Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Estensione del framework autonomico SelfLet con il servizio di cloud computing di Amazon

Relatore: Elisabetta Di Nitto
Correlatore: Nicolò Maria Calcavecchia

Tesi di Laurea di: Alessandro GANDINI matr. 725350

Anno Accademico 2009-2010

Ai miei genitori.

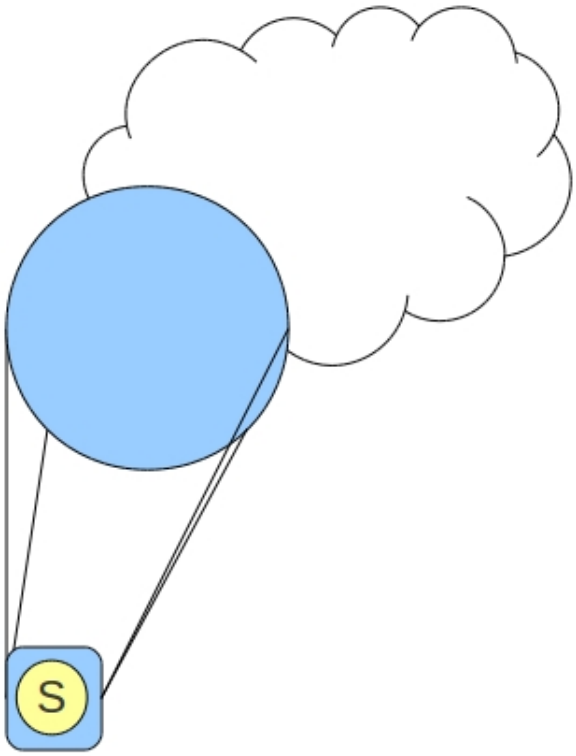

Figura 1: una cloud selflet

first steps of the thesis:

1) Make alot of coffee or tea, begin drinking it

2) Roll up your sleeves

3) Take a photo of your loved ones as a keepsake and say a long goodbye

4) Lock yourself in your office

5) Make more coffee

6) Let's go!

Indice

Elenco delle figure

Capitolo 1

Introduzione

L'autonomic computing è un paradigma di sviluppo emerso negli ultimi anni con l'obiettivo di gestire la crescente complessità dei sistemi software. I sistemi così realizzati sono in grado di gestirsi autonomamente, in maniera analoga a quanto avviene nel sistema nervoso autonomo del corpo umano. Il paradigma prevede un insieme di proprietà, chiamate Self-*: auto-configurazione, auto-guarigione, auto-ottimizzazione e auto-protezione. Il cloud computing è un modello di computazione nel quale l'esecuzione delle applicazioni avviene in un ambiente amorfo a cui vengono aggiunte e rimosse risorse dinamicamente. Le applicazioni e i servizi devono sfruttare la dinamicità intrinseca di questo ambiente, al fine di ottenere un comportamento ottimale in termini di costo, prestazioni e affidabilità. L'avvento del cloud computing introduce una nuova proprietà tra quelle previste dal paradigma autonomico: l'auto-approvvigionamento. Un sistema che esibisce questa proprietà ha la capacità di espandere e contrarre il proprio pool di risorse in accordo a politiche di alto livello che ne governano il comportamento. L'obiettivo di questo lavoro di tesi è quello di proporre una possibile soluzione al problema dell'integrazione tra il paradigma autonomico e il modello di computazione su cloud al fine di fare emergere la proprietà di auto-approvvigionamento. Per raggiungere questo obiettivo una particolare istanza di framework autonomico, il SelfLet framework, viene esteso con le funzionalità di interazione con il servizio di cloud computing di Amazon. Una SelfLet è un singolo componente autonomico autosufficiente che, posto in una rete, è capace di comunicare con altre SelfLets. Una SelfLet è caratterizzata da un insieme di obiettivi da ottenere, dai comportamenti capaci di raggiungere questi obiettivi e da regole per la

gestione di situazioni critiche.Inizialmente sono state studiate le metodologie per lo sviluppo di sistemi autonomici e di sistemi basati su cloud, con l'obiettivo di definire un modello per la gestione autonomica di risorse acquisite e rilasciate dinamicamente. Successivamente è stato sviluppato un framework per l'interazione con il servizio di cloud computing, e sono stati definiti i processi di deploy dei nodi di una rete composta da SelfLets. L'implementazione di questi processi è seguita dallo studio di un modello di interazione tra i nodi, per la gestione coordinata delle risorse su cloud e in accordo con il principio di località che governa le decisioni prese da ogni SelfLet. Il lavoro continua con la definizione delle operazioni di espansione e contrazione della rete all'interno della cloud e con l'estensione dell'architettura del SelfLet framework attraverso l'introduzione di nuove politiche di ottimizzazione. La soluzione proposta è stata validata attraverso lo sviluppo di un caso di studio, di cui sono stati raccolti i risultati sperimentali per l'analisi del contributo autonomico apportato dalla nuova funzionalità.

Organizzazione

Questa tesi è organizzata come segue:

- Nel Capitolo 2 viene introdotto lo stato dell'arte dei sistemi autonomici tenendo presente il modello di riferimento proposto da IBM. Viene descritto lo stato dell'arte del cloud computing e l'impatto di questo paradigma sullo sviluppo dei sistemi software.

- Il Capitolo 3 presenta due istanze delle tecnologie trattate nel Capitolo 2, il framework autonomico SelfLet e il servizio di cloud computing Amazon Web Services, che hanno costituito il punto di partenza di questo lavoro. Vengono inoltre presentati i motivi di una loro integrazione.

- Nel Capitolo 4 viene descritta la fase di design che comprende l'analisi dei requisiti, la definizione delle tipologie di SelfLet che compongono il sistema e i servizi che le caratterizzano.

- Il Capitolo 5 riguarda i dettagli di progetto e dell'implementazione. Viene descritta l'estensione dell'architettura del SelfLet framework con il modello di espansione su cloud.

- Il Capitolo 6 presenta i casi di studio implementati ed i risultati ottenuti dalla loro esecuzione.
- La tesi si conclude con il Capitolo 7. Sono discussi i risultati ottenuti e sono presentati alcuni possibili sviluppi futuri.

Capitolo 2

Analisi dello stato dell'arte

2.1 Autonomic computing

Il termine Autonomic Computing è stato coniato da IBM nel 2001 [22] con l'obiettivo di sviluppare soluzioni di elaborazione distribuita in grado di rispondere ai requisiti della crescente complessità ed eterogeneità dei sistemi informatici. La visione si ispira alla metafora del sistema nervoso autonomico (ANS): in particolare alla sua funzione di regolare l'omeostasi dell'organismo attraverso meccanismi autonomi, mascherandone la complessità alla volontà cosciente. Oggi, nella sua accezione più generale, tale definizione comprende l'insieme di tecniche e metodi utilizzati nello sviluppo di sistemi complessi che siano in grado di autoregolarsi senza intervento da parte dell'uomo. Alla proposta originaria si sono affiancate quelle offerte da altre discipline quali organic computing, bio-inspired computing, self-organizing systems, ultrastable computing, e sistemi adattativi. Il tentativo di rendere automatica la gestione delle risorse di computazione non è nuovo nel mondo della ricerca. Negli anni, i sistemi software sono stati fatti evolvere in modo da affrontare la complessità propria dei problemi di controllo, di adattamento, di condivisione delle risorse e di gestione delle operazioni. Sono stati così sviluppati sistemi in grado di gestire questa complessità rispetto a singoli domini applicativi come la sicurezza, la tolleranza ai gusti, l'intelligenza artificiale, le gestione di rete, gli agenti software e la gestione dinamica delle risorse. Il passo successivo di questo processo di evoluzione, è quello di far convergere queste singole discipline in un unico campo chiamato appunto Autonomic computing, introducendo il con-

cetto di self-management. I sistemi autonomici sono in grado di prendere decisioni in modo autonomo, controllando costantemente il loro stato e adattando il proprio comportamento al variare delle condizioni interne ed esterne di esecuzione, per esempio per migliorare le prestazioni, oppure per recuperare situazioni critiche o di errore.

2.1.1 Caratteristiche dei sistemi autonomici

Un sistema autonomico può essere visto come un insieme di componenti autonomici, ognuno dei quali gestisce il proprio comportamento interno e si relaziona con gli altri in accordo a definite politiche di alto livello. I principi che governano tutti i sistemi autonomici possono essere identificati da alcune caratteristiche generalmente sono chiamate proprietà self-* [22].

Conoscenza di sè stesso: un sistema autonomico deve avere una conoscenza dettagliata dei componenti che lo costituiscono, del proprio stato, delle capacità, limiti, risorse disponibili e delle interdipendenze con altri sistemi.

Auto-configurazione: un sistema autonomico deve avere la possibilità di configurare e riconfigurare sè stesso, o un suo insieme di componenti, al verificarsi di diverse e impredicibili condizioni. Deve inoltre essere in grado di bilanciare dinamicamente le proprie risorse basandosi sul proprio stato e sullo stato dell'ambiente di esecuzione.

Auto-ottimizzazione: un sistema autonomico deve mantenere elevata la propria efficienza attraverso operazioni di tuning delle risorse e load balancig dei carichi di lavoro. Un tale sistema deve continuamente monitorare sè stesso per individuare eventuali decrementi delle performance e ottimizzare il proprio comportamento sulla base dei cambiamenti delle proprie necessità o dell'ambiente di esecuzione.

Auto-riparazione: un sistema autonomico deve individuare e prevenire casi di errore, interruzioni delle proprie funzionalità e deve ripararsi in caso di malfunzionamenti. Dovrebbe essere in grado di ripristinare le proprie funzionalità di default a fronte di eventi imprevisti e scoprire il problema o le possibili cause che hanno causato l'interruzione dei propri servizi.

Infine dovrebbe cercare di ridurre ritardi o perdita di informazioni attraverso strategie di riconfigurazione o di utilizzi alternativi delle proprie risorse.

Auto-protezione: i sistemi autonomici devono garantire la sicurezza delle proprie informazioni e delle risorse anticipando, individuando e proteggendosi da attacchi di diversa natura come accessi non autorizzati o denial-of-service.

Conoscenza dell'ambiente e del contesto: un sistema autonomico deve essere consapevole del proprio ambiente di esecuzione e del contesto in cui svolge la propria attività e deve essere in grado di reagirne ai cambiamenti.

Ipotesi di mondo aperto: un sistema autonomico deve poter funzionare in ambienti eterogenei e deve pertanto essere costruito sulla base di protocolli e interfacce standard. In altre parole un sistema autonomico non dovrebbe essere una soluzione proprietaria.

Capacità anticipatorie: un sistema autonomico può anticipare l'ottimizzazione delle risorse necessarie per andare incontro alle necessità dell'utente, nascondendo la complessità. Questa caratteristica aggiunge pro-attività al self-management.

2.1.2 Livelli di automazione

Le capacità autonomiche di un sistema software vengono classificate per gradi. Questa metrica è stata introdotta da IBM per definire la situazione dei sistemi esistenti e per tracciare una road-map delle caratteristiche desiderabili che un sistema autonomico dovrebbe esibire nel prossimo futuro.

- Basic: operatori specializzati utilizzano tools di monitoring per individuare malfunzionamenti nel sistema ed intervenire manualmente
- Managed: monitoring tools automatizzati e meccanismi di analisi dei dati riducono il livello di amministrazione degli operatori
- Predictive: patterns comportamentali producono previsioni sul futuro stato del sistema

- Adaptive: in aggiunta alle capacità del precedente livello, il sistema può eventualmente agire su di se per eseguire operazioni di recovery o andare incontro ai requisiti di servizio

- Fully Autonomic: sistemi e componenti sono auto-gestiti dinamicamente attraverso regole e policies senza la necessità di intervento da parte di amministratori umani.

2.1.3 Architettura dei sistemi autonomici

Quando ha introdotto il concetto di sistema autonomico, IBM ha descritto un modello generale di architettura [42], valido ancora oggi, che dovrebbe essere seguito nella costruzione di questo tipo di sistemi. Un sistema autonomico consiste in un insieme di autonomic elements che si occupano di gestire delle managed resources e che forniscono servizi a utenti umani o ad altri autonomic elements. Ogni autonomic element è composto da un autonomic manager, da un touchpoint e da uno o più managed elements. Un autonomic manager è un componente che si occupa della gestione automatica dei componenti controllati, in accordo con il comportamento definito da un più alto livello attraverso un' interfaccia di gestione. Il componente implementa un loop di controllo chiamato MAPE. Perchè un sistema sia gestito autonomicamente è necessario un metodo automatico in grado di raccogliere le informazioni necessarie alla sua gestione. Queste informazioni vengono poi analizzate per individuare eventuali cambiamenti nel sistema e per la composizione di una sequenza di azioni che specificano i cambiamenti necessari. Le azioni vengono poi messe in atto. Il monitor (M) è la funzione che fornisce il meccanismo di collezione, aggregazione, filtrazione e generazione di report delle informazioni raccolte dalla managed resource. La funzione di analisi (A) fornisce un meccanismo di correlazione dei dati raccolti e di modellizzazione di situazioni complesse. Questi meccanismi consentono all'autonomic manager l'apprendimento rispetto alla risorsa gestita e di predire le situazioni future. La funzione di pianificazione (P) fornisce un meccanismo di selezione delle azioni necessarie a raggiungere i goal e gli obiettivi specificati attraverso un sistema di politiche ,come spiegato nella sezione 2.1.4. Infine la funzione di esecuzione (E) controlla l'attuazione delle operazioni individuate nella fase di pianificazione. Le parti descritte comunicano e collaborano attraverso lo scambio di informazioni; questo è possibile

utilizzando un componente per la memorizzazione della conoscenza condivisa del sistema.

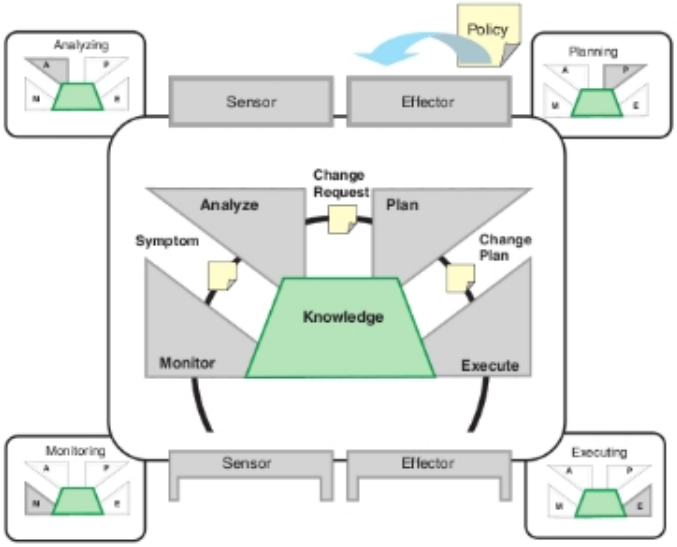

Figura 2.1: Autonomic manager e ciclo di controllo MAPE-K

Un touchpoint è un elemento nel sistema che espone lo stato e gestisce le operazioni su una specifica managed resource o su un loro insieme; implementa un'interfaccia tra queste e l'autonomic manager ed è costituita da sensori e attuatori. I sensori permettono le operazioni di monitoring della risorsa raccogliendo informazioni che la riguardano ed eventualmente organizzandole in metriche. Gli attuatori sono lo strumento attraverso il quale l'autonomic manager può mettere in pratica le decisioni prese nella fase di esecuzione del ciclo di controllo. Infine, le managed resources sono generiche risorse come router, server, application server o altri autonomic elements. In questo modo è possibile la realizzazione di sistemi complessi come composizione di parti elementari

2.1.4 Politiche

Seguendo una visione gerarchica dei sistemi autonomici [33], in cui ogni autonomic manager può gestire come managed resource un altro autonomic element, segue che gli elementi di un sistema possono funzionare a diversi livelli, seguendo diverse specifiche di comportamento. Ai livelli più bassi, i componenti hanno un insieme limitato di capacità e interazioni. Salendo di

livello, seguono invece goal i cui obiettivi sono specificati con delle politiche, e le interazioni tra i componenti possono evolvere. Le politiche giocano pertanto un ruolo fondamentale nell'autonomic computing in quanto guidano il comportamento del sistema.

Action policies: una action policy determina l'azione che dovrebbe essere presa quando il sistema si trova nello stato corrente Tipicamente questa azione ha la forma IF (condition) ELSE (action), dove la condizione può specificare un singolo stato o un insieme di stati che la soddisfano. Questo tipo di politica serve per fare in modo che il sistema esibisca un comportamento razionale.

Goal policies: piuttosto che specificare esattamente cosa fare nello stato corrente, questo tipo di politiche specificano uno stato desiderato o uno o più criteri che caratterizzano un insieme di stati desiderabili, lasciando che sia il sistema a decidere come raggiungere tali stati. Il sistema genera un comportamento razionale partendo da questi goal e usando algoritmi di modellizzazione e pianificazione. Questo tipo di politica permette maggiore flessibilità e permette di nascondere agli sviluppatori umani i dettagli di basso livello del funzionamento del sistema.

Utility-function policies: le utility-function policies definiscono una funzione obiettivo che guida il comportamento del sistema in ogni possibile stato. Invece di modellizzare una classificazione tra stati desiderabili e stati indesiderabili, queste esprimono una valutazione scalare per ogni stato. Siccome lo stato più desiderabile non è specificato esplicitamente, questo viene calcolato su base ricorsiva, selezionando lo stato con il maggiore valore di utilità rispetto a quelli presenti nell'insieme degli stati raggiungibili. Queste politiche forniscono una maggiore granularità e una più flessibile specifica dei comportamenti rispetto alle altre tipologie. Situazioni in cui una specifica definita da goal multipli potrebbe sollevare casi di conflitto (perché non raggiungibili simultaneamente) le utility functions permettono la non ambiguità, attraverso la valutazione razionale di un appropriato trade-off. Ciò nonostante, queste politiche rendono necessarie la specifica di insiemi di preferenze che sono difficili da elicitare.

2.2 Cloud computing

Il rapido sviluppo delle tecnologie di elaborazione e archiviazione e il successo di internet hanno reso più economiche, più potenti e più disponibili le risorse di calcolo. Questo trend tecnologico ha permesso la realizzazione di un nuovo modello computazionale chiamato cloud computing, nel quale le risorse (ad esempio la CPU e la memoria) sono fornite come utilities generiche che possono essere richieste e rilasciate on-demand dagli utenti attraverso la rete. Nell'ambito del cloud computing, il tradizionale ruolo del service provider è diviso in due tipologie: gli infrastructure providers che gestiscono le cloud platforms e rendono disponibili le risorse a seconda di un modello di prezzo basato sull'utilizzo, e i service providers, che utilizzano le risorse fornite da uno o più infrastructure providers per servire gli utenti finali. L'emergere del cloud computing ha avuto un impatto enorme sull'industria informatica negli ultimi anni, in merito alle seguenti caratteristiche:

- No up-front investment: il cloud computing prevede un modello di prezzo pay-you-go. Un service provider non ha bisogno di investire in infrastrutture. Può invece affittare risorse dalla cloud a seconda dei propri bisogni, pagandone l'utilizzo effettivo.

- Bassi costi operativi: le risorse in un ambiente cloud possono essere rapidamente allocate e deallocate on-demand. Un service providers non deve più dimensionare la capacità dell'infrastruttura informatica sulla base dei momenti di picco di carico. Ciò consente un grande risparmio perché le risorse possono esser rilasciate per risparmiare sui costi operativi quando la richiesta di servizio è bassa.

- Elevata scalabilità: gli infrastructure providers mettono a disposizione una grande quantità di risorse dai data centers e le rendono facilmente accessibili. Un service providers può scalare facilmente il numero di macchine a supporto dei servizi offerti in modo da gestire un rapido incremento della richiesta del servizio offerto (esempio: flash-crowd effect). Questa tecnica viene chiamata surge computing [32].

- Interazione dinamica: i servizi ospitati nella cloud sono generalmente web-based ed possibile interagire con essi attraverso interfacce, API o

comandi shell. Questo permette lo sviluppo di applicazioni che gestiscono le risorse dinamicamente.

- Riduzione dei costi di manutenzione: de localizzando l'infrastruttura sulla cloud, un service provider scarica i costi legati al mantenimento dell'infrastruttura (come ad esempio malfunzionamenti hardware) all'infrastructure provider. Ciò aumenta inoltre l'efficienza degli interventi di manutenzione.

L'idea alla base del cloud computing non è nuova. John McCarthy negli anni '60 aveva già immaginato che i servizi di calcolo potessero essere distribuiti al pubblico come un'utility [47]. Tuttavia, è stato dopo che il CEO di Google Eric Schmidt ha usato la parola per descrivere il modello di business di fornire servizi attraverso internet nel 2006 che il termine ha davvero cominciato a guadagnare popolarità. La definizione di cloud computing fornita dal The National Institute of Standards and Technology (NIST) [1] è la seguente: *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*
Il cloud computing non è una nuova tecnologia, ma piuttosto un nuovo modello operativo sviluppato sulla base di un insieme di tecnologie preesistenti, come la virtualizzazione e il modello di prezzo utility-based. Il cloud computing fa leva su di esse per incontrare i requisiti tecnologici ed economici dell'attuale richiesta di tecnologia.

2.2.1 Architettura

E' possibile descrivere l'architettura di un ambiente di cloud computing modellizzandola in quattro strati: hardware layer, infrastructure layer, platform layer e application layer. Segue la descrizione di ciascuno di questi livelli.

Hardware layer: questo strato è responsabile della gestione delle risorse fisiche della cloud, incluse le macchine fisiche (server), i routers, gli switches e i sistemi di alimentazione e raffreddamento. Lo strato hardware è tipicamente implementato nei data centers. Un data center contiene migliaia di server che sono organizzati in racks. La gestione

hardware comprende operazioni di configurazione, realizzazione di infrastrutture fault-tolerant, gestione del traffico di rete e gestione delle risorse.

Infrastructure layer: chiamato anche strato di virtualizzazione, questo strato fornisce le risorse fondamentali agli strati di più alto livello, generalmente servizi computazionali, data storage e servizi di rete, attraverso la creazione di pool di memoria e risorse computazionali partizionando le risorse fisiche attraverso tecnologie di virtualizzazione come Xen [31], KVM [28] and VMware [30]. L'infrastructure layer è un componente essenziale del cloud computing, perché molte caratteristiche chiave, come l'assegnamento dinamico delle risorse, sono disponibili soltanto attraverso il meccanismo di virtualizzazione.

Platform layer: gli utenti di questo strato sono gli sviluppatori, che implementano le loro applicazioni e ne fanno il deploying sulla cloud. Gli infrastructure providers supportano queste operazioni fornendo dei framework di sviluppo che consistono in insiemi di API per facilitare l'interazione con l'ambiente cloud o il cui scopo è quello di minimizzare il peso del deploy delle applicazioni all'interno delle macchine virtuali. Ne sono un esempio i servizi di auto-scaling e load-balancing offerti da Amazon, che si occupano di ottimizzare l'allocazione dinamica delle istanze virtuali in modo trasparente alle applicazioni. In generale gli sviluppatori hanno la possibilità di integrare on-demand questi servizi nelle loro applicazioni.

Application layer: questo è lo strato con cui interagiscono gli utilizzatori finali, i quali accedono ai servizi offerti attraverso il web. Il deploy di una cloud application viene eseguito sull'infrastruttura offerta da un infrastructure provider in modo che gli sviluppatori possano apportare modifiche al software o aggiungere nuove funzionalità senza che si renda necessaria la distribuzione di update o service pack. A questo livello diventano critici gli aspetti di sicurezza e di qualità del servizio, che vengono regolati attraverso i contratti di Service Level Agreement (SLA).

Rispetto ai tradizionali servizi di hosting, come le server farms, l'architettura del cloud computing è modulare. In modo simile al modello OSI dei protocolli di rete, ciascun livello ha basso accoppiamento con quelli ad esso

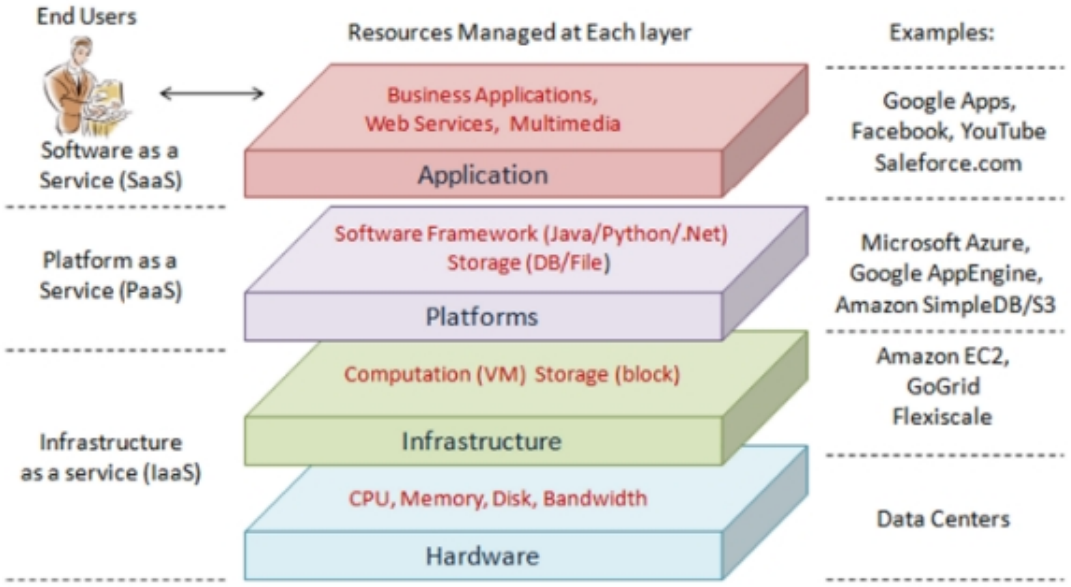

Figura 2.2: Modello a strati dell'architettura cloud

adiacenti, permettendo a ogni strato di evolvere separatamente.
Questa architettura permette un modello di sviluppo service-driven: ogni strato dell'architettura può essere implementato come un servizio verso lo strato superiore. Al contrario, ogni strato può essere percepito come client dello strato sottostante. I servizi di cloud computing offerti possono essere raggruppati in tre categorie: Infrastructure as a Service, Platform as a Service e Software as a Service.

IaaS: in questa categoria si collocano i servizi di approvvigionamento on-demand delle risorse infrastrutturali, queste sono solitamente macchine virtuali (VM). gli infrastructure providers che offrono servizi di tipo IaaS sono chiamato IaaS provider. Esempi di IaaS providers includono Amazon EC2 [4], GoGrid [16] e Flexiscale [26].

PaaS: in questo caso vengono fornite le risorse dello strato platform, inclusi il supporto al sistema operativo e framework di sviluppo software. Esempi di provider Paas sono Google App Engine [10] e Microsoft Windows Azure [29].

SaaS: per SaaS si intendono servizi forniti sotto forma di applicazioni on-demand su internet. Esempi di provider SaaS sono Salesforce.com [23] e Rackspace [21].

E' possibile dare una definizione di diversi tipi di cloud in base agli scopi del loro utilizzo e in base alle modalità con cui vengono offerti i servizi.

Cloud pubbliche: una cloud nella quale i service providers offrono le loro risorse come servizi al pubblico. Le cloud pubbliche offrono diversi benefici chiave ai service providers, tra cui la possibilità di evitare di far fronte ad investimenti iniziali sull'infrastruttura e lo spostamento dei costi di manutenzione sugli infrastructure providers. Attualmente, le cloud pubbliche peccano di controlli fini sui dati, reti e setting di sicurezza, che ostacolano il loro effettivo utilizzo in motli scenari di business (E' un esempio il caso del settore sanitario, nel quale i forti vincoli di privacy rendono necessario conoscere su quali macchine fisiche risiedono i dati clinici dei pazienti).

Cloud private: anche conosciute come cloud interne, le cloud private sono progettate per l'utilizzo esclusivo da parte di una singola organizzazione. Una cloud privata può essere costruita e gestita dall'organizzazione stessa o da provider esterni, e offre il più alto grado di controllo sulla piattaforma in termini di affidabilità e sicurezza. Tuttavia, la percezione comune è che queste architetture siano simili alle tradizionali server farms e non permettono di evitare i costi di mantenimento come nel caso delle cloud pubbliche.

Cloud ibride: una cloud ibrida è una combinazione di modelli pubblici e privati, che cerca di affrontare i limiti di entrambi gli approcci. In una cloud ibrida, una parte dei servizi infrastrutturali viene eseguita nella cloud privata, mentre la restante parte viene eseguita nella cloud pubblica. Le cloud ibride offrono più flessibilità rispetto alle precedenti categorie. In particolare, forniscono maggiori controlli e sicurezza sui dati delle applicazioni rispetto alle cloud pubbliche, mentre facilitano l'espansione e la contrazione on-demand. Ciò nonostante, progettare una cloud ibrida richiede di determinare attentamente il miglior bilanciamento tra le risorse dedicate alla sua parte pubblica e a quella privata.

Virtual cloud: una soluzione alternativa per affrontare i limiti delle cloud pubbliche e private si chiama Virtual Private Cloud (VPC). Una VPC è essenzialmente una piattaforma eseguita su una cloud pubblica. La principale differenza è che una VPC si basa sulla tecnologia di una rete virtuale privata (VPN) che permette ai service providers di avere pieno controllo nella configurazione dei settaggi di sicurezza nel progetto della topologia della rete. Le VPC non virtualizzano solamente i server e le applicazioni, ma anche la rete di comunicazione sottostante. Forniscono inoltre una più facile migrazione dei sistemi esistenti proprietari su una infrastruttura web-based grazie alla virtualizzazione dello strato di rete.

2.2.2 Caratteristiche del cloud computing

Il cloud computing presenta alcune caratteristiche che ne differenziano il modo in cui è possibile gestire le risorse rispetto agli approcci di sviluppo tradizionali.

- Multy-tenacy: in ambiente cloud, i servizi che appartengono a molti providers si trovano in un unico data center. Le performance di questi sistemi dipendono pertanto dalle scelte di gestione di questi servizi da parte dei providers. L'architettura a strati del cloud computing fornisce una naturale divisione delle responsabilità: il soggetto responsabile di ogni layer può focalizzarsi sugli specifici obiettivi associati al proprio dominio.

- Allocazione dinamica delle risorse: gli infrastructure providers offrono un insieme di risorse di calcolo che possono essere allocate dinamicamente ai diversi client. Questa capacità di assegnamento dinamico permette agli infrastructure providers di gestire il carico di utilizzo delle proprie risorse. Per esempio, un IaaS provider può far leva sulla tecnologia di migrazione delle macchine virtuali per fare server consolidation in modo da massimizzare l'utilizzo di risorse e ridurre i costi di alimentazione e raffreddamento.

- Auto-organizzazione: l'allocazione dinamica delle risorse permette ai service providers di gestirne la richiesta in maniera automatica. Questo permette alle applicazioni di rispondere velocemente ai rapidi cambiamenti nelle richieste di servizi.

- Geo-distribuzione: le cloud sono generalmente accessibili attraverso internet; il web viene utilizzato come rete di distribuzione di servizi. Inoltre, per mantenere prestazioni elevate e aumentare il livello di fault tolerance, molti servizi di cloud sono supportati da numerosi data centers indipendenti tra loro e localizzati in diverse parti del mondo.

- Orientamento al servizio: come menzionato precedentemente, il cloud computing adotta un modello operativo service-driven. Lo sviluppo di applicazioni basate su cloud pone pertanto grande enfasi sui servizi. In una cloud ogni provider IaaS, PaaS e SaaS offre i propri servizi in accordo con il Service Level Agreement (SLA) negoziato con il client. Il rispetto del Service Level Agreement è quindi un obiettivo critico per ogni provider.

2.2.3 Caratteristiche dello sviluppo in ambiente cloud

Le caratteristiche fin qui descritte di questo modello computazionale introducono elementi di complessità nello sviluppo delle applicazioni. Tra questi i più importanti sono gli aspetti legati alla sicurezza e la privacy, la necesssità da parte di service providers e utilizzatori di negoziare i livelli di Quality of Services (QoS) stabilendo dei contratti SLA, la progettazione di meccanismi e algoritmi per l'allocazione dinamica delle macchine virtuali e delle altre risorse in modo da rispettare gli SLA, ed eventualmente la gestione dei rischi associati alla loro violazione. Inoltre si rende necessario lo sviluppo di protocolli che permettano l'interoperabilità tra gli ambienti cloud di diversi infrastructure providers. Lo sviluppo di software in ambiente cloud è quindi una pratica multilaterale rispetto allo sviluppo di una normale appliazione. Le differenze tra i due approcci sono descritte nei seguenti punti.

- Software composition: nelle tradizionali applicazioni software, gli sviluppatori progettano e implementano un insieme coerente di moduli. Lo sviluppo in ambiente cloud è multidisciplinare, e attualmente prevede la composizione di componenti interoperabili forniti da terze parti e la selezione dei servizi.

- Disponibilità del codice: nello sviluppo di applicazioni standard, il codice sorgente dei progetti è interamente disponibile agli sviluppatori. In ambi-

ente cloud, essendo il software composto da numerosi componenti forniti da altri providers, si pone il problema della non completa disponibilità del codice e della necessità di avere una comprensione completa del sistema.

- Loosely Coupled Service Design: è necessario progettare i servizi in modo tale che i componenti delle applicazioni presentino basso accoppiamento. Quando le risorse fornite da un ambiente cloud non soddisfano i requisiti computazionali, più cloud possono essere unite per andare incontro alla domanda richiesta. In questo processo, le risorse potrebbero diventare condivise tra più componenti causando l'aggregazione di cloud clouster difficilmente separabili.

- Execution Model: le appliazioni software vengono generalmente eseguite su una sola macchina, mentre in ambiente cloud è spesso distribuito su un grande numero di macchine virtuali. Si pone dunque il problema della completa tracciabilità degli stati del sistema.

- Criteri di test: le interazioni tra componenti in ambiente cloud avvengono in modo simile a quanto succede in ambiente SOA. In questo modello di sviluppo lo scambio di dati tra i servizi può sollevare problemi di integrazione. Si rendono quindi necessari dei criteri di testing, similmente a quelli proposti per i sistemi basati su servizi.

Le principali difficoltà nello sviluppo di un'applicazione basata su cloud sono dovute al fatto che il cloud computing è un modello relativamente recente di utilizzo di risorse computazionali distribuite e condivise. Questo modello è reso possibile dalla disponibilità di enormi quantità di risorse di calcolo gestite e organizzate nei data centers di pochi players, gli infrastructure provider. Il livello di servizio proposto da ciascuno di questi è spesso differente, come spiegato nella sezione 2.2.1. Per questi motivi non esiste un white paper di riferimento per le scelte di tipo architetturale che accompagnano la fase di progetto di un'applicazione cloud-oriented. Nonostante i suggerimenti dei providers, che propongono delle best pratices orientate alla gestione della complessità relativa al proprio livello di servizio, non sono stati ancora definiti dei pattern standard. Nel caso di Amazon, le best pratices indicate [49] sono di carattere generale, come la necessità di implementare meccanismi di recovery che sfruttino l'allocazione dinamica delle macchine virtuali, il

disaccoppiamento dei componenti, rafforzare le proprietà elastiche delle applicazioni attraverso scaling proattivo, e la possibilità di progettare i processi in secondo una visione parallela. Oppure orientate al buon utilizzo delle tecnologie proprietarie, proponendo tecniche relative ai servizi di auto-scaling e load balancing. Gli approcci di utilizzo delle risorse virtuali spesso comportano l'implementazione nel sistema di algoritmi di capacity allocation basati sul carico di richieste ricevute dall'applicazione, senza che siano tenuti in conto aspetti legati alla natura distribuita dei sistemi, alle proprietà di auto-organizzazione o di comportamento globale emergente proprie ad esempio dei sistemi autonomici. Inoltre non sono ancora stati standardizzati meccanismi e strategie per le operazioni di auto-deploying delle applicazioni che siano più strutturate della semplice replicazione. A questi riguardi, sono stati trattati nella sezione seguente alcuni dei maggiori ambiti di ricerca inerenti al cloud computing. Infine, tra le molteplici possibilità di interazione offerte dai diversi servizi (come API, richieste basate su protocolli tipici dei web services o script), è necessario scegliere quello più adatto alle caratteristiche e agli obiettivi della propria applicazione.

2.2.4 Settori di ricerca

Nonostante il cloud computing sia oggi largamente utilizzato nell'industria, la ricerca in questo ambito è ancora in una fase iniziale. Molte tematiche non hanno ancora trovato una soluzione, mentre nuove sfide emergono dalle applicazioni industriali.

Automated service provisioning

Una delle caratteristiche chiave del cloud computing è la possibilità di acquisire e rilasciare risorse on-demand. L'obiettivo di un service provider in questo caso è di allocare e deallocare risorse dalla cloud ai fini di soddisfare i prorpri Services Level Objectives (SLOs) minimizzando i costi operazionali. Per raggiungere questo obiettivo è necessario determinare come mappare i Services Level Objectives rispetto ai requisiti di Quality of Services (QoS) del livello sottostante. Inoltre, per reagire tempestivamente alle rapide variazioni della domanda (es. il carico delle richieste per un dato servizio), le decisioni

sull'allocazione delle risorse devono essere prese online. Meccanismi di allocazione automatica delle risorse sono state studiate approfonditamente in passato. Questi approcci prevedono la costruzione di modelli di performance delle applicazioni. In questo modo è possibile predire il numero di istanze richieste per gestire un particolare carico di richieste e per soddisfare i requisiti di Quality of Service, e allocare dinamicamente un certo numero di macchine virtuali. I modelli di applicazione delle performance possono essere costruiti utilizzando varie tecniche. Queste includono Queuing theory [48], Control theory [43] and Statistical Machine Learning [35]. Esiste inoltre una distinzione tra le modalità di controllo proattivo e reattivo delle risorse. L'approccio proattivo usa i dati sul carico di richieste previsto per allocare periodicamente le risorse prima che siano effettivamente necessarie. Quello reattivo reagisce all'istantanea fluttuazione del carico di richieste senza utilizzare alcun modello di predizione. Entrambi gli approcci sono importanti e necessari per l'effettivo controllo delle risorse in un ambiente dinamico.

Virtual machine migration

La virtualizzazione è una tecnica che permette la migrazione delle macchine virtuali tra i server di un data center in modo da permettere il bilanciamento del carico. Inoltre questa tecnica rende robusto e altamente responsivo il servizio di approvigionamento nel data center. La migrazione delle macchine virtuali è una tecnica che deriva dai processi di migrazione. Recentemente, motori di virtualizzzione come Xen e VMWare hanno implementato un sistema di migrazione per le VMs ancora attive che richiede solo pochi millisecondi. I maggiori benefici della tecnica di migrazione sono l'evitare gli hotspots. Ciò non è ancora sufficiente, in quanto attualmente verificare il workload degli hotspot e iniziare una migrazione ha un ritardo nel rispondere alle fluttuazioni del carico di richieste. Inoltre lo stato in memoria dovrebe essere trasferito in maniera consistente ed efficienet, tenendo in considereazione le risorse per le applicazioni delle macchine fisiche.

Server consolidation

E' una tecnica finalizzata a massimizzare l'utilizzo delle risorse minimizzando il consumo di energia nei data center. Consiste nella migrazione su di un sin-

golo server delle VM attive su altri server sottoutilizzati, in modo da rendere possibile il risparmio energetico. La gestione automatica delle migrazioni non dovrebbe permettere di impattare sulle performance delle applicazioni. L'utilizzo di risorse (footprint) da parte di una singola VM può variare nel tempo. Le tecniche di server consolidation, applicate a risorse che sono condivise tra più macchine virtuali, come la banda, la memoria e i dichi I/O, potrebbero avere come risultato il congestionamento di queste quando la VM cambia il suo footprint rispetto alla macchina fisica su cui è montata. E' quindi importante osservare le fluttuazioni del footprint di una VM e utilizzare queste informazioni per mettere in atto queste tecniche in maniera efficiente. Infine, seguendo un approccio autonomico, il sistema dovrebbe reagire rapidamente al congestionamento delle risorse quando questo dovesse verificarsi.

Energy management

Migliorare l'efficienza energetica è un'altra delle maggiori problematiche nel cloud computing. E' stato stimato che il costo del processo di raffreddamento impatta per il 53% del costo totale dei data centers [41]. Nel 2006 i data centers negli Stati Uniti hanno consumato più dell'1.5% del totale dell'energia generata in quell'anno, e che la percentuale è in aumento di circa il 18% all'anno. Gli obiettivi in questo campo sono non solo la riduzione del costo energetico dei data centers, ma anche andare incontro a leggi governative e agli standard ambientali. La progettazione di data centers efficenti dal punto di vista energetico è un problema che ha ricevuto recentemente una grande attenzione. Questa tematica può essere affrontata da diverse prospettive. Per esempio, l'efficienza di un'architettura hardware che permette l'abbassamento della velocità della CPU e che effettua uno spegnimento parziale dei componenti è diventata comune. La schedulazione dei job secondo una prospettiva energetica e l'adozione di tecniche di server consolidation sono altri modi per ridurre il consumo energetico spegnendo le macchine momentaneamente inutilizzate. La chiave per una buona soluzione di queste problematiche è il raggiungimento di un buon trade-off tra l'energia salvata e le performance delle applicazioni. In questo senso, alcuni ricercatori hanno recentemente iniziato a studiare soluzioni di coordinamento per la gestione del consumo energetico negli ambienti di cloud computing.

Traffic management and analysis

L'analisi del traffico di rete è un altro aspetto importante negli attuali data centers. Per esempio, molte applicazioni web si basano sull'analisi del traffico in modo da ottimizzare l'esperienza degli utilizzatori. Gli operatori di rete necessitano di sapere come cambia il traffico attraverso la rete in modo da prendere decisioni di gestione e pianificazione. Ciò nonostante sono necessari miglioramenti nei metodi di misura e di analisi del traffico di rete. Per prima cosa se la densità di collegamenti tra i dispositivi nei data centers è molta alta, rendendo inefficaci gli strumenti di analisi esistenti. Inoltre molti metodi attuali possono calcolare matrici di traffico tra pochi centinaia di end hosts, mentre un data center può averne decine di migliaia. Infine, i metodi esistenti assumono dei pattern di flusso che sono ragionevoli per le reti internet, ma non lo sono per le applicazioni eseguite all'interno della cloud,(es. i jobs MapReduce [13] cambiano significativamente i pattern di traffico). Inoltre si ha un maggior accoppiamento nell'uso della rete, delle risorse di computing e di storage da parte delle applicazioni, di quanto non ve ne sia nel normale traffico di rete.

Data security

La sicurezza dei dati è un'altro importante argomento di ricerca nel cloud computing, in quanto i service providers tipicamente non hanno accesso al sistema fisico di sicurezza dei data centers, che è invece gestito dagli infrastructure providers. Anche per le cloud private, il service provider può solo specificare le impostazioni di sicurezza per via remota, senza avere garanzia che sarà effettivamente implementato. E' compito degli infrastructure provider, in questo contesto, raggiungere i seguenti obiettivi: confidenzialità, ovvero l'accesso e trasferimento sicuro dei dati; tracciabilità, per documentare se i dispositivi di sicurezza siano stati violati o meno. La confidenzialità è di solito perseguita utilizzando protocolli crittografici, mentre la tracciabilità può essere realizzata per mezzo di tecniche di documentazione remota. In un ambiente virtualizzato come le clouds, le VMs possono dinamicamente migrare da una macchina ad un'altra. In questo caso è necessario costruire meccanismi trust in ogni strato dell'architettura. Per esempio, il virtaulization layer deve essere monitorabile utilizzando secure virtual machine monitors. Recenti lavori hanno avuto l'obiettivo di progettare protocolli per il trust establishment and management.

Storage technologies and data management

I software frameworks come MapReduce [13] sono progettati per la computazione distribuita di tasks data-intensive. Questi frameworks tipicamente operano su file systems distribuiti come GFS e HDFS. Questi file systems sono diversi da quelli tradizionali e sono distribuitI in termini di storage structure, access pattern and application programming interface. In particolare, questi sistemi, non implementano l'interfaccia standard POSIX e quindi introducono questioni di compatibilità con applicazioni e file systems legacy.

Novel cloud architectures

Attualmente. La maggior parte delle clouds commerciali sono implementate in grandi data centers e gestiti in maniera centralizzata. Anche se questo tipo di architettura permette un'economia di scala ed elevate possibilità di gestione, presenta dei limiti come il grande consumo di energia e un elevato investimento iniziale per la costruzione dei data centers. Data centers con ridotte dimensioni possono essere più vantaggiosi: un piccolo data center non consuma molta potenza, pertanto non richiede un sistema di raffreddamento potente e costoso. Inoltre, data centers di piccole dimensioni sono più facili da costruire ed è possibile distribuirli meglio da un punto di vista geografico rispetto alle loro controparti maggiori. La geo-diversità è spesso desiderabile in caso di servizi in cui il tempo di risposta è un fattore critico, come applicazioni interattive o di tipo content-delivery. Un altro trend consiste nell'utilizzare risorse condivise dagli utenti per la creazione di data centers distribuiti. Le cloud costruite secondo questa architettura, oppure con architettura ibrida sono molto più economiche e più adatte per applicazioni non-profit come il calcolo scientifico. Tuttavia, questo tipo di soluzioni, solleva nuove problematiche legate alla necessità di gestire risorse eterogenee e caratterizzate da frequenti churn events. Sono inoltre oggetto di studio meccanismi di incentivi per favorire questo tipo di architetture.

2.2.5 Tecnologie correlate

Il cloud computing è spesso confrontato alle seguenti tecnologie, e condivide alcune caratteristiche con ciascuna di queste.

Grid computing: è un paradigma di calcolo distribuito che coordina le risorse di rete per raggiungere obiettivi computazionali comuni. Lo sviluppo del grid computing era originariamente guidato da necessità di supporto al calcolo delle applicazioni scientifiche che sono di solito computazionalmente onerose. Il cloud computing è simile al grid computing nel modo in cui impiega le risorse distribuite per raggiungere gli obiettivi a livello applicativo. Tuttavia il cloud computing introduce la virtualizzazione delle tecnologie a diversi livelli permettendo la condivisione delle risorse e la loro allocazione dinamica.

Utility computing: è il modello di utilizzo delle risorse on-demand. Il cloud computing, inteso come offerta di risorse di calcolo, adotta uno schema di prezzo utility-based. Con l'allocazione dinamica di queste risorse i service providers possono massimizzarne l'utilizzo e minimizzare i costi operativi.

Virtualizzazione: è una tecnologia che astrae i dettagli dell'hardware fisico e fornisce risorse virtualizzate per applicazioni di alto livello. Un server virtuale è comunemente chiamato virtual machine (VM). La virtualizzazione è la base del cloud computing, perché fornisce la capacità di offrire risorse di calcolo da clusters di server e di assegnare dinamicamente le risorse virtuali alle applicazioni sulla base delle loro richieste.

Autonomic computing: l'allocazione dinamica ottima delle risorse può essere vista come una disciplina propria dell'autonomic computing. Il cloud computing, pur esibendo alcune caratteristiche autonomiche come l'approvvigionamento automatico di risorse, ha come obiettivo la diminuzione del costo delle risorse piuttosto che la riduzione della complessità del sistema.

2.2.6 Modello generale di deploy su cloud

Il cloud computing propone un modello di richiesta e di gestione autonoma di un insieme di risorse computazionali. La Fig. 2.3 mostra l'architettura logica di un cloud computing deployment environment. Tipicamente, gli infrastructure providers forniscono un pool di sistemi hardware interconnessi tra loro e ciascuno di questi esegue un software di virtualizzazione come VMWare, Xen

o KVM. La virtualizzazione permette ad ogni sistema hardware di ospitare ed eseguire istanze multiple di macchine virtuali, offrendole come servizi di cloud. Questi servizi sono resi disponibili attraverso interfacce web o API fornite dal provider.

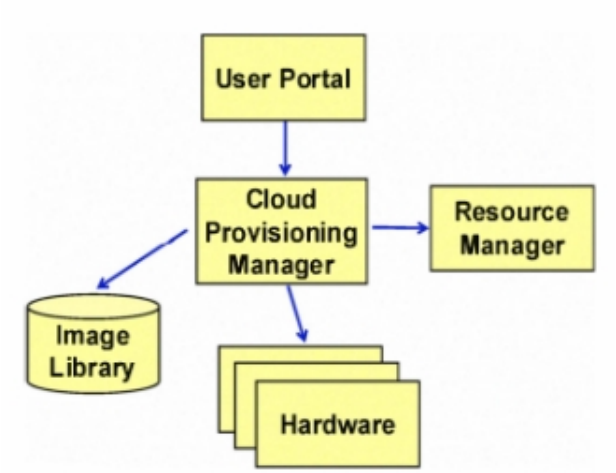

Figura 2.3: Architettura logica di un cloud deploy environment

Le richieste degli utenti vengono gestite dal Provisioning Manager, un sistema che si occupa di interrogare il Resource Manager per individuare le risorse hardware disponibili e in grado di eseguire la macchina virtuale richiesta. Successivamente, produce una copia dell'immagine della macchina virtuale recuperata dalla Image Library, con la quale configura la risorsa hardware selezionata creando una istanza virtuale. Il Provisioning Manager si occupa inoltre di configurare il software eventualmente preinstallato sulla macchina e di notificare all'utilizzatore l'avvenuta inizializzazione VM.
Una immagine è semplicemente la rappresentazione del disco di una macchina virtuale con un certo sistema operativo, mentre un'immagine di macchina virtuale precaricata con applicazioni addizionali viene chiamata Application Image. Generalmente, la descrizione di una immagine è contenuta in due file: il configuration file e l'actual disk image file, di cui il primo rappresenta i metadati associati a caratteristiche quali la posizione del file immagine, il display name e i devices. Le immagini virtuali rappresentano quindi i building blocks per la costruzione delle istanze. Il processo di selezione della macchina fisica più adatta per il deploying di una macchina virtuale è chiamato virtual machine

placement. Durante questa fase, le macchine di uno stesso pool vengono classificate sulla base dei requisiti propri della macchina virtuale richiesta e sulla base dei placement goal. I goal possono essere un basso costo delle operazioni di load balancing tra le macchine o la massimizzazione delle risorse dedicate a un singolo host.

2.3 Conclusioni

In questo capitolo è stato analizzato lo stato dell'arte dei paradigmi di autonomic e cloud computing. Sono state mostrate le necessità che ne animano lo studio e ne sono state presentate le più importanti caratteristiche. Il cloud computing, realizzato grazie alle tecnologie di virtualizzazione, è un paradigma di computazione orientato ai servizi che sta emergendo rapidamente; la chiave per la realizzazione di servizi disponibili e caratterizzati da elevate prestazioni è la capacità di pianificare e gestire dinamicamente le risorse fornite dalla cloud per far fronte alle variazioni della domanda. L'introduzione del cloud computing concorre inoltre ad aumentare la complessità dei sistemi software infatti, la presenza di numerose macchine virtuali in esecuzione su un'infrastruttura fisica sottostante, incrementa la struttura complessiva dei sistemi introducendo nuovi aspetti da gestire. Per questi motivi, il futuro successo del cloud computing dipende dalla possibilità di sviluppare nuove tecnologie basate sul modello autonomico, ovvero sulle capacità di auto-regolazione, che estendano le architetture dei sistemi software permettendo di ridurne la complessità e di gestirne in modo automatico gli strati sottostanti e composti da risorse distribuite. Viceversa, la gestione dinamica delle risorse resa possibile dall'introduzione del cloud computing, comporta un'estensione nella definizione di sistema autonomico. Un sistema autonomico deve ora essere caratterizzato dalla proprietà di auto-approvvigionamento. Questa proprietà è la capacità di aumentare e ridurre l'insieme delle risorse asservite al sistema in modo tale da far fronte alle variazioni dei carichi di lavoro minimizzando l'utilizzo di tali risorse e soddisfacendo i requisiti in termini di performance e di qualità del servizio.

Capitolo 3

Selflet e Cloud

Nel precedente capitolo è stato discusso lo stato dell'arte relativo ai paradigmi di autonomic computing e cloud computing, ne sono state mostrate le relazioni e sono stati descritti i motivi alla base di una loro possibile integrazione. L'obiettivo di questo lavoro di tesi è la realizzazione di un'estensione per un particolare framework autonomico, che ne aggiunga la proprietà di auto-approvvigionamento. Il SelfLet framework, sviluppato nell'ambito del progetto CASCADAS [15] con l'obiettivo fornire un supporto autonomico ad applicazioni distribuite e in grado di adattarsi all'incertezza dell'ambiente in cui si trovano modificando il proprio comportamento, è un'implementazione in linguaggio Java del modello ACE [39]. In questo capitolo verrà esaminata l'architettura del framework con particolare attenzione al suo funzionamento e ai suoi componenti. Sarà inoltre presentata una particolare istanza di servizio di cloud computing: Amazon Web Services. AWS è l'insieme di servizi per il cloud fornito da Amazon ed offre la possibilità di richiedere e rilasciare macchine virtuali secondo il modello di servizio IaaS descritto nella sezione 2.2.1.

3.1 Selflet framework

3.1.1 Modello concettuale

Una SelfLet è una parte di software autosufficiente situata in una rete logica o fisica, dove interagisce e comunica con altre SelfLets [34]. Il modello delle SelfLet è stato sviluppato tenendo presenti le principali proprietà descritte

da IBM nella sua definizione di sistema autonomico [22]. Ogni SelflLet è un componente autonomico in grado di realizzare il suo obiettivo attraverso l'esecuzione di behaviors, di interagire con altre SelfLets per ottenere il loro aiuto nel raggiungimento dell'obiettivo (per esempio condividendo il proprio carico di lavoro con SelfLets simili) o offrendo loro i propri servizi, e di monitorare il proprio stato e quello dei propri neighbors per individuare eventuali problemi e attivarsi nell'esecuzione di una risoluzione. E' capace inoltre di applicare delle politiche autonomiche per reagire a determinati eventi individuati nella fase di monitoring. Le azioni di reazione possono essere la ricerca di un nuovo SelfLet neighbor per collaborare con esso, la modifica dei propri obiettivi, oppure il cambiamento delle modalità con cui con cui raggiungere questi obiettivi. La figura 3.1 mostra il modello concettuale di una Selflet.

Attributes: gli attributi descrivono le proprietà generali di una SelfLet, come la sua location, il tipo e il gruppo di appartenenza

Services: I servizi sono descrizioni di alto livello delle operazioni da compiere per il raggiungimento degli obiettivi. Possono essere di due tipi. I servizi achievable, di cui la SelfLet conosce i dettagli implementativi, possono essere eseguiti localmente ed essere eventualmente offerti ai neighbors che li richiedono. I servizi di tipo needed sono quelli di cui una SelfLet può aver bisogno durante l'esecuzione per il raggiungimento dei propri obiettivi; in questo caso la SelfLet genera un messaggio di negoziazione verso i propri neighbors per la richiesta del servizio. Ogni servizio può avere dei parametri di ingresso e restituisce un risultato alla fine della computazione e può essere offerto o richiesto in due differenti modalità: Do o Teach. Quando un Servizio è richiesto in modalità Do, la SeflLet che necessita del servizio chiede ad un'altra SelfLet di eseguirlo localmente e di restituirne il risultato. Nel caso di una richiesta di tipo Teach la SelfLet richiede i dettagli implementativi del servizio perché possa eseguirlo. Come accennato, i servizi sono descrizioni di alto livello, pertanto non specificano esplicitamente come realizzarne il completamento, così come non descrivono un risultato atteso; in generale i servizi sono composizioni di sottoservizi più elementari rappresentano i sotto obiettivi da portare a termine per il suo completamento.

Behaviors: un behavior è l'implementazione di un servizio e descrive come

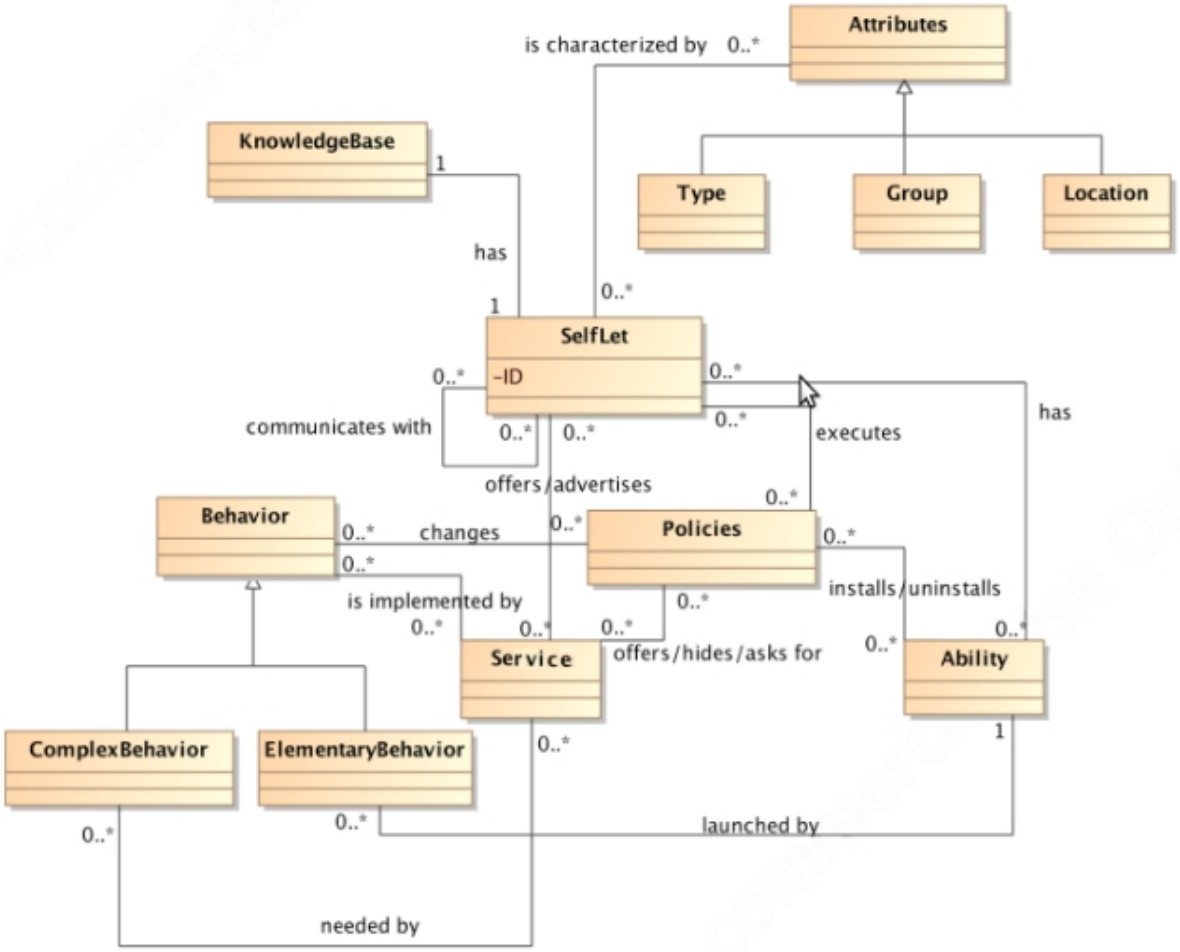

Figura 3.1: Modello concettuale di una SelfLet

realizzarlo. Ogni servizio, perché possa essere eseguito, deve prevedere almeno un behavior che può eventualmente essere definito su una SelfLet differente da quella che necessità del servizio (è il caso dei servizi needed). Un behavior è definito da un diagramma UML StateCharts. Dallo stato iniziale, la SelfLet si muove attraverso gli stati eseguendo una computazione valida per la realizzazione del servizio implementato. Le transazioni tra gli stati, rappresentate da archi, possono essere associate a condizioni espresse rispetto alle variabili contenute nella knowledge base (descritta nel seguito). Queste condizioni sono espresse all'interno di file XML. Uno stato rappresenta un sotto obiettivo, e quindi un sotto servizio, che necessita di essere completato per il proseguimento della computazione. Ogni stato è caratterizzato da un nome ed è associato a

un'azione. Un'azione è una porzione di codice Java realizzata per mezzo della tecnologia Javassist [27] ed è preposta alle operazioni di invocazione e gestione del risultato dell'effettiva esecuzione di un servizio. Esistono due tipologie di behavior. Un elementary behavior rappresenta un obiettivo di basso livello, un'operazione atomica la cui implementazione è contenuta in una ability (come descritto nel seguito) e il cui diagramma ha una struttura semplice come riportato in figura 3.2.

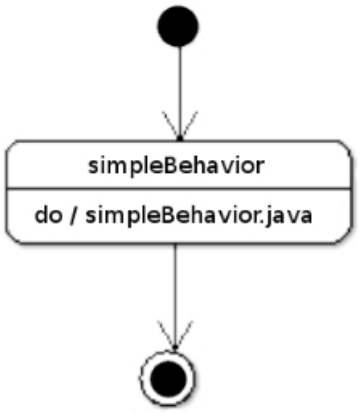

Figura 3.2: Esempio di elementary behavior

Un complex behavior è un behavior il cui diagramma prevede l'attraversamento di più stati ed è quindi necessario completare dei sottoservizi per raggiungere lo stato finale. Un esempio di complex behavior è il Default Behavior. Quest'ultimo è tipicamente strutturato come mostra la figura 3.3 e implementa il servizio con cui una SelfLet inizia la propria esecuzione. L'esecuzione del servizio prevede, in condizioni normali, un ciclo di cui uno stato è uno stato di attesa, utile per dare il tempo alla SelfLet di individuare eventuali errori nel corso dell'esecuzione. In caso di errore, la SelfLet si muove nello stato corrispondente e può attuare delle politiche di recovery.

Abilities: le abilities sono programmi dedicati all'esecuzione di uno specifico task. Vengono invocate dalle actions associate agli elementary behaviors. e sono contenute all'interno di file JAR. Le abilities vengono installate a run-time dalle SelflLets per mezzo del framework OSGi [20]. In questo modo è possibile l'apprendimento di nuovi servizi da parte di SelfLet che ne effettuano la richiesta in modalità Teach.

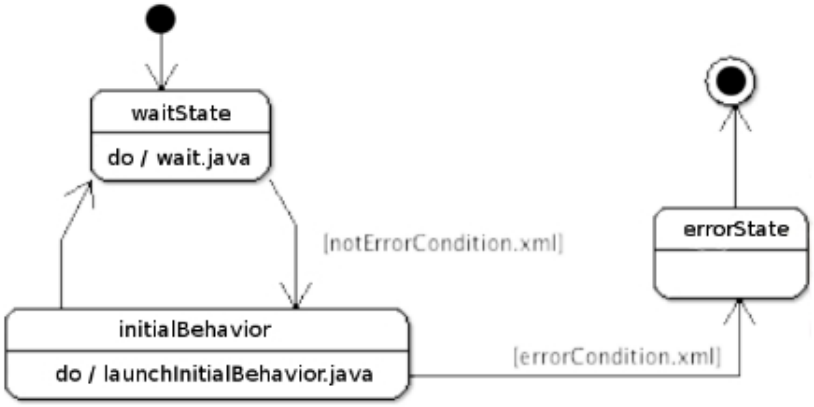

Figura 3.3: Esempio di complex behavior

Policies: le politiche autonomiche guidano il comportamento delle SelfLet specificando come reagire al verificarsi di determinati eventi e sono composte da regole. Ogni regola deve avere un nome, può avere degli attributi, ed è composta da un LHS (Left Hand Side) e un RHS (Right Hand Side). l'LHS contiene una condizione mentre il RHS contiene un'azione. Quando il ciclo di controllo delle politiche autonomiche verifica che una certa condizione è verificata, deve essere eseguita la rispettiva azione. Il tipo di operazioni che possono essere eseguite sono in generale di due tipi. Le default policies sono comuni a tutte le SelfLets e sono indipendenti dagli obiettivi propri dell'applicazione che questa sta eseguendo. Tra queste politiche troviamo la verifica della condizione di necessità dell'esecuzione di un certo servizio da parte di un behavior, per la quale viene attuato il meccanismo di richiesta del servizio. Il secondo tipo di politiche è invece dipendente dal tipo di applicazione e sono relative ai requisiti del singolo dominio applicativo. Le politiche sono implementate utilizzando Drools [17] un business rule management system (BRMS) open source e integrabile con il linguaggio Java.

3.1.2 Architettura interna

Una SelfLet è un sistema complesso, formato da diversi componenti. La sua architettura interna e le relazioni tra i componenti sono mostrati in figura 3.4.

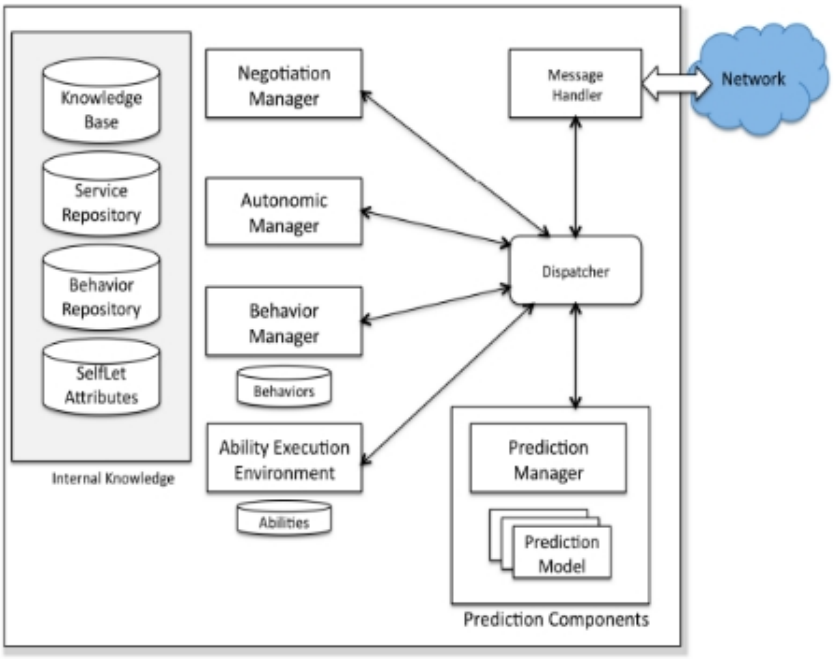

Figura 3.4: Architettura interna di una SelfLet

Internal knowledge

Ogni SelfLet ha un insieme di conoscenze relative al proprio stato interno, a quello dei propri neighbors, oppure proprie dell'applicazione che la SelfLet implementa. Queste informazioni sono memorizzate all'interno della Internal knowldege, una memoria di tipo associativo che permette di settare, aggiornare e cancellare variabili come coppie chiave-valore. L'Internal knowledge è divisa in quattro tipi, concernenti differenti aspetti della SelfLet:

- La type knowledge è usata per caratterizzare la SelfLet con le sue proprietà distintive.
- La service knowledge contiene le informazioni relative alla conoscenza sui servizi e alle modalità con qui questi sono offerti ai neighbors.
- La behavior knowledge che in modo simile alla service knowledge contiene le informazioni sui behaviors e sui costi della loro esecuzione.
- La knowledge base, una base di conoscenza generale utilizzata per memorizzare le informazioni dipendenti dall'applicazione implementata e i risultati prodotti dall'esecuzione dei servizi.

E' prevista la possibilità di condividere parzialmente o totalmente la conoscenza tra più SelfLet. A questo scopo si deve l'integrazione con Lime (Linda In a Mobile Environment) [25]. Questo middleware implementa un tuple-space distribuito che fornisce un modello di comunicazione con il quale le SelfLet possono comunicare e coordinare in maniera indipendente le operazioni sulla memoria condivisa senza la necessità di conoscere l'identità delle altre SelfLets.

Autonomic Manager

Il componente centrale dell'architettura delle SelfLets è l'Autonomic Manager, responsabile dell'evoluzione di una SelfLet nel suo ciclo di vita. L'Autonomic Manager implementa il ciclo di controllo autonomico MAPE, previsto dalla vision dell'autonomic computing (descritta nella sezione 2.1). Questo processo di controllo si occupa di guidare il comportamento del sistema in accordo con le autonomic policies installate. Attraverso il Drools Manager, è possibile iscrivere la SelfLet a categorie di eventi attivandone la risposta in caso si verifichino le relative condizioni. Il componente che si occupa di eseguire le azioni previste dalle politiche è invece l'Autonomic Attuator, che permette di modificare le modalità di esecuzione dei servizi, le modalità con cui questi vengono offerti, ecc.

Behavior Manager

Il Behavior Manager è il componente responsabile dell'esecuzione dei servizi da parte della SelfLet. In ogni momento, dovrebbe essere possibile l'esecuzione di uno o più behaviors. Questo può accadere per due ragioni: l'attivazione di un sottoservizio da parte di un behavior corrente, a cui segue l'esecuzione di un nuovo behavior oppure a causa di una richiesta di esecuzione di un certo servizio da parte di un neighbor. Quando un behavior viene attivato, viene creata all'interno della knowledge base una nuova variabile per il salvataggio del risultato del servizio.

Negotiation Manager

Le SelfLets possono comunicare tra loro attraverso lo scambio di messaggi asincroni secondo il paradigma publish/subscribe [38]. I messaggi possono essere

distinti in due categorie: i messaggi application-level e quelli negotiation-level. I primi dipendono dalla specifica applicazione che le SelfLets stanno implementando. I messaggi negotiation-level sono usati dalle SelfLets per scambiarsi informazioni sui servizi. Il Negotiation Manager è il componente che si occupa di gestire gli aspetti relativi alla comunicazione con gli altri nodi della rete. Attraverso il Message Handler è possibile intercettare i messaggi e recapitare i messaggi. Il meccanismo di comunicazione è mediato da REDS [24]. Sviluppato dal Politecnico di Milano, questo middleware implementa un dispatcher distribuito composto da più brokers. Uno dei vantaggi di questo approccio è che le SelfLets non hanno bisogno di conoscere i dettagli di rete dei propri neighbors, grazie al paradigma publish/subscribe.

Ability Execution Environment

L'Ability Execution Environment è incaricato dei processi di installazione, disinstallazione, salvataggio ed esecuzione delle abilities. Le abilities sono distribuite sotto forma di bundles OSGi [20]. Ogni bundle è un JAR file contenente le classi compilate e le informazioni necessarie alla loro esecuzione. Le abilities vengono invocate all'interno delle actions e i metodi da esse esposti devono essere specificati nella base di conoscenza della SelfLet.

Dispatcher

Il SelfLet framework implementa un sottosistema di eventi interni, attraverso il quale è possibile la comunicazione tra tutti i componenti che costituiscono una SelfLet. L'architettura usata segue il paradigma publish/subscribe e si divide in tre tipi di elementi: producers, listeners e dispatchers. Ogni producer specifica il tipo di eventi prodotti, i listeners dichiarano il tipo di eventi a cui sono interessati, mentre i dispatchers si occupano di inoltrare gli eventi ai listeners in accordo alle loro dichiarazioni di interesse.

3.1.3 Ciclo di sviluppo di un sistema basato su Selflets

L'approccio suggerito nel progetto di un sistema basato su Selflets [39] si divide nei seguenti passi: Dopo un'analisi preliminare dell'applicazione che si vuole sviluppare, identificando gli obiettivi e le caratteristiche principali, è necessario individuare i diversi tipi di Selflet che devono essere utilizzate. Definita

la tipologia dei componenti, la fase di design continua con la specifica dei servizi di alto livello, ognuno specificato dal proprio default behavior. Questi servizi rappresentano obiettivi di alto livello e come tali dovrebbero essere eseguiti attraverso l'esecuzione di servizi intermedi, rappresentati dagli stati del diagramma dei behaviors. Ogni sottoservizio è specificato da un default behavior e può a sua volta essere implementato da altri sottoservizi che compongono così una gerarchia di obiettivi. E' necessario individuare i servizi della gerarchia e le transazioni nei diagrammi dei behaviors. Le transazioni sono rappresentate dagli archi che collegano gli stati nel diagramma, sui quali è possibile specificare le condizioni per l'attraversamento, usando specifici files XML. E' importante che ogni servizio dell'applicazione sia implementato da almeno un behavior. Non rispettare questo vincolo può causare il fallimento della computazione della selflet. Se l'implementazione del behavior è locale al nodo che conosce un dato servizio, la sua esecuzione può essere eseguita localmente al nodo; può altrimenti avvenire in remoto. L'introduzione di nuovi servizi avviene fino a quando ogni sottoservizio che si sta definendo rappresenta un compito specifico e indivisibile, a questo punto può essere implementata da un'Abiliy. Come spiegato nel precedente capitolo, le Abilities sono incapsulate negli Elementary behaviors. La creazione di un behavior è accompagnata dalla specifica di una Action associata a ogni stato del diagramma. Le Actions sono usate solitamente per effettuare una qualche computazione che precede o segue l'esecuzione dell'Ability per il raggiungimento dell'obiettivo (es. setting di variabili nella knowledge base, effettuare operazioni sul valore di output, ecc.). Nonostante all'interno di una Action sia possibile inserire codice Java completo, è concettualmente errato inserire il codice relativo al completamento degli obiettivi all'interno delle Actions. Una volta definite le Abilities il design di una selflet è completo eccetto per la sua parte autonomica. E' quindi necessario specificare le policies, utilizzate per specificare come il componente reagisce a particolari eventi o a situazioni critiche come il verificarsi di errori o la mancanza dei servizi cercati.

3.2 Amazon Web Services

Amazon Web Services (AWS) [3] sono un insieme di servizi che permettono di utilizzare l'infrastruttura per il cloud computing [cap. 6] gestita da Amazon.

I componenti principali di questi servizi sono quelli che offrono le funzionalità di storage, computing, messaging e datasets. Questi forniscono una infrastruttura virtuale i cui componenti sono utilizzabili on-demand, con un modello di costo pay-for-use, e il cui punto di forza è l'elasticità ovvero la possibilità di scaling rispetto ai requisiti dell'utilizzatore. E' quindi possibile un modello di sviluppo delle applicazioni nel quale lo strato software utilizza lo strato dei servizi per rimodulare dinamicamente l'infrastruttura virtuale sottostante, secondo le proprie necessità. La figura 3.5 mostra questo modello, in cui lo strato applicativo è costruito sugli Amazon Web Services. Amazon fornisce delle interfacce basate sugli standard SOAP e REST per l'interazione con ogni servizio e dei Command Line Tools per la gestione delle macchine virtuali. Sono inoltre disponibili delle librerie per diversi linguaggi di programmazione tra i quali Java, Ruby, Python, PHP e C.

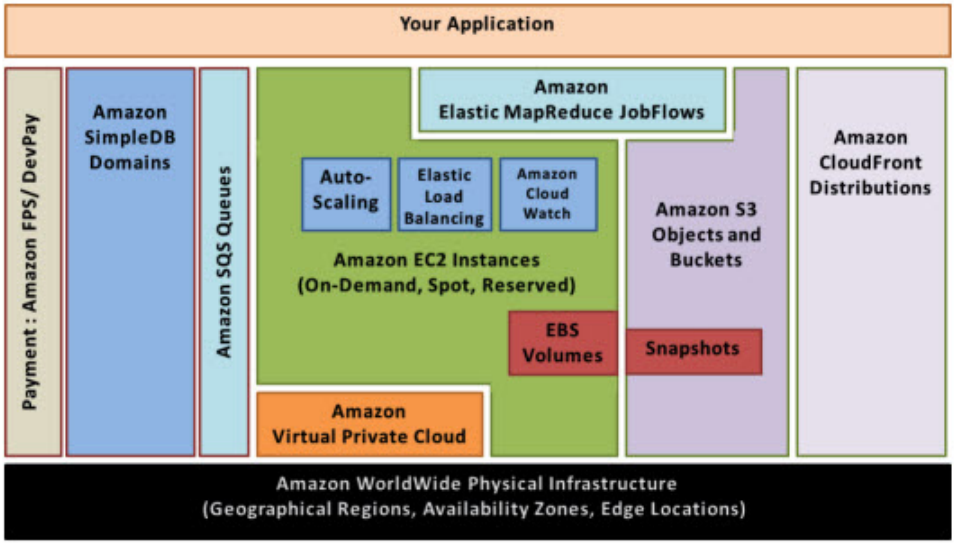

Figura 3.5: Architettura Amazon Web Services

3.2.1 Amazon Simple Storage Service

Simple Storage Service (S3) [6] è un servizio di storage e di recupero dei dati. I dati possono essere di ogni tipo ed è possibile salvarli o accedervi da qualunque dispositivo attraverso la rete internet. Il numero degli oggetti che si possono immagazzinare è illimitato e la dimensione di ognuno di essi può variare da 1 byte fino a 5 GB. Al momento della creazione dei buckets, oggetti simili alle directory di un normale sistema operativo che consentono il recupero dei

dati per mezzo di una chiave definita dall'utilizzatore, è possibile scegliere la storage location area in cui salvare i dati, scegliendo tra Stati Uniti, Europa e Singapore. Per ogni oggetto,il cui nome è un path name URI, è possibile specificare le restrizioni di accesso o renderlo disponibile per il download attraverso il protocollo BitTorrent. Il Service Level Agreement (SLA) relativo alla disponibilità dei dati è del 99,9% su scala mensile.

3.2.2 Amazon Simple Queue Service

Simple Queue Service (SQS) [9] è un servizio di messaging basato sull'infrastruttura di Amazon che permette di spedire e recuperare i messaggi da qualsiasi dispositivo attraverso REST, basato su richieste HTTP. E' possibile creare un numero arbitrario di code e spedire una quantità illimitata di messaggi. I messaggi vengono salvati da Amazon su più server, garantendo la ridondanza e l'affidabilità del sistema. Ogni messaggio può contenere fino a 64 KB di testo in ogni formato, e per ogni coda è possibile stabilire un timeout di visibilità per il controllo degli accessi da parte dei readers; in questo modo, ogni volta che un'applicazione legge un messaggio da una coda, questo non sarà più visibile agli altri readers fino allo scadere del timeout stabilito. Dopo questo intervallo, il messaggio può tornare ad essere visibile nella coda, per essere eventualmente letto da un altro processo reader. SQS può essere utilizzato con altri servizi AWS per lo sviluppo di sistemi che presentino un basso accoppiamento tra i componenti. Per esempio un sistema che fa uso di macchine virtuali fornite dal servizio EC2, può permettere la comunicazione delle istanze attraverso dei messaggi per il coordinamento del flusso di lavoro. L'accesso ai messaggi è inoltre regolato da un meccanismo di autenticazione.

3.2.3 Amazon Simple DB

SimpleDB (SDB) [8] è un servizio per il salvataggio, l'analisi e l'interrogazione per mezzo di query di insiemi di dati. Questi dati non sono memorizzati in classici database relazionali, ma piuttosto in schemi poco strutturati sui quali è possibile memorizzare e recuperare i valori per mezzo di chiavi. Ogni insieme di valori è un item che necessita di un nome univoco, può essere partizionato in domini, e può contenere fino a 256 coppie chiave - valore. E' possibile eseguire interrogazioni sui datasets rispetto a ogni dominio, anche se non sono

ancora supportate interrogazioni rispetto alle intersezioni tra i domini. Una caratteristica dovuta alla mancanza di vincoli sulla struttura dello schema è la possibilità di inserire dati on-the-fly: in qualunque momento è possibile aggiungere nuove colonne o chiavi dinamicamente. La gestione degli schemi è più semplice rispetto a quella necessaria per un database relazionale in quanto l'utilizzatore non ha la necessità di installazioni o configurazioni; Amazon si prendere carico dei compiti di carattere amministrativo, come l'indicizzazione automatica del database, e si occupa di rendere sempre disponibili i dati. Anche questo servizio ha la possibilità di scalare dinamicamente rispetto ai requisiti della domanda da parte dell'utilizzatore.

3.2.4 Amazon Elastic Compute Cloud

Elastic Compute Cloud (EC2) [4] è un servizio per la richiesta e la gestione di macchine virtuali che permette di adattare l'infrastruttura su cui viene eseguito un certo sistema ai suoi requisiti. E' possibile aumentarne la capacità computazionale lanciando nuove istanze di macchine virtuali oppure ridurla terminandole. Le istanze sono basate su Linux o altri sistemi operativi e sono in grado di eseguire qualsiasi software o applicazione. Per rendere disponibili le istanze, EC2 utilizza lo Xen Virtualization Engine [31]. Amazon fornisce un insieme di Amazon machine images (AMI) che svolgono la funzione di templates per le macchine virtuali da lanciare. Dopo la creazione e l'avvio di una istanza, è possibile caricare il software o modificarne la configurazione, e salvarne lo stato del sistema in una nuova immagine, utilizzabile per creare una copia identica della nuova macchina virtuale da lanciare in qualsiasi momento. E' inoltre possibile specificare i permessi di accesso per ogni macchina e la sede di esecuzione.

Amazon Machine Images

Le Amazon Machine Images sono dei packaged server environments, basati su linux o altri sitemi operativi, capaci di eseguire qualsiasi software o applicazione. Sono l'elemento centrale del servizio di cloud computing fornito da EC2. Esistono tre tipi di immagini:

Private: queste immagini sono create da un utente e sono di sua proprietà. E' possibile permettere ad altri utenti di utilizzare queste immagini.

Public: queste immagini sono create dagli utenti e sono rilasciate sulla AWS community, così che ogni sviluppatore possa usarle per lanciare le proprie istanze. E' disponibile una lista delle immagini pubbliche.

Paid: E' possibile creare immagini e renderle disponibili attraverso una forma di pagamento.

Le immagini possono essere salvate utilizzando il servizio Amazon S3. Registrando un'immagine, le viene assegnato un ID univoco che può essere utilizzato per identificarla e usarla per lanciare una nuova istanza. E' possibile creare un'immagine in diversi modi: per esempio utilizzando una immagine public come base per la creazione di una configurazione personale, attraverso i seguenti passi.

1. Lanciare una istanza da una AMI esistente, con una coppia di chiavi SSH
2. Collegarsi in SSH all'istanza lanciata
3. Modificare l'istanza installando software o modificandone la configurazione
4. Creare un bundle dell'istanza in esecuzione all'interno di una nuova AMI
5. Caricare il bundle attraverso il servizio di storage S3
6. Registrare la nuova immagine al servizio EC2 ottenendo un ID di riconoscimento
7. Lanciare una nuova istanza dall'immagine creata ed eventualmente ripetere le operazioni di configurazione e bundling

Un'altra possibilità di creare una nuova AMI è l'utilizzo di script disponibili presso la EC2 community. Questi script consentono di creare un'immagine da zero.

Instances

Le istanze sono le macchine virtuali che vengono lanciate ed eseguite attraverso l'uso delle AMI. Una volta avviata un'istanza è possibile vederne i dettagli, terminarla e gestirne il ciclo di vita utilizzando le API o le interfacce web

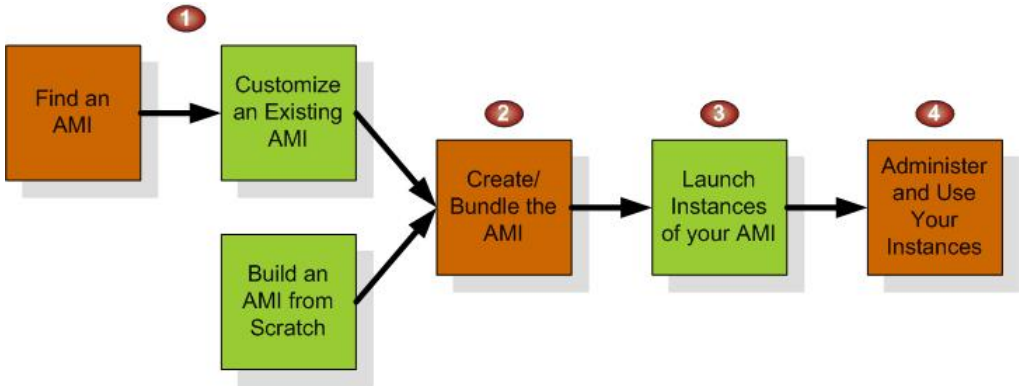

Figura 3.6: processo di lancio di una EC2 instance

fornite da Amazon. E' possibile accedere come utente root a ciascuna istanza ed interagire con essa come si farebbe con qualsiasi macchina. Amazon fornisce inoltre una console per il monitoraggio e il controllo manuale del pool di istanze attive. Il servizio permette di scegliere tra varie tipologie di istanze. Oltre alla configurazione iniziale dal punto di vista software definito nelle AMI, EC2 consente di selezionare una configurazione di memoria, CPU, capacità di storage e architettura a 32-bit o 64-bit. Le configurazioni si dividono in categorie, e vengono valutate secondo la metrica delle EC2 compute units (ECU). Ogni ECU fornisce l'equivalente, in termini di capacità computazionale, di un processore 2007 Opteron o 2007 Intel Xeon da 1.0-1.2 GHz.

Standard instances

- Small Instance (Default): 1.7 GB of memory, 1 virtual core with 1 ECU), 160 GB of local instance storage, 32-bit platform
- Large Instance: 7.5 GB of memory, 2 virtual cores with 2 ECU each, 850 GB of local instance storage, 64-bit platform
- Extra Large Instance: 15 GB of memory, 4 virtual cores with 2 ECU each 1690 GB of local instance storage, 64-bit platform

Micro Instances

- Micro Instance 613 MB of memory, up to 2 ECUs (for short periodic bursts),EBS storage only, 32-bit or 64-bit platform

High-Memory Instances

- High-Memory Extra Large Instance 17.1 GB memory, 2 virtual cores with 3.25 ECU each, 420 GB of local instance storage, 64-bit platform
- High-Memory Double Extra Large Instance 34.2 GB of memory, 4 virtual cores with 3.25 ECU each, 850 GB of local instance storage, 64-bit platform
- High-Memory Quadruple Extra Large Instance 68.4 GB of memory, 8 virtual cores with 3.25 ECU each, 1690 GB of local instance storage, 64-bit platform

High-CPU Instances

- High-CPU Medium Instance 1.7 GB of memory, 2 virtual cores with 2.5 ECU each, 350 GB of local instance storage, 32-bit platform
- High-CPU Extra Large Instance 7 GB of memory, 8 virtual cores with 2.5 ECU each 1690 GB of local instance storage, 64-bit platform

Cluster Compute Instances

- Cluster Compute Quadruple Extra Large 23 GB memory, 33.5 ECU, 1690 GB of local instance storage, 64-bit platform, 10 Gigabit Ethernet

Security groups

Ogni istanza lanciata nel EC2 environment può eseguire all'interno di un security group. Ogni security group definisce delle regole di firewall che specificano le restrizioni sull'accesso alle istanze del gruppo. E' possibile concedere o limitare gli accessi rispetto agli indirizzi IP o usare regole di classless inter-domain routing (CIDR) che consentono di specificare l'insieme dei port e i protocolli permessi per la comunicazione. Al momento del lancio di un'istanza è possibile specificarne il security group di appartenenza in modo che il suo accesso sia automaticamente regolato secondo le politiche definite per il gruppo.

Security keypairs

Al lancio di ogni istanza è necessario specificare una coppia di chiavi di sicurezza (pubblica/privata) SSH. Le chiavi sono necessarie per il controllo delle istanze avviate. Il servizio EC2 associa la chiave pubblica all'istanza che si sta

lanciando, e rende possibile accedervi per mezzo della corrispondente chiave privata. Queste chiavi di sicurezza sono differenti dalla AWS services access key e dalla security key (associate all'account per l'utilizzo del servizio), che sono invece usate per identificare l'utente che inoltra le richieste per l'utilizzo dei web services. Le security keys hanno lo scopo di rendere sicuro l'accesso alle macchine virtuali create senza che si renda necessario un meccanismo di password.

Availability zones

Amazon EC2 prevede la possibilità di collocare le istanze lanciate in diverse sedi di esecuzione. Il servizio è basato su un'infrastruttura fisica composta da molti data centers in diverse locations. Queste locations sono costituite di regioni geografiche e Availability Zones. Le regioni, che consistono in una o più Availability Zones, sono data centers geograficamente distinti, mentre le Availability Zones sono ambienti di computazione isolati in modo da garantire una potenziale immunità rispetto ai gusti. Queste ultime offrono una connettività di rete economica e a bassa latenza verso le altre zone della stessa regione. Con il lancio di istanze in diverse Availability Zone, è possibile proteggere le applicazioni dalle failures avvenute in una singola zona.

Elastic IP addresses

All'avvio di ogni istanza viene automaticamente assegnato un indirizzo IP pubblico ed uno privato. Mentre l'indirizzo pubblico può essere utilizzato per l'interazione con la macchina virtuale attraverso internet, quello privato è utilizzato per la comunicazione tra le istanze di una stessa regione. L'avvio e la terminazione delle istanze dovuta al bilanciamento della virtual infrustructure per incontrare i requisiti dell'applicazione, porta necessariamente alla variazione dell'insieme degli indirizzi pubblici nel corso della vita dell'applicazione. Se si utilizza un DNS dinamico per mappare l'indirizzo pubblico con un nome, eventuali cambiamenti in questa associazione possono impiegare fino a 24 ore per essere propagati attraverso Internet. EC2 introduce il concetto di IP elastico per risolvere questo problema. Ogni elastic IP address è un indirizzo statico associato all'utente del servizio piuttosto che a una specifica istanza. Il controllo di questo tipo di indirizzo è permanente fino a quando non

si scelga esplicitamente di rilasciarlo. In questo modo è possibile, lanciando nuove istanze e mappandole all'IP statico, rispondere alle failures del servizio mantenendolo attivo agli utilizzatori.

Elastic Block Store (EBS)

EBS è un servizio di storage e fornisce alle istanze dei volumi a blocchi che persistono indipendentemente dalla vita delle macchine virtuali. Il servizio permette di agganciare e sganciare i volumi dalle diverse istanze e di crearne degli snapshot, utilizzati per ricrearli successivamente in modo analogo a quanto avviene per le immagini delle istanze. Uno snapshot rappresenta lo stato di un volume in un certo istante di tempo, è possibile salvarlo attraverso il servizio S3, e può essere utilizzato come backup o come base di partenza per la creazione di un nuovo volume.. Ogni istanza può avere a disposizione diversi volumi ed ogni volume è associato a un'Availability zone e deve essere agganciato ad istanze della stessa zona.

Load Balancing

E' possibile attivare un servizio di elastic load balancig che permette di ridistribuire automaticamente il carico in ingresso all'applicazione tra le istanze attive. Il servizio è in grado di individuare le istanze guaste e si occupa di ridirigere il traffico verso un'istanza funzionante in attesa delle operazioni di recovery. E' possibile abilitare questa funzionalità su una o più Availability zones, e specificare delle metriche operazionali in modo da rispettare requisiti di costo o latenza.

Auto Scaling

EC2 prevede un servizio di auto scaling che permette di regolare automaticamente la capacità computazionale della virtual infrastructure attivando o spegnendo le istanze, in accordo alle politiche definite dall'utilizzatore. In questo modo è possibile delegare le operazioni operazioni di analisi, pianificazione ed esecuzione senza che l'applicazione si debba occupare esplicitamente di auto dimensionarsi.

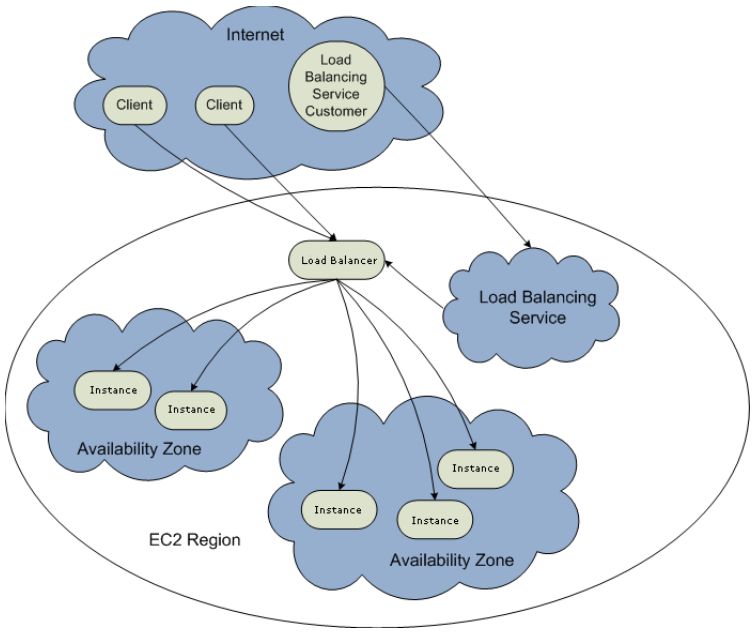

Figura 3.7: Servizio di load balancing

Cloud Watch

Amazon CloudWatch è uno strumento di gestione che raccoglie i dati dagli altri servizi di AWS e processa queste informazioni in metriche leggibili, quasi in tempo reale. Tali metriche includono, per esempio, l'utilizzo della CPU, della rete o del carico delle operazioni di lettura e scrittura su disco.

3.3 Selflet e Cloud: come integrarli

Nelle sezioni precedenti sono stati descritti il SelfLet framework e il servizio di cloud computing di Amazon. Obiettivo di questa tesi è estendere il framework con la proprietà di auto-approvvigionamento. Un sistema basato su SelfLets è composto da una rete di nodi che collaborano tra loro per il raggiungimento degli obiettivi; le risorse di ogni nodo sono l'insieme dei servizi messi a disposizione dai suoi vicini e l'hardware su cui il nodo viene eseguito. A questo proposito il livello di servizio IaaS messo a disposizione da EC2 per l'allocazione dinamica di macchine virtuali permette di gestire opportunamente le risorse asservite ad una SelfLet. E' infatti possibile richiedere una macchina virtuale e attivare su questa una nuova SelfLet, aumentando così il numero dei nodi che offrono un determinato servizio. In questo modo un sistema ha la possibilità di adattarsi alle variazioni dei carichi di lavoro sui nodi della rete. Facendo

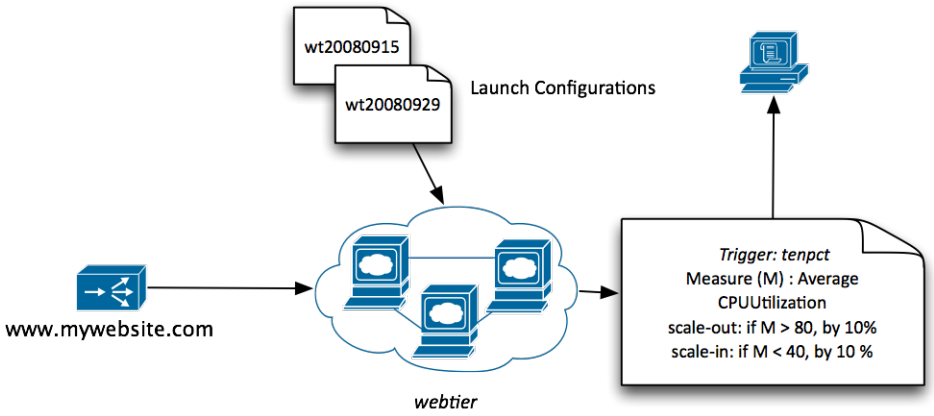

Figura 3.8: Servizio di auto scaling

riferimento al paradigma autonomico, le operazioni di creazione di nuovi nodi devono essere trasparenti all'effettiva esecuzione dei servizi riguardanti le funzionalità implementate dal sistema distribuito, mentre la gestione di questi nodi deve poter essere guidata da politiche di alto livello. Secondo le linee guide appena descritte, dovrà essere sviluppata un'architettura che permetta a un sistema basato su SelfLets di gestire la complessità inerente la gestione di risorse costituite dagli stessi nodi della rete. Dovrà essere inoltre sviluppato un modello di interazione tra i nodi affinché la corretta gestione dell'insieme di queste risorse emerga dal principio di località che governa le decisioni prese dalle singole SelfLets.

3.4 Conclusioni

In questo capitolo è stato descritto il SelflLet framework, il suo modello concettuale, i sottosistemi che ne costituiscono ogni componente autonomico e la natura delle loro interazioni. Il framework è open source e attualmente è ancora in fase di sviluppo, alcune funzionalità previste in fase di progetto non sono ancora state implementate, mentre altre possono essere oggetto di miglioramenti. Sono inoltre possibili molteplici estensioni dal punto di vista funzionale. Nel corso di questo lavoro di tesi, alcune buone proprietà e alcuni limiti dovuti allo stato dell'attuale implementazione hanno influito sulle scelte di progetto e sull'effettiva realizzazione; verranno descritte nel seguito le modifiche apportate al framework nel corso del processo di sviluppo. Suc-

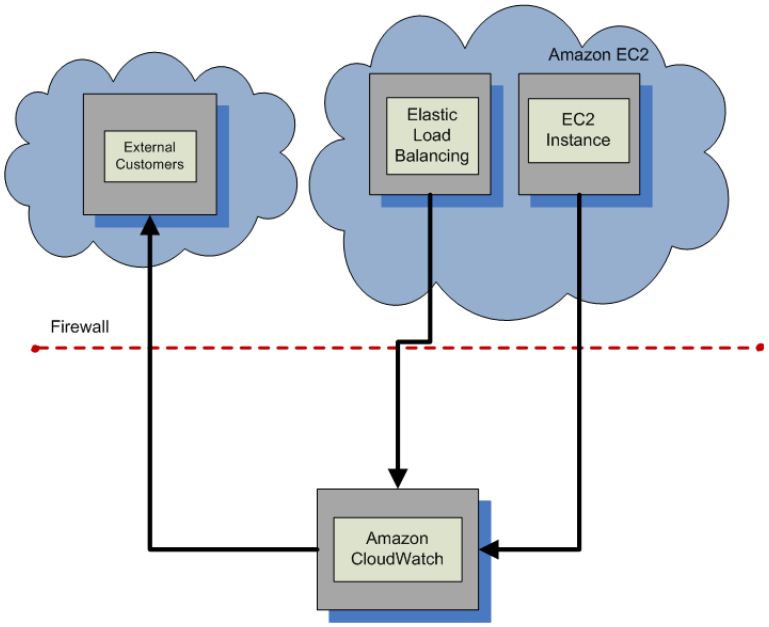

Figura 3.9: Schema concettuale del servizio CloudWatch

cessivamente è stato introdotto l'ambiente di cloud computing proposto ad Amazon, attraverso una panoramica dei principali servizi offerti ed è stato approfondito il servizio EC2, che permette l'allocazione dinamica di macchine virtuali. Si sono esaminate le modalità di interazione con il servizio, le sue principali caratteristiche e il modello di infrastruttura virtuale su cui è possibile costruire le applicazioni. Infine sono state elencate le caratteristiche desiderabili che motivano l'integrazione del paradigma dell'autonomic computing con il modello computazionale reso possibile dalle tecnologie per il cloud. La prima fase di questo lavoro di tesi è finalizzato allo studio approfondito delle tecnologie presentate, e alla realizzazione di semplici casi di studio per verificarne le caratteristiche. Nel Capitolo 4 verrà presentata un'analisi dei processi di sviluppo consigliati per la realizzazione di sistemi costruiti per sfruttare le possibilità offerte da queste tecnologie. Sarà inoltre descritta la fase di design per il sistema che si vuole realizzare, l'estensione del Selflet Framework per l'espansione di una rete autonomica all'interno di un ambiente di cloud.

Capitolo 4

Analisi del problema e definizione della soluzione

4.1 Complessità del ciclo di sviluppo

La fase di studio delle tecnologie introdotte al Capitolo 3, è stata seguita dall'analisi del problema che si vuole affrontare. L'obiettivo di estendere il funzionamento del Selflet Framework con le possibilità offerte dai servizi di cloud computing di Amazon presenta diversi livelli di complessità. Il primo tra questi è la necessità di adottare un approccio nel design adatto alle caratteristiche delle SelfLets. Il modello proposto dal SelfLet Framework [34] è quello di una rete di nodi nella quale ogni componente, una SelfLet appunto, è in grado di svolgere e offrire servizi per il raggiungimento degli obiettivi. La rete può essere composta da nodi aventi una struttura omogenea (es. caratterizzati dagli stessi servizi implementati dagli stessi behavior) oppure aventi diverse capacità, mentre il comportamento di ogni nodo è governato da politiche autonomiche e segue il principio di località.

Nel seguito di questo capitolo verranno presentate le scelte di design del sistema che si vuole implementare, tenendo conto delle problematiche dovute all'integrazione di un sistema autonomico con le caratteristiche proprie della cloud.

4.2 Analisi dei requisiti

Le caratteristiche delle tecnologie oggetto di studio al Capitolo 3, la particolarità delle metodologie di sviluppo e gli obiettivi dell'applicazione che si vuole realizzare hanno fatto emergere un insieme di requisiti che hanno guidato la fase di analisi e di progetto di questo lavoro. Nel Capitolo 5, in cui sono descritte le soluzioni implementative adottate, saranno mostrate le scelte operate nel rispetto delle seguenti caratteristiche:

- Il servizio di cloud computing utilizzato è Amazon EC2. Ciò nonostante si prevede un'astrazione lato SelfLet del servizio di Cloud Computing utilizzato. L'architettura realizzata dovrà pertanto disaccoppiare le abilities e l'interazione diretta con la cloud.

- Il cloud computing è un modello recente di utilizzo delle risorse, e sono quindi previste numerose ottimizzazioni ed estensioni future. L'architettura che gestisce i servizi relativi alla cloud deve essere sufficientemente generale per permettere facilmente estensioni future.

- Le SelfLets sono implementate per mezzo di actions e abilities scritte in linguaggio Java. La modalità di interazione più naturale con il servizio di cloud computing è l'utilizzo dell'interfaccia API per il linguaggio Java fornito da Amazon.

- L'insieme dei compiti che derivano dalla gestione dei nodi su cloud è ampio. Le SelfLets, in generale, possono essere applicazioni leggere che eseguono su hardware poco performanti. Pertanto, le abilità di gestione non devono essere implementate internamente al SelfLet framework, ma come una sua applicazione.

- Anche se lo scenario presentato ha un solo nodo che svolge la funzione di Cloud Manager, si prevede in futuro la possibilità da parte di una SelfLet che ne abbia bisogno di apprendere i servizi del Cloud Manager per auto-espandersi in maniera autonoma o per supportarlo nella gestione delle richieste. Pertanto le capacità del Cloud Manager devono essere organizzate in servizi che potranno essere richiesti in modalità Teach.

- Le politiche di estensione e contrazione nella cloud dipendono esclusivamente dalle necessità del singolo nodo. E' dunque necessario estendere il SelfLet Framework includendo azioni di ottimizzazione rispetto alla cloud, e scrivendo delle policies cloud-oriented che catturino la nuova tipologia di eventi.

- Il sistema deve essere in grado di eseguire operazioni di redeployment: deve cioè avere l'abilità di cambiare le configurazioni di deployment esistenti quando le condizioni dell'ambiente di computazione cambiano rispetto alle risorse disponibili.

- Una SelfLet è un componente autonomico, il sistema deve pertanto esibire un comportamento di tipo self-organizing: non deve essere coinvolto l'utente nel processo di deploy; i nodi della rete dovrebbero essere in grado di far emergere un comportamento globale coordinato relativamente al problema dell'appliaction deployment.

4.3 Analisi dell'applicazione: obiettivi e caratteristiche

4.3.1 Analisi dell'applicazione

Come spiegato nella sezione 3.3, l'obiettivo di questo lavoro di tesi estendere le proprietà autonomiche del SelfLet framework con la proprietà di auto-approvvigionamento. Questo obiettivo si traduce nella necessità di progettare e implementare un'architettura software che permetta ad un sistema formato da una rete di Selflets di gestire un generico insieme di nodi in esecuzione sulle macchine virtuali offerte dal servizio di cloud computing di Amazon. Globalmente, i componenti della rete devono poter prendere decisioni sulla gestione della parte di popolazione dei nodi all'interno della cloud. Tali decisioni vengono prese sulla base di alcuni parametri che rappresentano le necessità dei componenti stessi (es. il carico di utilizzo). Alcuni di questi componenti, ai fini di attuare le decisioni stabilite, devono avere la possibilità di agire sulle macchine virtuali.

Il servizio AWS che rende possibile questo tipo di azione è EC2, descritto nel Capitolo 3. Questo infatti è la piattaforma base del sistema di cloud computing

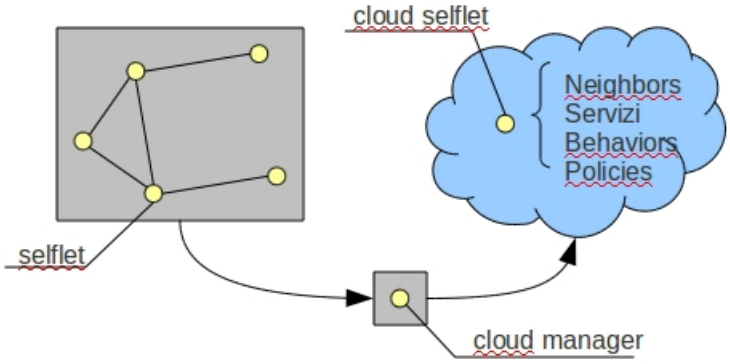

Figura 4.1: Scenario

proposto da Amazon. EC2 fornisce un meccanismo accessibile per controllare e avere a disposizione in qualsiasi momento le istanze virtuali create. Il modello del ciclo di vita della porzione del sistema che viene eseguito su cloud può essere semplicemente definito in tre fasi.

Apertura della cloud: il sistema si occupa di inizializzare il servizio di cloud computing e di preparare e lanciare l'istanza di Selflet che esegue sulla prima macchina virtuale.

Utilizzo della cloud: in questa fase centrale è possibile aggiungere e rimuovere altre istanze nella cloud, vengono monitorate le condizioni di esecuzione all'interno di essa e vengono attuate eventuali operazioni di recovery in caso di malfunzionamento di qualche componente.

Chiusura della cloud: in questa fase centrale è possibile aggiungere e rimuovere altre istanze nella cloud, vengono monitorate le condizioni di esecuzione all'interno di essa e vengono attuate eventuali operazioni di recovery in caso di malfunzionamento di qualche componente.

E' inoltre possibile modellizzare il ciclo di vita di ogni componente che sarà eseguito all'interno della cloud. Questo infatti è la composizione dei cicli di vita rispettivamente della macchina virtuale e del componente software, la Selflet, che girerà sulla macchina. In generale, la gestione di una macchina virtuale si esaurisce nei seguenti passi:
Fase di preparazione:

- selezione della configurazione di base tra quelle disponibili per l'istanza che si vuole creare (es. sistema operativo, software installato)
- attivazione della macchina virtuale
- preparazione della macchina virtuale (es. installazione di software, modifica della configurazione)
- salvataggio di una immagine della configurazione personalizzata

Fase di esecuzione:

- creazione di una nuova macchina virtuale utilizzando l'immagine salvata
- load del componente software
- esecuzione del componente software sull'istanza
- terminazione della macchina virtuale

Per quanto riguarda il ciclo di vita della Selflet, e ai fini della nostra applicazione, è necessario estendere il modello descritto tradizionalmente (così come descritto nel Capitolo 3 . Una Selflet inizia la sua esecuzione svolgendo un servizio di default, oppure rimanendo in attesa di richieste da parte di altre Selflets per l'esecuzione di un servizio. L'esecuzione prosegue attraversando l'insieme degli stati presenti nel diagramma dei behaviors dei servizi che sta eseguendo. Questa può terminare in uno stato di errore, oppure essere ripristinata da un meccanismo di recovery solitamente descritto all'interno del file contenente le politiche autonomiche. La configurazione della Selflet viene solitamente stabilita a priori dallo sviluppatore a seconda dell'applicazione che si vuole realizzare. Nel caso considerato, invece, tale configurazione viene preparata a run time da un componente preposto, come composizione di un insieme finito di configurazioni preesistenti e dipendenti dalla generica applicazione, e completata da un insieme di parti che permettono l'esecuzione della Selflet su una macchina virtuale generata dinamicamente. Da quanto descritto, la gestione su cloud di un generico nodo della rete, si riconduce ad un problema di deploy su macchine virtuali governato da politiche autonomiche.

4.3.2 Identificazione delle selflet instances

E' possibile individuare cinque differenti tipi di Selflet. E' importante notare che alcune di quelle descritte nel seguito sono delle meta-tipologie, in quanto la possibilità di un sistema di utilizzare il servizio di cloud computing è indipendente dal contesto applicativo, e dipendente solo da scelte progettuali dello sviluppatore della particolare applicazione.

Selflet normale: questa è una normale Selfet che viene eseguita su una macchina fisica al di fuori della cloud. Non partecipa ai meccanismi di gestione dei nodi che vivono su cloud ma può interagire con essi, per esempio richiedendo l'esecuzione di servizi. Tipicamente è un sensore, i cui servizi implementati dipendono esclusivamente dal fatto di trovarsi in un luogo fisico specifico; oppure un client per il quale le scelte progettuali non prevedono la possibilità di esecuzione su una macchina virtuale.

Selflet con politica di ottimizzazione su cloud: è una Selflet che offre servizi la cui esecuzione è indipendente dal contesto fisico di esecuzione. Questi sono generalmente servizi che necessitano come uniche risorse, quelle hardware della macchina su cui vengono eseguiti e dati eventualmente forniti da servizi di terzi. Le Selflet con politica di ottimizzazione implementano un meccanismo per effettuare una richiesta di espansione nella cloud, in base a una valutazione del proprio stato.

Cloud Manager: questa è la Selflet più importante dello scenario che si sta considerando. Possiede le informazioni per l'accesso al servizio di cloud computing, e ha la possibilità di interagire con esso per la creazione, la terminazione e la gestione delle macchine virtuali. Implementa meccanismi di monitoring e recovery delle istanze e offre agli altri nodi della rete servizi di espansione nella cloud. L'esecuzione della Selflet avviene necessariamente su una macchina fisica all'esterno della cloud e prosegue in parallelo al suo intero ciclo di vita. E' inoltre il componente preposto alle fasi di apertura e chiusura del servizio di cloud computing.

Cloud Manager clone: questa Selflet è una istanza che replica un sottoinsieme delle funzionalità del Cloud Manager. In particolare ne implementa i servizi operativi di creazione e terminazione dei nodi. Questi servizi

vengono richiesti solo dal Cloud Manager per la delega di operazioni onerose in termini di tempo. Una Selflet di tipo Cloud Manager clone può essere una Selflet con politica di ottimizzazione su cloud.

Cloud Selflet: definiamo Cloud Selflet una Selflet la cui esecuzione avviene su una macchina virtuale del servizio di cloud computing. E' un componente la cui configurazione è stata composta dal Cloud Manager sulla base delle richieste ricevute e il cui ciclo di vita è delimitato dalla fase di apertura e chiusura della cloud. Implementa un meccanismo di richiesta di contrazione (spegnimento) in base a una valutazione del proprio stato. Viene generato successivamente alla richiesta di espansione al Cloud Manager di un componente preesistente. Una Cloud Selflet può essere un Cloud Manager clone o una Selflet con politica di ottimizzazione su cloud. Può inoltre essere il clone di una Selflet particolare, oppure offrire un mix di servizi di diverse Selflets.

4.3.3 Specifica dei servizi di alto livello

I servizi principali individuati e necessari alla realizzazione del meccanismo di auto-approvvigionamento, appartengono a tre macro categorie.
La prima categoria è quella dei servizi di management. Ovvero l'insieme dei servizi necessari alla gestione del servizio di cloud nella sua interezza. Questi sono offerti solo dal Cloud Manager.

init Cloud Service: il Cloud Manager può rimanere in attesa di notifiche da parte delle altre Selflets o iniziare la propria esecuzione con questo servizio di default. In questo caso istanzia un numero arbitrario di nodi nella cloud. Solitamente le Selflet istanziate sono cloni del Cloud Manager; oppure viene eseguito il deploy di una intera applicazione basata sul Selflet Framework all'interno della cloud.

cloud Manager Service: questo è il servizio principale del Cloud Manager. In accordo al paradigma autonomico, implementa il ciclo MAPE di gestione e controllo del servizio di cloud computing.

cloud Control Service: suo è il compito di verificare le condizioni e le modalità per la creazione, la modifica o la distruzione delle Cloud Self-

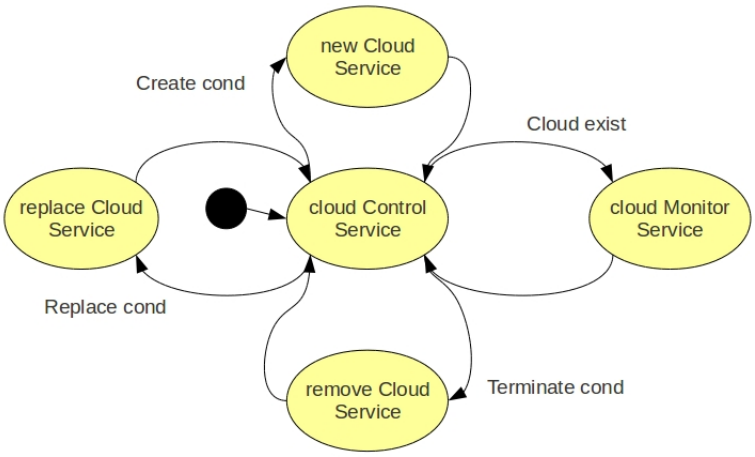

Figura 4.2: Behavior diagram del cloud Manager Service

lets. Tali condizioni possono essere interne o esterne allo stesso stato della cloud, e vengono esposte nel Capitolo 5.

cloud Monitor Service: questo servizio monitora periodicamente lo stato delle Cloud Selflets ed eventualmente attua delle azioni di modifica, recovery o reset sulle istanze interne alla cloud.

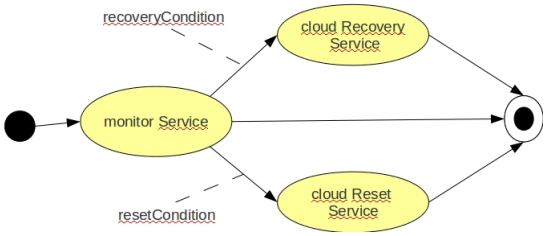

Figura 4.3: Behavior diagram del cloud Control Service

La seconda categoria è quella dei servizi di comunicazione. Questi servizi sono stati introdotti al fine di permettere alle varie Selflets di scambiarsi informazioni relative alla gestione della cloud, e di notificare il proprio stato. Questo livello di comunicazione esula dal meccanismo di aggiornamento dello stato tra neighbors prevista dal modello delle Selflet. Al contrario di quest'ultimo, infatti, è sporadico (il neighborStateManager aggiorna lo stato dei neighbors regolarmente dopo un certo intervallo di tempo) e dipendente dallo stato del singolo nodo e dalle sue politiche.

selflet Register Service: questo servizio, proprio del Cloud Manager, implementa un meccanismo di registrazione delle proprietà associate a una richiesta sollevata da una Selflet con politica di ottimizzazione su Cloud.

notify Cloud Manager Service: tutte le Selflet con politica di ottimizzazione su cloud implementano questo servizio, che svolge la funzione di recupero e preparazione delle proprietà associate alla propria richiesta verso il Cloud Manager. Tale servizio viene invocato sulla base di politiche interne alle singole Selflets.

describe Cloud Selflet Service: questo servizio è implementato dal Cloud Manager. Quando una Cloud Selflet si attiva iniziando la propria esecuzione chiama questo servizio per notificare al Cloud Manager l'avvenuta attivazione. In seguito viene chiamato periodicamente dalla Cloud Selflet per certificare il corretto funzionamento.

La terza categoria è quella dei servizi operativi. Questi servizi hanno lo scopo di svolgere le operazioni di deploy dei nodi sulle macchine virtuali, e sono implementati dal Cloud Manager e dai suoi cloni. La richiesta di esecuzione di questi servizi è effettuata solamente dal Cloud Manager in base alle sue decisioni.

new Cloud Selflet Service: questo servizio crea una macchina virtuale, effettua il deploy e avvia una nuova Cloud Selflet. Se le condizioni di controllo della cloud sono verificate, il servizio viene chiamato dal Cloud Manager in seguito a una richiesta di espansione da parte di una Selflet con ottimizzazione su cloud.

remove Cloud Selflet Service: questo servizio si occupa di disattivare una Cloud Selflet e di terminare la relativa macchina virtuale. Viene chiamato dal Cloud Manager in seguito a una richiesta di contrazione da parte di una Cloud Selflet.

replace Cloud Selflet Service: questo servizio ha una duplice funzione. Si occupa di disattivare e rimuovere una Cloud Selflet mantenendo attiva la relativa macchina virtuale. In seguito, può effettuare il deploy e attivare una Cloud Selflet con una diversa configurazione. E' un servizio di

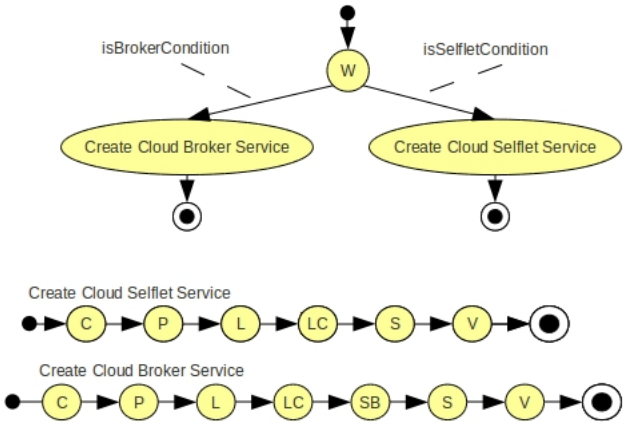

Figura 4.4: Behavior diagram del servizio new Cloud SelfLet

ottimizzazione rispetto al tempo di attivazione delle macchine virtuali e rispetto al loro costo economico. Il servizio viene chiamato dal Cloud Manager in seguito alle opportune richieste delle Selfelts, dopo aver verificato le condizioni sullo stato della macchina virtuale considerata.

Come descritto nel Capitolo 3 la comunicazione tra Selflets avviene attraverso un dispatcher distribuito implementato dal framework REDS [24]. All'interno dei servizi operativi si collocano quindi alcuni servizi dedicati al deploy dei nodi broker che compongono il dispatcher.

create Cloud Broker: il servizio svolge il deploy di un nuovo Cloud Broker, attivando una macchina virtuale, istanziando un REDS broker e agganciandolo a un broker già attivo su una macchina fisica esterna alla cloud. Istanzia inoltre una Cloud Selflet sulla stessa macchina virtuale. E' il primo servizio di deploy ad essere chiamato quando la rete di Selflet si espande in una Availability Zone non ancora utilizzata.

destroy Cloud Broker: il servizio disattiva un Cloud Broker, la Cloud Selflet eventualmente attiva sullo stesso nodo, e termina la macchina virtuale. Se esistono Cloud Selflets attive nell'Availability Zone del broker, queste vengono disattivate e vengono terminate le rispettive macchine virtuali.

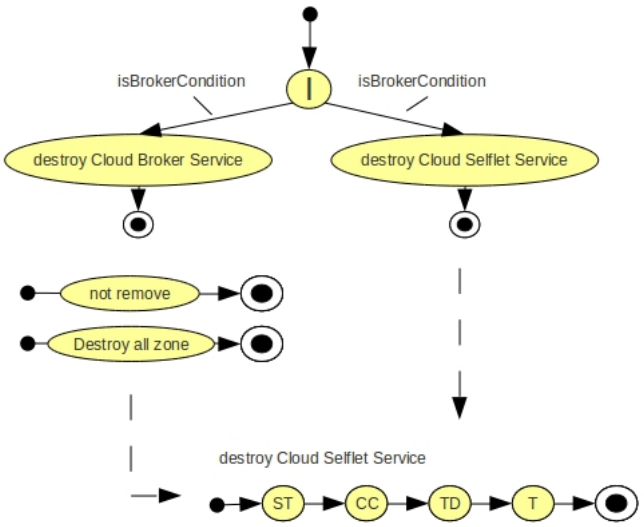

Figura 4.5: Behavior diagram del servizio remove Cloud SelfLet

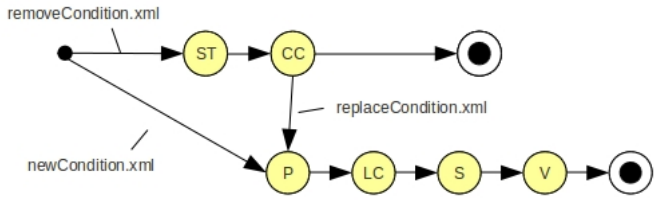

Figura 4.6: Behavior diagram del servizio replace Cloud SelfLet

4.3.4 Specifica dei servizi intermedi

Seguendo il ciclo di sviluppo previsto dal Selflet Framework, sono stati poi dettagliati i sottoservizi. Quelli che seguono sono i sottoservizi di management.

request Evaluation: le possibilità di espandere nuovi nodi nella cloud possono essere limitate da vincoli di carattere economico o da un numero massimo di macchine virtuali che è possibile istanziare. In questi casi il Cloud Manager , attraverso questo servizio, implementa un meccanismo di selezione delle richieste da servire, tra quelle sollevate dalle Selflets.

action Selector: una volta che il Cloud Manager ha selezionato una richiesta, deve scegliere l'azione da effettuare tra quelle previste per quella tipologia di richiesta. Questa azione dipende dallo stato della cloud. L'esposizione dell'algoritmo di decisione sarà trattato nel Capitolo 5.

recovery: questo servizio è invocato dal Cloud Manager in seguito al rilevamento di casi di malfunzionamento dei nodi della rete all'interno della cloud. Questo può effettuare diverse azioni come restaurare il Cloud Broker di una data Availability Zone oppure modificare le condizioni per l'azione del Cloud Manager rispetto a una certa istanza virtuale.

reset: questo è un servizio di emergenza che viene chiamato in casi di grave malfunzionamento del sistema. Questi includono la caduta del nodo che svolge la funzione di Cloud Manager: il primo nodo all'interno della cloud che vedrà verificata questa condizione si occuperà di disattivare l'intera porzione di sistema facente riferimento al manager. Lo stesso Cloud Manager può svolgere questo servizio in casi di errore, come l'eventualità che il numero o il costo delle macchine virtuali superi il livello soglia definito dal progettista.

Anche i servizi operativi sono stati scomposti in operazioni elementari. L'elevato numero dei sottoservizi individuati sono dovuti alla complessità della fase di deploy. I sottoservizi così ottenuti, rappresentati in figura 4.7 sono stati progettati in modo da ottenere i servizi descritti nella sezione precedente per composizione.

prepare Configuration: questo sottoservizio si occupa di preparare la configurazione della Cloud Selflet che si vuole generare aggiornando i dati relativi al broker di riferimento, i neighbors che la Selflet dovrà conoscere al momento della sua inizializzazione, selezionando un id e l'insieme dei servizi e delle abilities che la Selflet dovrà conoscere.

create Instance: il servizio effettua la richiesta di attivazione di una nuova macchina virtuale, recuperandone l'identificativo.

load Selflet: si occupa di caricare il Selflet Engine sulla macchina virtuale che si sta preparando.

load Configuration: in generale, la fase di preparazione della configurazione di una Cloud Selflet è seguita dalla sua installazione su macchina virtuale. Questo servizio si occupa di eseguire il corretto trasferimento della configurazione.

start Selflet: avvia una Cloud Selflet già configurata su una certa macchina virtuale.

start Broker: il servizio avvia il Cloud Broker di riferimento per una certa Availability Zone.

verify Selflet: si occupa di verificare il corretto funzionamento di una Cloud Selflet avviata.

stop Selflet: il servizio ferma l'esecuzione di una Cloud Selflet.

clean Configuration: effettua i passi di rimozione della traccia di configurazione di una Cloud Selflet

tear Down Instance: il servizio ripristina lo stato iniziale di una macchina virtuale ancora attiva.

terminate Instance: effettua la richiesta di spegnimento di una macchina virtuale.

who Is Broker: questo servizio si occupa di verificare che l'Availability Zone in cui si vuole lanciare una Cloud Selflet abbia un Cloud Brooker di riferimento.

Is Broker: il servizio si occupa di verificare che una certa macchina virtuale sia quella su cui sta eseguendo il broker di riferimento per una certa Availability Zone.

I servizi di comunicazione, infine, non sono stati ulteriormente dettagliati in quanto sono già di per sé costituiti da operazioni atomiche di notifica.

4.4 Tool utilizzati

In aggiunta alle tecnologie di base presentate nel Capitolo 3 sono state utilizzate, per la realizzazione del sistema, le seguenti tecnologie. **ArgoUml** [2] è stato utilizzato per la realizzazione degli StateCharts che rappresentano i behavior diagrams che implementano i servizi delle SelfLets. Le politiche autonomiche che ne governano il comportamento sono state scritte utilizzando

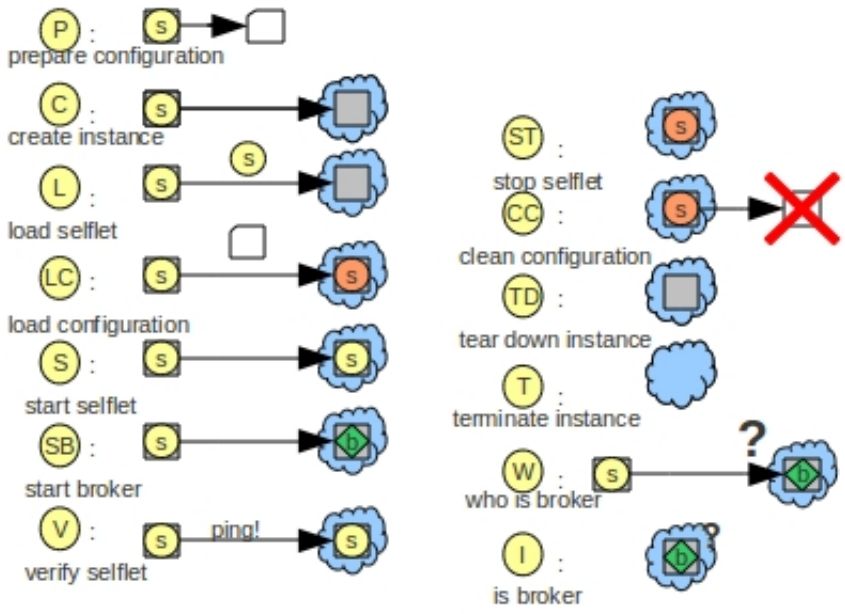

Figura 4.7: Rappresentazione funzionale dei sottoservizi

la piattaforma a regole **Drools** [17]. E' stato studiato brevemente il framework **OSGi** [20], utile all'installazione dinamica delle abilities da parte delle SelfLets, così come il middleware **REDS** [24], utilizzato per la creazione dei CloudBroker che permettono la comunicazione tra i nodi della rete che eseguono in ambiente cloud, con quelli al di fuori di esso. Le librerie **Swing** [12] sono state utilizzate per la realizzazione di un componente per il monitoring e la raccolta dei risultati di test. Tale componente, chiamato CloudLogger, sarà introdotto nel Capitolo 5. **JUnit** [18] è stato utilizzato per il testing dei metodi offerti dalle singole abilities e per testare il framework realizzato per l'interazione con il servizio di cloud computing di Amazon. E' stato studiato il framework **Log4j** [14] utilizzato nel SelfLet framework per la produzione di file di log. Infine, per il trattamento dei file XML di configurazione di ogni SelfLet, sono stati utilizzate le librerie **JAXB** e **JDOM** [19].

4.5 Conclusioni

In questo capitolo sono stati analizzati i requisiti che emergono dalle caratteristiche dei processi di sviluppo di sistemi basati sul SelfLet framework e dalle

problematiche sollevate dal progetto di sistemi cloud-oriented. E' stata poi descritta l'applicazione che si vuole realizzare ed è stata progressivamente dettagliata la natura dei componenti per la sua realizzazione secondo l'approccio di sviluppo previsto per le SelfLets. Sono stati definiti i requisiti desiderabili e introdotte le tecnologie utilizzate nel corso dello sviluppo.
Nonostante la fase di progettazione abbia seguito un approccio di tipo top-down, naturale al modello di sviluppo previsto per applicazioni basate sul Selflet Framwork, per l'implementazione del sistema si è seguita una strategia bottom-up, derivante dalla necessità di acquisire confidenza sia con il modello di funzionamento delle Selflet che con le modalità di interazione fornite dal servizio di cloud computing scelto. Per questo motivo, la fase di progettazione è stata anticipata e seguita da più fasi di studio delle tecnologie impiegate, e a diversi livelli di profondità. Lo studio teorico delle metodologie di sviluppo, i cui risultati sono esposti nel Capitolo 3, sono state messe in pratica attraverso l'implementazione di semplici casi di studio. Così, la fase di identificazione dei servizi e delle abilities esposta in questo capitolo è stata ulteriormente dettagliata solo nel corso dello sviluppo di un componente, chiamato CloudTouch 5, preposto a offrire i metodi base per l'integrazione del Selflet Framework con il servizio di cloud computing. L'esposizione di dettaglio di servizi e abilities sarà pertanto trattata nel seguito del Capitolo 5.

Capitolo 5

Progetto e sviluppo della soluzione

Le proprietà Self-* proposte da IBM rappresentano un modello di riferimento nello sviluppo dei sistemi autonomici. Il SelfLet framework è un'implementazione di questo modello, orientato alla realizzazione di reti di nodi che collaborano tra loro per il raggiungimento degli obiettivi. La natura del comportamento e delle interazioni tra i nodi è guidata da decisioni locali prese da ogni elemento. Secondo questa prospettiva il problema del controllo del ciclo di vita della popolazione di nodi appartenenti alla rete, la cui esecuzione avviene in ambiente cloud, è un aspetto di gestione autonomica della topologia della rete, che emerge dalle condizioni locali di ogni SelfLet. In questo capitolo sarà presentata una soluzione per l'estensione del SelfLet framework con le possibilità offerte dal servizio cloud computing di Amazon. Il progetto sviluppato si compone di quattro elementi. Il Cloud Touch framework, presentato nella sezione 5.2, permette l'interazione del sistema con il servizio di cloud computing. Il Cloud Manager è una SelfLet il cui scopo è la gestione centralizzata delle istanze virtuali in esecuzione ed è stato sviluppato in accordo al modello di componente autonomico previsto da IBM, come mostrato nella sezione 5.4. Cloud Logger è il componente preposto al monitoring delle SelfLet in esecuzione sulla cloud, e sarà presentato nella sezione 5.5. Infine è stata estesa l'architettura per la gestione delle politiche di ottimizzazione già presente nel SelfLet framework, ed è stato progettato un modello di interazione tra i nodi della rete ed il Cloud Manager per la gestione delle operazioni di espansione e contrazione della rete nella cloud. Il modello sarà dettagliato nella sezione 5.6.

5.1 Modello di deploy

Nelle sezioni precedenti sono stati utilizzati intuitivamente i termini di espansione e contrazione della rete, ne viene ora presentata una più completa definizione.

espansione: un passo di espansione della rete è un insieme di operazioni di deploy che hanno come conseguenza l'avvio di un nuovo nodo facente parte della rete. Il deploy viene attuato da un nodo, il Cloud Manager, in grado di interagire con il servizio di cloud computing. Una richiesta di espansione viene sollevata da una SelfLet che preveda politiche di ottimizzazione su cloud, in seguito alla verifica di condizioni del proprio stato.

contrazione: un passo di contrazione della rete è un insieme di operazioni di deploy che hanno come conseguenza la caduta controllata di un nodo della rete. Una mossa di contrazione è attuata dal Cloud Manager e può essere richiesta da una Cloud Selflet in seguito alla verifica del proprio stato.

La figura 5.1 mostra uno schema generale delle operazioni di espansione. Il Cloud Manager avvia la prima macchina virtuale in una nuova availability zone, installa un Cloud Broker agganciandolo alla rete dei broker già attivi e avvia una prima SelfLet. Un numero arbitrario di altre SelfLets possono essere successivamente avviate nella zona all'interno di nuove macchine virtuali allocate. La popolazione di SelfLets in esecuzione all'interno della cloud può comunicare con le altre parti della rete sfruttando il servizio di comunicazione publish/subscribe offerto dalla rete dei borkers.

5.2 Cloud Touch

Come è stato accennato, la fase di implementazione ha seguito un approccio di tipo bottom-up. Il primo passo nello sviluppo è stato quindi la costruzione di un componente per l'interazione con il servizio di cloud computing. Questo elemento rappresenta l'unico punto di accesso al servizio da parte di un sistema client ed è stato progettato per garantire il disaccoppiamento della logica di

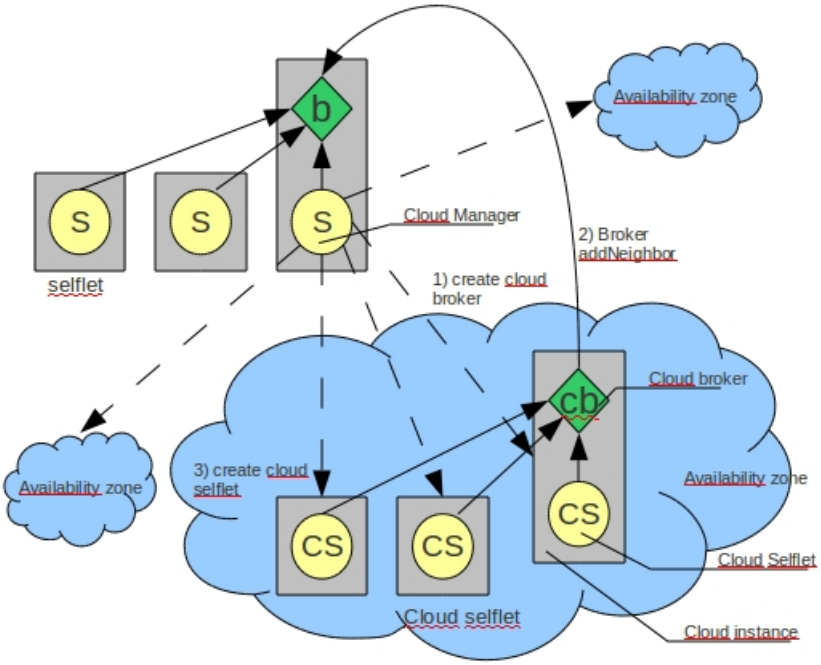

Figura 5.1: Scenario di deployment

tale sistema dalla specifica implementazione del servizio di cloud computing disponibile. Per questo motivo è stato adottato un pattern di tipo Factory. Come mostrato in figura 5.2 il client richiede al componente CloudFactory un oggetto di tipo CloudTouch. CloudTouch è un'interfaccia e la classe factory restituisce al client un'istanza di oggetto che la implementa, scegliendo tra le implementazioni disponibili.

Attraverso questa interfaccia, il componente che ne ha fatto richiesta può effettuare le chiamate ai metodi per l'effettiva interazione con il servizio. Il tipo di interazioni previste si divide in due categorie: quelle per la gestione del ciclo di vita delle macchine virtuali e le operazioni per l'interazione con la singola macchina virtuale. La prima categoria offre la possibilità di effettuare richieste per la creazione, la terminazione, e la configurazione delle VM. Offre inoltre la possibilità di monitorare le istanze create e di recuperare informazioni relative allo stato, al gruppo di appartenenza, al tempo di esecuzione, o altre caratteristiche come la regione geografica di appartenenza o l'indirizzo di rete. La seconda tipologia offre i metodi che consentono le operazioni per il deploy delle applicazioni e per l'effettivo utilizzo della macchina virtuale creata. tra queste troviamo metodi per la gestione del file system, per il trasferimento di

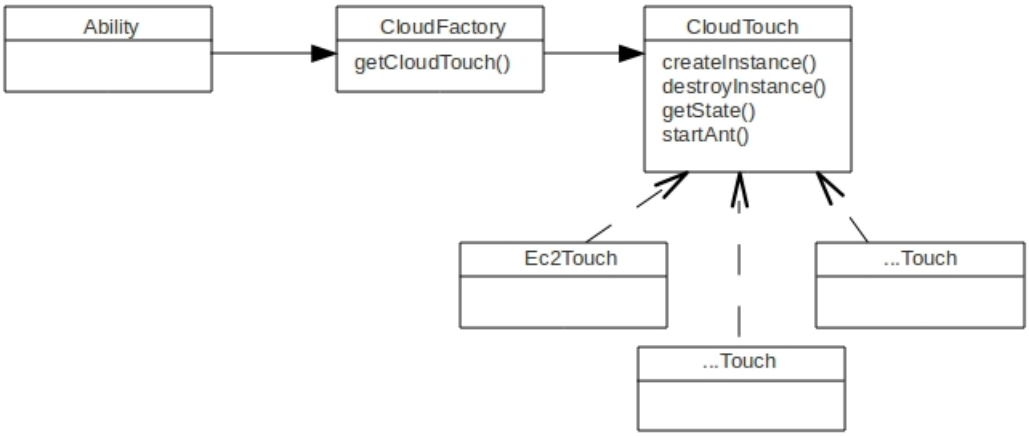

Figura 5.2: Pattern factory per il framework cloudTouch

files, per l'installazione di packages, l'avvio o la terminazione di applicazioni contenute all'interno di file JAR e l'esecuzione dei build files Ant. Essendo Amazon il cloud infrastructure provider scelto, l'implementazione attualmente disponibile è fornita dalla classe EC2Touch. La classe implementa i metodi descritti e interagisce con il servizio EC2 attraverso l'interfaccia Java API fornita dal provider. Questa prevede l'allocazione di un oggetto di tipo AmazonEC2 [7] che offre delle funzionalità più elementari per la composizione delle richieste al servizio. Le sole informazioni necessarie per l'interazione sono le credenziali di sicurezza. Queste consistono in una coppia di chiavi, AWS Acces Key ID e Secret Access Key [11], che vengono comunicate nella fase di autenticazione al servizio, che deve precedere ogni altra operazione.

5.3 Cloud Abilities

Sullo strato di interazione costituito dal CloudTouch framework sono state costruite le abilities del Cloud Manager. Come descritto nella sezione 3.1.1 un'ability è un programma auto-contenuto che implementa una funzionalità della SelfiLet ed è invocata da questa all'interno di una action, a sua volta associata a un elementary behavior, che rappresenta un'operazione atomica nella realizzazione di un servizio. Le abilities sono state definite attraverso un'ulteriore fase di dettaglio dei sotto-servizi descritti nella sezione 4.3.4. Tali sotto-servizi, che costituiscono i passi elementari delle operazioni di deploy, sono portati a termine attraverso l'esecuzione di più operazioni. Sono queste

operazioni, organizzate per tipologia, e ad essere incapsulate all'interno dei pacchetti OSGi che implementano le abilities. Nella figura 5.3 è mostrato uno schema generale del flusso di esecuzione del Cloud Manager e di come questo interagisce con il servizio AWS per mezzo delle abilities.

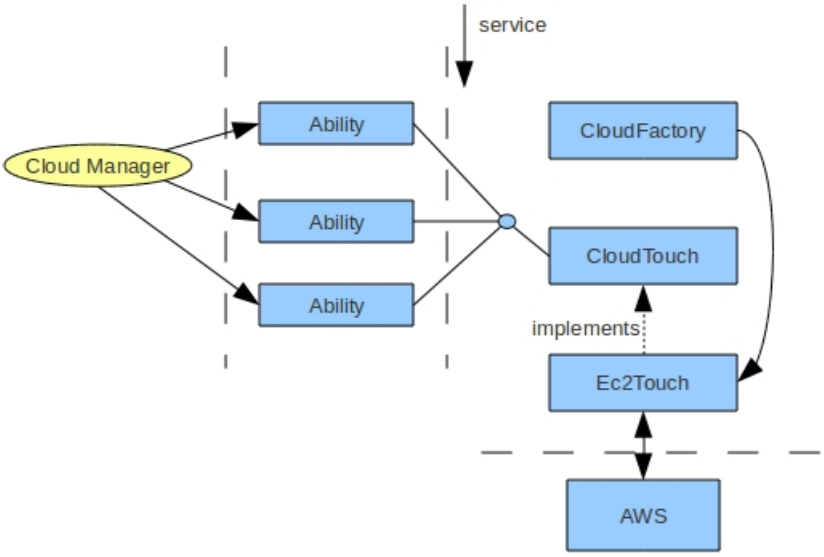

Figura 5.3: Interazione del Cloud Manager con AWS

Le tipologie di operazioni individuate sono state organizzate nel modo seguente:

CloudBrokerAbility: espone i metodi per la verifica dello stato dei cloud-Broker e per il recupero delle informazioni di rete associate.

CloudCleanSelfletConfigurationAbility: implementa le operazioni di rimozione della configurazione di una SelfLet su una macchina virtuale.

CloudCreateAbility: esegue la richiesta di una nuova macchina virtuale.

CloudLoadSelfletAbility: si occupa di caricare il SelfLet engine su una macchina virtuale, per l'esecuzione delle SelfLets.

CloudLogClientAbility: permette la creazione di un Cloud Logger client, il sistema di login remoto Cloud Logger è descritto nella sezione 5.5.

CloudLogServerAbility: avvia un Cloud Logger server, il sistema di login remoto Cloud Logger è descritto nella sezione 5.5.

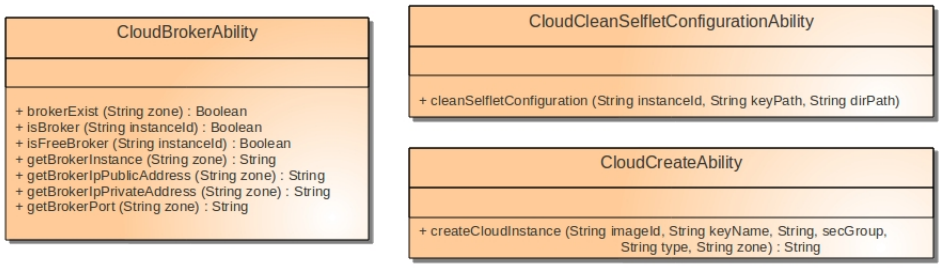

Figura 5.4: Diagrammi delle classi delle abilities

CloudMonitorAbility: espone i metodi per il monitoring delle istanze virtuali, delle Cloud SelfLets, e per la gestione del Cloud Pool, il cui funzionamento è descritto nella sezione 5.10.1.

CloudPrepareSelfLetConfigurationAbility: implementa le operazioni per la preparazione del file XML e la composizione dell'insieme di abilities, behavior, actions e servizi che costituiscono la configurazione di una SelfLet.

CloudStartSelfLetAbility: ha il compito di avviare una Cloud SelfLet configurata su una macchina virtuale.

CloudStopSelfLetAbility: si occupa di fermare l'esecuzione di una Cloud-SelfLet che sta eseguendo su una macchina virtuale

CloudTearDownInstanceAbility: gestisce le operazioni di ripristino di una macchina virtuale alla sua configurazione iniziale.

CloudTerminateInstanceAbility: inoltra la richiesta di spegnimento di una macchina virtuale attiva.

ReflectionAbility: fornisce a una SelfLet le funzionalità di preparazione della propria configurazione così da poterla comunicare al Cloud Manager.

5.4 Cloud Manager: touchpoint

Per la progettazione del Cloud Manager, si è fatto riferimento alla struttura logica di un autonomic element, descritta nella sezione 2.1.3. La figura 5.7.

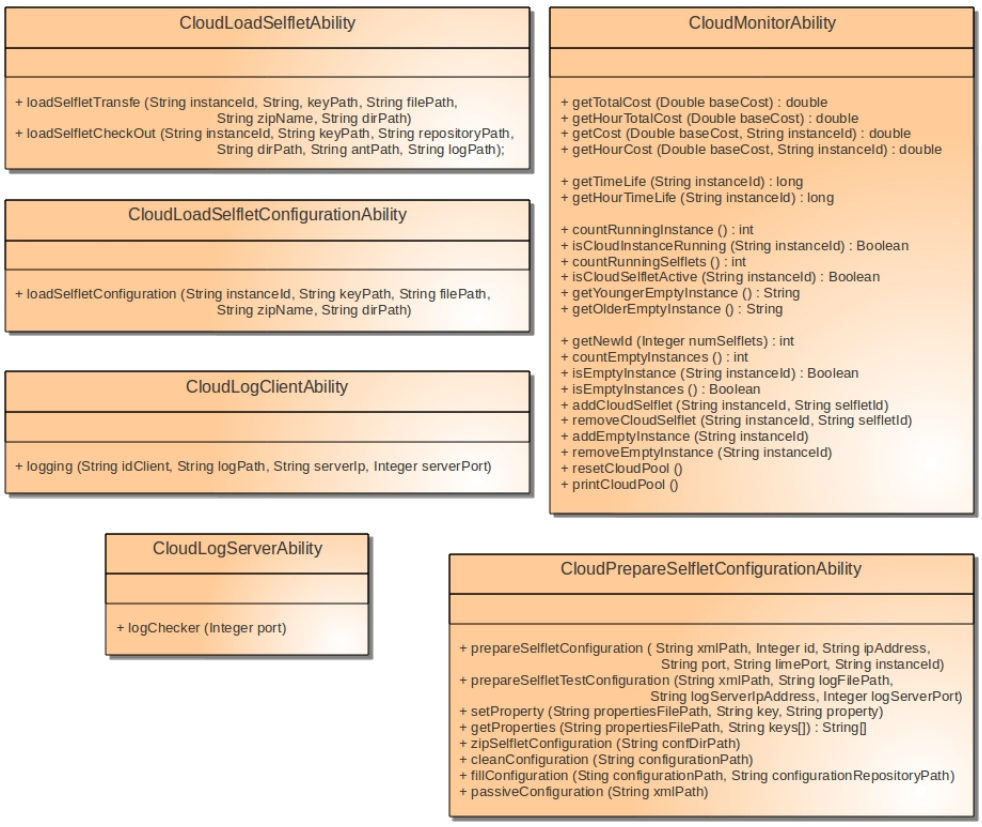

Figura 5.5: Diagrammi delle classi delle abilities

mostra l'architettura del Cloud Manager. In questa architettura la managed resource è un generico cloud element: questo può essere una Cloud Selflet, un Cloud Broker, una macchina virtuale oppure un insieme eterogeneo di questi elementi. Analogamente a quanto accade in un autonomic element, il Cloud Manager può interagire con la managed resource attraverso un componente, il touchpoint, che fornisce al manager una interfaccia per le operazioni di monitoring e di intervento sull'elemento gestito. Nel Cloud Manager, il ruolo di touchpoint è assunto dal framework CloudTouch e dalle abilities che lo utilizzano per l'esecuzione delle operazioni; questi componenti sono stati descritti nelle sezioni precedenti.

Nel seguito verrà esaminata la fase di esecuzione del ciclo MAPE-K implementata dal Cloud Manager.

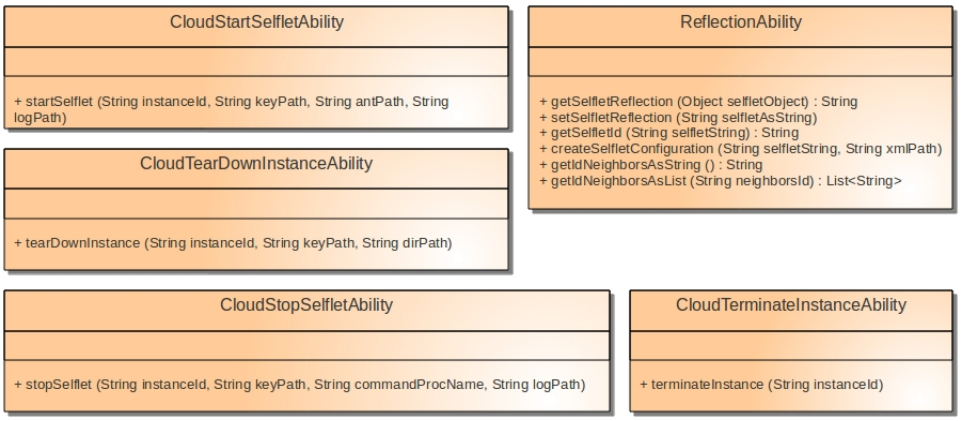

Figura 5.6: Diagrammi delle classi delle abilities

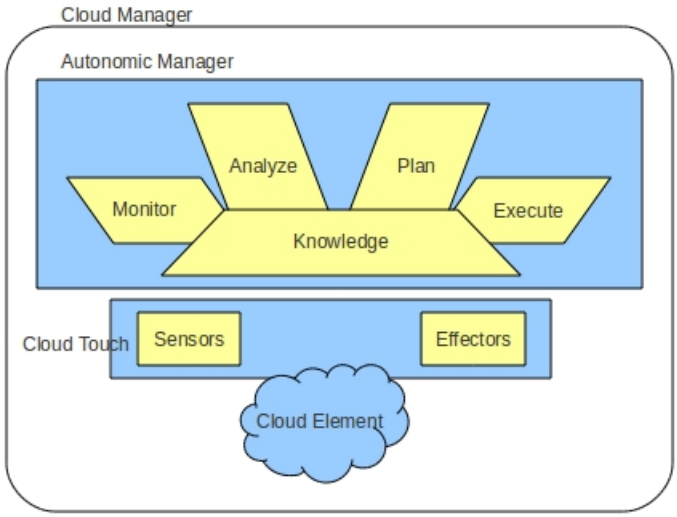

Figura 5.7: Architettura logica del cloud manager

Struttura di progetto di una SelfLet

Lo studio e lo sviluppo del processo di deploy parte dall'osservazione della struttura del progetto che costituisce la configurazione di una SelfLet. Come mostrato in figura 5.8, una configurazione si compone di un insieme di directory, ciascuna delle quali contiene una categoria di componenti: abilities, actions, behavior diagrams, conditions e rules.

A queste cartelle si aggiunge un file XML, il cui schema è mostrato in figura 5.9, che descrive le caratteristiche della SelfLet. In questo file troviamo informazioni circa l'iniziale contenuto della knowledge base, le modalità con cui vengono richiesti ed offerti i servizi e la loro definizione con la specifica del default behavior che li implementa.

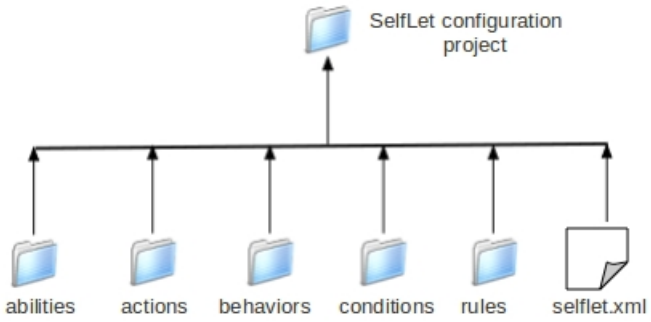

Figura 5.8: Struttura del progetto di configurazione di una SelfLet

Lo schema è concettualmente diviso in due parti: proprietà e risorse. La prima parte contiene le caratteristiche generali come il nome, la descrizione e i parametri di connessione al middleware REDS. Questa parte contiene anche una sezione che permette di settare il contenuto della type knowledge attraverso il tag selfletProperty per la specifica delle variabili. La seconda parte dello schema permette di dichiarare tutte le risorse appartenenti alla SelfLet e contenuta nelle directory di progetto. Il file di descrizione viene esaminato nella fase di inizializzazione della SelfLet e il suo contenuto viene memorizzato all'interno della knowledge base. Il motivo per cui il file XML di configurazione è particolarmente interessante è la necessità, da parte del CloudManager, di completarne la descrizione in una qualche fase di deploy al fine di inizializzare correttamente la Cloud Selflet che si vuole avviare. Inoltre, in fase di configurazione del sistema, è necessario configurare lo stesso Cloud Manager con alcune informazioni che saranno utilizzate dallo stesso nella fase di deploy. Queste informazioni sono inserite all'interno del nodo generalKnowledge del file XML per mezzo degli appositi tag selfletProperty.

Nei paragrafi seguenti saranno mostrati in dettaglio i processi di deploy relativi alla creazione e alla terminazione di un nodo della rete. Questi processi, che sono realizzati come composizione di operazioni elementari costruite sulla base del Cloud Touch framework e delle abilities presentate nelle sezioni precedenti, rappresentano l'effettiva implementazione dei servizi operativi descritti nella sezione 4.3.3.

```
<!ELEMENT selflet (properties,resources)>
    <!ATTLIST selflet name CDATA #REQUIRED>

<!ELEMENT properties (author, description, (active|passive),
                      reds, lime, knowledge?, type?)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT description (#PCDATA)>
    <!ELEMENT active EMPTY>
        <!ATTLIST active mainGoal CDATA #REQUIRED>
    <!ELEMENT passive EMPTY>
    <!ELEMENT reds (address, sport)>
        <!ELEMENT address (#PCDATA)>
        <!ELEMENT port    (#PCDATA)>
    <!ELEMENT lime (port)>
    <!ELEMENT knowledge (selfletProperty+)>
    <!ELEMENT type      (selfletProperty+)>
    <!ELEMENT selfletProperty EMPTY>
        <!ATTLIST selfletProperty name CDATA #REQUIRED>
        <!ATTLIST selfletProperty type CDATA #REQUIRED>
        <!ATTLIST selfletProperty value CDATA #REQUIRED>

<!ELEMENT resources (abilities,actions,behaviors,
                     conditions,goals,rules)>

    <!ELEMENT abilities (ability*)>
        <!ELEMENT ability (service,method+)>
            <!ELEMENT service (#PCDATA)>
            <!ELEMENT method  (paramType*)>
                <!ATTLIST method name CDATA #REQUIRED>
            <!ELEMENT paramType (#PCDATA)>
        <!ATTLIST ability file CDATA #REQUIRED>

    <!ELEMENT actions (action*)>
        <!ELEMENT action EMPTY>
            <!ATTLIST action file CDATA #REQUIRED>

    <!ELEMENT behaviors (behavior*)>
        <!ELEMENT behavior EMPTY>
            <!ATTLIST behavior file CDATA #REQUIRED>

    <!ELEMENT conditions (condition*)>
        <!ELEMENT condition EMPTY>
            <!ATTLIST condition file CDATA #REQUIRED>

    <!ELEMENT goals (goal*)>
        <!ELEMENT goal EMPTY>
            <!ATTLIST goal file CDATA #REQUIRED>

    <!ELEMENT rules (rule*)>
        <!ELEMENT rule EMPTY>
            <!ATTLIST rule file CDATA #REQUIRED>
```

Figura 5.9: File DTD di descrizione della SelfLet

new Cloud SelfLet Service

Il servizio newCloudSelfletService implementa il processo di creazione di una Cloud SelfLet secondo il modello di deploy presentato nella sezione 5.1. La figura 5.11 mostra una rappresentazione di questo processo.

Il Cloud Manager fa riferimento a una location di esecuzione del servizio di cloud specificata nella proprietà zone della general knowledge. Nel caso in cui il servizio di cloud computing utilizzato sia quello fornito da Amazon, questa proprietà contiene l'identificativo della availability zone che si vuole utilizzare tra quelle disponibili per il servizio EC2. Il primo passo di deploy consiste nella verifica della presenza di un Cloud Broker attivo all'interno della zona; questo elemento svolge la funzione di broker di riferimento per la popolazione

```
<!-- parametri di progetto-->
<selfLetProperty name="baseCost" type="double"/>
<selfLetProperty name="costThreshold" type="double"/>
<selfLetProperty name="timeThreshold" type="integer"/>
<selfLetProperty name="numSelflets" type="integer"/>

<!-- proprietà  di configurazione-->
<selfLetProperty name="amiId" type="string"/>
<selfLetProperty name="keyName" type="string"/>
<selfLetProperty name="secGroup" type="string"/>
<selfLetProperty name="VmType" type="string"/>
<selfLetProperty name="zone" type="string"/>
selfLetProperty name="keyPath" type="string" />
<selfLetProperty name="prototypeLocalPath" type="string"/>
<selfLetProperty name="prototypeReposPath" type="string"/>
<selfLetProperty name="remoteSelfletDirPath" type="string"/>
<selfLetProperty name="configurationPath" type="string"/>
<selfLetProperty name="configurationsRepositoryLocalPath"
                                          type="string"/>
<selfLetProperty name="xmlPath" type="string"/>
<selfLetProperty name="redsBuildPath" type="string"/
<selfLetProperty name="selfletBuildPath" type="string"/>
<selfLetProperty name="brokerPropertiesPath" type="string"/>
<selfLetProperty name="neighborsPropertiesPath"type="string"/>
<selfLetProperty name="configurationBuildPath" type="string"/>
<selfLetProperty name="logFilePath" type="string"/>
```

Figura 5.10: Proprietà di configurazione del Cloud Manager

di nodi all'interno della stessa zona. Se esiste un broker attivo ne vengono recuperate le informazioni di rete, l'indirizzo ip privato (raggiungibile solo da istanze la cui esecuzione avviene all'interno della stessa availability zone) e il port, che saranno utilizzati per la configurazione dei parametri di connessione al middleware di comunicazione necessari alla nuova SelfLet. Nel caso in cui non sia presente un broker attivo, il Cloud Manager memorizza dei parametri di rete predefiniti che saranno utilizzati per la connessione di una SelfLet il cui deploy avverrà sulla stessa macchina virtuale su cui sarà attivato il broker; in questo caso, il manager si occupa di preparare un file di proprietà contenente le informazioni di rete del proprio broker di riferimento. Il Cloud Broker che verrà attivato potrà utilizzare queste informazioni per agganciarsi alla rete dei broker già attivi permettendo la comunicazione tra i diversi gruppi di SelfLets. La condizione di esistenza del broker sarà utilizzata dal Cloud Manager per scegliere quale servizio eseguire tra i due sottoservizi disponibili: createCloud-BrokerService e createCloudSelfletService. I due servizi si differenziano per l'aggiunta, nel caso createCloudBrokerService, delle azioni per l'attivazione di un Cloud Broker.

Il passo successivo è la creazione di una macchina virtuale all'interno della zona di riferimento. Il tipo macchina richiesta e l'immagine utilizzata per la sua inizializzazione sono specificati nelle proprietà VmType e amiId. Il Cloud

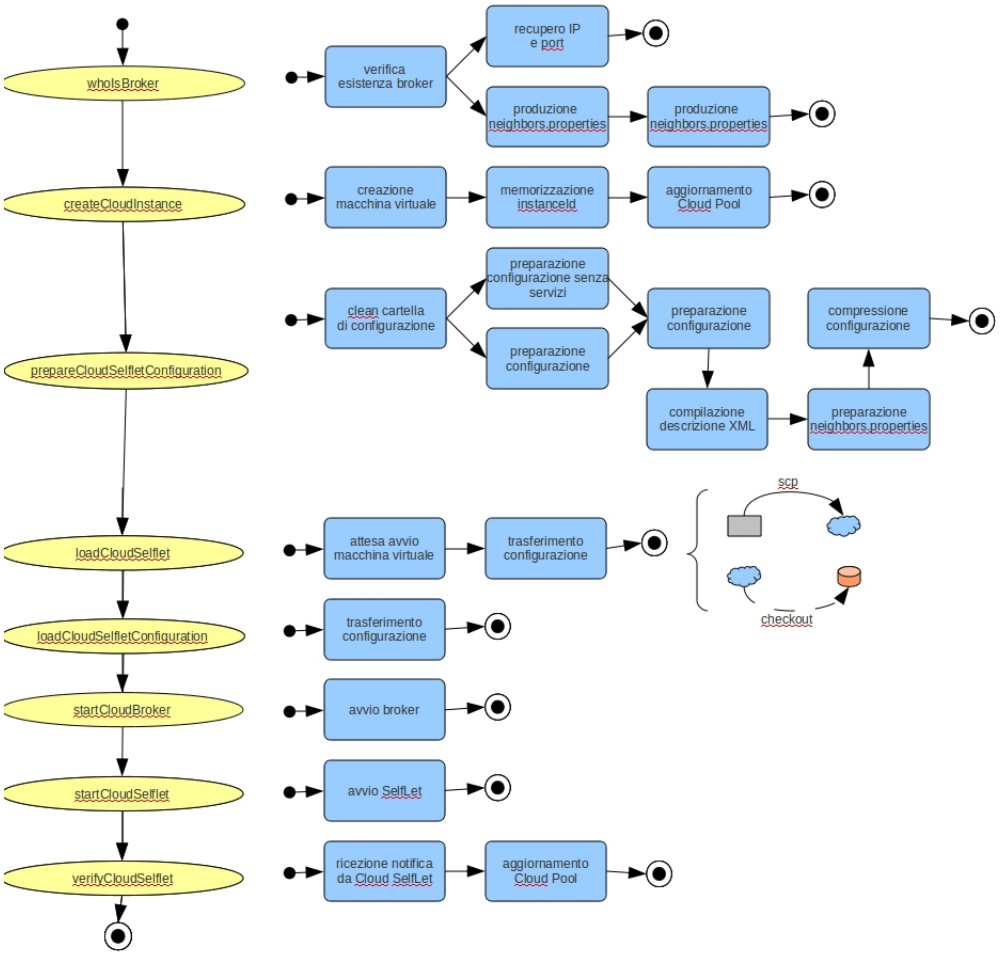

Figura 5.11: Processo di deploy implementato dal servizio newCloudSelfLetService

Manager effettua la richiesta di attivazione di una nuova istanza al servizio di cloud computing, recuperandone l'identificativo. Questo valore viene memorizzato come proprietà della knowledge base del manager e sarà utilizzato in seguito per la configurazione dell'istanza. Lo stesso valore viene inserito all'interno del CloudPool. Questo componente permette al Cloud Manager di effettuare le operazioni di monitoring e sarà trattato nella sezione 5.5. La macchina virtuale richiesta non è immediatamente disponibile. All'istante in cui viene effettuata la richiesta segue un intervallo di tempo necessario all'attivazione della VM durante il quale il Cloud Manager effettua le operazioni di configurazione.

Il progetto contenente la configurazione della SelfLet che si vuole avviare sarà contenuto all'interno di una directory specificata nella proprietà configu-

rationPath. Viene dunque eliminata l'eventuale configurazione dovuta a una precedente operazione di deploy e viene composta la nuova configurazione con le regole e le condizioni contenute nel repository locale al Cloud Manager e contenente le configurazioni pre-impostate per le tipologie di SelfLets previste nel sistema. Se specificato dalla richiesta di espansione che si sta servendo, viene aggiunta alla configurazione l'insieme dei behavior, delle actions e delle abilities che compongono i servizi della nuova SelfLet. In caso contrario, all'attivazione del nuovo nodo, questo si occuperà di apprendere i servizi specificati nel proprio file di descrizione attraverso il meccanismo Teach previsto dal SelfLet framework. Il Cloud Manager completa la configurazione aggiungendo al progetto il file di descrizione XML ricevuto come parametro della richiesta di espansione, compilandolo con le informazioni di rete del broker di riferimento e costruendo un file di proprietà contenente l'elenco dei neighbors che il nuovo nodo dovrà conoscere al momento dell'avvio. Il progetto così ottenuto viene compresso in un file ZIP.

Affinchè sia possibile avviare il nuovo nodo, il SelfLet framework deve essere installato sulla macchina remota. Completata la configurazione, il manager attende eventualmente che la VM richiesta passi in uno stato running, successivamente si occupa dell'installazione del framework. Sono previste due modalità per questa operazione: checkOut e Transfer. La prima opzione consiste nell'invio di un comando remoto alla macchina virtuale che scaricherà così il progetto SelfLetPrototype da un repository remoto specificato nella proprietà prototypeReposPath; la seconda modalità consiste nelll'invio di un file ZIP contenente il progetto SelfLet memorizzato nella sede di esecuzione del Cloud Manager. Le due possibilità si distinguono per le migliori performance dell'operazione di checkOut; ciò nonostante il repository remoto potrebbe non essere raggiungibile, questo è il motivo dell'introduzione della modalità Transfer.

L'installazione del SelfLet framework è seguita dal trasferimento sulla macchina remota del file ZIP contenente il progetto di configurazione della Cloud SelfLet che si vuole attivare. Se il servizio in esecuzione è createCloudBrokerService, il manager avvia un'istanza del broker inviando alla VM il comando di esecuzione del file ANT finalizzato a invocare il metodo Main della classe RedsCloudMiddleware, aggiunta durante lo sviluppo al package it.polimi.elet.selflet del SelfLet framework. Il Cloud Manager invia infine un ulteriore comando per l'esecuzione del file ANT preposto all'avvio del Self-

letCloudLauncher. Questo componente si occupa di avviare l'istanza di SelfLet contenuta nel progetto di configurazione, inizializzandola con l'insieme dei neighbors specificati nel file di configurazione. Avviata la SelfLet, il manager aggiorna il Cloud Pool con l'id del nodo appena creato.

remove Cloud SelfLet Service

Il servizio removeCloudSelfletService implementa il processo di terminazione di una Cloud SelfLet, come mostrato in figura 5.12.

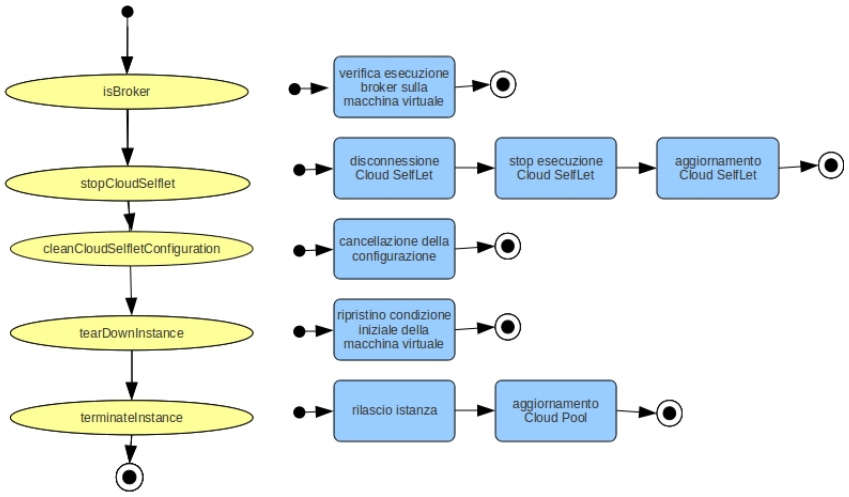

Figura 5.12: Processo di rimozione implementato dal servizio removeCloudSelfLetService

Il servizio viene eseguito in seguito ad una richiesta di contrazione da parte di una Cloud SelfLet. Ricevuta la richiesta, il Cloud Manager memorizza il parametro ad essa associata e contenente l'id della macchina virtuale sede dell'esecuzione del nodo che chiede di essere terminato. Questo parametro verrà utilizzato per le operazioni di terminazione.

Il Cloud Manager verifica la natura dell'istanza che si vuole terminare. Questa può essere la sede di esecuzione del broker di riferimento di una zona gestita dal manager. In questo caso, il manager verifica la presenza di eventuali nodi ancora attivi nella zona e serviti dal Cloud Broker. Se il broker è libero, ovvero non serve nessun nodo, la macchina virtuale può essere rilasciata. In alternativa il manager dovrà scegliere quale strategia adottare tra la terminazione della sola Cloud SelfLet in esecuzione sulla VM lasciando attivo il broker, oppure il reset dell'intera popolazione della zona. Il comportamento

di deafult consiste nello spegnimento della sola Cloud SelfLet. Le condizioni mostrate determinano l'esecuzione di uno tra i due servizi: removeCloudBrokerService o removeCloudSelfletService, in maniera analoga a quanto mostrato per il servizio di creazione.

L'operazione di stop del nodo consiste nell'invio di un comando remoto per la terminazione del processo associato all'esecuzione della SelfLet e nel successivo aggiornamento del Cloud Pool, eliminando l'id della SelfLet terminata. La successiva operazione di tear down prevede invece il ripristino delle condizioni iniziali della macchina remota, attraverso la rimozione del progertto di configurazione della SelfLet.

Il Cloud Manager invia infine una richiesta al servizio di cloud per il rilascio della macchina virtuale, aggiornando il Cloud Pool al termine dell'operazione.

replace Cloud SelfLet Service

Il servizio replaceCloudSelfletService implementa il processo di sostituzione di una Cloud SelfLet all'interno di una macchina virtuale attiva. Questo servizio è stato introdotto per andare incontro ai requisiti di costo e di prestazioni. Il primo requisito è dovuto al modello di costo adottato dai servizi di cloud computing. In generale questo modello prevede un costo su base oraria; il prezzo pagato in seguito alla richiesta di allocazione di una nuova macchina virtuale è associato all'intervallo di un'ora di vita di esecuzione della VM pertanto, all'interno di una finestra temporale inferiore all'ora, risultata economicamente sconveniente rilasciare l'istanza. Il tempo necessario all'avvio di una nuova macchina virtuale è un altro fattore da considerare. Questo periodo, in generale, varia dipendentemente dal tipo di istanza richiesta ed è proporzionale alle prestazioni richieste per l'istanza. Mantenere attiva una macchina virtuale e riutilizzarla come sede di esecuzione di un nuovo nodo della rete è una scelta progettuale per il miglioramento delle prestazioni complessive del sistema.

Come accennato nella sezione 4.3.4, i servizi operativi di alto livello sono costruiti come composizione di sotto-servizi elementari. Il servizio di sostituzione è quindi implementato come unione di un sottoinsieme ordinato delle operazioni che compongono i servizi di creazione e terminazione di un nodo. Da questa successione sono escluse le operazioni per la richiesta e il rilascio di macchine virtuali.

Alcune condizioni governano la modalità di esecuzione del servizio. Queste condizion sono associate alla necessità di esecuzione dell'intero ciclo di sostituzione oppure di una sua parte, per esempio la sola terminazione di una Cloud SelfLet mantenendo attiva la relativa macchina virtuale, oppure l'avvio di un nuovo nodo su una macchina virtuale non configurata ma ancora attiva in seguito a un recente utilizzo.

L'impostazione di quesete condizioni è dipendente delle politiche politiche che governano il comportamento del Cloud Manager, e sarà pertanto esposta nella sezione 5.10.2.

5.5 Cloud Logger

Una volta completata l'implementazione dei servizi preposti alla realizzazione della fase di esecuzione del Cloud Manager, è stato possibile avviare e testare i primi prototipi di Cloud SelfLet. L'esecuzione remota di questi componenti non rendeva possibile la verifica della correttezza delle operazioni di deploy, se non interagendo manualmente con la macchina virtuale avviata. Si è reso quindi necessario sviluppare un componente finalizzato al monitoraggio e al recupero delle informazioni relative alle SelfLet le cui esecuzioni avvenivano all'interno della cloud. Tale strumento è stato in seguito utilizzato per la raccolta dei risultati relativi ai casi di test descritti nel Capitolo 6. Come si vede dal diagramma delle classi mostrato in figura 5.13, l'elemento centrale è il LogChecker. Questo componente espone i metodi per la gestione degli InfoItem, che rappresentano le informazioni di log caratterizzate da un tipo, un messaggio, l'id del produttore, id del messaggio, e dall'istante di tempo in cui il messaggio è stato prodotto. La classe InfoItem implementa l'interfaccia Serializable così che sia possibile spedire i log ad un server.

LogCheker è una classe astratta ed è implementata da LogChecker_Client e LogChecker_Server. Il client può essere di due tipi: singleProcess o netClient, rispettivamente preposti all'inoltro degli InfoItem ad un server la cui esecuzione ha luogo sulla macchina locale oppure ad un server remoto. L'oggetto NetServer istanzia un LogChecker_Server, per intercettare l'arrivo dei messaggi di log, e un pool di monitor. Ogni volta che un nuovo client si iscrive al server iniziando a comunicare le proprie informazioni di log, il server aggiunge un nuovo monitor al pool di monitor per la visualizzazione dei messaggi relativi

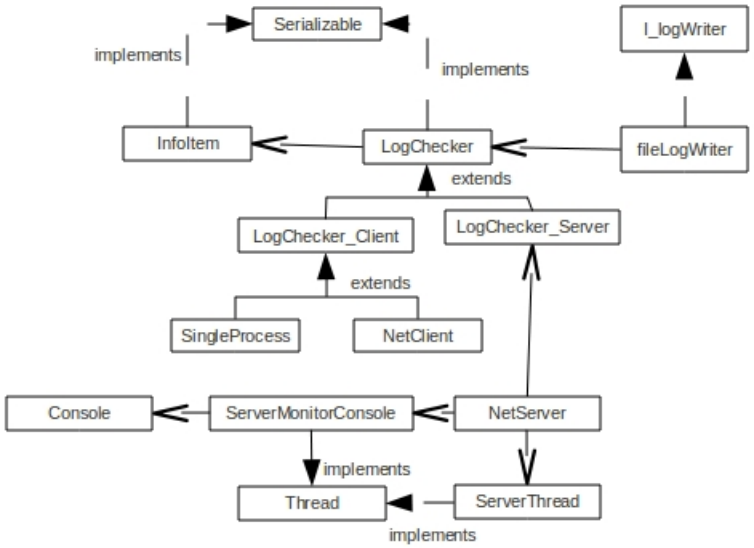

Figura 5.13: Diagramma delle classi del framework CloudLogger

a quel client. I monitor sono dei Threads che permettono la visualizzazione all'utente dei messaggi di log inviati dai client, e sono realizzati attraverso una semplice interfaccia SWING, mostrata in figura 5.14.

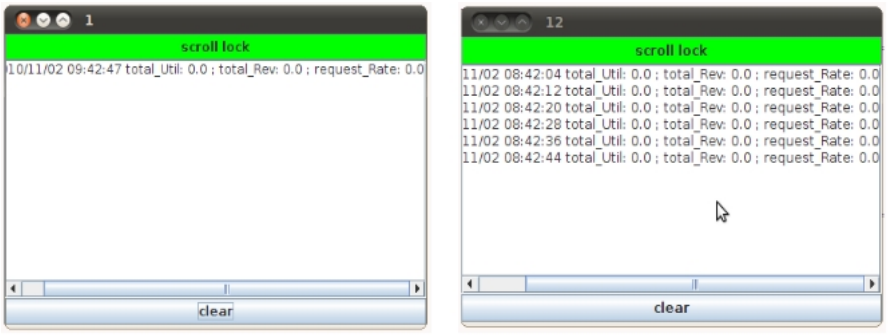

Figura 5.14: Esempio di interfaccia dei Monitor

Ultimata l'implementazione del Cloud Logger, il passo successivo è stata la sua integrazione con le SelfLet. Ai parametri specificati nella sezione general knowledge del file di descrizione XML del Cloud Manager, sono stati aggiunti quelli relativi alla configurazione del Cloud Logger Client riportati di seguito:

Queste informazioni, nel caso il parametro debug sia impostato a true, vengono aggiunte all'interno del file di descrizione XML della Cloud SelfLet che si sta configurando, che le userà per l'inoltro dei propri messaggi di log. Successivamente sono state introdotte due abilities: CloudLogClientAbility

```
<!-- parametri per la creazione di un logClient di test -->
<selfLetProperty name="debug" type="boolean"/>
<selfLetProperty name="logClientFilePath" type="string"/>
<selfLetProperty name="logServerIpAddress" type="string" />
<selfLetProperty name="logServerPort" type="integer"/>
```

Figura 5.15: Proprietà di configurazione del Cloud Manager

e CloudLogServerAbility, descritte nella sezione precedente e il cui utilizzo all'interno dei servizi di esecuzione è mostrato nella figura 5.16.

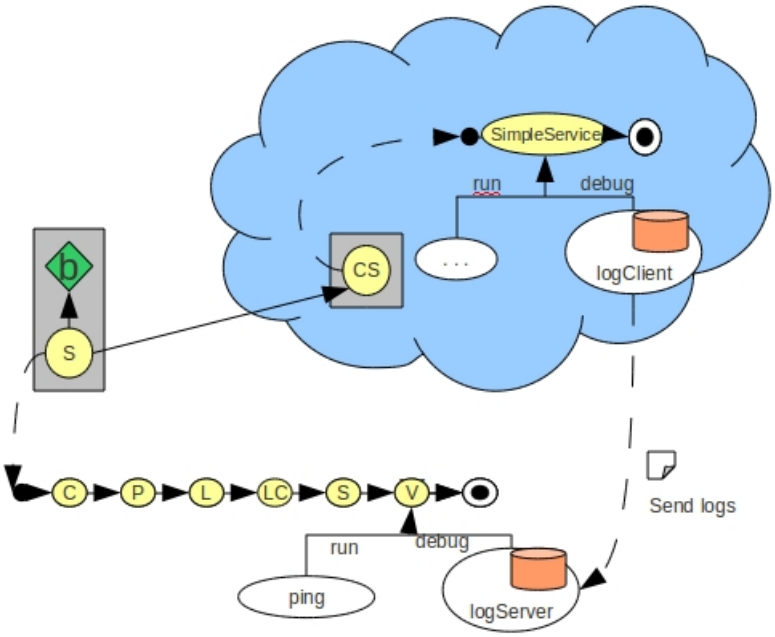

Figura 5.16: Meccanismo di utilizzo del cloud logger

Infine è stata implementata un'estensione della classe PeriodicLoggerThread presente all'interno del package Service.Utilization del SelfLet engine. Questa classe, che si occupa della produzione di un file di log contenente i dettagli di esecuzione di una SelfLet, è stato esteso con la possibilità di inoltrare questi log ad un server remoto attraverso il meccanismo di logging implementato dal framework Cloud Logger.

5.6 Cloud Manager: policies

Nelle sezioni precedenti sono stati descritti gli strumenti finalizzati all'interazione del Cloud Manager con il servizio di cloud computing, all'attuazione

delle operazioni di deploy delle Cloud SelfLets e al monitoraggio dei dettagli relativi all'esecuzione di queste ultime attraverso un meccanismo di logging remoto. Prima di descrivere i dettagli relativi alle politiche interne ad ogni nodo della rete di SelfLets, e ai meccanismi che portano alla decisione di avviare o terminare una Cloud SelfLet, è necessario analizzare il modello complessivo adottato dal sistema per i processi di coordinamento e gestione della popolazione dei nodi all'interno della cloud.

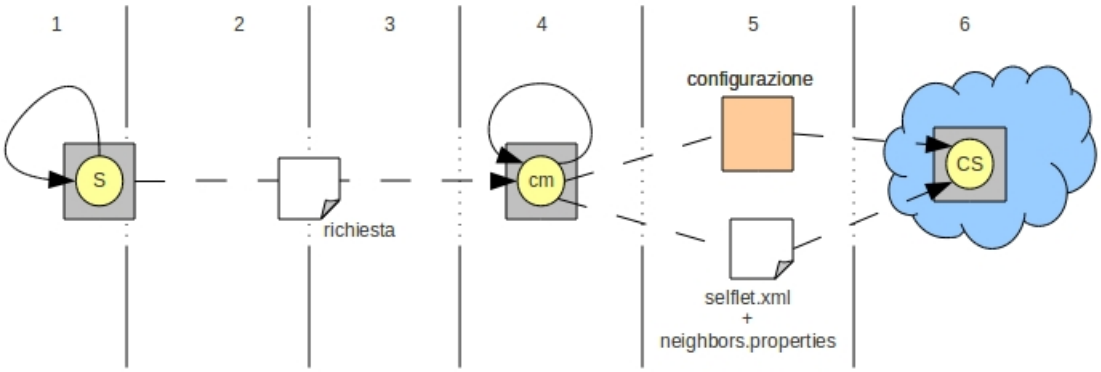

Figura 5.17: Processo di attivazione di una Cloud SelfLet

La figura 5.17 mostra la sequenza di passi che portano alla creazione di una Cloud SelfLet. Nella prima fase una SelfLet con attiva la politica di ottimizzazione su cloud monitora il proprio stato e verifica che siano soddisfatte le condizioni per sollevare una richiesta di espansione su cloud. Successivamente prepara ed inoltra la richiesta notificando le proprie caratteristiche. Il Cloud Manager riceve la richiesta che viene valutata in base allo stato attuale dei nodi già presenti nella cloud. Nel caso risultino verificate le condizioni per la creazione di un nuovo nodo, il manager attua le operazioni di deploy per l'attivazione di una nuova SelfLet, il cui ciclo di vita è indipendente da quello del nodo che ha sollevato la richiesta.

Durante la propria esecuzione, una Cloud SelfLet monitora il proprio stato ed eventualmente, in accordo alle proprie politiche di ottimizzazione, solleva richieste di espansione verso il Cloud Manager per la creazione di nuovi nodi. Una Cloud SelfLet verifica inoltre le condizioni per sollevare una richiesta di terminazione. Tipicamente queste condizioni sono soglie sul carico minimo di lavoro svolto dalla SelfLet, oppure il raggiungimento di un determinato obiettivo. Il processo che porta alla terminazione di un nodo è mostrato in

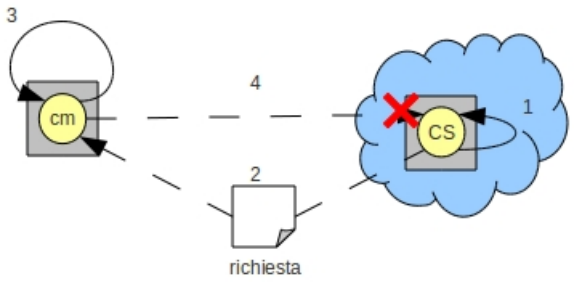

Figura 5.18: Processo di terminazione di una Cloud SelfLet

figura 5.18. Verificate le condizioni, la Cloud SelfLet solleva una richiesta di terminazione che viene inviata al Cloud Manager. Il manager valuta la richiesta e sceglie la strategia di spegnimento del nodo che l'ha sollevata: può terminare la SelfLet e rilasciare la macchina virtuale che ne ospita l'esecuzione, oppure mantenere attiva tale macchina per riutilizzarla caricando la configurazione di una nuova Cloud SelfLet, in accordo alle richieste di espansione ricevute.

5.7 SelfLet optimization policy

In generale il contesto di esecuzione delle SelfLets è altamente dinamico e il carico di lavoro di ogni nodo può variare per molte ragioni. Per esempio alcune SelfLets potrebbero aggiungersi alla rete oppure cadere a causa di un guasto, inoltre la frequenza delle richieste per i diversi servizi può variare per cause dovute a fattori esterni. Nell'ambito di un precedente lavoro di ricerca [37] è stato proposto un modello di predizione del livello di utilizzo della CPU da parte di una SelfLets che esegue un insieme di servizi. In particolare, la predizione del contributo in termini di utilizzo del servizio s per la SelfLet n è stato definito dalla seguente formula:

$$\hat{U}_n^s = f_{U_n^s}(\Lambda_n^s) \tag{5.1}$$

dove Λ_n^s è la predizione della frequenza delle richieste del servizio s alla SelfLet n, calcolato come funzione delle passate ω osservazioni:

$$\Lambda_n^s = f_{\Lambda_n^s}(\Lambda_n^s(t), \Lambda_n^s(t-1), \ldots, \Lambda_n^s(t-\omega)) \tag{5.2}$$

L'obiettivo della ricerca era l'introduzione di un insieme di politiche finalizzate alla ridistribuzione dell'utilizzo delle risorse tra le SelfLets che fanno parte di una rete. Per il raggiungimento di questo obiettivo sono state definite due soglie: $\bar{U}_{min}$ e $\bar{U}_{max}$. Il valore di queste soglie di utilizzo è stato definito in accordo alla teoria delle code [44] e riflette la capacità della SelfLet di completare l'esecuzione dei servizi richiesti entro un certo periodo di tempo; un alto livello di utilizzo della CPU implica tipicamente un alto tempo di risposta. Una SelfLet può trovarsi in uno dei seguenti stati in base alla corrente predizione del livello di utilizzo calcolato in (4.1):

- Normale: $\bar{U}_{min} < \hat{U}_n < \bar{U}_{max}$
- Sopra-utilizzata: $\hat{U}_n \geq \bar{U}_{max}$
- Sotto-utilizzata: $\hat{U}_n \leq \bar{U}_{min}$

In [37] è data la definizione del guadagno ottenuto da una SelfLet nell'esecuzione di un servizio entro un certo tempo di risposta indicato con $\bar{R}$. In caso di violazione di questa soglia, il service provider non genera alcun valore. Il guadagno è proporzionale alla frequenza di esecuzione dei servizi ed è definito per una SelfLet n che esegue un insieme di servizi s implementati dal behavior b:

$$netRev_n = \sum_{s \in S_n} (a_s^n (P[R < \bar{R}]) - c_n^{s,b}) \Lambda_n^s T \forall n \in \mathcal{N} \tag{5.3}$$

In accordo a questo modello, sono state individuate le seguenti azioni di ottimizzazione: Change service implementation, Service redirect e Service teach. La politica autonomica di ottimizzazione, attivata periodicamente (ogni 60 secondi), monitora lo stato della SelfLet n confrontandone il valore di utilizzo con i valori di soglia e, se necessario, attiva l'azione appropriata. Per evitare fluttuazioni nelle variazioni dell'utilizzo da parte dei nodi della rete, la politica monitora anche lo stato dei vicini; l'insieme dei vicini del nodo n è indicato con $\mathcal{N}_n$. In questo modo la scelta dell'azione di ottimizzazione, pur rimanendo locale, viene presa considerando il livello di utilizzo del sistema piuttosto che del singolo nodo. Una SelfLet può adottare una delle seguenti strategie, dipendentemente dallo stato del sistema monitorato:

Greedy : ha come obiettivo la massimizzazione del guadagno di una SelfLet senza tenere conto degli effetti sulle altre SelfLets. Questa strategia è adottata quando la SelfLet n e i suoi vicini sono entrambi in uno stato di esecuzione normale, ovvero con un livello di utilizzo entro di valori di soglia:

$$\bar{U}_{min} < \hat{U}_m < \bar{U}_{max} \forall m \in \mathcal{N}_n \cup \{n\} \tag{5.4}$$

Non-Greedy : l'obiettivo è ancora quello di massimizzare il guadagno; l'azione da eseguire viene selezionata in base all'impatto che ha sulle altre SelfLets. In particolare vengono scartate quelle azioni che comportano la violazione di una delle soglie da parte dei nodi vicini. Questa strategia è adottata da una SelfLet n se e solo se:

$$\exists m \in \mathcal{N}_n \cup \{n\} t.c. \bar{U}_{min} < \hat{U}_m < \bar{U}_{max} \tag{5.5}$$

In base alla strategia, viene generato un insieme di azioni candidate, caratterizzate dai valori di predizione del guadagno ottenuto dalla SelfLet che esegue l'azione, dalla predizione del livello di utilizzo della CPU e dal costo di attuazione dell'azione. L'insieme delle azioni candidate viene poi suddiviso in quattro insiemi secondo i seguenti criteri:

- Profittevoli P = $\{a \in candidate|(a) > 0\}$
- non profittevoli NP = $\{a \in candidate|(a) \leq 0\}$
- sovraccaricano i vicini ON = $\{a \in candidate|\exists m \in \mathcal{N}_n, \hat{U}_m > \bar{U}_max\}$
- sovraccaricano il nodo locale OL = $\{a \in candidate|U_n > \bar{U}_max\}$

Si noti che la validità di $P \cap NP = \emptyset$ e $ON \cap OL \neq \emptyset$

L'algoritmo 1 mostra la politica di load balancing. Viene innanzitutto stabilita la strategia adottata, e vengono generati gli insiemi di azioni di ottimizzazione. L'unione di questi insiemi viene poi classificata in base alle caratteristiche di profitto ed overloading dei nodi. Infine, sulla base della strategia, viene selezionata l'azione di ottimizzazione da attuare. Per i dettagli sugli algoritmi di selezione della migliore azione non-Greedy e della migliore azione Greedy si rimanda a [37].

Algorithm 1 Politica di load balancing della SelfLet n.

```
1: strategy ← getCurrentStrategy();
2: if {strategy == noAction} then
3:     return;
4: candidateActions ← actionsChangingImplementation() ∪
       actionsRedirectingRequests() ∪ actionTeachingServices();
5: P ← getProfitableActions(candidateActions);
6: NP ← getNonPrifableAction(candidateActions);
7: ON ← getOverloadNeighbours(candidateActions);
8: OL ← getOverloadLocal(candidateActions);
9: if {strategy == greedy} then
10:    a ← pickBestGreedyAction(P, ON, NP, OL);
11: else
12:    a ← pickBestNon-GreedyAction(P, ON, NP, OL);
13: actuateAction(a);
```

5.8 SelfLet cloud optimization policy

In accordo al modello descritto nella sezione precedente, il SelfLet framework prevede una politica di ottimizzazione attivabile settando a true la proprietà enableOptimizationPolicy del file di descrizione XML. Nel corso di questo lavoro, è stata estesa l'architettura del motore di ottimizzazione implementato dal framework con lo scopo di permettere alle SelfLets di sollevare richieste di espansione e contrazione per il cloud, secondo il modello descritto nella sezione 5.6. Il processo di estensione del motore è suddiviso in due fasi: l'introduzione di due nuovi tipi di azioni di ottimizzazione e l'introduzione della politica di ottimizzazione su cloud. Nel seguito sarà presentato questo processo.

5.8.1 Estensione dell'architettura

La struttura del progetto che implementa i componenti che costituiscono una SelfLet, così come è stata descritta nella sezione 3.1.1, è organizzata in package secondo un criterio funzionale. I packages selflet.optimization e selflet.optimization generators contengono le classi che realizzano il motore di

ottimizzazione del SelfLet framework. Il diagramma delle classi è mostrato in figura 5.19.

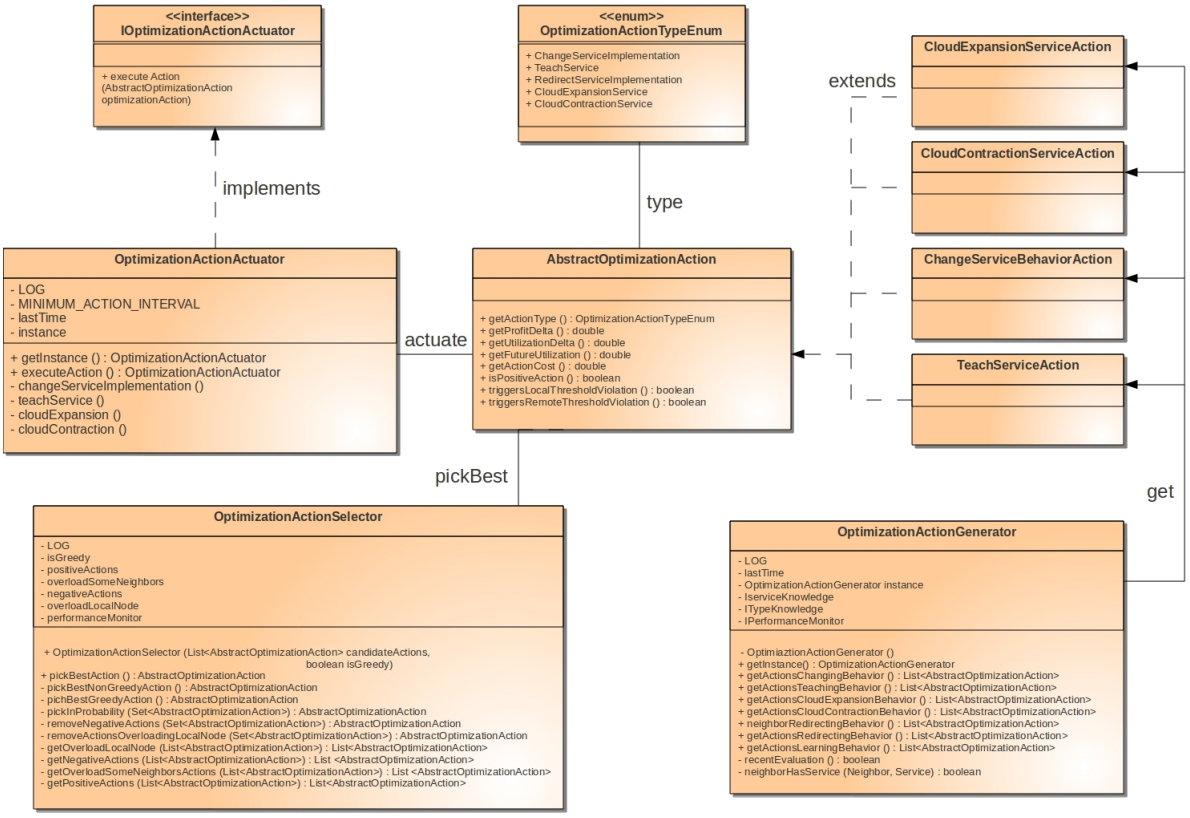

Figura 5.19: Class diagram del motore di ottimizzazione

Gli elementi principali sono AbstractOptimizationAction, OptimizationActionGenerator, OptimizationActionSelector e OptimizationActionActuator. Il primo componente è una classe astratta che rappresenta una generica azione di ottimizzazione; la classe viene estesa dall'effettiva implementazione delle azioni di ottimizzazione previste. Il secondo è il componente preposto alla generazione delle azioni di ottimizzazione ed espone i metodi che permettono la creazione degli insiemi relativi ad una certa categoria di azione. OptimizationActionSelector si occupa della selezione dell'azione considerata più vantaggiosa nel rispetto della strategia adottata. L'attuatore è l'elemento che permette l'effettiva realizzazione dell'azione di ottimizzazione scelta.

Le azioni introdotte sono relative alla possibilità di una SelfLet di poter sollevare una richiesta di espansione o contrazione e sono implementate dalle classi CloudExpansionServiceAction e CloudContractionServiceAction. Queste clas-

si estendono AbstractOptimizationAction allo stesso modo delle altre azioni di ottimizzazione già previste dal framework. L'invocazioone del metodo getActionType esposto restituisce un opportuno valore di enumerazione, OptimizationActionTypeEnum, anch'esso aggiornato con l'introduzione delle due nuove tipologie. Sono stati introdotti due metodi al componente ActionGenerator per la produzione dei nuovi insiemi di azioni. Infine è stato aggiornato l'attuatore perché possa realizzare il tipo di azione selezionata. Allo stato attuale dell'implementazione, le richieste di espansione da parte di una SelfLet hanno come risultato l'introduzione nella rete di un nuovo nodo clone del precedente. L'azione generata dal Action generator è quindi una, quella relativa alla semplice clonazione. Come descritto nel Capitolo 7, una futura estensione è la possibilità di una SelfLet di sollevare richieste di espansione per un sottoinsieme dei servizi conosciuti. L'architettura presentata permette una facile estensione in questa direzione. Si prevede che le future azioni vengano modellizzate nel rispetto del modello presentato nella sezione 5.7, pertanto il componente di selezione OptimizationActionSelelctor non è stato modificato. Infine, tra gli eventi disponibili sollevati internamente a una SelfLet, è stato introdotto un nuovo evento CloudOptimizationPolicyEvent che, come mostrato nella sezione seguente, viene sollevato da un componente di monitoring per l'avvio di un ciclo di gestione della politica di ottimizzazione.

5.8.2 Innesto della politica cloud optimization

Il servizio di cloud computing è in generale offerto dagli infrastructure provider come servizio a pagamento. La richiesta di espansione o contrazione da parte di una SelfLet ha l'effetto di modificare la topologia della rete cui la SelfLet appartiene. Per questi motivi, l'introduzione delle azioni di ottimizzazione su cloud rappresentano una possibilità che non deve essere in stretta alternativa alle azioni di ottimizzazione esistenti, ma deve accompagnare queste azioni nell'obiettivo di un miglioramento complessivo delle performance del sistema o di garantire un certo Service Agreement Level. A questo scopo, la politica di ottimizzazione su cloud è stata sviluppata come politica distinta dalla politica di ottimizzazione già presente, pur seguendone il naturale funzionamento. Lo schema del file di descrizione XML è stato modificato con l'introduzione di una nuova proprietà: enableCloudOptimizationPolicy; Settando a

true questo valore, la politica di ottimizzazione su cloud viene periodicamente attivata. I componenti preposti all'attivazione della SelfLet: SelfLetLauncher e SelfLetInstance, parte del SelfLet framework e contenuti nel package selflet, sono stati aggiornati per la corretta parsificazione del nuovo schema; allo stesso modo è stato modificata la classe SelfletProperties del package selflet.utilities.parser che mappa la relativa porzione dello schema XML in un oggetto Java. L'insieme dei componenti finalizzati alla gestione di monitoring delle performance e del livello di utilizzo di una SelfLet sono contenute nel package selflet.service.utilization. Il sequence diagram che descrive la sequenza di interazioni tra questi componenti, che porta all'effettiva esecuzione della politica di ottimizzazione, è mostrato in figura 5.20.

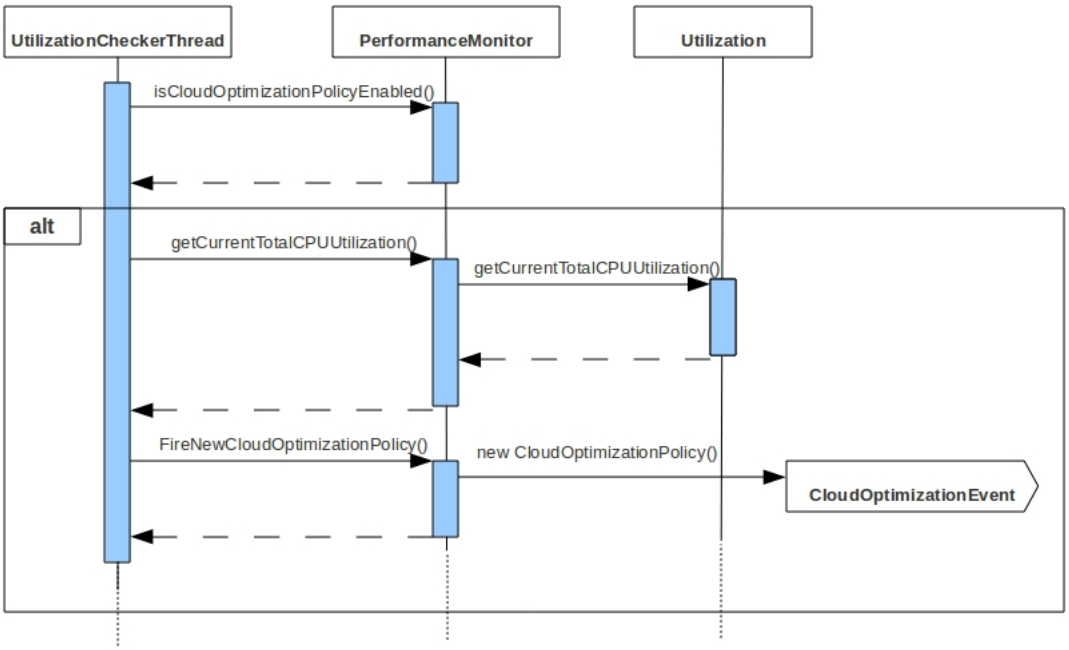

Figura 5.20: Sequence diagram di attuazione proattiva della politica

All'inizio della propria esecuzione, una SelfLet avvia un UtilizationCheckerThread, che si occupa di monitorare periodicamente il livello di utilizzo della SelfLet. Se la politica di ottimizzazione su cloud è attiva, il thread solleva un evento CloudOptimizationPolicy settandone la proprietà CurrentTotalNodeUtilization con il corrente valore di utilizzo rilevato dal PerformanceMonitor. Questo tipo di controllo è definito proattivo, ovvero viene effettuato senza che lo stato della SelfLet determini una condizione di emergenza tale da richiedere

l'attuazione di una politica di ottimizzazione.

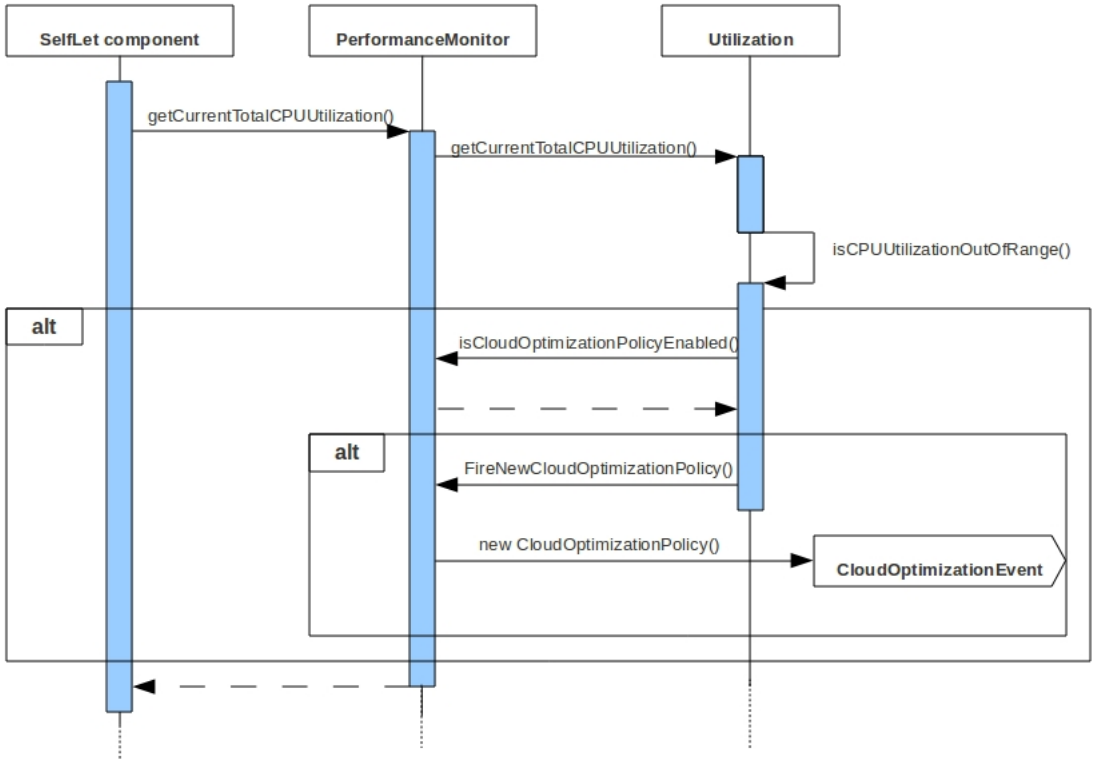

Figura 5.21: Condizione di emergenza di attivazione della politica

Il PerformanceMonitor viene arbitrariamente utilizzato da un generico componente interno alla SelfLet per la verifica del livello di utilizzo di risorse da parte della SelLet; tra questi il neighborStateManager, che si occupa di gestire lo scambio di informazioni sullo stato tra il nodo e i propri vicini, o il periodicLoggerThread che periodicamente produce un messaggio di log sullo stato dell'esecuzione della SelfLet. Quando questo accade, il monitor invoca il metodo getCurrentTotalCPUUtilization del sotto-componente Utilization, che procede al calcolo effettivo del livello di utilizzo. Se il calcolo produce un risultato al di fuori dei valori di soglia previsti, viene immediatamente sollevato un evento CloudOptimizationPolicy per la gestione della situazione anomala.

L'evento così sollevato viene raccolto dal dispatcher interno alla SelfLet che si occupa della comunicazione tra i componenti, e viene inoltrato al gestore delle politiche che attiva così la politica definita nel file di regole Drools. La gestione della politica è definita nelle sezioni precedenti.

5.9 Comunicazione tra SelfLet e Cloud Manager

All'avvio di una SelfLet, il file XML che la descrive viene parsificato e mappato da un oggetto che viene memorizzato nella base di conoscenza. L'attuazione di un'azione di ottimizzazione su cloud avvia l'esecuzione di un servizio di notifica, che recupera le informazioni necessarie alla gestione della richiesta da parte del Cloud Manager. Come mostrato in figura 5.22, il servizio notifyCloudManagerService recupera le informazioni relative al livello di utilizzo, al guadagno e al tipo di azione attuata, produce una lista degli identificativi dei propri vicini e prepara una rappresentazione in formato stringa dellla mappa del file XML memorizzata all'avvio. Questi valori vengono inviati al Cloud Manager come parametri del servizio registerSelfLetService offerto in modalità CanDo. Le operazioni di produzione e preparazione dei parametri sono implementate in ReflectionAbility. La stessa ability viene successivamente invocata dal Cloud Manager per l'unmarshalling delle informazioni ed il loro successivo utilizzo.

Figura 5.22: Trasferimento del file di descrizione XML

5.10 Cloud Manager: autonomic manager

Come è stato descritto nella sezione 5.4, il componente preposto alle operazioni di gestione e controllo della cloud, il Cloud Manager, è stato progettato sul modello generale di autonomic element proposto da IBM. Nella sezione 4.3.3 è stato individuato il servizio cloudManagerService come principale responsabile dell'azione del manager. Il servizio implementa il ciclo di controllo MAPE per la gestione dei cloud elements. La fase di esecuzione (E) è realizzata dalle oper-

azioni di delpoy su cloud discusse precedentemente; nel seguito sarà presentata l'architettura dell'autonomic manager implementato dal ciclo MAPE.

5.10.1 Architettura

La versione del SelfLet framework utilizzata nel corso di questo lavoro di tesi non implementa la funzionalità di esecuzione parallela dei servizi da parte di una SelfLet. Ciò impedisce l'abilità di una SelfLet che svolge la funzione di service provider, di servire una richiesta per l'esecuzione di un determinato servizio da, nel caso stia già eseguendo un servizio. Il problema si verifica in particolare nel caso di servizi il cui diagramma dei behavior è un ciclo. Per questo motivo il cloudServiceManager descritto nella sezione 4.3.3 è stato modificato come mostrato in figura 5.23.

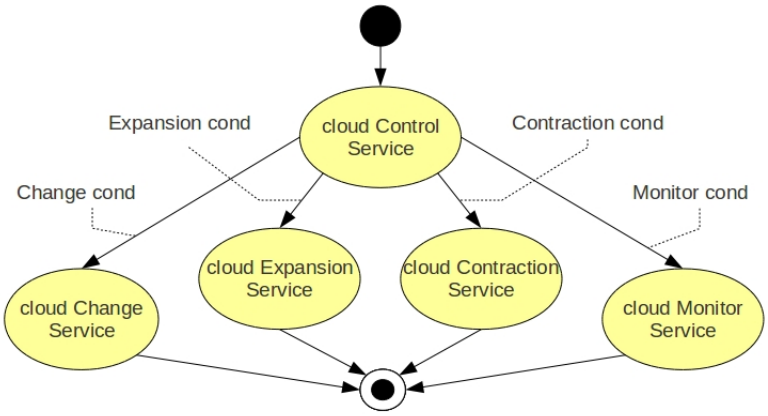

Figura 5.23: Behavior diagram di cloud Manager Service modificato

L'implementazione del ciclo MAPE, necessaria alla corretta gestione dei cloud elements, è stata realizzata distribuendolo nella politica di ottimizzazione su cloud delle SelfLet che sollevano le richieste. In questo modo, quando una SelfLet non seleziona alcuna azione di ottimizzazione, solleva al manager una richiesta di tipo monitor. Ricevuta la richiesta, il manager esegue un'iterazione del ciclo. Dallo schema del servizio così modificato, è possibile notare che i servizi newCloudSelfletService, removeCloudSelfletService e replaceCloudSelfletService sono stati sostituiti con i servizi cloudExpansionService, cloudContractionService e cloudChangeService. I nuovi stati rappresentano una generalizzazione dei servizi precedentemente previsti, che vengono così

invocati come sotto-servizi. Tale modifica permette il disaccoppiamento delle fasi di monitoring (M), analisi (A) e pianificazione (P) dalla fase di esecuzione, e potrà essere utilizzata per future estensioni così come descritto nel Capitolo 7.

Il servizio cloudControlService è stato ulteriormente dettagliato nei sotto-serivizi mostrati in figura 5.24.

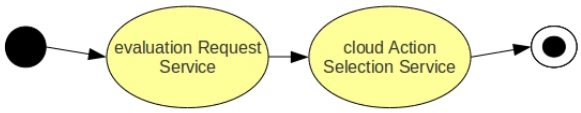

Figura 5.24: Behavior diagram di cloud Control Service

CloudRequestEvaluation è destinato alla selezione della richiesta da servire tra quelle ricevute dal Cloud Manager; L'attuale implementazione del servizio adotta la strategia First-Come First-Served. CloudActionSelector è invece il servizio che si occupa, data una richiesta, di scegliere il comportamento più appropriato del Cloud Manager in relazione allo stato dei cloud elements e ai parametri di progetto specificati nel proprio file di descrizione XML; l'algoritmo di decisione è dettagliato nella sezione successiva. Nel seguito di questa sezione sarà descritto il componente Cloud Pool.

Cloud Pool

La gestione dei cloud elements rappresenta un livello di servizio differente dalla normale interazione tra i nodi del sistema. In normali condizioni di funzionamento, una Cloud SelfLet per cui risultino verificate le condizioni di contrazione, invia un'opportuna richiesta al Cloud Manager. Nel caso in cui si verifichi un guasto, la SelfLet potrebbe non comunicare il problema al manager, trattenendo la macchina virtuale. Il Cloud Manager deve pertanto avere la possibilità di gestire il ciclo di vita delle macchine virtuali e del software caricato, indipendentemente dallo stato di esecuzione delle Cloud SelfLets. Per fare questo è stato introdotto un pool per la gestione della popolazione degli elementi in esecuzione all'interno della Cloud. Il Cloud Pool è una mappa di stringhe la cui gestione avviene attraverso la chiamata degli opportuni metodi definiti nella MonitorAbility. Ogni chiave della mappa è l'identificativo di una

macchina virtuale; il valore associato a una chiave è l'identificativo della SelfLet in esecuzione sulla macchina virtuale. Quando il Cloud Manager richiede una VM al servizio di cloud, inserisce l'id all'interno del pool associandolo ad un valore di default che indica che la fase di configurazione della VM è in fase di completamento. Alla fine di questa fase, il valore viene aggiornato con l'id della Cloud SelfLet appena avviata. Quando l'esecuzione termina, l'id del nodo viene settato a null per rappresentare un'istanza vuota. A seconda della strategia di contrazione, il manager può mantenere attiva l'istanza, e utilizzarla in un secondo momento per la configurazione di una nuova Cloud SelfLet senza la necessità di avviare una nuova macchina virtuale. Nel caso in cui il menager decida di terminare la VM, la relativa chiave viene rimossa dal Pool. La gestione centralizzata dei cloud elements da parte del Cloud Manager permette l'utilizzo del pool per azioni di ottimizzazione o di recovery delle istanze.

5.10.2 Algoritmi

Il listato 2 mostra l'algoritmo di decisione implementato dal cloudActionSelector rispetto all'azione da attuare in seguito a una notifica da parte di una SelfLet.

Inizialmente viene valutato il tipo della notifica. Nel caso di una richiesta di tipo monitor, il manager esegue un'iterazione del ciclo di monitoring. Se la richiesta è di tipo expansion, il nodo che l'ha sollevata necessita di un'operazione di espansione; il manager controlla il Cloud Pool e verifica l'esistenza di una macchina virtuale attiva vuota e, nel caso questa sia presente, invoca il servizio cloudChangeService dopo aver settato a fill il valore della condizione utilizzata dal servizio change per la valutazione del percorso da seguire nel diagramma dei behavior. Se nessuna istanza vuota è presente, il manager valuta le condizioni sul costo del servizio di cloud computing. Se la somma del costo totale (dovuto alle VM attive) con il costo previsto per l'attivazione di una nuova istanza è inferiore al valore di soglia definito nel file di descrizione come parametro di progetto, viene invocato il servizio cloudExpansionService. Nel caso in cui si sia ricevuta una richiesta di contrazione il manager verifica il tempo di esecuzione della VM associata alla richiesta. Se questo periodo, normalizzato all'ora corrente, è inferiore alla dimensione della finestra tempo-

Algorithm 2 Selezione dell'azione del Cloud Manager.

```
1: request ← getSelectedRequest();
2: if {request.getType() == monitor} then
3:     selectedAction ← monitor;
4: else if {request.getType() == expansion} then
5:     numEmptyInstances ← countEmptyInstances();
6:     if {numEmptyInstances > 0} then
7:         instanceId ← getYoungerEmptyInstance();
8:         selectedAction ← change;
9:     else
10:        totalCost ← getHourTotalCost(baseCost);
11:        if {totalCost + baseCost ≤ costThreshold} then
12:            selectedAction ← expansion;
13: else if {request.getType() == contraction} then
14:    instanceId ← request.getInstance();
15:    timeLife ← getHourTimeLife(instanceId);
16:    if {timeLife < timeThreshold} then
17:        selectedAction ← change;
18:    else
19:        selectedAction ← contraction;
20: needService(selectedAction);
```

rale espressa nel file XML come parametro di progetto, il Cloud Manager imposta ad empty il valore della condizione di change e invoca il servizio changeExpansionService che in questo caso rimuoverà la sola SelfLet. Se le condizioni sulla finestra temporale non sono verificate, il servizio invocato è cloudContractionService che rimuove la macchina virtuale.

L'attuale implementazione dell'algoritmo implementato dal servizio cloudMonitorService è mostrata nel listato 3.

Ad ogni ciclo di monitoring, il manager controlla il Cloud Pool individuando, se presente, l'istanza di macchina virtuale attiva, non utilizzata, per cui è maggiore la differenza tra il tempo di esecuzione normalizzato e la dimensione

Algorithm 3 Ciclo di monitoring del Cloud Manager.

```
1: olderEmptyInstance ← getOlderEmptyInstance();
2: if {olderEmptyInstance! = null} then
3:     timeLife ← getHourTimeLife(olderEmptyInstance);
4:     if {timeLife > timeThreshold} then
5:         instanceId ← olderEmptyInstance
6:         needService(contraction);

7: totalCost ← getHourTotalCost(baseCost);
8: if {totalCost > costThreshold} then
9:     needService(reset);
```

della finestra temporale definita come parametro di progetto nel file XML. Se il tempo di esecuzione è maggiore di questo parametro, il manager rilascia la macchina virtuale e ne rimuove l'id dal pool. In questo modo si evita di pagare il servizio per un'altra ora di utilizzo delle macchine virtuali non utilizzate. Durante il ciclo di monitoring il manager effettua una verifica sul costo totale dovuto alle VM attualmente attive. Se il costo è maggiore della soglia impostata come parametro di progetto, il manager considera il sistema in una condizione anomala, pertanto resetta la popolazione di nodi all'interno della cloud.

Implementazioni future di questo servizio dovranno prevedere operazioni di recovery dei nodi non funzionanti, come spiegato nel Capitolo 7.

5.11 Estensioni selflet framework

Il SelfLet framework è un progetto di ricerca attualmente in fase di sviluppo. Nel corso di questo lavoro sono stati individuati alcuni problemi e alcune funzionalità non implementate rispetto a quanto previsto dal modello di funzionamento teorizzato. Altre funzionalità sono invece state individuate come necessarie per la realizzazione di questo lavoro e non erano inizialmente previste nel framework. Segue una breve panoramica degli aspetti che sono stati affrontati.

Relazione di vicinanza tra nodi

Il primo aspetto che si è dovuto affrontare è legato alle variazioni della tipologia della rete dovuto alla creazione e alla distruzione dinamica dei nodi che ne fanno parte. L'insieme dei vicini di una SelfLet n, settato nella fase di inizializzazione, non veniva poi aggiornato durante la sua esecuzione. La mancanza dell'aggiornamento era causa di diversi inconvenienti. Primo tra questi la mancata rimozione dall'elenco dei nodi che non rispondono alla richiesta di aggiornamento delle informazioni sullo stato. con la conseguente occupazione di memoria e perdita di performance dovuta ad ulteriori tentativi di comunicazione con un nodo non più parte della rete. Altra importante conseguenza era l'asimmetria della relazione di vicinanza tra nodi. Nel momento in cui una nuova SelfLet richiede lo stato di un nodo definito come suo vicino questo, se attivo, notifica il proprio stato ma non aggiunge la SelfLet nell'elenco di propri vicini. La risoluzione di questi problemi ha permesso la corretta collaborazione tra i nodi già esistenti con quelli gestiti dal Cloud Manager. Per ottenere questo comportamento sono state apportate alcune modifiche al package selflet.negotiation del SelfLet framework.

Invio dei parametri necessari a un servizio richiesto in modalità Do

La necessità di introdurre un servizio di notifica al Cloud Manager per la richiesta delle operazioni di espansione e contrazione ha fatto emergere la mancata gestione dei parametri inviati al service provider, che venivano inviati senza però risultare disponibili alle actions, non permettendo la corretta esecuzione del servizio. il metodo executeAchievableService() della clase NegotiationEventReceiver è stata aggiornata con la memorizzazione dei parametri ricevuti all'interno della general knowledge, così che siano accessibili da parte della acions.

Esecuzione remota di un servizio disponibile localmente

La preparazione dei casi di test, che sarà descritta nel seguente capitolo, ha portato a individuare una mancanza nella modalità di esecuzione di un servizio. In particolare in un'applicazione per cui un insieme di SelfLet che offrono uno stesso servizio, la cui esecuzione da parte di ognuna comporta un differente risultato, può prevedere la necessità di richiedere l'esecuzione remota di questo

servizio da parte di una SelfLet per cui il servizio è disponibile localmente. Il problema è stato risolto con l'overloading del metodo executeAchiveableService della classe DefaultAutonomicAttuator del package selflet.autonomic. La nuova signature prevedere un parametro booleano che esplicita la necessità da parte di una SelfLet di eseguire un servizio localmente oppure di richiederlo ad un altro nodo.

Richiesta asincrona per l'esecuzione remota di un servizio

La versione del SelfLet framework utilizzata supportava solamente richieste sincrone di esecuzione di servizi tra i nodi della rete. Per il corretto funzionamento del sistema presentato si è reso necessario introdurre un meccanismo asincrono affinché una SelfLet che solleva una richiesta al Cloud Manager possa proseguire nella propria esecuzione senza dover attendere l'insieme di operazioni che portano alla creazione di una nuova Cloud SelflLet. Per ridurre al minimo l'impatto sul progetto dovuto all'introduzione di questa nuova funzionalità, è stato introdotta un'implementazione alternativa dell'interfaccia INegotiationManager, per mezzo del design pattern decorator [40].

Meccanismo a due fasi per la richiesta di un servizio remoto

Nel corso dei test funzionali del sistema è stata verificata la modalità di interazione tra i nodi per la richiesta remota di un servizio. In particolare, una SelfLet che conosce un provider per un certo servizio ne richiede l'esecuzione senza verificare che il provider scelto non abbia modificato il proprio stato. Per esempio, il nodo cui si inoltra la richiesta potrebbe essere caduto o potrebbe aver modificato la modalità di erogazione del servizio. In questo caso la SelfLet che effettua la richiesta si mette in attesa dell'esecuzione del servizio, prima di sollevare un errore dovuta alla mancata esecuzione. Il tempo di attesa può essere molto lungo, in quanto la SelfLet richiedente deve lasciare il tempo al provider eventualmente attivo, di completare l'esecuzione. Questo problema impatta fortemente sulle prestazioni della rete nel caso in cui sia frequente l'evento di rimozione di nodi da parte del Cloud Manager. Il meccanismo di richiesta remota è stato quindi modificato con l'introduzione una comunicazione a due fasi nel quale l'effettiva richiesta viene anticipata dall'interrogazione del provider. Nel caso questo non sia più disponibile per

l'erogazione di servizio, l'insieme dei fornitori del servizio viene aggiornata, e viene cercato un nuovo fornitore.

5.12 Conclusioni

In questo capitolo è stato presentato in dettaglio il progetto e lo sviluppo di estensione delle proprietà autonomiche del SelfLet framework per la gestione dei nodi in esecuzione sulla cloud. E' stato introdotto un modello di deploy su cloud e sono stati presentati gli strumenti di interazione con il servizio di cloud computing. In seguito è stato dettagliato il Cloud Manager, componente finalizzato alla gestione centralizzata del ciclo di vita delle Cloud SelfLets; è stato mostrato il modello di interazione tra i nodi della rete, l'estensione dell'architettura del framework e le politiche di gestione dell'insieme dei nodi della rete in esecuzione sulla cloud. Nella fase di implementazione sono state effettuate alcune scelte nel rispetto dei requisiti individuati nella sezione 4.2 e si è reso necessario apportare alcune modifiche ed estensioni al SelfLet framework. E' importante notare che l'insieme delle politiche presentate per la gestione dei nodi della rete sono solo una tra le strategie che è possibile adottare nell'attuazione dei passi di espansione e contrazione. Una traccia di altre possibili strategie individuate nel corso dello studio e dello sviluppo del sistema, sarà presentata nel Capitolo 7.

Capitolo 6

Valutazione (caso di studio: analisi e risultati)

6.1 Introduzione

In questo lavoro è stato realizzato un componente, il Cloud Manager, capace di effettuare le operazioni per la creazione o la terminazione su cloud dei nodi appartenenti a una rete di SelfLets. Sono state introdotte due nuove azioni di ottimizzazione che una SelfLet può attuare per rispondere alle variazioni del proprio livello di utilizzo ed è stato sviluppato un modello di interazione per la gestione, da parte del Cloud Manager, delle richieste di espansione e contrazione sollevate dalle altre SelfLets. In questo modo è possibile, per un sistema composto da nodi che espongono un insieme di servizi, aumentare il numero di elementi che offrono questi servizi al fine di migliorare le prestazioni (per esempio il tempo di attesa per l'esecuzione di un certo servizio) o garantire il livello di qualità del servizio SLA. In questo capitolo l'architettura realizzata verrà valutata attraverso lo sviluppo di un caso di studio.

6.2 Obiettivi

L'obiettivo del caso di studio è quello di validare la funzionalità autonomica di auto-approvigionamento sviluppata e di valutarne l'impatto in termini di prestazioni. Per individuare una metodologia da adottare nella realizzazione dello studio si è fatto riferimento a un precedente lavoro [36] in cui è stato valu-

tato il contributo autonomico apportato al SelfLet framework dall'introduzione della funzionalità di riconoscimento di eventi frequenti, come la richiesta di un servizio. L'obiettivo preposto in questa fase ha reso necessario la creazione di un caso di studio che abiliti la necessità per un sistema di reagire alla variazione del carico di richieste attuando le politiche di ottimizzazione su cloud che, come spiegato nel precedente capitolo, comportano l'invio al Cloud Manager di una opportuna richiesta di espansione o contrazione. Al fine di validare l'approccio proposto, è necessario raccogliere i dati sperimentali relativi alle caratteristiche che concernono la funzionalità che è stata sviluppata. Come sarà spiegato nelle sezioni successive, queste misure includono il tempo medio di attesa dei nodi che richiedono un certo servizio, il numero di Cloud SelfLets che vengono generate e il tempo di vita medio dei nodi avviati su cloud. Un ulteriore requisito è la significatività del caso di studio da realizzare; non essendo infatti possibile coprire tutto il possibile spazio degli esperimenti, è necessario individuare un caso rilevante a cui altri casi di studio possano fare riferimento. In questo modo sarà in futuro possibile proseguire nella valutazione dell'impatto di questa nuova funzionalità attraverso la realizzazione di altri casi.

6.3 Sviluppo del caso di studio

6.3.1 Metodologia adottata

Al fine di validare la funzionalità implementata, gli esperimenti sono stati organizzati in due fasi: la valutazione delle prestazioni del Cloud Manager e l'effettiva realizzazione del caso di studio. Nella prima fase sono state implementate delle semplici SelfLets il cui default behavior consiste nella preparazione e nell'invio di richieste di espansione, di contrazione e di monitoring (l'introduzione di quest'ultima tipologia di richiesta, non prevista in fase di progettazione, è stata poi inserita come spiegato nella sezione 5.10.1). E' stato così possibile valutare il tempo medio di esecuzione delle operazioni di deploy e di monitoring da parte del Cloud Manager. I dati raccolti sono stati in seguito utilizzati per la calibrazione del caso di studio. Nella seconda fase è stato realizzato il caso di studio adeguato a creare la necessità da parte di un insieme di nodi di sollevare richieste di espansione e contrazione. Più precisamente consideriamo due insiemi di SelfLets: S1 ed S2. I nodi appartenenti all'insieme S1 richiedono

l'esecuzione remota di un servizio, in modalità sincrona, CanDo, ai nodi dell'insieme S2. L'aumentare del numero di esemplari appartenenti all'insieme S1, comporta un aumento del numero di richieste ai nodi del secondo insieme, ed un loro conseguente aumento del carico di lavoro. Si rende così necessaria, da parte dei nodi appartenenti all'insieme S2, l'attuazione delle politiche di espansione su cloud. Si vuole testare il caso di studio con un numero crescente di SelfLet client e, per ogni test, è necessario raccogliere i dati dopo un certo intervallo di tempo, in condizioni di regime dell'evoluzione del sistema. Inoltre, per aumentarne la significatività, i casi di test devono essere calibrati impostando il tempo medio di risposta per il servizio richiesto sulla base dei risultati ottenuti nella prima fase dell'esperimento. In ogni test, questo tempo di risposta è omogeneo ad ogni SelfLet che eroga il servizio, e appartiene a tre differenti categorie, come spiegato nelle sezioni successive. L'esecuzione di ogni test prevede la preventiva accensione di un broker REDS in modo tale che ogni SelfLet possa connettersi ad esso per comunicare con le altre. La procedura di avvio di ogni SelfLet è invece stata automatizzata con il tool di building ANT. Il primo nodo ad essere avviato è il Cloud Manager, che deve essere già attivo e pronto a eseguire le richieste sollevate dalle altre SelfLets della rete. Durante l'esperimento, i dati relativi al tempo medio di attesa dei nodi client per l'esecuzione del servizio richiesto possono essere raccolti attivando la categoria di notifiche relative al NegotiationManager utilizzando il framework Log4J [14], già integrato nel SelfLet framework. Per il recupero dei dettagli relativi all'esecuzione delle Cloud SelfLets attivate dal Cloud Manager è possibile fare uso dello strumento di log remoto Cloud Logger, sviluppato nel corso di questo lavoro e descritto nella sezione 5.5.

6.3.2 Fase di valutazione del Cloud Manager

Il contributo in termini di prestazioni apportato a un sistema dall'introduzione della nuova funzionalità sviluppata, è direttamente dipendente dal tempo di esecuzione dei servizi di deploy e di gestione da parte del Cloud Manager. La documentazione fornita da Amazon relativa al livello di qualità SLA per il servizio di cloud computing EC2 [5] fa riferimento a misure di disponibilità. Al contrario non c'è garanzia sul tempo di attivazione delle macchine virtuali, che dipende dalla tipologia di istanza che si richiede e dall'immagine usata

per l'attivazione della VM. Inoltre le operazioni di deploy attuate dal Cloud Manager non si esauriscono nella sola attivazione o nel rilascio delle macchine virtuali, ma sono costituite da un insieme di operazioni più elementari. Per questi motivi, al fine di rendere significativa l'analisi dei dati raccolti attraverso il successivo caso di studio, è stata prevista una prima fase di valutazione delle prestazioni del Cloud Manager. In questa fase sono state sviluppate tre SelfLet, i cui default behaviors sono mostrati in figura 6.1.

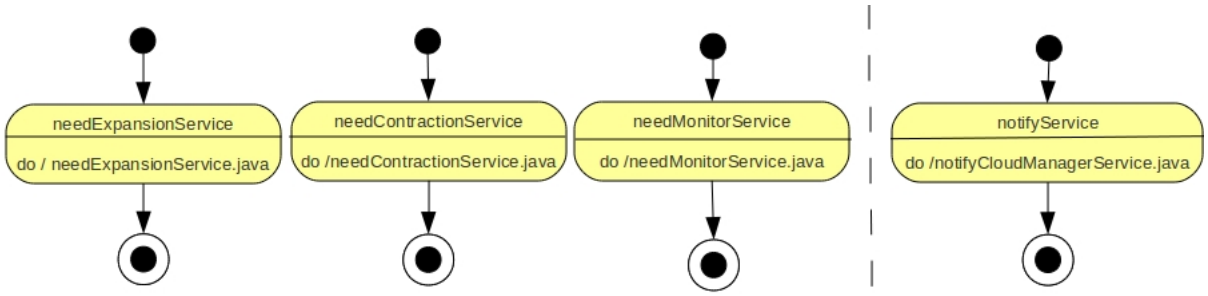

Figura 6.1: Behaviors SelfLets di verifica delle prestazioni del Cloud Manager

La prima SelfLet ha come default behavior un servizio che si occupa di preparare una richiesta di espansione. Il servizio imposta i parametri associati alla richiesta e invoca il servizio di notifica al Cloud Manager. In questo modo una richiesta di espansione viene sollevata all'avvio della SelfLet senza la presenza di una politica per la gestione della relativa azione di ottimizzazione. Analogamente, le altre SelfLets permettono di sollevare richieste di contrazione e di monitoring. Nel caso di una richiesta di contrazione è necessario impostare, tra le proprietà contenute nel file di descrizione XML, un parametro contenente l'id dell'istanza che si vuole terminare. Il servizio di notifica invoca L'esecuzione remota del servizio selfletRegisterService, implementato dal Cloud Manager e offerto in modalità CanDo. che memorizza i parametri associati a una richiesta e da inizio alle operazioni di gestione da parte del manager. Per effettuare le misurazioni, e contrariamente a quanto spiegato nella sezione 5.11, l'esecuzione del servizio di registrazione viene invocato in modalità sincrona. In questo modo è possibile verificare il tempo di esecuzione per un certo servizio guardando il log del tempo di attesa generato dalla SelLet che ha sollevato la richiesta.

Le misurazioni, che sono state ripetute per diversi giorni e in orari diversi della giornata, hanno dato i seguenti risultati, espressi in secondi:

- cloudExpansionService: 160 sec
- cloudContractionService: 37.8 sec
- cloudChangeService (avvio Cloud SelfLet): 46.8 sec
- cloudChangeService (rimozione Cloud SelfLet): 19.8 sec
- cloudMonitorService: 6 sec

I dati riportati sono valori medi e sono stati ottenuti impostando il Cloud Manager per la richiesta di istanze di tipo small, descritto nella sezione 3.2.4. L'immagine utilizzata per l'avvio delle macchine virtuali monta il sistema operativo Ubuntu 10.10, l'id dell'immagine utilizzata è: ami-0f43af66.

6.3.3 Fase di validazione del sistema

Realizzazione del caso di studio

La seconda fase dell'esperimento è iniziata con la realizzazione del caso di studio. Lo scenario sviluppato, in accordo con la traccia descritta nella sezione 6.3.1, è mostrato in figura 6.2

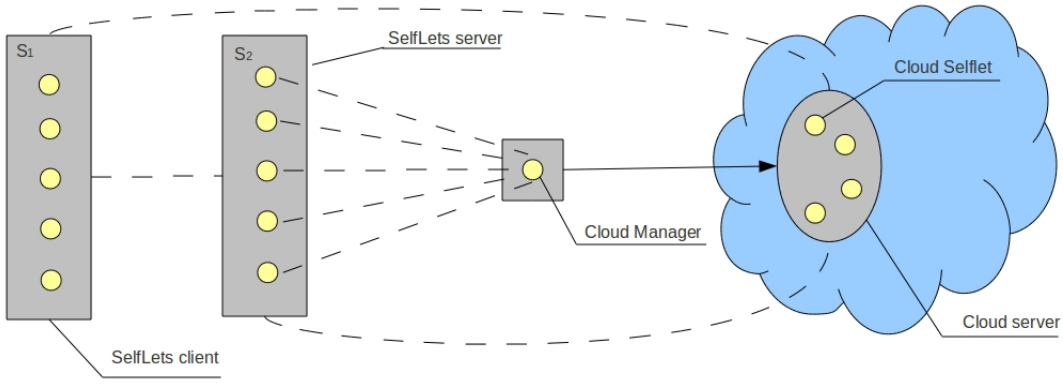

Figura 6.2: Scenario previsto dal caso di studio

Lo scenario prevede la presenza di un insieme di SelfLets S1, in cui ogni nodo implementa un client che richiede periodicamente il servizio di download di un video. Le SelfLets che appartengono all'insieme S2 implementano una batteria di server che forniscono il servizio richiesto dai client. Nel caso presentato, ogni nodo client conosce tutti i nodi server; le SelfLets che implementano i server si conoscono tra di loro. In questo modo, se un server non conosce

un video che gli è stato richiesto, può invocare un servizio di ricerca remota su un altro server, come descritto nel seguito. Ogni server conosce il Cloud Manager ed ha attiva la politica di ottimizzazione su cloud così come descritta nella sezione 5.8. le SelfLets il cui livello di utilizzo supera il valore di soglia sollevano una richiesta di espansione al Cloud Manager. Il manager riceve le richieste e genera le Cloud SelfLets che vanno così ad aggiungersi alla batteria di server nell'esecuzione dei servizi richiesti. In questo esperimento i nodi su cloud generati non sollevano a loro volta richieste di espansione. Generano invece richieste di contrazione nel caso in cui il proprio livello di utilizzo scenda al di sotto del valore di soglia minimo.

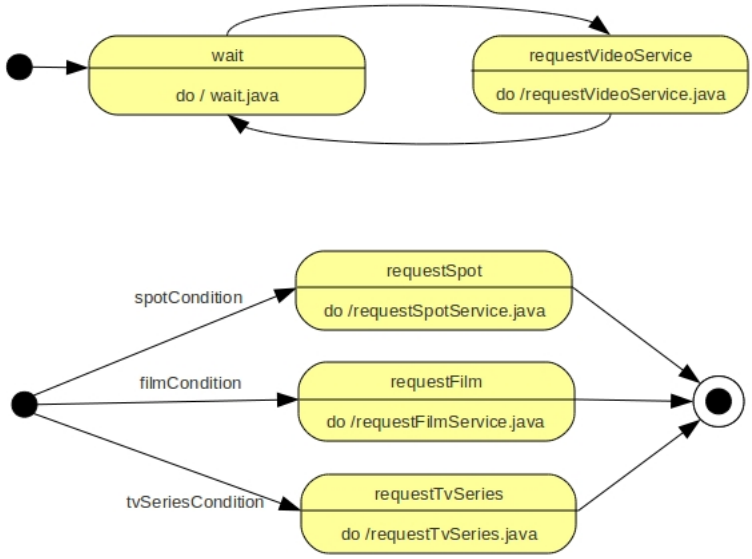

Figura 6.3: Behavior diagrams delle SelfLets client

I diagrammi dei behaviors delle SelfLet che implementano i client sono mostrati in figura 6.3. Ogni client esegue un ciclo in cui, dopo una fase di attesa, effettua la richiesta di un video. Il tipo del video cercato può appartenere a una delle tre categorie: spot, film, serieTv, che corrispondono a un tempo basso, medio o alto di esecuzione del servizio da parte dei server. Il tempo di attesa della fase di wait e la calibrazione del tempo medio di esecuzione dei servizi per le diverse categorie sono parametri di progetto definiti nella successiva sotto-sezione. Perché i test siano significativi rispetto agli obiettivi posti in questo lavoro di valutazione, gli esperimenti devono essere effettuati su insiemi di richieste omogenee rispetto alla categoria. Ciò nonostante sarà

possibile in futuro sviluppare differenti casi di studio modificando le condizioni che determinano la richiesta delle tipologie di video.

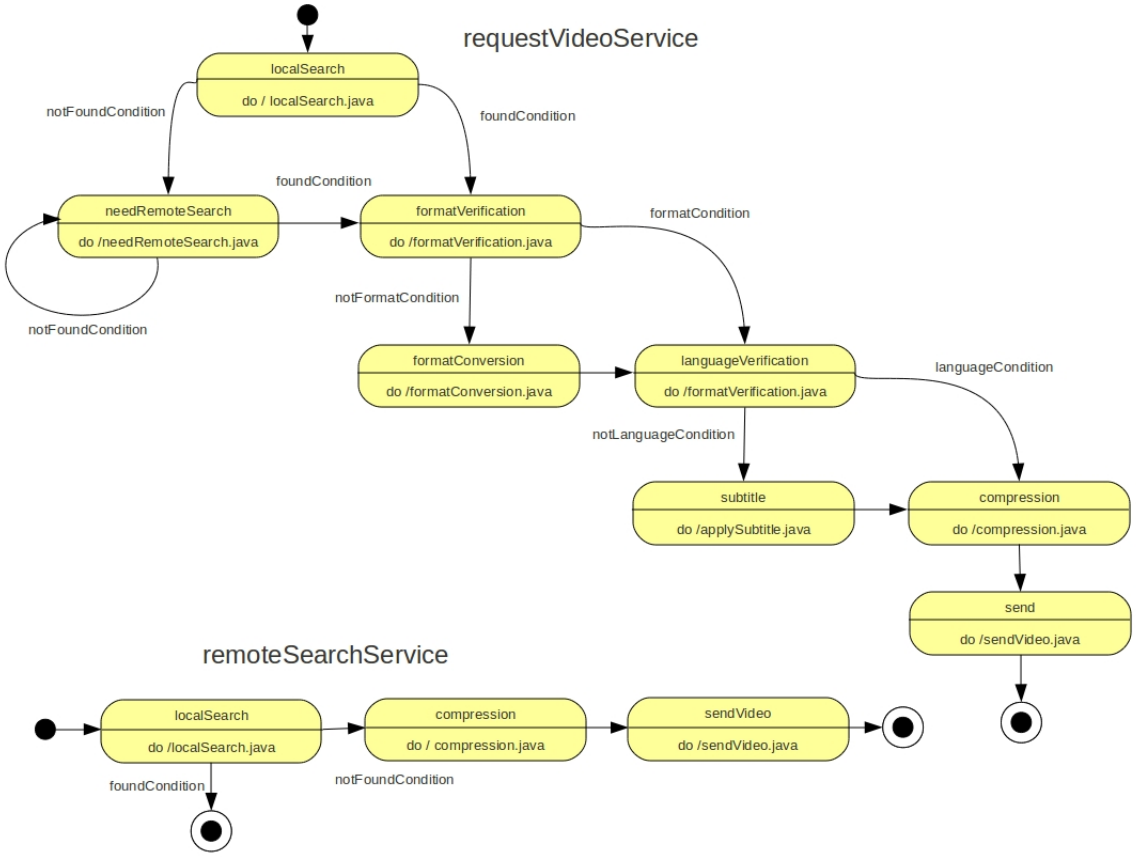

Figura 6.4: Behavior diagrams delle SelfLets server

In figura 6.4 sono mostrati i diagrammi dei behaviors implementati dai server. Il servizio searchVideoService viene esposto ai client in modalità Can-Do e prevede tre parametri: il titolo del video cercato, il formato e la lingua desiderati. Nei test sviluppati, quando il servizio viene richiesto, il parametro titolo viene impostato dal client con il valore associato alla tipologia del video richiesto. Gli altri parametri vengono invece generati casualmente e comportano piccole variazioni nel tempo di esecuzione dei servizi. Inizialmente, una SelfLet che serve una richiesta verifica la presenza locale del video cercato. Se il video viene trovato, il processo continua con la fase di verifica del formato; se il video non viene trovato viene sollevata una richiesta di ricerca remota che sarà servita da un'altra SelfLet appartenente alla batteria di server. Non essendo una situazione reale, non viene considerato il caso in cui nessuno tra i nodi presenti conosca il video associato alla richiesta, mentre la valutazione

della presenza di un video su un certo server viene fatta in probabilità. Per evitare il verificarsi di deadlock, la SelfLet che sta servendo il client continuerà a richiedere una ricerca remota fintanto che la ricerca del video non avrà avuto esito positivo. A questo punto il flusso di esecuzione prosegue come mostrato nello schema. Il sotto-servizio più oneroso è quello in cui viene eseguita la compressione del filmato. In questo caso la tipologia del video richiesto viene verificata e la SelfLet esegue una fase di elaborazione il cui tempo medio di esecuzione è definito in associazione al tipo di richiesta ricevuta.

L'introduzione dell'eventualità di una ricerca remota così come la verifica sul formato e sulla lingua richiesta è dovuta, oltre che alla necessità di creare un caso di test verosimile, al proposito di ottenere un caso di studio significativo. Una successione di eventi deterministica non consentirebbe infatti di verificare la proprietà di auto-approvvigionamento del sistema e la conseguente risposta alle variazioni del carico. La variabilità introdotta avvicina il caso di studio ad una situazione reale in cui variazioni impreviste nella distribuzione del carico possono comportare successive mosse di ottimizzazione su cloud, evidenziando eventualmente debolezze nella risposta del sistema.

Calibrazione del caso di studio e scelta dei parametri di progetto

Nelle sezioni precedenti si è fatto riferimento alla necessità di calibrazione degli esperimenti e ad alcune misure di valutazione. Viene ora fornita una definizione di questi parametri.

parametri di progetto del sistema

- n, numero dei client che effettuano le richieste
- m, numero dei nodi che offrono il servizio di ricerca dei film
- t_{sleep}, tempo di attesa tra due richieste successive da parte di un client
- $t_{execution}$, tempo medio di esecuzione di un servizio da parte di un server
- d, durata di un test

Si vogliono ottenere i risultati impostando i parametri come segue: $n \in \{1, 10\}$, $m = 3$, $t_{sleep} = 10sec$, $d = 30min$. In questo modo è possibile

confrontare la risposta di un sistema composto da più server (permettendo così la funzionalità prevista di richiesta remota di un video tra server) in differenti condizioni di carico di richieste. E' necessario ripetere gli esperimenti per diversi valori di $t_{execution}$ in modo da verificare l'impatto dell'azione di ottimizzazione nel caso in cui i servizi da realizzare prevedano un tempo di esecuzione mediamente inferiore, uguale o maggiore al servizio di espansione del Cloud Manager. E' stato scelto di ripetere gli esperimenti per tre categorie di richieste: spot, film e serieTV. Definendo $t_{expansion}$ il valor medio del tempo di esecuzione dell'operazione di espansione, $t_{execution}$ sarà fissato rispettivamente: $t_{execution} = t_{expansion}/2$; $t_{execution} = t_{expansion}$; $t_{execution} = t_{expansion} * 2$.

parametri di progetto del Cloud Manager

- vm_{max}, numero massimo di macchine virtuali che è possibile richiedere
- w, finestra temporale in cui attivare l'azione changeSelfLetService

Il parametro di configurazione del Cloud Manager relativo al numero massimo di macchine virtuali ammissibile viene impostato, nel caso dell'esperimento, a un valore elevato $vm_{max} = 100$ in modo da rilevare un eventuale malfunzionamento del modello di espansione, causa di un'esplosione nel numero di nodi attivati. Il secondo parametro è così impostato: $w = 15min$, e permette di osservare sul sistema a regime sia l'effetto delle azioni di sostituzione di configurazione delle SelfLets su una stessa macchina virtuale, sia il caso in cui anche la VM viene rilasciata.

misure da rilevare

- t_{wait}, tempo medio di attesa da parte di un client
- vm, numero di Cloud SelfLets generate dal Cloud Manager
- t_{life}, tempo di vita medio di una Cloud SelfLet

Le misure da rilevare sono state scelte in base a criteri di significatività. Il tempo di attesa dei client è indice dell'impatto in termini di qualità del servizio apportato dall'introduzione della politica di ottimizzazione su cloud. Il numero

e la durata del ciclo di vita delle Cloud SelfLet esprimono invece l'efficacia del modello di gestione dei nodi in esecuzione nella cloud. Per permettere una comparazione, i casi di studio devono inoltre essere valutati nel caso in cui i server non abbiano attiva la politica di ottimizzazione su cloud.

Risultati attesi

Il comportamento atteso dal sistema è il seguente: nel caso in cui il carico di richieste ai nodi server non è elevato, tipicamente quando il numero di client è inferiore al numero di server, il sistema non ha la necessità di attuare azioni di ottimizzazione su cloud. All'aumentare del numero di client, il sistema può eventualmente attivare sporadiche Cloud SelfLets per far fronte ai casi in cui le richieste vengono inoltrate ad un unico server aumentandone il carico di lavoro. Quando il numero di client è molto superiore ai server disponibili, il comportamento atteso è l'attivazione di un numero di Cloud SelfLets sufficiente a creare una condizione di bilanciamento del carico complessivo. Per questo motivo si prevede l'accensione di un numero limitato di nodi su cloud, in quanto l'attivazione di una Cloud SelfLet contribuisce a ridurre il carico sugli altri nodi della rete. Si vuole inoltre verificare che l'introduzione del servizio di ottimizzazione interno al Cloud Manager, reso possibile dal Cloud Pool (descritto nella sezione 5.10.1), e che consiste nella sostituzione delle Cloud SelfLets su macchine virtuali che non vengono rilasciate, permetta un'adeguata risposta del sistema anche nel caso in cui i servizi richiesti siano caratterizzati da un tempo medio di esecuzione inferiore al tempo necessario per eseguire una mossa di espansione della rete.

Scelte progettuali necessarie alla realizzazione

Durante la realizzazione del caso di studio, è emersa la necessità di operare alcune scelte inizialmente non previste in fase di progettazione:

estensione funzionalità di richiesta di un servizio remoto: La realizzazione di questo caso di studio ha fatto emergere la necessità di introdurre una nuova modalità di richiesta di un servizio, inizialmente non prevista dal SelfLet framework, e dovuta alla necessità di richiedere l'esecuzione remota di un servizio disponibile localmente; è questo il caso in cui un server, che implementa il servizio di ricerca remota,

deve richiederne l'esecuzione ad un altro server dopo aver verificato la non disponibilità locale di un video richiesto. L'introduzione di questa funzionalità è stata discussa nella sezione 5.11.

politica di contrazione dei nodi: L'operazione di contrazione di un nodo che sta eseguendo un servizio richiesto da un'altra SelfLet, può comportarne l'attesa per un lungo periodo di tempo, fino allo scadere del timeout stabilito per l'esecuzione di un servizio remoto. Per questo motivo la politica di contrazione delle Cloud SelfLets prevede che, quando un nodo verifica la condizione di contrazione, questo imposti a None la modalità di esecuzione dei propri servizi in modo che non vengano più offerti ai client che ne fanno richiesta; a questo punto il server finisce di servire le richieste presenti nella coda e successivamente invia una notifica di contrazione al Cloud Manager.

gestione delle richieste da parte del Cloud Manager: la versione del SelfLet framework utilizzata non implementa la possibilità per una SelfLet di eseguire più servizi in parallelo. Un'elevata frequenza delle richieste di espansione ricevute dal Cloud Manager insieme all'attivazione delle macchine virtuali, operazione che può richiedere diverso tempo rallentando il processo di deploy, può comportare alcuni inconvenienti. In particolare il Cloud Manager può trovarsi a servire richieste ricevute in precedenza, che non rappresentano più le correnti necessità dei nodi che le hanno sollevate. Per questo motivo è stata introdotta una politica di selezione delle richieste da servire, per la quale il manager scarta le richieste di espansione e di monitoring ricevute durante l'esecuzione del servizio di espansione.

Risultati sperimentali

Il comportamento atteso dal sistema non è stato verificato a causa di alcuni limiti dovuti all'attuale implementazione del SelfLet framework e al verificarsi di un fenomeno inatteso di instabilità in fase di avvio dei nodi su cloud. La versione del SelfLet framework utilizzata non implementa l'azione di ottimizzazione di redirect delle richieste di esecuzione dei servizi verso altri nodi. Inoltre, dai test effettuati, è emerso che il meccanismo implementato per la

ricerca dei provider da parte di una SelfLet che necessita di un servizio, prevede tale ricerca alla sola prima richiesta per il servizio. Le successive richieste vengono inoltrate ai provider già conosciuti. Per questi motivi, il comportamento verificato è stato l'attivazione delle Cloud SelfLets di supporto ai server da parte del Cloud Manager, senza che i nodi così generati possano essere individuati dai client come possibili provider per il servizio richiesto. Un altro effetto imprevisto ha minato la corretta realizzazione degli esperimenti. La politica di contrazione delle Cloud SelfLets prevede una verifica del livello di utilizzo e, se questo è inferiore al minimo livello di utilizzo previsto, il nodo su cloud richiede al Cloud Manager l'esecuzione delle operazioni di contrazione. E' stato così verificato che: dal momento in cui una Cloud SelfLet inizia la propria esecuzione, se non riceve alcuna richiesta per l'esecuzione di un servizio, la prima iterazione della politica di ottimizzazione su cloud comporta l'inoltro di una richiesta di contrazione al Cloud Manager. Si rivela quindi necessario introdurre un meccanismo di stabilizzazione per le Cloud SelfLets appena generate.

6.4 Conclusioni

In questo capitolo è stato descritto lo sviluppo del caso di studio finalizzato a testare la validità dell'architettura realizzata con lo scopo di estendere il SelfLet framework con la proprietà di auto-approvvigionamento. I vantaggi che derivano da questa proprietà risiedono nella possibilità di un sistema di adattare il proprio pool di risorse in modo da far fronte alle variazioni del carico di lavoro dei diversi nodi. La fase di validazione del sistema ha però fatto emergere alcuni problemi che attualmente impediscono all'architettura di apportare il contributo previsto. Nonostante la fase di valutazione del Cloud Manager abbia permesso di verificare la corretta gestione dei nodi in esecuzione su cloud, e di analizzare il tempo necessario all'esecuzione delle operazioni di deploy, la successiva esecuzione dei casi di test non ha portato ai risultati auspicati. Ciò nonostante, i problemi riscontrati e relativi a integrazione e stabilità sono risolvibili. Una futura implementazione del SelfLet framework che preveda l'implementazione della funzionalità di redirect delle richieste ricevute da un nodo verso i propri vicini permetterà la corretta collaborazione tra i nodi della rete. Il problema legato alla politica di contrazione delle Cloud

SelfLets potrà invece essere risolto attraverso l'introduzione di meccanismi di contrazione più raffinati.

Capitolo 7

Conclusioni e Sviluppi Futuri

7.1 Conclusioni

Questo lavoro di tesi è stato finalizzato al progetto e allo sviluppo di una soluzione che estende le proprietà del framework autonomico SelfLet con la funzionalità di gestione dell'esecuzione di una rete di nodi in ambiente cloud.

Inizialmente sono stati studiati i modelli per lo sviluppo dei sistemi autonomici e per la gestione dinamica delle risorse reso possibile dal recente sviluppo del cloud computing. Sono state mostrate le relazioni tra questi due paradigmi e individuate le proprietà emergenti da una loro possibile integrazione. La fase di studio è proseguita considerando due particolari istanze di questi modelli: il framework autonomico SelfLet e il servizio di cloud computing proposto da Amazon. Il SelfLet framework permette di sviluppare sistemi distribuiti costituiti da una rete in cui ogni nodo è un elemento autonomico. Amazon Web Services sono un insieme di servizi che forniscono funzionalità di allocazione dinamica di macchine virtuali secondo il modello Infrastructure as a Service (IaaS). In seguito sono state analizzate le possibilità di estensione del SelfLet framework con il servizio EC2 ed è stato individuato un insieme di requisiti. La seconda parte del lavoro è stata la progettazione e la realizzazione di un modello di gestione di una rete di SelfLets in ambiente cloud. A una panoramica sugli aspetti legati ai processi per lo sviluppo di sistemi basati su queste tecnologie è seguita la fase di design in cui sono stati definite le tipologie di SelfLets che implementano la soluzione proposta; è stato successivamente dettagliato l'insieme dei servizi necessari alla realizzazione del sistema ed è sono

state definite le operazioni di espansione e di contrazione di una rete all'interno della cloud. Il progetto del componente preposto alla gestione delle risorse interne alla cloud ha seguito il modello gerarchico previsto per la costruzione di sistemi autonomici. Il risultato di questo processo è il Cloud Manager che, in accordo al modello, è implementato da una SelfLet e costituisce, insieme al cloud element gestito, un elemento autonomico di più alto livello. E' stato poi definito un modello di comunicazione tra SelfLets per il quale le operazioni eseguite dal Cloud Manager sono il prodotto di un insieme di decisioni locali governato dalle politiche interne ai singoli nodi della rete. Per questo motivo, è stata estesa l'architettura del componente di ottimizzazione del SelfLet framework e sono state introdotte due nuove azioni di ottimizzazione. Il lavoro si è concluso con la preparazione di un caso di studio per la validazione del modello realizzato e per la verifica della risposta del Cloud Manager al carico di richieste sollevato dagli altri nodi della rete.

Il principale contributo di questo lavoro di tesi risiede nell'avere proposto una traccia di soluzione per il processo di integrazione tra sistemi autonomici e cloud computing, mostrata attraverso l'implementazione di un'estensione del framework autonomico SelfLets che ne aggiunge la proprietà di auto-approvvigionamento. Un sistema basato su SelfLets può ora adattare la sua topologia espandendo o contraendo la rete dei nodi che lo costituiscono in modo da far fronte alle variazioni di carico dei suoi componenti. Ciò nonostante, alcuni problemi riscontrati durante la fase di validazione del sistema hanno impedito la completa realizzazione del modello di funzionamento sviluppato. Alcune tracce di risoluzione di questi problemi sono date nel seguito di questo capitolo.

7.2 Sviluppi futuri

Il lavoro presentato rappresenta un punto di partenza per future estensioni che introducano nuovi aspetti di gestione autonomica del cloud element e che amplino le strategie di utilizzo delle risorse offerte dai servizi di cloud computing. A questi scopi l'architettura realizzata è stata progettata per essere facilmente estendibile e viene ora presentata una traccia dei possibili miglioramenti.

7.2.1 Estensioni del SelfLet framework

Cloud Manager Clone

Durante la fase di progetto è stata definita una tipologia di SelfLets, il Cloud Manager Clone, come un'implementazione parziale del manager che ne espone i soli servizi operativi. In questo modo il Cloud Manager può sfruttare il cloud element gestito allocando nuovi cloni e delegando loro le onerose operazioni di creazione di nuovi nodi, riuscendo così a soddisfare un maggiore numero di richieste di espansione.

Gestione autonomica della configurazione del servizio di cloud

La versione attuale del Cloud Manager prevede una fase di configurazione da parte dello sviluppatore per la scelta del tipo di macchina virtuale, dell'immagine di partenza montata su tale macchina e dei parametri relativi ai gruppi di sicurezza e alle zone di esecuzione cui le macchine appartengono. In futuro il Cloud Manager avrà un componente che si occuperà della configurazione automatica del servizio. A questo scopo il framework di interazione CloudTouch offre già i metodi per la gestione di questi aspetti.

Meccanismi di recovery

La fase di monitoring del cloud manager si occupa di rilevare e gestire alcune anomalie. Per esempio viene effettuato il reset della rete nel caso il costo del servizio superi in modo imprevisto il valore di soglia impostato. Il Cloud Manager dovrà essere esteso con l'introduzione di meccanismi per il recovery delle Cloud SelfLets guaste e dei broker caduti. Inoltre i nodi in esecuzione all'interno della cloud dovranno essere in grado di gestire la rete nel caso in cui sia il Cloud Manager a guastarsi.

Estensione Cloud Pool

Attualmente il Cloud Pool è una mappa delle istanze virtuali e delle Cloud SelfLet attive. Il pool può essere esteso per tenere traccia delle richieste ricevute da parte delle SelfLets in modo da attuare diverse politiche nella scelta delle richieste da servire. Si prevede inoltre un Service Pool che tenga traccia dell'insieme dei servizi che sono forniti dalle Cloud SelfLets attive nella cloud.

Implementazione delle funzionalità di base previste dal SelfLet framework

La versione del SelfLet framework utilizzata mancava di alcune funzionalità di base previste. Tra queste la possibilità per una SelfLet di eseguire servizi in parallelo, la completa implementazione delle azioni di Teach dei servizi e un meccanismo di Redirect delle richieste ricevute da una SelfLet, verso altri nodi della rete. L'introduzione di queste funzionalità permetterà rispettivamente di: migliorare le prestazioni del Cloud Manager e la frequenza di esecuzione dei servizi di espansione a lui richiesti; permettere l'introduzione di auto-apprendimento dei servizi da parte di una Cloud SelfLet appena attivata; migliorare la collaborazione tra le Cloud SelfLet create dinamicamente e quelle già esistenti nella rete.

Creazione di Cloud SelfLet ottenute per composizione

Attualmente le operazioni di espansione prevedono la creazione di cloni delle SelfLet che ne fanno richiesta. Gli elementi che compongono la Cloud SelfLet che si vuole generare sono memorizzati in un repository locale al nodo che svolge la funzione di Cloud Manager. In futuro il Cloud Manager dovrà essere in grado di selezionare un insieme di richiese e generare una Cloud SelfLet come composizione dei parametri di configurazione associati a queste richieste. Le singole SelfLet dovranno inoltre essere in grado di inviare al Manager solo porzioni di regole, di servizi e di condizioni che permettono la creazione di nodi che eseguano un sotto-insieme delle proprie funzionalità.

Cloud Manager distribuito

La soluzione presentata prevede il controllo centralizzato dei nodi della cloud da parte del Cloud Manager. I meccanismi di gestione sono stati implementati all'interno di servizi in modo che possano essere attuati da qualsiasi SelfLet. In futuro sarà possibile la gestione distribuita della Cloud da parte di un insieme di nodi. E' previsto uno scenario in cui ogni SelfLet possa essere Cloud Manager di sé stessa e in cui i servizi di gestione del cloud elelment possano essere insegnati alle SelfLets che ne fanno richiesta.

7.2.2 Altre strategie di utilizzo delle cloud selflet

Le politiche di ottimizzazione su cloud presentate in questo lavoro hanno come obiettivo il bilanciamento del carico di utilizzo dei nodi della rete. E' possibile adottare differenti strategie nell'utilizzo del cloud computing come risorsa di supporto a una rete, dipendentemente dagli obiettivi dell'applicazione. La figura 7.1 mostra alcune possibili alternative.

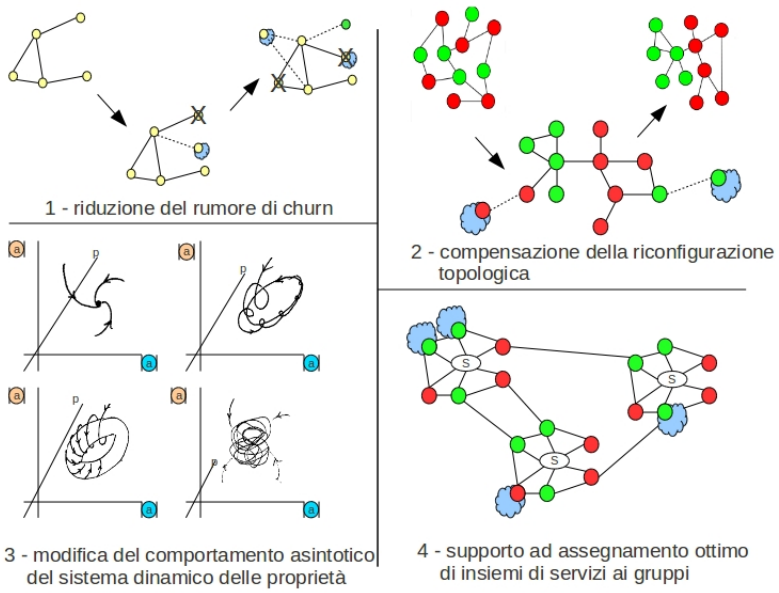

Figura 7.1: Esempi di strategie di cloud computing

Il riquadro (1) mostra un esempio di utilizzo della cloud per la riduzione del rumore topologico di una rete dovuto alla comparsa di nuovi nodi e alla caduta di altri. E' possibile utilizzare lo stesso meccanismo per mantenere attive funzionalità offerte dai nodi che si guastano. Nell'esempio (2) viene mostrato un esempio di supporto offerto dai nodi su cloud alle operazioni di riconfigurazione di una rete. In (3) si vede un utilizzo alternativo delle possibilità di allocazione dinamica dei nodi. Alcune proprietà di un sistema software potrebbero essere modellizzate secondo la teoria dei sistemi dinamici. L'evoluzione a regime del sistema potrebbe essere dipendente dal rapporto tra le quantità di diverse tipologie di nodi. Modulando il numero di esemplari per una certa tipologia è possibile far esibire alle proprietà del sistema un differente andamento asintotico. L'ultimo caso (4) mostra un esempio di supporto all'assegnamento ottimo di un insieme di jobs a insiemi di gruppi di lavoro composti da nodi eterogenei.

Un algoritmo di schedulazione potrebbe individuare dei gruppi incompleti a causa della topologia della rete. La possibilità di creazione di nuovi nodi rende possibile il completamento dei gruppi per l'esecuzione del lavoro.

Bibliografia

[1] Nist definition of cloud computing v15, csrc.nist.gov/groups/ SNS/cloud-computing/cloud-def-v15.doc.

[2] Argouml, http://argouml.tigris.org/.

[3] Amazon web services, http://aws.amazon.com/.

[4] Amazon elastic compute cloud, http://aws.amazon.com/ec2/.

[5] Ec2 sla, http://aws.amazon.com/ec2-sla/.

[6] Amazon simple storage service, http://aws.amazon.com/s3/.

[7] Aws sdk for java, http://aws.amazon.com/sdkforjava/.

[8] Amazon simpledb, http://aws.amazon.com/simpledb/.

[9] Amazon simple queue service, http://aws.amazon.com/sqs/.

[10] Google app engine, http://code.google.com/appengine.

[11] Ec2 user guide, http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/.

[12] Java swing, http://download.oracle.com/javase/tutorial/uiswing/.

[13] Mapreduce, http://labs.google.com/papers/mapreduce.html.

[14] Log4j, http://logging.apache.org/log4j/1.2/.

[15] Cascadas, http://www.cascadas-project.org/summary.php.

[16] Cloud hosting, cloud computing and hybrid infrastructure from gogrid, http://www.gogrid.com.

[17] Drools, http://www.jboss.org/drools/.

[18] Junit, http://www.junit.org/.

[19] Jaxb, http://www.oracle.com/technetwork/articles/javase/index-140168.html.

[20] Osgi, http://www.osgi.org/About/Technology.

[21] Dedicated server, managed hosting, web hosting by rackspace hosting, http://www.rackspace.com.

[22] Autonomic computing manifesto, http://www.research.ibm.com-/autonomic/manifesto/autonomic
_computing.pdf, 2001.

[23] Salesforce crm, http://www.salesforce.com/platform.

[24] Reds, http://zeus.ws.dei.polimi.it/reds/.

[25] Lime, lime.sourceforge.net.

[26] Flexiscale cloud comp and hosting, www.flexiscale.com.

[27] Javassist, www.javassist.org.

[28] Kernal based virtual machine, www.linux-kvm.org/page/ MainPage.

[29] Windows azure, www.microsoft.com/azure.

[30] Vmware esx server, www.vmware.com/products/esx.

[31] Xensource inc, xen, www.xensource.com.

[32] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. February 2009.

[33] J. Bell. Understand the autonomic manager concept. *IBM Autonomic Computing Technical Report*, 2004.

[34] S. Bindelli, E. Di Nitto, R. Mirandola, and R. Tedesco. Building autonomic components: The selflets approach. in automated software engineering. *Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference*, September 2008.

[35] P. Bodik. Statistical machine learning makes automatic control practical for internet datacenters. *Proc HotCloud*, 2009.

[36] N. M. Calcavecchia. Prediction models for autonomic systems: conceptual model and implementation of a development environment. Master's thesis, Politecnico di Milano, 2009.

[37] N. M. Calcavecchia, D. Ardagna, and E. Di Nitto. The emergence of load balancing in distributed systems: the selflet approach. In Danilo Ardagna and Li Zhang, editors, *Run-time Models for Self-managing Systems and Applications*, Autonomic Systems, pages 97–124. Springer Basel, 2010. 10.1007/978-3-0346-0433-8_5.

[38] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, August 2001.

[39] D. Devescovi. A conceptual model and an infrastructure for autonomic systems development: Design and implementation. Master's thesis, Politecnico di Milano, 2007.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements Of Reusable Object Oriented Software*. Addison Wesley, 1995.

[41] J. Hamilton. Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services. *Proc of CIDR*, 2009.

[42] IBM. *An architectural blueprint for autonomic computing*, 2005.

[43] E. Kalyvianaki. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. *Proc of international conference on autonomic computing*, 2009.

[44] L. Kleinrock. *Queuing Systems Volume I: Theory*. John Wiley and Sons, 1975.

[45] J. Caceres M. Lindner L. Vaquero, L. Rodero-Merino. A break in the clouds: towards a cloud definition. *ACM SIGCOMM computer communications review*, 2009.

[46] R. Maheswaran and T. Ba ar. Nash equilibrium and decentralized negotiation in auctioning divisble resources. *Group Decision and Negotiation, 12:361395*, 2003.

[47] D. Parkhill. *The challenge of the computer utility*. Addison- Wesley, 1966.

[48] B. Urgaonkar. Dynamic provisioning of multi-tier in- ternet applications. *Proc of ICAC*, 2005.

[49] J. Varia. Architecting for the cloud: Best practices. *Technical report, Amazon*, 2010.

Ringraziamenti

Il primo e più sincero pensiero al momento dei ringraziamenti va senza ombra di dubbio, oltre che naturalmente ai miei genitori, a Sara: per avere creduto in me.

Ringrazio tutte le persone con le quali ho condiviso questi anni avventurosi e, in particolare, i miei amici Gabriele, Stefano e Antonio.

Un ringraziamento speciale va a Luca, che per primo mi ha permesso di vedere la mia strada.

Infine ringrazio la Prof.ssa Di Nitto, i dottorandi Nicolò e Daniel per avermi guidato con pazienza e disponibilità nei passi incerti della mia tesi.

Mando un saluto a tutti i ragazzi del laboratorio di Ing. del Software, ai quali faccio i miei migliori auguri.

Milano, 24 Novembre 2010

Alessandro