Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

Progettazione e Sviluppo di un Sistema Cloud P2P

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:
Chiar.mo Prof.
Moreno MARZOLLA

Presentata da:
Michele TAMBURINI

Sessione I
Anno Accademico 2010-11

Ai miei genitori:
Luciana e Tiziano

Introduzione

Il Cloud Computing costituisce un modello in grado di abilitare l'accesso in rete in maniera condivisa, pratica e on-demand, di diverse risorse computazionali (come reti, server, memoria di massa, applicazioni e servizi), che possono essere rapidamente approvvigionate e rilasciate col minimo costo di gestione ed interazioni tra utenti e provider"[4].
Sebbene il panorama delle Cloud sia ancora estremamente giovane, negli ultimi anni ha acquistato sempre maggiore importanza nell'Information and Communication Technology (ICT). Ormai numerosi sono gli esempi del settore commerciale nel quale possiamo citare tra tutti: Amazon Elastic Computing Cloud [29], IMB Blue Cloud [31], Network.com di Sun Microsystems [43], Azure Services Platform di Microsoft [16], Google App Engine [30] e le soluzioni Dell Cloud [21].
Nel contesto accademico diverse sono le proposte di nuovi modelli ed architetture, alcuni dei quali si sono concretizzati nello sviluppo di sistemi [46], [13], [53],[27].

Questo lavoro ha come scopo la realizzazione di una Cloud privata, per la fornitura di servizi, basata su un'architettura P2P. L'elaborato vuole studiare il caso di un sistema P2P di livello infrastruttura e propone la realizzazione di un prototipo capace di sostenere un insime basilare di API. Verranno utilizzati protocolli di gossip per la costruzione dei servizi fondamentali.
Se da un lato lo scenario P2P costituisce una sfida implementativa, dall'altro offre i vantaggi di scalabilità, perché l'aggiunta di un nuovo nodo nella rete si concretizza semplicemente con l'avvio del pacchetto applicativo, e di

robustezza, in quanto le responsabilità sono suddivise tra tutti i componenti della rete e non vi sono ruoli specifici dettati a priori.

L'esposizione di questa tesi è organizzata come segue. Nel capitolo 1 viene presentata la definizione di Cloud Computing con la descrizione dei modelli che appartengono al paradigma. Vengono quindi esplorati lo scenario del Volunteer Computing e le difficoltà che si riscontrano nel realizzare sistemi Cloud. Nella rassegna dello stato dell'arte del capitolo 2 vengono scelti alcuni esempi dei quali si confrontano le architetture e gli insiemi delle interfaccie API. Nel successivo capitolo 3 si esibiscono alcuni casi di studio e si avanzano gli scenari per l'utilizzo dell'infrastruttura. Il capitolo 4 è dedicato alla progettazione del sistema e prevede la formulazione dell'architettura generale e della costruzione della rete, che fa uso di protocolli di gossip. Vengono infine descritti alcuni dettagli dell'implementazione nel capitolo 5 e presentato un esempio di utilizzo del prototipo sviluppato.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione al paradigma Cloud Computing

Questo capitolo presenta una breve introduzione al *Cloud Computing*, un concetto nuovo sotto alcuni punti di vista, che si pone l'obiettivo di gestire un sistema distribuito per fornire servizi agli utenti. Sono presentate dapprima le definizioni di *Utility Computing* e di *Cloud*, strettamente legate dall'argomento cardine. Segue poi in 1.2 un confronto tra i due sistemi Cloud e Grid che ne sottolinea le differenze. La sezione 1.3 illustra i due modelli noti in letteratura per la classificazione delle Cloud: rispetto ai servizi offerti ed alle modalità di deployment. In 1.4 viene esplorato lo scenario del *Volunteer Computing* che si discosta dal consueto caso fornitore-cliente, per promuovere la collaborazione tra gli utenti. Conclude il capitolo una lista di ostacoli alla realizzazione di un ambiente di Cloud Computing.

1.1 Cloud Computing e Cloud

Il significato di *Cloud Computing* unisce quelli di *Utility Computing* e *Cloud*.

Utility Computing indica un insieme di risorse, di diverso tipo, che vengono raggruppate e gestite da un provider, in modo da offrire potenza computazionale attraverso la rete ad un utilizzatore finale. Viene spesso associata a questo termine l'idea che la potenza di calcolo venga in un futuro erogata allo stesso modo di acqua, corrente elettrica o gas, e l'utente paghi poi solo l'effettivo utilizzo.

Conformemente alla definizione proposta in [46], una **Cloud** corrisponde ad "*[...] un tipo di sistema distribuito che consiste di un insieme di computer virtuali interconnessi che vengono forniti e presentati come un'unica risorsa[...]*".

Secondo il NIST (National Institute of Standard and Technology [4]) "*[...] il* ***Cloud Computing*** *è un modello in grado di abilitare l'accesso in rete in maniera condivisa, pratica e on-demand, di diverse risorse computazionali (come reti, server, memoria di massa, applicazioni e servizi), che possono essere rapidamente approvvigionate e rilasciate col minimo costo di gestione ed interazioni con il provider.* "

1.2 Confronto tra Cloud e Grid

Mentre il termine "Cloud" è apparso solo da qualche anno nella letteratura, l'idea di "grid computing" è nota già dai primi anni del 1990. Possiamo dunque porci la domanda: in cosa differiscono le Cloud rispetto alle Grid, e quali sono invece i punti in comune?

Secondo la definizione estratta da [12] si può definire una Grid come: "*un'infrastruttura hardware e software distribuita geograficamente su larga scala composta da risorse di rete possedute e condivise da diverse organizzazioni amministrative, le quali devono coordinarsi per fornire un supporto compu-*

tazionale che sia trasparente, affidabile e consistente per una vasta gamma di applicazioni.[...]".

Possiamo ora notare che: sia l'una che l'altra sono costituite da un sistema distribuito, nel quale cioè ogni macchina è dotata della propria CPU, RAM, memoria di massa, interfaccia di rete... Si presenta quindi in entrambe la necessità di coordinare un vasto insieme di calcolatori.

Una prima differenza emerge riflettendo sugli individui che se ne servono: mentre le Grid nascono dall'esigenza di compiere complessi calcoli scientifici ed essere quindi sfruttate da laboratori di ricerca o Università, le Cloud non hanno tutt'oggi un bacino di utenza ben definito. La categoria degli utilizzatori di questi nuovi sistemi, si estentende infatti dal singolo utente, alla media o grande impresa (presenteremo nel capitolo 3 una più dettagliata osservazione).

Possiamo in maniera più significativa notare che, rispetto ai sistemi distribuiti conosciuti, le Cloud mostrano tre caratteristiche che le differenziano dai precedenti e sono in grado di catalogarle in modo ben definito: virtualizzazione, elasticità e "pay-as-you-go".

1. **Virtualizzazione**: si intende l'aggiunta di un componente intermedio in grado di virtualizzare le risorse fisiche quali cpu, memoria e rete e software come istanze di strumenti applicativi, al fine di esporre un'interfaccia omogenea per le applicazioni che risiedono ai livelli superiori. Ad esempio il caso d'uso dell'utilizzatore di una Cloud infrastrutturale prevede che questi richieda al sistema l'allocazione di *istanze*, ovvero sistemi operativi in esecuzione all'interno di macchine virtuali. Ciauscun elaboratore collegato alla Cloud dispone infatti di un hypervisor[1].

2. **Elasticità**: indica la possiblità dell'utente di variare in qualunque momento le risorse occupate. Permette dunque di aumentare le istanze

[1]**Hypervisor**: è il livello software che si interpone tra il sistema operativo installato nella macchina fisica (host), e quello instanziato nell'ambiente virtuale, definito in genere come guest. L'hypervisor offre ai livelli superiori della macchina virtuale l'interfaccia per dialogare con il sistema operativo ospitante.

o la memoria di massa a fronte di un maggiore carico di lavoro, o di rilasciarle nel caso opposto.

3. **Pay-as-you-go**: all'utente verrà addebitato il solo utilizzo effettivo delle risorse che, dipendentemente dai servizi offerti dal provider, viene valutato come: numero di istanze allocate, memoria di massa occupata, GB di dati trasferiti o indirizzi pubblici posseduti. Questa caratteristica introduce il concetto di *accounting* (dall'inglese: contabilità), che ha come obiettivo il monitoraggio delle risorse occupate dall'utilizzatore.

La disponibilità di un ambiente virtuale permette di creare un servizio omogeneo e adattabile alle esigenze dell'utilizzatore, partendo da un insieme di macchine eterogenee. L'utente interessato ad avviare un'istanza potrà infatti deciderne il sistema operativo e le prestazioni (CPU, RAM e latenze di I/O), scegliendo in base al lavoro che dovrà svolgere. Come esempio si prenda in considerazione l'elenco piuttosto vasto [2] offerto da EC2, il sistema di Cloud Computing ideato da Amazon. La virtualizzazione introduce inoltre un modo di ragionare basato sulle *immagini virtuali*: file particolari in grado di contenere il sistema operativo, e tutti i dati necessari al lavoro, che vengono eseguiti e sfruttati dall'hypervisor. In caso di particolari esigenze software è dunque possibile preparare preventivamente un'immagine virtuale opportunamente settata e riutilizzarla per avviare una sessione di lavoro.

L'idea di sistema "elastico" definisce in realtà il fatto che le risorse possano essere richieste e rilasciate in qualsiasi momento. Un tale approccio mira alla fornitura di potenza computazionale on-demand, e non vincola l'utente ad una stima preventiva del workload delle proprie applicazioni. Il paradigma del Cloud Computing si presta in modo particolare a tutti quegli impieghi per i quali non se ne conosce a priori il carico di lavoro. Più esattamente in letteratura questo concetto viene espresso come: *scalabilità* di risorse.

[2] Consultabile al sito: `http://aws.amazon.com/ec2/instance-types/`

In questa nuova tipologia di sistema distribuito l'aspetto economico gioca un ruolo importante. La proprietà conosciuta come "pay-as-you-go" mette in relazione proporzionale l'utilizzo delle risorse con il costo complessivo. Prendendo in considerazione ancora una volta il caso di Amazon, ed osservandone la tabella dei prezzi[3], possiamo notare come questa sia esposta secondo una tariffa oraria e non giornaliera o mensile. Questa caratteristica, insieme con l'elasticità, costituisce un passo verso l'erogazione di potenza computazionale in maniera simile a quanto già accade per corrente elettrica, acqua e gas [42]. Visto da un punto di vista più scientifico, questo approccio solleva la necessità di *accouting* delle risorse ovvero: tracciare precisamente e senza errori i numeri di istanze, GB di dati trasferiti, GB di memoria di massa... dei quali l'utente ha usufruito.

In tabella 1.1 elenco altri punti che giudico importanti ai fini della progettazione di un sistema Cloud, in quanto evidenziano alcuni aspetti in contrasto tra l'architettura di quest'ultimo ed una Grid. Noteremo in 1.5 che privacy e sicurezza sono ancora in via di sviluppo per quanto riguarda le Cloud, mentre nel mondo delle Grid vi sono tecniche ben consolidate [18]. In entrambi i casi invece l'interoperabilità rimane una questione ancora da affrontare, ma nel caso del Cloud Computing, come vedremo successivamente in 1.5, emergono già importanti interessi economici. Alcuni punti vanno a favore dell'utilizzo di una Cloud. In particolare: la minor burocrazia, risolta interamente tramite browser con l'attivazione di un account ed inserimento delle proprie credenziali, vantaggi economici derivanti anche dalla scalabilità su richiesta ("on-demand") che evita l'acquisto di ingenti risorse che rischiano di non essere sfruttate. Nel caso di applicazioni HTC[4] le Grid risultano la scelta migliore, in quanto nate proprio con lo scopo di supportare ricerche scientifiche.

[3] Al sito: `http://aws.amazon.com/ec2/pricing/`

[4] HTC = High Throughput Computing applications

		CLOUD		GRID	
1)	Privacy and Security	work in progress	X	well working methods	√
2)	Interoperability		X		X
3)	Bureaucracy	User: Internet only, Provider: a Cloud is owned only by one provider	√	User: require a strict identification procedure, Provider: collaboration of many organizzations	X
4)	Cost	pay-as-you-go	√	contract defined	X
5)	Scalability	on demand	√	prior fixed size, dependent to the provider	X
6)	Fail-over	particular services provide run-time fail-over	√	relying on the Grid infrastructure	√
7)	Virtualization	provided	√	not provided	X
8)	HTC Applications	data transfer bottleneck	X	designed for scientific applications	√
9)	Application Type	web service, hosting, embarrassingly parallel		scientific, batch	

Tabella 1.1: **Cloud VS Grid** confronto di alcuni punti in contrasto tra l'architettura Cloud e Grid.

1.3 Classificazione dei Modelli di Cloud

In questa sezione seguiremo la stessa linea di [4], dove sono proposti due modelli di classificazione per le Cloud: il primo le suddivide in accordo alla tipologia di servizi offerti, ed il secondo rispetto alle possibilità di deployment.

1.3.1 Modelli basati sui Servizi

In precedenza abbiamo più volte ribadito come le Cloud nascano con l'obiettivo di offrire un servizio all'utente. L'attuale letteratura è andata via via uniformandosi fino ad adottare una gerarchia a livelli, che suddivide questi nuovi sistemi distribuiti in: *IaaS* (Infrastructure as a Service), *PaaS* (Platform as a Service), *SaaS* (Software as a Service). In figura 1.1 ho voluto semplicemente riassumere questa catalogazione, ed aggiungere per ciascuna classe alcuni esempi esistenti. Ho trovato estremamente chiarificatori [49], che offre un'ampia panoramica e dal quale proviene in parte la figura 1.1, e [40], che, sebbene si discosti leggermente dai tre canonici livelli, descrive una dettagliata gerarchia.

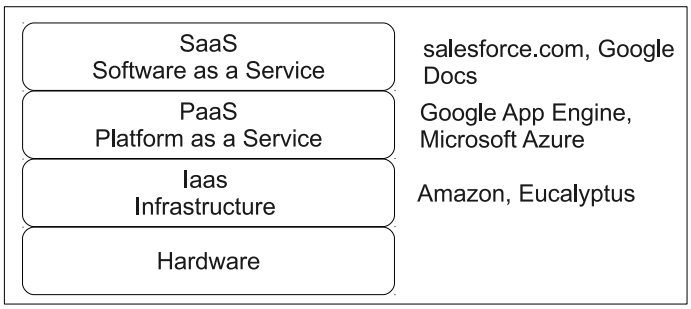

Figura 1.1: Rappresentazione delle architetture Cloud secondo la suddivisione in servizi.

- **IaaS**: è il livello più vicino all'hardware e probabilmente dall'architettura più semplice, tra quelli presentati. L'utente di queste Cloud gestisce completamente le istanze richieste al sistema, ed è in grado di

usarle come workstation accedibili da remoto. Dal punto di vista implementativo sono necessari metodi di coordinamento tra le macchine host, e interfacce che permettano di dialogare con l'hypervisor per lanciare macchine virtuali. Esempi di questa tipologia sono: EC2 (Elastic Compute Cloud) di Amazon, GoGrid, Mosso, e soluzioni opensource come Eucalyptus, OpenNebula, Nimbus e OpenStack.

- **PaaS**: il livello piattaforma sfrutta le operazioni basilari dell'infrastruttura per offrire funzioni più evolute, come ad esempio l'allocazione agevolata di un insieme di istanze, oppure particolari funzioni per il monitoraggio o le comunicazioni. Tuttavia, dal momento che ad oggi non esiste ancora uno stardard in grado di regolare lo sviluppo delle Cloud, non è per niente chiara la soglia che divide il livello interfaccia da quello piattaforma. Il risultato è che le modalità di interazione con questi sistemi cambiano da un provider all'altro. Nel caso di GAE (Google App Engine), ad esempio, occorre scaricare il pacchetto contenente le Application Programming Interface (API), e sfruttarlo nell'implementazione della propria applicazione. Altri esempi del livello piattaforma sono: Azure di Microsoft e SunCloud.

- **SaaS**: al livello software si collocano vere e proprie applicazioni pensate per problemi specifici. Google Docs ricade sotto questa categoria, come anche la Cloud Salesforce.com.

1.3.2 Deployment Models

Mentre in 1.3.1 abbiamo esaminato le Cloud da un punto di vista orientato all'utilizzatore, con i modelli di deployment ci sposteremo al lato provider. Elencando le modalità di implementazione di un tal sistema, possiamo distiguere tra: Cloud *pubbliche*, *private* o *ibride*.
Nella prima ricadono la maggior parte degli esempi citati fino a questo momento: Amazon, Google, Microsoft, IBM e Sun. In questi casi il fornitore eroga al pubblico un servizio di Cloud Computing, ed è responsabile della

manutenzione e del corretto funzionamneto delle macchine.
Nello scenario privato lo stesso ente che ne richiede la costruzione usufruisce anche del sistema. Possiamo immaginare ad esempio, la situazione nella quale un'azieda o università voglia sfruttare i PC che già possiede per calcoli distribuiti. In questo caso diverse sono le alternative, che spaziano da soluzioni open source come Eucalyptus o OpenNebula, per arrivare a compagnie in grado di installare il sistema e fornire manutenzione, come ad esempio opera Cisco.
Le Cloud ibride prevedono aspetti appartenenti sia all'una che all'altra modalità, in quanto oltre ad utilizzare macchine proprie ci si può rivolgere ad un provider pubblico, all'occorrenza di alti picchi di lavoro.

1.4 Volunteer Computing

Quando si parla di Volunteer Computing (VC) ci si riferisce alla condivisione volontaria delle risorse della propria macchina in rete. Alcuni noti progetti si sono avvalsi di questo approccio come ad esempio: SETI@home, Folding@home, Einstein@home, Rosetta@home fino all'attuale BOINC [9].
La domanda che ci poniamo ora è la seguente: è possibile sfruttare una tale architettura per realizzare un sistema in grado di offrire servizi simili a quelli proposti dalle Cloud pubbliche?
Possiamo rispondere asserendo che: anche in questo caso il sistema distribuito che si crea può essere considerato una Cloud, anche se le problematiche che si pongono nell'intraprendere questa strada sono ben evidenti. Prima fra tutte la raggiungibilità di un nodo, che in questo scenario non solo sarebbe condizionata dai guasti, ma anche dalla lecita decisione del proprietario di rimuoverlo. Per quanto riguarda la virtualizzazione occorre poi che tutti i nodi siano allineati per esporre la stessa interfaccia API. Va notato in aggiunta che: se da un lato un sistema a scala globale potrebbe raggiungere prestazioni pressoché elevatissime, dall'altro sarebbe estremamente difficile da tenere sotto controllo. In tabella 1.2 ho riportato alcune differenze tra la

tipologia classica ed il volunteer computing.

CLASSIC CLOUD	VOLUNTEER COMPUTING
eg: EC2, Azure, GAE, Eucalyptus, OpenNebula, Nimbus	eg: SETI@home, Einstein@home, Rosetta@home
virtualization	specific applications
high reliability (99.95%)	reliable level unpredictable
"local" scale	world-spreaded scale
may be public/private/hybrid	completedly public
the user is a client of the Cloud	users have to share some resources

Tabella 1.2: Differenze tra Cloud classiche e l'approccio del volunteer computing.

In [19] gli autori avanzano una proposta interessante che coinvolge i pc degli utenti, ed eventualmente anche alcuni provider. L'obiettivo è quello di creare un sistema di Cloud Computing pubblico e accessibile da chiunque: "Cloud@Home". Uno scenario di questo tipo non è del tutto irrealistico se pensiamo al successo avuto dalle applicazioni P2P. Tuttavia, come avremo modo di osservare in 1.5, il problema dell'interoperabilità è tutt'altro che risolto e costituisce forse l'ostacolo maggiore a questa proposta.

Supponiamo per un momento che esista un sistema Cloud basato su volunteer computing. Sarebbe ancora conveniente utilizzare le "vecchie" soluzioni pubbliche? La risposta data da [22] non è pienamente positiva, anzi solleva alcune questioni interessanti che vale la pena svelare:

- VC richiede 2.83 nodi attivi per egualiare un'istanza "small" di EC2;
- da un punto di vista prettamente economico e calcolando il costo in centesimi al FLOP sono necessari almeno 1404 nodi volontari prima che VC diventi più conveniente;

- 1000 istanze di EC2 in 4 mesi sono in grado di sostenere più di un anno del lavoro del progetto SETI@home;
- con 12K Dollari a mese (che corrisponde alla spesa mensile di SETI@home) è possibile acquistare istanze con alte prestazioni CPU fino a raggiungere un massimo di 2 TeraFLOP;
- bisogna inoltre considerare che la probabilità di oltrepassare la deadline di un progetto aumenta in VC, proprio a causa dell'arbitrarietà con la quale i nodi possono unirsi o lasciare la Cloud.

Nonostante le problematiche, il volunteer computing rimane un campo piuttosto esplorato, in quanto permette il raggiungimento di una potenza computazionale estremamente elevata. Se applicato in aziende o enti pubblici poi, consentirebbe l'utilizzo di risorse che altrimenti resterebbero infruttuose.

1.5 Difficoltà implementative

Facendo parte della classe dei sistemi distribuiti, le Cloud ereditano da questi non solo i benefici ma anche le difficoltà di implementazione. Abbiamo infatti già accennato al coordinamento di un vasto insieme di elaboratori indipendenti. Vanno considerati inoltre: la tecnologia di virtualizzazione e la necessità di dialogare con un hypervisor, e l'allocazione dinamica delle risorse per garantire la proprietà di elasticità. Tuttavia vi sono più rilevanti difficoltà implementative a cui devono far fronte i provider, e che in parte ostacolano lo sviluppo delle Cloud, riassunte nella lista che segue:

1. Sicurezza dei dati;
2. Licenze di software proprietario;
3. Interoperabilità tra Cloud;
4. Scalabilità della memoria di massa;

5. Colli di bottiglia nei trasferimenti;

6. SLA (Service Level Agreement);

7. Availability.

Sicurezza dei dati

Lo scenario delle Cloud solleva complicate questioni riguardo la sicurezza, in particolare nel modello pubblico, dove oltre alle consuete problematiche di un sistema distribuito, si aggiunge l'ingente numero di interazioni che il cliente richiede.

Ciacun utente infatti: produce traffico dati tra la propria postazione e la Cloud, o tra le istanze acquistate, può salvare dati in immagini virtuali anche per diversi GB di memoria o usare il browser per monitorare la situazione della proria applicazione. Il tutto deve essere eseguito in sicurezza, senza la possibilità di perdita o addirittura furto di informazioni.

Licenze di software proprietario

Il problema della distribuzione del software è tutt'altro che banale, in particolare nel settore aziendale. Oggigiorno esistono licenze che consentono di installare un determinato programma su più macchine.

Abbiamo anche osservato che tra i vantaggi delle Cloud risiede quello della scalabilità, utile in particolare per attività delle quali non si riesce a stimare il carico lavorativo a priori. È lecito assumere allora che l'obiettivo di un cliente sia quello di usufruire di questa potenzialità.

Ma quanto può costare ad un'azienda l'acquisto di un ingente numero di istanze sulle quali deve essere installato software proprietario? Sotto quali condizioni la propagazione del software si mantiene lecita? Le risposte e le soluzioni devono provenire dalle case sviluppatrici, che dovrebbero prevedere questa eventualità nel caso d'uso del software, fornire mezzi per il calcolo distribuito, ed ovviamente corredarlo di licenze adeguate.

Interoperabilità

Con interoperabilità si intende la possibilità di migrare un'applicazione da una Cloud ad un'altra senza dover scrivere codice di supporto per adattarla alla nuova interfaccia. Alcuni autori definiscono questo problema con il termine "data lock-in". Una volta infatti che l'utente ha scelto il fornitore più adeguato alle proprie esigenze, gli sarà molto difficile cambiarlo in futuro per via delle differenti API, nonostante magari i sistemi siano catalogati secondo lo stesso modello di servizio. Questo sembra al momento lo scoglio maggiore alla diffusione del Cloud Computing. Diversi avanzano l'ipotesi che una vera soluzione si possa raggiungere solo attraverso uno standard comune a tutti i provider. Proprio questi ultimi però ricavano interessi economici dalla mancanza di interoperabilità, che disincentiva l'allontanamento dei clienti.

Scalabilità della memoria di massa

Con lo scopo di pubblicizzare il Cloud Computing, in alcuni contesti l'illusione di memoria infinita è mostrata tra i vantaggi. Con quali tecniche sia però possibile mantenere una tal promessa rimane ancora tutto da dimostrare. Nella progettazione del sistema di memoria di massa per una Cloud rientrano diverse problematiche: la complessità delle strutture dati, le prestazioni degli algoritmi e dell'hardware e le strategie di manipolazione delle informazioni di controllo. Il tutto deve svolgersi poi mantenendo inalterato il vantaggio di elasticità che le Cloud offrono agli utilizzatori. Al concetto di memoria di massa si devono affiancare inoltre quello di durabilità e reperibilità dei dati anche a fronte di guasti, che si traducono nel mantenere copie ridondanti di questi, con ulteriore dispendio di risorse fisiche e computazionali.

Trasferimenti di dati

Gli applicativi software oggi tendono ad occupare sempre più spazio su disco, cosa che non desta particolare preoccupazione viste le odierne architet-

ture hardware. I problemi nascono quando dati di grandi dimensioni devono essere scambiati attraverso la rete. Se poi vi aggiungiamo anche i costi di trasferimento, calcolati in bandwidth e tempo macchina in entrata o in uscita da una Cloud, allora conviene selezionare accuratamente le informazioni davvero necessarie dalle altre. D'altro canto, grazie ai notevoli sforzi dei provider, che si concretizzano in protocolli sempre più efficienti e strategie implementative particolarmente curate, le Cloud raggiungono buone prestazioni per quanto riguarda invece lo scambio di dati tra istanze interne, anche se non sempre comparabile ai già collaudati sitemi Grid.

SLA

La sigla SLA (ovvero Service Level Agreement) indica un contratto tra il fornitore di un servizio e l'acquirente. Esso regola i rapporti di utilizzo e le prestazioni minime che il sistema deve fornire. Più elevata è la soglia delle garanzie che un provider assicura, e meno responsabilità (in termini di gestione dei guasti) resteranno da risolvere all'utente per permettere il corretto funzionamento della propria applicazione. In un ambiente come quello delle Cloud non è per niente banale poter garantire anche le proprietà più semplici come: tempi di risposta ragionevoli, schedulazione ordinata delle richieste, o l'assegnamento equilibrato delle risorse, dal momento che la stessa affidabilità delle macchine è soggetta ad un valore di incertezza dovuto agli inevitabili guasti.

Availability

Ad oggi la percentuale annua di reperibilità di un servizio di Cloud Computing pubblico in genere supera il 99.90%. A giudicare da [55], per grandi imprese, che dispongono di un mainframe o cluster completamente dedicato ai servizi di rete, un simile valore non è ancora sufficiente. Alcune organizzazioni arrivano addirittura a sfiorare il 99.987% di media annua. Parlando in altri termini: una simile impresa vanta solo un'ora all'anno di inattività dei

propri servizi in rete, tempo che sarebbe quadruplicato se si affidasse esclusivamente ad una Cloud pubblica. Per attirare anche le aziende che investono larga parte della propria visibilità online, è essenziale poter innalzare questo valore.

Capitolo 2

Stato dell'arte

Questo capitolo prende in esame alcuni esempi di Cloud tra quelli osservati. Viene introdotta l'infrastruttura di Amazon 2.1 che si colloca molto probabilmente tra le più citate in letteratura. Per questo motivo credo sia importante analizzarne i concetti basilari ed osservarla dal punto di vista dell'utilizzatore, per poi confrontarla con le attuali difficoltà implementative che si riscontrano nello scenario del Cloud Computing . In 2.2 introduco Eucalyptus, un software open source completamente compatibile con la Cloud di Amazon, attualmente disponibile già in alcune versioni di Linux. Seguirà poi in 2.3 una breve presentazione di altri due sistemi: OpenNebula e OpenStack. Tutte le precedenti architetture si posizionano al livello infrastruttura, e vengono messe a confronto in 2.4 per analizzarne i punti in comune e le divergenze rispetto alle API che offrono. Conclude il capitolo la sezione 2.5 con alcune esperienze di Cloud P2P.

2.1 Amazon EC2

In questa sezione esploreremo alcune caratteristiche di EC2 (Elastic Compute Cloud) di Amazon. Al sito riportato in [2] si trova la documentazione di tutti i servizi AWS (Amazon Web Services). Questa non vuole certo essere una guida esaustiva, quanto piuttosto un riassunto di alcuni concetti basilari per l'utilizzo del sistema, estratti dal manuale per lo sviluppatore [1].
Lo scopo di questa indagine è di raccogliere informazioni in grado di svelare con quali strategie gli sviluppatori hanno risolto i problemi architetturali durante la costruzione della Cloud e trarne eventualmente vantaggio nella fase di progettazione di un nostro sistema.

2.1.1 Istanze e AMI

Il caso d'uso comune per l'utilizzatore dei servizi Amazon prevede che questi faccia richiesta di un certo numero di istanze. L'**istanza** è il sistema risultante dall'avvio di un'**AMI** (Amazon Machine Image) ovvero un file d'immagine criptato, contentente tutto il necessario per l'esecuzione dell'istanza stessa. All'interno dell'AMI deve risiedere almeno la partizione di root con installato il sistema operativo. Ancora ad oggi non è stata rivelata la struttura interna di questi file e rimane difficile estrarne il contenuto. L'istanza può essere avviata da terminale con il comando `ec2-run-instances` <ami_id>, oppure attraverso l'interfaccia web, ed assomiglia ad un host tradizionale. Il possessore di questa ne ha il completo controllo, compreso l'accesso come amministratore. Amazon prevede un esteso elenco di AMI di default che possono essere usate per un primo avvio. L'utente è altresí autorizzato a crearne di proprie sfruttando un'istanza che già possiede, o eseguire l'upload tramite la propria postazione. Una volta selezionata l'immagine, può quindi scegliere tra due alternative per quanto riguarda il sistema d'appoggio di memoria di massa: avvalersi di EBS oppure S3.
EBS (Elastic Block Store) e **S3** (Simple Storage Service) provvedono in modi diversi alla memorizzazione permanente dei dati. Una terza opzione

in realtà esclude l'impiego di entrambi e sfrutta semplicemente la memoria che viene attribuita all'istanza, che tuttavia è da considerarsi volatile perchè cancellata ad ogni terminazione di questa. Ciascuno dei due sistemi prevede caratteristiche diverse. Un'AMI che si appoggia su EBS:

- supporta un meccanismo di snapshot per tutti i volumi EBS collegati, che vengono compressi in apposite immagini e salvati in S3,
- può riutilizzare un vecchio snapshot per creare e lanciare una nuova AMI,
- vanta tempi di avvio in genere migliori rispetto ad S3,
- può raggiungre come limite massimo di memoria di massa 1TB.
- consente la strategia stop/start.

La strategia di stop/start permette di portare l'istanza in uno stato di "*sospensione*" senza terminarla. All'istanza sospesa non sono addebitati i costi orari e vengono preservati i volumi ad essa collegati. Contrariamente in S3 un'immagine:

- viene compressa, quindi criptata e firmata,
- suddivisa in piccole parti che meglio si prestano all'upload,
- ne viene creato un file manifest che raccoglie le informazioni relative ai frammenti ed ai loro checksum,
- può raggiungere come limite massimo di memoria di massa 10GB,
- non consente la strategia di stop/start.

Gli elenchi precedenti vogliono semplicemente sottolineare che il momento della scelta dell'una o dell'altra alternativa può rivelarsi cruciale per il successo dell'applicazione o meno. Nel manuale infatti viene più volte ribadito che l'utente stesso deve impegnarsi a: mettere al sicuro i dati importanti, salvandoli nella memoria permanente (sfruttando EBS o S3), adottare la tipologia di macchina che meglio si adegua alle esigenze, controllare lo stato delle proprie risorse, usare le opportune misure di sicurezza.

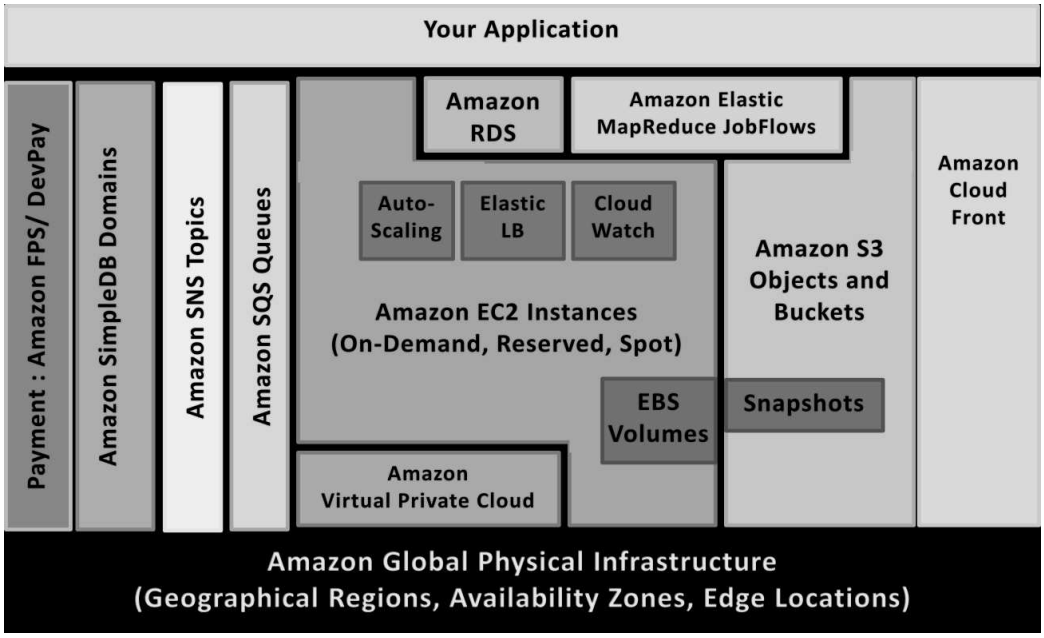

Figura 2.1: Architettura di Amazon Web Services come proposta da [59]

Amazon offre servizi addizionali in grado di semplificare i compiti all'utente. Uno tra questi è Cloud Watch: un monitor che permette di osservare le instanze ed i volumi posseduti. Amazon AutoScale consente invece di dimensionare il numero di risorse allocate attraverso la specifica di alcuni parametri, anche durante l'esecuzione dell'applicazione.

La gamma di prodotti proposta è così vasta che è possibile perdersi nell'analizzarli ad uno ad uno, contando poi anche il fatto che per ciascuno compare una corposa documentazione. Ho trovato utile lo schema di figura 2.1 che riporta, seppur a grandi linee, l'architettura di Amazon.

2.1.2 Altri concetti importanti

Dal momento che ricoprire interamente tutti gli argomenti concernenti gli AWS richiede un discreto dispendio di tempo, nelle righe seguenti cercheremo di riassumere alcuni tra gli aspetti più rilevanti.
Amazon è in grado di garantire la raggiungibilità degli host attraverso la

dislocazione spaziale delle macchine fisiche ed introduce i concetti di regioni e zone di raggiungibilità. Le **Regioni** sono distribuite in zone geografiche separate. All'interno di una regione si distinguono diverse aree dette **Availability Zones**. Ciascuna è progettata in modo da essere isolata dalle altre in caso di guasti, e fra tutte queste viene garantita una connessione a bassa latenza.
Un'istanza all'interno della Cloud possiede due indirizzi IP: **pubblico** e **privato**. Il primo consente l'accesso all'host tramite internet, mentre il secondo serve alle comunicazioni interne alla Cloud. Amazon permette inoltre d'adottare uno o più **indirizzi elastici**: questi si riferiscono all'utente e non ad una particolare macchina. Possono risultare utili per quei servizi che devono garantire un'elevata disponibilità. Starà poi allo sviluppatore dell'applicazione associare a ciascun indirizzo elastico un'istanza, e rimpiazzarla nel caso questa fallisca.
Prendendo in esame ora il problema della sicurezza, possiamo notare come Amazon si sia servita di diversi mezzi per raggiungere un buon livello. Chiaramente qualunque interazione con l'infrastruttura necessita l'autenticazione, il che rende l'utilizzatore identificabile. Al momento dell'attivazione dell'account viene generata una coppia di chiavi pubblica-privata per RSA. La connessione diretta ad un'istanza è sprovvista di password, ma viene consentita solo attraverso connessioni sicure in grado di gestire il suddetto protocollo: sull'istanza viene copiata la chiave pubblica, mentre l'utente disporrà di quella privata. L'accesso diretto corrisponde solo ad una tra le possibilità di comunicazione con gli host instanziati. Le alternative sfruttano l'approccio di richieste REST e l'invio di query oppure l'architettura a web service mediante il protocollo SOAP. Per i primi occorre generare una coppia di chiavi costituita da un identificatore di accesso (access key ID) ed una chiave segreta. Nel caso di interrogazione tramite web services è adottato un certificato X.509 accoppiato ad una chiave privata, che può essere scaricata una sola volta, ed in caso di smarrimento di quest'ultima è necessario generarne uno nuovo. Le AMI salvate in S3 vegono criptate per garantirne

l'accesso solo al possessore. Inoltre gli stessi autori consigliano di usare volumi criptati per manipolare dati importanti. Viene anche ribadito più volte che è responsabilità dell'utente rendere sicure le proprie sessioni di lavoro settando opportunamente le impostazioni dei firewall, dei **gruppi di sicurezza** (security groups), e scegliere quali porte aprire e quali mantenere chiuse.

2.1.3 Utilizzo della Cloud

La sequenza di passi necessari per usufruire di EC2 è la seguente:

1. ottenere un account,
2. autenticarsi presso il portale AWS,
3. cliccare sulla voce "launch instance" per avviare la procedura guidata di avvio di un'istanza,
4. scegliere un'AMI
   (a) selezionare una tra le proposte base,
   (b) sfruttarne una propria,
5. creare una coppia di chiavi per l'accesso futuro (ad esempio tramite ssh),
6. creare un gruppo di sicurezza: devono essere impostate le regole di accesso (ad esempio filtrare il traffico entrante ed uscente dalla porta 80),
7. lanciare le istanze selezionate (da questo momento vengono addebitate le ore di operatività consumate),
   (a) è bene appuntare l'indirizzo DNS pubblico dell'istanza per potervi accedere in fututro,
8. connettersi all'istanza (tramite ssh o analoghi programmi in ambiente Windows),
9. gestire l'host e l'applicazione che vi è destinata,
10. terminare l'istanza da console o da browser (fine dell'addebitamento per quella risorsa).

L'interazione con un'istanza avviene non solo mediante connessione con ssh e uso dei comandi da terminale, ma anche con l'interrogazione del sistema avvalendosi di REST oppure del protocollo SOAP e l'invio di richieste strutturate in file XML. In questo modo lo sviluppatore deve disporre della lista delle API per padroneggiare tutto il necessario per la propria applicazione.
Per effettuare una stima preventiva dei costi di una simile soluzione occorre consultare la tabella dei prezzi tenendo in considerazione alcune variabili. In particolare la granularità minima di un'ora attuata per le istanze, infatti dal primo momento di esecuzione viene addebitato all'account che la possiede il costo orario. Si devono anche considerare i GB di spazio disco utilizzato per il salvataggio qualora l'host venga terminato, e la quantità di dati in entrata ed in uscita dalla Cloud.
Amazon si solleva dalla responsabilità di ogni fallimento degli host richiesti. È bene dunque adottare diverse strategie per la gestione dei guasti, sia all'interno della propria applicazione con opportune scelte ingegneristiche di gestione distribuita, che con l'ausiglio di strumenti come AutoScale, i quali però contribuiscono ad innalzare i costi.

2.1.4 Difficoltà implementative in EC2

Amazon rappresenta un ottimo esempio di un sistema completamente funzionante in grado di supportare numerosi clienti. Vale dunque la pena spendere qualche parola cercando di analizzare come gli sviluppatori hanno fatto fronte alle difficoltà implementative presentate in 1.5.

1)**Sicurezza**	
•Credenziali della persona fisica raccolte nell'account,	√
•Immagini criptate in S3	√
•Tecniche di comunicazioni sicure: RSA, certificato X.509, coppia di chiavi d'accesso e segreta per interrogazioni con REST	√
2) **Licenze Software**	

•Il livello infrastruttura è troppo basso per affrontare questo problema. Amazon offre semplicemente host aventi come sistema operativo Linux o Microsoft Windows Server, per il quale probabilmente ha stipulato un accordo direttamente con la casa produttrice.	∼
•Non sono rese note le informazioni per decriptare ed estrarre le immagini di sistemi operativi proprietari.	√
3) **Interoperabilità**	
•Al momento l'azienda è ritenuta da molti come la favorita a diventare lo standard di fatto per quanto riguarda l'interfaccia che espone al pubblico. Sembra inoltre che stia compiendo alcuni passi che aprono la strata per un dialogo con altri provider. Tuttavia non compare ancora nulla di ufficiale.	∼
•Sebbene sia possibile scaricare le immagini AMI e utilizzarle in altri abienti, Amazon non ha tutt'ora aderito all' Open Virtualization Format (OVF) [61].	∼
4) **Scalabilità della memoria di massa**	
•Le soglie dei 10 GB per immagini in S3 e di 1 TB per quelle in EBS sono in grado di soddisfare le esigenze di un vasto numero di applicazioni.	√
•L'utente ha a disposizione uno spazio complessivo in memoria di massa piuttosto elevato: può arrivare ad una massimo di 100 volumi o 20 TB come occupazione massima in EBS.	√
•Nel caso si necessiti di maggiore memoria è possibile ampliare le specifiche di default compilando un modulo sul sito.	√
5) **Colli di bottiglia nei trasferimenti**	
•Per l'upload di un'immagine esistente Amazon consente fino a 5 connessioni attive allo stesso tempo per regione.	√
6) **SLA**	

•Le pagine online dedicate alle politiche di utilizzo di EC2 e S3 illustrano prestazioni eccellenti, questo se si evita il caso particolare di un'imponente azienda che cecessita di repreribilità superiore al 99,95%. •L'utente può avvalersi di diversi servizi quali AutoScale o LoadBalance per gestire il traffico e le risorse della propria applicazione e mantenere così alte le prestazioni. •Il sistema EBS è in grado di identificare i fallimenti e di reagire ad essi immediatamente con uno snapshot, per prevenire la perdita dei dati.	√

Tabella 2.1: Elenco delle difficoltà implementative raccolte nel capitolo 1.5 rapportate ai servizi offerti da Amazon.

Il successo di un tale sistema risulta giustificato visti gli sforzi apportati per le prestazioni della memoria, gli innumerevoli servizi proposti, le vantaggiose condizioni di utilizzo. Tuttavia non viene smentita la volontà di vincolare all'azienda il bacino d'utenza sfruttando l'ostacolo dell'interoperabilità. Nelle apparizioni al pubblico i responsabili del marchio tengono a sottolineare il fatto che non vogliono generare rapporti di dipendenza coi clienti. Tuttavia, allo stesso modo di altri importanti provider, non esibiscono ancora azioni ufficiali che conducano ad un utilizzo delle Cloud esteso a tutti i concorrenti [23, 58, 39].

2.2 Eucalyptus

Eucalyptus è un software open source volto allo sviluppo di un'infrastruttura (IaaS) per una Cloud ibrida. Nasce all'interno del progetto MAYHEM presso i laboratori del dipartimento di Informatica di Santa Barbara [3] grazie ad un gruppo di persone già esperto nel calcolo distribuito. Il progetto si

è talmente affermato che, a partire dal 2009, viene distribuito anche in una versione commerciale. Nelle ultime release di Linux Ubuntu lo si ritrova tra i pacchetti di default o addirittura integrato nell'installazione. In questa esposizione ci dedicheremo alla sezione open source, che si pone come obiettivo quello di indagare il mondo del Cloud computing mettendo l'implementazione del sistema a servizio della ricerca. A questo proposito la comunità di Eucalyptus offre un testbed pubblico e si impegna a garantirne un supporto il più continuativo possibile, anche se pur sempre best-effort, e ne esclude l'impiego per lo sviluppo di applicazioni commerciali.
Gli autori in [44] svelano alcune domande che hanno mosso la creazione del progetto, e che per molti aspetti coincidono con le motivazioni di questo lavoro. In particolare: quale architettura distribuita è in grado di supportare al meglio un sistema di Cloud Computing ? Quali politiche deve adottare e che caratteristiche deve avere uno scheduler che sia in grado di ottimizzare l'utilizzo delle risorse a fronte di molteplici richieste? Che tipo di interfacce sono appropriate per un sistema Cloud e che tipo di garanzie un utente può aspettarsi da questo?

L'architettura

Eucalyptus è realizzato con l'ausilio di diversi linguaggi, tra i quali C, Java e Phyton, e progettato con un approccio modulare per essere facile da installare ed il meno intrusivo possibile. Supporta le macchine virtuali di Xen e Kvm. Molti degli aspetti visti per EC2 vengono ripresi in Eucalyptus, come ad esempio: l'utilizzo delle immagini virtuali per l'avvio di istanze e la loro gestione, la presenza in queste di un duplice indirizzo IP pubblico e privato, la possibilità di definire gruppi di sicurezza (security groups). Possiamo quindi asserire che la filosofia di utilizzo ha molti punti in comune con la nota Cloud commerciale; in più l'insieme delle API risulta compatibile con EC2 e S3. Vedremo in seguito esserne proprio un sottoinsieme.

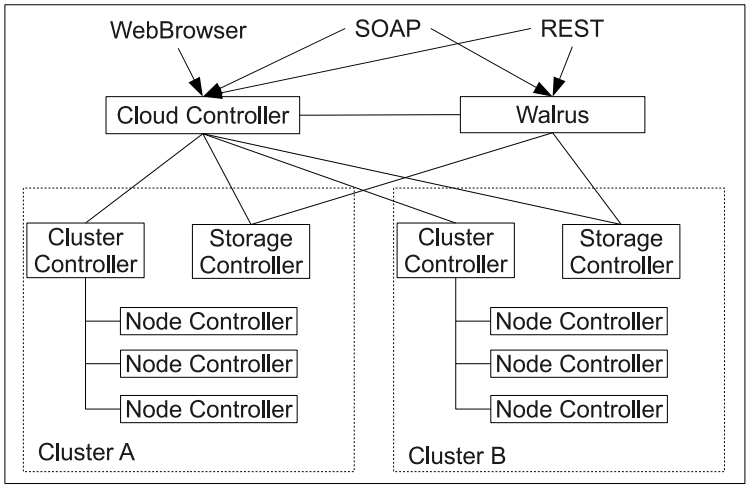

Figura 2.2: Architettura di Eucalyptus. Il sistema risulta suddiviso in moduli ciascuno dei quali svolge diverse mansioni ed ha una particolare posizione nella gerarchia. In figura viene anche mostrata la ripartizione nel caso venga costruita una rete dove diverse macchine formano più cluster.

Uno sguardo più approfondito all'architettura mostra che il sistema è costituito da cinque moduli ad alto livello e sono riassunti in figura 2.2.

- **Node Controller**: è il componente che ciascun nodo della cloud deve possedere. Controlla l'esecuzione, l'ispezione e la terminazione delle istanze virtuali poste nella macchina sulla quale vengono eseguite.

- **Cluster Controller**: nel caso vi sia la necessità di suddividere l'insieme delle macchine in diversi cluster, questo componente raccoglie tutte le informazioni necessarie per la schedulazione delle richieste, e gestisce le reti virtuali, tramite l'interrogazione di ciascun node controller appartenente al proprio gruppo.

- **Storage Controller (Walrus)**: è un servizio di allocazione di memoria di massa che implementa l'interfaccia di Amazon S3 e provvede un meccanismo per il mantenimento e l'accesso delle immagini virtuali degli utenti.

- **Cloud Controller**: è il componente di più alto livello ed il punto di accesso alla Cloud per utenti ed amministratori.

All'interno di ciascun nodo deve essere in esecuzione un Node Controller, che ne comanda il software e si occupa quindi dell'interazione con l'hypervisor. Risponde alle richieste di *describeResource*, *describeInstances*, *runInstances* e *terminateInstances* che gli pervengono dal Cluster Controller al quale è associato. Dopo aver accertato l'autorizzazione della richiesta e verificato le risorse, il Node Controller effettua una copia dei file di immagine da un repository remoto o dalla cache locale, ed istruisce l'hypervisor per eseguire l'istanza.
La consueta topologia di una Cloud che fa uso di Eucalyptus, prevede che vi sia almeno una macchina in grado di funzionare come front-end, sulla quale dovrà essere eseguito l'unico Cloud Controller. Questo componente definisce il punto di ingresso al sistema e ne consente l'esplorazione e la gestione tramite browser.
La disposizione del Cluster Controller non è invece altrettanto rigida. Infatti può risiedere sia sul nodo front-end che in un qualunque altro adibito alla comunicazione tra cluster differenti. Molte operazioni del Cluster Controller sono simili a quelle di un Node Controller ma distribuite tra più nodi dei quali dovrà collezionare le risposte. Possiamo riassumere le tre funzioni primarie di questo componente in: ripartire le richieste in ingresso, raccogliere le informazioni riguardo ai nodi inclusi nel cluster e gestire la rete virtuale di collegamento tra questi.
Il componente Walrus è il gestore della memoria di massa ed implementa interfacce REST e SOAP compatibili con S3. Condivide con il Cloud Controller le credenziali degli utenti che vengono usate per cifrare le immagini salvate. Similmente ad S3 suddivide i grossi file in parti più piccole per essere meglio gestite.

Utilizzo

Sul sito di riferimento [3] è possibile seguire passo passo la procedura di installazione sulla prima macchina destinata alla cloud. A questo punto, invece che ripeterla nuovamente su ciascun nodo, possiamo servici del comando da terminale `rsync`, che allinea due directory allo stesso contenuto. È bene quindi adottare lo stesso percorso come posizione root di Eucalyptus al fine di semplificare la distribuzione su tutti i calcolatori.
Per ultimare l'installazione di un host è necessario:

- aggiungere un utente apposito: "eucalyptus",
- configurare l'hypervisor,
- configurare la rete,
- scegliere i componenti necessari al nodo in base alla mansione che ricopre, pianificando cioè se dovrà comportarsi come semplice nodo (*Compute Node*) oppure come un controllore di cluster o della cloud,
- infine configurare gli script di avvio, per dare ad esempio la possibilità al nodo di collegarsi direttamente alla Cloud immediatamente dopo l'avvio.

Gli sviluppatori hanno provveduto una funzione particolare `euca_conf` in grado di agevolare il processo , che modifica gli opportuni parametri all'interno del file "eucalyptus.conf", essenziale al nodo in quando raccoglie tutte le informazioni necessarie per l'avvio. Come ultimo passo rimane solo quello di avviare la macchina front-end e *registrare* tutte le altre ad essa attraverso lo stesso comando di configurazione con gli opportuni parametri.
L'amministratore potrà accedere tramite browser al front end da https://<front-end-ip>:8443 e controllare così le condizioni della cloud. Avrà la possibilità anche di specificare alcune opzioni quali: la dimensione massima per volume, la quantità totale di spazio consentita per tutti i volumi e gli snapshot. Ciascun utente può effettuare l'upload e la registrazione di una qualunque immagine, ma solo gli amministratori possono introdurre e

rimuovere immagini kenel.
Mentre l'interfaccia browser è lo strumento preferenziale per l'amministratore, l'utente interagirà prevalentemente da linea di comando sfruttando le API di Eucalyptus, che prendono il nome di *euca2ools*, e che in larga parte si rifanno a quelle ideate da Amazon.
Per il monitoraggio, per così dire avanzato delle risorse, occorre invece affidarsi a strumenti esterni come ad esempio Ganglia o Nagios.

2.3 Altre Cloud prese in esame

Al fine di ampliare la visuale nel campo del Cloud Computing nelle seguenti sezioni presenterò molto brevemente due ulteriori esempi: OpenNebula ed OpenStack, che come si evince dai nomi, appartengono alla corrente open source. Il primo progetto si discosta da quanto abbiamo visto in 1.3 riguardo alla classificazione delle Cloud rispetto ai servizi, in quanto introduce un livello addizionale. Il secondo, attualmente alle prime release, è in fase sperimentale ma si prospetta estremamente promettente viste le importanti organizzazioni che lo sostengono. Entrambi si collocano al livello infrastruttura seppur con obiettivi e soluzioni completamente diverse, pertanto è interessante esporne, almeno a grandi linee le architetture.

2.3.1 OpenNebula

OpenNebula [54] nasce dalla collaborazione delle Università di Chicago e Madrid. L'obiettivo del progetto è quello di fornire un sistema di gestione all'interno di una Cloud privata o ibrida che si pone ad un livello intermedio tra l'infrastruttura e la piattaforma, denominato *Virtual Infrastructure Management*, in grado di amministrare in maniera ottimizzata le immagini virtuali. La figura 2.3 evidenzia l'iserimento di questo elemento addizionale rispetto alla classificazione data precedentemente in 1.3.1, e in rapporto alle architetture di altri sistemi esistenti come Amazon o Eucalyptus.
Tra le principali caratteristiche che gli sviluppatori tengono a sottolienare

segnaliamo: la possibilità di ampliare la stessa Cloud su altre esterne, ad esempio avvalendosi di EC2 o Eucalyptus, grazie al supporto di gestione delle immagini virtuali, il superamento della struttura monolitica delle attuali proposte, ed in particolare l'approvigionamento delle macchine virtuali attraverso strategie più elaborate delle comuni "first-fit", o round robin.
L'architettura si compone del modulo principale, il *core*, che governa le operazioni sulla memoria di massa, la rete e l'hypervisor sottostante. Il core opera su particolari *driver* che sono specifici della tecnologia di virtualizzazione e della rete. Proprio i driver sono le uniche parti che necessitano di modifiche qualora si decida di migrare il sistema, lasciando l'intero insieme di *servizi* del "virtual infrastructure management" inalterato. Infine lo *scheduler* è responsabile della raccolta delle richieste dai livelli superiori, del monitoraggio della situazione delle risorse allocate e della spedizione degli opportuni comandi per il deploy al componente core.
Fin dalle prime fasi gli autori hanno voluto mantenere una qualità del software elevata con lo scopo di soddisfare le esigenze degli ambienti produttivi. Ulteriori sforzi hanno permesso l'ingresso di OpenNebula nel progetto europeo RESERVOIR consentendone la prima esperienza su casi d'uso reali. L'inclusione nella versione di Linux Ubuntu 9.04 ne ha poi accelerato la diffusione ed il bacino di utenza.

2.3.2 OpenStack

Due illustri nomi, quali Nasa[1] e Rackspace[2], hanno fondato il marchio *OpenStack* che, attraverso i due progetti di *Nova* e *Swift*, offre un'ulteriore alternativa nel panorama del Cloud Computing . Il primo prende in considerazione tutto ciò che concerne il coordinamento delle risorse computazionali, mentre il secondo si occupa della memoria di massa. Entrambe le società vantano un'ingente potenza di calcolo. Condividono tuttavia lo stesso problema di scalabilità: devono far fronte ad elevati valori in termini di macchine

[1] `http://www.nasa.gov/`
[2] `http://www.rackspace.com/index.php`

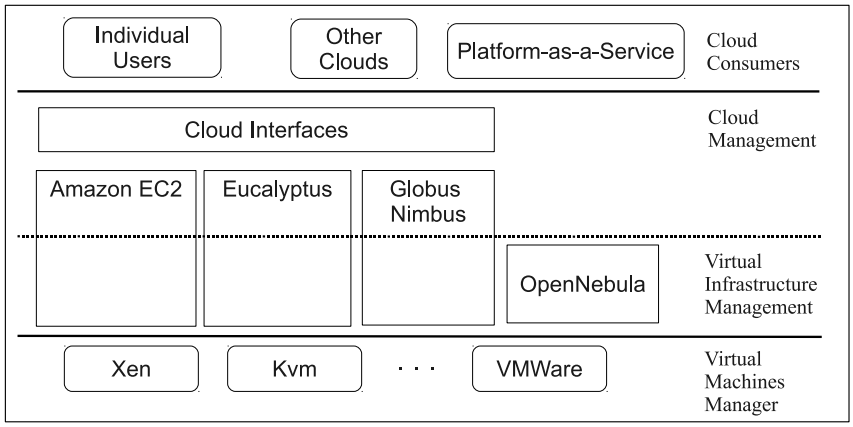

Figura 2.3: Panoramica architetturale per la costruzione di Cloud private nella quale è inserito uno strato addizionale tra il livello infrastruttura e piattaforma denominato *Virtual Infrastructure Management* all'interno del quale si inserisce OpenNebula.

allocate e rimosse (Rackspace conta circa 100.000 clienti), ed è proprio cercando una soluzione che nasce OpenStack.
Viene descritto come un "software in grado di controllare l'infrastruttura di una Cloud" per via del fatto che piuttosto che implementare completamente i moduli necessari al funzionamento, tende a riutilizzare molti di quelli già esistenti. Si affida infatti al software di virtualizzazione presente sulla macchina host senza implementarne uno ulteriore, ed espone lo stesso set di API di Eucalyptus, lasciando pensare ad un completo riutilizzo dello stesso. Le modalità di accesso comprendono la creazione di *progetti* con la specifica degli utenti autorizzati a lavorarvi.
Nella rimanente parte di questa sezione ci occuperemo in modo particolare di Nova e la figura 2.4 ne introduce l'architettura ed i componenti principali del sistema.
L'**API Server** costituisce il punto d'accesso. Manipola i comandi di controllo per l'hypervisor, la memoria di massa e la rete. Le API Endpoints sono semplici web services in grado di gestire l'autenticazione e le autorizzazioni, compresi alcuni comandi e funzioni di controllo. Il punto di smistamento

tra le richieste sottomesse dagli utenti e i moduli del sistema che dovranno occuparsene è rappresentato dall'oggetto **Message Queue**. È costruito seguendo una politica che esclude la condivisione e predilige invece lo scambio di messaggi.
All'invio di ogni richiesta l'API server ne verifica l'autorizzazione, se positiva la inserisce nella coda. I moduli Compute Worker periodicamente controllano la coda, ed eseguono le richieste contenute. Il **Network Controller** alloca l'indirizzo IP fisso dell'host e configura la rete LAN virtuale per i progetti.

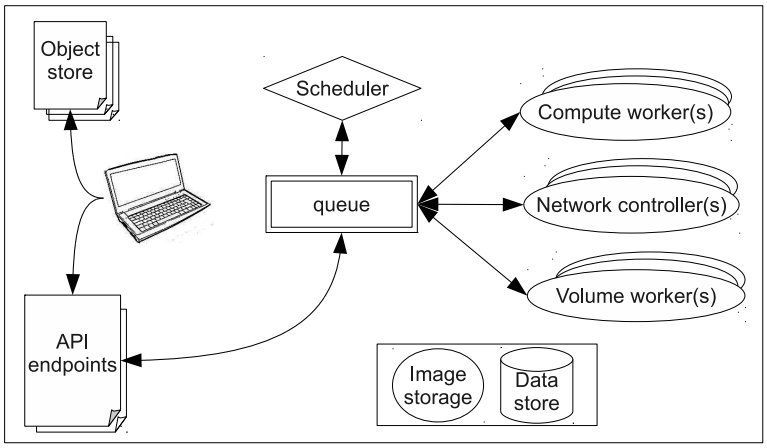

Figura 2.4: Architettura del progetto *Nova* di OpenStack che si occupa delle risorse computazionali del sistema Cloud.

2.4 Confronto tra API

In figura 2.2 ho raccolto e messo a confronto le interfaccie dei sistemi presentati nelle sezioni precedenti. Mi sono attenuto alla suddivisione, adottata anche da Amazon, che prevede gli strumenti raggruppati per: istanze, monitoraggio, volumi e memoria di massa, indirizzamento, immagini, gruppi di sicurezza e zone di raggiungibilità. Quello presentato non è l'elenco completo, ma un sottoinsieme minimale per l'utilizzo della Cloud al livello infrastruttura.

API COMPARISON					
	AMAZON (EC2 - S3)	**EUCALYPTUS**	**OpenNebula**	**OpenStack**	**Nimbus**
	Computation				
Instance Tools	ec2-run-instances	euca-run-instances	run instances	euca-run-instances	ec2-run-instances
	ec2-describe-instance-attribute				
	ec2-describe-instances	euca-describe-instances	describe instances	euca-describe-instances	ec2-describe-instances
	ec2-get-console-output	euca-get-console-output		euca-get-console-output	
	ec2-reboot-instances	euca-reboot-instances		euca-reboot-instances	ec2-reboot-instances
	ec2-start-instances				
	ec2-stop-instances				
	ec2-terminate-instances	euca-terminate-instance	terminate instances	euca-terminate-instances	ec2-terminate-instances
Monitor	ec2-monitor-instances	use Nagios, or Ganglia		Ganglia	
	ec2-unmonitor-instances				
EBS Tools	ec2-create-volume	euca-create-volume		euca-create-volume	
	ec2-attach-volume	euca-attach-volume		euca-attach-volume	
	ec2-create-snapshot	euca-create-snapshot		euca-create-snapshot	
	ec2-delete-snapshot	euca-delete-snapshot		euca-delete-snapshot	
	ec2-delete-volume	euca-delete-volume		euca-delete-volume	
	ec2-describe-snapshots	euca-describe-snapshots		euca-describe-snapshots	
	ec2-describe-volumes	euca-describe-volumes		euca-describe-volumes	
	ec2-detach-volume	euca-detach-volume		euca-detach-volume	
Addressing	ec2-allocate-address	euca-allocate-address		euca-allocate-address	
	ec2-associate-address	euca-associate-address		euca-associate-address	
	ec2-describe-addresses	euca-describe-addresses		euca-describe-addresses	
	ec2-disassociate-address	euca-disassociate-address		euca-disassociate-address	
	ec2-release-address	euca-release-address		euca-release-address	

Tabella 2.2: Confronto delle API delle Cloud osservate: Amazon EC2, Eucalyptus, OpenNebula, OpenStack e Nimbus.

Image Tools	ec2-create-image				
	ec2-bundle-image	euca-bundle-image		euca-bundle-image	
	ec2-bundle-vol	euca-bundle-vol		euca-bundle-vol	
	ec2-delete-bundle	euca-delete-bundle		euca-delete-bundle	
	ec2-deregister	euca-deregister		euca-deregister	
	ec2-describe-image-attribute	euca-describe-image-attribute		euca-describe-image-attribute	
	ec2-describe-images	euca-describe-images	describe images	euca-describe-images	ec2-describe-images
	ec2-download-bundle	euca-download-bundle		euca-download-bundle	
	ec2-migrate-bundle				
	ec2-migrate-image				
	ec2-migrate-manifest				
	ec2-modify-image-attribute	euca-modify-image-attribute		euca-modify-image-attribute	
	ec2-register	euca-register	register image	euca-register	
	ec2-reset-image-attribute	euca-reset-image-attribute		euca-reset-image-attribute	
	ec2-unbundle	euca-unbundle		euca-unbundle	
	ec2-upload-bundle	euca-upload-bundle		euca-upload-bundle	
	ec2-upload-disk-image		upload image		
Security Group	ec2-add-group	euca-add-group		euca-add-group	
	ec2-authorize	euca-authorize		euca-authorize	
	ec2-delete-group	euca-delete-group		euca-delete-group	
	ec2-describe-group	euca-describe-groups		euca-describe-groups	
	ec2-revoke	euca-revoke		euca-revoke	
Key Pair	ec2-add-keypair	euca-add-keypair		euca-add-keypair	ec2-add-keypair
	ec2-describe-keypairs	euca-describe-keypairs		euca-describe-keypairs	
	ec2-delete-keypair	euca-delete-keypair		euca-delete-keypair	ec2-delete-keypair
	ec2-import-keypair				
Availability Zone & Regions	ec2-describe-availability-zones	euca-describe-availability-zones		euca-describe-availability-zones	
	ec2-describe-regions	euca-describe-regions		euca-describe-regions	

Tabella 2.3: Confronto delle API delle Cloud osservate (continua).

2.5 Cloud P2P

La peculiarità delle architetture peer-to-peer risiede nel fatto che ciascun componente della rete viene considerato di pari livello rispetto a qualunque altro, con l'immediato effetto che non si debba fissare a priori alcuna gerarchia. Qualora sia poi necessario risolvere un problema di consenso occorre applicare degli opportuni algoritmi, che si ritrovano nella fornita letteratura dei sistemi distribuiti.
Consideriamo inoltre l'avvio del sistema. Nel caso "tradizionale" è necessario avviare i componenti in un determinato ordine ed eseguire diversi passi per la configurazione. Al contrario nello scenario peer-to-peer ogni nodo può essere eseguito indipendentemente dagli altri, in quanto le caratteristiche della rete ed il coordinamento dei servizi avvengono a run-time.
I concetti di Cloud Computing e P2P si trovano in contrapposizione. Per esplicitarne le differenze dobbiamo riflettere sul fatto che: il primo prevede la fornitura di risorse dalle alte garanzie prestazionali, accompagnate da ingenti costi per i provider, che si ripercuotono inevitabilmente in minima parte sull'utente finale. In un sistema P2P, per poter offrire garanzie sulle prestazioni, occorre gestire la presenza di churn [3]. Abbiamo già visto in 1.4 che l'unione dei due si concretizza nel caso del Volunteer Computing, dove le risorse degli elaboratori vengono volontariamente messe a disposizione della rete. Diverse sono le proposte che volgono a beneficiare di entrambi i vantaggi del Cloud Computing e delle architetture P2P, e promuovono una soluzione al problema di coordinamento [19, 57, 47].
Alcune esperienze sono giunte al successo nella realizzazione di una piattaforma Cloud utilizzata in ambito accademico e costituita da risorse volontarie come esposto in [13, 53]. Seguendo la classificazione dei modelli di deployment del capitolo 1.3.2, tali strutture rientrano in quello privato, in quanto ad usufruirne è il personale stesso dell'ente che sostiene il sistema. Questo

[3] Churn: termine che indica l'ingresso e l'abbandono di nodi all'interno di una rete in tempi completamente arbitrari e non prevenibili. Viene misurato come percentuale sulla dimensione totale della rete.

dimostra allora che il connubio tra risorse volontarie ed ambito privato si rivela più facilmente attuabile piuttosto che in quello pubblico.

Capitolo 3

Casi d'uso per sistemi di Cloud Computing

In questo capitolo presenterò alcune osservazioni e riflessioni che mi sono scaturite durante la fase di raccolta delle fonti e nella rassegna dello stato dell'arte. Queste vogliono fornire una visione sul panorama del Cloud Computing in grado di cosiderare diversi punti di vista, in modo da contribuire a rendere il più completa possibile la fase di progettazione.
Comincierò con l'esaminare in 3.1 come diverse classi di utilizzatori, dal singolo utente al caso delle imprese, possono interagire con una Cloud. Proseguirò con il mostrare la ripartizione dei casi di studio dei servizi Amazon in 3.2, avanzando qualche interrogativo ed osservazione. La sezione 3.3 propone una visione architetturale alternativa, nella quale compaiono gli individui che in diversi ruoli agiscono sulla Cloud. Conclude il capitolo l'esposizione degli scenari in 3.4, che si prestano alla realizzazione del seguente progetto.

3.1 Bacino d'utenza

Una volta chiarito cosa si intende per Cloud Computing ed aver osservato alcuni esempi, possiamo ora interrogarci su quale sia il bacino di utenza per questo modello. Abbiamo già accennato precedentemente al fatto che l'insieme degli utilizzatori non è ben definito. In questa sezione cercheremo di delineare alcuni elementi volti ad evidenziare i punti a favore e gli svantaggi che diverse classi di utenti potrebbero sperimentare nell'utilizzo di una Cloud. In particolare i tipi di utenti considerati sono: singoli utenti, piccole, medie e grandi imprese, ed infine organizzazioni pubbliche.
Le considerazioni che seguono sono il frutto delle riflessioni emerse durante la fase di consultazione delle fonti e la ricerca dello stato dell'arte.

Per scoprire quali motivazioni conducano il singolo utente a servirsi di un sitema di Cloud computing, bisogna indagare sui punti chiave che hanno portato al successo i social network, come si coniugano con questo nuovo modello di rete, ed infine cosa può prospettare il futuro di Internet. L'offerta di potenza computazionale infatti non rientra tra i servizi direttamente consumabili da una persona singola, ma può essere impiegata come supporto per altre applicazioni. L'esempio principe è rappresentato dal singolare avvenimento che ha per protagonista Facebook: l'azienda Animoto ha dovuto far fronte ad un aumento della richiesta, passata da 50 a 3500 server in 48 ore, dopo il lancio di una nuovo accessorio [42, 27].

Diverso è invece lo scenario delle imprese, alle quali l'intera gamma di sevizi Cloud sembra essere rivolta. Caratteristiche quali virtualizzazione, scalabilità on-demand e pay-as-you-go, costituiscono validi punti di interesse per le aziende, ed in particolare perchè si traducono in un risparmio economico alle volte considerevole, se rapportato ad una soluzione interna alla ditta e costruita interamente da zero.
Tra gli esempi più ecclatanti possiamo citare da [27] quello del New York Times, che ha visto la conversione, in formato PDF, di un vasto insieme di

immagini ad alta risoluzione dei quotidiani appartenenti agli anni 1851-1922. Grazie all'utilizzo di 100 istanze di EC2 il lavoro è stato compiuto in 24h con una spesa di $890. Una tale cifra non sarebbe nemmeno bastata per l'acquisto di un solo server, alla quale si sarebbero dovuti aggiungere i costi di manutenzione e raffreddamento, ed avrebbe impiegato circa tre mesi per ultimare la computazione.
Altrettanto sorprendentemente, un esperimento condotto dal DISA (Defence Information System Agency) ha mostrato come per un'implementazione del progetto "Tech Early Bird" interna all'azienda quindi con server acquistati appositamente, la spesa stimata si aggirasse intorno ai $30.000. Sarebbe potuta arrivare a costarne solo $5 se si fosse considerata una soluzione dalle stesse prestazioni facente uso dei servizi web di Amazon.
Cominciamo con l'esaminare il modello di Cloud pubblico, per il quale è interessante introdurre il caso studiato in [38] in un'industria di estrazione di petrolio e gas. Il documento illustra proprio l'analisi di un'eventuale migrazione del sistema informatico già in uso verso EC2. Senza scendere nei dettagli dell'articolo, ne riassumo il contesto, che coinvolge tre aziende tenute anonime. La prima, denominata A, si occupa dell'estrazione ed opera direttamente sul campo di lavoro, la seconda B, mantiene i server destinati alle comunicazioni, i dati pervenuti da A e tutto ciò che riguarda il backup di questi nonché le chiamate di assistenza di utenti esterni ed i rapporti dei tecnici interni. La terza infine (C), rappresenta l'ente che usufruisce di A attraverso i servizi offerti da B. Lo studio degli autori propone a B l'utilizzo della Cloud di Amazon allo scopo di: evitare le spese di manutenzione, rendere i servizi il più continuativi possibili, ed eventualmente essere in grado di offrire maggiore scalabilità. Sebbene questo sia un esempio piuttosto specifico, la situazione nella quale un'impresa si serve di altre per compiere il proprio lavoro, non è poi così rara. Dal punto di vista finanziario, la scelta della migrazione verso il sistema Cloud sembra essere la più appropriata, in quanto avrebbe portato ad un risparmio del 37% negli ultimi 5 anni di attività, ed una riduzione delle chiamate di assistenza del 21%. Tuttavia

le conclusioni dell'articolo illustrano come i benefici non siano sufficienti a motivare una completa migrazione per via di altri aspetti. Primo fra tutti il rischio di diventare dipendenti di un'infrastruttura non più appartenente all'azienda ma gestita da terze parti, per la quale il personale tecnico, seppur specializzato, può agire solo in maniera remota anche in caso di emergenze. L'intero sistema informatico poi andrebbe rivoluzionato ed adeguato non solo all'ambiente distribuito ma anche alle ben più frequenti comunicazioni di rete. Bisogna poi preventivare i costi addizionali introdotti dalla fase di avviamento, che necessita di un periodo di prova della nuova configurazione mantenendo interamente il vecchio sistema, per evitare così l'interruzione della produzione. Gli stessi autori ammetto poi che i prezzi esposti non tengono conto dei servizi che dovrebbero essere adottati per la tolleranza ai guasti. In questo caso occorrerebbe uno studio approfondito del problema, che dovrebbe tenere conto del livello di affidabilità che si vuole raggiungere e dedicarsi alla ricerca di tutte le possibilità atte far fronte agli immancabili problemi.

Questo esempio ci è utile per riflettere anche sulla tipologia di applicazione presa in analisi: si colloca tra le attività critiche dell'azienda (business-critical). In quest'ottica si comprende come gli aspetti economici passino in secondo piano rispetto alla sicurezza delle informazioni vitali per l'azienda, e la possibilità di un intervento manuale del personale addetto.

A complicare le cose solleviamo ora anche il seguente interrogativo: dal momento che più di un'agenzia si occupa della fornitura di un unico servizio, nella fattispecie B in collaborazione con un provider, in caso di guasti tanto estesi da arrecare danno agli utilizzatori, quale ente si prende carico della responsabilità dell'accaduto? Dobbiamo infatti pensare che se anche la sopra citata agenzia B si affidasse ad un sistema pubblico di Cloud Computing per il mantenimento dell'infrastruttura, rimarrebbe comunque responsabile dei dati che promette di gestire, e dovrebbe far fronte al malcontento e alle accuse dei clienti in caso di malfunzionamento. Il 21 Aprile 2011 si è verificata proprio una situazione di questo tipo [50], quando alcuni siti, affidatisi ai ser-

vizi di Amazon, hanno assistito al crollo delle poprie applicazioni in seguito ad un problema del provider. Immediatamente sono emerse contestazioni sul mancato adempimento dei contratti SLA, che come abbiamo visto in precedenza costiutiscono uno degli aspetti cruciali nello sviluppo di una Cloud. Bisogna però anche indagare se davvero gli sviluppatori di queste applicazioni si sono avvalsi di tutte le tecniche, più volte ribadite nelle guide allo sviluppo di Amazon, per prevenire la perdita di dati importati.
Possiamo concludere che non esiste al momento una strategia valida per ogni tipo di azienda, ma ciascuna deve ponderare attentamente la propria situazione ed i rischi che derivano dall'impiego di risorse di terze parti. Sicuramente l'utilizzo delle Cloud pubbliche è scoraggiato per attività critiche dell'azienda, e per quelle imprese in fase di avviamento, in quanto alcuni rischi sono difficilente quantificabili dalle sole stime preventive.

Ancora diverso è il caso di organizzazioni dalle dimensioni più ampie, che magari dispongono già di grossi cluster capaci di vantare prestazioni eccezzionali di avviabilità per la fornitura dei propri prodotti. In questo scenario come è presentato da [**?**], l'azienda ha ancora maggiore interesse a non esportare parte del proprio operato verso una Cloud pubblica, in quanto oltre ad esporsi ai rischi di un ambiente frequentato da numerosi utenti, potrebbe addirittura soffrire di un calo prestazionale dei propri siti con conseguente danno all'immagine. In situazioni di questo tipo [14] e per realtà di piccole dimensioni, Cisco propone soluzioni che prevedono il modello privato con eventuale evoluzione a quello ibrido [15]. Sebbene i motivi che conducono alla costruzione di una Cloud privata siano specifici per ogni singola impresa, è possibile individuarne alcuni che possono formare un denominatore comune. Tra questi compare il fatto che le imprese tendono a mantenere i dati cruciali all'interno dei propri sistemi. Vi possiamo probabilmente aggiungere anche il voler garantire l'attuale livello produttivo in minor tempo e con minor spesa. Questo può tradursi nello sviluppare le potenzialità del reparto informatico, rendendolo più versatile ai cambiamenti futuri e minimizzarne gli sprechi.

Quest'ultima visione può essere comune anche agli enti pubblici ed Università, per i quali il volunteer computing rappresenta un'opportunità per la condivisione delle risorse informatiche, come abbiamo visto in 1.4.

3.2 Utenti dei servizi Amazon

Può essere interessante ora chiedersi quali aziende hanno scelto i servizi di Amazon. L'azienda ha dedicato una pagina web chiamata "Case Studies" [1] dove sono raccolte le imprese ed organizzazioni che usufruiscono dei suoi prodotti e servizi. Amazon suddivide i propri clienti in categorie, che ho raccolto in tabella 3.1 insieme al numero totale delle aziende che compaiono per ciascuna di essa. Ho condotto due osservazioni a distanza di poco più di tre mesi l'una dall'altra, e più precisamente: il 9 Febbraio 2011 ed il 19 Maggio 2011. Nella terza colonna della tabella 3.1 ho aggiunto la percentuale di crescita derivante dalle due osservazioni.
Al momento non si avanzano conclusioni riguardo ai dati raccolti. Probabilmente è necessaria un'indagine più approfondita e prolungata nel tempo per poter formulare constatazioni accurate. Ci soffermeremo tuttavia su alcune domande ed osservazioni che derivano dalla lettura della tabella.

Salta immediatamente all'occhio la cifra della voce "Application Hosting" che si discosta dalle altre con un divario di quasi un ordine di grandezza. A cosa è dovuta questa differenza? Possiamo assumere che si tratti di una categoria tanto generica da raggruppare l'operato di un così alto numero di aziende, o altri sono i motivi che portano a tali cifre?
Scorrendo poi gli altri settori, come si giustifica la crescita nulla di "Backup and Storage", "Content Delivery", "E-Commerce" e "Search Engines"? Potremmo avanzare l'ipotesi che: per i primi due la difficoltà di scalabilità della memoria di massa influisca pesantemente sui costi, tanto da forzare il provider a fissare prezzi ancora non sufficientemente vantaggiosi da attirare nuovi clienti. Per quanto riguarda applicazioni E-Commerce e motori di ricerca,

[1] `http://aws.amazon.com/solutions/case-studies/`

Case Study	09 Feb 2011	19 May 2011	Crescita Percentuale
Application Hosting	92	126	31.28
Backup and Storage	14	14	0
Content Delivery	6	6	0
E-Commerce	4	4	0
High Performance Computing	15	18	0.45
Media Hosting	19	20	0.19
On-Demand Workforce	6	7	0.06
Search Engines	4	4	0
Web Hosting	12	16	0.48
Total	**172**	**215**	

Tabella 3.1: Lista dei Casi di Studio di Amazon raccolta nelle due osservazioni del 9 Febbraio e 19 Maggio 2011.

possiamo supporre che: gli elevati valori di affidabilià, richiesti per un'adeguato servizio, non siano ancora pienamente raggiungibili dalla Cloud, e le ditte del settore preferiscano basarsi sui propri mezzi. Qualora questa supposizione si dimostrasse vera, andrebbe allora a confermare quanto esposto in precedenza riguardo alle difficoltà nel garantire un alto livello di availability, che portano spesso le grandi imprese a scegliere altri tipi di investimenti, piuttosto che avvalersi del Cloud computing.
Lasciamo tuttavia spazio anche ad altre motivazioni puramente strutturali. Il livello infrastruttura potrebbe semplicemente essere inadeguato per lo sviluppo di determinate applicazioni, che meglio si potrebbero costruire avendo a disposizione una piattaforma o servizi di più alto livello. La soluzione di Amazon tuttavia sembra interessare diverse Università che risiedono in particolare nella classe HPC (High Performance Computing). Riporto un dato che

non compare in tabella, ovvero che: 2 delle 3 nuove aggiunte, nella categoria HPC, si riferiscono proprio ad ambienti accademici.

3.3 Definizione dei profili degli utenti

Se da un lato abbiamo analizzato quali applicazioni possono avere successo o meno realizzandole in un ambiente Cloud, dall'altro resta ora da delineare il profilo degli individui che sono coinvolti nell'interazione con il sistema. A tal proposito, credo sia importante conoscere quali competenze sono necessarie alla persona che intende usufruire della Cloud. Una volta quindi identificati gli *attori principali*, se ne può trarre beneficio nei casi d'uso nella successiva fase di progettazione.

Cominciamo col prendere in considerazione la divisione tra *Front-End* e *Back-End*, concetti non ancora riferiti al Cloud computing in letteratura, ma che oltre a comparire in alcuni elaborati e progetti, ritengo opportuni per indagare i ruoli degli individui che agiranno sul sistema.
Con il termine Back-End consideriamo tutti i componenti che si pongono al supporto dei servizi esposti dalla Cloud. Come abbiamo visto, l'infrastruttura (IaaS) costituisce il primo livello di interazione con l'utente, ed è composta da un insieme di API che andrà a definire l'interfaccia di più basso livello. La parte di Front-End riguarda invece l'insieme delle persone che, avvalendosi delle risorse della Cloud e diverse strategie, realizzano un'applicazione.

Da quella che è la mia attuale conoscenza, un utilizzatore di un qualunque servizio Cloud potrebbe essere identificato attraverso `User_X`. Esso non corrisponde ad una particolare figura professionale, ma necessita di prerequisiti tutt'altro che banali: oltre ad avere solide competenze informatiche e conoscere il dominio al quale l'applicazione è rivolta, deve anche possedere una buona esperienza nel campo delle architetture software, per poter far fronte alle complessità derivanti dalla gestione di componenti distribuiti.

Vogliamo però arrivare a semplificare l'utilizzo del nostro sistema in modo da renderlo accessibile anche a persone non esperte nel campo delle architetture distribuite. A tal proposito suggeriamo la definizione di un confine tra il livello infrastruttura e quello piattaforma. Dividiamo dunque in due insiemi tutte le operazioni che sono necessarie a `User_X` per gestire la Cloud ed implementare un'applicazione. Da un lato raccogliamo le funzioni che provvedono al mantenimento delle macchine. Queste comprendono ad esempio l'allocazione ed il rilascio delle istanze, o l'amministrazione della memoria di massa. Dall'altro posizioniamo invece quei compiti di più alto livello, per i quali la modifica delle risorse fisiche deve risultare completamente trasparente. Un esempio può essere rappresentato dalla sottomissione alla Cloud di un lavoro complesso che coinvolge più di un calcolatore. L'utente dovrà essere sollevato dal coordinamento delle macchine per inserire solo alcuni parametri essenziali, quali ad esempio il limite massimo di risorse da occupare oppure il tempo a disposizione per completarlo. In questo modo possiamo scomporre le operazioni di `User_X` e introdurre due personaggi più specifici che chiamiamo `Philip` e `Adriane`. `Philip` dovrà essere in grado di sfruttare l'infrastruttura per costruire un insieme di funzioni più articolato che costituirà un livello addizionale meglio conosciuto come "piattaforma" della Cloud (PaaS). `Adriane` sarà così sollevata dalla gestione delle macchine, per potersi concentrare esclusivamente allo sviluppo del programma applicativo. Una figura che spesso viene tralasciata è quella rappresentata da `Roger` che ricopre il ruolo di manutentore: deve occuparsi del monitoraggio delle risorse e della risoluzione sia di eventuali guasti fisici che di problemi software. Possiamo identificarlo come ingegnere informatico, capace di operare nel back-end, comprendere l'architettura del sistema ed eventualmente modificarla.

3.4 Scenari presi in considerazione

Questa sezione introduce l'ambito all'interno del quale costruiremo il prototipo del sistema. Il progetto ha l'obiettivo di risolvere le situazioni rappre-

sentate da due scenari.

Il primo considera il caso di un'impresa che vuole sfruttare le risorse computazionali già in suo possesso, per adempiere a compiti secondari ma che non possono essere eseguiti durante le ore di lavoro giornaliere, pur sempre evitando investimenti rilevanti dal punto di vista economico. Un esempio inerente a questo contesto potrebbe essere l'analisi dei file di log scritti dai firewall aziendali. Una volta avviato il processo a fine giornata, questo deve raccogliere i dati memorizzati in ogni calcolatore della rete interna, elaborarli e produrre poi il risultato. Si potrebbe ottenere un sensibile incremento delle prestazioni se ciascuna macchina elaborasse prima la propria porzione di dati, poi, attraverso la condivisione con le altre, si giungesse ad un risultato globale.

Il secondo si colloca in ambito accademico, ed anche in questo caso si vogliono sfruttare le risorse computazionali inoperose, come ad esempio il laboratorio didattico del Dipartimento di Informatica dell'Università di Bologna. L'idea prevede la costruzione di un'infrastruttura Cloud, in modo da tenere traccia delle risorse occupate in ciascuna macchina e riuscire a distribuire i nuovi compiti, sottomessi da diversi utenti, negli elaboratori più liberi.

Dagli scenari si evince dunque che adotteremo un modello privato. Questo perché è esente dai costi iniziali ai quali invece un provider pubblico deve far fronte. Inoltre non sono previsti sofisticati contratti con clienti che impongono oneri e garanzie sulla fornitura del servizio. Una Cloud privata darà inoltre la possibilità di valutare, oltre all'aspetto progettuale ed implementativo, anche la soddisfazione reale e con immediata risposta dell'utente, in quanto gli stessi utilizzatori del sistema rientreranno nel personale interno all'organizzazione e quindi di facile coinvolgimento.

L'ambiente nel quale il sistema dovrà svilupparsi non gode di un alto livello di affidabilità. Abbiamo dunque pensato di dirigere la progettazione verso un'architettura peer-to-peer che, oltre a rappresentare una sfida implementativa ed uno spunto scientifico, porta alcuni benefici pratici. In primo luogo l'assenza di alcuni componenti che si prendano a carico la gestione, evita

sia un futuro collo di bottiglia che un punto fragile di fallimento. Un altro vantaggio risiede nel non dover definire tra le macchine una gerarchia fissata a priori. Essa oltre a richiedere uno sforzo particolare per il mantenimento, deve essere garantita anche in condizioni anomale della rete (ad esempio con pochi elaboratori a disposizione). Nello scenario a nodi paritetici invece i problemi di coordinazione vengono risolti a run-time, e la rete viene adeguata di volta in volta in base alla disponibilità dei calcolatori attivi.

La nostra proposta si contraddistigue dalle soluzioni P2P osservate in quanto costruita attraverso *protocolli di gossip*, che mantengono le comunicazioni tra i nodi, e provvedono alla creazione della rete. Questa strategia prevede che ciascun nodo possieda una *vista parziale* contenente i collegamenti ai nodi vicini, con i quali scambierà informazioni che continuamente modificano la vista. In questo modo si viene a creare una rete che assume la forma di un grafo le quali caratteristiche variano dipendentemente dal tipo di algoritmo e dai suoi parametri. Essa gode inoltre di altre importanti proprietà come self-configuration, sefl-management, illustrate in [10], tra le quali compare quella di self-healing: ovvero la capacità far fronte in modo autonomo ai guasti senza alcun intervento umano o di gestione straordinaria, ma semplicemente continuando ad eseguire l'algoritmo di gossip che già viene impiegato.

Capitolo 4

Progettazione di "P2P Cloud System"

In questo capitolo presento la fase di progettazione del sistema P2P Cloud System (P2PCS) del quale andremo a costruire un prototipo. Seguiremo la proposta già avanzata da [45] nella quale l'insieme delle macchine, attraverso protolli di gossip, forma una rete P2P. In seguito alla sottomissione di una richiesta da parte dell'utente, alcuni nodi possono organizzarsi in una "*sotto-nuvola*"(subcloud) e formare così una nuova rete che si aggiunge ad un livello superiore a quella già esistente e che identifichiamo come *overlay*.
Introduciamo la fase di raccolta dei requisiti in 4.1, segue poi l'analisi dei casi d'uso della Cloud in 4.2. In 4.3 osserviamo la struttura interna di un nodo che comprende i moduli di *servizio*, di *sistema* e di *interfaccia* disposti a livelli. Termina il capitolo la fase di deployment 4.4, che sfrutteremo come tabella di marcia per l'implementazione.

4.1 Raccolta dei Requisiti

Per la raccolta dei requisiti mi sono servito dell'analisi dei casi di studio di Amazon e di Rackspace. La sezione relativa agli attori delle Cloud pubbliche [5], offre un ottimo spunto per la comprensione delle figure coinvolte nell'interazione tra gli utenti ed il sistema.
Seguiamo l'idea presentata precedentemente in 3.3, nella quale abbiamo ricavato i tre personaggi: `Roger`, `Philip` e `Adriane`. Questi sono utili per mantenere distinti i ruoli delle persone che operano sul sistema.
Nel Back-End, Roger identifica il manutentore ed eseguirà azioni di monitoraggio, correzioni di problemi software, sostituzione di hadware difettoso, fornirà assistenza a quanti la richiedono e provvederà al corretto funzionamento, garantendo politiche eque nella fornitura delle risorse.
Dal lato Front-End gli utilizzatori della Cloud vengono a loro volta suddivisi in due. Philip rappresenta l'insieme delle persone che, possedendo forti conoscenze nel campo delle architetture distribuite, usufruiscono dell'infrastruttura ed intendono agire in maniera approfondita sui dettagli delle macchine. Adriane include invece coloro che, pur non avendo le competenze di Philip, desiderano accedere alla Cloud per sottomettervi lavori che richiedono un numero elevato di calcolatori.
Tengo a sottolineare che, nonostante abbia usato il termine “personaggio” per le tre figure descritte in precedenza, queste non corrispondono al concetto previsto nell'analisi di Goal-Directed Design (GDD) [8]. Servono semplicemente ad evidenziare come, a mio avviso, sia necessario un approccio capace di considerare le esigenze delle persone che lavoreranno nell'ambito di P2PCS già durante la fase progettuale. L'applicazione di GDD è stata presa in considerazione, ma al momento viene posticipata per via della mancanza di individui da intervistare, i quali costituiscono una fonte insostituibile per la buona riuscita del metodo. In tabella 4.1 ho riassunto gli attori principali.
In questa prima stesura del progetto ci concentreremo in particolare sulla distinzione tra la parte Back-End e Front-End. Con lo scopo di semplificare la documentazione, indichiamo con Cloud User (CU) l'insieme degli utiliz-

zatori della Cloud. Differenzieremo le due sottoclassi, impersonate da Philip ed Adriane, solo in un secondo momento, quando cioè dovremo stabilire il confine tra le API dell'infrastruttura e quelle di piattaforma. Per poter accedere ai servizi del sistema è necessario essere identificati in modo da rendere tracciabili le risorse che ciascun utente richiede. Indichiamo con Unidentified User (UU) le persone che non hanno ancora eseguito la procedura di autenticazione per accedere ad sistema.
Dal lato Back-End introduciamo una semplice gerarchia nel gruppo di persone che si occupa della manutenzione. Possiamo supporre che ciascun componente abbia le competenze per poter lavorare all'interno del progetto e denominiamo questo ruolo Maintainer Engineer (ME). Tra questi possiamo supporre che spetti ad un incaricato il compito di coordinare, ed eventualmente controllare l'operato dei colleghi e definiamo questa figura come Maintainer Chief Administrator (CA).
I tre attori restanti rappresentano elementi essenziali del sistema, il primo, Dispatcher System (DY), gestisce e smista le richieste sottomesse dagli utenti, il secondo, Monitoring System (MY), è dedicato al monitoraggio delle risorse, ed infine il terzo, Storage System (SY) si occupa di tutto ciò che concerne la memoria di massa.

In tabella 4.2 ho raccolto i requisiti di sistema, suddivisi in tre parti. La colonna a sfondo grigio contiene l'indice globale di ciacuno di questi, mentre per ogni personaggio, oltre alla numerazione relativa, vi è aggiunta anche l'iniziale in modo da poter ricondurre sempre il requisito alla persona a cui appartiene. Questo sistema può risultare utile nella fase implementativa per rintracciare agevolmente la funzionalità dai documenti progettuali fino all'interno del codice. Perché adottare una simile divisione? Si tenga presente che i tre personaggi hanno obiettivi ed aspettative ben diverse, che possono portare ad implementazioni completamente differenti di uno stesso requisito. Così facendo voglio suggerire un modo per raggruppare quelle funzionalità che appartengono alla stessa categoria, come ad esempio "il monitoraggio", ed allo stesso tempo conservare la consapevolezza che ciascuno deve esse-

Stakeholders and Actors Summary	
Back-End	**Front-end**
• Cloud Maintaier Staff – Cloud Chief Administrator (CA), – Cloud Maintainer Engineer (ME) (`Roger`), • Monitoring system (MY), • Dispatcher system (DY), • Storage System (SY).	• Unidentified User (UU) • Cloud User (CU): – university researcher or student (`Adriane`), – enterprise IT architect (`Philip`).

Tabella 4.1: Tabella riassuntiva degli attori del sistema.

re modellato sulle esigenze del personaggio a cui è rivolto. Prendiamo ad esempio gli obiettivi che riguardano il monitoraggio dello stato dell'account e della memoria di massa, rispettivamente il numero 25 e 27 della numerazione globale. Le informazioni che devono essere restituite in output per ciascun personaggio sono diverse, così come il modo di presentarle. Per Roger possiamo aspettarci che il monitorare la memoria di massa significhi poter stimare la quantità libera rispetto a quella utilizzata dagli utenti. Contrariamente Philip ed Adriane saranno interessati al proprio stato di risorse occupate, ed eventualmente avere a disposizione un collegamento che ne mostri il costo. Per quanto riguarda il "prezzo" delle risorse, nel nostro scenario del laboratorio accademico, supponiamo che ogni utente abbia a disposizione delle "quote", calcolate come ore/macchina e memoria utilizzati.
L'alternativa di estrapolare requisiti con descrizioni completamente diverse senza raggruppamenti, forse da un lato parrebbe più chiara a prima vista, ma arriverebbe a moltiplicarne il numero, inserendo poi il rischio di confusione.

Il corretto funzionamento della Cloud è affidato al manutentore che deve

poter sorvegliarne lo stato ed eventualmente agire in prima persona, avendo anche la possibilità di generare dei documenti di rapporto, come riassunto dai requisiti 11, 12 e 21. Sebbene operante nel Back-End del sistema, questo ruolo non preclude le interazioni con gli utenti, si veda ad esempio il requisito numero 23. In questo lavoro non ci preoccupiamo di definire una soglia netta tra l'utilizzo della Cloud di Philip ed Adriane, tuttavia alcuni requisiti tendono a caratterizzarne le divergenze. Possiamo supporre che Adriane sia interessata a sottomettere alla Cloud un tipo di compito "batch", per il quale cioè sia necessario semplicemente specificare il tipo di lavoro, eventualmente fornendo del codice da eseguire, e alcuni parametri che descrivino cosa fare in caso di anomalie durante l'esecuzione. Un tale approccio richiede l'intervento dell'utente solo nelle fasi iniziali di avvio e conclusive per l'estrazione dei risultati, come pensato per i requisiti 9, 10, 14, 22, e 35.
Il termine di "istanza" ha lo stesso significato già incontrato nella rassegna delle Cloud esistenti svolta nel capitolo 2: corrisponde ad un elaboratore nel quale viene avviata una macchina virtuale attraverso un file d'immagine.

Tabella 4.2: La tabella raccoglie i requisiti di sistema suddivisi nei tre personaggi di `Roger`, `Philip` ed `Adriane`. La colonna a sfondo grigio rappresenta la numerazione globale, mentre per ciascuna delle tre parti è presente un indice relativo al quale è aggiunta l’iniziale del personaggio a cui si riferisce.

ROGER			PHILIP			ADRIANE		
1	R1	Accedere e gestire un nodo						
2	R2	Avere la visione completa della Cloud a run-time						
3	R3	Avere il controllo completo di una macchina	3	P1	Avere il controllo completo di una macchina			
			4	P2	Usare la Cloud come supporto di host per un'applicazione	4	A1	Usare la Cloud come supporto di host per un'applicazione
			5	P3	Dare all'applicazione costruita l'accesso web	5	A2	Dare all'applicazione costruita l'accesso web
6	R4	Usare immagini virtuali per avviare istanze	6	P4	Usare immagini virtuali per avviare istanze			
7	R5	Accedere e gestire un'istanza	7	P5	Accedere e gestire un'istanza			
			8	P6	Costruire un livello piattaforma con nuove funzionalità per la Cloud			
						9	A3	Utilizzare un insieme di istanze per avviare un processo “batch-job”
						10	A4	Utilizzare l'approccio di “Map-Reduce” e poter distribuire il carico di lavoro
11	R6	Comprendere un guasto attraverso l'analisi dei file di log						
12	R7	Correggere problemi e riparare i nodi guasti						
13	R8	Essere avvisato e riportare ogni problema	13	P7	Essere avvisato e riportare ogni problema			
						14	A5	Non essere avvisato per ogni problema. Devono invece essere risolti seguendo le direttive impostate a inizio lavoro
			15	P8	Essere in grado di installare librerie e strumenti per sviluppare applicazioni	15	A6	Essere in grado di installare librerie e strumenti per sviluppare applicazioni
16	R9	Essere in grado di avviare una procedura di emergenza per garantire un funzionamento base della Cloud						

17	R10	Scoprire un comportamento malevolo attraverso i file di log						
18	R11	Eseguire veloci verifiche durante le procedure di manutenzione standard o durante l'esecuzione	18	P9	Eseguire veloci verifiche durante le procedure di manutenzione standard o durante l'esecuzione	18	A7	Eseguire veloci verifiche durante le procedure di manutenzione standard o durante l'esecuzione
19	R12	Costruire una Cloud ed inserirvi una nuova macchina						
20	R13	Essere avvisato della terminazione o guasto di un nodo	20	P10	Essere avvisato della terminazione o guasto di un nodo			
21	R14	Essere in grado di identificare la posizione fisica di una macchina guasta						
						22	A8	Essere avvisato nel caso la computazione debba essere terminata
23	R15	Dare supporto ad un CloudUser						
24	R16	Impostare alcune strategie per far fronte a situazioni problematiche						
25	R17	Monitorare lo stato della memoria di massa	25	P11	Monitorare lo stato della memoria di massa	25	A9	Monitorare lo stato della memoria di massa
			26	P12	Monitorare l'attuale costo delle risorse occupate	26	A10	Monitorare l'attuale costo delle risorse occupate
27	R18	Monitorare lo stato dell'account	27	P13	Monitorare lo stato dell'account	27	A11	Monitorare lo stato dell'account
			28	P14	Allocare/Deallocare la memoria di massa	28	A12	Allocare/Deallocare la memoria di massa
29	R19	Utilizzare delle politiche che garantiscano un uso equo della Cloud tra gli utenti						
30	R20	Mantenere la memoria di massa						
			31	P15	Specificare politiche di sicurezza	31	A13	Specificare politiche di sicurezza
32	R21	Garantire la sicurezza tra i nodi						
						33	A14	Aggiungere a tempo di esecuzione un istanza ad una subcloud
			34	P16	Essere in grado di impostare contratti SLA e politiche QoS	34	A15	Essere in grado di impostare contratti SLA e politiche QoS
						35	A16	Sfruttare politiche di rilascio anticipato delle risorse durante un lavoro batch in modo da risparmiare risorse

4.2 Casi d’uso

Dopo aver analizzato i requisiti di sistema, approfondiamo ora il modello attraverso i casi d’uso, che ho suddiviso in tre parti. La prima è rappresentata dalla figura 4.1 e mostra l’interazione tra il sistema ed il Cloud User (CU). Segue poi l’immagine 4.2 che delinea il lavoro del manutentore (Maintainer Engineer (ME)) nel Back-End. All’interno del sistema compaiono invece alcuni componenti, ai quali spettano i compiti di coordinamento dei nodi. Come spiegato nel libro [17], i diagrammi presentati derivano dalla “lista dei compiti” degli attori, nella quale vengono elencati per ciascuno le attività di più alto livello e fissati i punti principali per la loro esecuzione.

Nel caso d’uso 4.1 ho inserito l’attore Unidentified User (UU) che deve potersi autenticare per usufruire del sistema, o creare un nuovo account nel caso non l’abbia ancora. Abbiamo visto come il Cloud User (CU) rappresenti l’insieme delle persone che già possiedono un account e sono autorizzate a sfruttare i servizi offerti. Può interrogare la Cloud con operazioni di monitoraggio, gestire le immagini virtuali e la memoria di massa, che verrà suddivisa in volumi. Per completezza della documentazione, e dal momento che è ancora prematuro voler differenziare i ruoli di Philip ed Adriane, ho inserito in questo diagramma due azioni caratteristiche dei due personaggi. Infatti il compito di gestire le istanze tramite l’accesso ssh o web è rivolto in particolare al ruolo di Philip, mentre possiamo pensare che Adriane sia interessata all’avvio di un lavoro “batch”. Possiamo inoltre supporre il problema della sicurezza essere comune ad entrambi. Al Maintainer Engineer (ME) di figura 4.2 sono affidati quei compiti che assicurano il corretto funzionamento del sistema come la sua costruzione attraverso l’aggiunta di nuove macchine, il suo monitoraggio e la riaparazione dei nodi difettosi. L’analisi dei log ha il duplice scopo di servire sia a controllare a posteriori lo svolgimento delle richieste sottomesse, che per rivelare il comportamento di azioni dannose, esercitate sui nodi, volontarie o meno. La stesura di un report avviene dopo un intervento del ME e può essere usata come metodo per tenere traccia sia dei problemi più frequenti che per formare una collezione dei metodi risolu-

Figura 4.1: Diagramma del caso d'uso dell'utente CU.

tivi. Bisogna poi garantire un utilizzo equo della Cloud per ogni utente che ne fa richiesta attuando opportune politiche di collocamento delle risorse.
All'interno della Cloud i moduli di Dispatcher System (DY), Monitoring System (MY) e Storage System (SY), svolgono il ruolo di coordinatori. Una volta sottomessa una richiesta, questa viene analizzata dal Dispatcher System che dovrà selezionare i nodi adeguati ed avviarne l'esecuzione. Dobbiamo organizzare il DY in modo da far fronte a situazioni particolarmente critiche attraverso strategie di emergenza che riportino il sistema ad uno stato operativo. Un esempio è costituito dalla diminuzione drastica delle risorse per via di numerosi guasti: in questo caso la rete deve essere ricostruita,

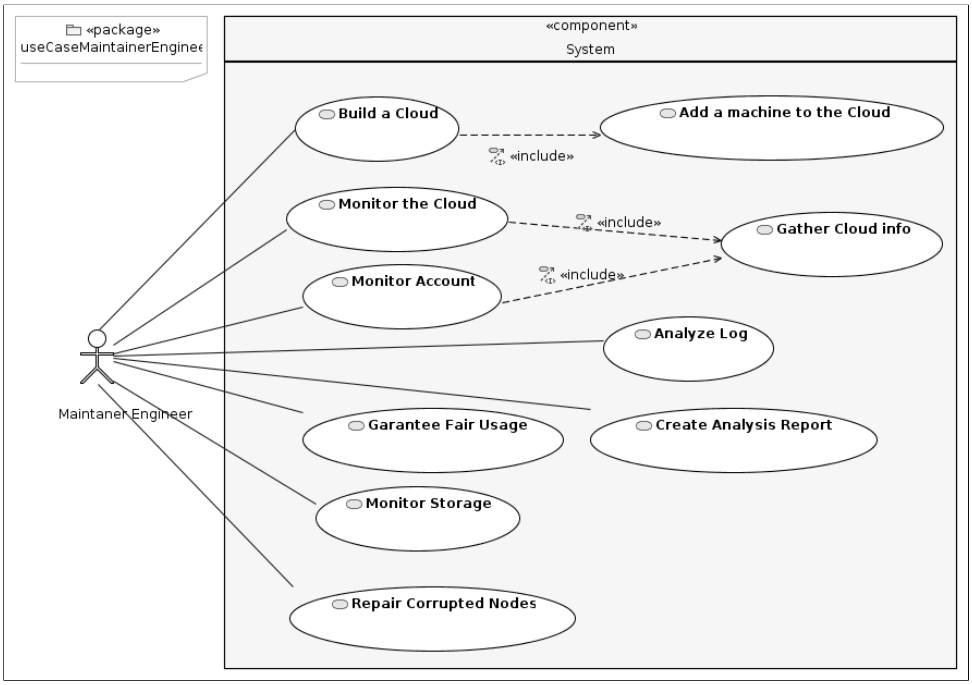

Figura 4.2: Diagramma del caso d’uso del manutentore ME.

per recuperare le connessioni tra i nodi, inoltre tutti i moduli di servizio e sistema devono aggiornare le strutture dati interne per avere una visione consistente della struttura della Cloud. Per finire le “subcloud” che necessitano di reperibilità, anche nel caso di abbandono di alcuni appartenenti all’overlay, devono poter continuare il proprio lavoro reclutando nuove macchine, o eventualmente informare l’utente che lo ha sottomesso dell’impossibilità di completare l’attività.
Il Monitoring System deve raccogliere informazioni sul sistema che verranno poi rielaborate e presentate a chi ne fa richiesta in modo da fornire una panoramica completa della Cloud. Allo Storage System viene affidata la gestione dei volumi della memoria di massa ed il monitoraggio di questa.

Volendoci attenere all’iter proposto in [17], segue poi la stesura delle tabelle dei casi d’uso che illustrano i passi che compongono lo scenario principale, quelli di estensione, e di fallimento. Di seguito ne mostro tre esempi

che descrivono due funzioni basilari del nostro prototipo cioè la costruzione della Cloud, con l'aggiunta di nuove macchine, e la gestione delle istanze da parte dell'utente.

<table>
<tr><td>CASO D'USO #1</td><td>Costruire una Cloud</td></tr>
<tr><td>Contesto d' Uso</td><td>Un compito particolare del MaintainerEngineer è quello di costruire una Cloud da un insieme di macchine appartenenti alla stessa rete. Il primo passo è quello di installare il software necessario in ogni macchina, quindi di inserirvi il pacchetto NodePackage. Il NodePackage contiene tutte le librerie e le dipendenze esplicite per l'installazione del nodo nell'infrastruttura.</td></tr>
<tr><td>Ambito</td><td>SO della macchina, P2PCloudSystem</td></tr>
<tr><td>Livello</td><td>Summary Goal</td></tr>
<tr><td>Attore Primario</td><td>MaintainerEngineer</td></tr>
<tr><td>Stakeholder e Interessi</td><td><table><tr><th>Stakeholder</th><th>Interesse</th></tr><tr><td>MaintainerEngineer</td><td>Introdurre ciascuna macchina all'interno della Cloud</td></tr></table></td></tr>
<tr><td>Precondizioni</td><td>Non vi sono errori nel Sistema Operativo della macchina.</td></tr>
<tr><td>Garanzie Minimali</td><td>Non tutte le macchine sono connesse, ma la Cloud rimane funzionante con gli elaboratori che hanno compiuto correttamente l'installazione.</td></tr>
<tr><td>Garanzie di successo</td><td>Tutte le macchine desiderate sono correttamente connesse nella Cloud e sono in grado di operare come Nodi.</td></tr>
<tr><td>Attivatore</td><td>Il MaintainerEngineer deve avviare il sistema P2PCloudSystem.</td></tr>
</table>

Descrizione	1. Aggiungere una nuova macchina alla Cloud(Caso d'Uso #2) 2. ripeti 1 per ciascuna macchina.
Estensioni	
Variazioni Tecnologiche e sui Dati	

CASO D'USO #2	**Aggiungere una nuova macchina alla Cloud**
Contesto d'Uso	Con lo scopo di avviare *P2PCloudSystem* il *MaintainerEngineer* deve eseguire lo script di avvio presente in *NodePackage* in ogni macchina che intende inserire. Questo caso d'uso è specifico di una macchina, che viene chiamata *TargetMachine* in questa tabella. Il *MaintainerEngineer* deve assicurarsi che: il SO sia correttamente in esecuzione, tutte le dipendenze software siano state soddisfatte, quindi posizionare *NodePackage* nella directory di lavoro.
Ambito	P2PCloudSystem
Livello	User Goal
Attore Primario	MaintainerEngineer
Stakeholder and Interesssi	Stakeholder: MaintainerEngineer — Interesse: Fare lavorare il calcolatore *TargetMachine* all'interno della Cloud

Precondizioni	La *TargetMachine* soddisfa tutti i requisiti software ed il sistema operativo è correttamente in esecuzione.
Garanzie Minimali	La *TargetMachine* non è entrata a far parte della Cloud, ma continua a funzionare correttamente.
Garanzie di Successo	La *TargetMachine* diventa un nodo della Cloud e può eseguire tutte le azioni elencate nell'attuale insieme API.
Attivatore	Il *MaintainerEngineer* inserisce il *NodePackage* nella directory di lavoro.
Descrizione	1. Il ME esegue lo script di avvio all'interno del *NodePackage*; 2. la *TargetMachine* inizializza tutti gli oggetti richiesti dai servizi; 3. il processo instanzia il sistema per la comunicazione remota; 4. viene avviato il Peer Sampling Service; 5. un nodo all'interno della Cloud risponde ad una richiesta sottomessa dal Peer Sampling Service della *TargetMachine*; 6. avviene il primo scambio di messaggi; 7. la *TargetMachine* viene correttamente aggiunta alla Cloud.

Estensioni	3.1 può avvenire un conflitto durante l’inizializzazione; 3.2 riprendi da 1 con diverse impostazioni per il servizio di comunicazione remota 5.1 nessun nodo risponde; 5.2 se questa è l’unica macchina della Cloud allora occorre attendere l’ingresso di nuovi nodi; 5.3 la vista così inizializzata non conduce ad alcun nodo attivo, in questo caso il ME deve verificare l’inizializzazione della vista.
Variazioni Tecnologiche e sui Dati	1. la scelta della tecnologia di comunicazione remota può portare ad architetture completamente diverse, almeno nei livelli più bassi del sistema.

CASO D’USO #3	**Gestire le Istanze**

<table>
<tr><td>Contesto d'Uso</td><td>Dal momento che la prima implementazione di P2PCloudSystem prevede una “Cloud privata” possiamo supporre che CloudUser stia agendo da una macchina appartenente alla Cloud. Una volta che CloudUser ha eseguito l'autenticazione è in grado di sottomettere richieste al P2PCloudSystem attraverso il livello Infrastruttura. Qualunque richiesta viene impachettata in un messaggio e consegnata al sistema appropriato: Dispatcher, Monitoring o Storage.</td></tr>
<tr><td>Ambito</td><td>The P2PCloudSystem</td></tr>
<tr><td>Livello</td><td>Summary Task</td></tr>
<tr><td>Attore Primario</td><td>CloudUser</td></tr>
<tr><td>Stakeholder and Interesssi</td><td><table><tr><td>Stakeholder</td><td>Interesse</td></tr><tr><td>CloudUser</td><td>Reperire e rilasciare nodi, creare e distruggere subclouds, ingrandire o restringere subclouds.</td></tr></table></td></tr>
<tr><td>Precondizioni</td><td>P2PCloudSystem è correttamente in esecuzione.</td></tr>
<tr><td>Garanzie Minime</td><td>La richiesta non può essere soddisfatta, ma P2PCloudSystem è ancora correttamente in funzione.</td></tr>
<tr><td>Garanzie di Successo</td><td>La richiesta viene inoltrata con successo ed il CloudUser può vedere il risultato non appena il processo è ultimato.</td></tr>
<tr><td>Attivatore</td><td>Il CloudUser avvia una funzione scelta dall'insieme API offerto dal livello Infrastruttura.</td></tr>
</table>

Descrizione	1. L'API selezionata: Funzione API compone il proprio *RequestMessage*(Caso d'Uso #4); 2. il messaggio è raccolto dal nodo locale ed avviene l'invio verso il proprio servizio: Inviare una richiesta *Request* ad un servizio(Caso d'Uso #5); 3. il sevizio esegue la richiesta Eseguire una richiesta *Request*(Use Case #6); 4. quindi viene mostrato il risultao al *CloudUser* al nodo dal quale è stata sottomessa la richiesta *Request*.
Estensioni	3.1 il servizio locale non è in grado di compiere la richiesta *Request* 3.2 Contattare un altro nodo per compiere una richiesta *Request* (Caso d'Uso #7); 3.3 Attendere la risposta ad un messaggio di richiesta *RequestMessage*(Caso d'Uso #8); 3.4 ricevere le risposte da tutti i nodi coinvolti nella richiesta. 3.3.1 la richiesta *Request* non può essere ultimata per la mancanza di risorse o per la presenza di un errore. 3.3.2 mostrare un messaggio di errore al *CloudUser*.

Variazioni Tecnologiche e sui Dati	Va nontato che al punto 3 ciascun servizio o funzione API possiede il proprio algoritmo. Per esempio la richiesta di creazione di un overlay richiede il coinvolgimento di altri nodi, ed il processo può fallire semplicemente perchè non vi sono nodi disponibili al momento. Ciascun servizio dovrà agire diversamente in relazione al messaggio di richiesta *RequestMessage* che riceve.

4.3 Architettura a livelli del Nodo

Prendiamo ora in esame l'aspetto fisico del sistema. La realizzazione di un'architettura P2P prevede la creazione di una rete di elementi paritetici. Questo significa che ciascun nodo, oltre ad essere in grado di comunicare ed organizzarsi in base alle risposte dei vicini, deve sostenere una parte del lavoro comune necessario al funzionamento della Cloud, come l'esecuzione di una richiesta, il monitoraggio di variabili globali (ad esempio il numero complessivo di elementi nella rete) o la gestione delle risorse condivise. Dunque è opportuno che in ogni nodo sia presente l'intero insieme di funzionalità previste dal sistema. Proseguiamo l'idea suggerita da [45], che illustra un'architettura suddivisa a livelli ed impiega algoritmi di gossip per la realizzazione della rete. La figura 4.3 vuole esserne una proposta implementativa. In ogni macchina andremo ad inserire il pacchetto che conterrà tutti gli oggetti raffigurati.

Peer Sampling Service : cominciamo con l'analizzare lo schema dal basso, dove compare il *Peer Sampling Service (PSS)* [36], che consente di costruire la rete di comunicazione necessaria per tutti gli oggetti di livello superiore. Questo protocollo costruisce infatti un grafo, la cui topologia converge ad un "random graph" con precisione e velocità che dipendono dagli attributi coi quali viene avviato. I parametri in ingresso modificano il comportamento dell'algoritmo e con esso anche le caratteristiche della rete costruita quali: la distribuzione del grado ("*degree distribution*"), il coefficiente di clustering ("*clustering coefficient*") o la lunghezza del cammino medio. In particolare la lunghezza del cammino medio risulta prossima a quella di un random graph. Tra le proprietà introdotte dal protocollo citiamo quelle di *self-organization*, che assicura la convergenza verso uno stato stabile della rete indipendentemente dalle condizioni iniziali di questa, e di *self-healing* che garantisce la continua sostituzione dei peer difettosi con quelli corretti

nella vista di ogni nodo.

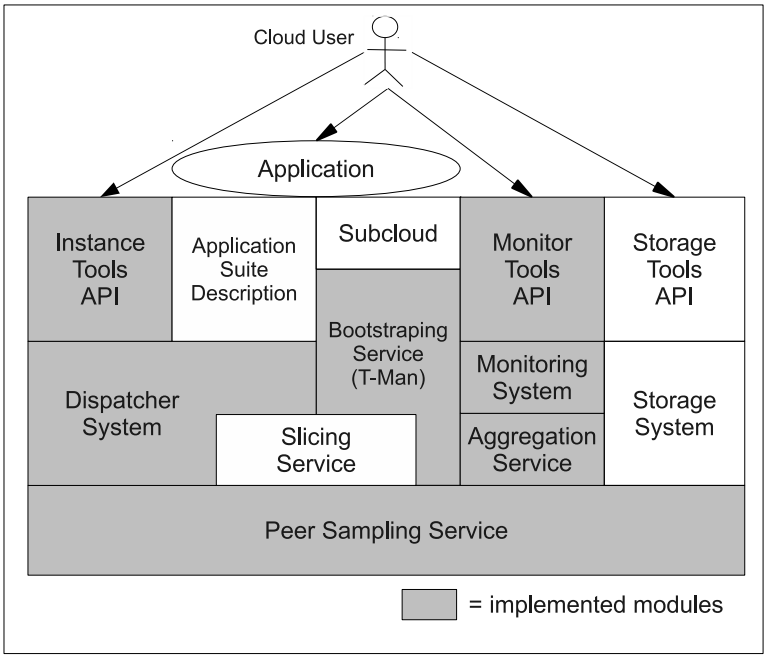

Figura 4.3: Diagramma dell'architettura a livelli interna al nodo. Ciascun nodo contiene il pacchetto all'interno del quale sono presenti tutti i moduli rappresentati in figura. Possiamo distiguere i moduli in *Servizi*, *Sistemi* o di *Interfaccia*. Sono stati evidenziati con il colore grigio gli oggetti che saranno implementati nel prototipo.

Slicing Service : sfruttando i collegamenti offerti dal PSS, lo *Slicing Service* [32] è in grado di ordinare i nodi rispetto ad un attributo X_i. Una volta terminato l'algoritmo è possibile richiedere al servizio delle porzioni ("*slice*") della rete, che corrispondono ad un sottoinsieme di questa. Una possibile interrogazione potrebbe essere ad esempio la richiesta di una particolare percentuale $y\%$ delle macchine con il più alto valore di bandwidth. Può risultare utile al nostro progetto nel momento della selezione delle risorse. Applicheremo dapprima l'ordinamento dei nodi attraverso lo Slicing Service su più attributi, quali ad esempio la potenza CPU, il carico di lavoro attuale o la percentuale di memoria libera.

In seguito alla richiesta di allocazione di risorse, il sistema protrà così restituire i nodi più appropriati, estraendoli in base ad una politica di selezione che si avvale degli attributi ordinati, e che tiene conto sia delle necessità dell'utente che di preservare le prestazioni della Cloud.

Aggregazione : il servizio di *Aggregazione* [33] distribuisce a ciascun peer informazioni globali come la dimensione della rete, avendo come unico requisito un'istanza del protocollo PSS attiva. L'algoritmo prevede che ciascun partecipante mantenga una variable condivisa. Questa viene aggiornata in seguito allo scambio del proprio valore con quello di un altro nodo, sul quale entrambi applicano la stessa funzione di `UPDATE`, che deve essere conosciuta a priori. Al termine dell'algoritmo tutte le variabili dei partecipanti tenderanno allo stesso valore. Attraverso l'aggregazione è possibile ricavare alcune importanti funzioni come il calcolo della media, l'estrazione degli estremi di massimo e minino, eseguire somme o moltiplicazioni, che possono rivelarsi utili per la costruzione del sistema di monitoraggio.

Boostraping e TMan : il compito del *Bootstraping Service (BS)* è invece quello di generare un *overlay* tra i nodi che partecipano al protocollo e si colloca in una posizione trasversale rispetto a diversi moduli, coi quali si troverà spesso in comunicazione. Un esempio comune di interazione vede il BS attivato dal dispatcher (DY), non appena questo riceve la richiesta di creazione di una sottonuvola. In accordo con [45] un utente può richiedere alla Cloud la formazione di una specifica rete tra alcuni nodi, che verranno messi in comunicazione diretta, e che prende il nome di *subcloud* (“sottonuvola”). Così facendo si creeranno degli overlay che formeranno dei sottoinsiemi disgiunti sulla rete costruita dal PSS, che è l'unico servizio richiesto dal BS. Dunque ciascuna sottonuvola viene affidata ad un BS dedicato, che opera in un certo senso come un controllore: fornisce l'accesso diretto all'applicazione costruita dall'utente, dialoga con il sistema di monitoraggio o i servizi sottostanti, quali ag-

gregazione o PSS, per eseguire attività di verifica. In questa versione del progetto prenderemo in considerazione T-Man [35], che adempirà alla costruzione degli overlay per le applicazioni. Questo protocollo di gossip genera un grafo la cui topologia è definita a priori attraverso una funzione `RANK`, che ordina i vicini di un nodo secondo un valore di similitudine calcolato dalla stessa. Con questo algoritmo riusciamo a collegare i nodi in anelli, alberi o T-Chord [35], una versione di Chord che fa uso di T-Man per la costruzione degli archi.

Moduli di sistema : osservando in modo più generale la figura 4.3 possiamo suddividere gli oggetti che compongono P2PCS tra moduli di *Sistema*, di *Servizio* e di *Interfaccia*. I moduli di sistema sono dedicati ad operazioni decisionali e di coordinamento. Abbiamo già accennato al monitoraggio delle risorse distinguendo quelle occupate da quelle ancora libere del quale si occuperà il *Monitoring System (MY)*. La gestione della memoria di massa e le strutture dati che conservano le informazioni sugli account saranno affidate allo *Storage System (SY)*, mentre il *Dispatcher System (DY)* curerà l'insieme delle azioni che servono per l'adempimento di una richiesta, compreso l'avvio degli eventuali moduli aggiuntivi. Negli sviluppi futuri del progetto possiamo attenderci l'ingresso di nuove funzionalità del sistema, alcune delle quali è bene prendere in considerazione fin da ora. Tra queste possiamo prevedere la necessità di registrare le azioni compiute, che potrebbe concretizzarsi nell'inserimento di un modulo di *log*. Non dobbiamo poi tralasciare il problema della sicurezza, che avrà bisogno di una sezione dedicata, e molto probabilmente dovremo dotare P2PCS e di un dispositivo capace di comunicare con un gestore di immagini virtuali.

Moduli di interfaccia : I moduli di interfaccia costituiranno l'insieme delle API dell'infrastruttura che saranno direttamente utilizzabili dall'utente. Abbiamo adottato nomi analoghi a quelli analizzati nel confronto delle API in 2.4. Gli oggetti "Instance Tools API", "Monitor

Tools API” e “Storage Tools API” espongono le funzionalità offerte dai sottostanti moduli di sistema all’utente. Contrariamente i concetti di “Application Suite Description” e “Subcloud” derivano da [45] e rappresentano rispettivamente la descrizione delle caratteristiche delle macchine da estrarre dalla rete e porre in un overlay, e l’insieme dei metodi per gestire questo sottoinsieme di peer. Questi si pongono invece a supporto dell’Applicazione che verrà costruita fornendo il mantenimento dei nodi raccolti nell’overlay.

Abbiamo evidenziato con il colore grigio gli elementi che andremo ad implementare per la creazione di un prototipo basilare. In questo elaborato volgeremo la nostra attenzione sul livello infrastrutturale della Cloud. Possiamo dunque pensare all’utente rappresentato dalla figura 4.3 come a Philip. Dovrà poter interagire con la Cloud sfruttando le API che metteremo a disposizione e riuscire così ad istanziare una sottonuvola sulla quale poi costruire un’applicazione.

4.4 Modello di Deployment

In questa sezione passeremo ad esaminare più nel dettaglio i moduli osservati in 4.3 costruendo un diagramma dei componenti per definire le interfacce tra ciascuno di essi e le modalità di interazione con gli oggetti esterni al sistema.

Il Peer Sampling Service è il modulo di livello più basso e rappresenta un requisito per molti altri. Deve essere dotato di un oggetto “*vista*”, che viene condiviso da entrambi i thread attivo e passivo, e dell’accesso al modulo di chiamata remota per la comunicazione con gli altri nodi. La vista di un protocollo di gossip si compone di una lista di *NodeDescriptor* ciascuno dei quali identifica una macchina della rete, e costituisce l’insieme degli archi uscenti dal nodo nel grafo generato dal

PSS. Il servizio espone la propria interfaccia `PSSAPI` che include le funzioni di inizializzazione del servizio *init()*, e selezione di un peer dalla vista *selectPeer()*. Lo Slicing Service potrebbe necessitare di un accesso diretto alla vista, per questa ragione prevediamo anche l'interfaccia *ViewInterface*.

L' Aggregazione sfrutta il PSS per il calcolo di valori globali, che poi renderà accessibili ad altri moduli, come ad esempio il sistema di monitoraggio. Questo servizio di gossip prevede che i due thread condividano una variabile, contenente la stima dell'attributo che si vuole calcolare, e che può essere acceduta tramite l'interfaccia *getGlobalValue()*. Anche l'Aggregazione deve poter accedere alle chiamate remote per interagire con i nodi vicini.

Lo Slicing Service deve poter accedere alla vista che il PSS mette a disposizine e contattare con i nodi presenti in questa. Il risultato del protocollo è la sequenza di macchine ordinate secondo i valori degli attributi selezionati. Questo modulo risulta utile sia al sistema di monitoraggio, che potrebbe utilizzarlo per eventuali statistiche, sia dal Dispatcher System che è così in grado di applicare strategie di selezione delle macchine più efficienti potendo disporre dell'elenco ordinato.

Il Bootstraping Service può avvalersi sia delle informazioni prodotte dallo SlicingService che accedere alla rete direttamente attraverso il PSS, e consultare le notifiche ricavate dal Monitoring System. Consente all'applicazione costruita dall'utente di interagire direttamente con gli altri nodi dell'overlay sfruttando le api dedicate *OverlayAPI*.

Nel Front-End collochiamo gli oggetti di supporto come gli strumenti per il Maintainer Engineer *MaintainerTools*, che sono inerenti al monitoraggio della Cloud e devono mostrare in output le informazioni che il sistema ha raccolto internamente dai nodi della rete. Gli script *UserRequestScript* devono tradurre le richieste dell'utente in chiamate agli

opportuni algoritmi in grado di realizzarle. Possiamo usare l'elenco già presentato nel paragrafo 2.4 come tabella di marcia per identificare le funzionalità che intendiamo implementare. Le interazioni tra Front-End e Back-End avvengono attraverso le porte fornite dal Dispatcher System, Monitoring System, Storage System e *VitualHandler*. Possiamo prevedere che ciascun modulo richieda una propria specifica modalità per comunicare con il Front-End. In particolare il VirtualHandler sfrutterà le librerie dell'ambiente virtuale per consentire l'avvio di macchine virtuali.

I moduli di sistema sono in stretta relazione. Qualora saranno implementati il sistema di log *LogSystem* e di sicurezza *SecuritySystem* possiamo prevedere che buona parte delle richieste, che giungono dal Front-End, debbano essere esaminate da questi prima di avviare le procedure per la loro esecuzione. A tal proposito questi sistemi espongono le proprie interfacce *LogSYAPI* e *SecuritySYAPI* al Monitoring System e Dispatcher System. Infatti al momento della sottomissione di una richiesta bisogna poterne verificare prima l'attendibilità, quindi inserire un'opportuna voce in un registro che mantiene l'elenco delle richieste.

Allo Storage System spettano compiti particolarmente onerosi. Possiamo supporre che il sistema offra immagini virtuali definite a priori, e lasci poi la possibilità all'utente di scegliere quella più appropriata (come abbiamo visto nel caso di Amazon in 2.1). In questo modo lo Storage System deve mantere l'insieme di queste immagini. In più si deve occupare di gestire la memoria di massa per ciascun utente, definendo eventualmente una politica che fa uso di quote.

Il software che produremo sarà contenuto in un unico pacchetto che chiameremo *NodePackage*, e che ciascuna macchina dovrà possedere. Per rendere il calcolatore un nodo attivo della Cloud, basterà quindi avviare il programma principale fornito con l'applicativo.

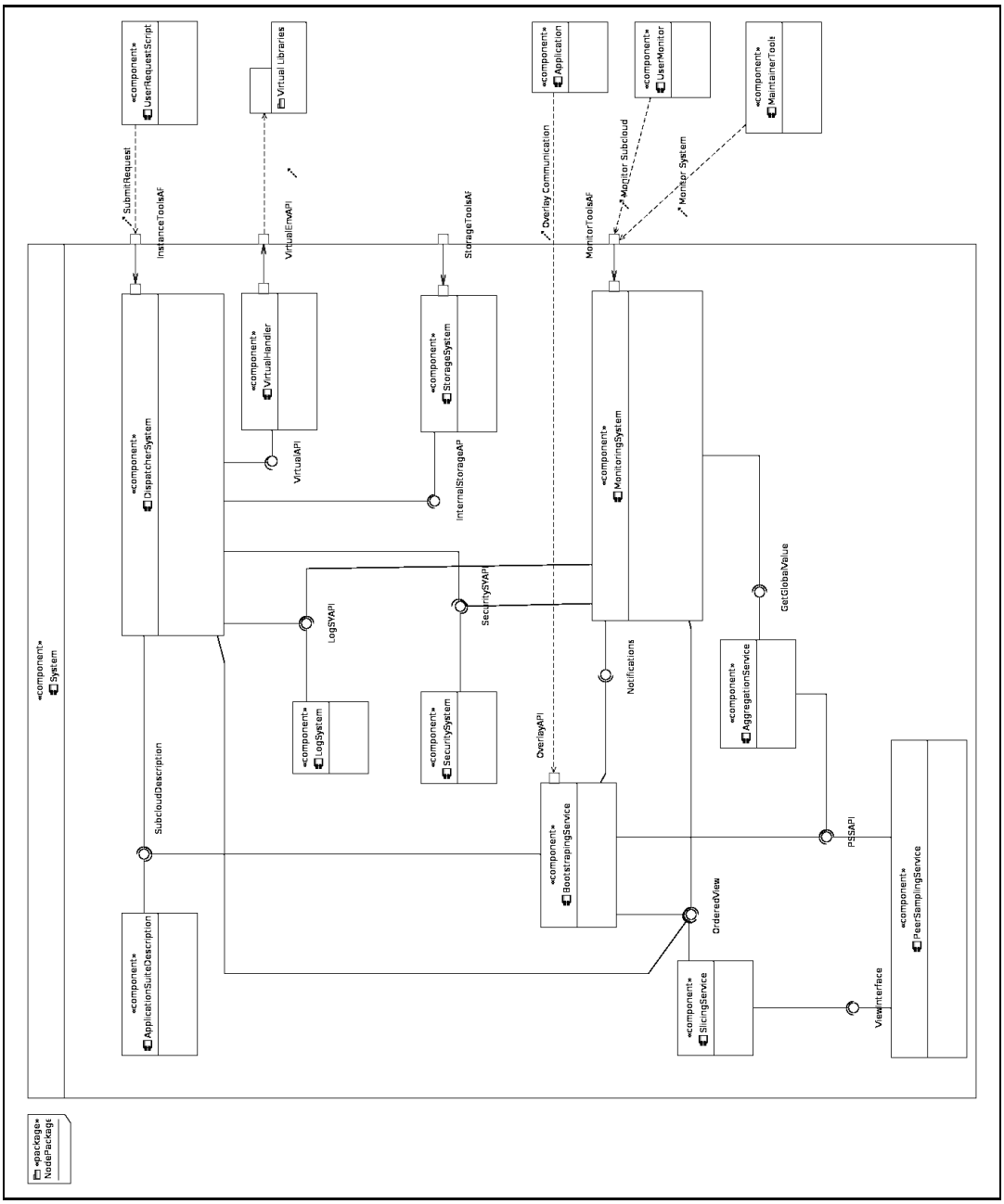

Figura 4.4: Diagramma dei componenti.

Capitolo 5

Implementazione di "P2P Cloud System"

In questo capitolo mostreremo gli aspetti più rilevanti dell'implementazione del sistema. Cominceremo con l'introdurre alcune scelte implementative in 5.1 che definiscono anche i confini dell'attuale lavoro. Seguiranno poi le sezioni relative alle comunicazioni tra un nodo e l'altro (5.2) e alla struttura base per implementare un algoritmo di gossip (5.3) che costituisce uno dei punti essenziali di questo prototipo. Ci occuperemo quindi delle modalità di interazione tra il lato Front-End ed il Back-End in 5.4. In sezione 5.5 illustro i passi per scaricare, compilare ed eseugire il prototipo. Concludono il capitolo alcune valutazioni sull'esecuzione del sistema in 5.6.
In questa esposizione tralascerò la descrizione degli algoritmi di gossip per concentrarmi su alcuni dettagli implementativi che ritengo doveroso illustrare ai fini della comprensione del progetto per gli sviluppi futuri. Il lettore interessato ad una più completa comprensione può fare riferimento direttamente alle fonti della bibliografia o allo pseudocodice degli algoritmi inserito in appendice.

5.1 Scelte Implementative

Il prototipo che andiamo a sviluppare prevede la creazione della rete attraverso il Peer Sampling Service (PSS). Implementiamo il servizio di Aggregazione per ricavare il numero di nodi n presente nella Cloud sfruttando la formula inversa per il calcolo della media $n = \frac{1}{m_i}$ dove m_i è il valore medio calcolato dal nodo i. Utilizziamo T-Man come Bootstraping Service, per generare un overlay tra un sottoinsieme dei nodi che, in accordo a [45], chiamiamo "sottonuvola" o subcloud, della quale dovremmo verificare la corretta topologia.
Ci avvaliamo del linguaggio Java e delle relative chiamate di procedura remota del paradigma JRMI.
Sviluppiamo un primo insieme minimale di API che comprende le funzioni di:

1. **run-nodes** $<$ *subcloud_id* $>$ $<$ *number* $>$: genera un overlay usando l'algoritmo di T-Man, selezionando un numero *number* di nodi dalla rete ed attribuisce alla sottonuvola il nome identificativo specificato dal parametro *subcloud_id*,

2. **terminate-nodes** $<$ *subcloud_id* $>$ $<$ $nodeName_1$ $>$ $[...nodeName_n]$: conttatta ciascun nodo inserito come parametro e lo estrae dalla sottonuvola specificata da *subcloud_id*,

3. **describe-instaces** $<$ *nodeName* $>$: produce in output la descrizione del nodo dal quale viene eseguito mostrando se questo appartiene ad una *subcloud* o meno e, in caso affermativo, come sono disposti i vicini,

4. **montor-instaces**: funzione di monitoraggio, pensata per il manutentore, che restituisce il numero dei nodi presenti nella rete avvalendosi del servizio di Aggregazione,

5. **unmonitor-instaces**: funzione complementare alla precedente, che ferma il monitoraggio del numero dei nodi sulla macchina dalla quale viene eseguita,

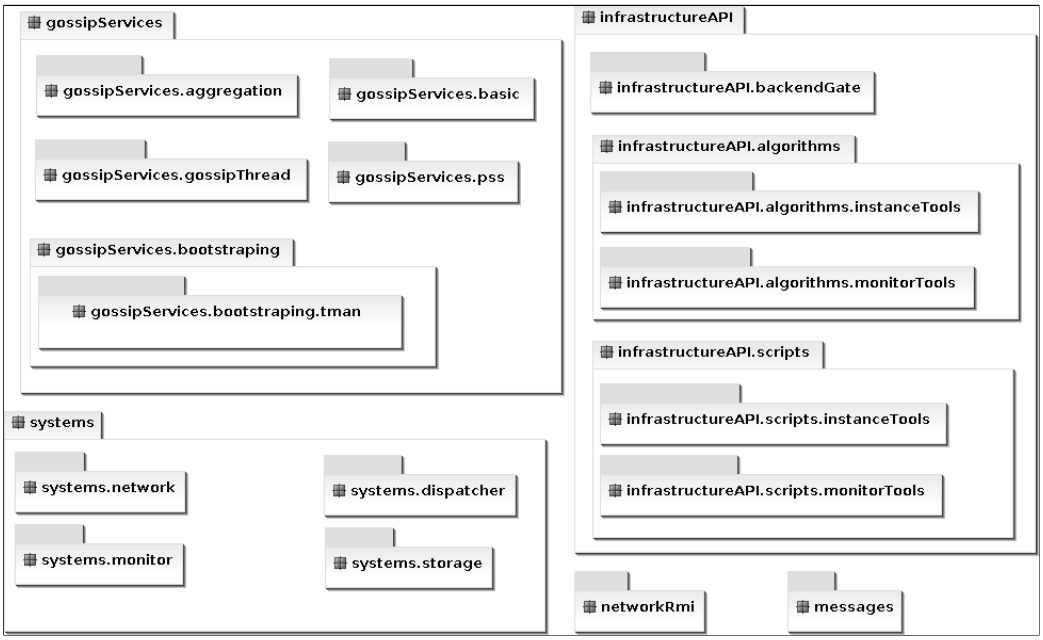

Figura 5.1: Diagramma dei pacchetti nei quali viene suddiviso il codice sorgente: *gossipServices* contenente i protocolli di gossip, *systems* che racchiude i moduli di sistema, *infrastructureAPI* che ospita le classi inerenti alle interazioni tra Front-End e Back-End e *networkRmi* che include gli oggetti per le comunicazioni remote.

6. **add-new_nodes** $< sudcloid\_id > < numberOfNodes >$: funzione caratteristica di questo prototipo che aggiunge un numero di $numberOfNodes$ nuovi nodi alla subcloud $< sudcloid\_id >$ estraendoli tra quelli non ancora coinvolti in altre attività di bootstraping.

Implementiamo ciascuna funzione dell'interfaccia con una coppia di oggetti. Chiamiamo il primo *algoritmo* e conterrà il codice che deve essere eseguito dal Back-End, mentre il secondo *script*, e consentirà all'utente dal lato Front-End di avviare l'azione desiderata grazie all'invio di una richiesta al sistema. Faremo quindi in modo che l'utente possa interagire con la Cloud tramite il terminale, dal quale potrà attivare gli script che andremo a preparare.

Organizziamo il codice secondo la gerarchia a pacchetti prevista da Java. La figura 5.1 illustra proprio questa suddivisione. La parte dei servizi gossip è raccolta in *gossipServices*, e comprende il Peer Sampling Service, l'Aggregazione, T-Man (T-Man) con l'aggiunta di un pacchetto per gli oggetti comuni a tutti i protocolli denominato *basic*. In *systems* compaiono i moduli di sistema: *dispatcher*, *monitor*, *storage* e *network* che si occupa delle comunicazioni con moduli di sistema di altri nodi. *InfrastructureAPI* contiene l'insieme delle API di infrastruttura e degli oggetti che controllano l'interazione tra Front-End e Back-End. Ciascuna funzione API è composta da due parti: una che implementa l'algoritmo vero e proprio che inseriamo in *algorithms* e l'altra, denominata "script", che consente all'utente di avviare la funzione desiderata, ed è contenuta nell'omonimo pacchetto *scripts*. Seguiamo la suddivisione delle API osservata in 2.4 che prevede la catalogazione delle funzioni in: *strumenti di istanza*, di *monitoraggio*, *indirizzamento*, ecc. Nel nostro caso compaiono quindi due pacchetti riguardanti proprio la gestione delle istanze *instanceTools* ed il monitoraggio *monitoringTools*. Un ulteriore pacchetto *networkRmi* raccoglie invece le classi che si prendono a carico la comunicazione remota tra i nodi.

Da questo momento assumiamo che su una macchina possa essere avviata una sola istanza del programma che la introduce nella Cloud come un nodo. Dunque nel seguito del capitolo useremo i termini "nodo" e "macchina" come sinonimi, a meno che non siano necessari chiarimenti. Dal momento che il progetto prevede la costruzione di un sistema privato, supporremo anche che ogni script di richiesta sia avviato da una macchina appartenente alla Cloud.

5.2 Comunicazioni di rete

Identifichiamo ciacun nodo con un oggetto chiamato "descrittore di nodo" `NodeDescriptor` che contiene il codice identificativo, il nome e l'indirizzo della macchina. Implementiamo le comunicazioni tra un nodo e l'altro attraverso l'invio e la ricezione di messaggi `Message`. Nella forma più generica un

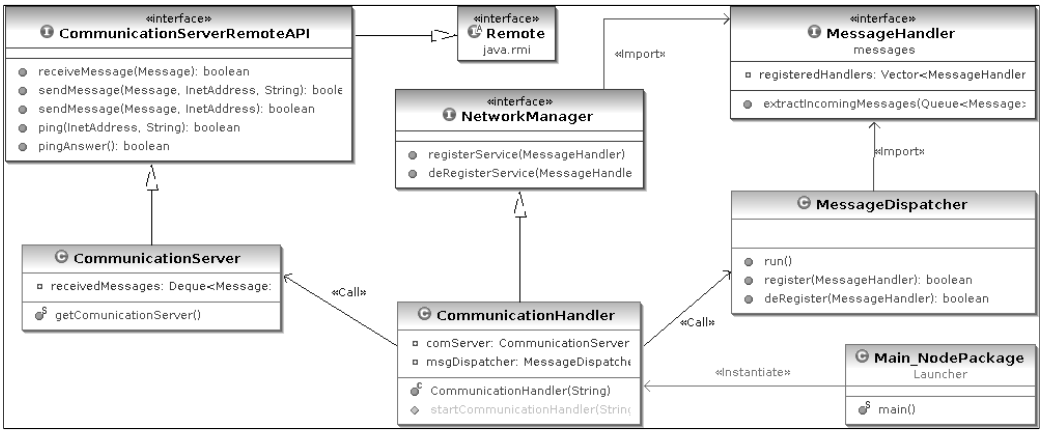

Figura 5.2: Diagramma di classe che mostra i tre controllori delle comunicazioni. Il *CommunicationHandler* si occupa di instanziare il *CommunicationServer* ed il *MessageDispatcher* relativi rispettivamente all'interazione con l'rmiregistry, ed allo smistamento dei messaggi.

messaggio contiene i nodi descrittori del mittente e del destinatario mentre il corpo appartiene alla classe *Object* . Diversi moduli devono poter scambiare informazioni con l'analoga parte presente su un altro nodo. Il caso più frequente è costituito dal protocollo di gossip e vede ad esempio l'interazione del thread attivo sul nodo i con il thread passivo nel nodo j. Questo significa che il messaggio viene indirizzato alla macchina specificata ad un particolare modulo che ne ha fatto richiesta. Dobbiamo allora fare in modo che a ciascun servizio o sistema siano consegnati i messaggi che gli spettano.

La figura 5.2 mostra gli oggetti che controllano le comunicazioni. Possiamo genereare un nodo grazie all'esecuzione della classe *Main_NodePackage* che contiene la funzione *main()*. In quel momento viene creato anche un *CommunicationHandler* che si fa carico dell'istanziazione degli oggeti necessari per la comunicazione: il *CommunicationServer* ed il *MessageDispatcher*. Il primo implementa l'interfaccia remota *CommunicationServerRemoteAPI* ed offre i meccanismi di interazione con l'*rmiregistry* che consentono l'invio e la ricezione di un messaggio o del segnale di "ping" per verficare la raggiungibilità di una macchina. Il compito del *MessageDispatcher* è in-

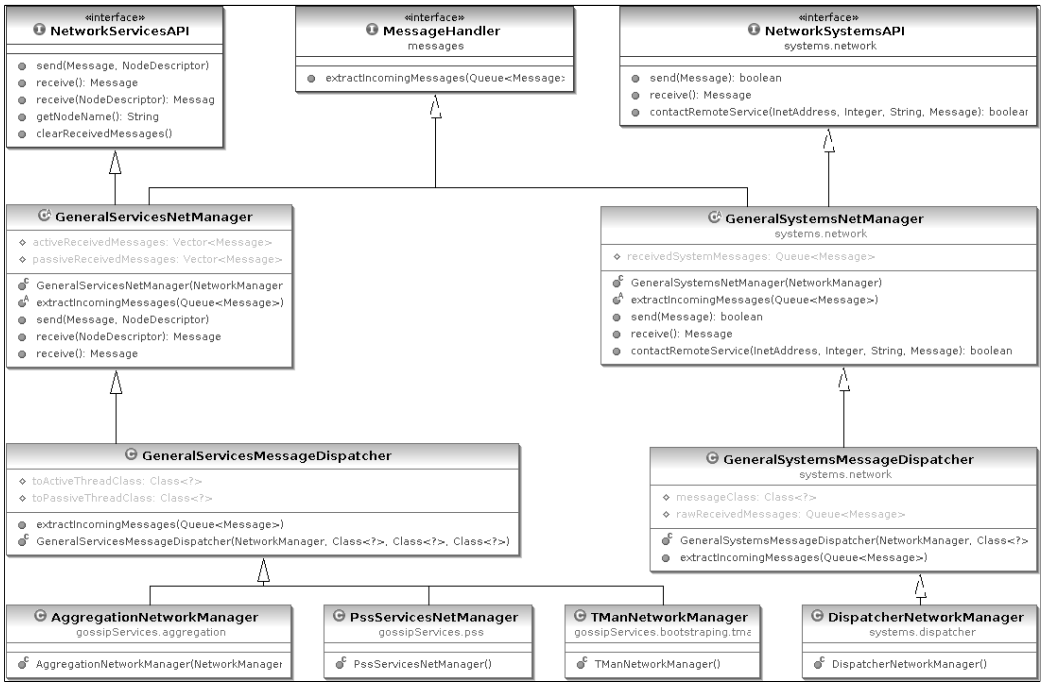

Figura 5.3: Diagramma di classe raffigurante le gerarchie dei gestori di rete dei moduli di servizio che implementano *NetworkServicesAPI* e di sistema che derivano da *NetworkSystemsAPI*.

vece quello di passare in rassegna i messaggi in arrivo contenuti nella coda del *CommunicationServer* e consegnare a ciascun modulo i propri. Perché questo meccanismo sia possibile, ogni componente che necessita la ricezione di una risposta deve implementare un oggetto *MessageHandler* per poter estrarre dalla coda generale i propri messaggi. Dovrà infine registrarlo al *NetworkManager* richiamando il metodo *registerService(MessageHandler)* per essere in grado di ricevere messaggi da altri nodi. L'interfaccia *MessageHandler* richiede infatti l'implementazione del metodo *extractIncomingMessages()* nella quale ciascun modulo deve specificare le azioni necessarie al riconoscimento dei propri messaggi. Osserviamo ora la figura 5.3 che mostra i gestori di rete specifici per alcuni moduli. Possiamo notare due gerarchie che si sviluppano l'una a partire dall'interfaccia *NetworkServicesAPI*, che offre supporto ai servizi di gossip, e l'altra che implementa *NetworkSystemsAPI*

ed è dedicata ai moduli di sistema. Consideriamo ora i tre servizi di gossip di Aggregazione, PSS e T-Man, che comunicano attraverso tre tipologie di messaggi distinte. Ogni modulo deve implementare il proprio gestore di rete e dotarlo delle azioni necessarie all'estrazione dei messaggi. `GeneralServicesMessageDispatcher` realizza l'interfaccia `MessageHandler` ed offre un metodo di estrazione che si basa sul tipo `Class<?>` della classe. Questo metodo viene sfruttato da tutti e tre i gestori in modo da semplificarne l'implementazione, che si riduce a specificare per ciascuno il tipo di classe dei messaggi da estrarre, ed i tipi dei thread di gossip coinvolti nell'interazione.

5.3 Struttura dei Thread di Gossip

Nel nostro prototipo tutti i servizi necessari sono realizzati grazie a protocolli di gossip. La struttura generale di un simile algoritmo prevede l'esecuzione di una coppia di thread denominati *Thread Attivo* e *Thread Passivo* che operano concorrentemente. Come abbiamo già accennato precedentemente nella sezione 5.2, il caso comune di interazione prevede che il Thread Attivo sul nodo i contatti quello Passivo sul nodo j e quest'ultimo invii la relativa risposta. È possibile che questi processi si trovino in concorrenza nell'accesso a variabili condivise. Ad esempio nel Peer Sampling Service entrambe le parti (attiva e passiva) devono operare sulla vista, nell'Aggregazione su una variabile di tipo *double* ed infine in T-Man sulla lista dei vicini.
Scendiamo ora più nel dettaglio seguendo il diagramma di figura 5.4. La classe `GossipThread` costituisce il componente cardine della struttura in quanto contiene gli elementi essenziali e le funzioni comuni a tutti i protocolli di gossip. Questi devono quindi essere specificati nel momento della creazione e corrispondono a: la lunghezza del ciclo in millisecondi, la vista contenente i nodi vicini ed una classe di tipo `GossipThreadInstructions` che raccoglie le istruzioni da eseguire. Per come è stata pensata, la struttura di un algoritmo di gossip prevede sempre la realizzazione di tre classi. Due sono dedicate all'implementazione delle istruzioni inerenti al Thread Attivo e Passivo. Una

terza racchiude invece le istanze delle precedenti e potrà essere utilizzata per avviare l'algoritmo. Lo sviluppatore può avvalersi di due strategie per la costruzione di un protocollo di gossip.
Nel primo caso la coppia dei thread attivo e passivo viene realizzata creando due oggetti che ereditano dalle classi astratte `ActiveGossipThread` e `PassiveGossipThread`. Devono poi essere scritte le istruzioni relative alla parte attiva o passiva nell'opportuno metodo astratto rispettivamente, *activeThreadInstructions()* o *passiveThreadInstructions()*. Una terza componente conterrà poi le istanze di questi due oggetti e provvederà ad avviarli. Abbiamo usato questo meccanismo per l'implementazione per Peer Sampling Service.
La seconda strategia consiste invece nell'avvalersi di `GossipBasedProtocol`, che crea entrambi i processi attivo e passivo secondo le impostazioni definite nel costruttore, e fornisce il metodo per l'avvio dell'algoritmo. Le classi, alle quali sono affidate le istruzioni della parte attiva e passiva, dovranno in questo caso realizzare le due interfaccie `ActiveGossipThreadInterface` e `PassiveGossipThreadInterface`. Questo secondo approccio è risultato utile per l'implementazione dell'Aggregazione, per la quale è stato necessario introdurre un'ulteriore classe intermedia: `AggregationMiddleware`, che si occupa di conservare tutte le informazioni condivise dai due processi, compresa la variabile del valore stimato.

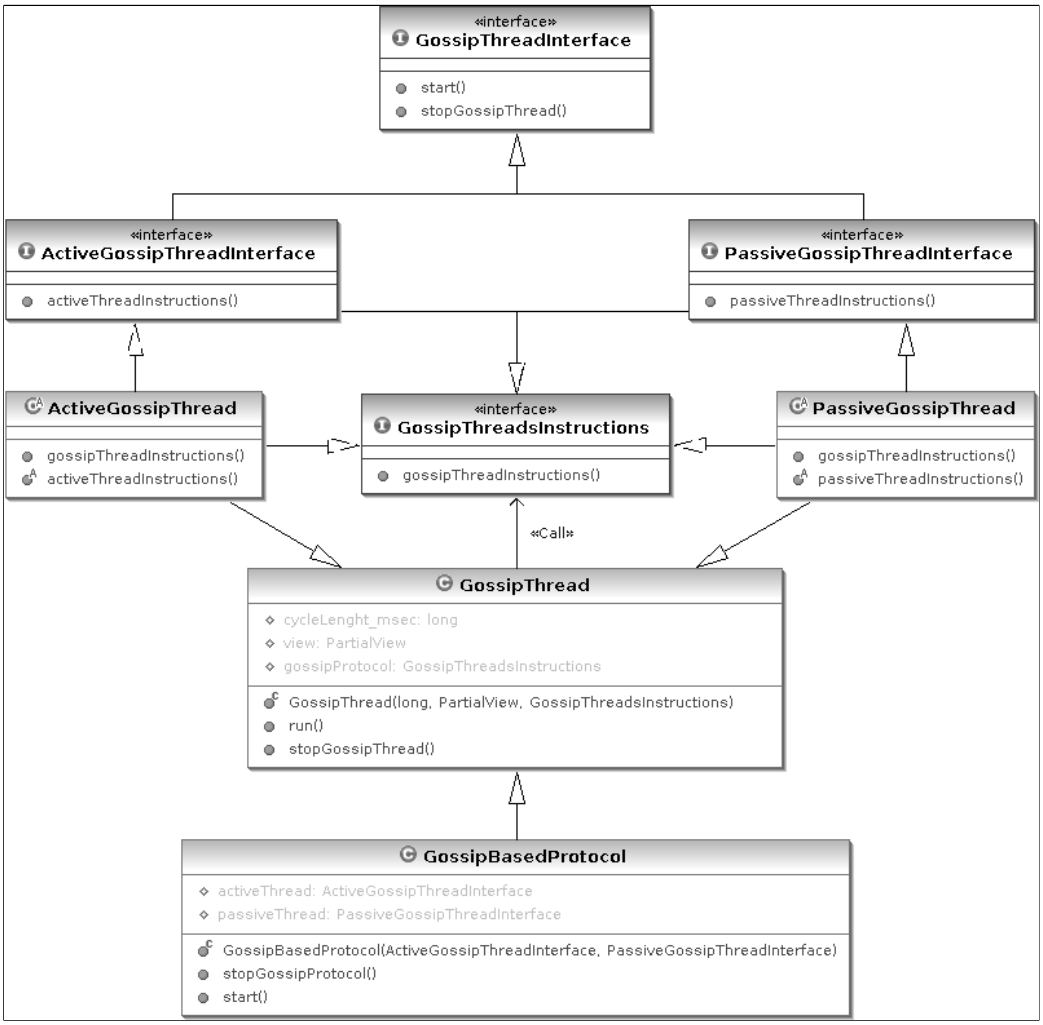

Figura 5.4: Diagramma di classe rappresentante la struttura basilare per i protocolli di gossip usati nel sistema. Un protocollo di gossip comprende sempre tre classi: due per l'implementazione del Thread Attivo e Passivo ed una terza che include le istanze delle precedenti e che sarà poi usata per avviare l'algoritmo.

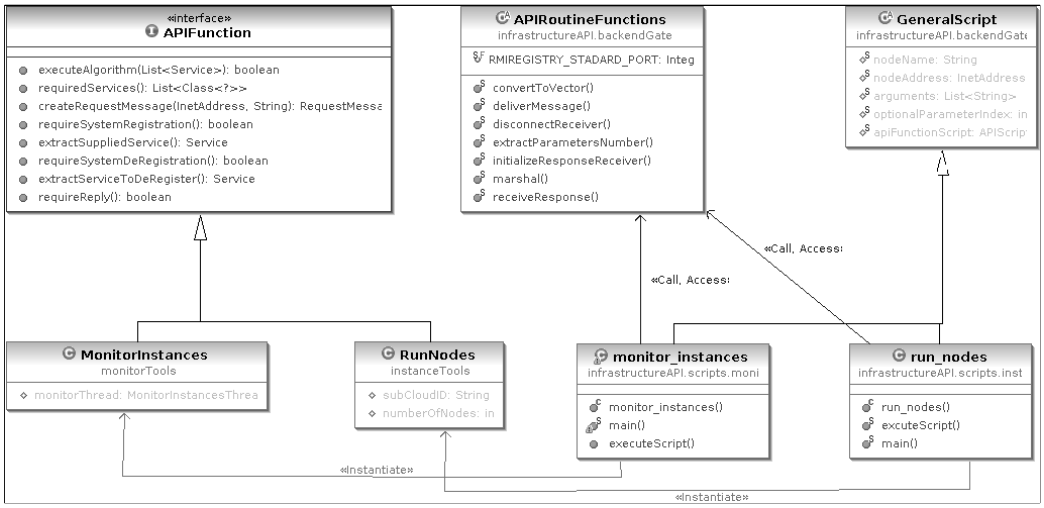

Figura 5.5: Diagramma di classe che illustra gli elementi che compongono la funzioni di *run-nodes* e *monitor-instances*. Dal lato Front-End l'utente avvierà *run_nodes* o *monitor_instances* che, assumono il ruolo di script di avvio. Ogni script gestisce i parametri in ingresso, istanzia la classe che implementa l'algoritmo *RunNodes* o *MonitorInstances*, ed infine spedisce il messaggio di richiesta nel Back-End del sistema.

5.4 Interazione tra Front-End e Back-End

In 5.1 abbiamo introdotto il fatto che le API dell'infrastruttura sono costituite da due parti: una che definiamo *algoritmo* e che risiede nel Back-End, e una seconda che chiamiamo invece *script* e che sarà eseguita dall'utente nel Front-End.

In figura 5.5 ho inserito le classi che concretizzano le funzioni di **run-nodes** e **monitor-instances**. Sebbene il diagramma illustri solo due esempi è facile intuire che per ciascuna nuova API si debbano aggiungere due oggetti alla gerarchia: uno che riproduce l'algoritmo e l'altro che si comporta come script.

L'interfaccia `APIFunction` riveste un ruolo fondamentale e deve essere implementata da ogni oggetto che si prende a carico la realizzazione dell algoritmo di una funzione dell'infrastruttra. In questo modo tale classe po-

trà garantire: l'insieme dei passi per l'esecuzione dell'algoritmo, che devono comparire nel metodo *executeAlgorithm()*, la generazione di un messaggio di richiesta `RequestMessage`, che sarà inoltrato al sistema, e la specifica dei servizi richiesti con *requiredServices()*. Una funzione API può anche introdurre un nuovo servizio attualmente non disponibile, che deve essere registrato per potervi accedere in futuro. Per questo scopo sono presenti nell'interfaccia APIFunction le coppie di metodi: *requireSystemRegistration()* ed *extractSuppliedService()* per l'aggiunta di un nuovo servizio, ed anche *requireSystemDeRegistration()* ed *extractServiceToDeregister*, per la "de-registrazione" e conseguente terminazione di un modulo attivo.
I compiti di uno script sono invece quelli di: verificare e gestire i parametri in ingresso, reperire l'identità del nodo dal quale si stà effettuando la richiesta, istanziare la classe APIFunction addetta all'adempimento della API ed inserirla in un messaggio di richiesta `RequestMessage`, quindi inoltrare il messaggio appena creato alla Cloud. A questo punto il processo script può attendere le risposte che perverranno dal sistema.
Consideriamo **run-nodes**, che richiede la costruzione di un overlay tra un sottoinsieme di nodi della rete, e sfruttiamola come esempio per analizzare più in dettaglio la struttura interna. Nel diagramma 5.5, la classe `RunNodes` costituisce la parte algoritmica dell'API, ed infatti implementa l'interfaccia APIFunction. Contrariamente `run_nodes` (dove è usato il carattere "underscore" nel nome) svolge il ruolo di script e sarà proprio questo ad istanziare l'algoritmo e a spedire la richiesta al sistema, avvalendosi delle procedure messe a disposizione dalla classe `APIRoutineFunctions`. La forte somiglianza tra il nome della funzione API ("run-nodes") e lo script ("run_nodes") è stata adottata per agevolare l'implementazione. Per rendere più semplice l'accesso all'infrastruttura da terminale, creiamo per ciascuna API il relativo programma in lingaggio *bash script*, che compie la chiamata del bytecode Java, e raccogliamo l'insieme di questi nella directory *API-scripts*. In questo modo l'utente deve semplicemente occuparsi dei parametri della funzione, per poterla eseguire. Vista tuttavia la completa coesione che esiste tra

la definizione della API e l'implementazione del relativo script, possiamo al momento accorpare i due concetti, riservandoci di esplicitare eventuali discrepanze qualora emergano. È importante invece mantenere separati i concetti di *RunNodes*, classe che definisce l'algoritmo della API nel Back-End, con **run-nodes** che corrisponde invece alla sua definizione (potremmo dire in inglese: signature).

Riferiamoci ora al diagramma 5.6 che offre una panoramica del pacchetto *infrastructureAPI*, osservato nell'immagine 5.1, addetto proprio alle interazioni tra Front-End e Back-End. In 5.6 è raffigurata la gerarchia dei servizi *Service* in relazione con il *DispacherSystem*, che occupa il ruolo di gestore implementanto l'interfaccia *ServiceHandler*. I moduli di sistema dovranno infatti mantenere l'insieme dei servizi attualmente attivi ed eventualmente modificarlo a tempo di esecuzione.

Seguiamo ancora l'esempio di **run-nodes** per l'esecuzione della quale deve essere attivato T-Man che costruisce un overlay tra i nodi selezionati. Al momento dell'attivazione di un nodo sono presenti i servizi del *NetworkManager* per la comunicazione remota, *AggregationService* per le azioni di monitoraggio ed il *PeerSamplingServiceAPI* che costruisce un grafo tra le macchine. Non appena un nodo riceve la richiesta della funzione API di **run-nodes**, seleziona un sottoinsieme di macchine dalla rete ed avvia un'istanza di T-Man. Viene così aggiunto un nuovo servizio al sistema: *TManService*. Questo deve essere registrato nel *DyspatcherSystem*, in modo da poter essere recuperato in un secondo momento, e sfruttarne così le funzionalità oppure terminarlo.

L'intero processo di esecuzione della API **run-nodes** è esplicitato nel diagramma di sequenza 5.7 che ne sottolinea lo svolgimento temporale.

5.5 Utilizzo del sistema

In questa sezione presento una breve descrizione che spiega l'utilizzo del prototipo del sistema.

È possibile scaricare il codice sorgente dall'indirizzo:

```
http://cloudsystem.googlecode.com/svn/trunk/source
```

con l'ausiglio di un qualunque client di versionamento. Il prototipo è stato testato su di un sistema Linux Ubuntu 9.10. È necessario avere installato Java versione 1.6.
È possibile testare il prototipo sulla propria macchina in locale oppure connettendosi al laboratorio didattico.

Compilazione e avvio su macchina locale

Dopo aver scaricato il codice sorgente compiliamo il progetto con il comando "ant". Possiamo digitare l'istruzione da terminale direttamente dalla directory usata per il download.

```
$ant
```

Assumiamo inoltre che tutti i comandi che seguono siano avviati dalla stessa directory nella quale abbiamo inserito il codice sorgente. Essendo questa la prima esecuzione del prototipo occorre creare gli script che consentono la chiamata delle funzioni di infrastruttura. Il procedimento è automatizzato dal processo "createSHScripts.sh" che deve essere semplicemente avviato da linea di comando:

```
    $ ./createSHScripts.sh
Script creation...
processing:
API-scripts/describe-instances.sh
API-scripts/terminate-nodes.sh
API-scripts/run-nodes.sh
API-scripts/add-new_nodes.sh
API-scripts/unmonitor-instances.sh
API-scripts/monitor-instances.sh
```

Abbiamo così creato tutti gli script di avvio relativi a ciascuna funzione API, e posti della directory API-scripts. A questo punto possiamo avviare il primo nodo utilizzando “startNode.sh”. Nel caso di avvio da macchina locale, occorre sempre specificare il nome del nodo che si vuole attivare. Il comando completo sarà dunque:

```
$./startNode.sh -n node1 -agg 1
```

dove l’opzione `-n` indica il nome della macchina, e `-agg` è il valore con il quale viene initializzata l’Aggregazione. È possibile inserire qualunque valore intero in input al servizio dipendentemente dalla funzione che si vuole calcolare. Nel nostro caso siamo interessati a ricavare il numero di calcolatori nella rete attraverso la formula $n = \frac{1}{m}$ dove m è la media calcolata. I successivi nodi saranno attivati inserendo solamente il nome, senza specificare alcun valore iniziale per l’Aggregazione:

```
$./startNode.sh -n node2
```

In fase di avvio del nodo è possibile inserire le sequenti opzioni:

- -n: nome del nodo che si vuole avviare,
- -lab: parametro che deve essere inserito solo se si avvia il nodo nel laboratorio didattico del Dipartimento di Informatica di Bologna,
- -agg: valore di inizializzazione del servizio di Aggregazione. Nel caso non venga specificato viene assunto come valore iniziale 0,
- -vv: (View Visualization) visualizza nel terminale la vista del PSS,
- -h: visualizza l’utilizzo dello script.

In caso di successo il terminale dovrebbe mostrare il seguente messaggio:

```
classpath: /home/michele/Desktop/prototypeTESI/bin/
codebase: file:///home/michele/Desktop/prototypeTESI/deploy/
cloudsystemNode.jar
security: file:///home/michele/Desktop/prototypeTESI/
security.policy
Creating the ComunicationServer..
Using an existing rmiregistry
Failed!
Trying to create a new registry
done
Services registered to rmiregistry on port: 1099
- node1###
Node is up registered with service name: node1###
0) 1121954548 node1### /127.0.0.1 50
1) 1121984339 node2### /127.0.0.1 50
2) 1122014130 node3### /127.0.0.1 50
3) 1122043921 node4### /127.0.0.1 50
4) 1122073712 node5### /127.0.0.1 50
5) 1122103503 node6### /127.0.0.1 50
6) 1122133294 node7### /127.0.0.1 50

Setting the node node1###
 with the Aggregation value of 1.0
AggregationLogWriter the new file is: log/node1###agg.log
AggregationLogWriter the new file is: log/node1###agg.log
PSS Show View: false
Started a gossip thread: pss1-Active
Started a gossip thread: pss1-Passive
Started a gossip thread: AggActive
Started a gossip thread: AggPassive
```

Le prime righe visualizzano le impostazioni adottate per l'attivazione, mentre più interessante è la vista iniziale del nodo comprensiva di 7 elementi. Ogni riga di questa corrisponde ad un *NodeDescriptor* che è formato: dall'identificatore *node_id*, dal nome, dall'indirizzo IP e dall'età. Infine sono mostrati i messaggi di successo per l'avvio dei servizi di Peer Sampling Service e Aggregazione.
Il paradigma JRMI di connessione remota prevede che le parti coinvolte nella comunicazione si registrino presso un server chiamato rmiregistry. Ogni nodo tenta inizialmente di registrare il proprio gestore di rete presso un server già esistente, e in caso di fallimento avvia una nuova istanza dell'rmiregistry.
È bene sottolineare che questo server non deve mai essere terminato per tutta la durata dei test, pena l'impossibilità dei collegamenti.
Il procedimento di avvio appena illustrato deve quindi essere ripetuto per tutti gli altri nodi. In alternativa possiamo sfruttare: “clusterNodes.sh” che aiuta l'utente nell'avvio in locale di un numero arbitrario di nodi. La sintassi del comando è la seguente:

```
$./clusterNodes <nodeName_1> [nodeName_2 ... nodeName_N]
```

Compilazione e avvio nel laboratorio dell'Università di Bologna

L'avvio dei nodi nel laboratorio didattico dell'Università di Bologna si differenzia dal caso su macchina locale solo per quanto riguarda la compilazione e l'avvio di un nodo. Per compilare il progetto occorre specificare l'opzione “laboratory” al comando ant:

```
$ant laboratory
```

Quindi è possibile procedere con la creazione degli script delle API con lo scritp “createSHScripts.sh”. Per avviare un nodo occorre specificare l'opzione

"-lab" durante l'avvio al posto di "-n". Dal momento che ogni macchina del laboratorio possiede un particolare nome non abbiamo più la ncessità di specificarlo tramite un paramtro. L'opzione "-lab" inizializza la vista con un insieme di macchine che appartengono al laboratorio:

```
$./startNode -lab
```

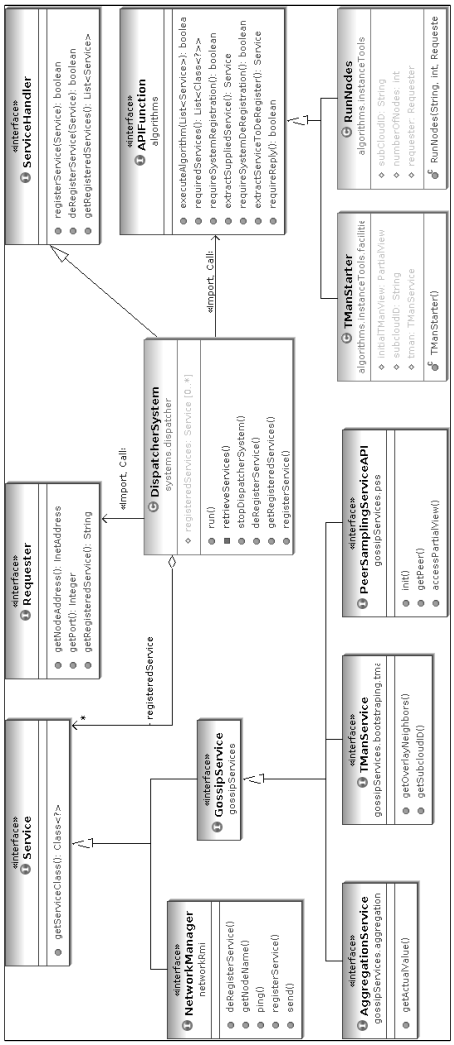

Figura 5.6: Diagramma di classe che rappresenta l’architettura dei servizi e degli algoritmi delle funzioni API. Ogni servizio viene registrato nel *`DispatcherSystem`* per essere eventualmente sfruttato dall’algoritmo di una API dell’infrastruttura.

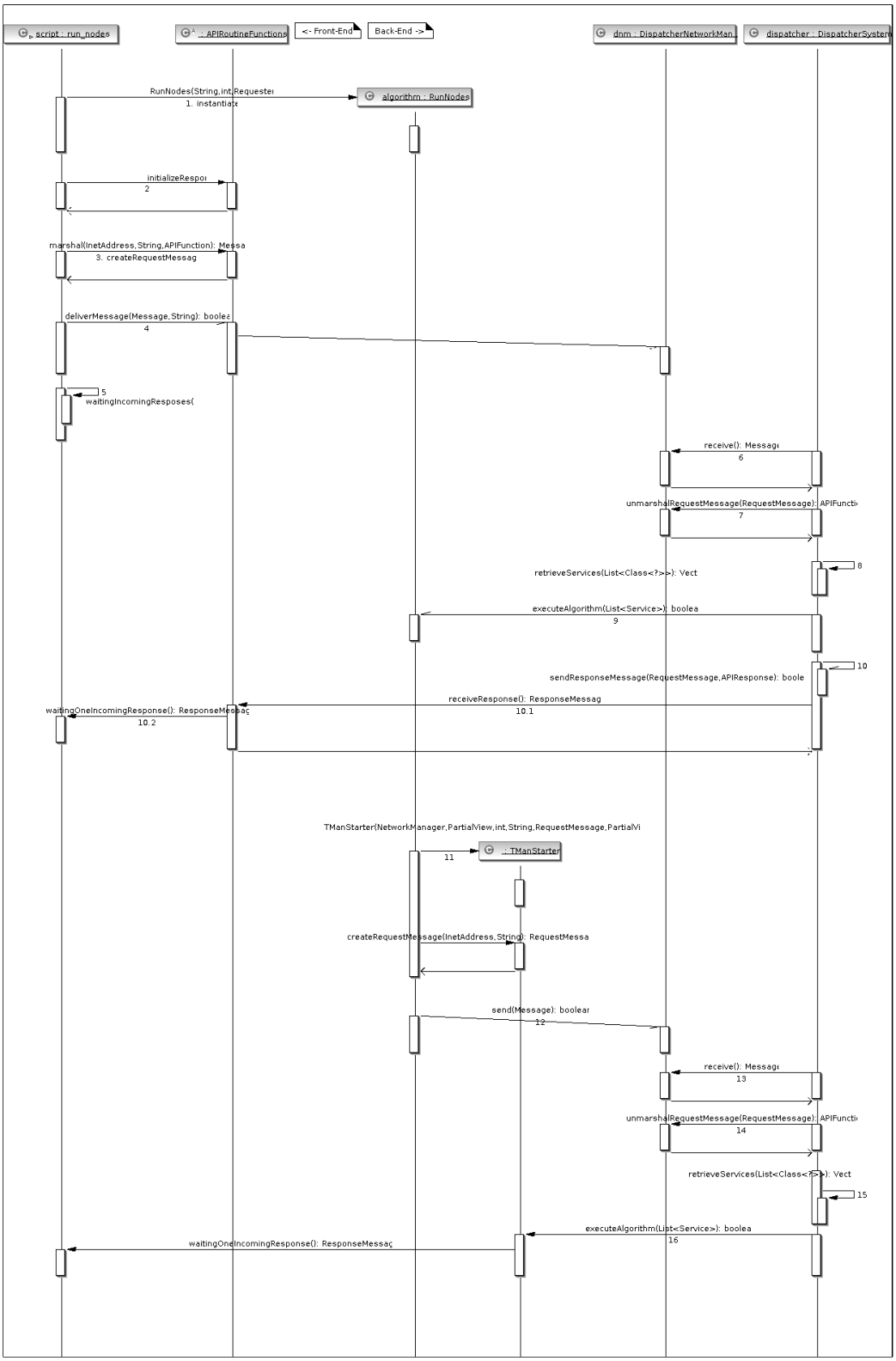

Figura 5.7: Diagramma di sequenza che mostra le azioni che ricorrono nel momento in cui l'utente richiede la creazione di un overlay attraverso la richiesta di **run-nodes**.

5.6 Valutazioni sperimentali

In questa sezione analizziamo il comportamento del prototipo.
Concentriamoci innanzitutto su uno dei componenti basilari: il Peer Sampling Service. È piuttosto difficile collezionare materiale in grado di produrre un'adeguata documentazione sul funzionamento di questo modulo in un ambiente distribuito. Tuttavia l'opzione “-vv”, con la quale è possibile avviare un nodo, visualizza sul terminale la vista parziale aiutandoci così a ricavare alcune osservazioni. Ho effettuato tutte le prove mantenendo una vista di 7 `NodeDescriptor`, con PSS impostato in modalità *push/pull*, estrazione *random* con il metodo *getPeer()*, e parametri *Healed* e *Swapped*, che agiscono sulla selezione della vista, pari a H = S = 2. (Per una descrizione dei parametri in ingresso al servizio si veda l'appendice A.)

Ho utilizzato fino a 20 nodi nel laboratorio e fino a 40 in locale sulla macchina personale. La rete mostra fin da questi esigui numeri tutte le ottime qualità citate in [36]: i nodi non più raggiungibili vengono via via eliminati dalle viste dei vicini mentre quelli nuovi sono gradualmente accolti nel grafo. Anche in condizioni disastrose con quasi il 50% di nodi terminati, la rete riprende completamente le comunicazioni. Dal momento che la vista viene visulizzata ad ogni modifica, la velocità di aggiornamento del terminale produce qualche informazione sulla dinamicità della rete. Più elevata è la frequenza di modifica sulla vista e maggiore sarà il numero di interazioni alle quali un nodo è soggetto. All'aumentare del numero di nodi accresce il livello di dinamicità della rete, rendendola estremamente reattiva all'inserimento o rimozione dei nodi.

Vediamo ora un esempio di utilizzo delle API citate in 5.1. Il nostro obiettivo sarà quello di realizzare una *subcloud* di 5 elementi in una rete con 10 nodi, prima in locale e poi nel laboratorio. Seguendo i passi visti nella sezione 5.5 avviamo 10 nodi che denominiamo: “node1”, “node2”, ...,“node10”. Ora eseguiamo la funzione **run-nodes** che ritroviamo nella directory “API-scripts” con gli opportuni parametri, cioè il nome della subcloud ed il numero di nodi, specificando anche da quale nodo stiamo effettuando la richiesta:

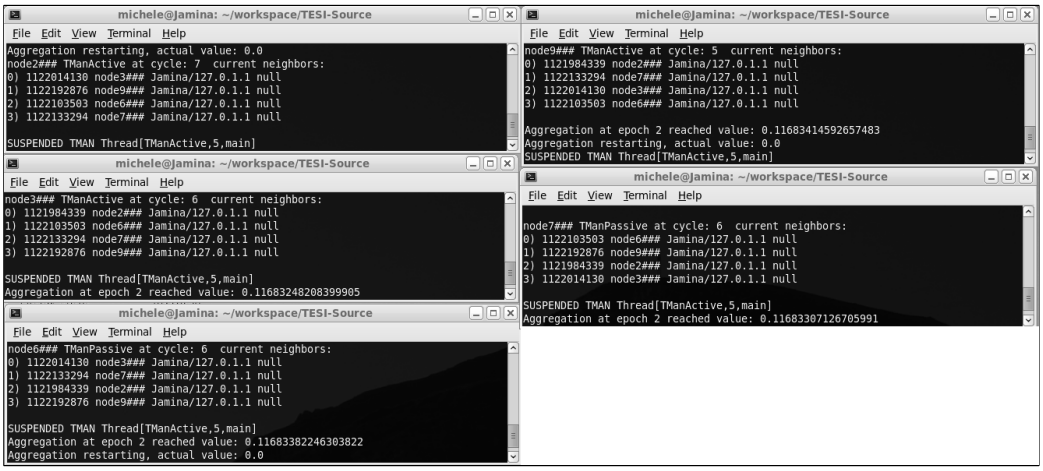

Figura 5.8: Risultato dell'esecuzione della API **run-nodes** sulla macchina locale: sono mostrati i terminali dei nodi coinvolti nella subcloud. È possibile notare la terminazione del protocollo di T-Man, che ha creato un anello, e la visualizzazione dei vicini. I vicini più prossimi al nodo risiedono nelle posizione di indice minore.

```
$./run-nodes.sh -n node3 firstSubcloud 5
```

In questo modo richiediamo al sistema la creazione di una subcloud denominata "firstSubcloud" nella quale saranno inseriti 5 nodi e sottomettiamo la richiesta da "node3". La figura 5.8 mostra i terminali dei nodi selezionati per formare l'overlay ovvero: 2, 3, 6, 7, 9. Nell'output sono compresi anche i messagi del servizio di Aggregazione, che si immettono per via della concorrenza tra i thread attivati. Possiamo riconoscere a quale macchina appartiene un terminale osservando la prima riga. Appartiene ad esempio a "node2" il primo riquadro in alto a sinistra, infatti la prima riga è formata da:

```
node2### TManActive at cycle: 7 current neightbors:
```

che indica il nodo sul quale è attiva l'istanza del protocollo T-Man e il numero del ciclo appena compiuto. Nelle righe che seguono sono illustrati i vicini che

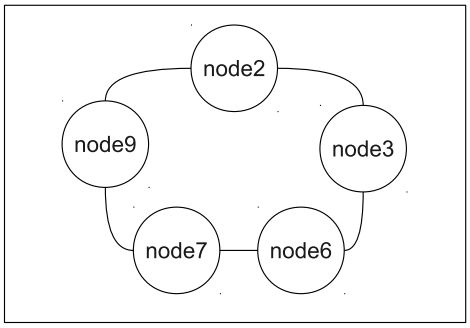

Figura 5.9: Rappresentazione dell overlay creato da T-Man che ha generato un anello tra i nodi coinvolti.

sono raccolti nella vista interna al servizio: scorrendola dall'alto verso il basso otteniamo in ordine quelli più prossimi. Nel caso dell'anello in particolare il primo ed il secondo `NodeDescriptor` (posizione 0 e 1) costituiscono il vicino destro e sinistro del nodo in questione, mentre i successivi corrispondono agli archi ridondanti in una direzione e dall'altra. Volendo osservare una rappresentazione di quanto accaduto possiamo considerare la figura 5.9 che mostra l'anello costituito dagli archi principali.

Ripetiamo ora le stesse azioni ma nel laboratorio didattico dell'Università di Bologna. Da una rete di 10 macchine abbiamo richiesto la creazione di una subcloud chiamata *myLabSubcloud* per la quale sono stati selezionati: *altoum*, *amonasro*, *doncurzio*, *morales* e *remendado*. Analogamente al caso precedente ho riportato l'output dei cinque terminali in figura 5.10. In questo caso è leggermente più complicato indovinare la disposizione dei nodi per via della diversa nomenclatura. Consideriamo il primo riquadro in alto a sinistra che rappresenta la situazione di *morales*: i vicini più prossimi sono *remendado* e *doncurzio*.

Nel caso di *remendado*, nel secondo riquadro dall'alto a sinistra, le prime due posizioni sono occupate da *amonasro* e *morales*. Continuando questa osservazione per tutti i terminali presentati possiamo ricondurre la forma dell'overlay all'anello riportato in figura 5.11.

Spingiamoci oltre per testare il funzionamento anche di **terminate-nodes**,

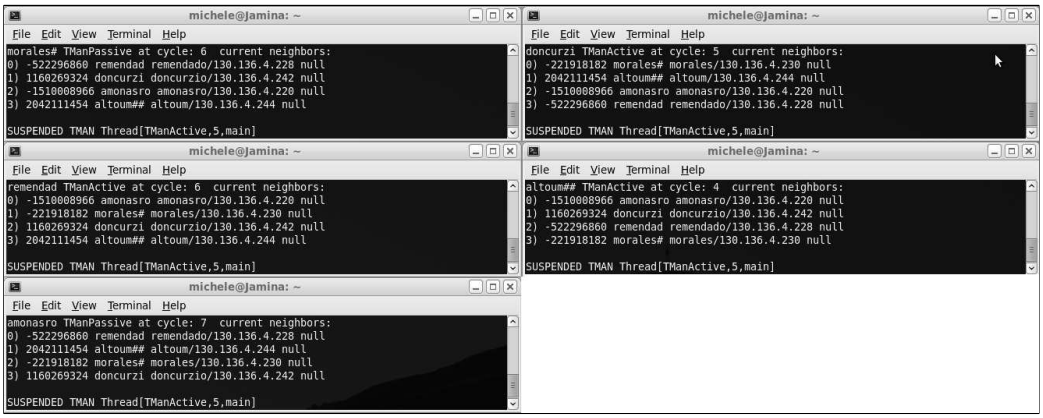

Figura 5.10: Risultato dell'esecuzione della API **run-nodes** all'interno del laboratorio didattico dell'Università di Bologna.

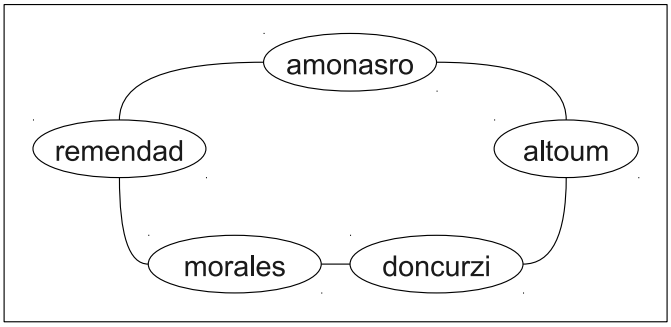

Figura 5.11: Rappresentazione dell'anello generato da T-Man tra i nodi selezionati nella rete creata in laboratorio.

che in questo caso rimuove dalla subcloud i nodi inseriti pur mantenendo attivi gli altri servizi di Aggregazione e PSS. Occorre quindi utilizzare un nuovo terminale per connetterci ad una macchina appartenente alla sottonuvola ed avviare lo script terminate-nodes:

```
$ssh mtamburi@altoum.cs.unibo.it
$cd source/API-scripts
$./teminate-nodes myLabSubcloud amonasro
```

Con questa istruzione abbiamo ordinato la rimozione del nodo *amonasro* dal-

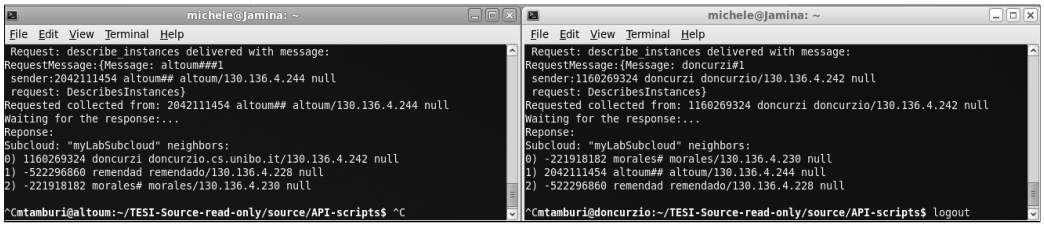

Figura 5.12: Output dei terminali dei due nodi *altoum* e *doncurzio* in seguito all’esecuzione di **describe-instances** per verificare che le viste siano state correttamente aggiornate.

la subcloud *myLabSubcloud*. Riceveremo un messaggio di successo da parte di *altoum* dal quale abbiamo sottomesso la richiesta. La rimozione di un nodo comporta la notifica di eliminazione a tutti gli appartenenti all’overlay, in modo che ciascuno possa poi riordinare la propria vista. In modo analogo opera anche **add-nodes**, che contrariamente aggiunge nuove macchine ad una subcloud.
Dobbiamo però ora verificare che gli aggiornamenti delle viste siano avvenuti correttamente. Ci serviamo della API **describe-instances**. Questa funzione deve essere sottomessa da un nodo appartenente ad una subcloud e restituisce il nome identificativo dell’overlay con l’elenco dei vicini. Viene invece sollevato un messaggio di errore se sulla macchina non è attivato alcun servizio di bootstraping. Scegliamo arbitrariamente le due macchine di *altoum* e *doncurzio*, connettiamoci ed avviamo su ciascuna lo script describe-instances. Otterremo l’output mostrato in figura 5.12: il riquadro sinistro illustra la situazione di *altoum*, mentre quello destro quella di *doncurzio*. Notiamo subito che entrambi hanno correttamente rimosso *amonasro* dalle loro viste. Inoltre hanno eseguito il riordinamento adeguando l’overlay alla nuova configurazione, riportata in figura 5.13, nella quale il nuovo vicino prossimo di *altoum* è ora *remendado*.

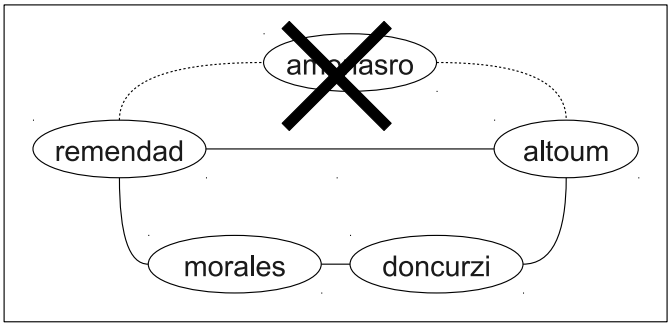

Figura 5.13: Anello risultante dalla rimozione di *amonasro* dall'overlay precedentemente generato.

Conclusioni

Nel corso di questo lavoro abbiamo cercato di esaminare un concetto piuttosto giovane dell'Informatica, come il Cloud Computing, da un punto di vista non solo teorico ma anche sperimentale. Questo studio si inserisce infatti nel progetto di costruzione di un sistema Cloud di tipo P2P, in grado di fornire servizi agli utilizzatori. Il contributo di questo elaborato consiste nel definire una prima proposta per indagare le possibilità di raggiungere lo scopo finale, ed eventualmente avanzare una strategia implementativa.

In una prima fase di studio, abbiamo esposto una definizione capace di catalogare le Cloud in relazione ai sistemi Grid e di esporne i diversi modelli: di dispiegamento (*pubbliche*, *private*, *ibride*), e basati sui servizi (*IaaS*, *PaaS*, *SaaS*). Importante ai fini della nostra ricerca è stata la stesura di un elenco riguardante le difficoltà implementative che ostacolano la realizzazione di una Cloud.
Nell'esplorazione dello stato dell'arte abbiamo selezionato alcuni sistemi dei quali abbiamo poi analizzato le architetture a diversi livelli di dettaglio. Questo ci ha permesso di confrontarne l'insieme delle API.
Abbiamo poi condotto un'analisi sugli utilizzatori delle Cloud e su alcuni casi di studio, in modo da maturare una maggiore consapevolezza sulle figure coinvolte nella realizzazione e utilizzo del sistema.
In fase di progettazione abbiamo quindi ideato la stesura di un prototipo per una Cloud P2P privata di livello Infrastruttura, che utilizza protocolli di gossip all'interno del laboratorio didattico: P2P Cloud System (P2PCS).

L'obiettivo di questo primo passo è quello di sfruttare le macchine inoperose per lavori che vengono inoltrati dalle persone frequentanti l'ambiente accademico. In questo modo, oltre a mantenere sotto osservazione il sistema, avremo un'immediata risposta sulla soddisfazione degli utenti.
Abbiamo provveduto infine allo sviluppo di un prototipo che comprende un insieme minimale delle funzioni del progetto finale, del quale sono esposti i punti più rilevanti dell'implementazione.

Con il lavoro presentato vogliamo offrire una proposta per la realizzazione del progetto finale che prende in esame sia l'aspetto progettuale, nella definizione dei requisisti, dell'architettura e delle principali funzioni di interfaccia, che quello implementativo. Abbiamo infatti sviluppato un sistema capace di: sfruttare algoritmi di gossip per la costruzione di una Cloud di piccole dimensioni, mantenere monitorato il numero di nodi, essere in condizioni di reagire autonomamente ai guasti e generare degli overlay tra sottoinsiemi di macchine.

I passi futuri vedono il completamento del modello presentato con lo sviluppo di tutte le funzionalità caratteristiche di un sistema infrastrutturale. In primo luogo l'utilizzo di un ambiente in grado di avviare macchine virtuale, quindi procedure per la sicurezza e l'autenticazione degli utenti. Occorre inoltre potenziare il monitoraggio e la gestione delle risorse, scendendo al dettaglio delle caratteristiche delle macchine come potenza CPU, memoria utilizzata o carico di lavoro. Il punto finale sarà quello di porre un utilizzatore in condizione di sfruttare tutte le potenzialità offerte dall'infrastruttura del laboratorio.

Appendice A

Algoritmi in pseudocodice

In questa appendice elenco lo pseudocodice dei protocolli di gossip relativi ai servizi inseriti nel prototipo: Peer Sampling Service, Aggregazione e T-Man. Nell'implementazione ho cercato di attenermi scrupolosamente alla sequenza di istruzioni descritta nelle fonti. La descrizione che segue sottolinea semplicemente i punti più rilevanti nell'implementazione di ciascun algoritmo. Il lettore interessato ad una più approfondita compresione può fare riferimento direttamente ai documenti riportati in bibliografia.
Compare inoltre l'attuale algoritmo del DispatcherSystem, che potrebbe subire sostanziali modifiche negli sviluppi futuri.

Il Peer Sampling Service dell'algoritmo 1 crea un grafo tra le macchine della rete, la cui topologia tende ad un "random graph". Ciascun nodo mantiene la propria vista parziale *view*, sulla quale agiscono concorrentemente sia il thread attivo che passivo. Questa è formata da una lista di oggetti `NodeDescriptor` che identificano le macchine contenendone il nome, l'identificativo univoco, l'indirizzo IP, e l'eta. I parametri in ingresso al protocollo sono:

- *peerSelection*: indica il metodo di selezione dei peer dalla vista nell'istruzione 4. Sono previste due modalità: `rand` che estrae con distri-

Algorithm 1 Peer Sampling Service [36]

```
1: function Active Thread
2: loop
3:    wait(T time units);
4:    p ← view.selectPeer();
5:    if push then
6:       //0 is the initial age
7:       buffer ←((MyAddress,0));
8:       view.permute();
9:       move oldest H items to the end of view;
10:      buffer.append(view.head(c/2 − 1));
11:      send buffer to p;
12:   else
13:      //empty view to trigger response
14:      send (NULL) to p;
15:   end if
16:   if pull then
17:      receive buffer_p from p;
18:      view.select(c,H,S,buffer_p);
19:   end if
20:   view.increaseAge();
21: end loop
22: end function

23: function Passive Thread
24: loop
25:   receive buffer_p from p;
26:   if pull then
27:      //0 is the initial age
28:      buffer ← ((MyAddress, 0));
29:      view.permute();
30:      move oldest H items to the end of view;
31:      buffer.append(view.head(c/2 − 1));
32:      send buffer to p;
33:   end if
34:   view.select(c,H,S,buffer_p);
35:   view.increaseAge();
36: end loop
37: end function

38: method view.sleelct(c,H,S,buffer_p)
39: view.append(buffer_p);
40: view.removeDuplicates();
41: view.removeOldItems(min(H,view.size−c));
42: view.removeHead(min(S,view.size−c));
43: view.removeAtRandom(view.size−c);
44: end method
```

buzione casuale uniforme un nodo, e `tail` che rimuove la coda della lista;

- c: dimensione della vista. Nonostante le continue modifiche, la vista conserva una dimensione costante. Rappresenta l'insieme degli archi uscenti di un nodo;

- H ("*Healed*"): numero di `NodeDescriptor` che vengono spostati in fondo alla vista nelle istruzioni di permutazione (righe 8, 29). La vista non viene mai ordinata secondo l'eta, ma la permutazione evita che gli elementi più vecchi vengano propagati agli altri nodi. Condizione: $0 \leq H \leq \frac{c}{2}$;

- S ("*Swapped*"): numero di `NodeDescriptor` che vengono prelevati dal messaggio in ingresso $buffer_p$ ed inseriti nella vista locale $view$. Condizione: $0 \leq S \leq \frac{c}{2} - H$;

- *push pull*: definiscono le modalità di *propagazione* della vista tra i nodi. Danno luogo a due tipologie: *push* nella quale i messaggi vengono solo inviati verso altri, e *push/pull* che prevede l'invio delle risposte ai messaggi in arrivo dando luogo a scambi di porzioni di vista tra le macchine.

L'istruzione 3 del thread attivo scandisce lo scorrere dei cicli del protocollo. L'eta di viene inizializzata a 0 e cresce con l'avanzare del tempo (righe 20, 35). Ogni nodo costruisce un messaggio $buffer$ che contiene il proprio `NodeDescriptor` (righe 7, 35) ed una porzione della vista dalla quale sono esclusi gli elementi più vecchi in seguito alla permutazione (righe 10, 31). In questo modo ogni macchina corretta propaga elementi "freschi" del proprio identificatore, mentre quelle non più raggiungibili vengono via via eliminate. Nel momento in cui viene ricevuto un messaggio $buffer_p$ da un nodo p (righe 17, 25), questo viene passato alla funzione *select* (righe 18, 34) che ne aggiunge il contenuto all'attuale vista e la modifica rimuovendo i duplicati, gli elementi più vecchi e mantenendo la dimensione c invariata.

Algorithm 2 Aggregazione [33].

1: **function** Active Thread
2: **do** exactly once in each consecutive δ time units
   at a randomly picked time
3: $q \leftarrow PeerSamplingService$.getNeighbor();
4: send s_p to q;
5: $s_q \leftarrow$ receive(q);
6: $s_p \leftarrow$ `UPDATE`(s_p, s_q);
7: **end function**

8: **function** Passive Thread
9: **loop**
10: $s_q \leftarrow$ receive(*);
11: send s_p to q;
12: $s_p \leftarrow$ `UPDATE`(s_p, s_q);
13: **end loop**
14: **end function**

Dove `UPDATE`(s_p, s_q) si riferisce alla specifica funzione di aggregazione.

L'algoritmo 2 presenta lo pseudocodice dell'Aggregazione [33]. Il protocollo è in grado di ricavare il valore globale di un attributo, che viene mantenuto da ciascuna macchina in una variabile s condivisa tra thread attivo e passivo. I nodi p e q conservano rispettivamente s_p ed s_q, che corrispondono alla stima dell'attributo ricercato. L'istruzione `UPDATE` delle righe 6, 12 aggiorna il valore della variabile locale applicando la funzione scelta per l'aggiornamento, che deve essere nota a priori a tutti i nodi. Per il calcolo della media, ad esempio, viene eseguita la formula $\frac{s_p+s_q}{2}$, mentre per la ricerca del valore massimo o minimo si possono adottare le consuete funzioni di $min(s_p, s_q)$ e $max(s_p, s_q)$.

L'algorimo 3 corrisponde invece ad una versione dell'Aggregazione più vicina al codice sorgente implementato, nella quale compaiono le istruzioni che controllano la sincronizzazione attraverso la gestione delle *epoche*. Allo scadere di ogni epoca la variabile s viene nuovamente impostata al valore iniziale. In questo modo si rende l'algoritmo robusto ai guasti ed all'ingresso di nuove macchine.

Algorithm 3 rappresenta il servizio di Aggregazione presentato con l'algoritmo 2 nel quale sono state inserite le istruzioni per la gestione delle *epoche* e la sincronizzazione.

```
1: function Active Thread
2: do each cycle
3: q ← PeerSamplingService.getNeighbor();
4: msg_p ← (MyDescriptor,s_p)
5: send msg_p to q;
6: msg_q ← receive(q);
7: if msg_q != null then
8:    exchange ← epochCheck(MyEpoch, msg_q.epoch)
9:    if exchange == true then
10:      s_p ← UPDATE(s_p, msg_q.s_q);
11:      elapsedCycle + +
12:   else
13:      synchronizeEpoch(msg_q)
14:   end if
15: end if
16: end function

17: function Passive Thread
18: loop
19:   msg_q ← receive(*);
20:   exchange ← checkEpoch(MyEpoch, msg_q.epoch)
21:   if exchange == true then
22:      msg_p ← (MyDescriptor, s_p)
23:      send msg_p to q;
24:      s_p ← UPDATE(s_p, s_q);
25:      elapsedCycle + +
26:   else
27:      synchronizeEpoch(msg_q)
28:   end if
29: end loop
30: end function
```

L'algoritmo 4 riporta lo pseudocodice di T-Man [35], un protocollo di bootstraping che genera un overlay tra i nodi selezionati nella rete. Si basa su un meccanismo che classifica gli elementi della vista di un nodo secondo un criterio di preferenza, dettato da una funzione `rank`, che deve essere definita a priori. Per la sua esecuzione sono necessari alcuni parametri in ingresso:

- Δ: lunghezza di un ciclo (riga 3);
- ψ: ciascun nodo seleziona i vicini coi quali interagire selezionandoli dai ψ elementi con la maggior preferenza (riga 4);
- ***m***: definisce il numero massimo di *`NodeDescriptor`* che può essere inserito in un messaggio (righe 7, 17);
- `rank(`*node*`,` *view*`)`: funzione che agisce sulla vista in ingresso *view* applicando l'algoritmo di costruzione della topologia prendendo come punto di riferimento il nodo inserito *node*.

L'implementazione dell'algoritmo ha subito una modifica. Per poter riutilizzare lo stesso codice sia per l'estrazione che per l'inserimento di nodi da una subcloud, le due chiamate alla funzione `rank` delle righe 6 e 16, sono state spostate dopo l'unione della vista locale *view* con la porzione in ingresso $buffer_p$ o $buffer_q$. In questo modo ciascun nodo esegue la classificazione della vista in base al proprio identificativo e come ultima modifica. L'obiettivo è di impedire a messaggi obsoleti in arrivo di corrompere l'overlay durante le operazioni di aggiunta o rimozione di macchine.

L'algoritmo 5 introduce, sottoforma di pseudocodice, il comportamento del *`DispatcherSystem`*. Questo modulo deve accogliere i messaggi di richiesta, estrarre i servizi che questa necessita, spedire un messaggio di conferma al nodo dal quale è stata inoltrata la richiesta e procedere con l'esecuzione della funzione API.

Algorithm 4 T-Man [35]

```
1: function Active Thread
2: loop
3:    wait(Δ);
4:    p ← selecPeer( ψ, rank(myDescriptor, view) );
5:    buffer ← merge(view, myDescriptor);
6:    buffer ← rank(p, buffer);
7:    send first m entries of buffer to p;
8:    receive buffer_p from p
9:    view ← merge(buffer_p, view);
10: end loop
11: end function

12: function Passive Thread
13: loop
14:    receive buffer_q from q
15:    buffer ← merge(view, myDescriptor);
16:    buffer ← rank(q, buffer);
17:    send first m entries of buffer to q;
18:    view ← merge(buffer_q, view);
19: end loop
20: end function
```

Algorithm 5 Algoritmo del modulo DispatcherSystem.

```
1: function Dispatcher System
2: loop
3:    receive msgRequest_n from n;
4:    request ← msgRequest_n.getRequest();
5:    services ← retrieveRequiredServices(request.getRequiredSerices());
6:    msgResponse ← composeResponseMessage(n);
7:    send msgResponse to n ;
8:    request.executeAlgorithm(services);
9:    if request.hasToRegister() then
10:      registerNewService(request.getService());
11:   end if
12: end loop
13: end function
```

Bibliografia

[1] *Amazon Elastic Compute Cloud: Developer Guide*, API Version 2010-11-15.

[2] Amazon web services documentation. `http://aws.amazon.com/documentation/`.

[3] Eucalyptus. website. `http://open.eucalyptus.com/`.

[4] NIST: National Institute of Standard and Technology. website. `http://www.nist.gov/index.html`.

[5] NIST: National Institute of Standard and Technology - Important Actors for Public Clouds. website. `http://www.nist.gov/itl/cloud/actors.cfm`.

[6] kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Ottawa, Ontario Canada, June 27-30 2007.

[7] *LADIS 2009: The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Kornell University, October 10-11 2009. `http://www.cs.cornell.edu/projects/ladis2009/index.htm`.

[8] Alan Cooper, Robert Reimann and David Cronin. *About Face 3, The Essential of Interacion Design.* Wiley, 10475 Crosspoint Boulevard Indianapolis, 2007.

[9] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID*, pages 4–10, 2004. `http://boinc.berkeley.edu/`.

[10] O. Babaoglu and M. Jelasity. Self-∗ properties through gossiping. In *Philosophical Transactions A of the Royal Society*, volume 366, pages 3747–3757. October 2008.

[11] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, David P. Anderson. Discovering Statistical Models of Availability in Large-Scale Distributed Systems: An Empirical Study of SETI@home. In *17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, London,UK, September 2009.

[12] Miguel L. Bote-Lorenzo, Yannis A. Dimitriadis, and Eduardo Gómez-Sánchez. Grid characteristics and uses: A grid definition. In *European Across Grids Conference*, pages 291–298, 2003.

[13] Justin Cappos, Ivan Beschastnikh, Arvind Krishnamurthy, and Tom Anderson. Seattle: a platform for educational cloud computing. In *SIGCSE*, pages 111–115, 2009.

[14] Cisco. Independent review organization builds private cloud. White Paper, 2010.

[15] Cisco. Cloud: What an enterprise must know. white paper, 2011.

[16] Microsft Co. Azure services platform. URL. `http://www.microsoft.com/windowsazure/`.

[17] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, Boston, MA, USA, 2001.

[18] Erin Cody, Raj Sharman, Raghav H. Rao, and Shambhu Upadhyaya. Security in grid computing: A review and synthesis. *Decis. Support Syst.*, 44:749–764, March 2008.

[19] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Volunteer computing and desktop cloud: The cloud@home paradigm. *Network Computing and Applications, IEEE International Symposium on*, 0:134–139, 2009.

[20] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Applying software engineering principles for designing cloud@home. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 618–624, Washington, DC, USA, 2010. IEEE Computer Society.

[21] Dell. Dell cloud computing solutions. `http://www.dell.com/cloudcomputing`.

[22] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello and David Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *18th International Heterogeneity in Computing Workshop*. IEEE, may 2009.

[23] John Foley. Cloud interoperability? amazon and microsoft play nice. website, March, 31 2009. `http://www.informationweek.com/blog/229207385`.

[24] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[25] George Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems concepts and design*. Addison-Wesley, fourth edition, 2005.

[26] Luis Miguel Vaquero Gonzalez, Luis Rodero-Merino, Juan Caceres, and Maik A. Lindner. A break in the clouds: towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[27] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical report, Georgia Institute of Technology, College of Computing, April 2009.

[28] Ian T. Foster and Yong Zhao and Ioan Raicu and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *CoRR*, abs/0901.0131, 2009.

[29] Amazon Inc. Elastic compute cloud. URL, Nov 2008. `http://aws.amazon.com/ec2/`.

[30] Google Inc. Giigke application engine. URL. `http://code.google.com/appengine/`.

[31] IBM Inc. Blue cloud project. URL, June 2008. `http://www-03.ibm.com/press/us/en/pressrelease/22613.wss`.

[32] Márk Jelasity and Anne-Marie Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing*, pages 117–124, 2006.

[33] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[34] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. The bootstrapping service. In *ICDCS Workshops*, page 11, 2006.

[35] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[36] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[37] Ke Xu, Meina Song, Xiaoqi Zhang, Junde Song. A cloud computing platform based on p2p. In *IT in Medicine & Education, 2009. ITIME '09. IEEE International Symposium on*, pages 427–432. Jinan, August 2009. Inspec Accession Number: 10868983.

[38] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville. Cloud migration: A case study of migrating an enterprise it system to iaas. *CoRR*, abs/1002.3492, 2010.

[39] George Lawton. Addressing the challenge of cloud-computing interoperability. website, Sept 2009. `http://www.computer.org/portal/web/computingnow/archive/news031`.

[40] Lenk, A.; Klems, M.; Nimis, J.; Tai, S.; Sandholm, T.; . What's inside the cloud? an architectural map of the cloud landscape. In *Software Engineering Challenges of Cloud Computing, 2009. CLOUD '09. ICSE Workshop on*, pages 23–31, Vancouver, BC, 23 May 2009.

[41] Markus Klems, Jens Nimis and Stefan Tai. Do clouds compute? a framework for estimating the value of cloud computing. In *Designing E-Business Systems. Markets, Services, and Networks*, volume 22, pages 110–123, 2009. Lecture Notes in Business Information Processing.

[42] Michael Armbrust, Armando Fox, Rean Griffith, Antony D. Joseph, Randy Katz, Andy Kowinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Electrical Engineering and Computer Sciences University of California at Berkeley, 10 February 2009. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[43] Sun Microsystem. Network.com. URL. `http://www.sun.com/solutions/cloudcomputing/`.

[44] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131, 2009.

[45] O. Babaoglu, M. Jelasity, A.-M. Kermarrec, A. Montresor and M. van Steen. Managing clouds: A case for a fresh look at large unreliable dynamic networks. In *ACM SIGOPS Operating Systems Review (Special Issue on Self-Organizing Systems)*, volume 40, July 2006.

[46] R. Buyya, Chee Shin Yeo, S. Venugopal,. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, Dalian, 25-27 Sept. 2008. IEEE.

[47] Rajiv Ranjan, Liang Zhao, Xiaomin Wu, Anna Liu, Andres Quiroz and Manish Parashar. Peer-to-peer cloud provisioning: Service discovery and load-balancing. In *Computer Communications and Networks*, volume 0, pages 195–217, 2010.

[48] Rajkumar Buyya, Chee Shin Yeoa, Srikumar Venugopala, James Broberga, and Ivona Brandicc. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. In *Future Generation Computer Systems*, volume 25, pages 599–616. Elsevier B.V, June 2009. Issue 6.

[49] Rimal, B.P.; Eunmi Choi; Lumb, I.;. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, Seul, 25-27 Aug 2009.

[50] Matt Rosoff. Inside amazon's cloud disaster. website, Apr, 22 2011. http://www.businessinsider.com/amazon-outage-enters-its-second-day-lots-of-sites-still-down-201 1 - 4.

[51] Jan Sacha, Jeff Napper, Corina Stratan, and Guillaume Pierre. Adam2: Reliable distribution estimation in decentralised environments. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2010.

[52] Peter Sempolinski and Douglas Thain. A comparison and critique of eucalyptus, opennebula and nimbus. In *CloudCom*, pages 417–426, 2010.

[53] Simon Caton and Omer Rana. Towards autonomic management for cloud services based upon volunteered resources. *Concurrency and Computation: Practice and Experience*, 9 Mar 2011.

[54] Borja Sotomayor, Rubén S. Montero, Ignacio Martín Llorente, and Ian T. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[55] Kunwadee Sripanidkulchai, Sambit Sahu, Yaoping Ruan, Anees Shaikh, and Chitra Dorai. Are clouds ready for large distributed applications? *Operating Systems Review*, 44(2):18–23, 2010.

[56] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485. Springer, 2005. Lecture Notes in Computer Science.

[57] Corina Stratan, Jan Sacha, Jeff Napper, and Guillaume Pierre. Coordinated self-adaptation in large-scale peer-to-peer overlays. Technical Report IR-CS-60, Vrije Universiteit, Amsterdam, The Netherlands, September 2010. `http://www.globule.org/publi/CSALSPTPO_ircs60.html`.

[58] James Urquhart. Exploring cloud interoperability, part 1. website, May, 2 2009. `http://news.cnet.com/8301-19413_3-10231290-240.html`.

[59] Jinesh Varia. Architecting for the cloud: Best practices. white paper, Jan 2010.

[60] Mladen A. Vouk. Cloud computing: Issues, research and implementations. In *Proceedings of the ITI 2008 30th Int. Conf. on Information Technology Interfaces*, pages 31–40, Cavtat,Croatia, June 23-26 2008.

[61] Wikipedia. Ovf, open virtualization format. website. `http://en.wikipedia.org/wiki/Open_Virtualization_Format`.

[62] Wikipedia. Volunteer computing. website. `http://en.wikipedia.org/wiki/Volunteer_computing`.

[63] Zhijia Chen and Yang Zhao and Chuang Lin and Xin Miao and Qingbo Wang. P2p accelerated mass data distribution over booming internet: Effectiveness and bottlenecks. In *ICDCS Workshops*, pages 239–244, 2009.

Ringraziamenti

Deridero ringraziare innanzitutto il professor Moreno Marzolla, per avermi condotto in questo lavoro offrendomi supervisione e preziosi consigli, aver sopportato i miei frequenti errori, ed aver portato pazienza per le mie tediose email.

Ringrazio di cuore i miei genitori, Luciana Mutti e Tiziano Tamburini, ai quali è dedicata questa tesi. Nonostante i momenti difficili mi hanno sempre offerto il loro supporto ed incoraggiato nelle mie scelte. Senza di loro non avrei potuto intraprendere questa carriera universitaria.

In questi due anni di corso Magistrale ho avuto modo di collaborare con colleghi eccezzionali: Luca Mandrioli (Mandro), che ha rappresentato un punto fisso in qualunque lavoro di gruppo, Marco Patrignani (Squera), al quale faccio i migliori auguri per la carriera accademica, e Diego Bonfigli, che ho conosciuto solo in un progetto ma è stato sufficiente per apprezzarne subito le abilità di programmatore.
Un abbraccio a Stefania Prudente, fedele collega di interminabili ore di studio per affrontare l'esame di Algoritmi Avanzati. Purtroppo non ho mai avuto il piacere di consegnare un progetto insieme a Mattia Lambertini, col quale ho trascorso piacevolissime pause pranzo e ore di lezione.

Ringrazio in modo particolare Andrea Rappini (Rappo), venuto improvvisamente a mancare il 21-03-2011. Con lui ho trascorso numerose ore di studio e di collaborazione in progetti, ho condiviso momenti al di fuori dell'ambito lavorativo che mi hanno fatto apprezzare quale grande persona fosse. Descriverlo come un ragazzo splendido è comunque limitativo e ricordarlo con affetto credo sia il minimo. Un forte abbraccio anche a Chiara, alla quale

auguro un futuro sereno.

Tengo a ringraziare anche il nostro gruppo musicale: i Blaus. In questi anni abbiamo superato momenti difficili e nonostante siano cambiati alcuni volti non è ancora scomparsa la voglia di suonare e stare bene insieme. Ringrazio tutti, anche coloro che ad oggi hanno scelto di non farne più parte, perché le prove della domenica sera mi hanno dato occasione di non abbandonare completamente la musica.

Non posso certo tralasciare Giulia Cacciari e Simona Paladino, che oltre a condividere il ruolo di educatrici del gruppo Giovanissimi con il sottoscritto, mi accompagnano in molte attività della nostra piccola parrocchia.

Infine voglio ringraziare tutti coloro che, nonostante mi sia fatto di fumo nell'ultimo periodo, continuano a chiamarmi amico. Senza di loro sarebbe molto più difficile digerire le continue frustrazioni che il mondo dell'Informatica riserva.