DBMS dovrà essere allocato su una macchina diversa per questioni di sicurezza e di modalità di gestione. Questa situazione può essere caratterizzata,

ad esempio, con il seguente sistema di disequazioni:

$$z_{i,k,1} + z_{i,k,3} \leq 1$$
$$z_{i,k,1} + z_{i,k,4} \leq 1$$
$$z_{i,k,2} + z_{i,k,4} \leq 1$$
$$z_{i,k,3} + z_{i,k,4} \leq 1$$

Come si evince dalle prime due disequazioni il *web server* (il primo tier applicativo) può condividere un server solo con un *servlet engine* (il secondo tier applicativo) essendogli vietata la convivenza sia con l'*application server* (terzo tier applicativo) che con il *DBMS* (quarto tier applicativo). Come si può notare poi dalle ultime tre disequazioni del sistema il *DBMS* dovrà obbligatoriamente essere allocato su un server dedicato. Il gruppo di disequazioni introduce dei vincoli di implicazione, ordinando le applicazioni, dato che se una applicazione di tier j non può essere instanziata sullo stesso server di una applicazione di tier j' con $j' > j$, allora l'applicazione di tier j non può parimenti essere instanziata insieme ai tier j'' per ogni $j'' > j'$ (per esempio, se un *web server* non può essere allocato con un *application server*, lo stesso *web server* non può essere allocato neanche con un DBMS). Questi vincoli indicano dunque le applicazioni e le classi che possono coesistere o meno sullo stesso server al momento dell'instanziazione, e combinati con i vincoli $\mathcal{A}$ garantiscono una estrema adattabilità del sistema, potendo rappresentare ogni situazione possibile.

Essendo x_i, $\lambda_{i,k,j}$, $\phi_{i,k,j}$ e $z_{i,k,j}$ le variabili decisionali, il problema posto è di programmazione non lineare intera mista. Anche nel caso in cui il numero dei server accesi sia fisso e le applicazioni siano preventivamente assegnate ai server, il problema congiunto di load balancing e capacity allocation rimane comunque complesso, dal momento che la funzione obiettivo del problema non lineare corrispondente non risulta né concava né convessa (si faccia riferimento a [8] per la dimostrazione).

3.4 Algoritmo di ottimizzazione

I tool commerciali riescono a risolvere il problema ($P1$) solo per piccole istanze, poco significative rispetto alle dimensioni dai Service Center realmente esistenti. Da questa necessità nasce l'esigenza di implementare un algoritmo che trovi euristicamente la soluzione al problema.

L'algoritmo è composto da due fasi. La prima prevede la ricerca di una soluzione iniziale attraverso un'euristica greedy, mentre la seconda ha come obiettivo quello di migliorare la soluzione corrente attraverso un algoritmo di ricerca locale (local search) che esplora lo spazio delle soluzioni "vicine". L'idea di base della tecnica di local search è quella di modificare iterativamente la soluzione

iniziale considerando, ad ogni passo, delle soluzioni "adiacenti" a quella corrente costruite modificandola tramite l'ausilio di "mosse".

3.4.1 Ricerca di una soluzione iniziale a partire da una soluzione precedente

```
1  Inizializza l'insieme dei carichi residui R;
2  for i = 1 to M do
3  |  for k = 1 to K do
4  |  |  for j = 1 to N_k do
5  |  |  |  Aggiorna il valore del parametro di load balancing
   |  |  |  λ_{i,k,j} = (Λ_k / Λ̄_k) · λ̄_{i,k,j};
6  |  |  |  if λ_{i,k,j} > (C_{i,h̄} f_{i,h̄}) μ_{k,j} φ_{i,k,j} Ūt then
7  |  |  |  |  if h̄ ≠ h_{max,i} then
8  |  |  |  |  |  h̄ = h_{max,i};
9  |  |  |  |  end
10 |  |  |  |  if λ_{i,k,j} > (C_{i,h̄} f_{i,h̄}) μ_{k,j} φ_{i,k,j} Ūt then
11 |  |  |  |  |  r = λ_{i,k,j} − (C_{i,h̄} f_{i,h̄}) μ_{k,j} φ_{i,k,j} Ūt;
12 |  |  |  |  |  if ∄Γ_{k,j} ∈ R then
13 |  |  |  |  |  |  Γ_{k,j} = r;
14 |  |  |  |  |  |  Aggiungi Γ_{k,j} ad R;
15 |  |  |  |  |  else
16 |  |  |  |  |  |  Γ_{k,j} = Γ_{k,j} + r;
17 |  |  |  |  |  end
18 |  |  |  |  |  λ_{i,k,j} = (C_{i,h̄} f_{i,h̄}) μ_{k,j} φ_{i,k,j} Ūt;
19 |  |  |  |  end
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24 Applica l'ALGORITMO 2 all'insieme dei carichi residui R;
```

Algoritmo 1: Algoritmo di allocazione delle applicazioni sui server a partire dalla soluzione del periodo di controllo precedente.

Come visto in precedenza, dato il funzionamento periodico del sistema di Resource Allocation, potrebbe essere utile "derivare" la soluzione del successivo periodo di controllo basandosi su quella disponibile per il periodo corrente. Inoltre sarebbe anche opportuno cercare di limitare il più possibile le differenze tra le due soluzioni per quanto riguarda, ad esempio, la distribuzione delle applicazioni e l'insieme dei server usati, in modo da minimizzare i costi di overhead

1 Ordina le applicazioni con del carico non ancora assegnato ($k \in \mathcal{R}$) per numero decrescente di vincoli $\mathcal{B}$ e per valori decrescenti di $\Gamma_{k,j}$;
2 Ordina i server disponibili secondo il rapporto $\frac{c_i}{C_i}$;
3 Sia i l'indice del primo server acceso che può supportare l'applicazione $\Gamma_{k,j}$; Fissa $x_i = 1$ e la capacità residua $r_i = C_i \frac{\sum_{k=1}^{K} \Lambda_k N_k}{\sum_{k=1}^{K} \Lambda_k (\sum_{j=1}^{N_k} \frac{1}{\mu_{k,j}})}$;
4 $\forall j, k$ fissa $demanding_{k,j} = \Gamma_{k,j}$;
5 **for** $k = 1$ *to* K **do**
6 **if** $\exists$ *un server* i *tale che l'applicazione* $\Gamma_{k,j}$ *può essere allocata sul server* $i \wedge x_i$*=1* **then**
7 $y_{i,k,j} = \min(demanding_{k,j}, r_i)$;
8 $r_i = r_i - y_{i,k,j}$;
9 $demanding_{k,j} = demanding_{k,j} - y_{i,k,j}$;
10 **if** $demanding_{k,j} = 0$ **then**
11 $j = j + 1$;
12 **end**
13 **if** $j = N_k + 1$ **then**
14 $k = k + 1$; $j = 1$;
15 **end**
16 **else**
17 Accendi un altro server. Sia i l'indice del primo server che può supportare l'applicazione $\Gamma_{k,j}$; Fissa $x_i = 1$ e $r_i = C_i \frac{\sum_{k=1}^{K} \Lambda_k N_k}{\sum_{k=1}^{K} \Lambda_k (\sum_{j=1}^{N_k} \frac{1}{\mu_{k,j}})}$;
18 **end**
19 **end**

Algoritmo 2: Algoritmo di allocazione del carico residuo sui server.

che un numero eccessivo di differenze farebbe lievitare. Per raggiungere questi obiettivi è stato implementato in [8] un algoritmo che, in caso di variazioni del carico delle richieste per le classi di servizio, partendo dalla precedente soluzione cerca di applicare il minor numero possibile di modifiche alla soluzione per quanto riguarda l'accensione di nuovi server e la riallocazione di applicazioni (Algoritmo 1). L'obiettivo è quello di introdurre, se serve, nuovi server con utilizzo al più pari a $\overline{Ut}$. Inizialmente (Passo 5) l'algoritmo calcola per ogni server i acceso e per ogni coppia classe-applicazione allocatavi (k, j) dei nuovi parametri di load balancing secondo l'espressione:

$$\lambda_{i,k,j} = \frac{\Lambda_k}{\overline{\Lambda}_k} \cdot \overline{\lambda}_{i,k,j}$$

dove Λ_k indica il nuovo valore di carico delle richieste di classe k, $\overline{\Lambda}_k$ rappresenta il carico del periodo di controllo precedente ed infine $\overline{\lambda}_{i,k,j}$ rappresenta il vecchio

parametro di load balancing. Nel caso il carico precedente fosse stato nullo ($\overline{\Lambda}_k = 0$) non saranno presenti server con il compito di elaborare richieste per la classe in questione, per cui l'intero carico associatovi andrà assegnato in una fase successiva e, per fare questo, viene quindi inserito nell'insieme dei carichi da assegnare $\mathcal{R}$ un termine $\Gamma_{k,j} = \Lambda_k, \quad \forall j \in \{1..N_k\}$. Si procede quindi alla ricerca di possibili violazioni dei vincoli di utilizzo sui server (Passo 6), ovvero casi in cui per un server i con allocata una certa coppia classe-applicazione (k, j) si abbia:

$$\lambda_{i,k,j} > (C_{i,\overline{h}} f_{i,\overline{h}}) \mu_{k,j} \phi_{i,k,j} \overline{Ut}$$

dove $\overline{h}$ indica la corrente frequenza operativa del server i. In questi casi l'algoritmo, se possibile, porta il server i alla massima frequenza supportata (Passo 8) e verifica nuovamente il vincolo. Nel caso la violazione permanga, o il server stia già operando alla massima frequenza consentita, l'algoritmo (Passo 11) assegna il massimo carico ammissibile al server (pari a $\tilde{\lambda}_{i,k,j} = (C_{i,\overline{h}} f_{i,\overline{h}}) \mu_{k,j} \phi_{i,k,j} \overline{Ut}$) ed aggiorna l'insieme $\mathcal{R}$ aggiungendovi il carico che non è stato possibile assegnare (ossia $\Gamma_{k,j} = \lambda_{i,k,j} - \tilde{\lambda}_{i,k,j}$). In particolare, per quanto riguarda l'aggiornamento di $\mathcal{R}$, se nell'insieme è già presente un termine Γ associato alla medesima coppia classe-applicazione, il valore di questo termine viene aggiornato aggiungendovi il carico in eccesso appena riscontrato (Passo 16), in caso contrario si aggiungerà all'insieme un nuovo termine con valore pari al carico in eccesso (Passi 13 e 14).

Dopo aver tentato di gestire il nuovo carico incrementando le frequenze dei server, l'algoritmo verifica se l'insieme $\mathcal{R}$ risulta vuoto. In caso affermativo si è ottenuta una nuova soluzione iniziale a partire dalla precedente senza che sia stato necessario spostare applicazioni o accendere nuovi server. In caso contrario, l'algoritmo (Algoritmo 2) tenterà nuovamente di allocare il carico in eccesso utilizzando solamente i server già accesi e procedendo all'instanziazione di nuove applicazioni su questi. Infatti nella fase precedente, l'incremento delle frequenze operative dei server avrà prodotto un aumento delle capacità elaborative dei server, liberando capacità che sarà possibile utilizzare per l'allocazione di nuove applicazioni. L'algoritmo, dopo aver applicato una procedura euristica che ordina le classi per numero numero crescente di vincolo $\mathcal{B}$ e per numero crescente di $\Gamma_{k,j}$ (Passo 1) ed i server per $\frac{c_i}{C_i}$ (Passo 2), procede quindi cercando di assegnare il carico associato ad ogni termine $\Gamma_{k,j}$ ai server già accesi, compatibilmente con i vincoli $\mathcal{A}$ e $\mathcal{B}$ ed i vincoli di utilizzo (Passi 3 e 4). In particolare, una volta individuato un server i adatto ad accogliere il carico associato ad un termine $\Gamma_{k,j}$, l'algoritmo calcola il massimo valore di carico che il server può supportare (Passi 7 e 8), pari a:

$$\lambda_{i,k,j,max} = (\sum_{h \in H_i} C_{i,h} f_{i,h}) \mu_{k,j} \overline{Ut} \cdot (1 - \sum_{(\overline{k},\overline{j}) \in \mathcal{P}_i} \phi_{i,\overline{k},\overline{j}})$$

dove $\mathcal{P}_i$ rappresenta l'insieme delle coppie classe-applicazione già allocate sul server. Come si può notare, il principale termine che vincola questo valore è l'ultimo termine nell'espressione, il quale impedisce alla somma dei parametri di capacity allocation di violare il vincolo di saturazione del server (3.5). Si prefigurano quindi due casi: il primo in cui il valore massimo di carico supportato dal server permette di allocarvi l'intero carico $\Gamma_{k,j}$ (Passo 11), ossia $\Gamma_{k,j} \leq \lambda_{i,k,j,max}$, il secondo invece quando ciò non è possibile. Nel primo caso l'algoritmo procede impostando i parametri di load balancing e di capacity allocation del server nel seguente modo:

$$\begin{aligned} \lambda_{i,k,j} &= \Gamma_{k,j} \\ \phi_{i,k,j} &= \frac{\Gamma_{k,j}}{(\sum_{h \in H_i} C_{i,h} f_{i,h}) \overline{Ut} \mu_{k,j}} \end{aligned}$$

ossia allocando l'intero carico residuo sul server i e derivando il corrispondente parametro di load balancing. Inoltre, poiché l'intero carico residuo è stato allocato con successo, l'algoritmo provvede a rimuovere dall'insieme $\mathcal{R}$ il termine $\Gamma_{k,j}$.

Nel secondo caso andando ad assegnare l'intero carico residuo al server si andrebbe a violarne il vincolo di saturazione, per cui sarà possibile allocarvi solo una parte del carico, pari a $\lambda_{i,k,j,max}$. Si effettueranno quindi i seguenti assegnamenti:

$$\begin{aligned} \lambda_{i,k,j} &= \lambda_{i,k,j,max} \\ \phi_{i,k,j} &= (1 - \sum_{(\overline{k},\overline{j}) \in \mathcal{P}_i} \phi_{i,\overline{k},\overline{j}}) \\ \Gamma_{k,j} &= \Gamma_{k,j} - \lambda_{i,k,j,max} \end{aligned}$$

ovvero si assegnerà al server il massimo carico possibile, si imposterà il parametro di capacity allocation al massimo valore possibile evitando di violare il vincolo di saturazione ed infine si aggiornerà il valore del carico residuo ancora da assegnare presente nell'insieme $\mathcal{R}$. Ovviamente in questo caso il restante carico in eccesso dovrà essere allocato su un server differente, inoltre a questo punto il server i sarà saturato.

Questi passi saranno reiterati più volte fino a che l'insieme $\mathcal{R}$ non sarà vuoto (soluzione ammissibile) oppure conterrà ancora dei termini $\Gamma_{k,j}$, ma non saranno più disponibili nuovi server (soluzione non ammissibile). In caso di soluzione non ammissibile è possibile comunque ricercare una soluzione iniziale completamente nuova, in modo da trascurare i vincoli introdotti derivando la nuova soluzione dalla precedente (utilizzo degli stessi server accesi, utilizzo degli stessi parametri λ e ϕ, ecc..), tuttavia in questo modo si dovranno considerare i costi e gli overhead descritti in precedenza ed ovviamente non si avrà comunque la certezza

di trovare una soluzione ammissibile (specie in caso di elevati aumenti di carico). Per approfondimenti, si faccia riferimento a [8].

3.5 I problemi di load balancing e capacity allocation

Una volta ottenuto un assegnamento iniziale delle applicazioni ai server, la politica di capacity allocation su ciascun server ed il load balancing delle richieste possono essere ulteriormente migliorati. In [14] è stato dimostrato che il problema di massimizzazione dei revenue è equivalente a di minimizzare la media pesata dei tempi di risposta per le richieste di ogni classe: questo problema può essere risolto anche utilizzando software di ottimizzazione non lineare, ma solo per piccole istanze. Nel caso le dimensioni del problema siano maggiori si richiede un approccio differente. Nella soluzione proposta in [14] è stata implementata una procedura di *fixed point iteration* che trova iterativamente l'ottimo per un sottoinsieme di variabili mantenendo fisso il valore delle variabili restanti, nel caso in questione i due sottoinsiemi di variabili corrispondono alle variabili λ ed alle variabili ϕ. Le espressioni in forma chiusa di queste variabili vengono ricavate mediante l'applicazione delle condizioni KKT.

3.5.1 *Fixed point iteration* basata sulle condizioni KKT

Sia $\mathcal{I}$ l'insieme dei server accesi. Se si considerano fissi i valori delle variabili $\phi_{i,k,j} = \overline{\phi}_{i,k,j}$, ossia la politica di capacity allocation è fissa per ogni server, il problema di load balancing può essere formulato come segue:

$$\min \sum_{k=1}^{K} m_k \sum_{i \in \mathcal{I}} \sum_{j=1}^{N_k} \frac{\lambda_{i,k,j}}{C_i \mu_{k,j} \overline{\phi}_{i,k,j} - \lambda_{i,k,j}} \tag{3.11}$$

$$0 \leq \lambda_{i,k,j} < C_i \mu_{k,j} \overline{\phi}_{i,k,j}; \forall i \in \mathcal{I}, \forall k, j$$

$$\sum_{i \in \mathcal{I}} \lambda_{i,k,j} = \Lambda_k; \forall k, j \tag{3.12}$$

$$\lambda_{i,k,j} \leq A_{i,k,j} \Lambda_k; \forall i \in \mathcal{I}, \forall k, j \tag{3.13}$$

dove $\lambda_{i,k,j}$ sono le variabili decisionali. Si noti che il vincolo (3.13) è equivalente a $\lambda_{i,k,j} > 0 \Leftrightarrow A_{i,k,j} = 1$. Essendo fissati i parametri di capacity allocation il problema (3.11) diventa separabile ed i $\sum_{k=1}^{K} N_k$ sottoproblemi di load balan-

cing ottenibili (uno per ciascuna coppia classe-tier applicativo) possono essere risolti indipendentemente l'uno dall'altro. Inoltre, dato che la funzione obiettivo è in questo caso convessa[2], è possibile ottenere la soluzione ottima per ogni singolo sottoproblema.

In [14]è stata ottenuta una formula chiusa per la soluzione del problema (3.11) di load balancing. In particolare, trascurando gli indici del tier k, j e ponendo $U_i = C_i \cdot \mu \cdot \phi_i$[3] è possibile esprimere il carico ottimo da assegnare ad un server i in base a quello di un determinato server, ad esempio 1:

$$\lambda_i = U_i - \sqrt{\frac{U_i}{U_1}}(U_1 - \lambda_1)$$

e in cui:

$$\lambda_1 = \frac{\Lambda - \sum_{i \in \mathcal{I}-\{1\}}(U_i - \sqrt{U_1 U_i})}{\sum_{i \in \mathcal{I}} \sqrt{\frac{U_i}{U_1}}} \quad .$$

Una volta calcolato questo valore è possibile calcolare in sua funzione il valore di ogni altra variabile λ_i.

In [14] è stato dimostrato che il problema di load balancing può essere risolto effettuando $O(M^2 \cdot \sum_{k=1}^{K} N_k)$ operazioni.

Si consideri ora il problema di capacity allocation. Considerando fissate le variabili di load balancing, ossia $\lambda_{i,k,j} = \overline{\lambda}_{i,k,j}$, il problema di assegnamento delle risorse può essere formulato come segue:

$$\min \sum_{k=1}^{K} m_k \sum_{i \in \mathcal{I}} \sum_{j=1}^{N_k} \frac{\overline{\lambda}_{i,k,j}}{C_i \mu_{k,j} \phi_{i,k,j} - \overline{\lambda}_{i,k,j}} \tag{3.14}$$

$$\phi_{i,k,j} \geq 0$$
$$\phi_{i,k,j} > \frac{\overline{\lambda}_{i,k,j}}{C_i \mu_{k,j}}$$

$$\sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \leq 1 \tag{3.15}$$

In questo caso le variabili decisionali sono le $\phi_{i,k,j}$, l'obiettivo rimane quello

[2] La matrice Hessiana è data da $Diag\left(\frac{2m_k C_i \mu_{k,j} \overline{\phi}_{i,k,j}}{(C_i \mu_{k,j} \overline{\phi}_{i,k,j} - \lambda_{i,k,j})^3}\right)$ e gli autovalori sono positivi.

[3] U_i rappresenta la capacità allocata sul generico server i per gestire il carico associato a k, j.

di minimizzare la media pesata dei tempi di risposta. Essendo fissato il load balancing, il problema (3.14) diventa separabile ed è possibile risolvere M sottoproblemi di capacity allocation indipendenti. Anche in questo caso inoltre la funzione obiettivo è convessa[4], per cui è possibile trovare la soluzione ottima per ogni sottoproblema.

In [14] è stata ottenuta una formula chiusa per la soluzione del problema (3.14) di capacity allocation. In particolare, rispetto al tier $1,1$:

$$\phi_{k,j} = \frac{1}{U_{k,j}} \left(\sqrt{\frac{m_k U_{k,j} \overline{\lambda}_{k,j}}{m_1 U_{1,1} \overline{\lambda}_{1,1}}} (U_{1,1}\phi_{1,1} - \overline{\lambda}_{1,1}) + \overline{\lambda}_{k,j} \right) \tag{3.16}$$

Con $\phi_{1,1}$ pari a:

$$\phi_{1,1} = \frac{\overline{\lambda}_{1,1}}{U_{1,1}} + \frac{1 - \sum_{k=1}^{K} \sum_{j=1}^{N_k} \frac{\overline{\lambda}_{k,j}}{U_{k,j}}}{\sum_{k=1}^{K} \sum_{j=1}^{N_k} \sqrt{\frac{m_k U_{1,1} \overline{\lambda}_{k,j}}{m_1 U_{k,j} \overline{\lambda}_{1,1}}}} \quad .$$

Dopo aver calcolato $\phi_{1,1}$ è possibile ottenere il valore delle restanti variabili applicando la (3.16). Il problema di capacity allocation può essere risolto effettuando al più $O(M \cdot \sum_{k=1}^{K} N_k)$ operazioni.

La *fixed point iteration* risolve quindi iterativamente i problemi di load balancing e di capacity allocation. Non è possibile garantire che l'algoritmo converga ad un ottimo globale, ma è possibile dimostrare che l'algoritmo converge sempre. Infatti le funzioni (3.11) e (3.14) sono convesse, quindi le soluzioni ottime dei problemi corrispondenti possono essere trovate. Ad ogni passo l'assegnamento corrente delle λ e delle ϕ viene migliorato fino al raggiungimento di un ottimo locale.

3.6 L'algoritmo di ricerca locale

La variabile decisionale principale del sistema è indubbiamente l'insieme dei server accesi, poiché ogni server acceso influisce direttamente sui costi e sulle prestazioni dell'intero sistema. Per ogni singolo server attivato le variabili $\phi_{i,k,j}$ e $\lambda_{i,k,j}$ costituiscono variabili di *fine tuning*, e mediante la *fixed point iteration* è possibile ottenere un ottimo locale per la data configurazione di server accesi. Basandosi su questa considerazione, in [14] è stata proposta un'esplorazione dello spazio delle soluzioni attraverso le mosse illustrate in seguito. In teoria sarebbe necessario eseguire la *fixed point iteration* per ogni passo dell'esplorazione

[4] La matrice Hessiana è data da $Diag\left(\frac{2m_k C_i^2 \mu_{k,j}{}^2 \overline{\lambda}_{i,k,j}}{(C_i \mu_{k,j} \phi_{i,k,j} - \overline{\lambda}_{i,k,j})^3}\right)$ e gli autovalori sono positivi.

così da valutare quale tra gli ottimi locali esplorati sia effettivamente il migliore. Tuttavia una volta eseguita una prima *fixed point iteration*, per raggiungere l'ottimo locale a partire dallo stato iniziale, si può ragionevolmente pensare che in condizioni di carico medio-basso il load balancing ottimo venga solo perturbato dallo spegnimento o dall'accensione di un singolo server, e quindi la nuova soluzione sia sufficientemente vicina al suo ottimo locale.

L'algoritmo di ricerca locale (ALGORITMO 3) prevede la possibilità di applicare alla soluzione corrente una delle sei mosse che l'algoritmo è in grado di compiere. Ogni mossa viene valutata singolarmente (PASSO 5) partendo dalla soluzione corrente. Una volta individuata una mossa profittevole essa viene applicata (PASSO 7) e l'algoritmo continua con l'esplorazione del nuovo intorno. Qualora nessuna mossa permette di migliorare la soluzione viene eseguita nuovamente una *fixed point iteration* (PASSO 9) e quindi l'algoritmo tenta nuovamente di applicare le mosse. Se anche in questo caso non è possibile migliorare la soluzione l'algoritmo termina e la soluzione corrente è un ottimo locale.

```
1  Esegui la fixed point iteration sulla soluzione corrente;
2  repeat
3  |  repeat
4  |  |  Memorizza il valore corrente della soluzione corrente e poni
   |  |  Δ = 0;
5  |  |  Valuta, per ogni mossa ammissibile a partire dalla soluzione
   |  |  corrente, il Δ della funzione obiettivo;
6  |  |  Seleziona la mossa ammissibile migliore (max Δ e Δ > 0);
7  |  |  Applica la mossa alla soluzione corrente;
8  |  until Δ > 0;
9  |  Esegui la fixed point iteration sulla soluzione corrente;
10 |  Calcola l'incremento Δ della funzione obiettivo;
11 until Δ > 0;
```

Algoritmo 3: Algoritmo di ricerca locale.

3.6.1 Spegnimento di un server

Questa mossa seleziona un singolo server tra quelli correntemente accesi e lo spegne andando a riallocare il suo carico sui server accesi rimanenti. Poiché la ricerca locale è soggetta a vincoli di tempo, non è possibile controllare ogni server per verificare se sia opportuno o meno spegnerlo, per questo motivo vengono controllati solo il server con utilizzo minimo Ut_i e tutti quei server che hanno utilizzo inferiore a $\alpha \cdot \min Ut_i$, dove α è una costante (sperimentalmente

determinata tra 1,1 e 1,2). L'utilizzo del generico server i viene calcolato come:

$$Ut_i = \frac{1}{|\mathcal{K}|} \cdot \sum_{k,j \in \mathcal{K}} \frac{\lambda_{i,k,j}}{C_i \mu_{k,j} \phi_{i,k,j}}$$

dove $\mathcal{K}$ corrisponde all'insieme delle classi e delle applicazioni attualmente presenti sul server i ed ovviamente $|\mathcal{K}|$ ne rappresenta la cardinalità, ovvero il numero complessivo di coppie classe-applicazione correntemente allocate sul server i.

Per ogni server $\hat{i}$ preso in considerazione va inoltre verificato che sia possibile effettivamente spegnerlo. Può infatti accadere che un server sia rimasto il solo tra quelli accesi a poter supportare una determinata classe od applicazione, e quindi, non potendo riallocare sui restanti server il carico in questione, il server non potrà essere spento. Infatti per ogni server $\hat{i}$ candidato allo spegnimento è necessario poterne riallocare il carico, e per poter far questo è possibile calcolare due matrici che rappresentino la capacità residua dei server in funzione delle coppie classe-applicazione:

$$S_{i,k,j} = C_i \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j} \tag{3.17}$$

$$S_{k,j} = \sum_{i=1, i \neq \hat{i}}^{M} S_{i,k,j} \tag{3.18}$$

Il carico del server $\hat{i}$ verrà quindi riallocato tra i server accesi e compatibili rimanenti, proporzionalmente alla loro capacità residua, secondo la formula:

$$\tilde{\lambda}_{i,k,j} = \lambda_{i,k,j} + \lambda_{\hat{i},k,j} \cdot \frac{S_{i,k,j}}{S_{k,j}}$$

Una volta calcolati gli elementi delle due matrici, prima di poter procedere alla riallocazione è necessario verificare che la riallocazione del carico non provochi la saturazione di alcuno dei server su cui si effettuerà la riallocazione, ossia andrà controllato che:

$$\tilde{\lambda}_{i,k,j} < C_i \mu_{k,j} \phi_{i,k,j} \quad \forall \tilde{\lambda}_{i,k,j}$$

Facendo le opportune sostituzioni si ottiene chela condizione che deve essere verificata è:

$$\lambda_{\hat{i},k,j} < S_{k,j} \tag{3.19}$$

L'espressione (3.19) fornisce quindi un vincolo unico per quanto riguarda la riallocazione del carico, infatti se la disequazione è soddisfatta per ogni coppia classe-applicazione presente sul server candidato allo spegnimento $\hat{i}$ si può procedere alla riallocazione senza ulteriori verifiche, in caso contrario la riallocazione non è fattibile ed è quindi necessario procedere alla valutazione del server

successivo.

Una volta effettuato l'assegnamento del nuovo carico $\tilde{\lambda}_{i,k,j}$ per tutti i server accesi compatibili, il server $\hat{i}$ viene inserito nell'insieme dei server spenti e viene valutato il nuovo valore della funzione obiettivo. Se questo valore permette di migliorare il risultato, la soluzione viene memorizzata ed è una soluzione candidata a diventare la successiva base di partenza. L'algoritmo proposto per questa mossa ha complessità pari a $O(M \cdot \sum_{k=1}^{K} N_k)$ nel caso pessimo in cui tutti i server siano accesi, abbiano il medesimo utilizzo e le applicazioni siano compatibili con tutti i server.

3.6.2 Accensione di un server

Questa mossa individua il server maggiormente utilizzato ed effettua una ricerca tra i server correntemente spenti alla ricerca del server compatibile e con le caratteristiche migliori. Una volta trovato, lo accende, assegnandogli una parte del carico del server maggiormente utilizzato. Anche in questo caso per ragioni di tempo non è possibile valutare per la ripartizione del carico ogni singolo server acceso, quindi ci si limita a considerare il server acceso con utilizzo Ut_i massimo (ossia il server *bottleneck* del sistema) e tutti quei server che hanno utilizzo superiore a $\beta \cdot \max Ut_i$, dove β è una costante (sperimentalmente determinata tra 0,9 e 0,95). Per ognuno dei server $\hat{i}$ considerati vengono valutati tutti i server spenti, una volta trovato un server che, se acceso, permette di migliorare la soluzione, si verifica che questo possa condividere il carico con il server *bottleneck* in esame, ossia che supporti tutte le applicazioni e le classi attualmente allocate sul server $\hat{i}$. Ovviamente il server candidato all'accensione non deve necessariamente supportare tutte le classi e le applicazioni che possono potenzialmente essere supportate dal server $\hat{i}$, per poter condividere il carico attuale è infatti sufficiente che siano supportate solo le classi e le applicazioni correntemente allocate sul server.

Individuato il primo server $\tilde{i}$ adatto, ossia per cui non sono possibili violazioni dei vincoli di tipo $\mathcal{A}$, si calcola la nuova ripartizione del carico ([14]). Per i parametri di capacity allocation ϕ del server candidato all'accensione $\tilde{i}$ si useranno i medesimi valori delle corrispondenti ϕ del server con cui verrà condiviso il carico $\hat{i}$.

L'algoritmo proposto per questa mossa ha una complessità di $O(M^2 \cdot \sum_{k=1}^{K} N_k)$ nel caso pessimo in cui tutti i server abbiano il medesimo utilizzo, tutti i server supportino tutte le applicazioni per tutte le classi e metà dei server sia accesa. Tuttavia questo caso è poco frequente e di conseguenza la complessità del caso medio è molto inferiore.

Per quanto riguarda la gestione del DFS, in [8] è stata introdotta una semplice gestione delle frequenze operative per quanto riguarda il server da accendere. Infatti, quando si sta verificando se una certo server sia in grado o meno di

garantire un miglioramento della soluzione, nel caso siano disponibili più frequenze operative differenti, si ricerca anche la frequenza più opportuna con cui accendere il server. In particolare si considerano, quando possibile, fino a tre frequenze, ossia la massima, la minima ed una intermedia. Infine, in caso di ricerca della soluzione a partire da quella del periodo di controllo precedente, si dovrà considerare che l'accensione del server comporta un costo aggiuntivo.

3.6.3 Riallocazione di applicazioni

Questa mossa utilizza un approccio diverso rispetto alle precedenti poiché non accende né spegne server, ma tenta di riallocare classi all'interno dei server già accesi in cerca di soluzioni maggiormente profittevoli. Spesso infatti la *fixed point iteration*, durante la soluzione del problema di load balancing, tende a "spegnere" una o più classi su qualche server. Questo avviene andando a fissare le corrispondenti variabili λ a zero e di conseguenza la corrispondente coppia-applicazione sul server non riceve più richieste. La *fixed point iteration* tuttavia può solo "spegnere" classi e applicazioni sui server, poiché per costruzione non è invece in grado di attivarne di nuove, anche nel caso ciò sia profittevole. Infatti se un parametro λ sul server è posto a zero, anche il corrispondente parametro ϕ viene posto a zero e di conseguenza l'applicazione non verrà mai riallocata nuovamente sul server. Inoltre va considerato che andando a "spegnere" una o più coppie classe-applicazione su un server è possibile che vengano "liberati" dei vincoli di tipo $\mathcal{B}$, e sia quindi possibile allocare sul server applicazioni che in precedenza avrebbero violato i suddetti vincoli. Questa mossa si occupa quindi di trovare le coppie classe-applicazione allocabili (e non correntemente allocate) su ogni singolo server acceso e di valutare il miglioramento dato dalla riallocazione di ciascuna di esse.

Per riattivare una classe di servizio è necessario rendere disponibile della capacità di elaborazione su un server. Si considerano quindi tutti i server con utilizzo $Ut_i < \hat{Ut}$, dove $\hat{Ut}$ rappresenta un utilizzo massimo (determinato sperimentalmente a 0,6).

L'algoritmo proposto ha complessità nel caso pessimo $O(M^2 \cdot (\sum_{k=1}^{K} N_k)^2)$, ovvero quando la metà dei server è sotto l'utilizzo fissato, l'altra metà ha utilizzo invece superiore ed infine tutte le classi sono riattivabili su tutti i server.

Per quanto riguarda il DFS e gli overhead dovuti all'instanziazione di nuove VM (SESSIONE 3.3), valgono considerazioni simili a quelle fatte circa la mossa di spegnimento di un server. L'allocazione di una una certa coppia class-tier applicativo, per come è stata progettata questa mossa, comporta sempre l'overhead dovuto all'instanziazione di una nuova VM su di un server che invece ne era privo nella soluzione precedente.

3.6.4 Mosse introdotte dal DFS

Le mosse presentate in questa sezione sfruttano il meccanismo di Dynamic Frequency Scaling.

3.6.4.1 Incremento di frequenza

Questa mossa, come quella di riallocazione, non va a modificare il numero di server accesi od a spostare VM, ma tenta invece di migliorare la soluzione incrementando la frequenza operativa di un server. Nel caso infatti si abbia un server con un valore medio-alto di utilizzo e che non stia operando alla sua frequenza massima, incrementandone la frequenza, e di conseguenza la capacità, si potrebbero ottenere dei miglioramenti per quanto riguarda i tempi di risposta per le classi che vi sono allocate, a discapito di un incremento dei costi accettabile (il costo di un server è infatti proporzionale al cubo della sua frequenza operativa corrente). La mossa di incremento di frequenza effettua quindi una ricerca tra i server accesi e per ciascuno di essi (o solo per un certo numero, in base a dei parametri di inizializzazione) verifica se il server stia già operando alla massima frequenza possibile, in caso contrario l'algoritmo verifica se, incrementando la frequenza del server al valore corrispondente al p-state successivo, sia possibile ottenere un miglioramento della soluzione.

La complessità della mossa è pari $O(M \cdot \sum_{k=1}^{K} N_k)$ nel caso pessimo in cui tutti i server siano accesi, non stiano operando alla massima frequenza e che su ogni server siano allocate tutte le possibili coppie classe-applicazione.

3.6.4.2 Decremento di frequenza

Questa mossa si comporta esattamente all'opposto della mossa di incremento. Se in quest'ultima si cercava di ridurre i tempi di risposta per le classi allocate su un server ad utilizzo medio-alto aumentando la capacità del server, nella mossa di decremento della frequenza si tenta invece di ridurre il costo d'uso di server con un utilizzo medio-basso. Infatti è possibile che il vantaggio dato dalle maggiori prestazioni dovute all'alta frequenza operativa (tempi di risposta minori), non controbilanci in modo favorevole il maggior costo d'uso del server, dovuto anch'esso all'elevata frequenza. In altre parole, mantenere il server alla frequenza corrente costa più di quanto si ricavi dalla riduzione dei tempi di risposta che sono garantiti dalla maggiore capacità erogata. Analogamente alla mossa di incremento, l'algoritmo effettua quindi una ricerca tra i server accesi, e per ciascuno di essi (o solo per un certo numero, in base a delle impostazioni del tool sviluppato) verifica se, portandone la frequenza al valore corrispondente al p-state di grado immediatamente inferiore al corrente, sia possibile migliorare il bilanciamento tra costi e prestazioni.

Per quanto riguarda invece la complessità della mossa, valgono tutte le considerazioni fatte in precedenza per la mossa di incremento di frequenza, salvo il fatto che il caso peggiore si verifica quando tutti i server sono accesi, ogni server ha allocato ogni possibile coppia classe-applicazione e nessun server sta operando già alla minima frequenza.

3.7 Gestione dell'availability

Finora nella ricerca della soluzione il sistema è stato guidato solamente dall'obiettivo di ottenere tempi di risposta bassi evitando al contempo di incorrere in costi d'uso dei server eccessivi. Tuttavia, nel mercato odierno, non è più sufficiente allocare le risorse ai servizi basandosi unicamente su criteri prestazionali e di costo, in quanto ha assunto grande importanza la possibilità di dare garanzie sulla disponibilità dei servizi offerti, ossia sulla loro *availability*. Spesso infatti, durante la stesura degli SLA, vengono concordati anche i livelli minimi di disponibilità per ciascun servizio che il cliente richiede. Come nel caso dei vincoli sui tempi di risposta, questi vincoli di availability, se violati, comportano il pagamento di penali.

È importante notare inoltre che la ricerca della soluzione illustrata in Sezione 3.4 tendeva a "dedicare" i server alle singole applicazioni comportando quindi livelli di availability inferiori rispetto a quelli ottenibili allocando la medesima applicazione su più di un server. Di conseguenza i livelli di disponibilità ottenibili con la precedente versione dell'algoritmo di resource allocation spesso non permettono di garantire i livelli minimi di availability concordati negli SLA.

Nelle sezioni successive si descriverà quindi l'integrazione della gestione dell'availability nella definizione delle politiche di allocazione delle risorse.

3.7.1 Modello del sistema con vincoli di availability

Come accennato in precedenza, i vincoli di availability rappresentano la necessità di garantire per ciascuna classe k un livello minimo di availability, definito a priori. In particolare, questo livello di disponibilità può essere raggiunto attraverso l'assegnamento opportuno dei server del Service Center, ognuno con un proprio valore di availability, all'elaborazione delle richieste appartenenti alla classe k.

Si rende quindi necessaria l'introduzione di nuovi parametri nel modello del sistema. Questi rappresenteranno il minimo livello di availability richiesto per ogni classe di servizio e l'availability propria di ciascun server. Si introducono quindi i seguenti parametri:

$$\begin{aligned} \overline{A}_k &:= \text{minimo livello di availability richiesto per la classe } k; \\ a_i &:= \text{availability del server } i. \end{aligned}$$

3.7.2 Problema di ottimizzazione con vincoli di availability

Per quanto riguarda la formulazione del problema di ottimizzazione si rende ovviamente necessaria l'introduzione dei vincoli di availability. Il problema può essere quindi formulato nuovamente come:

$$\begin{aligned} \max \sum_{k=1}^{K} \left(-m_k \cdot \sum_{i=1}^{M} \sum_{j=1}^{N_k} \frac{\lambda_{i,k,j}}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}} \right) + \\ - \sum_{i=1}^{M} \left(\sum_{h \in H_i} c_{i,h} f_{i,h} \right) - \sum_{i=1}^{M} cs_i \cdot \max\left(0, x_i - \overline{x}_i\right) + \\ - \sum_{i \in M, k \in K, j \in N_k} cm \cdot \max\left(0, z_{i,k,j} - \overline{z}_{i,k,j}\right) \end{aligned} \tag{3.20}$$

$$\sum_{i=1}^{M} \lambda_{i,k,j} = \Lambda_k; \qquad \forall k, j \tag{3.21}$$

$$\sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \leq 1; \qquad \forall i \tag{3.22}$$

$$z_{i,k,j} \leq A_{i,k,j} x_i; \qquad \forall i, k, j \tag{3.23}$$

$$\lambda_{i,k,j} \leq \Lambda_k z_{i,k,j}; \qquad \forall i, k, j \tag{3.24}$$

$$\lambda_{i,k,j} < \left(\sum_{h in H_i} C_{i,h} f_{i,h} \right) \mu_{k,j} \phi_{i,k,j}; \qquad \forall i, k, j \tag{3.25}$$

$$\sum_{h \in H_i} f_{i,h} = x_i; \qquad \forall i \tag{3.26}$$

$$\overline{A}_k \leq \prod_{j=1}^{N_k} \left(1 - \prod_{i=1}^{M} (1 - a_i)^{z_{i,k,j}} \right); \qquad \forall k \tag{3.27}$$

$$\sum_{j=1}^{N_k} z_{i,k,j} \leq 1; \qquad \forall i, k \tag{3.28}$$

$$\phi_{i,k,j} \geq 0; \qquad \forall i, k, j$$

$$\lambda_{i,k,j} \geq 0; \qquad \forall i, k, j$$

$$x_i,\ z_{i,k,j},\ f_{i,h} \in [0, 1]; \qquad \forall i, k, j;\ \forall h \in H_i$$

I nuovi vincoli introdotti sono definiti dalle espressioni (3.27) e (3.28), mentre per i restanti vincoli valgono le medesime considerazione fatte nella prima parte del corrente capitolo.

Il vincolo (3.27) impone che il livello di availability ottenuto a seguito dell'assegnamento dei server a supporto della classe k non possa essere inferiore al valore dell'availability concordato per la classe in questione. L'availability raggiunta viene calcolata considerando come funzionanti in parallelo i server assegnati ad ogni tier. Si ha quindi che per il generico tier j della classe k l'availability (indicata col termine $\tilde{A}_{k,j}$) può essere calcolata considerando i server in parallelo, tramite la formula:

$$\tilde{A}_{k,j} = 1 - \prod_{i=1}^{M} (1 - a_i)^{z_{i,k,j}}$$

Una volta calcolata l'availability per lo specifico tier $\tilde{A}_{k,j}$, considerando in serie i vari tier della medesima classe è possibile ottenere il livello di availability

raggiunto per la classe k $\tilde{A}_k$ dato l'assegnamento corrente dei server:

$$\tilde{A}_k = \prod_{j=1}^{N_k} \tilde{A}_{k,j} = \prod_{j=1}^{N_k} \left(1 - \prod_{i=1}^{M} (1 - a_i)^{z_{i,k,j}} \right)$$

Per rispettare il vincolo, quindi per garantire un sufficiente livello di availability, questo valore deve essere maggiore o uguale al valore di availability concordato con il cliente per la classe ($\tilde{A}_k \geq \overline{A}_k$).

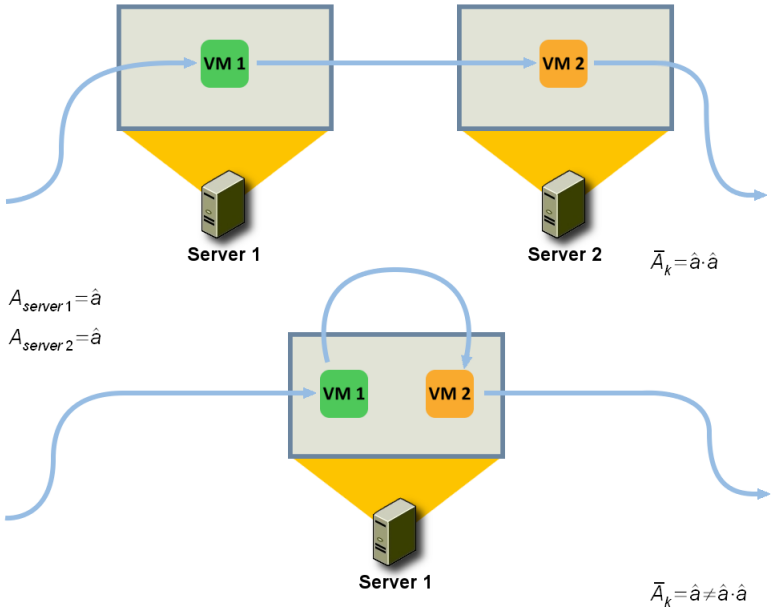

Figura 3.2: In alto, le due VM vengono disposte su due server differenti in modo da non violare il vincolo di l'availability del tier. In basso, il vincolo viene trascurato. Entrambe le VM associate ai tier applicativi della classe sono allocate su un unico server.

Il secondo vincolo introdotto (3.28) (che sostituisce il precedente vincolo (3.6)) si occupa di impedire che su uno stesso server vengano allocati tier applicativi differenti, ma appartenenti alla medesima classe. In altre parole, se un tier $\tilde{j}$ della classe k è stato assegnato ad un server i allora nessun altro tier di k dovrà trovarvi allocazione. Se non si imponesse questo vincolo verrebbe meno l'indipendenza tra le availability dei tier, indipendenza che è richiesta affinché il vincolo (3.27) non perda significato. A dimostrazione di questo fatto è sufficiente considerare il seguente esempio. Sia k una classe con due tier j_1 e j_2 e si supponga che per questa classe si richieda un livello di availability pari a $\overline{A}_k$.

Si supponga inoltre che siano disponibili per l'assegnamento due server i_1 e i_2, ognuno con availability $\hat{a}$, e che assegnando ciascun server ad un solo tier sia possibile soddisfare il vincolo di availability per la classe. In altre parole, assegnando il tier j_1 al server i_1 ed il tier j_2 a i_2, ossia considerandoli operanti in serie dal punto di vista della classe k, si ha che $\overline{A}_k \leq \hat{a} \cdot \hat{a}$ (FIGURA 3.2).
Questo risultato rispetta sia il vincolo (3.27), sia il vincolo (3.28), in quanto i due tier della classe sono allocati su server diversi. Tuttavia, se si ignorasse il vincolo (3.28) ed almeno uno dei due server, si supponga i_1, fosse in grado di supportare contemporaneamente le richieste per entrambi tier, allora in questo caso il sistema potrebbe allocare entrambi i tier sul solo i_1 in modo da ridurre i costi. Per come viene valutata l'availability della classe da parte del vincolo (3.27), il valore raggiunto sembrerebbe nuovamente pari a $\hat{a} \cdot \hat{a}$ in quanto il sistema "vedrebbe" due server in serie. In realtà l'availability raggiunta dalla classe è invece pari ad $\hat{a}$, in quanto entrambi i tier sono allocati sul solo i_1 e per questo le probabilità di guasto per i due tier non sono indipendenti. Come si può capire per preservare la validità del vincolo (3.27), è necessario introdurre anche il vincolo (3.28), in modo da garantire l'indipendenza tra le availability dei tier per ogni classe.

3.7.3 Assegnamento iniziale con vincoli di availability

L'algoritmo è molto simile a quello descritto nel PARAGRAFO 3.4.1 e ne costituisce una estensione in modo da considerare anche del vincolo (3.28), ossia evitare l'assegnamento di un server a tier differenti della medesima classe. Infatti si suppone che la soluzione del periodo di controllo precedente, da cui si parte per derivare la nuova, rispetti già tutti i vincoli. Non è quindi necessario tener conto dei vincoli di availability (3.27) poiché l'algoritmo non comporta in alcun modo lo spegnimento di server o la deallocazione di classi[5], ma al più alloca nuove coppie classe-tier applicativo su server già accesi oppure, quando necessario, accende nuovi server. Per cui, partendo da una soluzione che già rispetta i vincoli di availability, l'unico risultato di queste due azioni è un incremento del valore di availability precedentemente ottenuto. L'ipotesi appena esposta prevede che la soluzione precedente debba considerare che per ogni tier k, j associato ad una classe k per cui si richiede un'availability pari a $\overline{A}_k$ debba possedere un valore di availability almeno pari a $\sqrt[N_k]{\overline{A}_k}$. Scelto un insieme di

[5]In effetti la deallocazione di classi è contemplata, può infatti essere effettuata nel caso in cui il nuovo valore di carico per la classe sia nullo. L'algoritmo provvede quindi alla deallocazione della classe da tutti i server rendendo nulla l'availability. Tuttavia ciò non comporta problemi poiché in caso di carico nullo non si rende necessario garantire il livello di availability concordato.

server $\mathcal{S}_{k,j}$ per elaborate il carico del tier, deve essere verificato che:

$$\begin{aligned} \sqrt[N_k]{\overline{A_k}} &\leq 1 - \prod_{i \in \mathcal{S}_{k,j}} (1 - a_i) \\ \Lambda_k = \Gamma_{k,j} &\leq \overline{Ut} \sum_{i \in \mathcal{S}_{k,j}} \left(C_{i,\overline{h}} f_{i,\overline{h}} \right) \mu_{k,j} \left(1 - \sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \right) \end{aligned}$$

ossia, rispettivamente, l'insieme dei server in $\mathcal{S}_{k,j}$, operanti in parallelo, permette almeno di eguagliare l'availability richiesta per il tier $\sqrt[N_k]{\overline{A_k}}$ ed i server hanno sufficiente capacità per poter gestire l'intero carico Λ_k senza violare i vincoli di utilizzo massimo $\overline{Ut}$.

Il funzionamento dell'algoritmo prevede la determinazione dell'eventuale carico in eccedenza e che non può essere gestito con il corrente assegnamento, quindi tenta in un primo momento di assegnarlo effettuando nuove allocazioni di coppie classe-tier su server già accesi compatibilmente con i vincoli (3.23) e (3.28). Nel caso non si riesca ad assegnare completamente l'eccedenza, l'algoritmo prova ad accendere nuovi server (sempre rispettando i vincoli) per procedere a nuovi assegnamenti fino a che o l'intero carico non è stato assegnato (soluzione ammissibile) oppure è rimasto del carico residuo che non potrà essere assegnato senza comportare la violazione di vincoli (soluzione non ammissibile). Nel secondo caso non è comunque detto che una soluzione non esista, è infatti possibile che i vincoli impliciti introdotti dalla necessità di appoggiarsi ad una soluzione precedente (assegnamento precedente, insieme dei server già accesi, ecc...) abbiano limitato eccessivamente la ricerca della nuova soluzione, per questo motivo è comunque possibile ricercare una soluzione completamente nuova e che permetta di gestire l'intero carico.

3.7.4 Load balancing e capacity allocation con vincoli di availability

Per quanto riguarda la politica di capacity allocation ed il load balancing delle richieste, si rende necessario operare delle modifiche a questa procedura poiché dovranno essere considerati anche i vincoli di availability (3.27), ma non i vincoli (3.28). Infatti, ipotizzando di applicare la procedura ad un assegnamento valido, non è possibile che i vincoli (3.28) vengano violati, poiché la *fixed point iteration* non contempla l'allocazione di nuove coppie classe-tier sui server, ma al più è possibile che effettui delle deallocazioni. È proprio quest'ultima possibilità che rende necessario modificare la procedura, in quanto la deallocazione di una certa coppia k-j da un server i potrebbe portare ad una violazione di un vincolo.

Come descritto nella Sezione 3.5.1, può accadere un'applicazione venga deallocata da un server ponendo $\lambda_i = 0$. Il problema sorge a questo punto

poiché procedendo alla deallocazione è possibile ottenere una soluzione migliore, ma anche andare a violare il vincolo di availability per la classe k. Infatti ciò significherebbe rimuovere dal tier un server e, come conseguenza, ridurre il livello di availability. L'algoritmo standard procederebbe alla selezione del server con il minore valore U associato e deallocherebbe da questo il carico corrispondente. La *lambda-iteration* modificata calcola invece il valore di availability per la classe che si otterrebbe a seguito della rimozione del server e verifica se il nuovo valore comporti la violazione del vincolo. Nel caso non vi sia violazione, l'algoritmo procede alla deallocazione, in caso contrario passa invece al server con minor U successivo ed effettua nuovamente il controllo sulla possibile violazione. Questi passi sono ripetuti fino a trovare un server adatto oppure fino all'esaurimento dei server per cui è possibile procedere alla deallocazione. In questo secondo caso, in cui ogni possibile deallocazione comporterebbe la violazione del vincolo, la *lambda-iteration* procede ad elaborare la successiva coppia classe-tier applicativo senza modificare la soluzione.

3.7.5 Ricerca locale con vincoli di availability

Verranno ora presentate le modifiche effettuate alle mosse della ricerca locale descritte nella Sezione 3.6 per evitare che queste portino a violazioni dei vincoli di availability. Ovviamente le mosse di incremento e di decremento di frequenza non sono state modificate, dato che queste non comportano trasferimenti di carico né modifiche all'insieme dei server accesi.

3.7.5.1 Accensione di un server

La mossa di accensione di un server non è stata ulteriormente modificata a seguito dell'introduzione dei vincoli di availability. Infatti questa mossa verifica se accendendo un nuovo server e distribuendo il carico tra questo ed un server ad alto utilizzo sia possibile migliorare la soluzione. Dato che la soluzione iniziale cui si tenta di applicare la mossa è già essere ammissibile, l'accensione di un server $\tilde{i}$ ed il trasferimento su questo di parte del carico assegnato ad un altro server $\hat{i}$ ha come conseguenza l'aumento dell'availability per ognuna delle coppie classe-tier applicativo allocate su $\hat{i}$.

3.7.5.2 Riallocazione di applicazioni con vincoli di availability

La mossa di riallocazione richiede una modifica atta ad evitare la violazione del vincolo (3.28), dato che la soluzione cui viene applicata la mossa è già ammissibile e dato che l'allocazione di un'applicazione su un nuovo server comporta l'aumento dell'availability per la classe corrispondente. Una volta identificata la coppia classe-tier applicativo $\tilde{k}$-$\tilde{j}$ la cui riallocazione potrebbe migliorare la soluzione, la mossa cerca il miglior server su cui effettuare tale allocazione tra

quelli correntemente accesi. Tuttavia, una volta identificato un server candidato, va verificato che su di esso non siano già allocati tier applicativi differenti associati alla medesima classe k, in quanto ciò comporterebbe una violazione di (3.28). La mossa procede quindi a questa verifica e, nel caso si evidenzi la possibilità di una violazione, la mossa scarta il corrente server candidato e passa a verificare quello successivo.

3.7.5.3 Spegnimento di un server con vincoli di availability

Dato che questa mossa spegne un server, spostando il carico che esso gestiva su altri server senza però allocare nuove coppie classe-tier applicativo, essa provocherà in ogni caso una riduzione dell'availability garantita per ciascuna classe k che era presente sul server spento. Per questo motivo, prima di procedere allo spegnimento, si verifica se questo comporti la violazione di un qualsiasi vincolo (3.27) per la generica classe k. In caso di violazione, il server candidato non viene spento e la mossa procede alla ricerca di un ulteriore server da spegnere.

3.7.6 Algoritmo di ricerca locale con vincoli di availability

```
1  Inizializza la lista delle violazioni 𝓛;
2  Inizializza BestSol; //Migliore soluzione ammissibile nota
3  Inizializza CurrentSol;
4  go = true;
5  while go == true do
6  |  Applica la ricerca standard (Algoritmo 5);
7  |  Applica la ricerca con violazioni di availability permesse (Algoritmo 6);
8  |  Applica la risoluzione delle violazioni di availability (Algoritmo 7);
9  |  if la risoluzione delle violazioni di availability non è riuscita then
10 |  |  go = false;
11 |  end
12 end
13 Restituisci BestSol;
```

Algoritmo 4: Algoritmo di ricerca locale con vincoli di availability.

Le modifiche apportate alle mosse della ricerca locale permettono di garantire l'availability minima richiesta per ogni classe nel caso siano applicate ad una soluzione iniziale ammissibile. Tuttavia utilizzare l'Algoritmo 3 applicando le mosse modificate è in qualche modo eccessivamente restrittivo in quanto obbligherebbe il sistema a mantenere in ogni istante una soluzione ammissibile per quanto riguarda i vincoli di availability, senza permettergli di esplorare soluzioni alternative che potrebbero portare a soluzioni potenzialmente migliori.
Per questo motivo è stata implementata una nuova versione dell'algoritmo di ricerca locale che permette di violare temporaneamente i vincoli di availability così da ridurre le restrizioni alla ricerca (Algoritmo 4). Il nuovo algoritmo si articola in tre fasi:

```
1  retry = false;
2  while numero massimo di mosse non raggiunto do
3  |  Memorizza il valore corrente della soluzione corrente CurrentSol;
4  |  Tenta di applicare le mosse a CurrentSol;
5  |  if ∄mossa applicabile ∧ retry == true then
6  |  |  //Nessuna mossa applicabile e fixed point //iteration già
   |  |  applicata
7  |  |  Restituisci CurrentSol;
8  |  end
9  |  if ∄mossa applicabile ∧ retry == false then
10 |  |  //Nessuna mossa applicabile e fixed point //iteration non
   |  |  ancora applicata
11 |  |  Applica la fixed point iteration a CurrentSol;
12 |  |  retry = true;
13 |  end
14 |  if ∄mossa applicabile senza violare l'availability then
15 |  |  //Salva la migliore soluzione ammissibile nota
16 |  |  BestSol = CurrentSol;
17 |  |  Applica la mossa con violazioni migliore a CurrentSol;
18 |  |  Aggiungi ad ℒ le violazioni rilevate;
19 |  |  break;
20 |  else
21 |  |  Applica la mossa senza violazioni migliore a CurrentSol;
22 |  |  retry = false;
23 |  end
24 end
```

Algoritmo 5: Algoritmo di ricerca standard con vincoli di availability.

- *Fase 1*: (Algoritmo 5) in questa fase il sistema tenta di applicare tutte le mosse miglioranti in modo da evitare di incorrere in violazioni dei vincoli di availability, ossia utilizza le mosse modificate descritte in precedenza (Passi 21-22). Nel caso non venga trovata alcuna mossa migliorante, l'algoritmo esegue una *fixed point iteration*, indicando di averla eseguita ($retry = true$), e ripete la ricerca (Passi da 9 a 12). Se non vengono nuovamente trovate mosse miglioranti, dato che si è già applicata la *fixed point iteration* (ossia $retry = true$), la ricerca locale viene interrotta e l'algoritmo restituisce la soluzione corrente (Passi da 5 a 7).
  Quando invece l'algoritmo trova una o più mosse miglioranti, ma tali per cui l'applicazione di ognuna di queste comporti violazioni dei vincoli di availability, allora l'algoritmo salva la soluzione corrente (che è la migliore soluzione ammissibile nota), seleziona la mossa migliore, inserisce in una lista le violazioni di availability che la mossa provocherà, quindi la applica

```
1  retry = false;
2  while numero massimo di mosse con violazione non raggiunto do
3  |  Memorizza il valore corrente della soluzione corrente CurrentSol;
4  |  Tenta di applicare le mosse a CurrentSol;
5  |  if ∄mossa applicabile ∧ retry == true then
6  |  |  //Nessuna mossa applicabile e fixed point //iteration già
   |  |  applicata
7  |  |  break;
8  |  end
9  |  if ∄mossa applicabile ∧ retry == false then
10 |  |  //Nessuna mossa applicabile e fixed point //iteration non
   |  |  ancora applicata
11 |  |  Applica la fixed point iteration a CurrentSol;
12 |  |  retry = true;
13 |  end
14 |  if ∃mossa applicabile then
15 |  |  Applica la mossa migliore (anche con violazioni) a CurrentSol;
16 |  |  Aggiungi le eventuali nuove violazioni rilevate a ℒ;
17 |  |  retry = false;
18 |  end
19 end
```

Algoritmo 6: Algoritmo di ricerca con violazioni di availability permesse.

e passa alla fase successiva (Passi da 14 a 19).

- *Fase 2*: (Algoritmo 6) l'algoritmo giunge in questa fase dopo aver applicato una mossa migliorante, ma che ha provocato almeno la violazione di un vincolo di availability. Di conseguenza, la soluzione corrente non è ammissibile. Il sistema procede quindi all'applicazione di tutte le mosse miglioranti trovate, permettendo in questo caso a queste di introdurre nuove violazioni di availability. Ogni volta in cui viene selezionata per l'applicazione una mossa che comporta nuove violazioni, queste vengono aggiunte alla lista delle violazioni in modo da poterle risolvere in seguito (Passi da 14 a 17).
  Questa fase termina quando, anche a seguito dell'applicazione della *fixed point iteration*, non vengono trovate nuove mosse miglioranti (Passi da 5 a 7), oppure quando è stato applicato il numero massimo di mosse con violazione[6] permesso. Il sistema passa quindi alla fase successiva;

- *Fase 3*: (Algoritmo 7) giunti in questa fase si hanno una soluzione non ammissibile ed una lista di violazioni. Il sistema tenterà quindi di risolvere le violazioni presenti nella lista andando ad effettuare delle mosse

[6] Numero specificato come parametro di configurazione del Resource Allocator.

```
1  Ordina 𝓛 per valori di violazione dell'availability decrescenti;
2  for l = 1 to lenght(𝓛) do
3  |  Tenta di risolvere la violazione l usando i server già accesi;
4  |  if Violazione risolta then
5  |  |  Rimuovi l da 𝓛;
6  |  else
7  |  |  Tenta di risolvere la violazione l usando anche i server spenti;
8  |  |  if Violazione risolta then
9  |  |  |  Rimuovi l da 𝓛;
10 |  |  else
11 |  |  |  Restituisci la migliore soluzione ammissibile conosciuta BestSol;
12 |  |  end
13 |  end
14 end
15 Applica la fixed point iteration alla soluzione corrente CurrentSol;
16 retry = false;
17 while val(BestSol) > val(CurrentSol) do
18 |  Memorizza il valore corrente di CurrentSol;
19 |  Tenta di applicare le mosse a CurrentSol;
20 |  if ∄mossa applicabile senza violazioni ∧ retry == true then
21 |  |  Restituisci la migliore soluzione ammissibile conosciuta BestSol;
22 |  end
23 |  if ∄mossa applicabile senza violazioni ∧ retry == false then
24 |  |  Applica la fixed point iteration a CurrentSol;
25 |  |  retry = true;
26 |  end
27 |  if ∃mossa applicabile senza violazioni then
28 |  |  Applica la mossa migliore a CurrentSol;
29 |  |  retry = false;
30 |  end
31 |  if val(CurrentSol) > val(BestSol) then
32 |  |  BestSol = CurrentSol;
33 |  |  break;
34 |  end
35 end
```

Algoritmo 7: Algoritmo di risoluzione delle violazioni di availability.

specifiche. In particolare, per ognuna delle violazioni nella lista, acquisirà la coppia classe-tier k-j associatavi, quindi tenterà in un primo momento di recuperare l'availability richiesta riallocando la coppia k-j su insiemi di server già accesi e successivamente, nel caso non sia stato possibile risolvere la violazione usando solo server già accesi, andando a considerare combinazioni sia di server accesi che di server spenti. Una volta identificato un insieme di server $\mathcal{N}$ grazie al quale è possibile risolvere la violazione riallocandovi parte del carico k-j dall'insieme $\mathcal{I}$ dei server già associati al tier, l'algoritmo procede a ripartire il carico tra gli insiemi $\mathcal{I}$ e $\mathcal{N}$ (accendendo se necessario i server spenti presenti in $\mathcal{N}$) in modo da equalizzare i tempi di risposta (Passi da 2 a 13). Dopo aver rimosso tutte le violazioni la soluzione avrà un valore inferiore della funzione obiettivo, per cui, nel

tentativo di migliorarla, si eseguiranno una *fixed point iteration* (Passo 15) ed una nuova ricerca locale in cui verranno nuovamente considerate le sole mosse miglioranti e che non comportino violazioni (Passi da 17 a 35). Questa fase terminerà quando la nuova soluzione ottenuta sarà migliore della precedente miglior soluzione ammissibile che era stata memorizzata al termine della prima fase (Passi da 31 a 34). Quindi l'algoritmo ritornerà alla prima fase per tentare un ulteriore miglioramento.
Nel caso invece il sistema non riesca a risolvere tutte le violazioni nella lista (Passi da 10 a 12) oppure nel caso in cui, anche a seguito di una *fixed point iteration*, la ricerca locale termini senza riuscire ad ottenere una soluzione migliore (Passi da 20 a 22), allora l'algoritmo terminerà e restituirà la soluzione memorizzata al termine della prima fase in quanto questa è la migliore soluzione ammissibile ottenuta.

Questa nuova estensione della ricerca locale implementata in [8] costituisce una tecnica di diversificazione e in [7] è stato dimostrato sperimentalmente che è possibile sfuggire a ottimi locali, tuttavia aumentano la complessità e i tempi di calcolo che non permettono di gestire Service Center di ampie dimensioni, sia in termini del numero di applicazioni da elaborare che del numero di server da gestire.

Obiettivo della tesi è sviluppare tecniche gerarchiche efficienti che verranno mostrate nel capitolo successivo.

Capitolo 4

Tecniche gerarchiche per l'allocazione delle risorse in Service Center di grandi dimensioni

In questo capitolo verranno descritte le estensioni che sono state apportate al modello di ottimizzazione e le conseguenti modifiche degli algoritmi descritti nel precedente capitolo. In particolare verrà introdotto un ulteriore vincolo atto a considerare l'occupazione di memoria RAM delle macchine virtuali che vengono allocate sui server. Nella Sezione *4.1 verranno descritte le modifiche per l'implementazione di questo vincolo aggiuntivo. Verrà inoltre introdotto un nuovo modello di consumo energetico per un calcolo più realistico dei costi dei server e verranno presentate le modifiche effettuate agli algoritmi precedentemente progettati.*

I Service Center odierni sono caratterizzate da grandi dimensioni: non è infrequente infatti incontrare sistemi con migliaia di classi applicative e server fisici. Il sistema di ottimizzazione (centralizzato) fin qui proposto non riesce a gestire istanze di questo tipo. Infatti, per grosse istanze gli algoritmi proposti sono caratterizzati da lunghi tempi di elaborazione. Per porre rimedio a questa limitazione è stato sviluppato un sistema di ottimizzazione gerarchico, basato sulla suddivisione dell'intero Service Center in partizioni applicative. Nella Sezione *4.2 verranno mostrate le tecniche utilizzate per implementare il sistema di ottimizzazione gerarchica. In particolare, verranno descritto un modello generale di ottimizzazione gerarchica e le scelte implementative utilizzate per adattare tele modello per risolvere il problema di resource allocation.*

4.1 Estensione del modello e tecniche di resource allocation

Nel capitolo precedente è stata descritta la metodologia di Resource Allocation basata su utility function lineari che è stata presentata in [14] [8] e si è discussa la possibilità di introdurre vincoli di availability. Come descritto in [7], tali vincoli sono l'anello debole dei sistemi cloud.

Nei sistemi virtualizzati è necessario considerare un ulteriore problema prestazionale: la quantità di RAM a disposizione di una VM. In un sistema reale infatti, per ogni classe-tier applicativo è necessario individuare una quantità di memoria minima per poter eseguire correttamente il servizio applicativo corrispondente. Nel sistema descritto nel capitolo precedente questo aspetto viene trascurato, permettendo di poter allocare infinite VM su di un server con l'unico vincolo che questo ne riesca ad elaborare il carico. Si configura in tal modo uno scenario che si discosta da un sistema reale. Ecco la concreta necessità per cui si è introdotto un ulteriore vincolo che tiene in considerazione la memoria correntemente occupata.

Oltre al vincolo sulla memoria, è stata introdotta nel sistema un'ulteriore estensione. Il modello illustrato nel precedente capitolo infatti, utilizza per ciascun server un sistema di calcolo del consumo energetico semplificato. Si suppone che il consumo di un elaboratore sia legato solamente alla frequenza operativa utilizzata. Viceversa, nei sistemi reali il consumo energetico dipende linearmente anche dall'utilizzo della CPU fisica. Questa relazione viene quindi descritta mediante una famiglia di rette (una per ciascuna frequenza), in cui ciascuna retta fornisce il consumo di ciascun server in relazione con il suo utilizzo.

Nella Sezione 4.1.1 verrà illustrata l'estensione del modello contenente il vincolo per la gestione della memoria RAM, mentre nella Sezione 4.1.2 verrà preso in considerazione il nuovo modello di consumo energetico.

4.1.1 Estensione con vincolo di RAM

Verranno ora illustrate le modifiche apportate al modello di ottimizzazione presentato nella Sezione 3.7.2 a seguito dell'introduzione di un vincolo per il controllo e la gestione delle RAM. Per facilitare il lettore viene di seguito riportata

la notazione che verrà utilizzata nel seguente capitolo:

K := numero di classi di richieste;

N_k := numero complessivo di tier applicativi coinvolti nell'esecuzione delle richieste di classe k;

M := numero complessivo di server nel Service Center;

H_i := insieme delle frequenze operative per il server i;

$C_{i,h}$:= capacità del server i operante alla frequenza h ($h \in H_i$);

$c_{i,h}$:= costo per unità di tempo associato al server i se acceso ed operante alla frequenza h ($h \in H_i$);

$\mu_{k,j}$:= massima frequenza di elaborazione delle richieste per il tier applicativo j della classe k sostenibile da un server di capacità 1;

$R_{i,k,j}$:= tempo medio di risposta del server i a richieste per il tier applicativo j della classe k;

cs_i := overhead causato dall'accensione del server i;

cm := overhead causato dal trasferimento di una virtual machine su un server differente.

$\overline{RAM}_i$:= memoria RAM fisicamente installata sul server

$RAM_{k,j}$:= RAM necessaria all'esecuzione del tier j della classe k.

Nel modello devono essere tenuti in considerazione due nuovi parametri atti ad esprimere la RAM fisica disponibile e la RAM utilizzata da ciascun applicazione. ($\overline{RAM}_i$ e $RAM_{k,j}$). Per quanto riguarda la variabili di decisione è sufficiente considerare le stesse variabili presentate in Sezione 3.2:

x_i := variabile binaria che vale 1 se il server i è acceso, 0 altrimenti;

$\lambda_{i,k,j}$:= frequenza di arrivo sul server i delle richieste per il tier applicativo j della classe k;

$\phi_{i,k,j}$:= percentuale di allocazione del processore del server i per il tier applicativo j della classe k;

$z_{i,k,j}$:= variabile binaria che vale 1 solo se il tier applicativo j per la classe k è allocato sul server i;

$f_{i,h}$:= variabile binaria che vale 1 solo se il server i sta operando alla frequenza h, con $h \in H_i$.

Il modello di ottimizzazione che si ottiene è il seguente:

$$\max \sum_{k=1}^{K} \left(-m_k \cdot \sum_{i=1}^{M} \sum_{j=1}^{N_k} \frac{\lambda_{i,k,j}}{\left(\sum_{h \in H_i} C_{i,h} f_{i,h}\right) \mu_{k,j} \phi_{i,k,j} - \lambda_{i,k,j}} \right) +$$
$$- \sum_{i=1}^{M} \left(\sum_{h \in H_i} c_{i,h} f_{i,h} \right) - \sum_{i=1}^{M} cs_i \cdot \max\left(0, x_i - \overline{x}_i\right) +$$
$$- \sum_{i \in M, k \in K, j \in N_k} cm \cdot \max\left(0, z_{i,k,j} - \overline{z}_{i,k,j}\right) \tag{4.1}$$

$$\sum_{i=1}^{M} \lambda_{i,k,j} = \Lambda_k; \qquad \forall k, j \tag{4.2}$$
$$\sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \leq 1; \qquad \forall i \tag{4.3}$$
$$z_{i,k,j} \leq A_{i,k,j} x_i; \qquad \forall i, k, j \tag{4.4}$$
$$\lambda_{i,k,j} \leq \Lambda_k z_{i,k,j}; \qquad \forall i, k, j \tag{4.5}$$
$$\lambda_{i,k,j} < \left(\sum_{hinH_i} C_{i,h} f_{i,h} \right) \mu_{k,j} \phi_{i,k,j}; \qquad \forall i, k, j \tag{4.6}$$
$$\sum_{h \in H_i} f_{i,h} = x_i; \qquad \forall i \tag{4.7}$$
$$\overline{A}_k \leq \prod_{j=1}^{N_k} \left(1 - \prod_{i=1}^{M} (1 - a_i)^{z_{i,k,j}} \right); \qquad \forall k \tag{4.8}$$
$$\sum_{j=1}^{N_k} z_{i,k,j} \leq 1; \qquad \forall i, k \tag{4.9}$$
$$\sum_{j \in N_k} RAM_{k,j} z_{i,k,j} \leq \overline{RAM}_i \qquad \forall i \in M, k \in K \tag{4.10}$$
$$\phi_{i,k,j} \geq 0; \qquad \forall i, k, j$$
$$\lambda_{i,k,j} \geq 0; \qquad \forall i, k, j$$
$$x_i,\ z_{i,k,j},\ f_{i,h} \in [0,1]; \qquad \forall i, k, j;\ \forall h \in H_i$$

Come si può notare, rispetto al modello precedentemente illustrato nella Sezione 3.7.2, compare un nuovo vincolo (4.10) che garantisce che la memoria fisica installata sul server i-esimo sia sufficientemente ampia da supportare la RAM richiesta dalle VM che vi sono allocate. In altre parole, la somma dei requisiti di RAM di tutti i classe-tier applicativi in esecuzione sull'i-esimo server,

deve essere inferiore alla quantità di memoria fisicamente installata. Questo vincolo quindi tiene in considerazione la capacità finita di memoria presente su un server avvicinando ulteriormente il modello di ottimizzazione ai sistemi reali.

Di seguito vengono introdotte le estensione agli algoritmi del capitolo precedente necessarie per supportare questo nuovo vincolo.

4.1.1.1 Ricerca di una soluzione iniziale a partire da una soluzione precedente

L'algoritmo di ricerca di una soluzione a partire da una soluzione precedente (si faccia riferimento alle Sezioni 3.4.1 e 3.7.3) in caso di variazioni del carico delle richieste per le classi di servizio, cerca di applicare il minor numero possibile di modifiche alla soluzione per quanto riguarda l'accensione di nuovi server e la riallocazione di applicazioni. Si suppone che la soluzione precedente rispetti già i vincoli di RAM e si mette in evidenza il fatto che questo algoritmo non può in alcun modo deallocare applicazioni o spegnere server che erano attivi nella soluzione data in input. Rispetto all'algoritmo precedentemente illustrato nelle Sezioni 3.4.1 e 3.7.3, viene introdotto un passo di inizializzazione che calcola i parametri che descrivono per ogni server la RAM ancora disponibile ($\overline{ResRAM_i}$), come:

$$\overline{ResRAM_i} = \overline{RAM_i} - \sum_{k,j \in \mathcal{K}} RAM_{k,j} \ , \forall_{i \in I}$$

Effettuata questa operazione l'algoritmo, come primo passo calcola i nuovi parametri di load balancing e computa i residui di carico da assegnare. Nella fase successiva tenta di assegnare questi carichi aggiuntivi mediante l'incremento delle frequenze dei server aggiuntivi. Se questo non fosse sufficiente l'algoritmo procede, dopo aver ordinato le classi in maniera decrescente per numero di vincoli $\mathcal{B}$ ed in maniera crescente per valori di $\Gamma_{k,j}$ ed i server per $\frac{c_i}{C_i}$, ad instanziare il carico rimanente come nuove applicazioni sui server già attivi. In questa fase l'algoritmo dovrà garantire che, dato un server $\overline{i}$ e un classe-tier applicativo $\overline{k}, \overline{j}$:

$$\overline{ResRAM_{\overline{i}}} \geq RAM_{\overline{k},\overline{j}}$$

In tal modo, non sarà concesso installare una VM su di un server che non può garantire sufficiente memoria per la sua corretta esecuzione. Infine, se anche questo tentativo non permettere di coprire tutto il carico della nuova soluzione, si procede ad accendere dei nuovi server. Quando si tenta di assegnare un classe-tier applicativo ad un server da accendere viene verificato se il server in questione possiede abbastanza RAM per rendere possibile l'esecuzione della VM da allocare. In caso affermativo l'applicazione viene allocata sul server consi-

derato, aggiornando la RAM residua della macchina in base a quanto richiesto dalla VM:

$$\overline{ResRAM_{\bar{i}}} = \overline{ResRAM_{\bar{i}}} - RAM_{\bar{k},\bar{j}}$$

con $\bar{i}$ che rappresenta l'indice del server da accendere per l'allocazione del tier applicativo $\bar{k}, \bar{j}$. In caso contrario si prenderà in considerazione il prossimo server nell'ordinamento. Se l'applicazione ha la necessità di essere allocata su più server la memoria occupata dall'applicazione verrà decrementata da ciascun elaboratore a cui viene assegnata.

4.1.1.2 Fix point iteration basata sulle condizione KKT

L'algoritmo che si occupa di gestire i problemi di load balancing e capacity allocation descritto nelle SEZIONI 3.5.1 e 3.7.4 è stato modificato in modo da considerare il vincolo sulla RAM. Queste procedure, infatti, permettono di deallocare delle applicazioni dai server.

Durante la procedura di fix point iteration può accadere che la lambda iteration durante la sua esecuzione possa porre alcune variabili λ_i a zero. Impostare λ_i a zero vuol dire deallocare del traffico per quel classe-tier applicativo. Bisogna quindi considerare quest'allocazione nel computo della RAM residua sul server. Dato un server $\bar{i}$ e un classe-tier applicativo $\bar{k}, \bar{j}$, si andrà a re-incrementare la sua RAM residua:

$$\overline{ResRAM_i} = \overline{RAM_i} + \sum_{k,j \in \mathcal{K}} RAM_{k,j} \ , \forall_{i \in I}$$

4.1.1.3 Ricerca locale

In questa sezione verranno illustrate le modifiche rispetto alla local search descritta in 3.6 e in 3.7.5 al fine di impedire la violazione del vincolo sulla RAM.

Accensione di un server

Questa mossa cerca un server sovraccarico nel sistema, accende un server compatibile e permette la ripartizione del carico tra i due. In questa mossa è necessario accendere un server che abbia a disposizione un quantitativo di RAM sufficiente ad eseguire tutte le applicazioni attive sul primo server. Dato un primo server sovraccarico $\widetilde{i}$, il server $\hat{i}$ da accendere deve soddisfare il requisito:

$$\overline{RAM_{\hat{i}}} \geq \sum_{(k,j) \in \mathcal{K}(\widetilde{i})} RAM_{k,j}$$

dove $\mathcal{K}(\widetilde{i})$ indica le classi e i tier presenti sul server $\widetilde{i}$.

Ovviamente, se viene effettuato l'aggiornamento delle classi al server $\widetilde{i}$, dovrà essere effettuato il decremento della RAM con il quantitativo di memoria necessario all'esecuzione delle varie applicazioni. Se durante la ricerca del nuovo server da accendere vengono considerate le tipologie in cui vengono suddivisi gli elaboratori, sarà preferibile considerare prima i server appartenenti alla stessa tipologia che, avendo la stessa struttura fisica, avranno certamente RAM a sufficienza per la corretta esecuzione delle VM.

Spegnimento di un server

Questa mossa spegne un server $\hat{i}$ nel sistema spostandone il carico su altri server correntemente accesi. In questo algoritmo è stata implementata una semplice modifica che, prima di spegnere il server candidato $\hat{i}$, si preoccupa di rimuovere le varie applicazioni riassegnando al server la RAM precedentemente occupata da ciascuna VM. Quando si andranno a spostare i tier applicativi di $\hat{i}$ su altri server non sarà necessario nessun decremento di memoria. I server destinazione infatti eseguono già Virtual Machine che elaborano ciascun classe-tier applicativo di cui si sta trasferendo il carico.

Riallocazione di applicazioni

Questa mossa consente la migrazione di applicazioni da un server ad un altro. Come nei casi precedenti, quando si cercherà un server candidato per accogliere il nuovo tier applicativo, bisognerà verificare che questo non comporti una violazione del vincolo sulla RAM. Non devono quindi essere considerati nel processo di ricerca tutti quei server privi di un quantitativo sufficientemente ampio di memoria per eseguire la nuova VM. Inoltre, bisognerà gestire l'incremento della memoria residua sul server che cede il tier applicativo ed il decremento su quello che lo riceve, in maniera analoga a quanto visto in precedenza nelle mosse di accensione e spegnimento di un server.

Mosse di frequency scaling

Questa tipologia di mosse comprende sia l'incremento, sia il decremento della frequenza operativa dei server con l'obiettivo di aumentare o a diminuire la capacità di un server. Dato che queste operazioni non hanno alcun impatto sui vincoli di memoria non è stato necessario modificare le mosse descritte nella Sezione 3.6.4.

4.1.2 Estensione del modello di power consumption

Verranno ora illustrate le modifiche apportate al modello di ottimizzazione presentato nella Sezione 3.7.2 a seguito dell'introduzione di un nuovo modello che

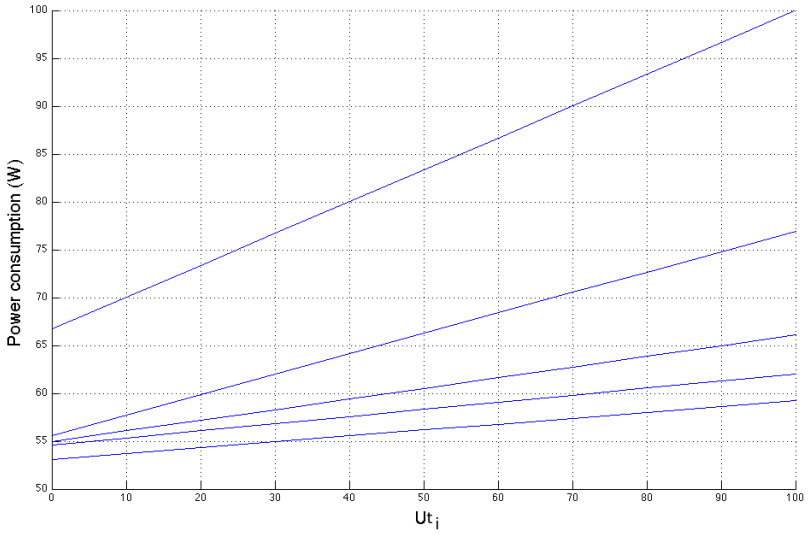

Figura 4.1: Modello Energetico

regola il consumo energetico degli elaboratori.

Da esperimenti reali è stato appurato che il consumo energetico di ciascun server non è solo funzione della sua frequenza operativa, ma anche al valore di utilizzo della CPU della macchina. HP, attraverso alcune prove, è riuscita a determinare il modello di consumo energetico dei server di famiglia Intel (FIGURA 4.1) con una tolleranza del 10%.

Un server in sleep state (stato a più basso consumo) ha un costo energetico costante denotato con $\overline{c_i}$. Quando il server i viene acceso, il suo costo dinamico dipende da una costante data da $\overline{c_i}+p_{i,h}$ che è funzione della frequenza operativa e varia linearmente con l'utilizzo della CPU (Ut_i) con coefficiente angolare $b_{i,h}$.

L'introduzione del nuovo modello energetico necessita di una modifica alla funzione obiettivo per esprimere il legame tra costi e ricavi dati dagli SLA. Indicando con T il prossimo orizzonte temporale di controllo e denotando con $\overline{x}_i$ e $\overline{z}_{i,k,j}$ il valore delle variabili x_i e $z_{i,k,j}$ della soluzione nel precedente periodo di controllo, per la classe k il revenue è dato da :

$$V_k \, \Lambda_k \, T = (v_k - m_k \, R_k) \, \Lambda_k \, T.$$

L'utilizzo sull'i-esimo server è dato da :

$$Ut_i = \sum_{j \in N_k, k \in K} \frac{\lambda_{i,k,j}}{C_{i,h} \, \mu_{k,j}}$$

Il costo totale è dato dai costi dei server in power sleep, dai costi di consumo energetico dei server operativi e dagli overhead di accensione, spegnimento dei server e di migrazione delle VM. Quindi la differenza tra profitti e costi è data da:

$$\begin{aligned}\Bigg[\sum_{k\in K}\Bigg(-m_k\sum_{i\in I,j\in N_k}\frac{\lambda_{i,k,j}}{\left(\sum_{h\in H_i}C_{i,h}\,y_{i,h}\right)\mu_{k,j}\,\phi_{i,k,j}-\lambda_{i,k,j}}\Bigg)+\\ -\sum_{i\in I,h\in H_i}\left(p_{i,h}y_{i,h}+\,b_{i,h}\,y_{i,h}\frac{\sum_{j\in N_k,k\in K}\lambda_{i,k,j}}{\sum_{h\in H_i}C_{i,h}\,\mu_{k,j}\,y_{i,h}}\right)+\\ +\sum_{k\in K}v_k\Lambda_k-\sum_{i\in I}\bar{c}_i\Bigg]\,T+\\ -\sum_{i\in I}cs_i\max(0,x_i-\overline{x}_i)+\\ -\sum_{i\in I,k\in K,j\in N_k}cm\max(0,z_{i,k,j}-\overline{z}_{i,k,j})\end{aligned}$$

I termini $\sum_{k\in K}v_k\Lambda_k$ e $\sum_{i\in I}\bar{c}_i$ possono essere trascurati poiché non dipendono dalle variabili di decisione. Assumendo gli intervalli T, dei periodi di controllo, tutti uguali e pari ad 1 la funzione obiettivo risulta:

$$\begin{aligned}\max\Bigg\{\sum_{k\in K}\Bigg(-m_k\sum_{i\in I,j\in N_k}\frac{\lambda_{i,k,j}}{\left(\sum_{h\in H_i}C_{i,h}\,y_{i,h}\right)\mu_{k,j}\,\phi_{i,k,j}-\lambda_{i,k,j}}\Bigg)+\\ -\sum_{i\in I,h\in H_i}\left(p_{i,h}y_{i,h}+\,b_{i,h}\,y_{i,h}\frac{\sum_{j\in N_k,k\in K}\lambda_{i,k,j}}{\sum_{h\in H_i}C_{i,h}\,\mu_{k,j}\,y_{i,h}}\right)+\\ -\sum_{i\in I}cs_i\max(0,x_i-\overline{x}_i)+\\ -\sum_{i\in I,k\in K,j\in N_k}cm\max(0,z_{i,k,j}-\overline{z}_{i,k,j})\Bigg\}\end{aligned}$$

Considerando anche il vincolo di RAM discusso in 4.1.1, il modello di otti-

mizzazione può essere così espresso:

$$\max\Bigg\{\sum_{k\in K}\left(-m_k\sum_{i\in I,j\in N_k}\frac{\lambda_{i,k,j}}{\left(\sum_{h\in H_i}C_{i,h}\,y_{i,h}\right)\mu_{k,j}\,\phi_{i,k,j}-\lambda_{i,k,j}}\right)+$$
$$-\sum_{i\in I,h\in H_i}\left(p_{i,h}y_{i,h}+\,b_{i,h}\,y_{i,h}\frac{\sum_{j\in N_k,k\in K}\lambda_{i,k,j}}{\sum_{h\in H_i}C_{i,h}\,\mu_{k,j}\,y_{i,h}}\right)+$$
$$-\sum_{i\in I}cs_i\max(0,x_i-\overline{x}_i)+$$
$$-\sum_{i\in I,k\in K,j\in N_k}cm\max(0,z_{i,k,j}-\overline{z}_{i,k,j})\Bigg\}$$

$$\sum_{i=1}^{M}\lambda_{i,k,j}=\Lambda_k;\qquad\forall k,j \tag{4.11}$$
$$\sum_{k=1}^{K}\sum_{j=1}^{N_k}\phi_{i,k,j}\le 1;\qquad\forall i \tag{4.12}$$
$$z_{i,k,j}\le A_{i,k,j}x_i;\qquad\forall i,k,j \tag{4.13}$$
$$\lambda_{i,k,j}\le\Lambda_k z_{i,k,j};\qquad\forall i,k,j \tag{4.14}$$
$$\lambda_{i,k,j}<\left(\sum_{hinH_i}C_{i,h}f_{i,h}\right)\mu_{k,j}\phi_{i,k,j};\qquad\forall i,k,j \tag{4.15}$$
$$\sum_{h\in H_i}f_{i,h}=x_i;\qquad\forall i \tag{4.16}$$
$$\overline{A}_k\le\prod_{j=1}^{N_k}\left(1-\prod_{i=1}^{M}(1-a_i)^{z_{i,k,j}}\right);\qquad\forall k \tag{4.17}$$
$$\sum_{j=1}^{N_k}z_{i,k,j}\le 1;\qquad\forall i,k \tag{4.18}$$
$$\sum_{j\in N_k}RAM_{k,j}z_{i,k,j}\le\overline{RAM}_i\qquad\forall i\in M,k\in K \tag{4.19}$$
$$\phi_{i,k,j}\ge 0;\qquad\forall i,k,j$$
$$\lambda_{i,k,j}\ge 0;\qquad\forall i,k,j$$
$$x_i,\ z_{i,k,j},\ f_{i,h}\in[0,1];\qquad\forall i,k,j;\ \forall h\in H_i$$

Anche con l'introduzione di questo nuovo modello di consumo energetico la forma della funzione obiettivo non è né concava né convessa e conseguentemente il problema associato è un problema di difficile soluzione.

Di seguito vengono introdotte le modifiche rispetto agli algoritmi illustrati nel capitolo precedente necessarie per supportare il nuovo modello di consumo energetico.

4.1.2.1 Ricerca di una soluzione iniziale da soluzione precedente

Il nuovo modello di consumo energetico entra in gioco nel processo di ottimizzazione fin dalla ricerca di una soluzione iniziale (Sezioni 3.4.1 e 3.7.3), anche se l'idea di base risulta inalterata. Come già accennato infatti, il costo viene messo in relazione con due variabili: frequenza operativa e livello di utilizzo della CPU. All'inizializzazione dell'algoritmo vengono calcolati i nuovi parametri di load balancing e computati i residui di carico da assegnare. Durante l'assegnamento dei carichi residui sui server accesi attraverso la modifica della frequenza operativa, l'algoritmo deve passare ad utilizzare una diversa retta del modello di consumo per poter calcolare la variazione di costo. La relazione con l'utilizzo della CPU in questo caso può essere calcolata ed aggiornata in maniera esatta poiché, in questa fase, i server sono già stati attivati nella soluzione precedente. Conseguentemente l'aggiornamento di volta in volta del costo del server risulta semplice. Se però, non si riesce ad allocare tutto il traffico residuo, bisognerà assegnare il rimanente tramite l'accensione di nuovi server. In questa situazione occorrerà ordinare le classi aventi dei residui da allocare in maniera decrescente per numero di vincoli $\mathcal{B}$ ed in maniera crescente per valori di $\Gamma_{k,j}$mentre i server dovranno essere ordinati in modo crescente secondo il rapporto $\frac{c_i}{C_i}$. Il termine C_i non dipende dal modello di consumo, ma soltanto dalla frequenza operativa correntemente utilizzata. Purtroppo non si può dire altrettanto per il termine c_i, il quale indica il costo del server i-esimo. Poiché i costi energetici dipendono dal consumo, c_i è fortemente caratterizzato dalla modellizzazione energetica appena introdotta. L'algoritmo di ricerca parte a considerare il server alla frequenza operativa minima, per poi considerare in successione le altre frequenze solo nel caso in cui non si riesca ad allocare tutto il carico. Si intuisce quindi come l'algoritmo può sempre scegliere la retta opportuna associata alla frequenza correntemente utilizzata. Il temine c_i non dipende però solo dalla frequenza ma anche all'utilizzo della CPU. Tuttavia, contrariamente al passo precedente, è difficile stabilire un valore di utilizzo se non è stata ancora formulata un'allocazione delle applicazioni sui server. Peraltro, le rette posseggono coefficienti angolari differenti e non è quindi possibile stabilire una relazione d'ordine tra esse. Dai test effettuati si è verificato che considerando l'utilizzo pari al 50% si riesce ad avere un buon algoritmo di ordinamento. Considerando ciascuna retta in un punto si riesce bene a valutare quali siano i valori di consumo maggiori e

quali siano i minori, permettendo di creare una relazione d'ordine tra i termini di costo.

4.1.2.2 Fix point iteration basata sulle condizione KKT

L'algoritmo che si occupa di gestire i problemi di load balancing e capacity allocation descritto in 3.5.1 e in 3.7.4 è stato modificato in modo da considerare il nuovo modello energetico. In particolare la lambda iteration, così come precedentemente implementata, non è più adeguata alla risoluzione del problema di load balancing. Denotando con $I_{active} = \{i \in I | \, x_i = 1\}$ l'insieme dei server attivi, $\overline{C}_i = \sum_{h \in H_i} C_{i,h} y_{i,h}$ e $\overline{b}_i = \sum_{h \in H_i} (b_{i,h} + p_{i,h})\, y_{i,h}$. Il problema congiunto di capacity allocation e load balancing può essere formalizzato come segue:

$$\min \sum_{k \in K} \left(m_k \sum_{i \in I_{active}, j \in N_k} \frac{\lambda_{i,k,j}}{\overline{C}_i \mu_{k,j} \Phi_{i,k,j} - \lambda_{i,k,j}} + \frac{\overline{b}_i}{\overline{C}_i} \sum_{i \in I_a, j \in N_k} \frac{\lambda_{i,k,j}}{\mu_{k,j}} \right)$$

$$\sum_{i \in I_{active}} \lambda_{i,k,j} = \Lambda_k^{res} \qquad \forall k \in K, j \in N_k,$$

$$\sum_{k \in K, j \in N_k} \phi_{i,k,j} \leq 1 \qquad \forall i \in I_{active},$$

$$\lambda_{i,k,j} \leq \Lambda_k^{res} z_{i,k,j},\ \lambda_{i,k,j} < \overline{C}_i \mu_{k,j} \phi_{i,k,j},\ \lambda_{i,k,j}, \phi_{i,k,j} \geq 0 \quad \forall i \in I_{active}, k \in K, j \in N_k,$$

Considerando il modello di consumo energetico del modello discusso nella SEZIONE 3.3, la lambda iteration tentava, considerando costanti i parametri $\phi_{i,k,j}$, di migliorare la soluzione modificando i parametri $\lambda_{i,k,j}$ che potevano essere determinati in forma chiusa. Tuttavia, con il modello di consumo energetico introdotto in questa sezione non è più possibile esprimere in forma chiusa i valori di λ_i e di $\lambda_{\tilde{i}}$.

Considerando la variabili $\phi_{i,k,j}$ costanti e denotandole con $\overline{\phi}_{i,k,j}$, possiamo esprimere il problema di load balancing come segue:

$$\min \sum_{k \in K} \sum_{j \in N_k, i \in I_{active}} \lambda_{i,k,j} \left(\frac{m_k}{\overline{C}_i \mu_{k,j} \overline{\phi}_{i,k,j} - \lambda_{i,k,j}} + \frac{\overline{b_i}}{\overline{C}_i \mu_{k,j}} \right) \tag{4.20}$$

$$\textstyle\sum_{i \in I_{active}} \lambda_{i,k,j} = \Lambda_k \qquad \forall k \in K, j \in N_k,$$

$$\lambda_{i,k,j} \leq \Lambda_k z_{i,k,j}, \quad \lambda_{i,k,j} < \overline{C}_i \mu_{k,j} \overline{\phi}_{i,k,j}, \quad \lambda_{i,k,j} \geq 0 \quad \forall i \in I_{active}, k \in K, j \in N_k,$$

dove $\lambda_{i,k,j}$ sono le uniche variabili decisionali.

Lo scopo è minimizzare la media pesata dei tempi di risposta di ogni tier su ogni server. Il problema (4.20) può essere risolto mediante la risoluzione di $\sum_{k \in K} |N_k|$ sottoproblemi di load balancing, ciascuno riferito ad ogni tier di

ciascuna classe risolvibile indipendente degli altri. La funzione obiettivo risulta essere convessa[1] e gli autovalori sono tutti positivi, per cui è possibile trovare la soluzione ottima ad ogni sottoproblema.

Ignorando gli indici k e j di ciascun sotto-problema (4.20) e omettendo il vincolo $\lambda_i \leq \Lambda_k z_{i,k,j}$ sotto l'ipotesi di un corretto assegnamento tra tier applicativi e server, il problema (4.20) può essere scomposto come segue:

$$min \sum_{i \in I_{active}} \lambda_i \left(\frac{m_k}{U_i - \lambda_i} + \frac{\overline{b_i}}{\overline{C}_i \mu} \right)$$

$$\sum_{i \in I_{active}} \lambda_i = \Lambda, 0 \leq \lambda_i < U_i \forall i \in I_{active}.$$

Se un tier non è allocato su di un server $U_i = 0$, altrimenti U_i sarà pari a $\overline{C}_i \cdot \mu \cdot \overline{\phi}_i$.

Il lagrangiano è dato da :

$$\mathcal{L}(\lambda, L, L_i, L'_i) = \sum_{i \in I_{active}} \lambda_i \left(\frac{m_k}{U_i - \lambda_i} + \psi_i \right) - L \left(\Lambda - \sum_{i \in I_{active}} \lambda_i \right) + \\ - \sum_{i \in I_{active}} L_i \left(U_i - \lambda_i \right) - \sum_{i \in I_{active}} L'_i \lambda_i$$

con $\psi_i = \frac{\overline{b}_i}{\overline{C}_i \, \mu}$. Le condizioni KKT di ottimalità sono:

$$\frac{\partial \mathcal{L}(\lambda, L, L_i, L'_i)}{\partial \lambda_i} = \frac{m_k \, U_i}{(U_i - \lambda_i)^2} + \psi_i + L + L_i - L'_i = 0 \quad \forall i \in I_{active}, \tag{4.21}$$

$$\begin{aligned} \sum_{i \in I_{active}} \lambda_i = \Lambda, \quad & \forall i \in I_{active} \\ 0 \leq \lambda_i < U_i, \quad & \forall i \in I_{active} \\ L_i(U_i - \lambda_i) = 0 \quad L'_i \lambda_i = 0, \quad L_i, \quad & \forall i \in I_{active} \\ L'_i \geq 0, \quad & \forall i \in I_{active} \end{aligned}$$

Poiché le condizioni di equilibrio impongono la disuguaglianza stretta $\lambda_i < U_i$

si ha che i moltiplicatori L_i sono nulli. Tuttavia, se realisticamente si assume che $\lambda_i > 0$ (che implica $L'_i = 0$), le condizioni $\frac{\partial \mathcal{L}(\lambda, L)}{\partial \lambda_i} = 0$ diventano

$$\frac{mk \, U_i}{(U_i - \lambda_i)^2} + \psi_i + L = 0.$$

[1] la matrice Hessiana è data da $Diag\left(\frac{2 m_k \overline{C}_i \mu_{k,j} \overline{\phi}_{i,k,j}}{(\overline{C}_i \mu_{k,j} \overline{\phi}_{i,k,j} - \lambda_{i,k,j})^3} \right)$

Ciò implica che per ogni $i_1, i_2 \in I_{active}$ si ha:

$$\frac{m_k U_{i_1}}{(U_{i_1} - \lambda_{i_1})^2} + \psi_{i_1} = \frac{m_k\, U_{i_2}}{(U_{i_2} - \lambda_{i_2})^2} + \psi_{i_2}.$$

Conseguentemente si ha:

$$\frac{m_k U_{i_1}}{(U_{i_1} - \lambda_{i_1})^2} = \frac{m_k\, U_{i_2} + (\psi_{i_2} - \psi_{i_1})\,(U - \lambda_{i_2})^2}{(U_{i_2} - \lambda_{i_2})^2},$$

da cui:

$$(U_{i_1} - \lambda_{i_1}) = \frac{\sqrt{m_k\, U_{i_1}}\,(U_{i_2} - \lambda_{i_2})}{\sqrt{m_k\, U_{i_2} + (\psi_{i_2} - \psi_{i_1})\,(U_{i_2} - \lambda_{i_2})^2}}. \tag{4.22}$$

L'equazione (4.22) permette di esprimere il valore di λ_i in funzione di una classe $\lambda_{\bar{i}}$ scelta arbitrariamente :

$$\lambda_i = U_i - \frac{\sqrt{m_k\, U_i}\,(U_{\bar{i}} - \lambda_{\bar{i}})}{\sqrt{m_k\, U_{\bar{i}} + (\psi_{\bar{i}} - \psi_i)\,(U_{\bar{i}} - \lambda_{\bar{i}})^2}}. \tag{4.23}$$

Assumendo $\sum_{i \in I_{active}} \lambda_i = \Lambda$, segue che:

$$\lambda_{\bar{i}} + \sum_{i \in I_{active} - \{\bar{i}\}} U_i - \frac{\sqrt{m_k\, U_i}\,(U_{\bar{i}} - \lambda_{\bar{i}})}{\sqrt{m_k\, U_{\bar{i}} + (\psi_{\bar{i}} - \psi_i)\,(U_{\bar{i}} - \lambda_{\bar{i}})^2}} = \Lambda,$$

quindi :

$$\sum_{i \in I_{active}} U_i - \frac{\sqrt{m_k\, U_i}\,(U_{\bar{i}} - \lambda_{\bar{i}})}{\sqrt{m_k\, U_{\bar{i}} + (\psi_{\bar{i}} - \psi_i)\,(U_{\bar{i}} - \lambda_{\bar{i}})^2}} = \Lambda. \tag{4.24}$$

In questo modo è possibile calcolare tutti i valori λ_i. Se $\lambda_i \geq 0\ \forall i \in I_{active}$, la soluzione è accettabile ed ottima. Altrimenti se alcuni valori diλ_i sono minori di zero la formula precedente deve riessere calcolata assumendo al meno un valore $\lambda_i = 0$. Il calcolo deve essere ripetuto dopo aver imposto $\lambda_{i'} = 0$, dove ad i' corrisponde l'indice del minor U_i. In questo modo il tempo medio d risposta di una classe condiviso su più server è minimizzato allocando il traffico sui server migliori (caratterizzati da grandi valori di U_i).

Per risolvere la (4.24) si può ricorrere ad un metodo iterativo per il calcolo degli zeri di una funzione come ad esempio quello di Newton. Nell'implementazione dell'algoritmo per ragioni di efficienza è stato utilizzato il metodo di Newton-Raphson abbinato al metodo di bisezione. L'algoritmo iterativo di Newton-Raphson è uno dei metodi del secondo ordine per il calcolo approssimato degli zeri di un'equazione. Esso si applica dopo avere determinato un intervallo $[a, b]$ che contiene una sola radice. La bisezione è un metodo iterativo più lento di Newton-Raphson che ad ogni passo dimezza l'intervallo di partenza, restringendo il campo di ricerca dello zero. Combinandoli si ottiene un metodo

che ad ogni passo permette di ridurre l'intervallo su cui effettuare la ricerca dello zero sfruttando l'efficienza di Newton-Raphson [1].

Per quanto riguarda il problema di capacity allocation può essere modellizzato come segue:

$$\min \sum_{k=1}^{K} m_k \sum_{i \in \mathcal{I}} \sum_{j=1}^{N_k} \frac{\overline{\lambda}_{i,k,j}}{C_i \mu_{k,j} \phi_{i,k,j} - \overline{\lambda}_{i,k,j}} \tag{4.25}$$

$$\phi_{i,k,j} \geq 0$$
$$\phi_{i,k,j} > \frac{\overline{\lambda}_{i,k,j}}{C_i \mu_{k,j}}$$

$$\sum_{k=1}^{K} \sum_{j=1}^{N_k} \phi_{i,k,j} \leq 1 \tag{4.26}$$

Come si può osservare, questo problema risulta totalmente analogo a quello già mostrato in Sezione 3.5.1, quindi non sono state apportate modifiche alla phi iteration.

Riassumendo si può affermare che la nuova lambda iteration non permette più di trovare in forma chiusa il valore della λ_i e $\lambda_{\bar{i}}$, ma bisogna ricorrere ad un metodo numerico che consenta la risoluzione dell'equazione (4.24). Per il resto l'algoritmo nel suo complesso è analogo a quello mostrato in Sezione 3.5.1.

4.1.2.3 Ricerca locale

Nel processo di ricerca locale è stato necessario applicare per ogni mossa delle modifiche per poter considerare la funzione obiettivo presentata in Sezione 4.1.2 per il corretto calcolo dei costi. In particolare, in ogni mossa, nella valutazione del costo energetico deve essere considerato l'utilizzo del server. Ad esempio accendendo un nuovo server sarà necessario calcolare il suo utilizzo considerando il traffico che condividerà con il server saturo.

4.2 Tecniche di resource allocation gerarchica

Un sistema di autonomic computing è un sistema informatico complesso, costituito da molti componenti interconnessi che operano su diverse scale temporali per soddisfare determinati obiettivi, stabiliti ad alto livello [21]. Questi obiettivi, come descritto nel capitolo precedente, sono solitamente perseguiti rappresentando, tramite modelli matematici, l'architettura sottostante e i vincoli o le leggi

che regolano l'intero sistema. Negli ultimi anni i Service Center hanno seguito un forte evoluzione in termini di complessità e di dimensioni. Questa evoluzione è stata possibile grazie alle tecnologie di virtualizzazione, fattore abilitante della cosiddetta server consolidation che prevede un utilizzo delle risorse efficiente ed ottimizzato. I Service Center sono cresciuti anche in dimensioni. Oggigiorno in un Service Center si può arrivare ad includere fino a 10.000 server e gestire migliaia di applicativi [17]. In questo scenario si può ben capire perché gli algoritmi di ottimizzazione debbano essere sempre più efficienti. La crescente complessità dei sistemi informatici attuali e futuri suggerisce quindi un approccio decentralizzato al problema, in modo da poter gestire un'infrastruttura flessibile e scalabile. Il risolutore (centralizzato) discusso nel capitolo precedente ha come limite quello di poter gestire agevolmente ed in tempi utili piccoli sistemi, con un numero di server fisici inferiore a 500. Questa soglia non permette di valutare le prestazioni e l'impatto su i grossi sistemi, realmente implementati.

Questa limitazione ha reso necessaria l'implementazione di un risolutore gerarchico che possa considerare dei sottosistemi di dimensioni adeguate per la risoluzione tramite l'ottimizzatore centralizzato precedentemente sviluppato. L'obiettivo di questa nuova tecnica di resource allocation sarà quello di trovare una soluzione che si accosti il più possibile all'ottimo centralizzato ma in tempi molto più brevi rispetto a quelli che caratterizzano le tecniche di gestione delle risorse discusse nel capitolo precedente. Inoltre la maggior efficienza non dovrà compromettere la qualità delle soluzione finale. Tuttavia, ci si aspetta di dover gestire un compromesso tra tempi di soluzione e precisione. Supponendo di dividere un Service Center in sottosistemi formati da gruppi di server ed applicazioni, le tecniche gerarchiche tendono a considerare indipendentemente ciascuna porzione dell'infrastruttura e quindi a ridurre le possibilità di ottimizzazione. Un aspetto critico infatti sarà valutare la relazione che intercorre tra la dimensione dei sottosistemi, i tempi di esecuzione e la precisione dell'ottimo che verrà trovato.

L'idea di decentralizzazione gerarchica utilizzata in questo lavoro di tesi è presentata in [43] e verrà illustrata nella sezione successiva.

4.2.1 Modello gerarchico per l'allocazione delle risorse in Service Center di larga scala

In [43] è stato presentato un approccio gerarchico che cerca di garantire le proprietà self-* che caratterizzano i sistemi autonomici (Sezione 2.5). Il metodo è quindi molto generale e nel presente lavoro di tesi è stato adattato per sviluppare una tecnica di resource allocation gerarchica che si basa sul partizionamento delle classi applicative. L'obiettivo è quello di implementare un risolutore decentralizzato che permetta di assegnare a ciascun gruppo di applicazioni una determinata quantità di risorse. Internamente a ciascun partizione applicativa

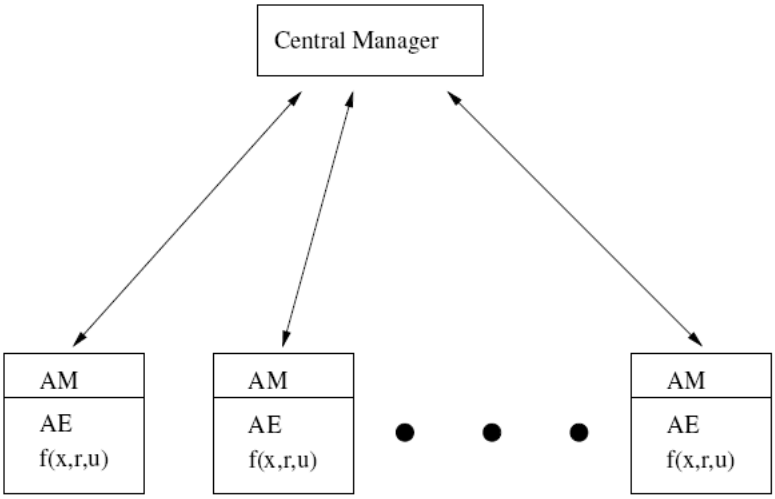

Figura 4.2: Modello del sistema decentralizzato

le risorse disponibili verranno allocate mediante l'utilizzo dell'ottimizzatore illustrato nel capitolo precedente. Verrà ora illustrato il modello generale discusso in [43], si procederà poi a considerazioni specifiche per le tecnica di resource allocation che costituiscono l'obiettivo della tesi.

Un ottimizzatore gerarchico utilizza un modello basato su partizioni dell'intera infrastruttura. Ciascuna sottosistema rappresenta un problema di ottimizzazione distinto da risolvere. Ogni partizione costituisce un Application Environment (AE), ognuno dei quali ha un Application Manager (AM) che ottimizza e controlla l'allocazione delle risorse all'interno della stessa, ricercando l'ottimo per il sottosistema. Inoltre, ciascun AM permette di gestire le operazioni tra i vari AE che compongono il modello. Ogni sottosistema quindi è costituito da un insieme di tier applicativi, con determinate caratteristiche, e da un insieme di macchine sulle quali poterli eseguire. (Si faccia riferimento alla Figura 4.2).

Tutti gli Application Manager sono controllati da un Central Manager (CM), il cui compito è quello di assegnare le risorse del sistema ai vari AM. L'obiettivo del Central Manager è quello di ottimizzare la funzione obiettivo dell'intero sistema che dipenderà dagli ottimi trovati dagli AM e conseguentemente dalle risorse che vengono messe a disposizione a ciascun ambiente applicativo. In seguito con il termine "ottimo centralizzato" ci si riferirà alla soluzione ottima trovata mediante l'algoritmo, illustrato nel capitolo precedente, applicato all'intera struttura mentre l'ottimo trovato dal CM, che verrà chiamato ottimo distribuito.

Il sistema gerarchico appena illustrato può contenere più livelli, dove ciascun AM controlla e distribuisce le risorse ai suoi livelli sottostanti. Nel presente lavoro di tesi ci si è soffermati a considerare un solo livello.

Si consideri una funzione obiettivo $f_i(x_i, r_i, u_i)$ da massimizzare, simile a quella discussa per il risolutore centralizzato, associata all'AM i-esimo. Si denoterà con x_i l'insieme di variabili decisionali utilizzate nell'Application Manager i, r_i sarà l'insieme delle risorse allocate dal CM all'AM i-esimo. Infine, u_i sarà l'insieme delle variabili esterne che influenzano il comportamento dell'Application Manager i. Considerando il modello usato per progettare il solver centralizzato discusso in precedenza, r_i includerà l'insieme dei server assegnati all'Application manager i-esimo, x_i saranno tutte le variabili decisionali mostrate in SEZIONE 4.1.1, mentre u_i conterrà il traffico esterno verso l'Application Manager in questione.

Come già illustrato nel capitolo precedente l'insieme delle variabili x_i deve rispettare alcuni vincoli che verranno denotati con V_i ($x_i \in V_i$ rappresenta la regione ammissibile per l'Application Manager i). I vincoli, in generale, dipenderanno anche da r_i e da u_i ($r_i \in R_i$).

Secondo un approccio centralizzato, la funzione obiettivo totale del sistema sarà data da:

$$h(\, f_1(x_1, r_1, u_1)\,, ...,\, f_n(x_n, r_n, u_n)\,)$$

dove h rappresenta una funzione che permette di combinare le varie funzioni obiettivo degli AM. Le funzioni h possono essere modellate come somma, massimo oppure minimo.

L'obiettivo globale del sistema centralizzato è quello di massimizzare la differenza tra revenue e costi del sistema. Conseguentemente si ottiene la seguente funzione formulazione:

$$h_c = \max_{x_i \in V_i, r_i \in R_i} h(\, f_1(x_1, r_1, u_1)\,, ...,\, f_n(x_n, r_n, u_n)\,) \tag{4.27}$$

dove ($r_1 \ldots r_n$) $\in R$ (insieme finito di tutte le risorse disponibili nel sistema) e $x_i \in V_i(r_i, u_i)$.

Il valore h_c rappresenta l'ottimo per l'intero sistema, quindi proprio l'ottimo centralizzato.

In ottica di ottimizzazione distribuita ciascun Application Manager tenta di massimizzare la propria funzione obiettivo:

$$g_i(r_i, u_i) = \max_{x_i \in V_i} f_i(x_i, r_i, u_i) \tag{4.28}$$

dove $x_i \in V_i(r_i, u_i)$ e r_i è l'insieme delle risorse assegnate dal Central Manager all'Application Manager i.

L'algoritmo di ottimizzazione prevede che il Central Manager determini l'allocazione delle risorse risolvendo il seguente problema di ottimizzazione:

$$h_d = \max_{r_i \in R_i} h\left(g_1(r_1, u_1), \ \ldots \ , g_n(r_n, u_n)\right) \tag{4.29}$$

Quindi, ciascun AM i ottimizza la propria funzione obiettivo e passa il valore ottimo trovato al CM. Questo procedimento fa si che il Central Manager non debba conoscere la forma di f.

Definizione 4.1 (funzione OPGT)**.** Data una funzione $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$, essa preserva l'ordine rispetto all'operatore di confronto $\geq$ (OPGT) se $g(x) \geq g(y)$ con $x \geq y$.

Esempi di funzioni OPGT sono somma e minimo, massimo.

Sotto l'ipotesi che u_i sia costante durante il periodo in considerazione (ipotesi non restrittiva) in [43] è stato dimostrato un teorema che afferma la bontà dell'ottimo decentralizzato h_d rispetto all'ottimo centralizzato h_c.

Teorema 4.2.1. *Se la funzione di aggregazione f è OPGT, allora $h_c \equiv h_d$. In altre parole la qualità della soluzione ottima ottenuta mediante un sistema gerarchico è equivalente a quella di un sistema centralizzato.*

Il teorema appena enunciato risulta verificato per il modello di ottimizzazione considerato nel lavoro di tesi, in quanto si è associata ad h la funzione somma. Il modello di ottimizzazione distribuita risulta quindi essere:

$$h_d = \max_{r_i \in R_i} \sum_i g_1(r_1, u_1), \ \ldots \ , g_n(r_n, u_n) \tag{4.30}$$

Un algoritmo che ottimizza ciclicamente il sistema tramite l'Equazione (4.30) risulta essere più efficacie ed efficiente se, oltre a poter valutare la funzione obiettivo:

$$h(r_1, \ \ldots \ , r_n) = h(g_1(r_1, u_1), \ \ldots \ , g_n(r_n, u_n)) \tag{4.31}$$

è in grado di valutare anche il suo gradiente $\nabla\tilde{h}$:

$$\nabla\tilde{h} = \sum_i \nabla_i h \cdot \frac{\partial g_i}{\partial r} \tag{4.32}$$

con $\frac{\partial g_i}{\partial r_i} = 0$ per tutte le $i \neq j$.

Si assumano i vincoli $x_i \in C_i\ (r_i, u_i)$ scritti nella forma $c_i(x_i, u_i) = r_i$, oppure $c_i(x_i, u_i) \leq r_i$. Ne segue che i termini $-\frac{\partial g_i}{\partial r_i}$ sono i moltiplicatori di Lagrange relativi all'equazione (4.28). Ciascun Application Manager, oltre ad inviare al Central Manager il valore dell'ottimo locale $g_i(r_i, u_i)$, deve comunicare anche

il corrispondente moltiplicatore lagrangiano, in modo che il CM possa agevolmente calcolare il relativo gradiente. In base alla definizione di moltiplicatore di Lagrange esso rappresenta la variazione della funzione obiettivo verificatasi in presenza di una variazione delle risorse messe a disposizione all'Application Manager. Ad esempio, supponiamo che sia presente un cluster contenente 5 server e 2 applicazioni. L'Application Manager computa per la prima volta g e trova un valore ottimo pari a 0.1. A fronte di questo risultato il Central Manager decide di incrementare ulteriormente le risorse a disposizione di tale gruppo, mettendo così a disposizione ulteriori 2 server. L'AM ricalcola il valore ottimo che sale a 0.2. Il coefficiente lagrangiano sarà quindi dato da: 0.2-0.1/(7-5). Dall'esempio appena illustrato è facile capire perché il moltiplicatore di Lagrange sia importante per rendere efficiente l'algoritmo: esso "indica" al Central Manager se conviene incrementare o meno le risorse di un dato cluster, guidando nella giusta direzione il CM verso l'ottimo.

1 **repeat**
2 | Il CM assegna le r_i a ciascun AM i;
3 | Se necessario il CM comunica l'insieme delle variabili esterne u_i a ciascun AM i (queste potrebbero già essere disponibili agli AM) ;
4 | Ciascun AM calcola $g_i(r_i, u_i)$ ed invia l'ottimo trovato al CM e il moltiplicatore lagrangiano corrispondente;
5 | Il CM calcola h e $\nabla\tilde{h}$ e trova la prossima allocazione delle risorse $(r_1, \ldots, r_n)$;
6 **until** *l'algoritmo converge, oppure viene trovata una buona allocazione, oppure vi è un evento di time out*;

Algoritmo 8: Algoritmo dei sistema decentralizzato.

L'Algoritmo 8 usa uno schema iterativo (Passo 6) che tenta, passo dopo passo, di avvicinarsi all'ottimo. In ciascuna iterazione non è richiesta la comunicazione tra tutti gli AM e il CM, ma solo da parte di coloro che hanno ricevuto una variazione di risorse. Un discorso analogo può essere fatto in presenza di risorse r_i multiple: $g_i(r_i, u_i)$ non deve essere computata per tutte ma solo per il sottoinsieme di risorse possedute dall'AM. Inoltre, come verrà mostrato in seguito, il valore $g_i(r_i, u_i)$ può essere calcolata (Passo 4) in maniera approssimata (attraverso bound) e può essere raffinato man mano, nel corso delle iterazioni successive.

4.2.2 Implementazione dell'algoritmo di ottimizzazione gerarchica

In questa sezione verranno esplicitate le scelte implementative utilizzate per adattare il modello discusso nella Sezione 4.2.1 al problema di allocazione del-

le risorse. Come primo aspetto, si devono stabilire quali siano le variabili (x_i, u_i) e le risorse (r_i) in gioco nel sistema. Come già accennato, vengono associate alle variabili x_i tutte le variabili decisionali, già prese in considerazione nel modello contenuto nella SEZIONE 4.1.2. Queste variabili sono considerate interne all'Application Manager. I loro valori determinano l'allocazione ottima, entro una regione ammissibile fornita dei vincoli del modello alla SEZIONE 4.1.2. Le variabili u_i, invece, vengono considerate esterne. Rappresentano eventi che possono modificare le decisioni di allocazione dell'AM. Nel caso del modello preso in considerazione, esse trovano riscontro nel traffico in arrivo alle varie applicazioni. In fine, sia associa alle r_i, ovvero le risorse del sistema, la capacità elaborativa di un insieme di server. Ciascun elaboratore infatti, possiede una certa capacità elaborativa che mette a disposizione del cluster di risorse associato ad una partizione. Nel modello distribuito, visto in precedenza, occorre che il Central Manager possa aumentare o diminuire a ciascun cluster la capacità. Nel seguito, quindi, astrarremo dal concetto di server e considereremo le risorse assegnate ad una partizione come capacità elaborativa; in altre parole se verranno assegnate 5 unità elaborative (C.E.) sarà ininfluente il fatto che esse appartengano ad un server o vengano fornite da più macchine.

Attraverso questa visione del modello gerarchico risulta chiaro come l'ottimizzatore centralizzato, precedentemente discusso (si faccia riferimento alla SEZIONE 4.1.1), costituisca l'Application Manager, che dovrà operare considerando un sotto-insieme di risorse (capacità elaborative) ed un sotto-insieme di applicazioni da allocare. Tuttavia, per ragioni di efficienza l'algoritmo adottato dagli AM utilizzerà solamente un sottoinsieme di tutte le mosse presentate. Sperimentalmente è stato verificato che le mosse di accensione e spegnimento di un server sono quelle che consentono di accostarsi più velocemente all'ottimo. Si è scelto quindi di utilizzare solamente queste due mosse.

L'algoritmo che gestirà il Central Manager si occuperà di inizializzare correttamente il sistema formando le varie partizioni applicative. Dovrà poi, variare iterativamente le risorse a disposizione degli AM. La formazione delle partizioni deve essere effettuata tramite un criterio che permetta di aggregare al meglio le classi applicative che costituiscono sotto-sistemi disgiunti di Virtual Machine. Una volta formati i sotto-insiemi di virtual machine si procederà, tramite un algoritmo greedy, ad assegnare un insieme di server. Sarà da questo insieme di risorse che l'algoritmo procederà, tramite mosse, nelle varie iterazioni. Queste mosse permetteranno sia di aggiungere o rimuovere C.E. dai cluster, sia di scambiare unità di capacità elaborativa tra gli Application Manager.

L'algoritmo gerarchico dovrà operare su una scala temporale di 24 ore, pianificando l'allocazione delle risorse ora per ora. Precisamente:

- Ad inizio di ciascun giorno verrà partizionato l'insieme delle classi applicative, attraverso una procedura euristica spiegata in seguito, considerando

previsioni di traffico di tutta la giornata. La divisione in gruppi delle classi deve considerare almeno 24 (una per ora) previsioni di traffico per ciascuna applicazione.

- L'assegnamento della C.E. alle partizioni applicative verrà effettuata ora per ora considerando il traffico relativo al periodo al periodo corrente.

4.2.2.1 Implementazione degli Application Manager

L'Application Manager riceve dal Central Manager un insieme di classi applicative ed un insieme di macchine fisiche. Ciascun AM quindi rappresenta un sotto-sistema del Service Center globale. Come già accennato, l'algoritmo che regola ciascun Application Manager è quello illustrato nel capitolo precedente (Sezione 4.1.2). Le uniche modifiche introdotte riguardano l'utilizzo di un sistema di thread che incapsula le varie procedure, dando la possibilità di eseguire in parallelo la risoluzione di tutti i sotto-sistemi. L'implementazione tramite thread ha aiutato a migliorare prestazionalmente la ricerca dell'ottimo. Una seconda modifica introdotta è stata finalizzata a limitare l'algoritmo di ricerca entro un tempo prestabilito. Si è quindi introdotto un "time limit" allo scadere del quale l'algoritmo non procede oltre nella risoluzione del problema ma riporta il risultato ottenuto fino a quel momento. Questo metodo permette di garantire l'esecuzione dell'algoritmo di ricerca entro un determinato periodo temporale. Un'ulteriore modifica eseguita sul software ha permesso di segnalare, in presenza di soluzione unfeasible (non accettabile), la causa del problema, al fine di recuperare la correttezza della soluzione nel Central Manager. In fine, come già accennato, a seguito di esperimenti effettuati si è ritenuto vantaggioso attivare nell'algoritmo dell'AM solamente le mosse di accensione e spegnimento dei server che permettono di avvicinarsi in poco tempo al valore ottimo. Le altre mosse infatti costituisco azioni di fine tunning della ricerca locale. Con questa soluzione si permette di aumentare l'efficienza della procedura di local search a discapito di una sottostima del valore ottimo.

4.2.2.2 Partizionamento delle classi applicative

Il primo compito del Central Manager è quello di suddividere il Service Center in sottosistemi. Questo procedimento viene effettuato mediante il partizionamento dell'insieme delle applicazioni. A ciascuna partizione applicativa sarà poi assegnato un cluster di macchine fisiche, ottenuto mediante un partizionamento periodico (ora per ora) dei server. Tipicamente, la ripartizione delle risorse elaborative nelle partizioni viene effettuata basandosi sulla previsione di picco del traffico in ingresso alle applicazioni. In tal modo si ha la sicurezza di riuscire ad elaborare in tempo utile tutte le richieste, massimizzando le revenue. Di contro con tale metodo, si allocano risorse che resteranno inutilizzate per lunghi

periodi di tempo con il conseguente effetto di spreco della capacità elaborativa che potrebbe essere utilizzata in altri gruppi. L'algoritmo di partizionamento progettato, si pone come obiettivo quello di effettuare un partizionamento opportuno delle applicazioni in modo da evitare tale problema. Per raggiungere questo scopo si andranno ad utilizzare delle previsioni sul carico di ciascuna classe.

Un ulteriore fattore da tenere in considerazione durante questo processo, riguarda il numero di partizioni da creare. Questa dimensione influisce sia sulla velocità, sia sulla qualità dell'algoritmo gerarchico. Infatti, partizionando molto il sistema verranno assegnate poche unità elaborative a ciascun gruppo e non sarà possibile effettuare alcuna condivisione delle risorse. Per risolvere questo problema l'algoritmo dovrà ricorrere a dei meccanismi di condivisione delle risorse tra partizioni scariche e partizioni sovraccariche. Questo meccanismo andrà ad incrementare il tempo di elaborazione e il livello di complessità dell'algoritmo. Viceversa considerando poche partizioni ci si accosta ad modello centralizzato, affetto da lunghi tempi di risoluzione. La scelta quindi delle dimensioni del numero di partizioni è stata stabilita sperimentalmente e viene calcolata a run-time basandosi su le dimensioni massime che ogni partizione applicativa deve possedere.

Verranno ora introdotte alcune definizioni che serviranno per esprimere la metrica di aggregazione delle applicazioni. Si assume d'ora in poi che ogni traccia contenente le previsioni di traffico per le applicazioni, sia composta da T previsioni.

Definizione 4.2 (Area Under the Forecast)**.** Assumendo che ciascun periodo di previsione sia uguale a 1/T, può essere definita l'area sottostante la curva di previsione di una classe applicativa $\Lambda_k(t)$:

$$AUF(k) := \frac{1}{T} \sum_{t=1}^{T} \Lambda_k(t)$$

Definizione 4.3 (Area Summation)**.** Alla r-esima iterazione dell'algoritmo di class partitioning, per una data partizione P_n, l'area totale è definita come la sommatoria di tutte le aree sottostanti la curva di previsione di tutte le classi assegnate a P_n, nelle precedenti r-1 iterazioni. Quindi l'area totale della partizione P_n, è data:

$$AS(P_n) := \sum_{k \in P_n} A(k)$$

Osservazione 1.
Da quest'ultima definizione (4.3) si può dedurre che:

$$AS(P_i \cup P_j) := \sum_{k \in P_n \cup P_j} A(k) = AS(P_i) + AS(P_j)$$

Come sottolineato in precedenza, è desiderabile che due classi facenti parte di una stessa partizione abbiamo due profili di traffico tali che quando il carico di richieste della prima sia sopra la media, quello dell'altra sia sotto la media. In questo modo, le risorse non utilizzate dalla seconda possono essere utilizzate dalla prima. Più in generale: quando l'algoritmo deve decidere se raggruppare due partizioni in una unica o lasciare le due partizione separate verrà utilizzato un criterio basato sul calcolo del rapporto delle aree (Area ratio).

Definizione 4.4 (Area Ratio)**.** Date due partizioni di classi P_i e P_j, l' Area Ratio è dato da:

$$AR(P_i, P_j) := MAX(P_i \cup P_j)/AS(P_i \cup P_j)$$

Si supponga di voler raggruppare due partizioni P_i e P_j che hanno una previsione di carico simile. Dopo il raggruppamento la Maximum Resquest Function risulta leggermente modificata, ma l'Area Summmation aumenta di $A(P_j)$. Viceversa prendendo due distribuzioni con andamento complementare, la Maximum Resquest Function è l'unione delle due, mentre l'Area Summation viene incrementata sempre dello stesso valore. In questo secondo caso l'Area Ratio è massimo. Si considerino ad esempio i profili di traffico mostrati in FIGURA 4.3. L'area ratio della partizione 1-2 risulta essere 1, mentre quella della partizione 1-3 è uguale a 0.5. Si può notare come raggruppando (Figura 4.4) le classi 1 e 2, le cui previsioni di traffico sono complementari, la Maximum Request Function risulti essere pari a 0.5. Tuttavia, raggruppando le classi 1 e 3, aventi previsioni di traffico simili, la Maximum Request Function aumenta il suo valore durante le prime 12 ore e non assegna carico nelle successive. Da questo esempio si intuisce come nella partizione 1-3 sia stato più che raddoppiato il picco di richieste nella prima parte della giornata, costringendo ad assegnare una quantità di risorse tale da consentire una efficacie elaborazione. Tuttavia, tutte le risorse assegnate per gestire le prime 12 ore saranno inutilizzate per il resto della giornata.

Si considerino ora le previsioni di traffico in FIGURA 4.5. Raggruppando due classi (4 e 5) le cui previsioni di traffico sono simili ci si trova nella situazione in cui i picchi delle previsioni coincidono, andando a sommarsi. Viceversa, raggruppando due classi (4 e 6) in cui i picchi dell'una coincidono con le valli dell'altra si ottiene un profilo di traffico della partizione più omogeneo. Quindi raggruppare le classi 4 e 5 è meno conveniente rispetto a raggruppare le classi

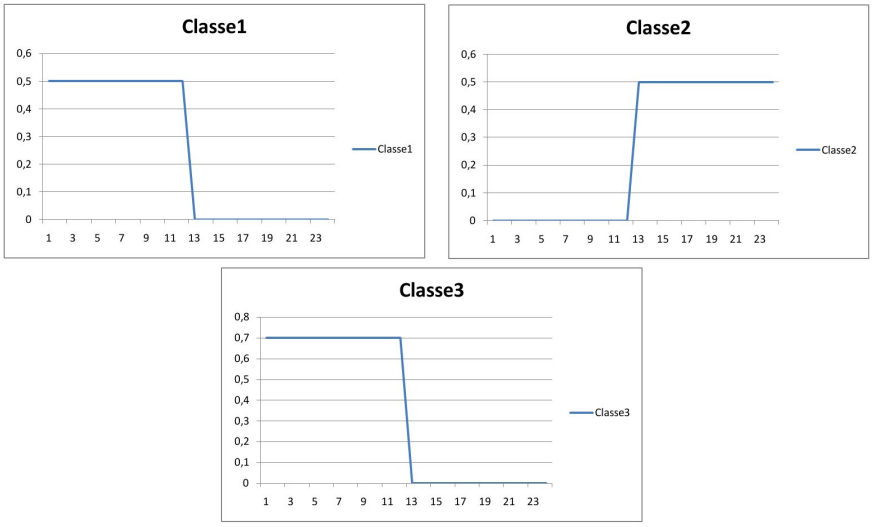

Figura 4.3: Esempi di due profili di traffico complementari (1, 2) e di due profili simili (1, 3).

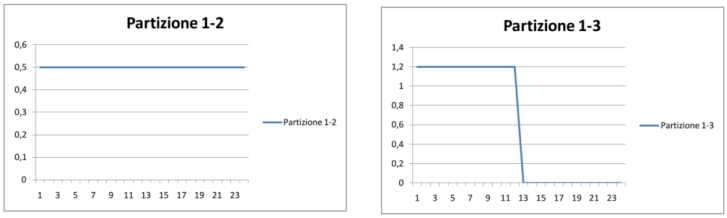

Figura 4.4: Esempi di Maximum Request Function.

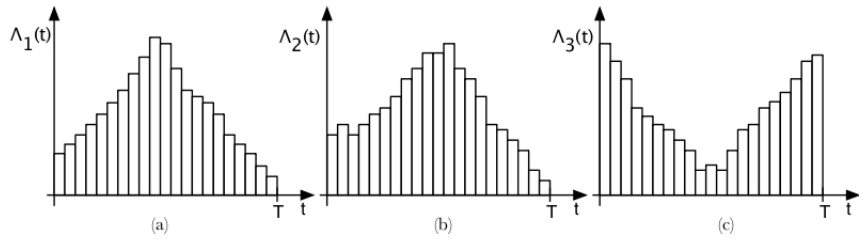

Figura 4.5: Tre grafici di previsione di traffico per le classi 4 (a), 5 (b) e 6 (c)

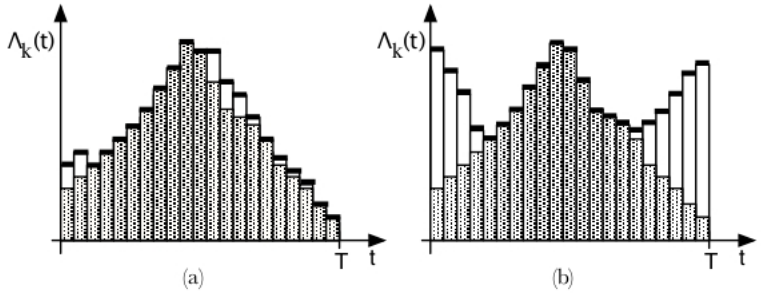

Figura 4.6: Grafici dei raggruppamenti effettuati. 4-5 (a), 4-6 (b)

4 e 6. Infatti, osservando la FIGURA 4.6 in (a) l'area sottostante la Maximum Request Function sono simili all'area di entrambe le classi 4 e 5. L'area Summation della partizione 4-5 risulta essere pari a 2.7, mentre $MAX(4 \cup 5)$ vale 1.4. L'Area Ratio circa pari a 0.5. Viceversa, in (b), l'area sotto la Maximum Request Function è più "omogenea" e riflette la complementarietà delle due classi. In questo caso l'Area Ratio vale 0.93. In sintesi, si può affermare che il criterio appena illustrato tende ad aggregare previsioni di traffico in cui i picchi della prima curva cadono nelle valli della seconda, in modo da avere un traffico mediamente più omogeneo. Non si aggregheranno quindi classi con distribuzioni di traffico simili, con il positivo effetto di evitare di allocare risorse per gestire i picchi e che poi resteranno inutilizzate per il resto del tempo. La FIGURA 4.7 mostra un esempio di una classe con un eccesso di risorse (area scura) quando il carico è inferiore alla media delle richieste. Queste risorse saranno utilizzate solo per gestire il picco.

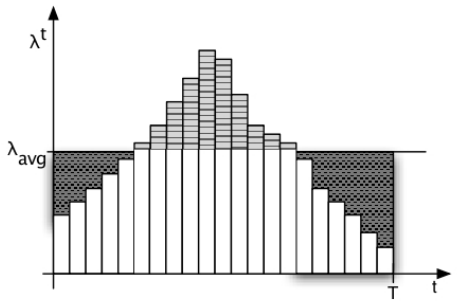

Figura 4.7: Spreco di risorse nell'allocazione di picco.

Il passo di inizializzazione (Passi da 1 a 3) dell'algoritmo di partizionamento delle classi applicative (Algoritmo 9) prevede la formazione di un numero di partizioni pari al numero delle applicazioni presenti nel Service Center. Per ciascuna coppia di partizioni (P_i, P_j) verrà calcolato l'Area Ratio (Passo 5) utilizzando la formula: $AR(P_i, P_j) := MAX(P_i \cup P_j)/AS(P_i \cup P_j)$. Tutte le partizioni così formate vengono inserite in una lista. Questa lista viene ordinata in maniera decrescente confrontando, partizione per partizione i valori di AR precedentemente calcolati (Passo 8). In questo modo le partizioni con Area Ratio più elevato, cioè quelle raggruppabili con maggior convenienza, si troveranno nella testa della lista. Al Passo 10, l'algoritmo seleziona sequenzialmente le partizioni. Se esse hanno un AR maggiore di una certa soglia τ (posta sperimentalmente pari a 0.5) le partizioni vengono collassate in una nuova partizione. La nuova partizione $P_i \cup P_j$ avrà come il profilo di traffico la somma, ora per ora, dei valori Λ_i dei due profili di traffico di partenza. Una volta resa attiva una partizione si scorre tutta la lista in cerca di altre partizioni contenenti quelle di partenza (P_i eP_j). Infatti quelle partizioni conterranno modi alternativi di formare i raggruppamenti che però saranno meno convenienti in quanto la lista è ordinata decrescentemente per AR.
L'algoritmo procederà applicando questa procedura iterativamente fino a raggiungere un livello di raggruppamento tale per cui il numero di partizioni formate sia quello desiderato. Il numero di gruppi viene scelto all'inizio dell'applicazione sulla base del numero massimo di classi che potranno far parte del gruppo. Quest'ultima dimensione è stata stabilita sperimentalmente. La complessità del logaritmo nel caso in cui ad ogni iterazione si riesca ad aggregare solo una partizione risulta essere pari a $O(n^3 log\, n)$.

```
   Data: K Insieme delle classi
   Result: Insieme delle partizioni of partitions P = {P_1, ..., P_N}
 1 for k ← 1 to |K| do
 2 |  P_k ← {k} ;
 3 end
 4 repeat
 5 |  foreach coppia (P_i, P_j) do
 6 |  |  clacolo di AR(P_i, P_j) ;
 7 |  |  inserimento P_n=P_i ∪ P_j nella lista delle partizioni ;
 8 |  end
 9 |  ordinamento per area ratio della lista di partizioni in modo
   |  decrescente ;
10 |  foreach pair (P_i, P_j) presa in ordine decrescente di AR do
11 |  |  if AR(P_i, P_j) > τ then
12 |  |  |  Attivazione della partizione : P_n = P_i ∪ P_j e calcolo del
   |  |  |  nuovo profilo di traffico ;
13 |  |  |  Calcellazione dalla lista tutte le partizioni contenenti P_i e
   |  |  |  P_j ;
14 |  |  end
15 |  end
16 until è ottenuto il numero desiderato di cluster;
```

Algoritmo 9: Algoritmo di partizionamento delle classi applicative.

Come già detto, l'ottimizzatore gerarchico tenta di assegnare le risorse ora per ora per tutta la giornata, la divisione in partizioni delle classi deve considerare almeno 24 previsioni di traffico (una per ora). La ripartizione delle classi è quindi un processo che viene eseguito una sola per ogni giorno.

4.2.2.3 Assegnamento dei server alle partizioni applicative

Dopo aver raggruppato l'insieme delle classi in gruppi, il CM deve completare l'assegnamento iniziale assegnando a ciascun gruppo un determinato numero di server. Restando nell'ottica di ottimizzare il sistema per l'ora successiva, questo processo dovrà essere in grado di associare un numero sufficiente di server ai gruppi per gestire il carico che si presenterà durante tutta la prossima ora.

Per fare questo è stato pensato un algoritmo greedy (Algoritmo 10). Durante l'inizializzazione i server vengono ordinati per costo/ capacità ($\frac{c_i}{C_i}$) utilizzando la procedura di ordinamento euristica (mostrata in Sezione 4.1.2.1) che calcola i costi dei server ipotizzando un utilizzo pari al 50% (Passo 1). Considerando sequenzialmente tutti i gruppi, verranno considerate le rispettive applicazioni. Successivamente, per ogni applicazione verrà valutato il relativo carico Λ_k di ciascun tier applicativo (Passo 5). Si tenterà, poi, di assegnare il

primo server compatibile nella lista al primo tier della prima applicazione della prima partizione (Passo 7). Se un solo server non bastasse se ne assegneranno altri in maniera tale da coprire tutto il carico elaborativo previsto per il tier. L'aggiornamento di traffico residuo (Passo 8) del tier k, j dopo l'assegnamento ad un server i viene calcolato come segue:

$$\lambda_{res} = \lambda_{res,old} - U_{max} \cdot C_i \cdot \mu_{k,j}$$

dove U_{max} è il massimo utilizzo dei server consentito.

La complessità dell'algoritmo risulta essere pari a $O(M \cdot logM + P \cdot M \cdot K \cdot \sum_{k=1}^{K} N_k)$, nel caso pessimo in cui venga selezionato in ciascuna iterazione l'ultimo server nella lista e nessun server sia saturo (bisogna sempre considerare la compatibilità del server con l'applicazione correntemente sotto allocazione). In particolare si devono considerare tutte le partizioni, tutti i server e tutti i tier applicativi inclusi.

Data: L Insieme dei server non assegnati, K Insieme delle classi, P Insieme delle partizioni
Result: Insieme dei server associato ad ogni partizione

```
1  Ordinamento della lista dei server non assegnati per c_i/C_i in modo
   decrescente ;
2  for p ∈ P do
3      for k ∈ K : k ∈ p do
4          for j ∈ J : j ∈ k do
5              λ_res ← Λ_k ;
6              while λ_res > 0 do
7                  Assegno alla partizione p il primo server disponibile
                   nella lista L e compatibile con k-j ;
8                  λ_res ← max {0, λ_res − U_max · C_i · μ_{k,j}} ;
9              end
10         end
11     end
12 end
```

Algoritmo 10: Algoritmo di assegnamento dei server alle partizioni applicative.

4.2.3 Implementazione del Central Manager

Come già discusso nelle sezioni precedenti, il Central Manager (CM) è il principale autore dell'ottimizzazione gerarchica poiché si occupa di gestire le risorse disponibili nel sistema. Gli obiettivi di questo algoritmo, implementato come procedura di ricerca locale, sono:

- assegnare le risorse prelevandole dal un pool condiviso
- permettere la migrazione di capacità elaborativa tra le partizioni.

Il primo obiettivo riguarda l'assegnamento di capacità elaborativa ai vari Application Manager. Le risorse vengono prelevate da un insieme di server condiviso tra le partizioni ed ordinato per c_i/C_i, denominato Free Server Pool. Per raggiungere lo scopo il Central Manager si avvale di due mosse: l'incremento e il decremento di capacità elaborativa (verranno discusse in seguito). Queste mosse provano ad aggiungere ed a rimuove capacità alle partizioni, registrando le variazioni di valore della funzione obiettivo. Successivamente all'applicazione di queste mosse ed al relativo calcolo del gradiente, il CM sarà in grado di aggiungere o rimuovere server ad ogni gruppo in modo da poter incrementare la funzione obiettivo globale.

Il secondo obiettivo del Central Manager è quello di permettere alle varie partizioni applicative di potersi scambiare una certa frazione della propria capacità elaborativa. A volte infatti può capitare che nell'assegnamento iniziale delle risorse vengano scelti tutti i server migliori, lasciando nel Free Server Pool solo i server più "lenti". Questa mossa ha il compito di verificare se spostando da una partizione ad un'altra (secondo le direzioni indicate dai gradienti) un certa quantità di capacità elaborativa può essere migliorato il valore della funzione obiettivo globale. Verranno quindi individuati delle partizioni che dovranno cedere delle risorse ed altre che saranno chiamate ad acquisirle. Per ciascuno scambio individuato si procederà ad applicare un'ulteriore procedura che effettivamente trasferirà i server secondo la direzione più vantaggiosa.

Il funzionamento dell'Algoritmo 11 è di tipo iterativo. L'inizializzazione della procedura prevede che ad inizio giornata, considerando le previsioni di carico, venga effettuo un partizionamento delle classi (Passo 1) che resterà valido per tutto il giorno (Algoritmo 9). Ad ogni ora l'algoritmo effettua un assegnamento dei server alle varie partizioni (Passo 3), prendendo in considerazione i profili di traffico per l'ora corrente (Algoritmo 10).

Dopo il passo di inizializzazione, per ogni partizione, vengono applicate le mosse di incremento (*incCapacity()*) della capacità elaborativa, registrandone la variazione delle funzione obiettivo in ciascuna partizione (Passo 7). Come già accennato questa tipologia di mosse preleva alcuni server da un Pool condiviso tra le partizioni (Free Server Pool) e li assegna a quest'ultime. Come vedremo in dettaglio successivamente la mossa di decremento (*decCapacity()*) della capacità elaborativa viene eseguita solo in presenza di una situazione critica (Passo 8) come ad esempio la scarsità di server nel Free Server Pool (Passo 9). Analizzando i valori di variazione della funzione obiettivo calcolando il relativo gradiente, il Central Manager cede o toglie risorse alle partizioni in modo da applicare le mosse effettivamente profittevoli (Passo 13). Questo procedimento

continua finché non è più conveniente aggiungere o togliere capacità elaborativa alle partizioni.

La seconda fase dell'algoritmo si occupa di spostare capacità elaborativa da una partizione all'altra. Ogni partizione libera (*giveCapacity()*) e riceve (*receiveCapacity()*) una certa quantità di capacità (Passi 18 e 20). Anche in questo caso il Central Manager registra le variazioni delle funzioni obiettivo rispetto all'incremento di capacità elaborativa, calcola il gradiente (Passi 19 e 21) ed in caso ci fossero opportunità di miglioramento applica la mossa (*migrateCapacity(fromPartition, toPartition)*) che permette di effettuare la migrazione di capacità secondo la direzione più opportuna (Passo 26). L'algoritmo continua finché non si raggiunge una convergenza e cioè non si hanno più variazioni significative della funzione obiettivo globale (Passo 28). Ad ogni passo questo meccanismo consente di incrementare la funzione obiettivo globale e quindi di avvicinarsi all'ottimo dell'algoritmo centralizzato. L'ultimo compito del Central Manager è quello di calcolare la funzione obiettivo globale e di tenere traccia di tutta la sequenza di mosse effettuate. Infatti come verrà esposto in seguito, per ragioni prestazionali ciascuna mossa cerca di stimare (attraverso bound) quale sarà il valore della funzione obiettivo dopo la sua applicazione. Il vero valore della funzione obiettivo di una partizione si saprà con esattezza solo dopo avere effettivamente applicato i cambiamenti nel sistema ed aver eseguito l'algoritmo AM utilizzando le mosse viste nel capitolo precedente (aumento della frequenza, riduzione della frequenza, ...). Infine, può capitare che dopo l'esecuzione di una mossa ci si trovi difronte ad una soluzione non accettabile. Per risolvere il problema è stato implementato un algoritmo in grado di incrementare il numero di server dell'AM unfeasible prelevandoli dal Free Server Pool. L'algoritmo tenta di capire il problema che genera l'errore (mancanza del rispetto dei vincoli di availability, numero non sufficiente di server nel sistema, ...) ed assegna nuove macchine, opportunamente scelte, alla partizione. Se il problema persiste dopo due tentativi il sistema "disfa" le mosse fatte e torna al passo precedente, dove verrà scelto di applicare un'altra mossa.

La complessità totale dell'algoritmo appena illustrata risulta essere pari a :

$$O(n^3 log\, n \cdot I_{max} T \cdot [\cdot M \cdot logM + P \cdot M \cdot K \cdot \textstyle\sum_{k=1}^{K} N_k \cdot \\ P \cdot max\,\{M, \rho\} \cdot I_{max,1} + \\ + P \cdot (max\,\{M \cdot logM, \rho\} + P^2) \cdot I_{max,2}])$$

Il calcolo della complessità tiene conto del partizionamento applicativo, dell'assegnamento dei server alle partizioni e della complessità dell'algoritmo utilizzato dagli AM (ρ). La mossa di $incCapacity()$ ha complessità pari ad $O(M)$, poiché nel caso pessimo si devono assegnare tutti i server presenti nel Free Server

Pool, la mossa di *decCapacity()* ha complessità $O(1)$, quella di *giveCapacity()* ha complessità $O(M \cdot logM)$,poiché utilizza un ordinamento, mentre quella di *receiveCapacity()* ha complessità pari a $O(1)$. Infine la mossa di *migrateServer()* ha capacità pari a $O(P^2)$.

```
   Data: P Insieme delle partizioni, T Insieme dei periodi temporali
   Result: Ottimo gerarchico
 1 partizionamento delle classi ;
 2 for t ∈ T do
 3 |   assegnamento dei server alle classi ;
 4 |   repeat
 5 |   |   repeat
 6 |   |   |   forall the p ∈ P do
 7 |   |   |   |   incCapacity() ;
 8 |   |   |   |   if sistuazione critica then
 9 |   |   |   |   |   decCapacity() ;
10 |   |   |   |   end
11 |   |   |   |   valutazione della variazione di funzione obiettivo della
   |   |   |   |   partizione p ;
12 |   |   |   end
13 |   |   |   attuazione delle mosse profittevoli ;
14 |   |   |   calcolo della variazione della funzione obiettivo globale e dei
   |   |   |   gradienti;
15 |   |   until valore della variazione della funzione obiettivo globale >0 o
   |   |   si raggiunge il limite massimo di iterazioni I_max,1;
16 |   |   repeat
17 |   |   |   forall the p ∈ P do
18 |   |   |   |   giveCapacity() ;
19 |   |   |   |   valutazione della variazione di funzione obiettivo della
   |   |   |   |   partizione p ;
20 |   |   |   |   receiveCapacity() ;
21 |   |   |   |   valutazione della variazione di funzione obiettivo della
   |   |   |   |   partizione p;
22 |   |   |   end
23 |   |   |   analisi delle variazioni della funzione obiettivo, calcolo dei
   |   |   |   gradienti e valutazione degli scambi di capacità elaborativa tra
   |   |   |   le partizioni ;
24 |   |   |   MIGRATESET ← (p_i, p_j) vantaggiose ;
25 |   |   |   forall the (p_i, p_j) ∈ MIGRATESET do
26 |   |   |   |   migrateServer(p_i, p_j) ;
27 |   |   |   end
28 |   |   until valore della variazione della funzione obiettivo globale >0 o
   |   |   si raggiunge il limite massimo di iterazioni I_max,2;
29 |   until non è raggiunta la convergenza o si raggiunge il limite massimo
   |   di iterazioni I_max;
30 end
```

Algoritmo 11: Algoritmo del Central Manager.

Mossa di incremento e decremento della capacità elaborativa

Durante la prima fase dell'algoritmo, il Central Manager tenta di incrementare la capacità elaborativa delle partizione (*incCapacity()*), prelevando dei server dal Free Server Pool. Precisamente si estraggono dal Free Server Pool un numero opportuno di server ed a turno si provano ad assegnare a ciascuna partizione, valutandone l'incremento della funzione obiettivo. Tali valori vengono inseriti in maniera decrescente in una lista. L'algoritmo procede nella scansione della lista applicando le mosse profittevoli. Poiché la quantità di capacità elaborativa da assegnare è fissata, il calcolo del gradiente si riduce ad una valutazione della variazione del valore della funzione obiettivo rispetto all'aumento della capacità elaborativa. Vengono considerate profittevoli le mosse con un incremento della funzione obiettivo superiore ad una soglia data. L'obiettivo di questa mossa è quello di assegnare, se vantaggioso, i server ancora liberi nel sistema. In questo senso il Free Server Pool svolge un compito fondamentale: tenere traccia dei server inutilizzati dagli AM ed ordinarli secondo il rapporto $\frac{c_i}{Ci}$ in modo da selezionare prima i server migliori.

La mossa che rimuove capacità elaborativa (*decCapacity()*) viene applicata solamente in circostanze critiche. Si supponga il caso in cui il Service Center disponga molti server disponibili e non ancora assegnati a nessuna partizione. In tale situazione sarebbe poco sensato rimuovere server dagli AM per lasciarli inutilizzati nel Free Server Pool. Viceversa, si supponga che una soluzione relativa ad una data partizione risulti unfeasible (non accettabile) dopo l'applicazione di alcune mosse. Se il Central Manager dispone di pochi server nel Free Server Pool, non sufficiente a far fronte alla situazione critica verificatasi, si chiede ad uno o più AM di rilasciare un certo numero di server. Ciascuna partizione destinata a rilasciare i server valuterà il suo insieme dei server spenti e non utilizzati in modo da non decrementare la propria funzione obiettivo e li renderà disponibili al CM che a sua volta provvederà ad assegnarli al'Application Manager critico.

Mossa di migrazione della capacità elaborativa

Nella seconda fase l'algoritmo del Central Manager permette lo scambio di capacità elaborativa tra i gruppi. In questa fase quindi non sarà più considerato il pool condiviso di risorse disponibili (Free Server Pool). Dovranno bensì essere individuati dei gruppi designati a rilasciare delle risorse ed altri destinati a riceverle. Il meccanismo implementato consta di due fasi che si occupano di trovare dei bound per poter stimare la variazione della funzione obiettivo di ciascuna partizione a seguito della mossa di migrazione. Nella prima fase il Central Manager si avvale di una mossa (*receiveCapacity()*) che consente di incrementare la capacità elaborativa di una partizione attraverso l'aggiunta di

server. Con lo scopo di dover trovare un bound all'incremento della funzione obiettivo si è deciso di porsi nel caso pessimo, ovvero nella situazione in cui l'incremento capacitivo avvenga aggiungendo alcuni server fittizi appartenenti alla peggiore tipologia ($\frac{c_i}{Ci}$) presente nel sistema, anche se la partizione non ne ha attualmente a disposizione. In tal modo si valuterà l'incremento del valore ottimo nel caso in cui si ricevano le risorse peggiori. Dopo aver applicato questa mossa si valuteranno i gradienti, espressi come variazioni di ciascuna funzione obbiettivo, verranno inseriti i corrispondenti valori in una lista ordinata in maniera decrescente per incremento del valore della funzione obiettivo.

La seconda fase si occupa invece di cedere un certo insieme di server (*giveCapacity()*). Nell'ottica di trovare un bound alla variazione della funzione obiettivo della mossa di migrazione, anche in questo caso ci si porrà nel caso peggiore: i server da rilasciare saranno i migliori presenti all'interno della partizione. Siccome i server migliori di ciascuna partizione possono avere capacità differenti, si dovrà tenere in considerazione la capacità effettivamente rilasciata. Ad esempio, su supponga di dover sottrarre ad un gruppo una capacità pari a 5 unità e si supponga che i server migliori dell'AM in considerazione abbiano capacità pari a 2. In tal caso verranno ceduti tre server con una quantità totale di 6 unità elaborative. Anche in questa mossa il Central Manager procederà poi a memorizzare in una lista i valori dei gradienti, questa volta calcolati come segue:

$$\frac{Variazione\,della\,funzione\,obiettivo}{Capacità\,ceduta}$$

La variazione della funzione obiettivo, in questa fase sarà al più pari a zero, nel caso fossero rimossi dei server spenti. La lista verrà poi ordinata attraverso tali valori.

Giunto a questo punto (Algoritmo 11 Passo 23) l'algoritmo del Control Manager è in grado di stabilire come far migrare la capacità elaborativa. Vengono considerate entrambe le liste contenenti i valori dei gradienti relativi alle due mosse appena illustrate. Dalla lista relativa alla mossa di *receiveCapacity()* viene selezionata la partizione corrispondente al valore del gradiente maggiore, mentre per la lista della mossa di *giveCapacity()* sarà selezionato il gradiente più prossimo a zero. Verrà quindi valutato se l'incremento delle funzione obiettivo dovuto alla mossa di *receiveCapacity()* sia strettamente maggiore al decremento dovuto alla mossa di *giveCapacity()*.

Essendoci posti nel caso peggiore in entrambe le simulazioni si riesce a valutare un lower bound di guadagno della funzione obiettivo della mossa di migrazione. Conseguentemente applicando tale mossa si otterrà un guadagno della funzione obiettivo maggiore, rispetto a quanto stimato. Se quindi tale variazione è positiva verrà applicata la mossa di migrazione e la capacità elaborativa sarà spostata dall'Application Manager associato al minor decremento alla par-

tizione corrispondente al maggior incremento. La mossa di *migrate* si occuperà, contrariamente a quanto fatto per ricavare i bound, di rimuovere i server peggiori dalla partizione destinata a cedere risorse e ad assegnarli alla partizione designata a riceverli.

Capitolo 5

Analisi sperimentali

In questo capitolo verranno descritte le prove sperimentali che sono state effettuate per testare la bontà dell'algoritmo gerarchico sviluppato nel Capitolo *4. In particolare nella* Sezione *5.1 verrà presentato l'algoritmo usato per generare istanze di test per Service Center di larga scala.*

Nella Sezione 5.2 *verranno illustrati dei test comparativi tra l'algoritmo gerarchico e quello centralizzato precedentemente sviluppato in [8]. Verranno effettuare molteplici prove dove verranno modificate le dimensioni delle istanze per stabilire la qualità della soluzione dell'algoritmo gerarchico rispetto a quello centralizzato. Nella* Sezione 5.3 *verrà compiuta un'analisi di scalabilità dell'algoritmo gerarchico e verranno illustrati i risultati di test su grosse istanze. Infine, nella* Sezione 5.4 *verranno fatti variare i parametri prestazionali in modo da verificare il funzionamento dell'algoritmo gerarchico per particolari casi di studio.*

5.1 Generatore di istanze di Service Center di larga scala

Lo scopo del sistema gerarchico è quello di rendere l'architettura del risolutore più flessibile e scalabile. Infatti, si vogliono poter gestire grossi Service Center contenenti fino a 2500 Virtual Machine. In [8] è stato implementato un algoritmo che permette di costruire in modo random istanze di Service Center. Tale generatore, a partire da un insieme di parametri, genera otto istanze identiche tra loro salvo che per i valori di carico. Questi infatti sono differenziati in modo che, assegnando il carico a tutti i server, il valore di utilizzo sia variabile tra il 20% ed il 90% con passo 10%. In altre parole, si avranno al termine della computazione otto livelli di carico, ognuno dei quali comporta un utilizzo dei server differente. Il carico viene stabilito risolvendo un'istanza del problema e scalando opportunamente i valori delle Λ_k di ciascuna classe applicativa. In tal modo, se si vogliono risolvere grosse istanze, la fase di risoluzione contenuta nella generazione del Service Center rende molto lunga tale procedura. Si è pensato allora di implementare un software che permettesse di accelerare la procedura di creazione dei test case. L'idea di base prevede l'utilizzo del generatore precedentemente sviluppato per la risoluzione di molte istanze di piccole dimensioni che andranno poi combinate in modo opportuno per generare il Service Center di larga scala. Questo processo permette di concludere in breve tempo la procedura di generazione poiché ogni singolo test case è generabile in breve tempo. Sarà poi un generatore supervisore che si occuperà di unire le descrizioni dei vari Service Center in un unico caso di test più ampio. La soluzione sviluppata funziona come segue (si faccia riferimento all'Algoritmo 12). Vengono dati in input un file contenente i parametri di ciascun sottosistema, le dimensioni del test case di larga scala ed un file contenente 24 valori valori percentuali che rappresentano un profilo di traffico standard. Verranno generati una serie di testcase di piccole dimensioni (Passo 3) che verranno uniti in un unico caso di test con il livello di utilizzo delle macchine approssimativamente attorno a 0.9 (Passo 5). Verranno poi, per ogni classe, generati due valori casuali che rappresentano rispettivamente un valore di shift (tra 0 e 23) (Passo 7) ed un valore di guadagno (tra -0,1 e + 0,1) (Passo 8), che verranno applicati ad un profilo di traffico bimodale standard (Figura 5.1).
I valori di traffico delle classi applicative saranno calcolati moltiplicando i valori di carico $\tilde{\Lambda_k}$ del caso di test, creato tramite l'unione dei test case iniziali, per il profilo di traffico generato randomicamente (Passo 12).

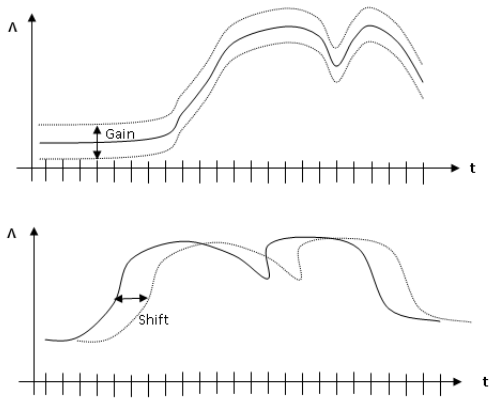

Figura 5.1: Modifiche ai profili di traffico.

Data: Parametri del Service Center da generare (tipologie di server, classi,..), File con un profilo di traffico standard (Λ_{Std}(t))
Result: Test case con ampio numero di server e applicazioni

1 Il Service Center viene scomposto in un insieme S di sottosistemi più piccoli ;
2 **for** $s \in S$ **do**
3 | Il sottosistema s viene generato dal generatore centralizzato, generando i valori di Λ_k in modo random per ottenere un utilizzo globale di 0.9 ;
4 **end**
5 I sottosistemi vengono fusi in un unico Service Center ;
6 **for** $k \in K$ *: insieme delle classi* **do**
7 | $shift_k \leftarrow$ generazione di un numero intero casuale tra [0,23] ;
8 | $noise_k \leftarrow$ generazione di un numero casuale tra [-0.1,0.1];
9 **end**
10 **for** $t \in T$ *: insieme degli intervalli temporali* **do**
11 | **for** $k \in K$ **do**
12 | | $\Lambda_k = \tilde{\Lambda_k}$ $[\Lambda_{Std}((t+shift_k)\ mod\ 24) \cdot (1+ noise_k)]$
13 | **end**
14 | Generazione del test case t-esimo con i nuovi valori di Λ_k ;
15 **end**

Algoritmo 12: Algoritmo di generazione di grandi casi di test.

Frequenze [MHz]	2600	2400	2200	2000	1800	1000
$b_{i,h}$	39.73	35.98	32.98	31.48	29.09	23.54
$\overline{c_i} + p_{i,h}$	60.2699	58.4708	57.5712	56.9715	56.5217	55.0225

Tabella 5.1: La tabella mostra per ogni frequenza relativa ad un p-state, il modello a rette utilizzato nei test.

5.1.1 Generazione delle istanze

Per valutare l'efficienza dell'algoritmo gerarchico si è proceduto a generare un gran numero di istanze di Service Center utilizzando il generatore appena descritto.

Durante la prima fase di test il numero dei server contenuto in ciascun test case è stato fatto variare tra 200 e 4800, mentre il numero di classi applicative è stato fatto variare tra 20 e 480. Sono stati considerate applicazioni operanti su 5 tier, in questo modo il numero di VM considerate è stato almeno pari a 480x5=2400 dato che per garantire l'availability alcuni tier vengono replicati. In questo modo è stato considerato un numero di applicativi abbastanza ampio da poter approssimare ragionevolmente Service Center reali. Per ogni VM si è ipotizzato un consumo di RAM generato casualmente nell'intervallo [1GB, 2GB], mentre per quanto riguarda le pendenze delle utility function sono state fatte variare tra 0.2 e 10 in modo da ottenere un'alta diversificazione dei ricavi tra le classi applicative. L'availability da garantire per ciascuna classe è stata fatta variare tra 0.95 e 0.999, mentre il parametro $\mu_{k,j}$ è stato generato casualmente nell'intervallo [1, 10]. Conseguentemente $\overline{R}_k$ è posto proporzionale al demanding time: $\overline{R}_k = 10 \cdot \sum_{j \in N_k} \frac{1}{\mu_{k,j}}$. Per quanto riguarda le caratteristiche dei server, sono state considerate macchine con 6 frequenze operative differenti ed altrettante rette che descrivono il modello energetico (descritto nella Sezione 4.1.2). I valori utilizzati per l'implementazione del modello energetico sono riportati in Tabella 5.1.

Il costo energetico orario è stato posto pari a 0.15 [$/kWh] ed è stato considerato un overhead dovuto al raffreddamento del Service Center pari allo 0.7. Ciò significa che per ogni dollaro speso in energia per i server sono stati considerati 0.7 dollari aggiuntivi per calcolare il costo totale tenendo conto anche dell'infrastruttura di climatizzazione. Infine, ciascun server possiede un quantitativo di memoria RAM pari a 32 GB ed una availability di 0.99999.

Diversamente a quanto fin qui esposto, nella seconda fase di test verranno modificati, uno alla volta, alcuni di questi parametri. In particolare per 5 classi applicative :

- il parametro Λ_k viene moltiplicato per 25 rispetto al Service Center di partenza preso come baseline;

(|K|, |M|)	Tempi di generazione [s]
(20,200)	8.65
(40,400)	24.52
(80,800)	78.14
(100,1000)	109.14
(120,1200)	155.78
(400,4000)	1726.32
(480,48000)	2678.53

Tabella 5.2: Tempi medi di generazione delle istanze.

- il valore di m_k viene moltiplicato per 100;
- il valore μ_k viene diviso per 10;
- il valore di availability viene posto pari a 99.999%.

L'obiettivo di questo insieme di test sarà quello di studiare il comportamento dell'algoritmo gerarchico rispetto alle 5 classi considerate.

I tempi medi di generazione delle istanze dell'algoritmo sono illustrati in TABELLA 5.2. La media è stata calcolato considerando la generazione di quattro istanze per ogni dimensione, tuttavia i dati sono caratterizzati da bassa varianza. Si mette in evidenza il fatto che il generatore di istanze presentato in [8] non è in grado di generare Service Center contenenti più di 800 server e 80 applicazioni. Non è stato quindi possibile effettuare valide comparazioni con l'algoritmo sviluppato in [8] sui tempi di generazione.

Come già illustrato, i profili di traffico giornalieri delle classi applicative vengono generati applicando ad un profilo standard uno Shift ed un Gain, generati casualmente. Il profilo di traffico standard (FIGURA 5.2) segue una distribuzione bimodale ispirata a delle tracce di log di un sistema Web di medie dimensioni utilizzato anche in [8], con due picchi: uno attorno alle 12 mentre l'altro alle 14.

Un esempio di profilo di traffico generato attraverso l'ALGORITMO 12 è mostrato in FIGURA 5.3. Come si nota l'andamento del traffico in ingresso al Service Center può essere molto variabile.

Il sistema utilizzato per la risoluzione di tutte le istanze sperimentali è basato su Ubuntu 10.4, con due CPU Intel Nehalem 2.4 GHz quad-core, con 24 GByte di memoria RAM.

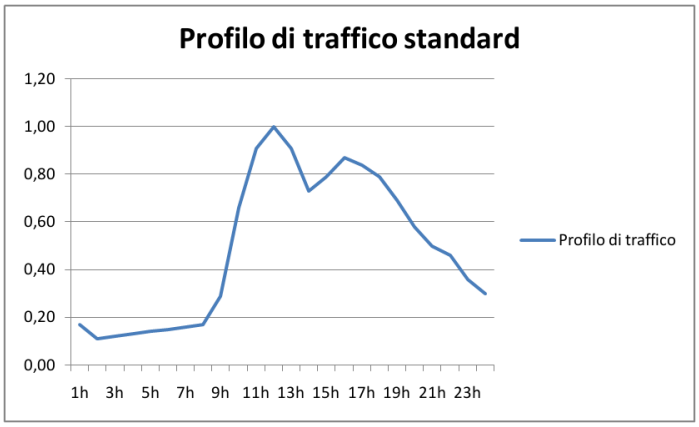

Figura 5.2: Profilo di traffico bimodale.

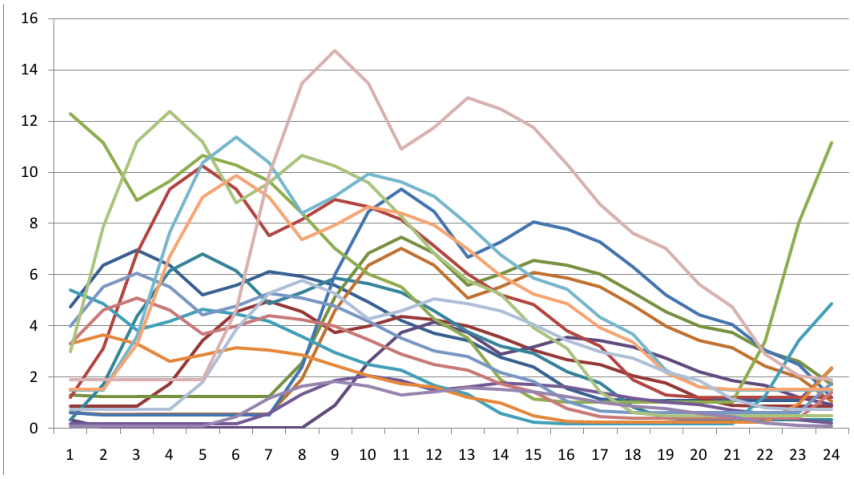

Figura 5.3: Profili di traffico delle classi.

5.2 Confronto tra algoritmo centralizzato e gerarchico

Il primo obiettivo che si è posto è stato quello di confrontare il funzionamento dell'algoritmo gerarchico presentato nella SEZIONE 4.2 rispetto a quello presentato nella SEZIONE 4.1che nel seguito verrà chiamato *algoritmo centralizzato*. Si vorranno valutare le performance in termini di tempo di risoluzione e la precisione dell'ottimo trovato. Il test verrà effettuato considerando sistemi composti da 200-400-800-1000-1200 server. Non si è potuti procedere oltre i 1200 server poiché il sistema centralizzato non è in grado di trovare una soluzione per Service Center di tali dimensioni. I parametri utilizzati nel sistema saranno quelli descritti nella sezione precedente. Per ogni istanza verranno considerate partizioni applicative con dimensione variabile, per studiare l'impatto di tale parametro sull'algoritmo. In particolare si sono formati partizioni aventi come massimo numero di classi pari a 10, 20 e 30.

Verranno riportati di seguito i risultati computazionali delle prove effettuate. Nei grafici sono riportati alcuni casi rappresentativi.

Test con 200 server e 20 applicazioni

I test di questi dimensioni risultano agevolmente risolvibile con l'algoritmo centralizzato. Nonostante ciò si è proceduti a studiare il comportamento del modello gerarchico.
La FIGURA 5.4 riporta il valore della funzione obiettivo e il tempo di calcolo dei due sistemi. Vengono rappresentati i risultati del modello gerarchico con partizioni da 10, 20 e 30 classi ed i valori relativi all'algoritmo centralizzato. Nella parte alta della figura vengono rappresentati i valori assunti dalla funzione obiettivo per le 24 della giornata, mentre nella parte bassa sono indicati i tempi utilizzati per risolvere le istanze. Come si può notare, il sistema centralizzato tende ad assumere valori maggiori della funzione obiettivo. Questo comportamento è dovuto al fatto che l'algoritmo centralizzato ha più maggiori gradi di libertà nell'ottimizzazione, che vengono correttamente sfruttati. Viceversa l'algoritmo gerarchico, lavorando su partizioni, ha una struttura più rigida che permette di utilizzare un dominio di ottimizzazione meno ampio. Si osserva inoltre che le istanze di queste dimensioni vengono risolte in minor tempo dall'algoritmo centralizzato. Questo avviene poiché l'algoritmo gerarchico è affetto da un overhead dovuto ad alcune azioni preliminari come ad esempio la suddivisione del sistema in partizioni. Si noti peraltro che il tempo di risoluzione massimo è poco maggiore ai 10 secondi.
Infine, confrontando i risultati ottenuti variando la dimensione delle partizioni si può affermare che tale parametro ha basso impatto sui valori della soluzione trovata e sul tempo di risoluzione.

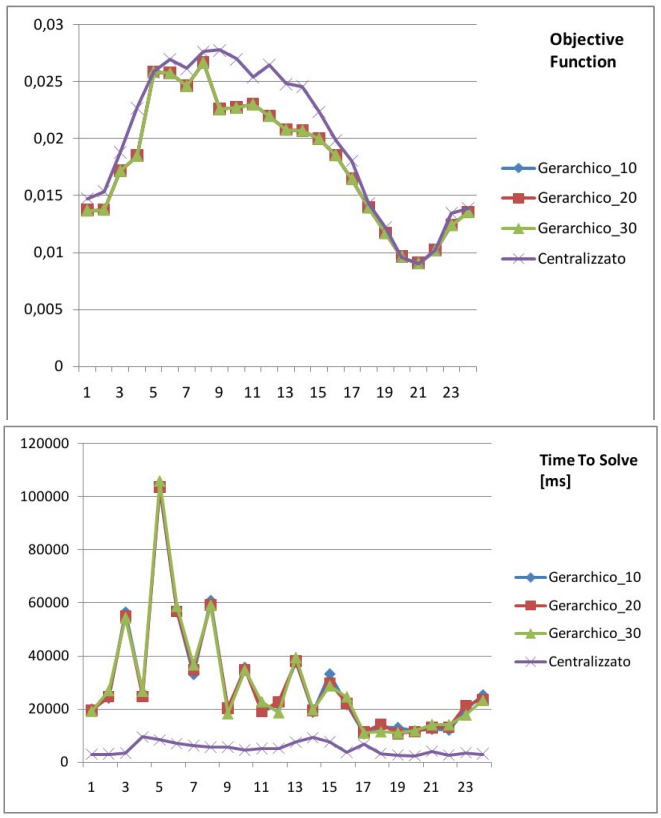

Figura 5.4: Risultati con un Service Center di 200 server e 20 applicazioni. In alto, i valori assunti dalla funzione obiettivo; in basso, i tempi di risoluzione.

Test con 400 server e 40 applicazioni

Anche questo caso, come nei test precedenti, istanze di queste dimensioni vengono risolte abbastanza agevolmente tramite l'algoritmo centralizzato. Infatti, anche in questa situazione i tempi del solutore centralizzato non sono affetti da tutti gli overhead di inizializzazione dell'algoritmo gerarchico. Sebbene l'algoritmo centralizzato è ancora più performante di quello gerarchico, con la FIGURA 5.5, si vuole mettere in evidenza una problematica abbastanza frequente dell'algoritmo centralizzato per risolvere istanze di questo tipo: come si può notare tra la seconda e la quarta ora e tra la quindicesima e la diciassettesima l'algoritmo centralizzato è affetto da soluzione unfeasible (rappresentate dai punti rossi), mentre l'algoritmo gerarchico riesce comunque a trovare una soluzione. Questo comportamento è dovuto al partizionamento. Come descritto nel capitolo precedente il partizionamento tende ad aggregare un insieme di classi applicative i cui i picchi del profilo di traffico dell'una corrispondono alle valle dalle

altre. Questo processo ha la capacità quindi di ottenere un profilo di traffico omogeneo, senza picchi improvvisi, per ciascuna classe. Un tale sistema è di più agevole risoluzione e la sua complessità di elaborazione risulta minore.

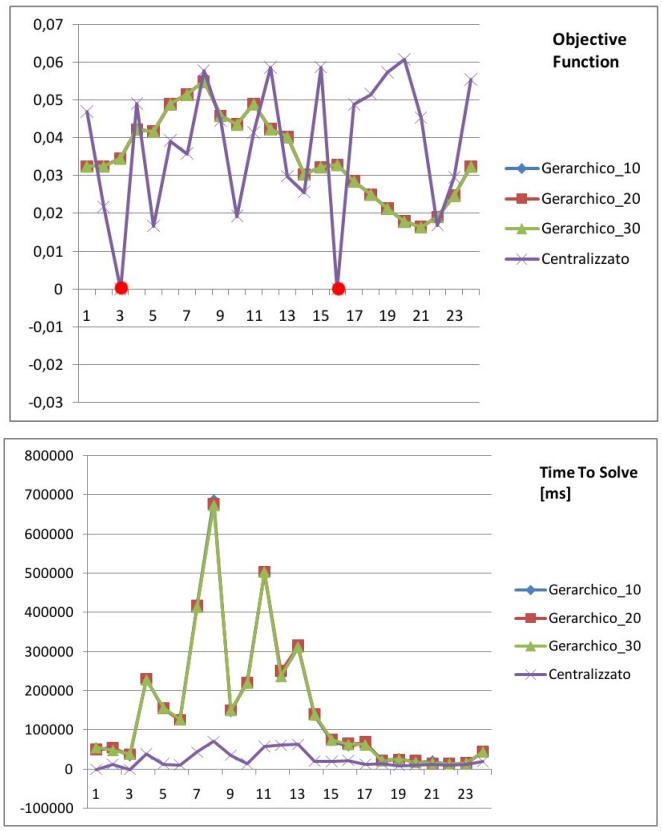

Figura 5.5: Risultati con un Service Center di 400 server e 40 applicazioni. In alto, i valori assunti dalla funzione obiettivo; in basso, i tempi di risoluzione.

Test con 800 server e 80 applicazioni

In questa serie di test si nota come l'algoritmo centralizzato abbia una perdita di performance notevole rispetto al modello gerarchico. I tempi di overhead di inizializzazione dell'algoritmo gerarchico non sono più così preponderanti e permettono di risolvere le istanze in modo efficiente. Dall'analisi dei dati raccolti si è visto come in questo scenario si verifichi uno speedup pari al 445%. Tuttavia, risulta esserci ancora una discrepanza sulla funzione obiettivo dei due modelli. L'errore percentuale medio risulta essere pari al 10.81%. In FIGURA 5.2 è mostrato un esempio che illustra i tipici comportamenti dei modelli per questa classe di istanze.

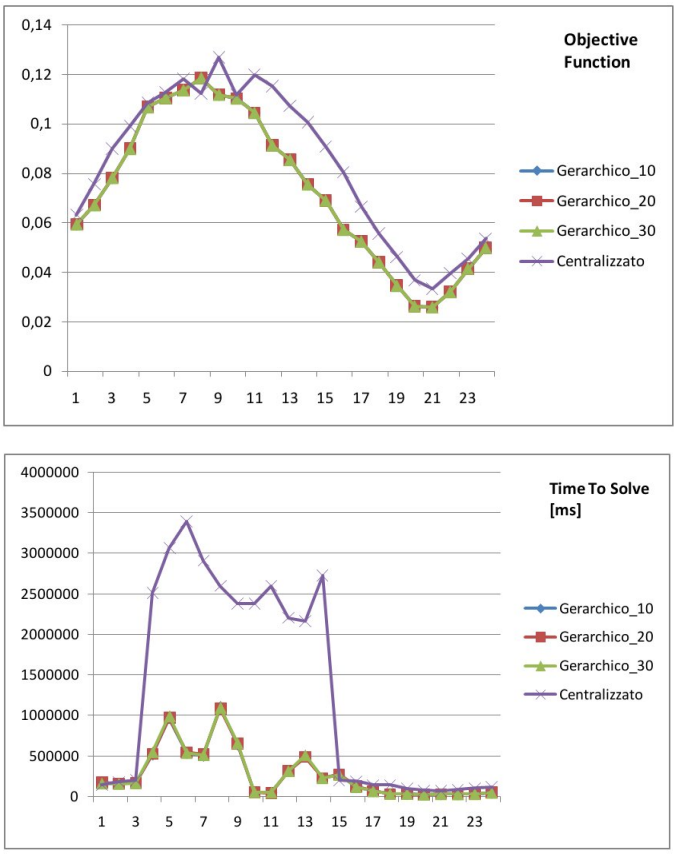

Figura 5.6: Risultati con un Service Center di 800 server e 80 applicazioni. In alto, i valori assunti dalla funzione obiettivo; in basso, i tempi di risoluzione.

Test con 1000 server e 100 applicazioni

Con queste dimensioni i test i tempi di risoluzione dell'algoritmo centralizzato sono molto elevati. Infatti, come si nota dalla FIGURA 5.2 si hanno picchi con valori intorno alle due ora e mezzo. Per risolvere l'intera istanza, in media occorrono 16 ore, ma alcuni test case hanno impiegato più di un giorno per essere risolti. Tuttavia il modello gerarchico ben si comporta in questi test risolvendo le istanze in breve tempo. Dai dati si ottiene uno speedup del 697% ed un errore percentuale medio del 9.98%.

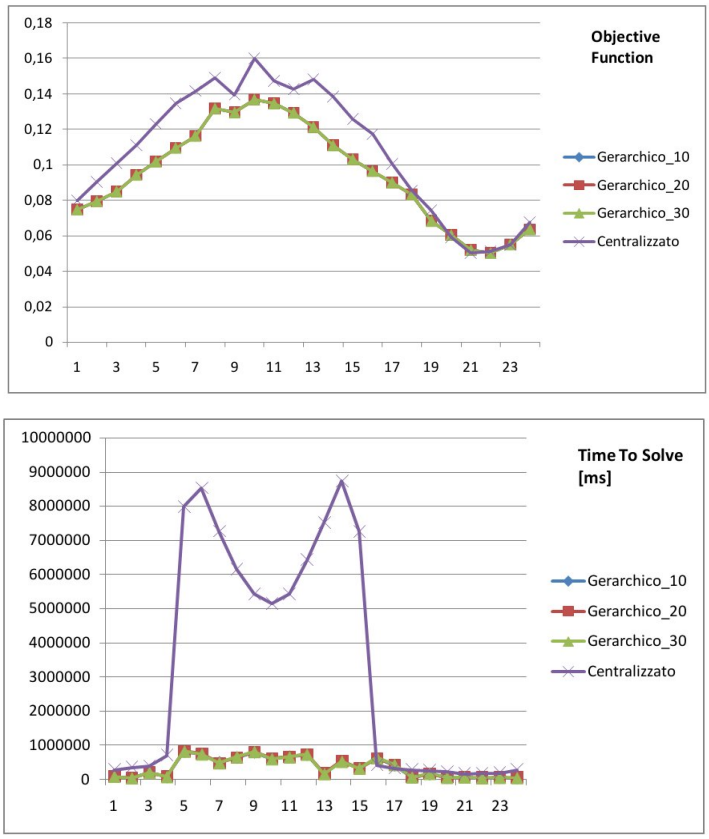

Figura 5.7: Risultati con un Service Center di 100 server e 100 applicazioni. In alto, i valori assunti dalla funzione obiettivo; in basso, i tempi di risoluzione.

Test con 1200 server e 120 applicazioni

Questa serie di test rappresenta il limite risolvibile con il sistema centralizzato. Con istanze maggiori, il problema di allocazione delle risorse tramite l'algoritmo centralizzato non può essere risolto in quanto è caratterizzato da un enorme allocazione in RAM e necessita di una grande potenza di calcolo. Come si può notare dalla FIGURA 5.2, i valori della funzione obiettivo dei modelli gerarchici si accostano maggiormente all'algoritmo centralizzato. Dalle analisi dei dati infatti si ottiene un errore percentuale medio del 7.75%. I tempi di calcolo sono nettamente a favore dell'algoritmo gerarchico, con uno speedup pari al 761%.

Dalle precedenti sessioni di test abbiamo riscontrato che la dimensione massima delle classi applicative contenute in ciascuna partizione era quasi ininfluente

sia rispetto ai tempi di elaborazione, sia rispetto ai valori assunti dalla funzione obiettivo. Si è voluto quindi testare limiti dimensionali differenti, passando a considerare partizioni contenenti rispettivamente 20 o 50 classi applicative. Come si può notare dalla FIGURA 5.2, anche in questo caso non sembrano esserci variazioni significative. Dai dati emerge come si abbiano variazioni di valore della funzione obiettivo alla quarta cifra decimale, quindi del tutto trascurabili.

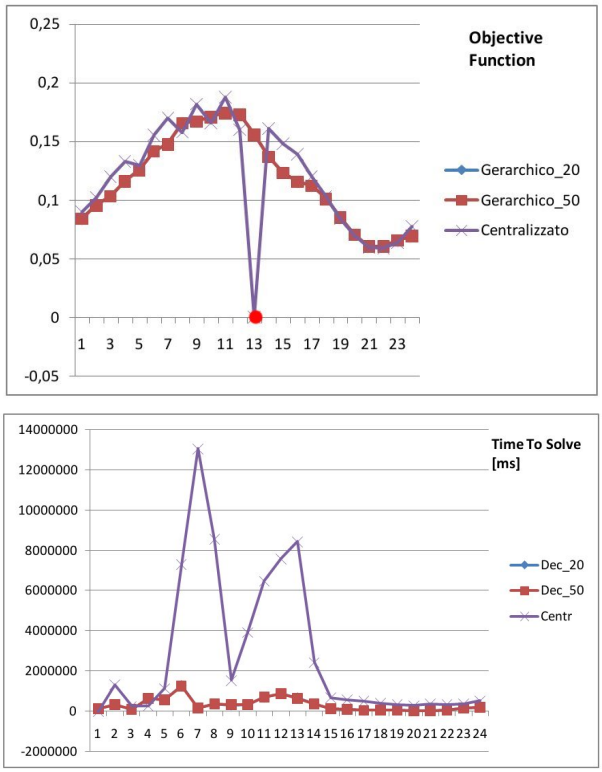

Figura 5.8: Risultati con un Service Center di 1200 server e 120 applicazioni. In alto, i valori assunti dalla funzione obiettivo; in basso, i tempi di risoluzione.

Conclusioni

In questa sequenza di test si è mostrato il confronto tra l'algoritmo gerarchico e quello centralizzato. Si può affermare che il modello gerarchico introduce degli overhead temporali di inizializzazione che possono essere trascurati solo per la risoluzione di istanze contenenti un numero maggiore di 400 server e 40 classi

applicative. Si è notato anche come la dimensione delle partizioni abbia un basso impatto sull'algoritmo e come tale parametro non influisca significativamente sui valori della funzione obiettivo trovati. Questo sta a dimostrare la robustezza dell'algoritmo implementato rispetto alla variazione di questo parametro.

In TABELLA 5.3 vengono mostrati gli speedup[1] e gli errori percentuali medi[2] ottenuti con l'algoritmo gerarchico. Osservando i dati si può notare come non solo si sia raggiunto lo speedup teorico (su di un processore con 8 CPU e nel caso di completa parallelizzazione dell'algoritmo sviluppato in [8]) ma in alcuni casi è stato anche possibile migliorare la qualità della soluzione ottenendo un valore maggiore della funzione obiettivo. L'algoritmo sviluppato nei precedenti lavori di tesi infatti, spendeva molto tempo in mosse di fine tunning. I risultati dimostrano come lavorando a "grana più grossa" considerando solo le mosse di accensione e spegnimento dei server e con l'ausilio di meccanismi di partizionamento e ottimizzazione tra le partizioni si arrivino ad ottenere risultati superiori alle aspettative. Infine, si può notare come l'errore percentuale rispetto alla funzione obiettivo dell'algoritmo centralizzato si riduca all'aumentare delle dimensioni delle istanze. Con molte classi e molti server il partizionamento rende efficienti i meccanismi di ottimizzazione implementati.

(|K|,|M|)	Errore Percentuale Medio	Speedup Medio
(20,20)	11.00 %	24%
(40,400)	11.00 %	49%
(80,800)	10.81%	445 %
(100,1000)	9.98%	697 %
(120,1200)	7.75%	761%

Tabella 5.3: Valori di errore percentuale medio e speedup del sistema gerarchico per le diverse dimensioni considerate.

5.3 Analisi di scalabilità

Come accennato i Service Center odierni stanno assumendo sempre maggiori dimensioni ed il consumo energetico di tale strutture non può più essere trascurato. Prendendo come riferimento il Service Center di Microsoft situato a Quincy, esso è in grado di ospitare quasi 10000 server e che ci si aspetta supporti un numero ancora maggiore di Virtual Machine attive contemporaneamente. Il consumo energetico stimato è attorno ai 54W/piede [17]. Per far fronte a le grandi dimensioni di questi Service Center occorre un algoritmo che permetta in

[1] Lo speedup è calcolato come: $\frac{tempo\,risoluzione\,centralizzato}{tempo\,risoluzione\,gerarchico}$

[2] L'errore percentuale è calcolato come: $\frac{|funzione\,obiettivo\,centralizzato - funzione\,obiettivo\,gerarchico|}{funzione\,obiettivo\,centralizzato}$

Tempo Totale per la risoluzione delle 24 ore [h]	Tempo Medio di risoluzione della singola ora [m]
10.44	26.10
9.73	24.32
12.25	30.62
13.08	32.69
7.59	18.99

Tabella 5.4: Insieme di prove sperimentali per rilevare i tempi d'esecuzione di istanze con 4800 e 480 classi applicative.

tempo utile di risolvere grosse istanze e che permettano di calcolare l'allocazione ottima delle risorse per l'ora successiva e per tutta la giornata.

Con questo insieme di prove sperimentali si è voluto verificare l'efficienza dell'algoritmo gerarchico nel risolvere istanze che si avvicinano a Service Center reali. Le variabili che più incidono sui tempi di esecuzione sono: il numero di classi e il livello di utilizzo del Service Center. Intuitivamente si può affermare che più è presente del traffico da allocare, più tempo impiegherà l'algoritmo per risolvere l'allocazione. Da questa considerazione si è voluto risolvere istanze ad alto utilizzo (più del 90%) e con un numero di VM pari ad almeno 2400. Conseguentemente sono state presi in considerazione Service Center contenenti 480 classi applicative e 4800 server. Per l'esecuzione dei test si è scelto di utilizzare una dimensione massima delle partizioni pari a 20 classi applicative, poiché esso rappresenta una via intermedia tra i valori discussi nella sezione precedente.

Risolvendo le istanze si è rilevato un tempo medio di esecuzione di circa 10 ore e 30 minuti. Ciascuna istanza oraria è stata risolta in media in 26 minuti, con un picco massimo di 62 minuti (per alcuni esempi si veda la Tabella 5.4). Questi dati confermano l'efficienza del modello preso in esame in quanto risolvendo in sequenza le varie istanze della giornata è possibile sempre calcolare con l'allocazione ottima per l'ora successiva. In particolare, avendo a disposizione le previsioni di carico, nelle ore di punta è sempre possibile iniziare l'elaborazione in anticipo in modo da terminare entro l'ora o, più semplicemente, adottare un elaboratore leggermente più potente di quello usato per questi test.

5.4 Casi di studio

In questa serie di test si è voluto verificare la sensibilità dell'algoritmo gerarchico rispetto alla variazione di alcuni parametri che caratterizzano le classi applicative. Verranno considerate delle istanze di Service Center contenenti 2000 server

e 200 classi applicative, per un numero minimo di Virtual Machine pari a 1000. Da tali istanze verranno modificati per alcune applicazioni i valori:

- del parametro Λ_k, moltiplicato per 25 rispetto al Service Center di partenza;
- di μ_k, diviso per 10 volte;
- di m_k, moltiplicato per 100;
- del parametro $\overline{A_k}$, impostato a 99.999%.

Nelle sezioni successive verranno confrontati i risultati ottenuti nelle istanze *originali*, considerate come baseline, e quelli ottenuti dalla modifica dei parametri.

Variazione del valore di Λ_k

In questa sessione di test si è incrementato di 25 volte il valore di Λ_k delle 5 classi applicative che nel Service Center erano caratterizzate dal maggior carico di picco. In questo modo si è potuto ottenere cinque classi applicative che ora per ora possiedono un carico significativamente maggiore delle altre. Attraverso questo test si vuole verificare il funzionamento del meccanismo di partizionamento e le variazioni sulla funzione obiettivo.

Osservando i risultati, ci si potrebbe aspettare che il processo di partizionamento tenda a non aggregare nella stessa partizione le 5 classi considerate. Infatti, esse hanno un un valore di traffico più alto rispetto alle rimanenti. Tuttavia, può capitare che queste classi applicative vengano inserite nella stessa partizione. Seguendo passo passo l'algoritmo si nota come l'inserimento nella partizione di una classe caratterizzate da un elevato valore di traffico avviene nelle fasi finali, ossia quando il profilo di traffico dell'intera partizione è ormai già abbastanza elevato. Per chiarire meglio questo concetto si osservi la FIGURA 5.9. Il grafico rappresenta l'ultima partizione del Service Center finora creata. La partizione contiene già la classe 1, ad alto carico, e si deve decidere se inserire anche la classe 3 che ha anch'essa un valore Λ_k elevato. Come si nota il profilo di traffico della partizione è abbastanza alto è quindi corretto inserire anche la classe 3. Come si vede dal grafico la curva del traffico totale aumenta, ma la variazione attorno al picco è più "dolce", sfruttando il picco e la valle, esistenti prima dell'integrazione, nell'intervallo temporale 9 -11.

Analizzando i test eseguiti si nota come i valori delle funzioni obiettivo delle classi modificate siano più alti rispetto a quelle dei Service Center originali. Questo comportamento è dovuto al fatto che in ciascuna partizione sono presenti abbastanza server per consentire un'elaborazione efficiente delle richieste, quindi il maggior traffico offre possibilità di guadagni più elevati. (FIGURA

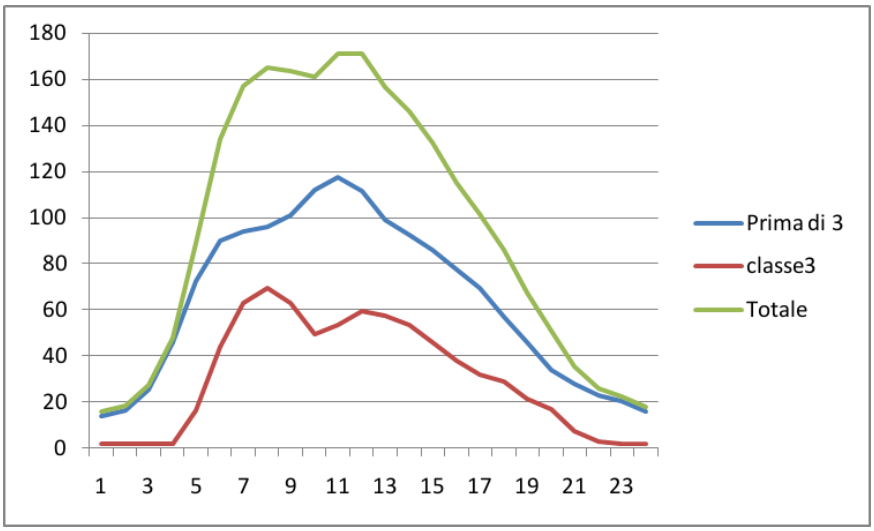

Figura 5.9: Grafico della previsione di traffico di una partizione applicativa. In questa partizione vengono inserite due partizioni (1 e 3) con valori di Λ_k elevati. Questo grafico rappresenta la situazione dell'Algoritmo 9 del Capitolo 4 nel il momento in cui si deve decidere se includere la classe 3. L'Area Ratio tra le curve è di circa 0.65.

5.10). Infine, si può notare come l'algoritmo consideri una quantità maggiore di capacità elaborativa da dare assegnare agli Application Manager che devono gestire le partizioni ad alto carico: rispetto alle istanze originali del problema iniziale si ha un incremento mediamente pari al 12%.

Variazione del valore di μ_k

Per ogni istanza presa in esame si sono considerate le 5 classi con il minor valore di μ_k(che corrispondono quindi alle classi più onerose dal punto di vista prestazionale). Si è poi provveduto a dividere per 10 il valore di tale parametro. Abbassando il valore di μ_k si aumentano i Demanding Time delle richieste che quindi diventano (che corrispondono quindi alle classi più onerose dal punto di vista prestazionale) CPU intensive, di conseguenza sarà necessario istanziare una maggiore capacità elaborativa alle partizioni contenenti queste classi. Infatti, dai risultati emerge come la variazione capacitiva per le partizioni considerate oscilli tra il 9% e il 12% e come la capacità totale a disposizione degli AM (Figura 5.11) venga incrementata mediamente del 7%.

Nei test effettuati la funzione obiettivo dopo la modifica dei parametri μ_k a subito un leggero decremento, stimato attorno al 4%, ciò è dovuto all'aumento dei costi per l'elaborazione delle classi applicative modificate (Figura 5.12).

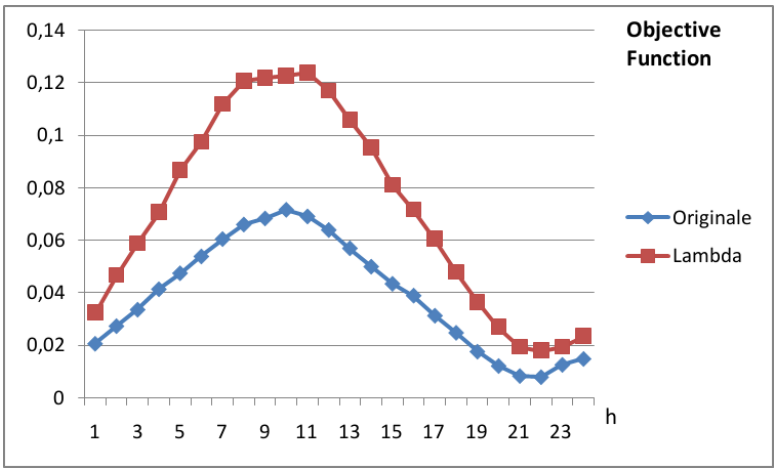

Figura 5.10: Variazione, ora per ora, della funzione obiettivo dell'istanza originale e di quella contenente classi ad alto carico.

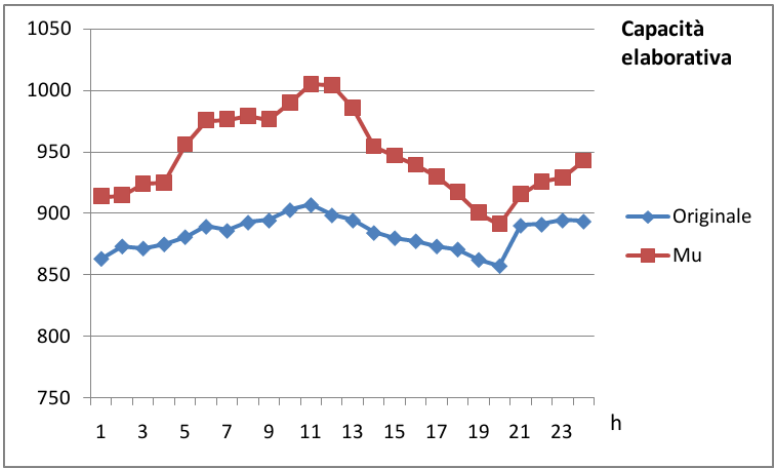

Figura 5.11: Variazione, ora per ora, della capacità elaborativa totale a disposizione degli AM dell'istanza originale e di quella contenente classi con basso μ_k.

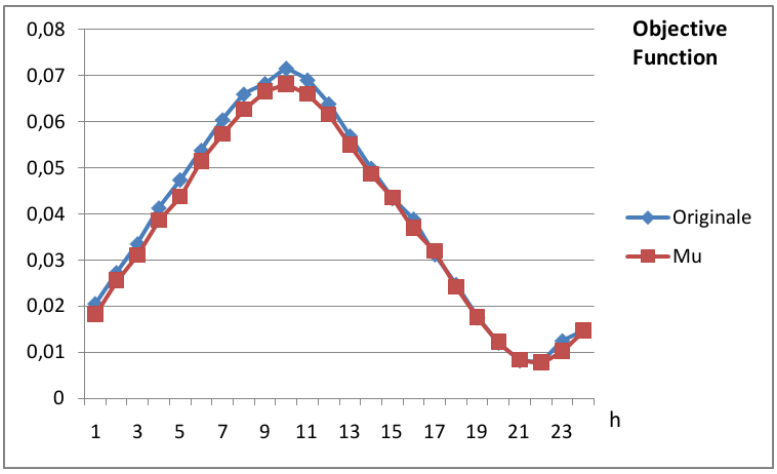

Figura 5.12: Variazione, ora per ora, della funzione obiettivo dell'istanza originale e di quella contenente classi con basso μ_k.

Variazione del valore di m_k

Con questa analisi si è voluto modificare il valore del coefficiente angolare della retta che descrive il contratto SLA (si faccia riferimento alla SEZIONE 2.3) per le cinque classi con i valori di m_k maggiori che sono stati moltiplicati per 100. La variazione di tale parametro permette di aumentare la sensibilità delle classi ai tempi di risposta. Volendo mantenere costante la soglia che separa la regione di profitto da quella di penale (z_k) è stato modificato anche il valore di v_k, incrementando anch'esso di un fattore 100 (FIGURA 5.13).

Osservando i risultati ottenuti dalle prove sperimentali si nota (Figura 5.14) come la variazione di capacità elaborativa a disposizione degli Application Manager rispetto all'istanza originale sia rimasta limitata (Figura 5.15) ed è stato anche possibile ottenere un incremento deciso della funzione obiettivo (+34 %).

Variazione del valore di availability $\overline{A_k}$

In questa fase, si è voluto incrementare per le prime 10 classi il valore di $\overline{A_k}$ ponendolo pari a 99,999%, in modo da raggiungere un livello di availability pari a *five nine*. L'availability dei server fisici disponibili è stata impostata a 99,999%. L'unico modo per il sistema di garantire un elevato livello di availability è quello di porre un determinato tier applicativo su più server in parallelo. Infatti, l'availability di elaboratori in parallelo, assumendo indipendenza tra le failure dei singoli server, è data da:

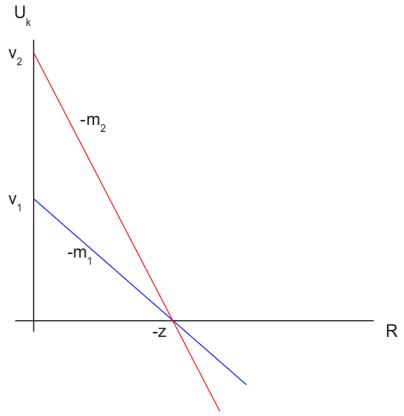

Figura 5.13: Esempio di variazione della pendenza m_k. La retta 1 corrisponde all'istanza originale, mentre la 2 è stata ricavata raddoppiando il coefficiente angolare e il valore dell'intercetta sull'asse U_k.

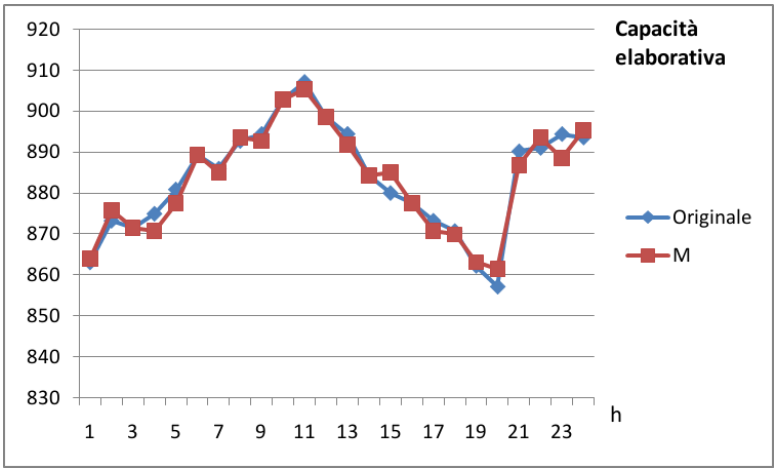

Figura 5.14: Variazione, ora per ora, della capacità elaborativa totale a disposizione degli AM dell'istanza originale e di quella contenente classi con alto valore di m_k.

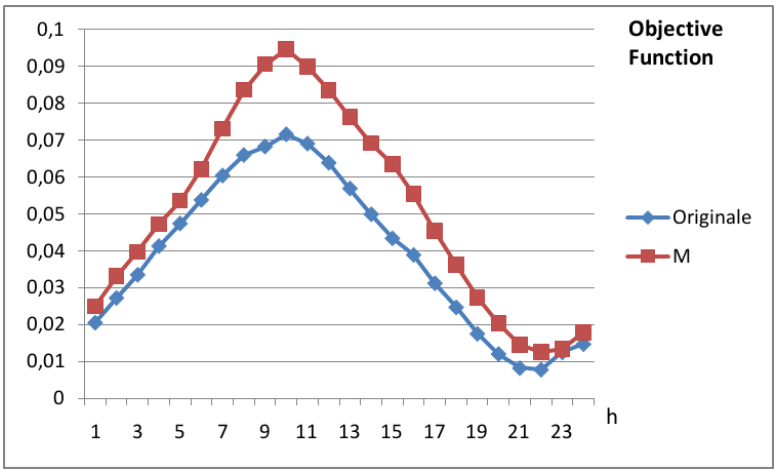

Figura 5.15: Variazione, ora per ora, della funzione obiettivo dell'istanza originale e di quella contenente classi con alto m_k.

$$A_s = 1 - \prod_{i=1}^{n} [1 - A_i]$$

dove A_i è l'availability del singolo server mentre A_s è l'availability di un tier istanziato su n macchine fisiche poste in parallelo. In questo modo è possibile ottenere un livello di availability anche superiore a quella del singolo componente.

Osservando i risultati riscontrati nella prove sperimentali si nota come la capacità totale a disposizione degli AM sia più bassa nelle istanze modificate rispetto alle istante originali (FIGURA 5.16). Questo comportamento è dovuto all'algoritmo di assegnazione iniziale dei server alle partizioni. Le classi applicative vengono ordinate in maniera decrescente per numero di vincoli $\mathcal{B}$ ed in maniera crescente per valori di $\Gamma_{k,j}$. Questo ordinamento non tiene in considerazione l'availability richiesta dalle classi applicative e considera prima altre classi rispetto a quelle ad alto livello di disponibilità. Quando vengono selezionati i tier ad alta disponibilità può succedere che siano rimasti solo i server peggiori e che non possono garantire il livello di availability richiesto. Tuttavia l'algoritmo procede nell'assegnamento iniziale, non curandosi di questo problema. Successivamente gli algoritmi di ottimizzazione utilizzati dagli Application Manager operano su partizioni con meno server rispetto all'istanza originale. Tuttavia, dato il basso utilizzo dei Service Center considerati, gli AM riescono comunque a trovare una soluzione ammissibile. Indubbiamente l'algoritmo di assegnamento iniziale dei server alle partizione può essere migliorato e sarà oggetto di una futura direzione di ricerca.

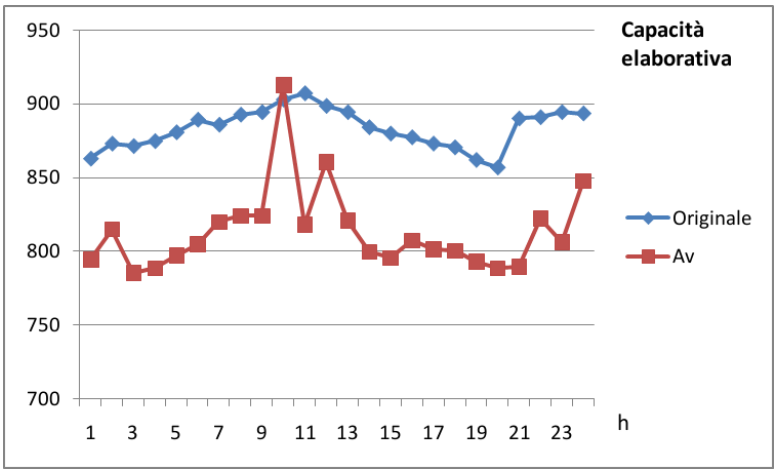

Figura 5.16: Variazione, ora per ora, della capacità elaborativa totale a disposizione degli AM dell'istanza originale e di quella contenente classi con alto valore di $\overline{A_k}$.

Capitolo 6

Direzioni future di ricerca e conclusioni

Questo lavoro di tesi si poneva l'obiettivo di permette la risoluzione del problema di allocazione delle risorse per Service Center di larga scala. I Service Center odierni infatti sono composti da migliaia di server e migliaia di classi applicative. Le tecniche precedentemente studiate in [14, 8] e quelle attualmente presenti in letteratura non consentono una gestione efficacie per sistemi di queste dimensioni.

In una prima fase della tesi si è proceduto ad estendere il modello di ottimizzazione 3.7.2. Le modiche implementate hanno riguardato due aspetti: l'aggiunta di un vincolo per la RAM e l'implementazione di un nuovo modello per il calcolo dei consumi energetici dei server. Il meccanismo per la gestione della memoria RAM è nato dalla necessità di voler meglio adeguare il modello a sistemi reali. Nel modello implementato in [14, 8] infatti era possibile allocare infinite Macchine Virtuali su di un server senza considerare opportunamente lo spazio di memoria occupato da ciascuna VM. Successivamente si è implementato un modello energetico che permette di mettere in relazione i consumi non solo con la frequenza operativa correntemente utilizzata da un server, ma anche con il livello di utilizzo dello stesso. Attraverso questo nuovo modello energetico è stato possibile calcolare in maniera più precisa e realistica i costi energetici dei Service Center.

Nella seconda fase della tesi si è progettato un algoritmo gerarchico che permettesse di risolvere agevolmente il problema di Resource Allocation su larga scala. L'idea di base di questo algoritmo è quella di suddividere gerarchicamente in partizioni l'intero Service Center. In particolare è stato progettato un algoritmo di partizionamento delle classi applicative basato su previsioni di carico estese ad una intera giornata. Per ciascuna di queste partizioni l'algoritmo si

occupa di assegnare un insieme di server e di risolvere, attraverso l'algoritmo esteso nella prima fase della tesi, il problema dell'allocazione delle risorse per ogni ora. L'ottimizzazione gerarchica viene eseguita attraverso una procedura di Local Search che prevede l'utilizzo di mosse che permettono ad ogni partizione di cedere, prendere o scambiare capacità elaborativa.

A seguito dell'implementazione si sono svolte delle analisi sperimentali per valutare il comportamento e l'efficienza dell'algoritmo. Confrontando i risultati ottenuti con quelli dell'algoritmo non gerarchico emerge come per grosse istanze di Service Center si ottenga uno speedup lineare. Questo dato è molto buono poiché risulta essere pari allo speedup teorico che può essere raggiunto eseguendo il codice sviluppato in [14, 8] su una macchina parallela. Questo speedup consente di risolvere grosse istanze in tempi comparabili con il periodo di controllo considerato nei moderni Service Center. Oltre all'incremento prestazionale si è verificato come l'errore percentuale medio si riduca con l'aumentare delle dimensioni del problema riuscendo inoltre ad individuare una soluzione ammissibile quando viceversa la versione precedente fallisce. La riduzione dell'errore mostra come l'efficacia dei meccanismi di ottimizzazione gerarchica sia maggiore su grandi istanze di Service Center. Nella parte conclusiva della fase sperimentale si è cercato di osservare i comportamenti dell'algoritmo in condizioni particolari, ottenute variando per un sottoinsieme delle classi applicative alcuni parametri prestazionali. Infine, si è cercato di valutare quanto la dimensione delle partizioni influisse sul valore della soluzione e si è verificato come la soluzione proposta risulti robusta alla variazione di questo parametro.

Le possibili direzioni di ricerca riguardano lo sviluppo di un sistema gerarchico a più livelli in cui per ogni sotto-partizione viene eletto un supervisore che avrà compiti di coordinamento della gestione delle risorse tra le sotto-partizioni. In questo modo sarà possibile rendere ancora più scalabile il problema di Resource Allocation. Occorrerà sviluppare un nuovo algoritmo di partizionamento e progettare un nuovo metodo di assegnamento delle risorse alle partizioni.

Un ulteriore direzione di ricerca riguarda la possibilità di realizzare un modello di Resource Allocation completamente distribuito in cui gli Application Manager descritti nel Capitolo 3 verranno implementati come entità pari che interagiscono tra di loro senza nessuna forma di coordinamento centralizzato.

Infine, verranno valutate estensioni all'algoritmo di partizionamento applicativo discusso nel Capitolo 4 considerando durante la creazione delle partizioni il livello di availability richiesto da ciascuna classe applicativa. Precisamente verranno introdotte due soglie per la valutazione dell'Area Ratio nella fase di aggregazione: una soglia ξ_1 che permette l'aggregazione delle partizioni con bassa availability ed soglia $\xi_2 < \xi_1$ per quelle con elevata richiesta di disponibilità. In questo modo ci si aspetta che sarà possibile aggregare le classi con alta availability in poche partizioni che potranno essere gestite in modo efficace

condividendo tra di loro un alto numero di server a supporto dei tier applicativi.

Bibliografia

[1] Numerical recipes. Cambridge University Press, 2007. http://www.nr.com/.

[2] T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoreticalapproach. *IEEE Transactions on Parallel and Distributed Systems*, 15(2), March 2002.

[3] Rajkumar Buyya Anton Beloglazov. Energy efficient resource management in virtualized cloud data centers. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010.

[4] D. Ardagna, M. Trubian, and L. Zhang. Sla based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing*, 67(3):259–270, 2007.

[5] Danilo Ardagna, Marco Trubian, and Li Zhang. Sla based profit optimization in multi-tier systems. *NCA '05: Proceedings of the Fourth IEEE International Symposium onNetwork Computing and Applications*, pages 263–266, 2005.

[6] Hari Balakrishnan John Guttag Bruce Maggs Asfandyar Qureshi, Rick Weber. Cutting the electric bill for internet-scale systems. *SIGCOMM*, 2009.

[7] B. Panicucci L. Zhang B. Addis, D. Ardagna. Autonomic management of cloud service centers with availability guarantees. *Cloud 2010 Proceedings*, pages 220–227, 2010.

[8] Folco Angelo Bombardieri. Tecniche di resource allocation per la gestione dei consumi energetici di service center con vincoli di availability. *Dipartimento di Elettronica ed Informazione, V FacoltÃ di Ingegneria, Politecnico di Milano*, 2008.

[9] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. *SIGMETRICS*, 2005.

[10] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. *SIGMETRICS Performamce Evaluation Review*, 33(1):303–314, 2005.

[11] J. Choi, S. Govindan, B. Urgaonkar, and A. Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In *MASCOTS*, 2008.

[12] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification, revision 3.0b. http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf, 2006.

[13] Elmootazbellah Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. *Proceedings of the Workshop on Power-Aware Computing Systems*, February 2002.

[14] Danilo F. Ghirardelli. Progetto di allocatori di risorse in sistemi di autonomic computing. *Dipartimento di Elettronica ed Informazione, V FacoltÃ di Ingegneria, Politecnico di Milano*, 2004.

[15] V. Gupta and M. Harchol-Balter. Self-adaptive admission control policies for resource-sharing systems. In *SIGMETRICS*, 2009.

[16] Shuan Zhao Yanbo Han Hailue Lin, Kai Sun. Feedback-control-based performance regulation for multi-tenant applications. *2009 15th International Conference on Parallel and Distributed Systems*, 24:134–141, December 2009.

[17] James Hamilton. Datacenter power efficiency. http://perspectives.mvdirona.com/2010/11/12/DatacenterPowerEfficiency.aspx, December 2010.

[18] IDC. Idc, the premier global market intelligence firm. http://www.idc.com/.

[19] European Commission DG INFSO. Impacts of information and communication technologies on energy efficiency.

[20] Intel. Intel server products power budget analysis tool. http://www.intel.com/support/motherboards/server/sb/cs-016976.htm.

[21] David M.Chess Jeffrey O.Kephart. The vision of autonomic computing. *IEEE Computer*, pages 41–50, January 2003.

[22] Daniel Mosse Luciano Bertini Julius C.B. Leite, Dara Kusic. Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster.

[23] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC2009 Proc.*, 2009.

[24] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *ICAC Proc.*, June 2007.

[25] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *ICAC2009 Proc.*, 2009.

[26] D. Kusic and N. Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *ICAC 2006 Proc.*, 2006.

[27] D. Kusic, J. O. Kephart, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC 2008 Proc.*, 2008.

[28] Dara Kusic and Nagarajan Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioningin enterprise computing systems. *Cluster Computing*, 10(4):395–408, 2007.

[29] PierGiorgio Malusardi. Risparmio energetico, green computing, windows server 2008 e la virtualizzazione - parte 1. http://blogs.technet.com/pgmalusardi/archive/2008/04/24/risparmio-energetico-green-computing-windows-server-2008-e-la-virtualizzazione-1.aspx.

[30] PierGiorgio Malusardi. Risparmio energetico, green computing, windows server 2008 e la virtualizzazione - parte 3. http://blogs.technet.com/pgmalusardi/archive/2008/05/05/risparmio-energetico-green-computing-windows-server-2008-e-la-virtualizzaione-3.aspx.

[31] Wolfgang Schott Nedeljko Vasic, Thomas Scherer. Thermal-aware workload scheduling for energy efficient data centers.

[32] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef. Performance management for cluster-based web services. *IEEE Journal on Selected Areas in Communications*, 23(12), December 2005.

[33] Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance management for cluster-based web services. *IFIP/IEEE Eighth International Symposium on Integrated Network Management*, pages 247–261, 2003.

[34] Francesco Lo Presti Paolo Campegiani. A general model for virtual machines resources allocation in multi-tier distributed systems.

[35] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[36] W. Qin and Q. Wang. Modeling and control design for performance management of web servers via an lpv approach. *IEEE Transactions on Control Systems Technology*, 13(1), Jan 2002.

[37] N. Kandasamy R. Wang*, D. Kusic. A distributed control framework for performance management of virtualized computing environments.

[38] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: coordinated multi-level power management for the data center. *SIGARCH Comput. Archit. News*, 36(1):48–59, 2008.

[39] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. *SIGARCH Comput. Archit. News*, 34(2), 2006.

[40] M. Steinder, I. Whalley, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. *SIGOPS Oper. Syst. Rev.*, 42(1), 2008.

[41] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW2007*, 2007.

[42] G. Tesauro, N.K.Jongand R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *ICAC Proc.*, June 2006.

[43] Mark S. Squillante Tomasz Nowicki and Chai Wah Wu. Fundamentals of dynamic decentralized optimization in autonomic computing systems. *Mathematical Sciences Department, IBM Thomas J. Watson Research Center.*

[44] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi. Analytic modeling of multitier internet applications. *ACM Transaction on Web*, 1(1), January 2007.

[45] B. Urgaonkar and P. Shenoy. Sharc: Managing cpu and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1):2–17, 2004.

[46] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, MikeSpreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS internationalconference on Measurement and modeling of computer systems*, pages 291–302, 2005.

[47] X. Feng W. Feng and R.Ge. Green supercompunting comes of age. *IT professional*, 2008.

[48] Mianyu Wang, Nagarajan Kandasamy, Allon Guez, and Moshe Kam. Adaptive performance control of computing systems via distributed cooperative control: application to power management in computing clusters. *Proceedings of the 3rd International Conference on Autonomic Computing, ICAC 2006*, pages 165–174, 2006.

[49] X. Wang, Z. Du, Y. Chen, and S. Li. Virtualization-based autonomic resource management for multi-tier web applications in shared data center. *J. Syst. Softw.*, 81(9):1591–1608, 2008.

[50] X. Wang, Z. Du, Y. Chen, and S. Li. vgreen: a system for energy efficient computing in virtualized environments. *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (ISLPED '09)*, 81:243–248, August 2009.

[51] Zhi-Li Zhang, Don Towsley, and Jim Kurose. Statistical analysis of generalized processor sharing schedulingdiscipline. *SIGCOMM Computer Communication Review*, 24(4):68–77, 1994.

[52] X. Zhu, D. Young, B. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D.Gmach, R. Gardner, T. Christian, and L. Cherkasova:. 1000 islands: An integrated approach to resource management for virtualized data centers. *Journal of Cluster Computing*, 12(1):45–57, 2009.