ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

CLOUD COMPUTING: ANALISI DEI MODELLI ARCHITETTURALI E DELLE TECNOLOGIE PER LO SVILUPPO DI APPLICAZIONI

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
MARCO DE CANAL

Relatore:
Prof. ALESSANDRO RICCI

Co-relatori:
Prof. ANTONIO NATALI
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2011–2012
SESSIONE II

PAROLE CHIAVE

Cloud Computing

Architecture

SaaS

Pattern

Cloud Application

Ai miei ragazzi che, senza saperlo, mi sono stati di grande aiuto

Indice

Introduzione **xiii**

1 Cloud Computing: una panoramica **1**

1.1 Caratteristiche del Cloud . . . 1
1.2 Modelli di servizi . . . 2
1.2.1 Infrastructure as a Service . . . 2
1.2.2 Platform as a Service . . . 3
1.2.3 Software as a Service . . . 4
1.3 Modelli di Cloud . . . 5
1.3.1 Public Cloud . . . 6
1.3.2 Private Cloud . . . 6
1.3.3 Community Cloud . . . 7
1.3.4 Hybrid Cloud . . . 7
1.4 Architettura generale del Cloud . . . 8
1.4.1 Virtualizzazione . . . 9
1.4.2 Architettura di rete . . . 11
1.4.3 Architettura di scalabilità . . . 12
1.5 Cloud OS . . . 13
1.5.1 Scopo del Cloud OS . . . 14
1.5.2 Componenti . . . 15
1.6 Proprietà architetturali del Cloud . . . 15
1.6.1 Proprietà delle piattaforme virtualizzate . . . 16
1.6.2 Proprietà delle piattaforme Cloud . . . 17

2 Cloud Operating System **21**

2.1 Analogie con Desktop OS e modularizzazione . . . 22
2.2 Virtual Machine Manager . . . 23

2.3 Network Manager 24
2.4 Storage Manager 24
2.5 Image Manager 25
2.6 Information Manager 26
2.7 Federation Manager 26
2.8 Scheduler 27
2.9 Service Manager 28
2.10 Interfacce 29
2.11 Autenticazione e autorizzazione 30
2.12 Accounting ed auditing 31
2.13 Livello di accoppiamento di Cloud 31
2.14 Architetture di federazione 33
2.15 Breve analisi di Cloud OS esistenti 35
2.15.1 OpenStack 35
2.15.2 Eucalyptus 37
2.15.3 OpenNebula 39

3 Un esempio di IaaS: Amazon Web Services (AWS) 41
3.1 Benefici derivanti dall'uso di AWS 43
3.1.1 Benefici economici 43
3.1.2 Benefici tecnici 45
3.2 Elasticità secondo AWS 46
3.2.1 Confronto con approcci diversi 47
3.2.2 Considerazioni 49
3.3 I servizi di AWS 50
3.3.1 *Amazon Elastic Compute Cloud* 50
3.3.2 *Elastic IP* e *Amazon Elastic Block Storage* 51
3.3.3 *Amazon CloudWatch* e *Auto-scaling* 51
3.3.4 *Elastic Load Balancing* 51
3.3.5 *Amazon Simple Storage Service* 52
3.3.6 *Amazon CloudFront* 52
3.3.7 *Amazon SimpleDB* 52
3.3.8 *Amazon Relational Database Service* 53
3.3.9 *Amazon Simple Queue Service* 53
3.3.10 *Amazon Simple Notifications Service* 53
3.3.11 *Amazon Elastic MapReduce* 53
3.3.12 *Amazon Virtual Private Cloud* 53

3.3.13 *Amazon Route53* 54
3.3.14 *AWS Identity and Access Management* 54

4 Persistenza e scalabilità dei dati - Bigtable 55
4.1 Panoramica generale 55
4.2 Introduzione a Bigtable 57
4.3 Il modello dei dati 57
4.3.1 Righe 58
4.3.2 Famiglie di colonne 59
4.3.3 *Timestamp* 61
4.4 API 61
4.5 Un esempio concreto: Google Earth 63
4.6 Note conclusive 64

5 Un esempio di PaaS: Google App Engine 67
5.1 Introduzione al *PaaS* 67
5.1.1 Limiti dell'*IaaS* 67
5.1.2 Scopo ed orientamento dell'*IaaS* 68
5.1.3 Introduzione ed obbiettivi del *PaaS* 68
5.2 Google Cloud Platform 70
5.3 Google App Engine 70
5.3.1 Vantaggi di Google App Engine 71
5.3.2 Linguaggi ed ambiente di esecuzione 72
5.3.3 Servizi ed API 72
5.3.4 Amministrazione del sistema 73
5.4 Altri servizi Cloud 74
5.4.1 Google Cloud Storage 74
5.4.2 Google Compute Engine 75
5.4.3 Google Cloud SQL 76
5.4.4 Google BigQuery 77
5.5 Conclusioni 78

6 SaaS - Sviluppo di applicazioni Cloud 79
6.1 Definizione di applicazioni Cloud 79
6.2 Principi generali per sviluppare applicazioni in ambiente Cloud 81
6.2.1 Processo di sviluppo del software 81
6.2.2 Influenza della piattaforma sulle applicazioni 82
6.2.3 Consapevolezza di caratteristiche fondamentali 83

6.2.4 Gestione e riduzione del traffico 85
6.3 Pattern architetturali 89
6.3.1 Definizione e scopo dei pattern 89
6.3.2 Necessità di pattern nel Cloud Computing 89
6.3.3 Applicazioni modulari 90
6.3.4 Accoppiamento debole 91
6.3.5 Componenti *stateless* 92
6.3.6 Componenti a singola istanza 93
6.3.7 Componenti a singola istanza configurabili 94
6.3.8 Componenti idempotenti 95
6.3.9 Map Reduce 96
6.3.10 Tipi di Cloud e modelli di servizi 97
6.4 Sviluppo di applicazioni in AWS 97
6.4.1 Considerare la possibilità di errori 98
6.4.2 Favorire il disaccoppiamento dei componenti 100
6.4.3 Implementare l'elasticità 103
6.4.4 Sfruttare il parallelismo 105
6.4.5 Posizionare i dati opportunamente 106
6.5 Sviluppo di applicazioni con Google App Engine 107
6.5.1 Differenze nella documentazione 107
6.5.2 Sviluppo di applicazioni Cloud 108
6.6 Conclusioni 110

7 Nuovi modelli e paradigmi - Orleans 113
7.1 Il modello ad attori 114
7.1.1 Introduzione 114
7.1.2 Gli attori 114
7.1.3 Programmi ad attori 115
7.1.4 Sincronizzazione 116
7.1.5 Pattern di programmazione parallela 118
7.1.6 Proprietà del modello ad attori 120
7.2 Orleans 120
7.2.1 *Grain* 121
7.2.2 *Activation* 122
7.2.3 *Promise* 123
7.2.4 Esecuzione di *Activation* 124
7.2.5 Persistenza dello stato 125

7.2.6 Transazioni 126
7.2.7 Meccanismi e politiche di scalabilità 127
7.2.8 Interfacce dei *grain* 127
7.2.9 Riferimenti ai *grain* 128
7.2.10 Creazione ed uso dei *grain* 128
7.2.11 Classi dei *grain* 129
7.2.12 Confronto con il modello ad attori 130
7.3 Conclusioni 131

8 Conclusioni **133**

Introduzione

Uno dei temi più discussi ed interessanti nel mondo dell'informatica al giorno d'oggi è sicuramente il *Cloud Computing*. Nuove organizzazione che offrono servizi di questo tipo stanno nascendo ovunque e molte aziende oggi desiderano imparare ad utilizzarli, migrando i loro centri di dati e le loro applicazioni nel Cloud. Ciò sta avvenendo anche grazie alla spinta sempre più forte che stanno imprimendo le grandi compagnie nella comunità informatica: Google, Amazon, Microsoft, Apple e tante altre ancora parlano sempre più frequentemente di *Cloud Computing* e si stanno a loro volta ristrutturando profondamente per poter offrire servizi Cloud adeguandosi così a questo grande cambiamento che sta avvenendo nel settore dell'informatica.

Tuttavia il grande movimento di energie, capitali, investimenti ed interesse che l'avvento del *Cloud Computing* sta causando non aiuta a comprendere in realtà che cosa esso sia, al punto tale che oggi non ne esiste ancora una definizione univoca e condivisa. La grande pressione inoltre che esso subisce da parte del mondo del mercato fa sì che molte delle sue più peculiari caratteristiche, dal punto di vista dell'ingegneria del software, vengano nascoste e soverchiate da altre sue proprietà, architetturalmente meno importanti, ma con un più grande impatto sul pubblico di potenziali clienti.

L'obbiettivo che ci poniamo con questa tesi è quindi quello di esplorare il nascente mondo del *Cloud Computing*, cercando di comprenderne a fondo le principali caratteristiche architetturali e focalizzando l'attenzione in particolare sullo sviluppo di applicazioni in ambiente Cloud, processo che sotto alcuni aspetti si differenzia molto dallo sviluppo orientato ad ambienti più classici.

La tesi è così strutturata: nel primo capitolo verrà fornita una panoramica sul *Cloud Computing* nella quale saranno date anche le prime definizioni e verranno esposti tutti i temi fondamentali sviluppati nei capitoli successivi. Il secondo capitolo costituisce un approfondimento su un argomen-

to specifico, quello dei *Cloud Operating System*, componenti fondamentali che permettono di trasformare una qualunque infrastruttura informatica in un'infrastruttura Cloud. Essi verranno presentati anche per mezzo di molte analogie con i classici sistemi operativi desktop. Con il terzo capitolo ci si addentra più a fondo nel cuore del *Cloud Computing*, studiandone il livello chiamato *Infrastructure as a Service* tramite un esempio concreto di Cloud provider: Amazon, che fornisce i suoi servizi nel progetto *Amazon Web Services*. A questo punto, più volte nel corso della trattazione di vari temi saremo stati costretti ad affrontare le problematiche relative alla gestione di enormi moli di dati, che spesso sono il punto centrale di molte applicazioni Cloud. Ci è parso quindi importante approfondire questo argomento in un capitolo appositamente dedicato, il quarto, supportando anche in questo caso la trattazione teorica con un esempio concreto: *BigTable*, il sistema di Google per la gestione della memorizzazione di grandi quantità di dati. Dopo questo intermezzo, la trattazione procede risalendo lungo i livelli dell'architettura Cloud, ricalcando anche quella che è stata l'evoluzione temporale del *Cloud Computing*: nel quinto capitolo, dal livello *Infrastructure as a Service* si passa quindi a quello *Platform as a Service*, tramite lo studio dei servizi offerti da *Google Cloud Platform*. Il sesto capitolo costituisce invece il punto centrale della tesi, quello che ne soddisfa l'obbiettivo principale: esso contiene infatti uno studio approfondito sullo sviluppo di applicazioni orientate all'ambiente Cloud. Infine, il settimo capitolo si pone come un ponte verso possibili sviluppi futuri, analizzando quali sono i limiti principali delle tecnologie, dei modelli e dei linguaggi che oggi supportano il *Cloud Computing*. In esso viene proposto come possibile soluzione il modello ad attori; inoltre viene anche presentato il framework *Orleans*, che Microsoft sta sviluppando negli ultimi anni con lo scopo appunto di supportare lo sviluppo di applicazioni in ambiente Cloud.

Capitolo 1

Cloud Computing: una panoramica

1.1 Caratteristiche del Cloud

Il *Cloud Computing* permette a chi ne usufruisce di rilasciare ed accedere a servizi e distribuire applicazioni tramite un'infrastruttura messa a disposizione da un'organizzazione che si occupa di mantenerla e gestirla. Tale organizzazione è detta *Cloud provider*. Ciò che rende interessante l'utilizzo di tali tecnologie da parte dei clienti dei cloud provider e che caratterizza il Cloud come particolare classe di sistema distribuito può essere riassunto nelle seguenti caratteristiche:

- possibilità di non doversi occupare della gestione dell'infrastruttura fisica, che è affidata al cloud provider;

- elasticità, definita come scalabilità automatica ed altamente dinamica, che permette alle applicazioni Cloud di scalare, sia espandendosi che riducendosi, eventualmente anche di vari ordini di grandezza, in funzione del carico a cui tali applicazioni sono sottoposte;

- modello di pagamento *pay-per-use*, che nasce da un accurato monitoraggio delle risorse usate da ciascun utente, e che permette a quest'ultimo di non dover pagare per risorse che spesso non vengono utilizzate pienamente, come succede nel caso di noleggio o acquisto di server, sia virtuali che reali;

- omogeneità dovuta ad un ambiente di virtualizzazione condiviso che permette di nascondere le differenze, sia a livello di hardware che di software, che possono esistere tra i vari componenti utilizzati per realizzare concretamente l'architettura del sistema;

Volendo cercare di dare una definizione di cosa sia il Cloud, potremmo affermare che esso è un ambiente costituito da un'infrastruttura altamente dinamica tramite la quale le risorse sono distribuite come servizi (*'as a Service'*).

1.2 Modelli di servizi

I cloud provider tipicamente organizzano i servizi da loro forniti classificandoli in tre categorie: *Infrastructure as a Service* (Iaas), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS). Queste tre classi di servizi spesso vengono strutturati secondo un'architettura a livelli:

- il livello più inferiore (*IaaS*) si occupa di fornire server, memorie di massa ed infrastrutture di rete virtualizzando l'hardware reale;
- il livello intermedio (*PaaS*) fornisce l'ambiente e gli strumenti più ad alto livello necessari per la realizzazione e l'esecuzione delle applicazioni, appoggiandosi all'infrastruttura del livello sottostante;
- il livello più alto (*SaaS*) è costituito dalle applicazioni vere e proprie e dai servizi di cui esse necessitano.

Un ulteriore categoria di servizi sta recentemente cominciando ad emergere: *Composite as a Service* (CaaS). Questi servizi dovrebbero permettere agli utenti di comporre gli altri servizi offerti dal provider nella maniera a loro più congeniale: tale categoria si appoggerebbe quindi su tutte e tre le altre classi sopra elencate, richiedendone esplicitamente l'esecuzione dei relativi servizi.

1.2.1 Infrastructure as a Service

Obbiettivo

L'idea che sta alla base di questo tipo di servizio è che i server virtuali di un'infrastruttura elastica dovrebbero essere offerti ad utenti differenti, i

quali devono restare isolati gli uni dagli altri, sia per quanto riguarda i loro dati che le performance. Riuscire ad ottenere un buon livello di isolamento può tuttavia risultare complesso, perché in realtà gli utenti condividono l'infrastruttura fisica che si trova sotto lo strato virtualizzato, costituita da componenti di rete, unità di memorizzazione di massa e server.

Inoltre deve essere adottato il modello di pagamento *pay-per-use*: ciò implica la capacità di monitorare e raccogliere informazioni dettagliate riguardanti le risorse utilizzate dagli utenti per poter allocare a ciascuno di essi il reale costo di utilizzo.

Soluzione

É necessario effettuare un controllo degli accessi, autenticando gli utenti e monitorandone l'uso delle API durante la loro attività di gestione dei server virtuali. Inoltre occorre potenziare i componenti del sistema che si occupano della gestione delle risorse in modo da impedire che il loro eccessivo utilizzo da parte di un utente porti a dei significativi cali nelle performance degli altri utenti. Ovviamente occorre anche potenziare gli aspetti legati alla sicurezza del sistema, per impedire agli utenti di condividere le loro informazioni private con altri, anche a fronte di attacchi diretti.

1.2.2 Platform as a Service

Obbiettivo

L'obbiettivo del *PaaS* è quello di ospitare i componenti software creati da utenti diversi in un ambiente di esecuzione condiviso, il quale offre loro le funzionalità più comuni ed usate.

Per poter far ciò è necessario potenziare ulteriormente le politiche di isolamento, il quale non deve essere più garantito solo a livello di macchine virtuali, ma a livello dei componenti, che possono trovarsi ad essere ospitati ed eseguiti sullo stesso server.

La piattaforma stessa deve quindi diventare consapevole di ospitare più utenti e deve essere in grado di riconoscerli, in modo da poter impedire, quando opportuno, che componenti di un determinato utente accedano a dati e funzionalità di componenti sviluppati da altri. Quando invece è necessario essa deve poter offrire la possibilità a tali componenti di accedere

vicendevolmente ai propri servizi, comunicare tra loro ed usare funzionalità di memorizzazione di massa messe a disposizione dalla piattaforma.

Soluzione

Tali requisiti vengono soddisfatti fornendo agli utenti delle API che permettano loro di distribuire i propri componenti presso una piattaforma, e di registrare e configurare per i propri componenti i servizi di comunicazione e memorizzazione di massa. Tali servizi devono inoltre essere potenziati per garantire agli utenti delle performance equivalenti.

Anche il controllo dell'accesso deve essere potenziato, poiché non è più sufficiente riconoscere gli utenti, ma occorre che sia esteso ai singoli componenti, per garantirne l'isolamento.

Servizi di piattaforma

Esempi di servizi di comunicazione usati tipicamente in un ambiente Cloud sono lo scambio di messaggi, l'invocazione di metodi e la notifica di eventi.

I servizi di memorizzazione di massa invece vengono quasi sempre offerti come:

Database relazionali in grado di gestire informazioni molto strutturate che supportino operazioni anche molto complesse;

Database non relazionali usati quando occorre memorizzare grandi moli di dati sui quali non vengono però eseguite operazioni particolarmente complesse;

Blocchi di memoria tipicamente impiegati per gestire dati molto poco strutturati, spesso nell'ambito degli stream multimediali.

1.2.3 Software as a Service

Obbiettivo

Il concetto attorno al quale si sviluppa il *SaaS* è la possibilità di offrire lo stesso Software tramite la rete ad utenti diversi e isolati. É importante fare in modo che ogni utente abbia la percezione di essere l'unico utilizzatore

del Software in questione, senza tuttavia replicare le istanze dell'applicazione, poiché spesso è un approccio infattibile. L'applicazione stessa deve essere consapevole di avere più utenti diversi: solo in questo modo i suoi componenti possono essere condivisi mantenendo comunque l'isolamento.

Inoltre, in determinati casi, potrebbe essere necessario che l'utente sia in grado di configurare l'applicazione in modo che risponda al meglio ai propri bisogni. Semplici esempi di configurabilità possono essere il formato delle date e la valuta utilizzata, oppure la posizione o il contenuto dei menu di un'interfaccia grafica.

Soluzione

Ancora una volta la soluzione può essere trovata potenziando il controllo dell'accesso, che può avvenire in questo caso o con delle API o tramite interfaccia grafica. La configurazione personalizzata dell'applicazione viene mantenuta in una base di dati, la quale stabilisce quindi come i vari componenti applicativi si comportano.

Tipicamente, se l'accesso è consentito tramite interfaccia grafica, significa che il Software viene utilizzato dagli utenti tramite un Web Browser, mentre se è consentito grazie a delle API, probabilmente l'applicazione è esterna al Cloud, ma utilizza alcuni servizi che vengono distribuiti tramite esso; ad esempio possiamo pensare ad un'applicazione Web che viene eseguita su server proprietari dell'azienda, quindi non appartenenti al Cloud, ma che si appoggia ai suoi servizi di mailing.

1.3 Modelli di Cloud

Tipicamente vengono identificate quattro categorie di Cloud, alcune delle quali si discostano leggermente dalla descrizione fornita nella sezione introduttiva di questo capitolo, ma comunque tutte in grado di fornire i principali vantaggi legati all'architettura Cloud. Tali categorie sono:

- Public Cloud
- Private Cloud
- Community Cloud
- Hybrid Cloud

1.3.1 Public Cloud

Si tratta del tipo più comune di Cloud, che si adatta meglio degli altri alla descrizione iniziale. In una Public Cloud tipicamente un'organizzazione si occupa di implementare, gestire e mantenere tutta l'infrastruttura, dai componenti hardware, come server e dispositivi di rete, a quelli software, come load balancer, monitor e software per la virtualizzazione. Tale organizzazione poi, sulla base dei propri piani tariffari, tipicamente pay-per-use, affitta le proprie risorse ad altre organizzazioni o utenti, che solitamente, in ambiente Cloud, vengono chiamati *tenant*.

1.3.2 Private Cloud

Questo particolare tipo di Cloud viene utilizzato quando specifici requisiti legati alla privacy o restrizioni derivanti dall'ambito legislativo rendono impossibile o sconsigliabile l'uso di una Public Cloud.

In questo caso l'infrastruttura Cloud non viene condivisa, ma è realizzata all'interno di un'unica organizzazione. Il cloud provider, se presente, si occupa dell'installazione e mantenimento del Cloud sui dispositivi appartenenti all'organizzazione committente. I servizi offerti dal Cloud sono quindi accessibili solo da quest'ultima, garantendo il massimo livello di sicurezza e privacy, a scapito tuttavia della dinamicità tipica di questi sistemi poiché i picchi di lavoro non possono essere gestiti utilizzando risorse che altre organizzazioni in quel momento non stanno impiegando, come avviene nello scenario tipico.

Centralizzazione

Solitamente all'interno delle organizzazioni, l'adozione di una Private Cloud comporta una centralizzazione dei servizi informatici, in modo da ottenere i migliori vantaggi dalle economie di scala. Inoltre anche lo sviluppo ed il mantenimento di applicazioni possono beneficiare notevolmente della condivisione di risorse e della standardizzazione introdotta dall'ambiente virtualizzato distribuito.

Virtual Private Cloud

Una possibile variazione di questa soluzione, quando privacy e sicurezza lo permettono, consiste nella possibilità di non realizzare fisicamente il Cloud con i dispositivi dell'organizzazione, ma appoggiarsi ad una Public Cloud realizzando una Private Cloud virtuale. Essa, a differenza della Private Cloud di cui sopra, presenta una maggiore dinamicità, potendo scalare oltre i limiti dell'organizzazione, sfruttando appieno le potenzialità della Public Cloud, perdendo tuttavia le grandi garanzie di sicurezza che un Cloud fisicamente isolato offre.

1.3.3 Community Cloud

Questo tipo di Cloud può essere pensato e realizzato a partire da una Private Cloud o da una Public Cloud, sebbene anche in questo scenario siano talvolta presenti i vincoli di sicurezza e privacy che, come visto precedentemente, impediscono o sconsigliano quest'ultimo approccio. Ciò che più caratterizza questo tipo di Cloud è l'esistenza di un insieme di organizzazioni che si fidano le une delle altre e che spesso hanno bisogno di condividere informazioni, o più in generale, risorse per il tipo di attività che svolgono.

Poiché sia le Public Cloud che le Private Cloud dispongono di meccanismi di isolamento per impedire la comunicazione tra determinate risorse che risiedono in esse, è possibile in questo modo realizzare una Community Cloud come una porzione isolata di Cloud dei primi due tipi. In alternativa è possibile realizzare la Community Cloud come un ambiente computazionale completamente isolato.

Le Community Cloud contengono al loro interno tutti i servizi e le informazioni che le organizzazioni devono usare congiuntamente per poter svolgere le loro attività.

1.3.4 Hybrid Cloud

Questo tipo di Cloud nasce come combinazione tra Private Cloud e Public Cloud. Lo scenario tipico a cui si applicano comprende sia una parte di servizi, risorse e informazioni che determinate organizzazioni non vogliono condividere con altre per ragioni di sicurezza, sia una parte che invece hanno interesse a condividere per ridurre i costi necessari al loro uso e per fornire loro una maggiore dinamicità.

Configurazione tipica

La soluzione tipicamente messa in atto consiste nel considerare la Public Cloud come un'estensione della Private Cloud, alla quale si ricorre quando i picchi nel carico di lavoro rendono necessarie più risorse; ciò solitamente avviene perché la Private Cloud, se interna all'organizzazione, non ha dei costi legati all'uso ma solo al mantenimento, che verrebbero comunque pagati anche se non fosse utilizzata al meglio. Durante i periodi in cui il carico di lavoro è nella media tutti i servizi, le informazioni memorizzate e la computazione vengono mantenuti interni alla Private Cloud; a fronte di picchi invece, stabiliti quali servizi, informazioni e computazioni hanno i vincoli di privacy e sicurezza meno restrittivi, questi ultimi vengono fatti migrare nella Public Cloud.

Varietà nelle soluzioni

Ovviamente questa configurazione può essere soggetta a molte variazioni. Non è raro ad esempio che un'azienda decida di appoggiarsi direttamente ai servizi di storage di una Public Cloud per la memorizzazione delle informazioni meno sensibili, oppure potrebbe essere necessario esporre certi servizi direttamente nella Public Cloud, proprio perché essi devono essere resi pubblici ad altri utenti ed organizzazioni.

Altre soluzioni potrebbero essere messe in campo invece quando fossero presenti o necessarie le Community Cloud.

Molte configurazioni diverse possono quindi essere pensate ed utilizzate per realizzare una Hybrid Cloud, tuttavia ciò che contraddistingue sempre tale categoria di Cloud è l'integrazione tra Cloud di tipo diverso, per raggiungere una struttura che si adegui al meglio alle necessità.

1.4 Architettura generale del Cloud

Per introdurre l'architettura di riferimento di un Cloud ne affrontiamo in questa sezione tutti gli aspetti più significativi, quali la virtualizzazione, l'infrastruttura di rete e le scelte architetturali volte a garantire la scalabilità, come esposto anche in [7].

1.4.1 Virtualizzazione

Le tecnologie di virtualizzazione permettono di astrarre dai componenti reali fornendone una rappresentazione virtuale, la quale interagisce con le risorse reali in maniera il più possibile trasparente per gli utilizzatori. Analogamente ad un sistema operativo, il quale fornisce un'astrazione dei componenti fisici di un computer garantendo all'utente un accesso di più alto livello, la virtualizzazione permette a chi ne usufruisce di utilizzare risorse tramite un'interfaccia nota, senza sapere quali risorse reali siano presenti dietro lo strato di virtualizzazione. Possono essere virtualizzate sia risorse hardware, come CPU, memoria volatile, memoria di massa o interi computer, sia risorse software, come i sistemi operativi.

I componenti fondamentali su cui si basano le tecnologie di virtualizzazione sono due:

Virtual Machine è la rappresentazione virtuale di un intero calcolatore, dotata di tutto l'hardware che si ritiene necessario, e funziona da contenitore logico del sistema operativo ospite; può essere memorizzata come immagine del disco rigido del computer, più alcune meta-informazioni, come le risorse disponibili e le loro caratteristiche; è interessante notare per l'ambito del Cloud Computing che una macchina virtuale può essere spostata da un server a un altro.

Hypervisor chiamato anche Virtual Machine Manager (VMM), è il componente che gestisce i sistemi operativi ospiti in esecuzione su un server fisico, e presenta loro una vista virtualizzata delle risorse hardware fisiche.

Hypervisor

Gli Hypervisor possono essere classificati in due categorie: nativi, chiamati anche *bare-metal hypervisor*, o ospiti.

Gli Hypervisor nativi vengono eseguiti direttamente sull'hardware fisico, nella modalità di esecuzione delle istruzioni più privilegiata. Essi hanno quindi accesso e controllo diretto alle risorse fisiche della macchina sottostante. Esempi di Hypervisor nativi sono ESXi di VMware[1], Hyper-V di

[1]http://www.vmware.com/it/products/datacenter-virtualization/vsphere/esxi-and-esx/overview.html

Microsoft[2] o Xen Hypervisor[3], un progetto open source usato da molti Cloud provider, come Amazon[4] e Rackspace[5].

Gli Hypervisor ospiti invece vengono eseguiti su un sistema operativo, che è a sua volta in esecuzione sull'hardware fisico. Risultano chiaramente meno performanti di quelli nativi, ragione per cui non vengono usati nell'ambito Cloud, ed hanno, in generale, scopi diversi. La loro principale utilità è legata al fatto che permettono di utilizzare contemporaneamente sia il sistema operativo ospite che quello ospitante. Non vengono usati quindi per garantire isolamento, scalabilità e multi-utenza di sistemi, quanto piuttosto per fornire funzionalità tipiche di sistemi operativi diversi al sistema operativo ospitante e per avere un maggiore controllo del sistema operativo ospitato. Esempi di Hypervisor ospiti sono i vari prodotti di VMware[6] per sistemi desktop, Oracle VirtualBox[7], Windows Virtual PC[8] o Parallels Workstation[9].

Tipi di virtualizzazione

Esistono vari modi diversi di supportare la virtualizzazione, che si distinguono per il livello di astrazione e le performance fornite:

Virtualizzazione completa : detta anche *full virtualization*, in questo caso si ha una totale virtualizzazione ed astrazione dell'hardware; il sistema operativo ospitato non deve essere cambiato e tutte le sue chiamate di sistema vengono intercettate dall'Hypervisor ed emulate con istruzioni speciali.

Questo approccio è il più generico e flessibile poiché separa completamente il sistema operativo ospitato dall'hardware sottostante, tuttavia porta ad un sovraccarico computazionale che influisce negativamente sulle performance.

[2] http://www.microsoft.com/it-it/server-cloud/windows-server/hyper-v.aspx
[3] http://www.xen.org/products/xenhyp.html
[4] http://aws.amazon.com
[5] http://www.rackspace.com
[6] http://www.vmware.com
[7] https://www.virtualbox.org
[8] http://www.microsoft.com/italy/windows/virtual-pc/default.aspx
[9] http://www.parallels.com/eu/landingpage/dskd67-6

Paravirtualization : questo approccio richiede invece delle modifiche al sistema operativo ospitato, poiché le chiamate di sistema devono essere sostituite da chiamate all'Hypervisor, il quale non virtualizza completamente l'hardware, ma ne offre un'interfaccia simile a quella reale.

Il vantaggio di questa soluzione consiste nel fatto che hardware funzionalmente simili ma che offrono interfacce diverse, obbligando i sistemi operativi ad adeguarsi a ciascuna di esse, vengono invece uniformati dall'interfaccia comune dell'Hypervisor. Inoltre le performance sono migliori, non essendo più necessaria l'emulazione delle chiamate di sistema. Tuttavia il disaccoppiamento tra hardware e sistema operativo non è netto come nel caso della *full virtualization* e c'è bisogno di cambiare il codice del sistema operativo.

Hardware-assisted virtualization : questa è una soluzione che permette di realizzare la virtualizzazione supportandola direttamente dall'hardware. Questa categoria è più difficile da definire perché, mentre alcuni lo ritengono un vero e proprio approccio diverso alla virtualizzazione, che si distingue quindi dalla *full virtualization* e dalla *paravirtualization*, altri lo intendono più come un supporto tecnologico alle due modalità di virtualizzazione di cui sopra; in entrambi i casi esso implica comunque l'adozione di specifiche soluzioni hardware per aumentare le performance dei sistemi virtualizzati. Ad esempio Intel Virtualization Technology (Intel VT[10]) implementa il supporto hardware alla virtualizzazione accelerando il passaggio di controllo tra il sistema operativo ospitante e quello ospitato, permettendo di assegnare alcuni dispositivi di input o output unicamente al sistema ospitato ed ottimizzando l'uso delle reti.

1.4.2 Architettura di rete

Una tipica architettura fisica di rete di un sistema Cloud si compone di un vasto numero di calcolatori, componenti di memorizzazione di massa e dispositivi di rete, collegati tra loro in maniera strutturata, formando solitamente dei *cluster* in modo da supportare architetture di rete virtualizzate in grado di adattarsi al contesto applicativo specifico.

[10]http://ark.intel.com/Products/VirtualizationTechnology

Come esempio possiamo citare l'architettura tipica di un'applicazione Web, che adotta il modello client-server a più livelli. Essa si compone solitamente di un server Web, incaricato di interagire con i client degli utenti fornendo loro una rappresentazione delle informazioni secondo gli standard dell'HTML5; questo a sua volta si rivolge come client al server applicativo, che contiene e gestisce la *business logic* caratteristica dell'applicazione e non permette agli utenti di rivolgersi direttamente ad esso, ma accetta connessioni solamente provenienti dagli sviluppatori; infine il server dei dati, che ha come client il server applicativo, al quale quest'ultimo si rivolge per ottenere servizi di memorizzazione persistente delle informazioni. La figura 1.1 mostra quanto appena esposto.

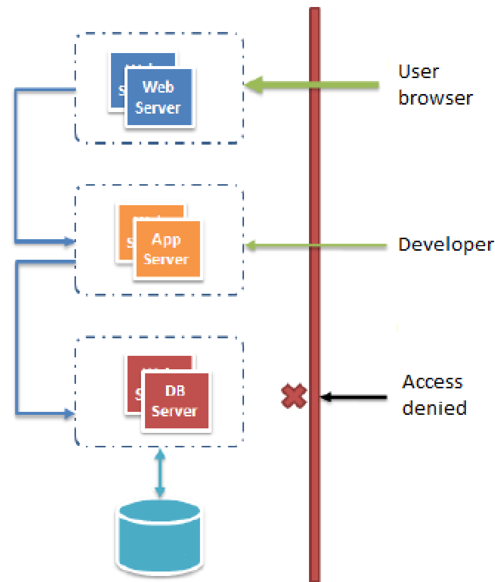

Figura 1.1: Architettura tipica di un'applicazione Web

1.4.3 Architettura di scalabilità

Per poter garantire la scalabilità del sistema, la sua architettura deve essere costruita in modo da poter utilizzare più calcolatori in grado di condividere

il carico di lavoro e fornire un grande livello di disponibilità.

Tali architetture, organizzate anch'esse su più livelli, sono tipicamente costituite da vari elementi, tra i quali si trovano il *Load Balancer*, che ha il compito specifico di ripartire equamente il carico di lavoro tra i server, una *server farm* funzionante come *memcache*, ovvero una memoria volatile distribuita, implementata solitamente come tabella hash distribuita, e dei server incaricati di gestire la memorizzazione persistente dei dati, appoggiandosi ai dispositivi fisici ma fornendo una rappresentazione ed un modello dei dati differenti da quelli concretamente utilizzati e nascondendo ai propri client i dettagli legati a questioni come la tolleranza ai guasti e la replicazione.

I dati costituiscono proprio il punto critico di tali architetture, poiché spesso rappresentano il collo di bottiglia delle applicazioni. Per risolvere questo problema, è possibile attuare politiche di partizionamento, *caching* e replicazione, in modo da distribuire i dati su più server, ripartendo in tal modo anche il carico di lavoro. Tuttavia questo fa sorgere la necessità di ulteriore traffico e computazione per riuscire a gestire la consistenza dei dati, mantenendo sincronizzate le varie copie. Nasce quindi anche il bisogno di definire ed individuare il miglior modello di consistenza applicabile all'applicazione, il quale a sua volta potrebbe influire sul modello dei dati. Come abbiamo già potuto osservare nella sezione riguardante il *PaaS*, infatti, il modello relazionale non è più l'unico che viene messo a disposizione. Queste questioni verranno affrontate nel dettaglio nel quarto capitolo.

Un'ultima osservazione che è opportuno fare parlando di architetture di scalabilità geograficamente molto distribuite è che, per poter ridurre la latenza dovuta al tempo di accesso ai dati, sarebbe opportuno mantenere tali dati vicini agli utenti che li utilizzano, soprattutto se si presentano molto statici e cambiano raramente.

1.5 Cloud OS

La complessa infrastruttura virtualizzata che sta alla base del Cloud, di cui abbiamo appena esposto i concetti basilari, deve essere gestita in modo da offrire servizi sicuri, efficienti e scalabili. Tale compito può essere molto complesso. É quindi necessaria la presenza di un componente che si prenda carico della gestione dell'infrastruttura, governandone la virtualizzazione,

l'allocazione delle risorse agli utenti ed applicando le corrette politiche che gli permettano di raggiungere i requisiti fondamentali di un sistema Cloud, di cui abbiamo già precedentemente parlato.

1.5.1 Scopo del Cloud OS

Tale componente è il Cloud Operating System, o Cloud OS, ed ha come obbiettivo principale quello di fornire l'infrastruttura come un servizio (*IaaS*), sia per i tenant del Cloud che per i livelli superiori, come *PaaS* e *SaaS*. Per far ciò esso deve saper gestire sia l'architettura fisica sottostante che quella virtualizzata e deve poter fornire ai suoi client le risorse virtuali richieste. Più nello specifico, deve fornire un'interfaccia pubblica che permetta di accedere ai servizi dell'infrastruttura dall'esterno nascondendone la virtualizzazione ed i dettagli legati alla sua gestione. Tale interfaccia deve essere indipendente dall'hardware fisico sottostante, come per i normali sistemi operativi, e dalla tecnologia di virtualizzazione adottata. É inoltre necessaria un'interfaccia più ricca, disponibile solo per gli amministratori del sistema, tramite cui essi possano intervenire in maniera più profonda nella gestione del sistema, ad esempio, controllando e monitorando i componenti virtuali e fisici del sistema.

Categorie principali di Cloud OS

Si possono distinguere principalmente due tipi di Cloud OS, poiché mentre alcuni sono più orientati alla gestione di Public Cloud, come OpenStack[11], altri si focalizzano maggiormente sulla virtualizzazione dei centri di dati verso un'architettura Cloud, come ad esempio Eucalyptus[12] o OpenNebula[13]. Entrambi forniscono funzionalità simili per quanto riguarda gli aspetti centrali di un Cloud OS, ma quelli orientati ai centri di dati virtualizzati tipicamente forniscono anche dei moduli per l'accesso ed integrazione di infrastrutture Cloud remote.

[11]http://www.openstack.org
[12]http://www.eucalyptus.com
[13]http://opennebula.org

1.5.2 Componenti

Per fornire un'astrazione dell'infrastruttura sottostante, il Cloud OS fa uso di adattatori e driver che gli permettono di interagire con le varie tecnologie di virtualizzazione. I componenti che costituiscono il nucleo di un Cloud OS si appoggiano a questi driver per gestire, monitorare e distribuire l'infrastruttura virtualizzata. I componenti principali sono:

- *Virtual Machine Manager (VMM)*, che ha un ruolo centrale, poiché i Cloud OS tipicamente definiscono le macchine virtuali come l'unità base di esecuzione, analogamente ai thread nel caso di sistemi operativi desktop multi-threaded;

- *Network Manager*, che gestisce la comunicazione tra i componenti, la creazione di reti private e l'assegnazione di indirizzi pubblici;

- *Storage Manager*, che fornisce l'accesso a funzionalità di memorizzazione persistente delle informazioni;

- *Information Manager*, che monitora e raccoglie informazioni riguardanti lo stato delle macchine virtuali, i server fisici ed altri componenti.

Un Cloud OS si può inoltre dotare di driver e adattatori che gli permettano di accedere a provider remoti. Il componente che utilizza questi driver è detto *Federation Manager*, e si occupa di gestire, monitorare e distribuire risorse virtuali remote, provenienti dai provider di cui sopra.

1.6 Proprietà architetturali del Cloud

Riassumendo quanto finora detto riguardo al Cloud Computing, in questa sezione enunceremo tutte le caratteristiche importanti delle piattaforme Cloud, come descritte anche in [9], cominciando ad ottenere utili indicazioni per lo sviluppo di applicazioni orientate al Cloud.

Definizione di piattaforma

Una piattaforma è l'insieme di concetti, componenti e paradigmi architetturali che formano il nucleo di un ambiente di sviluppo e distribuzione di applicazioni. Ogni piattaforma ha le sue proprietà architetturali, le quali influenzano le applicazioni che tramite di essa vengono sviluppate.

Piattaforme e sviluppo software

Un progettista dovrebbe sempre tenere in considerazione tali proprietà durante il processo di sviluppo del software, poiché esse sono in grado di arricchire il suo prodotto di qualità e caratteristiche che lo aiutano a soddisfare requisiti sia funzionali che non.

Analizzeremo ora più nel dettaglio le caratteristiche dell'architettura Cloud, affrontando prima quelle delle piattaforme virtualizzate più in generale, poi quelle specifiche delle piattaforme Cloud. Da esse dedurremo alcuni principi generali che permetteranno di realizzare applicazioni in grado di sfruttare le potenzialità delle piattaforme Cloud.

1.6.1 Proprietà delle piattaforme virtualizzate

Le principali proprietà delle piattaforme virtualizzate possono essere così sintetizzate:

- Astrazione delle risorse hardware fisiche e delle entità software
- Astrazione dell'interfaccia hardware
- Isolamento degli ambienti di calcolo e multi-tenancy a livello di macchina virtuale
- Scarso isolamento delle performance
- Incapsulamento di tutto il software eseguito su un hardware fisico
- Capacità di controllo dell'hardware limitate da parte delle macchine virtuali
- Possibilità di salvare lo stato di una macchina in un qualsiasi momento per poterlo successivamente ripristinare
- Possibilità di migrare una macchina virtuale

Molte di queste proprietà hanno recentemente trovato applicazione in maniera innovativa proprio nello sviluppo di piattaforme orientate al Cloud.

1.6.2 Proprietà delle piattaforme Cloud

La ragione che ci ha indotto ad analizzare prima delle piattaforme orientate al Cloud quelle virtualizzate, è che queste ultime formano il nucleo di tutto il sistema del Cloud computing. Di conseguenza le proprietà elencate nella sezione precedente sono disponibili anche nelle piattaforme orientate al Cloud.

Oltre ad esse possiamo distinguere ulteriori proprietà, alcune condivise da tutte le piattaforme Cloud, altre che si specializzano sia in base al servizio offerto (*Iaas*, *Paas* o *Saas*) oppure al tipo di Cloud adottato (*Public*, *Private*, *Community* o *Hybrid*).

Proprietà generiche

Le proprietà condivise da tutte le piattaforme Cloud sono:

- Natura elastica delle risorse computazionali
- Capacità di automatizzare l'assegnazione delle risorse
- Assegnazione delle risorse sulla base della richiesta
- Misurazione dei servizi erogati
- Computazione intesa come un servizio accessibile attraverso la rete
- Mancanza di standard per servizi importanti come la sicurezza o il controllo e la gestione delle macchine virtuali
- Mancanza di controllo assoluto sulla custodia dei dati e della computazione
- Tecniche di misurazione non ottimali, derivanti dallo scarso isolamento delle performance delle macchine virtuali
- Mancanza di meccanismi per il monitoraggio dettagliato delle risorse
- Difficoltà nel comprendere la struttura delle licenze software, soprattutto negli scenari più complessi, in cui più licenze di software diversi sono applicate in uno stesso ambiente

Proprietà specifiche dell'Iaas

Di seguito sono elencate le proprietà caratteristiche di una piattaforma Cloud che fornisce servizi secondo il modello *Iaas*:

- L'utente del Cloud è responsabile dell'installazione e della gestione di tutto il software delle proprie macchine virtuali
- Possibilità di monitorare l'uso delle risorse e di reagire a differenti tipi di eventi a livello di piattaforma
- Le applicazioni in esecuzione nella macchina virtuale sono responsabili di gestire le reazioni a tali eventi: ad esempio, a fronte dell'allocazione di una nuova istanza della macchina sono le applicazioni che devono farsi carico di sfruttare al massimo le nuove capacità introdotte da tale nuova istanza, suddividendo il carico di lavoro tra le due macchine
- Controllo limitato sui componenti di rete come, ad esempio, i firewall delle macchine ospiti

Proprietà specifiche del Paas

Per quanto riguarda le piattaforme che forniscono servizi di tipo *Paas*, le principali proprietà in esse individuate sono:

- Nessun controllo sull'infrastruttura sottostante, ovvero sui server, sulla rete, sui sistemi operativi o sul sistema di memorizzazione di massa
- Possibilità di controllare le applicazioni distribuite ed eventualmente la configurazione del loro ambiente di esecuzione
- Possibilità di utilizzare solo i linguaggi di programmazione, i tool, le API ed i componenti supportati dal provider: non è possibile, ad esempio, installare nuovo software, aggiornarlo o introdurre un ambiente che non sia tra quelli forniti dal provider

Proprietà specifiche del Saas

Le piattaforme che forniscono servizi di tipo *Saas*, sono caratterizzate dalle seguenti proprietà:

- Nessun controllo sull'infrastruttura sottostante
- Controllo solo su un insieme limitato di impostazioni di configurazione, specifiche per ogni utente

Proprietà specifiche delle Public Cloud

Anche in base a quanto esposto precedentemente possiamo in questo modo riassumere le caratteristiche delle *Public Cloud*:

- L'infrastruttura Cloud è resa disponibile, a pagamento, a qualunque utente
- La proprietà dell'infrastruttura Cloud è detenuta dall'organizzazione che ne vende i servizi
- Il Cloud provider detiene la custodia ed il controllo di tutti i dati, delle applicazioni e della computazione ospitati presso la sua infrastruttura
- L'ambiente di virtualizzazione spesso è omogeneo

Proprietà specifiche delle Private Cloud

Le *Private Cloud* invece presentano tipicamente le seguenti proprietà:

- Viene utilizzata solamente da un'organizzazione
- Controllo, custodia e proprietà totali delle applicazioni, dei dati e della computazione
- Configurazione personalizzata dell'infrastruttura Cloud
- L'ambiente di virtualizzazione spesso è omogeneo

Proprietà specifiche delle Community Cloud

Le proprietà più rilevanti delle *Community Cloud* si possono così riassumere:

- Infrastruttura Cloud condivisa tra più organizzazioni
- Supporta una determinata comunità di organizzazioni che hanno precedentemente condiviso obbiettivi ed ambiti lavorativi

- Le organizzazioni appartenenti alla comunità hanno il controllo, la custodia e la proprietà totali delle applicazioni, dei dati e della computazione

Proprietà specifiche delle Hybrid Cloud

Per quanto riguarda le *Hybrid Cloud*, le caratteristiche principali di tali piattaforme sono:

- Struttura logica che unisce due o più Cloud, eventualmente anche di tipo diverso
- Ogni Cloud che la costituisce rimane un entità separata, e mantiene le sue proprietà architetturali

Capitolo 2

Cloud Operating System

Oggi i Cloud Operating System stanno rapidamente acquistando una grande importanza, non solo nel mondo deI Cloud provider, ma anche nei contesti aziendali di molte grandi e medie imprese, che fino a poco tempo fa non avevano alcun interesse o bisogno di rivolgersi al Cloud Computing. Ciò è dovuto al fatto che molte organizzazioni si stanno rendendo conto che centralizzare le proprie risorse informatiche, garantendone una visione omogenea, una gestione di più alto livello ed un modello di utilizzo elastico, che sono caratteristiche tipiche degli ambienti Cloud, può portare ad una notevole riduzione del lavoro necessario al mantenimento e all'uso di tali risorse, quindi ad una riduzione dei costi, legata ad un miglioramento nelle prestazioni. Di conseguenza, molte aziende in tutto il mondo attualmente stanno seguendo il processo evolutivo che le porta a trasformare i loro centri di dati, eventualmente già virtualizzati, in infrastrutture di tipo Private Cloud.

Questa trasformazione porta con sé notevoli vantaggi:

- disaccoppiamento tra applicazioni e servizi ed i relativi server fisici su cui sono in esecuzione;
- disaccoppiamento tra i dati ed i dispositivi di memorizzazione di massa in cui sono contenuti;
- possibilità di estendere l'infrastruttura locale privata con risorse remote provenienti da altre infrastrutture che hanno subito lo stesso cambiamento o da Public Cloud.

Dal punto di vista dell'amministrazione dell'infrastruttura ciò si traduce in una semplificazione dei processi di gestione dei server per ottenere la potenza desiderata, in una grande facilità nel ridimensionamento dell'infrastruttura fisica, in un maggiore isolamento dei servizi e degli utenti, nella possibilità di bilanciare il carico di lavoro automaticamente tra le macchine per migliorarne l'efficienza e l'utilizzo, e in una semplificata gestione della replicazione dei server per aumentare la tolleranza ai guasti. Per operare questa trasformazione, organizzando quindi la propria infrastruttura secondo un'architettura di tipo Cloud, è indispensabile la presenza di un Cloud Operating System.

Nel capitolo introduttivo si è già accennato ad essi, collocandoli all'interno dell'architettura Cloud, spiegando i loro obbiettivi ed elencandone i principali componenti. In questo capitolo i Cloud OS verranno affrontati in maniera più approfondita; in particolare ne studieremo più nel dettaglio i componenti, come descritto in [8], e daremo una breve panoramica dei Cloud OS al momento più diffusi.

2.1 Analogie con Desktop OS e modularizzazione

Ci sono molte similitudini tra i classici sistemi operativi e quelli Cloud. I primi hanno come scopo principale la gestione delle risorse di un computer; allo stesso modo i Cloud OS devono saper gestire le risorse dell'infrastruttura virtualizzata sottostante. I sistemi operativi desktop devono fornire un ambiente di esecuzione sicuro e multi-threaded per le applicazioni utente che garantisca il loro isolamento; allo stesso modo quelli Cloud devono fornire un ambiente di esecuzione sicuro, multi-tenant e che garantisca ai servizi utente l'isolamento reciproco. Infine i sistemi operativi desktop devono permettere la condivisione di risorse tra utenti diversi, astraendo dai dettagli dell'hardware sottostante ed offrendo interfacce, sia agli utenti comuni che agli amministratori, per poter interagire con il calcolatore, ciascuno secondo il proprio ruolo; analogamente i Cloud OS devono permettere la condivisione di risorse tra i vari tenant, astraendo dalle infrastrutture sottostanti, sia quella a livello fisico che quella virtualizzata, e fornendo un insieme di API per permettere l'interazione con il Cloud secondo ruoli diversi.

Non deve sorprendere quindi che, come i sistemi operativi desktop sono organizzati in moduli, per poter dominare la complessità del problema e per offrire una maggiore flessibilità e configurabilità, similmente i Cloud OS sono organizzati in componenti, ciascuno dei quali si occupa di risolvere un sotto-problema e si colloca in un determinato ambito di lavoro.

2.2 Virtual Machine Manager

I sistemi operativi desktop definiscono i thread come unità d'esecuzione, fornendone molti meccanismi di comunicazione e sincronizzazione, e le applicazioni multi-threaded come l'entità di base eseguibile dall'utente. I sistemi operativi Cloud invece definiscono come entità di base i servizi virtualizzati che gli utenti o i tenant possono invocare e le macchine virtuali, organizzate solitamente in architetture a più livelli, come unità d'esecuzione, garantendone strumenti di comunicazione e configurazione. Questa architettura concorre a migliorare la scalabilità poiché è sufficiente aggiungere macchine virtuali al servizio corrente (scalabilità orizzontale) o ridimensionare una macchina virtuale, se supportato (scalabilità verticale).

Le applicazioni e i servizi sono tra loro isolati, mentre le macchine virtuali appartenenti ad un'applicazione o ad un servizio non lo sono: ad esse infatti vengono fornite reti virtuali ed altri servizi per permettere e governare la comunicazione.

Il Virtual Machine Manager è responsabile dell'intero ciclo di vita delle macchine virtuali e deve essere in grado di eseguire varie operazioni in base alle azioni degli utenti o alle loro politiche di scheduling. Tali operazioni comprendono l'istanziazione, la migrazione, il sospendimento e la ripresa o lo spegnimento delle macchine virtuali. Inoltre il Virtual Machine Manager deve essere in grado di garantire il rispetto dei *Service-Level Agreements* (SLA) contratti con gli utenti, che solitamente si esprimono in termini di disponibilità delle macchine virtuali. Esso viene quindi arricchito con dei meccanismi che lo rendono capace di individuare i crash delle macchine virtuali e di riavviarle in caso di errore.

Le macchine virtuali vengono solitamente definite tramite un insieme di parametri ed attributi, tra i quali, ad esempio, il kernel del sistema operativo da esse utilizzato, l'immagine del loro disco rigido, la quantità di memoria volatile e la potenza di CPU assegnate.

2.3 Network Manager

Per poter sviluppare servizi ed applicazioni nel Cloud non c'è bisogno solamente di macchine virtuali, ma occorre anche istanziare reti per interconnettere i vari componenti di un servizio, permettendo loro di comunicare, e per rendendere il servizio accessibile anche da utenti esterni, se necessario. Il Network Manager deve quindi saper gestire reti private per le connessioni interne tra i componenti dei servizi ed indirizzi IP pubblici per i componenti di front-end accessibili tramite Internet.

Poiché anche le reti fornite dal Network Manager sono virtuali, esso deve disporre di meccanismi e politiche che gli permettano di assegnare correttamente sia gli indirizzi MAC che quelli IP e deve anche riuscire a garantire l'isolamento del traffico tra varie reti virtuali, anche se queste condividono la stessa infrastruttura di rete fisica.

2.4 Storage Manager

La funzione principale dello Storage Manager è quella di fornire servizi e sistemi di memorizzazione persistente virtuali, secondo le specifiche tipiche del Cloud. Ciò significa che il sistema di storage deve essere rapidamente scalabile e deve saper crescere automaticamente in base alla necessità degli utilizzatori, deve avere grande disponibilità ed affidabilità per evitare perdite di dati in caso di errori o crash, deve garantire ottime performance per supportare grossi carichi di lavoro, ad esempio nel caso di applicazioni orientate ai dati o che ne fanno un uso intensivo, e deve essere semplice da gestire, permettendo agli utenti di astrarre dalla complessità dell'infrastruttura di storage sottostante, che si può comporre anche di dispositivi diversi e tra loro eterogenei.

Come già accennato nel capitolo introduttivo, una questione molto importante che riguarda in generale la memorizzazione persistente di dati in un sistema altamente scalabile e dinamico è il modello dei dati. Finora infatti, dopo che nella seconda metà del '900 in ambito di ricerca erano stati proposti vari modelli, come quello gerarchico o quello reticolare, non aveva tardato ad imporsi su tutti gli altri il modello relazionale, dominando per anni la scena in ogni ambito, da quelli didattici e di ricerca a quelli aziendali. Esso deve la sua fortuna alla sua espressività ed alla sua intuitività, che gli hanno permesso di diffondersi ovunque, ed alle alte prestazioni offerte

dai sistemi che lo adottano, come i Database relazionali interrogati da linguaggi appartenenti alla famiglia dell'SQL. Tali sistemi infatti hanno subito nel corso del tempo una grande attività di studio volta ad ottimizzarne le prestazioni. I linguaggi derivanti dall'algebra relazionale inoltre sono dotati di una grande espressività e permettono di realizzare ed eseguire operazioni anche molto complesse. Tuttavia tali sistemi hanno delle grosse difficoltà ad implementare la scalabilità ed è per questo motivo che negli ultimi anni sono stati rivalutati tutti gli altri modelli che ormai da decenni non venivano quasi più presi in considerazione. Ciò ha chiaramente delle ripercussioni anche nello sviluppo di applicazioni che devono essere eseguite nel Cloud, poiché occorre imparare a modellare e gestire i dati secondo i nuovi modelli adottati. Nei capitoli successivi ci dedicheremo più approfonditamente a questo tema, prendendo in analisi un caso particolare: *Bigtable*, il sistema di storage distribuito di Google.

2.5 Image Manager

La gestione delle immagini delle macchine virtuali è molto importante, sia nei centri di dati virtualizzati, sia nei sistemi più propriamente Cloud. Tali sistemi infatti devono essere in grado di gestire una grande quantità di immagini, appartenenti a vari utenti, con sistemi operativi differenti e diverse configurazioni. Tale gestione deve essere sicura ed efficiente, inoltre i Cloud OS devono anche possedere degli strumenti per gestire i *repository* di immagini.

Le immagini di macchine virtuali tipicamente vengono determinate da un insieme di attributi, come il nome, la descrizione del contenuto, il livello di condivisione (privata, pubblica o condivisa a pochi), il proprietario e la sua locazione all'interno del *repository*. Le funzionalità di base per gestire le immagini includono la creazione di una nuova immagine in un certo *repository*, la sua cancellazione, la clonazione di un'immagine esistente, l'aggiunta o il cambiamento di determinati attributi, la condivisione di un'immagine da parte di un utente con altri utenti o la sua pubblicazione per permetterne l'utilizzo da parte di tutti gli utenti.

2.6 Information Manager

Questo modulo è il responsabile del monitoraggio e della raccolta di informazioni riguardanti le macchine virtuali, i server fisici, ed altri componenti appartenenti sia alla struttura fisica che a quella virtuale. Il suo lavoro è essenziale per assicurarsi che tutti i componenti stiano funzionando correttamente e stiano fornendo buone prestazioni. Il controllo dei server fisici può essere effettuato installando determinati strumenti sui server stessi, che interagiscono con l'Information Manager. Il controllo delle macchine virtuali invece può essere fatto sulla base delle informazioni fornite dall'Hypervisor, che tuttavia risultano spesso abbastanza scarse e soprattutto possono variare da un Hypervisor ad un altro, oppure installando anche in questo caso determinati strumenti sulle macchine virtuali. In questo caso però c'è bisogno del consenso dell'utente proprietario della macchina in questione, e si tratta comunque di una soluzione abbastanza invasiva.

L'Information Manager può fornire vari sensori, ognuno dei quali è responsabile di fornire informazioni riguardo aspetti diversi del sistema, come ad esempio il carico di CPU, l'uso della memoria volatile, il numero di processi in esecuzione, l'uso della memoria persistente, il consumo di energia o di banda. Inoltre può permettere di creare sensori personalizzati in grado di monitorare metriche diverse, più specifiche ed adatte al tipo di servizio o applicazione sviluppata.

2.7 Federation Manager

Il Federation Manager è il componente che abilita l'accesso alle infrastrutture Cloud remote, sia che si tratti di infrastrutture private governate da un simile Cloud Operating System, sia che si tratti di Public Cloud accessibili tramite le loro interfacce, di cui parleremo nelle prossime sezioni. Tale componente deve fornire i meccanismi che permettano di distribuire le risorse presso Cloud remote, gestirle durante l'esecuzione, ritirarle o terminarle quando necessario, monitorarle secondo i parametri desiderati, autenticare gli utenti che vi accedono e creare immagini di macchine virtuali, le quali probabilmente avranno formati diversi. In base alle capacità offerte dal Cloud remoto, che determinano il livello di interoperabilità raggiungibile nell'interazione tra i due Cloud, può essere possibile fornire meccanismi anche più avanzati, come quelli per la migrazione di macchine virtuali at-

traverso le diverse infrastrutture, o quelli per la creazione di reti e sistemi di storage condivisi.

Il Federation Manager deve essere implementato come un componente interno del Cloud OS per poter supportare la *federation* a livello di infrastruttura. É tuttavia possibile appoggiarsi a dei servizi esterni, come il progetto Aeolus[1], per raggiungere la *federation* a livello di utente. Nelle sezioni successive verrà approfondito il concetto di *federation*, spiegando quali architetture esistono e quali sono i livelli di interoperabilità raggiunti da esse.

2.8 Scheduler

In un'infrastruttura Cloud esistono due livelli di scheduling: il primo, atto a stabilire, a livello fisico, quali macchine virtuali mandare in esecuzione e quali processori assegnare ad ogni macchina virtuale, viene gestito dallo scheduler dell'Hypervisor; il secondo è di competenza dello scheduler del Cloud OS ed è responsabile di decidere su quali server fisici distribuire le macchine virtuali. Ciò viene fatto in base a vari criteri, ciascuno dei quali induce le proprie politiche di allocazione e migrazione delle macchine virtuali. I criteri sono i seguenti:

- Riduzione del numero di server in uso, per minimizzare il dispendio energetico. Le nuove macchine virtuali dovrebbero essere allocate, quando possibile, presso server già attivi.
- Bilanciamento del carico di lavoro, per evitare la saturazione di alcuni server con conseguente crollo delle performance; in tal caso le macchine virtuali dovrebbero essere equamente distribuite in numero tra tutti i server disponibili.
- Bilanciamento dell'uso della CPU, che ha lo stesso obbiettivo del punto precedente, ma applica la politica secondo la quale le macchine virtuali devono essere allocate presso i server con la più alta percentuale di CPU libera.
- Bilanciamento termico, per evitare il surriscaldamento dei server e ridurre la necessità di meccanismi di raffreddamento. In questo caso

[1] http://www.aeolusproject.org/index.html

le macchine virtuali dovrebbero essere allocate presso i server con la temperatura più bassa.

Come si può facilmente notare spesso i criteri sono in contrasto tra loro, ed un buon scheduler deve riuscire a tenerli tutti in considerazione trovando il giusto equilibrio. Ovviamente lo scheduler non è incaricato solamente di allocare i server fisici alle macchine virtuali nel momento della creazione o della loro accensione, ma deve costantemente monitorare la situazione per saper decidere quando è necessario migrare le macchine virtuali già in esecuzione.

In presenza di contesti in cui è disponibile la *federation* lo scheduler può anche decidere di allocare o migrare le macchine virtuali presso server remoti.

Infine lo scheduler deve anche tener conto di ulteriori vincoli imponibili dagli utenti, come i livelli minimi di CPU e memoria volatile allocati alle proprie macchine, la necessità di tenere due o più macchine virtuali in esecuzione sullo stesso server, ad esempio per minimizzare il traffico di rete se tali macchine interagiscono molto frequentemente, vincoli legati alla distribuzione geografica, che sorgono spesso a causa delle differenti leggi vigenti nei vari Paesi riguardanti il trattamento di dati personali, e vincoli legati al rispetto dei *Service-level Agreements*, esprimibili come capacità di CPU garantita ed affidabilità dei servizi.

Come per i sistemi operativi desktop, lo scheduler viene invocato ogni volta che una nuova macchina virtuale è in attesa che le venga allocato un server fisico e periodicamente per eseguire le ottimizzazioni necessarie sull'intera infrastruttura virtuale, riallocando, se necessario, alcune macchine virtuali. Per far ciò, questo modulo deve poter interagire con il Virtual Machine Manager ed, eventualmente, con il Federation Manager

2.9 Service Manager

Un Cloud OS deve essere in grado di gestire servizi virtualizzati, spesso costruiti su un'architettura a più livelli e costituiti da più componenti legati tra loro da relazioni di dipendenza. I servizi vengono resi disponibili distribuendoli tipicamente su più macchine virtuali, le quali a loro volta ereditano o generano nuove relazioni di dipendenza, in base ai componenti

o livelli che ospitano. Inoltre i servizi solitamente necessitano di appoggiarsi ad elementi di memorizzazione persistente e di comunicazione.

Una delle funzionalità del Service Manager, la prima ad entrare in gioco a fronte della realizzazione di un nuovo servizio, è il controllo che viene effettuato sul servizio stesso per determinare se esso può essere accettato nel Cloud o se deve essere rifiutato, in base alle risorse che richiede, alle autorizzazioni di cui necessita ed alle disponibilità del Cloud.

Una volta che un servizio è stato accettato, il Service Manager diventa responsabile del suo intero ciclo di vita, che include operazioni come il deployment, la sua sospensione o riattivazione e la sua cancellazione. Per poter effettuare il deployment del servizio il Service Manager deve interagire con lo scheduler in modo da poter determinare correttamente dove distribuire le macchine virtuali di cui tale servizio necessita..

Un'altra importante funzionalità del Service Manager è la gestione dell'elasticità dei servizi. Il Service Manager per poter adempiere a questo compito incorpora solitamente vari meccanismi per innescare la scalabilità automatica dei servizi, i quali si basano su regole precise che determinano come scalare il servizio (verticalmente o orizzontalmente) e quando (in base al superamento di alcune soglie su determinate metriche imposte dall'utente, o in maniera automatica).

2.10 Interfacce

Le funzionalità del Cloud OS possono essere rese pubbliche ed esposte agli utenti tramite determinate interfacce. Queste interfacce costituiscono ciò che l'utente del Cloud percepisce effettivamente come *Infrastructure as a Service*, poiché determinano l'insieme dei servizi di cui egli può disporre per interagire con l'infrastruttura.

Solitamente ogni Cloud provider fornisce le proprie API con strumenti diversi. La modalità più diretta, ma anche più complessa, per richiedere i servizi del Cloud è tramite le richieste del protocollo HTTP, il quale è diventato ormai lo standard per l'interazione secondo l'architettura SOA. Tuttavia, per semplificare lo sviluppo di applicazioni e per fare in modo che le richieste vengano rappresentate in una modalità più ricca di significato per i linguaggi di programmazione, spesso vengono anche fornite della librerie specifiche per ogni linguaggio supportato. Per permettere l'invocazione dei

servizi direttamente dagli utenti, piuttosto che dalle applicazioni, vengono anche forniti dei tool, sia grafici sia da linea di comando.

Il problema tuttavia è che, nonostante alcune di queste interfacce siano talmente diffuse e ben progettate che stanno ormai diventando quasi degli standard *de facto*, come le API di Amazon Web Services, che vengono infatti supportate nativamente dal Cloud OS Eucalyptus, il livello di eterogeneità è ancora molto alto e ciò rende molto complessa l'interoperabilità e la portabilità attraverso Cloud diverse.

Esistono oggi due approcci diversi che cercano di risolvere il problema dell'eterogeneità: i cosiddetti *Cloud adapters*, come Apache Deltacloud o Libcloud, che prendono in considerazione un vasto numero di Cloud tra le più diffuse e cercano di realizzare uno strato intermedio che ne permetta la comunicazione, o gli organismi di standardizzazione che cercando invece di stabilire delle interfacce comuni in grado però di permettere ai Cloud provider di offrire i loro servizi eventualmente anche differenziandosi gli uni dagli altri. Tra gli standard che attualmente si stanno diffondendo e stanno acquistando sempre più visibilità e importanza ci sono *OGF OCCI*[2], descritto anche in [3], e *DMTF CIMI* e *OVF*[3].

2.11 Autenticazione e autorizzazione

Come per ogni ambiente condiviso, anche per il Cloud è importante possedere dei meccanismi in grado di riconoscere ed autenticare gli utenti e gli amministratori, per poter anche fornire loro accesso alle risorse per le quali dispongono delle autorizzazioni necessarie.

I componenti che si occupano dell'autenticazione in particolare hanno come obbiettivo proprio quello di verificare e confermare l'identità dell'utente che cerca di accedere alle risorse del Cloud. I metodi che possono essere utilizzati per implementare queste funzionalità sono svariati e vanno dalla semplice verifica di una password, a protocolli molto più complessi, che oggi si stanno largamente diffondendo a causa dell'importanza sempre maggiore che sta acquisendo la sicurezza nell'ambito del Cloud ed, in generale, nei sistemi altamente distribuiti ed aperti.

[2]http://occi-wg.org
[3]http://dmtf.org/standards/cloud

Le politiche di autorizzazione invece sono responsabili del controllo e della gestione dei privilegi e dei permessi dei vari utenti riguardanti l'accesso di ogni tipo di risorsa, dalle macchine virtuali, alle reti ed ai sistemi di storage. Il controllo degli accessi può essere implementato tramite dei meccanismi basati sul concetto di ruolo. Il ruolo viene definito come un insieme di permessi di svolgere determinate operazioni su alcune specifiche risorse. Ad ogni utente possono quindi essere assegnati più ruoli. In aggiunta si possono usare dei meccanismi per limitare l'ammontare di risorse che un utente può utilizzare, ad esempio in termini di larghezza di banda, quantità di memoria, e potenza di calcolo disponibile.

2.12 Accounting ed auditing

Lo scopo dei componenti che si occupano dell'accounting è di ottenere informazioni riguardanti l'uso di determinate risorse da parte dei servizi. Esso fa quindi affidamento sul Federation Manager per accedere alle informazioni riguardanti le metriche da quest'ultimo monitorate. L'accounting è essenziale per poter implementare i meccanismi di allocazione dei costi in maniera tale che ad ogni utente sia imposto il pagamento delle sole risorse utilizzate dalle sue applicazioni.

Un altro aspetto fondamentale degli ambienti Cloud è l'auditing: esso consiste nella raccolta delle informazioni riguardanti l'attività delle varie risorse, in particolare chi ne ha ottenuto l'accesso, quando l'ha ottenuto e che tipo di utilizzo ne ha fatto. Queste informazioni risultano utili per migliorare la sicurezza del Cloud e per proteggerlo da varie minacce, come accessi non autorizzati, utilizzo abusivo delle risorse ed intrusioni illecite.

2.13 Livello di accoppiamento di Cloud

Come già accennato nella sezione relativa al Federation Manager, approfondiamo ora il tema della federazione, analizzando in particolare il grado di interoperabilità che è raggiungibile da due Cloud diversi. Esso viene determinato in funzione delle possibilità di interagire che i Cloud si offrono reciprocamente e comprende aspetti come il controllo ed il monitoraggio di risorse remote, l'istanziazione di reti che attraversino i confini dei Cloud e la

migrazione di macchine virtuali. In base agli scenari più diffusi è possibile classificare i livelli di accoppiamento in tre principali categorie:

Interoperabilità debole. Lo scenario tipico in questo caso è costituito da istanze di Cloud indipendenti, come per Cloud privati che si espandono appoggiandosi ad un Cloud pubblico, caratterizzati da una scarsa interazione. Solitamente non c'è, o è molto debole, il controllo sulle risorse remote, il monitoraggio delle risorse è assai limitato e le funzionalità più avanzate, come la migrazione di macchine virtuali, non sono supportate. In questo caso, ad esempio, un Cloud può non avere alcun controllo sulla localizzazione delle macchine virtuali dell'altro e può ricevere informazioni solo riguardanti le metriche più importanti e basilari, come l'uso della CPU e della memoria da parte di una macchina virtuale.

Interoperabilità media. In questo scenario solitamente sono presenti Cloud le cui organizzazioni hanno stipulato qualche forma di contratto stabilendo a priori sotto quali termini e condizioni un Cloud può utilizzare le risorse dell'altro. Tale contratto può abilitare un controllo più sofisticato sulle risorse remote ed un loro più fine monitoraggio; ad esempio, possono essere fornite informazioni sulla localizzazione ed il consumo energetico delle risorse e può esserci la possibilità di intervenire in qualche modo su questi aspetti. Inoltre tipicamente è possibile creare delle reti virtuali che attraversano i confini dei Cloud.

Interoperabilità forte. In questo caso solitamente i Cloud appartengono tutti alla stessa organizzazione e vengono gestiti dallo stesso Cloud OS. Possono quindi essere garantiti un controllo molto avanzato sulle risorse remote ed un monitoraggio molto dettagliato su tutte le metriche disponibili. Sono talvolta supportate anche le funzionalità più avanzate, come la migrazione di macchine virtuali o la creazione di sistemi di storage virtuale che attraversino i confini dei Cloud. Il livello di accoppiamento può essere elevato al punto da poter definire tale federazione, almeno da un punto di vista esterno, come un unico Cloud, non potendo percepire le differenze ed i confini tra i Cloud che la compongono.

2.14 Architetture di federazione

I livelli di accoppiamento analizzati nella sezione precedente trovano riscontro nelle architetture tipicamente istanziate con lo scopo di ottenere la federazione tra Cloud. Tali architetture non sono ancora degli standard uniformemente riconosciuti, ma sono state dedotte studiando gli scenari di federazione più diffusi oggi nel mondo del Cloud. Esse possono essere sintetizzate in quattro categorie principali.

Cloud bursting. Quest'architettura è in realtà già stata analizzata quando abbiamo affrontato i principali modelli di Cloud: si tratta infatti di un Cloud ibrido. Essa viene istanziata quando c'è la necessità di integrare ed espandere un'infrastruttura esistente, tipicamente ma non necessariamente organizzata a sua volta come un Cloud, appoggiandosi ad un Cloud pubblico. Il livello di accoppiamento raggiunto in questo caso è debole, poiché il Cloud OS remoto interagisce con quello locale come se fosse un qualunque client.

Cloud broker. In questo caso l'architettura fa uso di un componente particolare che funge da mediatore, o *broker* appunto, il quale ha accesso a diversi Cloud pubblici. Nella sua implementazione più semplice esso deve essere in grado di gestire le risorse degli utenti nei Cloud che essi desiderano. Broker più complessi sono anche in grado di applicare determinate politiche di scheduling, basate su criteri di ottimizzazione, ad esempio dei costi o delle prestazioni, decidendo su quali Cloud distribuire i servizi o eventualmente anche i singoli componenti. Poiché tipicamente i Cloud provider non consentono la gestione avanzata delle proprie risorse da parte di utenti esterni, anche in questo caso il livello di interoperabilità raggiungibile dai vari Cloud coinvolti è piuttosto basso. Esempi di Cloud broker attualmente disponibili sono BonFIRE[4], Open Cirrus[5] o FutureGrid[6].

Aggregazione di Cloud. L'aggregazione di Cloud viene realizzata a partire da due o più Cloud che interagiscono e uniscono in qualche modo

[4]http://www.bonfire-project.eu/
[5]https://opencirrus.org/
[6]https://portal.futuregrid.org/

alcune delle proprie risorse in modo da fornire agli utenti un'infrastruttura virtuale più grande. L'interoperabilità in questo caso è medio livello, poiché le organizzazioni possono fornirsi vicendevolmente qualche forma di controllo avanzato, essendo in grado di distinguere le interazioni provenienti dal Cloud aggregato da quelle degli utenti comuni. Solitamente per poter realizzare tale architettura è necessario stipulare un contratto e stabilire i termini secondo i quali è possibile interagire, come già visto nella sezione precedente, analizzando il grado di accoppiamento che corrisponde a questa architettura. É tuttavia possibile riscontrare delle aggregazioni di Cloud in grado di cooperare più profondamente: ciò avviene tipicamente quando i Cloud coinvolti appartengono tutti alla stessa organizzazione. Un esempio di architettura di aggregazione di Cloud è RESERVOIR[7].

Architettura multi-livello. Questo tipo di architettura consiste di due o più Cloud, ciascuno dei quali possiede il proprio Cloud OS, che vengono a loro volta gestiti da un altro Cloud OS, posto gerarchicamente ad un livello superiore. Poiché solitamente questa architettura viene istanziata da Cloud provider che possiedono più infrastrutture separate, disposte geograficamente in regioni spesso molto distanti per garantire tolleranza ai guasti e riduzione delle latenze, il livello di interoperabilità raggiungibile è il massimo. Il Cloud OS a livello superiore ha un controllo completo delle risorse dei vari Cloud e ne fornisce un'interfaccia agli utenti tale da far loro percepire la presenza di un unico Cloud sottostante.

Concludiamo questa sezione osservando che, mentre per le architetture che forniscono un livello di accoppiamento debole è stato finora fatto molto lavoro e sono ora disponibili anche sul mercato molte utili soluzioni, per quanto riguarda le altre architetture è forse necessario progredire ulteriormente con lo sviluppo, soprattutto in merito a quelle multi-livello, che sono finora utilizzate solamente secondo la modalità sopra esposta.

[7]http://reservoir-fp7.eu/

2.15 Breve analisi di Cloud OS esistenti

In questa sezione verrà offerta una panoramica generale dei Cloud OS già citati in precedenza nel capitolo introduttivo: OpenStack, OpenNebula ed Eucalyptus.

2.15.1 OpenStack

OpenStack[8] è un sistema operativo *open source* che permette di realizzare sia *Private* che *Public* Cloud. Questo progetto è stato cominciato nel 2010 da Rackspace Cloud[9] e NASA[10] che vi hanno fatto confluire alcuni altri loro progetti, tra cui Swift[11], Nebula[12] [13] e Nova[14], i quali sono successivamente diventati parte di OpenStack, talvolta addirittura trasformandosi in alcuni suoi moduli[15]. Tale Cloud OS infatti è dotato di un'architettura modulare, come mostrato in figura 2.1, i cui componenti principali verranno ora analizzati.

Compute

OpenStack permette ai provider di offrire risorse computazionali on-demand gestendo grandi reti di macchine virtuali. Tali risorse sono accessibili tramite delle API da parte della applicazioni e tramite un'interfaccia Web da parte degli utenti ed amministratori. L'architettura di computazione è progettata per scalare orizzontalmente con grande facilità, appoggiandosi ad hardware di qualità ordinaria, senza la necessità di utilizzare quindi macchine molto costose.

Questo modulo, chiamato *Nova*, supporta gran parte delle tecnologie di virtualizzazione attualmente esistenti; gli hypervisor consigliati in particolare sono XenServer[16] e KVM[17].

[8] http://www.openstack.org
[9] http://www.rackspace.com/cloud
[10] http://www.nasa.gov
[11] http://www.rackspace.com/blog/storage-systems-overview
[12] http://nebula.nasa.gov
[13] https://www.nebula.com
[14] https://code.arc.nasa.gov/nova
[15] http://nebula.nasa.gov/blog/2012/05/29/nasa-and-openstack-2012
[16] http://www.citrix.com/products/xenserver/overview.html
[17] http://www.linux-kvm.org/page/Main_Page

Storage

I servizi di storage offerti da OpenStack si suddividono in due moduli. Il primo, *Swift*, fornisce una piattaforma di Storage a oggetti distribuita ed accessibile tramite API che può quindi essere perfettamente integrata nelle applicazioni oppure usata per questioni come il backup o l'archiviazione di informazioni. Il secondo invece, chiamato *Cinder*, permette di gestire dispositivi di memorizzazione di massa virtualizzati connettendoli a specifiche macchine virtuali per aumentarne le capacità e le prestazioni. Inoltre, quest'ultimo modulo permette anche di gestire facilmente gli *snapshot* dei dispositivi, facendo in modo che essi possano essere ripristinati oppure replicati.

Networking

Questo componente, chiamato *Quantum*, è un sistema altamente scalabile e controllabile tramite API per la gestione di reti ed indirizzi IP. Esso è progettato prestando particolare attenzione agli aspetti che permettono di fare in modo che le reti non diventino il collo di bottiglia o, in generale, una limitazione per l'infrastruttura virtuale o le applicazioni.

Quantum permette sia l'assegnazione di indirizzi IP statici che l'adozione di protocolli di configurazione automatica, come il DHCP. Inoltre i cosiddetti indirizzi flottanti, ovvero che possono essere agilmente riassegnati, permettono di redirigere il traffico da una macchina ad un'altra. Inoltre esso è dotato di funzionalità aggiuntive che gli permettono di fornire ulteriori servizi, come il bilanciamento del carico di lavoro, l'individuazione di intrusioni, l'istanziazione di *Virtual Private Network* (VPN) o la creazione di *firewall*.

Dashboard

Il modulo chiamato *Horizon* fornisce agli utenti ed agli amministratori un'interfaccia grafica per accedere, fornire ed automatizzare le risorse. Essa è progettata in modo da essere estensibile, permettendo l'installazione di software fornito da altri in modo da aumentarne le funzionalità, ad esempio tramite tool che permettano di gestire il monitoraggio o l'allocazione dei costi. Essa è implementata come un'applicazione Web ed è rivolta sia agli utenti, per la gestione delle proprie risorse, sia agli amministratori, ai

quali fornisce una visione d'insieme dell'intero Cloud. Tramite essa inoltre gli utenti possono rifornirsi automaticamente di nuove risorse, rispettando i limiti che gli amministratori possono imporre, sempre grazie all'uso della *Dashboard*.

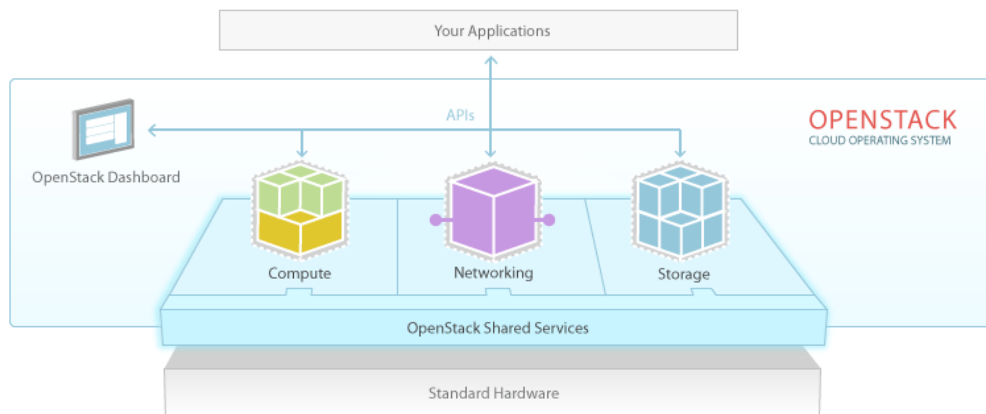

Figura 2.1: Semplice rappresentazione dell'architettura di OpenStack, in cui è possibile vedere i suoi principali moduli.

2.15.2 Eucalyptus

Eucalyptus[18] è un software *open source* che permette di realizzare *Hybrid* o *Private* Cloud. Esso fornisce dei servizi di tipo *IaaS* che vengono esposti come servizi Web. Le sue API sono sviluppate in modo da essere compatibili con quelle di *Amazon Web Services*[19], chiamato anche AWS, un Cloud provider che fornisce anch'esso servizi di tipo *IaaS*, come studieremo in seguito. Grazie a questa scelta progettuale è possibile raggiungere un ottimo grado di portabilità tra le due diverse piattaforme. Ad esempio, un'applicazione scritta per AWS dovrebbe essere in grado di funzionare, se necessario con qualche cambiamento relativamente piccolo, anche su una piattaforma governata da Eucalyptus. Inoltre anche molti dei tool sviluppati per AWS sono compatibili con questo Cloud OS.

[18]http://www.eucalyptus.com
[19]http://aws.amazon.com

Eucalyptus può essere configurato in modo da offrire servizi ad alta affidabilità (*high-availability* o *HA*), in modo da adottare automaticamente dei meccanismi per il recupero dagli errori. Quando questa modalità viene attivata, ogni componente o servizio di Eucalyptus viene copiato in una replica, detta passiva. In caso di errore o fallimento del servizio attivo, quello passivo prende il controllo delle operazioni che esso stava svolgendo, sostituendolo nel suo lavoro di gestione di alcuni aspetti del Cloud OS.

Molti dei sistemi operativi Windows e Linux sono supportati dalle macchine virtuali di Eucalyptus, le quali possono essere memorizzate tramite vari formati: *Eucalyptus Machine Image* o EMI, che è il formato nativo di Eucalyptus, ed *Amazon Machine Image* o AMI, che, come vedremo successivamente, è il formato adottato da AWS per le proprie macchine virtuali. Inoltre anche i formati caratteristici di VMware possono essere gestiti, convertendoli in uno dei due formati precedenti.

Dal punto di vista della virtualizzazione Eucalyptus non si occupa direttamente di questo aspetto, ma supporta molti degli ambienti ed hypervisor oggi più diffusi per virtualizzare l'infrastruttura fisica sottostante.

L'interfacciamento al sistema da parte degli amministratori può svolgersi completamente per mezzo di uno strumento dotato di interfaccia grafica,

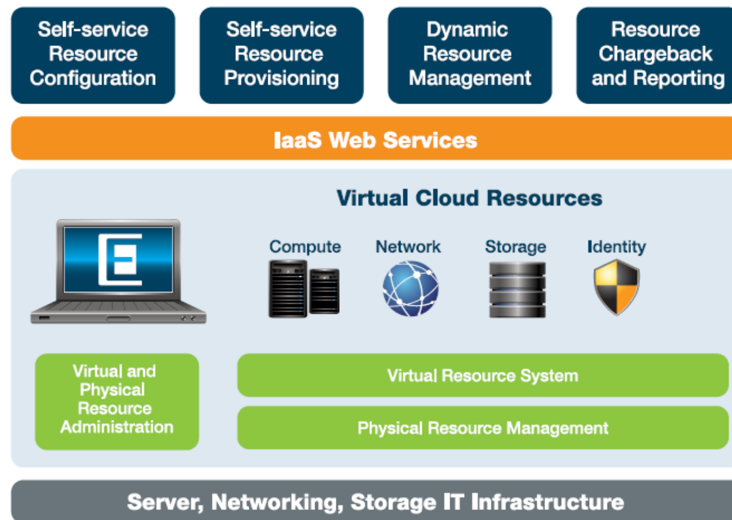

Figura 2.2: Semplice rappresentazione della struttura di Eucalyptus

Eucalyptus Dashboard, che permette di svolgere le attività tipiche di gestione e supervisione di un'infrastruttura Cloud, tra cui la configurazione ed il monitoraggio delle risorse sia virtuali che fisiche. É possibile inoltre specificare delle quote massime utilizzabili da vari utenti o gruppi di utenti riguardanti determinate risorse e metriche, come la memoria allocabile o il traffico di rete. Anche i meccanismi di pagamento tipici del modello pay-per-use sono nativamente supportati.

Eucalyptus infine supporta nativamente dei sistemi di autenticazione ed autorizzazione degli utenti.

2.15.3 OpenNebula

OpenNebula[20] è un Cloud OS completamente *open source*, rilasciato con licenza Apache. Esso presenta buone caratteristiche di interoperabilità, poiché permette di scegliere quali interfacce adottare da un insieme piuttosto vasto, che comprende sia standard ufficiali, come *OGF OCCI*, già citata nelle sezioni precedenti, sia standard *de-facto*, come le API di AWS.

Esso permette di interagire con il sistema sia tramite un'interfaccia a linea di comando (*Command Line Interface*, o CLI) che tramite un tool che mette a invece disposizione un'interfaccia grafica, chiamato *Sunstone*. É disponibile in OpenNebula anche un *Marketplace* per immagini di macchine virtuali già pronte, chiamate *Virtual Appliance*, le quali possono essere subito avviate senza bisogno di ulteriori configurazioni per renderle in grado di essere operative nell'ambiente di tale Cloud OS (esistono in realtà delle differenze tra il concetto di immagine di macchina virtuale e quello di *Virtual Appliance*, che tuttavia non è stato ritenuto importante approfondire in questo contesto). OpenNebula mette anche a disposizione dei servizi di integrazione che permettono di facilitare la migrazione di componenti da un Cloud all'altro, funzionalità utile soprattutto in presenza di *Hybrid Cloud* realizzati per ottenere un'estensione delle capacità di un *Private Cloud.* La figura 2.3 mostra quanto è stato esposto in questo paragrafo.

L'autenticazione degli utenti può avvenire tramite password oppure altri meccanismi più complessi. Le tecnologie di virtualizzazione supportate comprendono Xen[21], VMware[22] e KVM, già citato per OpenStack. Il moni-

[20] http://opennebula.org
[21] http://www.xen.org
[22] http://www.vmware.com

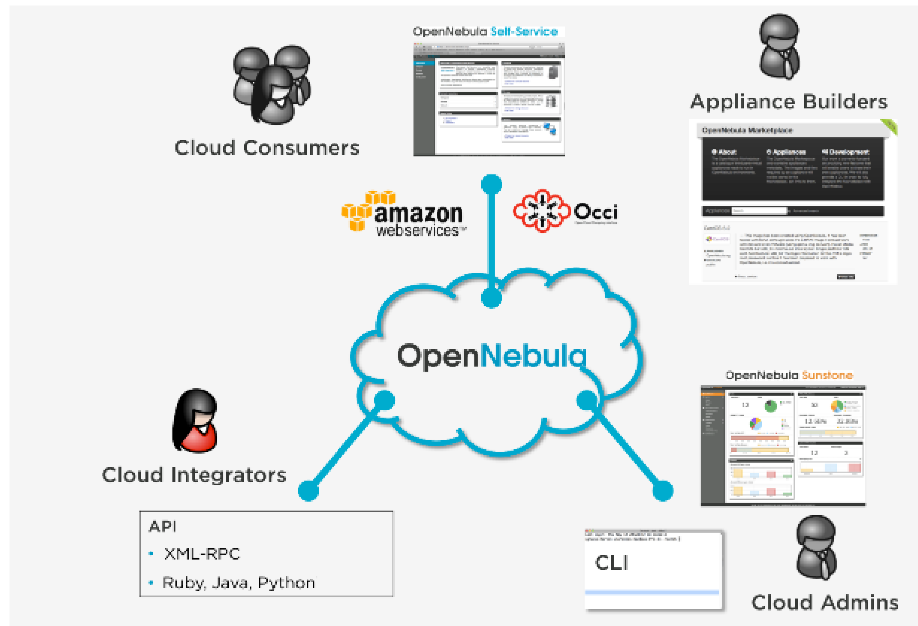

Figura 2.3: Rappresentazione delle caratteristiche principali di OpenNebula

toraggio può essere effettuato sia tramite gli strumenti forniti nativamente da OpenNebula, sia tramite dei tool esterni che quest'ultimo supporta, come Ganglia[23]. Viene infine offerto supporto, per quanto riguarda l'aspetto dei Database, alle tecnologie *MySQL*[24] e *SQLite*[25].

[23]http://ganglia.sourceforge.net
[24]http://www.mysql.com
[25]http://www.sqlite.org

Capitolo 3

Un esempio di IaaS: Amazon Web Services (AWS)

Dopo aver introdotto in generale il Cloud Computing ed averne analizzato più nel dettaglio le caratteristiche del sistema operativo, affrontiamo ora lo studio di un Cloud provider orientato ai servizi di tipo *IaaS*, per valutare quali opportunità esso offre e tramite quali interfacce, permettendoci così anche di capire come le funzionalità di un Cloud OS vengono offerte agli utenti o ai tenant del Cloud.

Nel capitolo introduttivo, analizzando i modelli di servizi, abbiamo detto che spesso essi sono organizzati in una struttura a livelli, in cui ogni livello può accedere solo a quello sottostante. Questa specifica architettura tuttavia non è uno standard né deve necessariamente essere presa come riferimento da parte dei Cloud provider; al contrario, essa nasce come tentativo di strutturare e sintetizzare, per poterlo meglio comprendere, il mondo del Cloud Computing, che, avendo avuto un'evoluzione fortemente spinta dalle forze del mercato, spesso si presenta come una vera e propria giungla agli occhi di coloro che, non conoscendone i concetti fondamentali, cercano per la prima volta di approcciarsi ad esso. La suddivisione rigorosa in *IaaS*, *PaaS* e *SaaS* quindi non è quasi mai riscontrabile in maniera netta e ben delineata, ma anzi spesso un determinato servizio lo si potrebbe pensare come appartenente a categorie diverse, a seconda del punto di vista. Tuttavia ciò non deve indurci a pensare che questo tentativo di dare definizioni più rigorose di quanto non si riscontri nella realtà sia inutile. Infatti, grazie ad esso, possiamo ora affermare che AWS offre servizi principalmente di tipo

IaaS e, anche se talvolta risulta difficile definire la categoria di appartenenza dei singoli servizi, quest'affermazione è per noi carica di significato. Ci permette infatti già di sapere, pur senza ancora aver analizzato un solo servizio tra quelli offerti da AWS, che essi permetteranno all'utente di interagire con le macchine virtuali, che egli dovrà saper gestire la scalabilità tramite l'allocazione di tali macchine virtuali e che sarà lui a dover decidere dove effettuare il deployment delle sue applicazioni e dei loro componenti. Come vedremo in questo capitolo tali deduzioni si riveleranno esatte. Il concetto di *IaaS* quindi, anche se non perfettamente delineato nella realtà, ci ha già permesso di inquadrare AWS e di sapere in linea generale quali saranno i servizi da lui offerti.

Un altro beneficio della suddivisione in *IaaS*, *PaaS* e *SaaS* lo riscontreremo quando affronteremo lo studio di un altro Cloud provider, Google App Engine, il quale è invece molto più orientato a servizi di tipo *PaaS*. Come vedremo esso offre servizi assai diversi da quelli di AWS, tuttavia, la conoscenza dei concetti fondamentali del Cloud Computing e la categorizzazione dei servizi, ci permetterà di analizzarne le funzionalità effettuando anche dei paragoni con AWS che altrimenti, data la loro profonda diversità, non sarebbero possibili. Chi non fosse a conoscenza di tali concetti infatti potrebbe persino non comprendere che essi sono entrambi dei Cloud provider e di conseguenza le fondamentali caratteristiche che hanno in comune, ovvero quelle che discendono dalle piattaforme Cloud.

Introduzione ad AWS

Amazon ha cominciato nel 2006 a fornire servizi di infrastruttura alle aziende esponendoli come servizi Web ed evolvendosi successivamente sempre più verso il paradigma del Cloud Computing. Amazon Web Services (AWS) oggi viene definito da Jinesh Varia, personaggio cardine all'interno di Amazon, come un Cloud che fornisce un'infrastruttura altamente affidabile e scalabile per realizzare applicazioni in grado di raggiungere facilmente le dimenisioni del Web, minimizzando i costi di amministrazione e in maniera più flessibile di quanto ci si possa aspettare da una qualunque altra infrastruttura non Cloud. Attualmente Amazon offre una grande varietà di servizi, che successivamente esporremo. Prima però introduciamo e presentiamo un elenco di benefici come vengono individuati dal team di AWS, che in parte si sovrappongono, confermandoli, a quelli già presentati nel capitolo introduttivo,

mentre altri sono più caratteristici e legati ai servizi offerti da Amazon Web Services.

3.1 Benefici derivanti dall'uso di AWS

I benefici, così come vengono anche presentati nella documentazione di Amazon Web Services, vengono suddivisi in due categorie: la prima mette in luce gli aspetti più legati al mondo del mercato e mostra i vantaggi economici che si possono conseguire scegliendo di adottare alcuni servizi di AWS; la seconda invece comprende i vantaggi più tecnici ed è quindi quella più rivolta agli ingegneri del Software ed altro personale specializzato.

3.1.1 Benefici economici

I vantaggi economici derivanti dall'uso di Amazon Web Services da parte di un'azienda possono essere così riassunti:

- Mancanza di investimenti iniziali
- Infrastruttura dinamica
- Utilizzo efficiente delle risorse
- Pagamento in base all'uso
- Riduzione del tempo necessario a presentare il prodotto sul mercato

Mancanza di investimenti iniziali

Costruire un sistema su larga scala necessita di grandi investimenti iniziali, sia per quanto riguarda l'acquisto dell'hardware, come ad esempio server e componenti di rete, sia per la gestione di tale hardware, che include aspetti come il rifornimento di energia elettrica ed il raffreddamento, sia, infine, per l'assunzione di personale in grado di amministrare e gestire il sistema e le sue sottoparti.

In ambito aziendale tipicamente questo si traduce anche in un notevole aumento dei tempi necessari ad iniziare il progetto, poiché esso deve essere valutato ed approvato con cura, come ogni grande investimento.

Grazie al Cloud Computing è possibile eliminare quasi completamente tutti i costi appena citati.

Infrastruttura dinamica

Può succedere, ed in passato è purtroppo successo svariate volte, che se un'applicazione diventa popolare e si diffonde largamente e se l'infrastruttura dell'azienda non scala sufficientemente, tale azienda diventa vittima del suo stesso successo. Viceversa, se essa investe pesantemente in un'architettura di scalabilità ma i suoi prodotti non divengono popolari, va incontro ad un pericoloso fallimento. Sviluppando l'applicazione tramite il Cloud, grazie alla sua capacità di scalare l'infrastruttura rapidamente e su richiesta, non è più necessario preoccuparsi di fornire in anticipo alla propria infrastruttura la possibilità di espandersi in futuro. Ciò aumenta la flessibilità, riduce i rischi ed anche i costi, poiché si scala solo quando è necessario e si paga solo per ciò che si usa.

Utilizzo efficiente delle risorse

Solitamente gli amministratori di un sistema hanno sempre dovuto occuparsi di procurare nuovo hardware, quando le capacità di quello presente non erano più sufficienti, e di fornirne una migliore utilizzazione quando invece c'erano capacità non utilizzate e per le quali l'azienda ha comunque pagato. Nel Cloud la situazione cambia poiché c'è la possibilità di utilizzare le risorse in maniera più efficiente, richiedendole quando necessario e rilasciandole quando non vengono più utilizzate.

Pagamento in base all'uso

Tramite il modello di pagamento pay-per-use, i tenant del Cloud pagano solo per ciò che effettivamente usano, in contrasto con lo scenario tipico precedente al Cloud in cui spesso venivano pagate risorse poi usate solo parzialmente.

Ciò apre anche nuove opportunità e strategie per ridurre i costi: ottimizzando infatti un'applicazione Cloud in modo che faccia un uso più efficiente delle proprie risorse, ad esempio introducendo un livello di caching, il risparmio è immediato e visibile già nelle bollette successive.

Riduzione del tempi di commercializzazione

Questa caratteristica si applica soprattutto per applicazioni e prodotti che richiedono operazioni computazionalmente molto pesanti, come l'analisi di dati di mercato e la generazione di report. Queste operazioni possono essere accelerate notevolmente sfruttando la parallelizzazione e la dinamicità dell'infrastruttura del Cloud Computing, permettendo al prodotto finito di raggiungere il mercato in tempi molto più brevi senza correre il rischio di perdere preziose opportunità.

3.1.2 Benefici tecnici

I vantaggi di cui possono beneficiare i progettisti e gli sviluppatori di software sono i seguenti:

- Automatizzazione, grazie all'infrastruttura programmabile
- Scalabilità automatica
- Scalabilità proattiva
- Testabilità migliorata
- Capacità di recupero dai guasti migliorata
- Possibilità di dirottare nel Cloud il traffico in eccesso

Infrastruttura programmabile

AWS offre la possibilità di programmare l'infrastruttura, in modo da poter creare ripetutamente ed in maniera automatizzata ambienti di costruzione e sviluppo di applicazioni.

Scalabilità automatica

La scalabilità automatica consiste nella possibilità di scalare le applicazioni, sia in crescita che in riduzione, senza bisogno dell'intervento umano. Questo porta ad una maggiore automatizzazione, flessibilità ed efficienza, anche in termini di velocità di risposta all'aumento di carico di lavoro.

Scalabilità proattiva

Analizzando attentamente il comportamento di talune applicazioni si possono talvolta riscontrare dei pattern ripetitivi nei picchi di carico di lavoro e di traffico. Tali picchi diventano quindi prevedibili ed anticipabili, permettendo di sviluppare opportune pianificazioni dei meccanismi di scalabilità in modo da rendere l'applicazione in grado di reagire in maniera ancora più efficiente.

Testabilità migliorata

La disponibilità di un'infrastruttura altamente virtualizzata e dinamica offre la possibilità di creare istantaneamente l'ambiente necessario per testare l'applicazione ed i suoi singoli componenti in ogni fase del processo di sviluppo. Tale ambiente può essere creato seguendo configurazioni già precedentemente utilizzate e può venire rilasciato una volta terminata la fase di testing dell'applicazione o del componente in esame.

Capacità di recupero dai guasti

Il Cloud offre nativamente varie opzioni che permettono di creare server appositamente per la gestione del *Disaster recovery* e per la memorizzazione persistente dei dati. Inoltre, è possibile sfruttare la proprietà di distribuzione geografica del Cloud per replicare porzioni del sistema in altre località in pochi minuti.

Redirezione del traffico nel Cloud

Servendosi di una buona politica per il bilanciamento del carico è possibile creare un'applicazione esterna al Cloud che sia tuttavia in grado di gestire sovraccarichi di traffico e richieste, redirigendo ciò che non riesce a smaltire nel Cloud.

3.2 Elasticità secondo AWS

Riportiamo ora alcune analisi e riflessioni riguardanti il concetto di elasticità, caratteristica fondamentale del Cloud Computing, che ci aiutano a

comprendere ancora meglio i benefici che l'adozione di tale paradigma può portare.

3.2.1 Confronto con approcci diversi

Lo scopo di questa sezione è analizzare e confrontare tra loro i diversi approcci che un progettista software può utilizzare per scalare le proprie applicazioni in modo da renderle in grado di soddisfare e servire tutte le richieste che vengono loro inoltrate.

Scalabilità verticale

In questo approccio non ci si preoccupa di fornire all'applicazione un'architettura scalabile ma si effettuano grandi investimenti per potenziare i propri server, cercando così di renderli in grado di smaltire l'accresciuto carico di lavoro.

Questa soluzione tuttavia ha dei grossi svantaggi: innanzitutto è molto costosa; inoltre può succedere che, prima che le nuove risorse siano disponibili, l'incremento delle richieste raggiunga il livello critico e lo superi, eccedendo così le capacità delle vecchie risorse che stanno per essere rimpiazzate e causando la perdita di molte richieste, che si traduce nel caso migliore in malcontento da parte degli utilizzatori.

Scalabilità orizzontale

Questa soluzione è stata la più diffusa, almeno fino a quando non ha fatto la sua comparsa il Cloud Computing. In questo caso si modella un'architettura in grado di scalare orizzontalmente, ovvero su nuovi componenti che vengono di volta in volta aggiunti all'infrastruttura, la quale quindi cresce in maniera più regolare e con investimenti di più piccola portata rispetto al caso precedente. La maggior parte delle applicazioni Web a larga scala diffuse oggi adottano questo tipo di approccio, distribuendo i propri componenti su più server ed impiegando un pattern architetturale orientato ai servizi.

Questo approccio funziona decisamente meglio della scalabilità verticale, tuttavia presenta ancora dei problemi, tra cui la necessità di cercare di anticipare l'aumento di richieste a intervalli regolari allargando man mano l'infrastruttura disponibile. Questo porta quasi sempre ad un eccesso di

capacità, che non vengono completamente sfruttate, ed alla necessità di un monitoraggio manuale costante. Inoltre non permette di gestire i casi in cui l'applicazione subisce il cosiddetto *Slashdot Effect*, che consiste in un enorme ed imprevisto aumento delle richieste che, in ambito Web, tipicamente si verifica quando siti o applicazioni molto popolari e diffusi pubblicano dei riferimenti a siti o applicazioni di dimensioni molto ridotte, costringendoli ad affrontare una mole di traffico che spesso non sono in grado di smaltire.

In figura 3.1 viene riportato un grafico che mostra le differenze tra la scalabilità verticale (scale-up approach), quella orizzontale (scale-out approach) e gli effetti dell'elasticità del Cloud Computing.

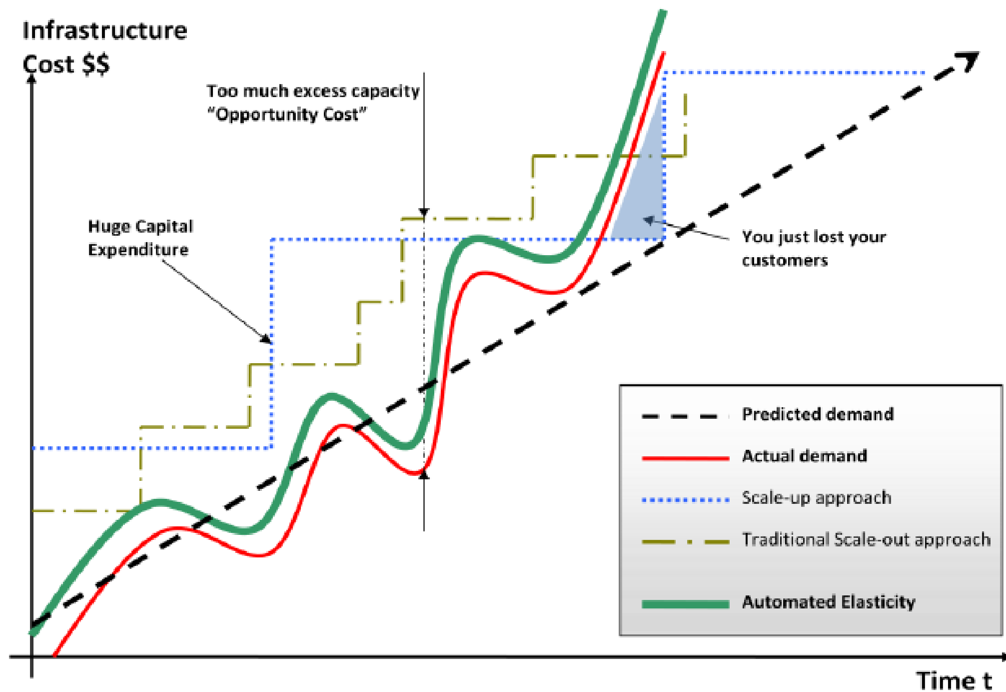

Figura 3.1: Differenza tra i vari approcci alla scalabilità: scalabilità verticale, scalabilità orizzontale ed elasticità.

3.2.2 Considerazioni

Le infrastrutture tradizionali in generale hanno bisogno di predire le dimensioni delle risorse computazionali di cui avranno bisogno per un periodo che può coprire anche qualche anno. Se tali dimensioni vengono sottostimate, le applicazioni distribuite su tale infrastruttura non saranno in grado di gestire il traffico in entrata, mentre se vengono soprastimate si verifica uno spreco di denaro dovuto all'utilizzo non efficiente delle capacità.

L'elasticità tipica dell'approccio Cloud permette all'infrastruttura di essere strettamente allineata con la quantità effettiva delle richieste che vengono ricevute dall'applicazione, aumentando quindi l'efficienza delle risorse e riducendo i costi.

L'elasticità è quindi una proprietà fondamentale dei sistemi Cloud. Essa è definita come la capacità di scalare le risorse computazionali, sia in espansione che in riduzione, facilmente il più rapidamente possibile. Per poter sfruttare al massimo le potenzialità del Cloud Computing è quindi necessario capire appieno il concetto di elasticità per poterlo innestare all'interno dell'architettura delle applicazioni.

Le applicazioni tradizionali sono scritte per essere eseguite su un'infrastruttura rigida, fissa e fornita anticipatamente. Solitamente infatti non vengono aggiunti o rimossi dei server con frequenza giornaliera. Di conseguenza la maggior parte delle architetture di applicazioni tradizionali non sono in grado di gestire la rapida espansione o riduzione dell'infrastruttura sottostante. Inoltre non c'è mai stata la necessità di ottimizzare l'utilizzazione dell'hardware, poiché si è sempre ritenuto accettabile che esso non fosse sempre pienamente sfruttato. Da ciò si deduce che l'elasticità non è mai stata tenuta in considerazione nella realizzazione delle architetture di applicazioni tradizionali; finora infatti non è mai stato possibile dotarsi di nuove risorse computazionali nell'arco di pochi minuti.

Nel contesto del Cloud Computing le assunzioni invece devono cambiare profondamente. Esso infatti ottimizza ed accelera di notevoli ordini di grandezza il processo di acquisizione delle risorse necessarie, poiché le applicazioni stesse possono richiedere in maniera automatica tali risorse pochi minuti prima di poter cominciare ad utilizzarle. Diventa inoltre possibile rilasciare le risorse che non vengono utilizzate per minimizzare i costi.

Concludendo quindi, l'elasticità dovrebbe diventare uno dei requisiti non funzionali di maggior rilievo nell'applicazione ed una proprietà della sua

architettura.

3.3 I servizi di AWS

Una volta comprese le motivazioni ed i vantaggi che possono derivare dall'utilizzo delle tecnologie Cloud, analizziamo ora i vari servizi che Amazon Web Services mette a disposizione per poter sfruttare al meglio le proprietà del Cloud. Essi sono accessibili, secondo lo standard dei protocolli SOA, tramite l'uso di metodi HTTP ma vengono anche forniti degli SDK per molti dei linguaggi più diffusi, tra cui Java, PHP, Python, C# e tutto l'ambiente .NET, che includono API di più alto livello.

3.3.1 *Amazon Elastic Compute Cloud*

Noto anche come *Amazon EC2*, è un servizio Web che fornisce capacità computazionale ridimensionabile tramite il Cloud, ovvero fornisce il controllo e l'accesso alle macchine virtuali ed è quindi la base per lo sviluppo di applicazioni che vengono eseguite completamente nel Cloud.

Esso dà la possibilità di personalizzare le proprie macchine virtuali scegliendo il sistema operativo, i software applicativi e le impostazioni di configurazione desiderate. Tutto ciò viene poi strutturato e memorizzato tramite l'immagine di tale macchina virtuale, chiamata *Amazon Machine Image* o AMI.

É possibile usare le AMI per creare più istanze virtuali tramite delle chiamate standard al servizio Web; allo stesso modo è possibile fermare tali istanze, raggiungendo in tal modo molto semplicemente una prima forma di scalabilità, sia in crescita che in riduzione.

Le istanze possono essere di tre tipi diversi, a seconda del modello di pagamento utilizzato:

On-Demand in cui il pagamento avviene in base all'utilizzo che ne viene fatto ogni ora;

Reserved che permettono pagando solo una piccola quota iniziale senza costi ulteriori legati all'uso, di usare un'istanza con delle ridotte capacità;

Spot in cui le risorse non utilizzate vengono rimesse a disposizione del sistema, anche mentre la macchina è ancora attiva, permettendo ulteriormente di ridurre i costi.

Le istanze possono essere lanciate in determinate regioni geografiche. Ogni regione contiene al suo interno più *Availability Zone*. Esse sono locazioni distinte progettate appositamente per essere isolate in caso di fallimento le une dalle altre e per fornire connessioni di rete economiche e a bassa latenza con altre *Availability Zone* presenti nella stessa regione geografica.

3.3.2 *Elastic IP* e *Amazon Elastic Block Storage*

Elastic IP è un servizio che permette di allocare indirizzi IP statici ed assegnarli a determinate istanze. Ciò può essere fatto manualmente oppure tramite programmazione.

Amazon Elastic Block Storage invece, spesso indicato anche con *Amazon EBS*, permette di creare spazi di memorizzazione persistente collegati tramite la rete ad un'istanza di *Amazon EC2*.

3.3.3 *Amazon CloudWatch* e *Auto-scaling*

Amazon CloudWatch è un servizio che rende possibile il monitoraggio di un'istanza di *Amazon EC2* per rendere visibile l'utilizzo delle risorse, le prestazioni e i pattern delle richieste. Tra le metriche messe a disposizione da Amazon sono presenti l'uso della CPU, le letture e scritture su disco e il traffico di rete.

Auto-scaling invece è un servizio che permette di scalare automaticamente le capacità del sistema in funzione dell'avverarsi di determinate condizioni, basate sulle metriche di cui sopra. Tale servizio può essere quindi facilmente programmato grazie a *Amazon CloudWatch*.

3.3.4 *Elastic Load Balancing*

Questo servizio permette di distribuire il traffico in entrata tra le varie istanze creando un *elastic load balancer*.

3.3.5 *Amazon Simple Storage Service*

Questo servizio, al quale spesso ci si riferisce con *Amazon S3*, è un sistema di memorizzazione dei dati persistente ed altamente distribuito. Le informazioni possono essere memorizzate in grandi quantità e sono strutturate in oggetti, i quali vengono contenuti nei cosiddetti *bucket*. L'accesso a questi oggetti può essere fatto in qualunque momento tramite un'interfaccia Web standard, che permette di interagire con *Amazon S3* per mezzo dei metodi dell'HTTP.

Copie degli oggetti possono essere distribuite e gestite come cache in 14 diverse posizioni, ovunque nel mondo, creando una cosiddetta *distribution* tramite il servizio *Amazon CloudFront*, che si occupa di gestire la distribuzione dei contenuti.

Amazon S3 può essere usato come servizio anche al di fuori di un contesto propriamente Cloud, come d'altronde anche gli altri servizi di memorizzazione che affronteremo più avanti. Un'applicazione può infatti decidere di appoggiarsi ad esso come servizio esterno, pur non essendo essa stessa eseguita nel Cloud.

3.3.6 *Amazon CloudFront*

Si tratta di un servizio per la distribuzione e consegna dei dati. Esso permette di creare copie degli oggetti di *Amazon S3*, organizzandole in quella che viene appunto chiamata una distribuzione, e di gestirle come *cache*, memorizzandole presso 14 diversi nodi che costituiscono il confine del Cloud di *Amazon* con l'ambiente esterno di Internet. Tali nodi sono quindi i punti che possono minimizzare la distanza con gli utenti finali, riducendo notevolmente anche le latenze di rete.

3.3.7 *Amazon SimpleDB*

Questo servizio offre le funzionalità di base di un Database non relazionale, come ricerche in tempo reale o, più in generale, esecuzione di semplici query. Le informazioni vengono organizzate in domini, i quali consistono in collezioni di *item*, ciascuno dei quali viene descritto da un insieme di coppie attributo-valore. Le query possono essere eseguite su tutte le informazioni presenti in un dominio.

3.3.8 *Amazon Relational Database Service*

Indicato spesso con *Amazon RDS*, è un servizio che permette di creare, interagire e scalare facilmente un Database relazionale nel Cloud. É possibile lanciare istanze di un Database alle quali ci si interfaccia tramite il linguaggio MySQL, senza preoccuparsi della ordinaria amministrazione del Database, che include operazioni come l'aggiornamento del DBMS o di sue componenti ed la creazione di copie di backup.

3.3.9 *Amazon Simple Queue Service*

Chiamato anche *Amazon SQS*, è un servizio affidabile ed altamente scalabile in grado di gestire i messaggi scambiati tra applicazioni diverse o tra componenti di una stessa applicazione tramite l'uso di code distribuite.

3.3.10 *Amazon Simple Notifications Service*

Noto anche come *Amazon SNS*, permette di gestire facilmente notifiche e segnalazioni ad applicazioni o utenti secondo il modello *publish-subscribe*, tramite la creazione di *Topic*.

3.3.11 *Amazon Elastic MapReduce*

Questo servizio permette di utilizzare il modello di programmazione MapReduce (non l'omonima implementazione sviluppata da Google, già incontrata durante lo studio di Bigtable) per l'elaborazione parallela di grandi moli di dati. L'implementazione di questo modello, supportata tramite *Amazon EC2* e *Amazon S3*, è *Apache Hadoop*[1].

3.3.12 *Amazon Virtual Private Cloud*

Questo servizio, nominato anche *Amazon VPC*, consente di estendere la rete proprietaria di un'azienda tramite una Private Cloud contenuta in AWS. Esso utilizza l'insieme standard di protocolli IPsec per creare una connessione sicura tramite gateway dell'azienda e gateway di AWS.

[1] http://hadoop.apache.org/

3.3.13 *Amazon Route53*

Amazon Route53 è un DNS altamente scalabile che permette di gestire i propri domini tramite la creazione di *HostedZone*.

3.3.14 *AWS Identity and Access Management*

Noto anche come *IAM* è un servizio che permette ai tenant di AWS di creare i propri utenti, ciascuno con le proprie credenziali, e di gestirne i permessi tramite il proprio account AWS. IAM viene nativamente supportato da tutti gli altri servizi, di conseguenza le API che li forniscono, e quindi le applicazioni che con tali API sono state scritte, non cambiano se si decide di fare uso di IAM.

La figura 3.2 mostra come i servizi sono strutturati, in particolare a quali di essi le applicazioni possono accedere direttamente e quali di essi accedono direttamente all'infrastruttura sottostante.

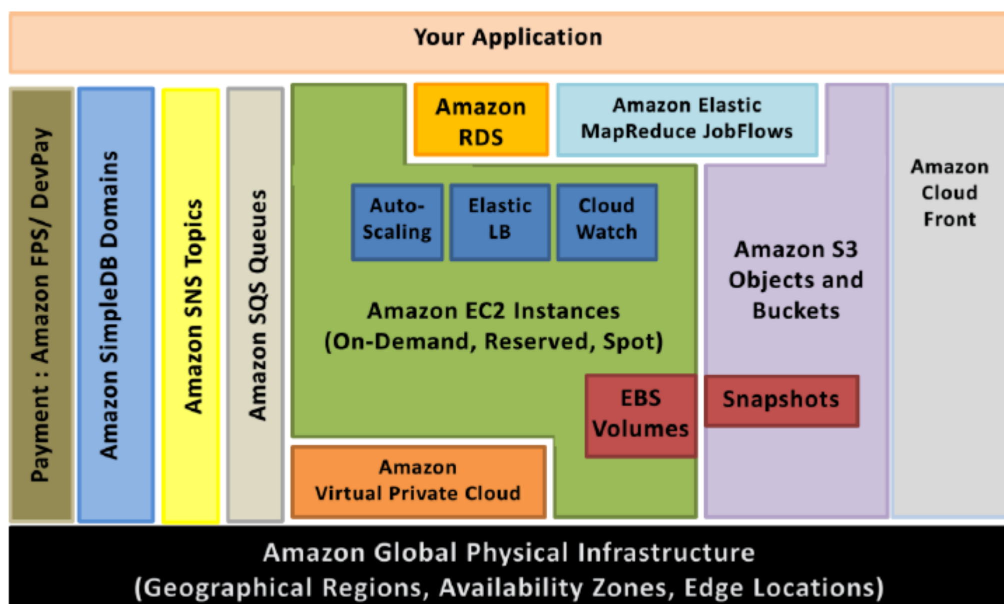

Figura 3.2: Struttura dei servizi di Amazon

Capitolo 4

Persistenza e scalabilità dei dati - Bigtable

4.1 Panoramica generale

Nel capitolo introduttivo abbiamo già accennato alla questione della memorizzazione persistente delle informazioni nel Cloud Computing ed abbiamo visto come le soluzioni messe in campo non comprendono solo i classici Database relazionali. Abbiamo poi visto nel capitolo precedente che anche Amazon Web Services offre soluzioni di diverso tipo per quanto riguarda lo storage dei dati, in particolare il servizio *Amazon SimpleDB*, che fornisce l'accesso ad un Database non relazionale. Appare ora quindi necessario ed importante affrontare questa tematica in maniera più approfondita cercando di capire il motivo per cui le soluzioni ed i modelli che da molto tempo vengono utilizzati nell'ambito della memorizzazione persistente delle informazioni, anche se non vengono completamente sostituiti, sono quantomeno messi in discussione. La gestione dei dati infatti, sia nel Cloud che, più in generale, nei sistemi altamente distribuiti e scalabili, induce a prendere in considerazione modelli dei dati differenti da quello relazionale. Questo, nell'ambito della ricerca informatica, è un campo che sta attualmente sperimentando una grande evoluzione, di conseguenza è facile notare, a chiunque cerchi di informarsi, che tuttora mancano standard e soluzioni comunemente accettate e condivise, come è stato finora per i Database relazionali. Aziende leader nel settore hanno opinioni contrastanti e investono i loro capitali in ricerche che vanno in direzioni diverse, talvolta anche opposte tra loro.

Ad esempio, GigaSpaces[1] sostiene che la scalabilità può essere raggiunta solo con l'adozione di un modello dei dati completamente diverso da quello relazionale, mentre Base One[2] cerca di ottenere la scalabilità a partire proprio da quest'ultimo modello. In generale, le soluzioni atte al raggiungimento della scalabilità con RDBMS sono centrate sul concetto di *sharding*, ovvero sul partizionamento orizzontale dei database. Ciò che sembra emergere tuttavia dagli studi attuali è che effettivamente la scalabilità migliore può essere ottenuta tramite sistemi non relazionali, che tuttavia presentano problematiche di altro tipo, tra cui ad esempio:

- modelli di consistenza meno rigidi; tipicamente infatti viene offerto supporto per operazioni BASE (*Basic Availability*, *Soft-state*, *Eventual consistency*) piuttosto che ACID (*Atomic*, *Consistent*, *Isolated*, *Durable*), anche in virtù del *CAP theorem*[3], o teorema di Brewer, che afferma che un sistema distribuito non può avere contemporaneamente tutte e tre le caratteristiche di consistenza, disponibilità, e tolleranza ai guasti di una singola parte, ma può solo garantirne al massimo due;

- mancanza di supporto per operazioni complesse come quelle esprimibili nel linguaggio dell'algebra relazionale o nei linguaggi della famiglia dell'SQL;

- una maggiore complessità di comprensione e di utilizzo, soprattutto per chi, per anni, ha trattato solamente con il modello relazionale.

In particolare, in virtù di quest'ultimo punto, ultimamente stanno nascendo molti progetti di ricerca che tendono ad unire l'usabilità e le funzionalità di database relazionali con la scalabilità di quelli non relazionali. Come esempio possiamo citare Google F1, che si basa sul precedentente progetto Google Spanner, un database relazionale distribuito sviluppato da Google, e che cerca di unire caratteristiche e funzionalità dei Database relazionali e non relazionali. Forse in un prossimo futuro sarà quindi possibile sviluppare architetture di dati per le applicazioni utilizzando il modello relazionale che finora è stato studiato ed appreso dalla maggior parte delle persone che lavorano in questo campo dell'informatica, tuttavia non è questa la situazione

[1] http://www.gigaspaces.com
[2] http://www.boic.com
[3] http://en.wikipedia.org/wiki/CAP_theorem

attuale e potrebbero esserci sviluppi futuri non ancora previsti. Abbiamo per questo ritenuto importante approfondire un modello di dati differente, come quello di Bigtable, sistema descritto dettagliatamente in [2] e riassunto in questo capitolo.

4.2 Introduzione a Bigtable

Bigtable è un sistema di memorizzazione persistente distribuito, sviluppato da Google e non distribuito esternamente alla loro infrastruttura, a cui tuttavia essi forniscono accesso come un servizio tramite apposite API di Google App Engine. Si appoggia, al livello sottostante, al file system proprietario di Google, GFS (*Google File System*).

Bigtable è progettato per gestire informazioni strutturate e per scalare fino a grandi dimensioni: in termini di memoria si parla di svariati PetaByte di dati ed in termini di server si arriva fino a migliaia di macchine.

Attualmente molti altri progetti e servizi di Google si appoggiano e memorizzano i loro dati in Bigtable, ad esempio, Google Earth, Google Maps, Google Finance, Gmail, YouTube o l'indicizzazione del Web legata al loro motore di ricerca. Per queste applicazioni Bigtable è in grado di soddisfare i più svariati requisiti, sia in termini di dimensioni, che vanno dai pochi byte degli URL alle dimensioni delle fotografie satellitari, sia in termini di latenza, che vanno dall'esecuzione di operazioni che coinvolgono grosse moli di dati da eseguire in modalità batch a richieste da eseguire nel più breve tempo possibile.

Bigtable non supporta un modello relazionale completo, come già affermano, bensì un modello di dati più semplice, basato su tabelle, che permette agli utenti di controllare dinamicamente il formato dei dati e le proprietà legate alla loro localizzazione. L'accesso avviene tramite i nomi delle righe e delle colonne e i dati sono trattati come stringhe arbitrarie, che non vengono interpretate da Bigtable. Per questo è più corretto parlare di dati piuttosto che di informazioni.

4.3 Il modello dei dati

Bigtable è in realtà una mappa distribuita, persistente, multidimensionale e ordinata. Il dominio di tale mappa è la tripla costituita da due stringhe,

che rappresentano il nome della riga e della colonna della tabella, e da un numero intero a 64 bit che rappresenta un istante di tempo (*timestamp*). Questo particolare modello è stato dedotto dagli sviluppatori di Google dopo aver analizzato una serie di potenziali usi che avrebbero potuto essere stati fatti del sistema Bigtable. Come esempio essi riportano *Webtable*, la tabella usata per l'indicizzazione delle pagine Web da parte del motore di ricerca. In tal caso essi usano l'URL della pagina (invertendo l'ordine dei componenti, nella prossima sezione verrà illustrato il motivo di questa scelta) come indice di riga e nella colonna dei contenuti, *contents:*, inseriscono i contenuti della pagina Web nei vari istanti di tempo in cui sono state analizzate. La figura 4.1 mostra quanto fin qui esposto.

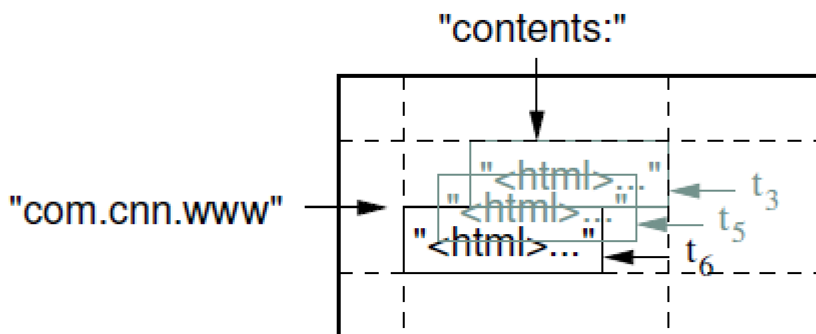

Figura 4.1: Vista parziale di una riga della tabella *Webtable*, in cui si nota come indice di riga l'URL rovesciato della pagina ed i suoi vari contenuti in istanti di tempo diversi.

4.3.1 Righe

La chiave di riga in una tabella è costituita da una stringa arbitraria, di lunghezza massima di 64KB (sperimentalmente hanno osservato che in realtà la lunghezza della chiave di riga, per la maggior parte dei client che usano Bigtable, si assesta attorno ai valori dai 10 ai 100 byte). Ogni lettura o scrittura effettuata tramite un'unica chiave di riga è atomica, a prescindere dal numero di colonne coinvolte nell'operazione. Questa scelta architettura-

le permette agli utenti di comprendere meglio il comportamento del sistema a fronte di più letture e scritture concorrenti effettuate sulla stessa riga.

L'insieme delle righe di una tabella è automaticamente e dinamicamente partizionato, quindi l'utente non ha un controllo diretto sulla distribuzione dei dati. Tuttavia le righe vengono ordinate alfabiticamente in base al loro indice di riga: questo permette un controllo, seppur non totale, su come i dati vengono sparsi sulla rete. Ciò significa che letture su piccoli insiemi di righe adiacenti tipicamente richiedono la comunicazione di poche macchine e sono quindi più efficienti. I client possono sfruttare questa proprietà per ottimizzare l'esecuzione dei propri programmi. I partizionamenti di righe che vengono creati dal sistema sono detti *tablet*, e sono quella che viene chiamata unità di distribuzione e di bilanciamento del carico.

In base alla proprietà di ordinamento delle righe, si può ora capire anche perché, nella figura 4.1 la chiave di riga in realtà non è esattamente l'URL della pagina, che, trattandosi della home page della CNN dovrebbe essere *www.cnn.com*, bensì la stringa ottenuta invertendo l'ordine delle componenti di tale URL: *com.cnn.www*: in tal modo infatti le pagine appartenenti agli stessi domini sono raggruppate in righe contigue. Così facendo l'analisi dei domini e delle pagine è molto più efficiente.

4.3.2 Famiglie di colonne

Le chiavi di colonna sono raggruppate in insiemi chiamati famiglie di colonne, o *column families*, che formano l'unità di base per gestire il controllo degli accessi. Tutti i dati contenuti in una *column family* sono generalmente dello stesso tipo. Prima di poter inserire dati in una qualunque chiave di colonna è necessario creare la famiglia di colonne a cui tale chiave appartiene. Una volta creata, qualunque chiave appartenente a tale famiglia può essere usata. Generalmente il numero di *column families* distine in una tabella dovrebbe essere abbastanza ridotto, aggirandosi attorno all'ordine delle centinaia. Inoltre le famiglie di colonne dovrebbero cambiare raramente durante il ciclo di vita di un'applicazione. Al contrario invece, una tabella può avere un numero arbitrariamente grande di colonne.

La chiave di colonna viene costruita seguendo la sintassi *famiglia:colonna*. Esistono in realtà delle restrizioni riguardanti la stringa che identifica la *column family*, sulle quali tuttavia sorvoliamo poiché non è nostra intenzio-

ne concentrarci sui dettagli del sistema, quanto invece su una sua visione globale.

Come esempio di identificatore per una famiglia di colonne, sempre per quanto riguarda *Webtable*, possiamo citare *language*, che contiene le lingue nelle quali una pagina Web è scritta. Tale famiglia di colonne avrà al suo interno una sola colonna, nella quale verranno scritti gli identificatori delle lingue (si noti la non atomicità del contenuto di una singola cella, che contrasta con le ipotesi tipicamente associate al modello relazionale, in particolare con la prima forma normale generalmente applicata ai Database di questo tipo). Un altro esempio potrebbe invece essere la famiglia di colonne *anchor*, che contiene i riferimenti alla pagina. Ogni singola colonna di tale famiglia contiene il testo del riferimento, mentre la sua chiave è composta dall'URL della pagina da cui il riferimento proviene. Questo esempio viene rappresentato nella figura 4.2.

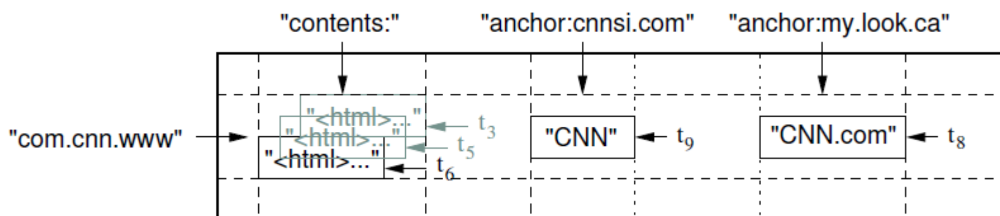

Figura 4.2: Vista parziale di una riga della tabella *Webtable*, in cui si può osservare come vengono gestite le colonne e le *column families* per i riferimenti (*anchor*).

Il controllo degli accessi e l'accounting dei dischi e della memoria vengono entrambi effettuati a livello di *column families*. Nell'esempio di *Webtable*, ciò permette di realizzare varie applicazioni: alcune di esse che raccolgono dati dal Web e li aggiungono alla tabella, altre che invece leggono tali dati senza poterli modificare e creano, derivandoli da essi, delle nuove famiglie di colonne ed altre ancora che invece sono solo autorizzate a leggere alcuni dei dati esistenti, senza inserirne di nuovi.

4.3.3 *Timestamp*

Ogni cella in Bigtable può contenere più versioni di uno stesso dato; tali versioni sono indicizzate in base al loro *timestamp*, che, come già detto, è un intero a 64 bit. I *timestamp* possono essere assegnati automaticamente da Bigtable oppure esplicitamente dalle applicazioni client. Nel primo caso il *timestamp* rappresenta, in microsecondi, l'istante reale in cui il dato è stato inserito, nella migliore approssimazione possibile, tenendo conto che nei sistemi distribuiti è molto difficile, se non impossibile, oltre che spesso inutile, raggiungere una nozione univocamente condivisa da tutti i suoi componenti di tempo assoluto. Applicazioni che necessitino di evitare conflitti infatti devono inserire autonomamente il valore di *timestamp*, al quale possono pervenire utilizzando specifici protocolli, la cui complessità può variare molto in base alle loro esigenze.

Le diverse versioni del contenuto di una cella sono ordinate in maniera decrescente di *timestamp*, in modo che le versioni più aggiornate possano essere lette per prime.

Per rendere meno onerosa la gestione delle versioni dei dati, Bigtable supporta due parametri di configurazione specifici, a livello di *column family* per poter automaticamente eliminare le versioni più vecchie. I client possono infatti specificare sia di mantenere in memoria solo un certo numero di versioni, ovviamente le più aggiornate, oppure di mantenere quelle abbastanza aggiornate, in base alle loro esigenze. Per esempio è possibile specificare di eliminare le versioni dei dati che sono state inserite da più di un giorno o di una settimana.

Nel caso di *Webtable* il *timestamp* delle pagine memorizzate nella colonna *contents:* è settato esplicitamente come tempo in cui le pagine sono state effettivamente visitate dai *crawler* e vengono tenute in memoria solamente le ultime tre versioni più aggiornate.

4.4 API

In questa sezione non verranno descritte nel dettaglio le API di Bigtable, anche perché, come abbiamo già detto, esse non sono fornite pubblicamente in quanto tale sistema non è rivolto a utenti esterni; questi ultimi possono solo utilizzarne alcune funzionalità, fornite loro tramite diverse interfacce. Ciò che si analizzerà quindi sono alcune delle funzionalità generali offer-

te da tali API, astraendo dai dettagli più tecnici, che al momento non ci riguardano.

Le API di Bigtable permettono di creare o eliminare tabelle e famiglie di colonne; consentono inoltre di modificare i meta-dati riguardanti tali tabelle e famiglie di colonne, ad esempio, per gestirne il controllo degli accessi. Inoltre è possibile scrivere o cancellare valori in una tabella, cercare specifici valori in un'unica riga, o iterare un determinato sottoinsieme di dati. Di seguito vengono riportati degli esempi di codice in linguaggio C++, come sono forniti dalla documentazione ufficiale di Google, che fanno riferimento all'esempio già precedentemente usato della tabella Webtable e che sono scritti in maniera semplificata permettendoci di astrarre dai dettagli più tecnici.

In particolare possiamo osservare, nella figura 4.3, come per gestire una serie di aggiornamenti venga fornita l'astrazione *RowMutation*. L'invocazione della funzione *Apply()* invece esegue un aggiornamento atomico dei dati che appartengono ad una stessa riga.

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figura 4.3: Codice C++ semplificato che mostra come è possibile aggiornare il contenuto di una riga di Bigtable inserendo ed eliminando dei dati.

Nell'esempio in figura 4.4 viene fornito il codice necessario ad effettuare una lettura e successiva scansione degli *anchor* di una riga. In particolare, l'astrazione *Scanner* permette di iterare attraverso tali *anchor*. Le iterazioni possono essere effettuate attraverso più famiglie di colonne ed esistono molti meccanismi che permettono di limitare il numero di righe, colonne e *timestamp* prodotti da una scansione. Ad esempio è possibile effettuare dei *pattern matching* con delle espressioni regolari.

Esistono molte altre funzionalità che permettono ai client di manipolare i dati secondo modalità più complesse. Innanzitutto vengono supportate le transazioni a livello di singole righe, che possono essere usate per effettuare letture seguite da scritture in maniera atomica, tramite un'unica chiave di riga. Non vengono invece supportate transazioni su più righe, poiché, come spiegheremo in seguito, analizzando l'utilizzo che viene fatto di Bigtable è stato notato che in realtà questo meccanismo non era necessario. Viene invece offerta la possibilità ai client di eseguire degli script, scritti in un particolare linguaggio sviluppato appositamente, chiamato Sawzall. Agli script, per questioni di sicurezza non è permesso scrivere direttamente dati in Bigtable, tuttavia possono eseguire una serie di trasformazioni e aggiornamenti tramite degli operatori appositamente forniti. Infine Bigtable può essere utilizzato tramite *MapReduce*, un framework, sempre sviluppato da Google, che permette di parallelizzare computazioni che devono essere eseguite su grandi moli di dati.

4.5 Un esempio concreto: Google Earth

Come esempio di applicazione che utilizza Bigtable per memorizzare ed elaborare le proprie informazioni possiamo fare riferimento a Google Earth, il

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figura 4.4: Codice C++ semplificato che mostra come leggere il contenuto di una riga filtrandolo in base ad una particolare famiglia di colonne.

noto sistema che permette agli utenti di accedere ad immagini satellitari della superficie terrestre. Questo sistema, come viene spiegato nella documentazione di Bigtable, sfrutta una tabella per una fase preliminare di elaborazione dei dati ed un insieme di tabelle per servire le informazioni agli utenti.

Durante la fase di elaborazione preliminare le immagini grezze vengono memorizzate all'interno di una tabella, per poter essere successivamente raffinate. Questa tabella contiene circa 70 TeraByte di dati. Ogni riga corrisponde ad un segmento geografico e le chiavi di riga sono state nominate in modo che segmenti geografici adiacenti corrispondano a righe della tabella mantenute vicine. La tabella si compone inoltre di una *column family* in cui vengono tenute effettivamente le immagini: ciascuna di esse in particolare è contenuta in una singola colonna.

La computazione necessaria a svolgere il lavoro preliminare si avvale pesantemente del sistema precedentemente già citato per eseguire computazione intensiva e parallela su grandi moli di dati, *MapReduce*.

Infine, la parte di sistema che si occupa di fornire le immagini agli utenti utilizza una tabella per indicizzare le informazioni mantenute direttamente nel *Google File System*. Tale tabella ha dimensioni abbastanza ridotte, che si aggirano attorno ai 500 GigaByte, tuttavia deve riuscire a servire un'enorme quantità di richieste ogni secondo, minimizzando la latenza: questo obbiettivo è raggiunto distribuendo la tabella attraverso centinaia di server, ciascuno dei quali si occupa di processare le richieste riguardanti il proprio tablet.

4.6 Note conclusive

Durante la progettazione, lo sviluppo ed il mantenimento di Bigtable coloro che hanno partecipato al progetto hanno dedotto dalla loro esperienza delle osservazioni generali sullo sviluppo di sistemi distribuiti, che hanno ritenuto importanti a tal punto da essere incluse nei documenti ufficiali riguardanti Bigtable. Poiché esse possono essere interessanti anche nell'ambito dei sistemi Cloud, verranno di seguito riportate.

Innanzitutto è stata fatta esperienza che i grandi sistemi distribuiti sono vulnerabili a molti tipi di guasti, non solo a quelli tipici delle comunicazioni che avvengono su reti o legati al crash di alcuni componenti. Ad esempio,

sono stati riscontrati dei problemi legati alla corruzione e deterioramento delle memorie e dei componenti di rete, a grandi sfasamenti nei clock dei dispositivi, a macchine difettose, a porzioni di reti troppo grandi o partizionamenti asimmetrici, a bug in sistemi e servizi già testati di cui ci si serve, i quali precedentemente non avevano mai dovuto però far fronte a carichi di lavoro molto grandi ed alla mancata pianificazione della manutenzione dell'hardware. Tutti questi problemi, per i sistemi distribuiti di piccole o medie dimensioni sono così eccezionali e sporadici da non dover necessariamente essere trattati esplicitamente e metodicamente. Spesso è sufficiente risolverli con un intervento straordinario al loro presentarsi; inoltre la loro individuazione e soluzione è abbastanza semplice. Tali ipotesi decadono invece quando ci si trova a dover gestire ed amministrare sistemi distribuiti di grandi dimensioni, come quelli dei Cloud provider, per esempio. Può essere quindi necessario cambiare i protocolli finora utilizzati ed abbandonare determinate assunzioni che i vari componenti fanno su altri.

Inoltre, i grandi sistemi distribuiti, in particolare quelli Cloud, sono per loro natura altamente impredicibili quindi gli interventi di aggiornamento, sia software che hardware, oltre che risultare molto onerosi, possono portare a conseguenze non previste. Questo si traduce nel principio che sarebbe bene ritardare l'aggiunta di nuove caratteristiche o funzionalità finché non è chiaro come esse verranno utilizzate. Ad esempio, inizialmente per Bigtable era stato deciso di fornire supporto per le transazioni che riguardassero anche più righe, tuttavia tale servizio non è stato subito implementato. Successivamente, dopo aver osservato molte delle reali applicazioni che utilizzano Bigtable è stato dedotto che in realtà il supporto alle transazioni era scarsamente richiesto dai client. Inoltre, quando essi lo ritenevano necessario, era principalmente per la creazione ed il mantenimento di indici secondari oltre alla chiave di riga. É stato quindi deciso di aggiungere un meccanismo per supportare specificamente questa funzionalità che, sebbene meno generale di quello per gestire le transazioni distribuite, copre in realtà comunque tutti i casi d'uso ed è sicuramente più efficiente e semplice da mantenere.

Un altro aspetto molto importante che è stato messo in luce durante lo sviluppo di Bigtable è l'importanza di fornire meccanismi in grado di effettuare un appropriato monitoraggio del sistema. Ad esempio, per Bigtable, gli sviluppatori hanno esteso il sistema di RPC (*Remote Procedure Call*) in modo che per un sottoinsieme di chiamate a runtime esso fosse in

grado di mantenere una traccia dettagliata delle azioni avvenute durante l'esecuzione.

Ciò che gli sviluppatori di Google ritengono sia stata la lezione più importante che hanno imparato durante lo sviluppo di Bigtable è l'importanza di una progettazione semplice e chiara. A causa delle dimensioni del sistema e del codice scritto e del fatto che il codice stesso si evolve nel corso del tempo seguendo strade inaspettate, è stato di fondamentale importanza poter disporre di un modello chiaro, implementato in un codice anch'esso il più chiaro possibile. Questo ha permesso di facilitare ed accelerare notevolmente le fasi di debugging e mantenimento del sistema, evitando inutili e dispendiosi sprechi di tempo che troppo spesso purtroppo si rendono necessari per reinterpretare e capire ciò che era stato precedentemente prodotto.

Capitolo 5

Un esempio di PaaS: Google App Engine

5.1 Introduzione al *PaaS*

5.1.1 Limiti dell'*IaaS*

Abbiamo già precedentemente analizzato il livello dell'*IaaS*, mostrandone le caratteristiche e le funzionalità specifiche e focalizzando l'attenzione sulla realizzazione di applicazioni Cloud appoggiandosi a servizi appartenenti a questa categoria. Dovrebbe a questo punto essere chiaro che, nonostante l'elasticità e la scalabilità tipiche dell'ambiente Cloud ed il grande livello di automazione in esso raggiungibile, molto lavoro, soprattutto in fase di progettazione, è necessario per la gestione di aspetti non strettamente legati al contesto dell'applicazione. Ad esempio, è necessario occuparsi della gestione delle macchine virtuali, stabilendo quali componenti esse debbano ospitare, come anche progettare l'architettura migliore, sempre in termini di macchine virtuali, dispositivi di storage e di rete, per offrire un determinato servizio, replicandola poi se si rende necessaria una sua scalatura. Inoltre è necessario, una volta generate le varie repliche, gestirne il carico di lavoro tramite opportuni *Load Balancer*.

Tutto questo lavoro potrebbe essere automatizzato, lasciando che il progettista sia responsabile solamente della realizzazione di una buona architettura per la sua applicazione, che permetta a quest'ultima di sfruttare

l'elasticità dell'ambiente Cloud, come vedremo nel dettaglio nel prossimo capitolo.

5.1.2 Scopo ed orientamento dell'*IaaS*

Il grado di automazione auspicato nella sezione precedente non viene inserito nell'ambito dell'*IaaS* perché tale livello non è strettamente orientato allo sviluppo di applicazioni. Esso nasce per fornire un'infrastruttura virtuale e dinamica ai propri utenti, i quali poi possono usarla, sulla base delle loro necessità, anche per realizzare applicazioni scalabili, ma non solo. L'orientamento anzi che si comincia a percepire oggi, nonostante ciò non sia ancora un fatto affermato e riconosciuto, è che il livello dell'*IaaS* trova la sua migliore applicazione nell'estensione di centri di dati ed infrastrutture esistenti e soprattutto nell'esecuzione di compiti computazionalmente pesanti. Per trovare tracce ed indizi a conferma di questa ipotesi, è sufficiente prestare attenzione al rilievo che viene dato ad aspetti come la computazione massiva parallela (Map Reduce, framework Hadoop) e l'esecuzione di processi in modalità batch, per i quali non manca mai il supporto presso i provider di servizi *IaaS*. Inoltre anche il nome dei servizi chiave che appartengono a questa categoria è significativo da questo punto di vista: *Amazon Elastic Compute Cloud* e *Google Compute Engine* (di cui si parlerà nelle sezioni successive). Entrambi fanno riferimento alla computazione piuttosto che alle applicazioni. D'altra parte il termine stesso *Cloud Computing* si orienta in questa direzione, per ragioni storiche: esso nasce infatti come modo nuovo di concepire e fornire la computazione e solo successivamente viene pensato anche come ambiente d'esecuzione di applicazioni. Ciò avviene contestualmente alla nascita dei servizi di tipo *PaaS*.

5.1.3 Introduzione ed obbiettivi del *PaaS*

Il livello del *PaaS* nasce proprio per fornire un'astrazione ancora superiore agli sviluppatori e progettisti software in modo da permettere loro di concentrarsi sullo sviluppo di applicazioni. Il termine stesso di piattaforma, che può intendersi sia come laboratorio di sviluppo di software che come ambiente d'esecuzione, esprime infatti concetti molto più legati all'ambito delle applicazioni che non a quello della computazione. Esso si compone di una serie di servizi, forniti tipicamente tramite librerie ed API, che permet-

tono di sfruttare le funzionalità dell'infrastruttura sottostante in maniera completamente trasparente; è possibile ad esempio utilizzare servizi di memorizzazione senza dover gestire i dispositivi di storage virtuali ed il loro collegamento alle macchine virtuali. Anzi il concetto stesso di macchina virtuale non è più utilizzato: le applicazioni sviluppate tramite i servizi di *PaaS* non vengono situate in una macchina ma in un ambiente di esecuzione. Dove tale ambiente sarà poi reso effettivamente disponibile è una questione che viene completamente gestita dal sistema. In aggiunta, non c'è nemmeno la garanzia che un'applicazione venga completamente eseguita su un'unica macchina virtuale, anzi, se essa è ben progettata (come raggiungere questo obbiettivo sarà il tema fondamentale del prossimo capitolo) può trarre notevoli vantaggi dal fatto che i suoi vari componenti vengano distribuiti su macchine diverse, permettendo loro di scalare indipendentemente gli uni dagli altri. Infine, l'ultima caratteristica fondamentale del *PaaS* è che la scalabilità può avvenire in maniera completamente automatica, senza che sia necessario al progettista specificare né i meccanismi né le politiche con cui metterla in atto. Ovviamente un certo livello di controllo è necessario, infatti ad esempio Google permette agli utilizzatori di App Engine di specificare dei limiti oltre i quali non scalare le proprie applicazioni, cosicché i loro costi di esecuzione siano sempre mantenuti sotto una soglia nota.

L'introduzione del livello del *PaaS*, se possibile, ha dato un'ulteriore spinta all'*IaaS* orientandolo ancora di più verso gli aspetti legati alla computazione ed all'estensione di infrastrutture esistenti, anche tramite *Hybrid Cloud*, operazione che è tipica di questo livello e che risulta invece molto più complessa da realizzare, se non impossibile, tramite il *PaaS*.

Il diverso orientamento dei livelli *IaaS* e *PaaS* viene ancora più evidenziato se si analizza l'evoluzione che ha subito Google nell'ambito del *Cloud Computing*: essi infatti hanno seguito un percorso che è inverso a quello del mondo del Cloud, introducendo inizialmente dei servizi a livello *PaaS* (sono anzi stati tra i primi ad offrire servizi di questo tipo, una volta capito che era necessario colmare il divario esistente tra l'*IaaS* e lo sviluppo di applicazioni) e solo recentemente hanno cominciato ad offrire servizi anche a livello di *IaaS*, in modo che potessero coprire gli aspetti del *Cloud Computing* che non vengono gestiti dal *PaaS*, come quelli citati nel paragrafo precedente. Quest'introduzione differita dell'*IaaS* ha infatti permesso a Google di specializzarlo ulteriormente nell'ambito della computazione massiva.

5.2 Google Cloud Platform

Google Cloud Platform[1] raccoglie in sé tutti i servizi Cloud offerti da Google. Il nucleo di tali servizi è costitiuto da Google App Engine[2] [3], che rappresenta il cuore del livello *PaaS* di Google. Ulteriori servizi, tra i quali Google Cloud Storage[4] [5], Google Cloud SQL[6] [7] e Google BigQuery[8] [9], sono introdotti per gestire la memorizzazione e l'elaborazione dei dati, interagendo anche con Google App Engine. Infine Google Compute Engine[10] [11], come già citato precedentemente, è il servizio più recentemente aggiunto a Google Cloud Platform, il quale consiste nei servizi di livello *IaaS* di Google.

5.3 Google App Engine

Google App Engine è definito come una piattaforma di sviluppo ed esecuzione di applicazioni, che permette a queste ultime di appoggiarsi e sfruttare l'infrastruttura di Google, la stessa che viene utilizzata per i loro prodotti.

Storicamente, quando nel 2008 comparve sul mercato, il concetto di *PaaS* non era ancora diffuso e ben conosciuto, ed i cloud provider fino a quel momento offrivano solamente servizi di infrastruttura, *IaaS*, oppure applicazioni già pronte, a livello quindi di *SaaS*. Lo scopo con cui Google App Engine è stato creato è quello di trasformare l'infrastruttura in un ambiente, o piattaforma, in grado di ospitare e gestire lo sviluppo di applicazioni, permettendo quindi ai propri utenti di realizzarne di proprie, colmando, come già detto, il *gap* tecnologico esistente tra l'*IaaS* e la creazione di applicazioni.

[1] https://cloud.google.com
[2] https://cloud.google.com/products
[3] https://cloud.google.com/files/GoogleAppEngine.pdf
[4] https://cloud.google.com/products/cloud-storage
[5] https://cloud.google.com/files/CloudStorage.pdf
[6] https://cloud.google.com/products/cloud-sql
[7] https://cloud.google.com/files/CloudSQLDatasheet.pdf
[8] https://cloud.google.com/products/big-query
[9] https://cloud.google.com/files/BigQuery.pdf
[10] https://cloud.google.com/products/compute-engine
[11] https://cloud.google.com/files/GoogleComputeEngine.pdf

5.3.1 Vantaggi di Google App Engine

I principali vantaggi economici che vengono evidenziati per le aziende sono la possibilità di raggiungere automaticamente una grande scalabilità ed un'accelerazione del processo di sviluppo e distribuzione del software, caratteristiche che risultano ancora più nette di quanto non lo fossero nell'ambito dell'*IaaS*, ad esempio, con AWS.

Ciò è dovuto, da un punto di vista più tecnico, al fatto che una notevole mole di lavoro è ulteriormente risparmiata agli sviluppatori e progettisti, poiché di essa si fa carico la piattaforma. In particolare tale lavoro comprende la gestione dell'infrastruttura, la scalabilità delle applicazioni e la tolleranza ai guasti, tutti aspetti gestiti automaticamente da Google App Engine. Non è più necessario ad esempio occuparsi del monitoraggio delle risorse, del bilanciamento del carico, dell'allocazione e dell'inizializzazione di macchine virtuali. Per poter quindi sviluppare applicazioni tramite Google App Engine è sufficiente fornirne il codice e scegliere pochi parametri di configurazione, tra cui ad esempio l'ambiente di esecuzione desiderato.

Infrastruttura

Per quanto riguarda la distribuzione del software, Google App Engine decide automaticamente su quali macchine virtuali e in quali centri di dati posizionare i componenti ed i servizi, sulla base di politiche interne e cercando di ottimizzare vari aspetti, tra cui le prestazioni (massimizzazione dell'efficienza) e l'utilizzo di risorse (minimizzazione dei costi).

Un grado di libertà aggiuntivo che può essere dato agli sviluppatori, per le questioni riguardanti alla localizzazione delle proprie applicazioni, consiste nel permettere loro di decidere se distribuire le proprie applicazioni presso centri di dati localizzati nell'Unione Europea o negli Stati Uniti d'America. Ciò è legato principalmente a questioni legali.

Scalabilità

Anche la scalabilità viene gestita in maniera trasparente allo sviluppatore, poiché, se si verifica un incremento delle richieste, Google App Engine può stabilire di replicare alcuni componenti o servizi, posizionandoli geograficamente ed a livello di rete nei punti che ritiene più opportuni, e di ritirare poi tali repliche una volta che il traffico in entrata torna a ridursi.

5.3.2 Linguaggi ed ambiente di esecuzione

I linguaggi attualmente supportati da Google App Engine sono Python, Java e Go[12], un linguaggio sviluppato da Google. Viene inoltre fornito supporto anche per molti framework e tecnologie legati a tali linguaggi, tra cui, per quanto riguarda Java ad esempio, il framework *Spring*[13], la tecnologia *Java Servlet*[14] e la relativa estensione *JSP*[15] [16]. In aggiunta, grazie alla presenza proprio di Java, è possibile utilizzare anche altri linguaggi che vengono eseguiti su *Java Virtual Machine*, tra cui, ad esempio, Scala[17] ed alcune versioni di PHP (Quercus[18]), JavaScript (Rhino[19]) e Ruby (JRuby[20]).

Gli ambienti d'esecuzione di questi linguaggi non sono tuttavia quelli classici che vengono forniti, ad esempio, nelle distribuzioni desktop, poiché alcune funzionalità non sono supportate per ragioni di sicurezza. Questo discende dalla scelta di eseguire ogni applicazione all'interno di un ambiente molto restrittivo, chiamato *sandbox*, con lo scopo di impedire ad agenti o utenti esterni di accedere al codice o ai dati di tale applicazione.

Tra le azioni che non sono consentite all'interno di una *sandbox* troviamo l'accesso ai file locali, l'utilizzo di *socket* ed, in generale, le chiamate al sistema operativo.

5.3.3 Servizi ed API

Le restrizioni sopra elencate non devono impedire alle applicazioni di svolgere determinati compiti. Per questo vengono introdotti dei servizi e delle API in grado di fornire tutte le funzionalità che verrebbero altrimenti perdute a causa dell'ambiente restrittivo delle *sandbox*. Tali funzionalità sono inoltre presentate con un livello di astrazione maggiore e con interfacce di più alto livello di quelle che generalmente vengono fornite dai classici ambienti

[12]http://golang.org
[13]http://www.springsource.org
[14]http://www.oracle.com/technetwork/java/javaee/servlet/index.html
[15]http://www.oracle.com/technetwork/java/index-jsp-138231.html
[16]Per approfondimenti: http://en.wikipedia.org/wiki/JavaServer_Pages
[17]http://www.scala-lang.org
[18]http://quercus.caucho.com/
[19]https://developer.mozilla.org/en-US/docs/Rhino
[20]http://jruby.org

d'esecuzione e dalle librerie standard. Ad esempio, piuttosto che lavorare con delle *socket* è possibile utilizzare servizi che permettono ai servizi ed ai componenti di scambiarsi messaggi. Oppure, al posto dell'interfacciamento al file system locale, è possibile utilizzare sistemi più avanzati per lo storage delle informazioni, come i database.

I principali servizi e le API fornite riguardano i seguenti aspetti:

- gestione dei *log* delle applicazioni, loro memorizzazione durevole e loro accesso direttamente dalle applicazioni stesse;
- elaborazione parallela di dati tramite il framework *Map Reduce*;
- creazione di indici ed ottimizzazione della ricerca nelle strutture di dati utilizzate;
- autenticazione degli utenti tramite i loro Google Account;
- interfacciamento dell'applicazione a dispositivi *mobile*;
- gestione degli aggiornamenti istantanei ai browser degli utenti, nel caso di applicazioni Web;
- supporto per lo scambio di messaggi;
- supporto per l'invio di *e-mail*;
- possibilità di programmare l'esecuzione di determinati lavori ad intervalli di tempo regolari o in background;
- accesso facilitato ai servizi Web esterni al Cloud, o ad altre risorse presenti in Internet;
- memorizzazione persistente delle informazioni.

5.3.4 Amministrazione del sistema

Sicuramente alleviare gli sviluppatori del compito di gestire personalmente molti degli aspetti che non dipendono dal contesto specifico dell'applicazione è un vantaggio, sia dal punto di vista economico che tecnico. Tuttavia può essere importante fornire un certo livello di controllo sul sistema, permettendo loro così di non perdere completamente la visibilità delle proprie

applicazioni. Se così non fosse, sarebbe impossibile verificare che queste ultime stiano effettivamente funzionando correttamente ed in maniera efficiente e potrebbero verificarsi situazioni in cui il costo di esecuzione di un'applicazione sia molto più alto di quanto potrebbe essere a causa dell'uso scorretto o sub-ottimale di determinate risorse.

Questo livello di visibilità viene fornito tramite una serie di tool che permettono, tra le altre cose, di monitorare le metriche più importanti riferite ad una certa applicazione, come l'uso di CPU e memoria.

5.4 Altri servizi Cloud

5.4.1 Google Cloud Storage

Google Cloud Storage è un servizio per la memorizzazione persistente dei dati che fornisce caratteristiche di grande affidabilità, supporto dei più diffusi standard per la sicurezza, scalabilità virtualmente illimitata e possibilità di utilizzo di complessi tool per l'analisi dei dati. Esso permette ai suoi utenti di concentrarsi sull'accesso, sulla memorizzazione, sulla condivisione e sull'analisi dei propri dati, senza doversi occupare di questioni problematiche spesso legate a quest'ambito, come il mantenimento dei server, la creazione di firewall, la gestione dei dispositivi di storage, dei backup e delle politiche di recupero dai disastri e, ancora una volta, della scalabilità.

Questo servizio si appoggia sul sistema BigTable, già precedentemente studiato, ed è integrabile con tutti gli altri servizi di Google Cloud Platform. In particolare, tramite apposite API, è possibile accedervi direttamente dalle applicazioni realizzate tramite Google App Engine.

Analogamente ad Amazon Simple Storage Service, Google Cloud Storage permette di memorizzare le informazioni strutturandole in oggetti, ciascuno dei quali è contenuto in uno specifico *bucket*.

Precisazioni sul modello di servizio

Come era già stato affermato nel capitolo dedicato ad AWS, non è sempre identificabile in maniera netta ed univoca la categoria di appartenenza di un certo servizio. Ciò in particolare è valido per questo servizio, ma anche per Google Cloud SQL e per Google BigQuery, che verranno analizzati più

avanti. Per quanto infatti Google Cloud Storage sia perfettamente integrabile al servizio *IaaS* Google Compute Engine, che verrà preso in esame nella sezione successiva, esso non può essere ricondotto a tale categoria, poiché nasconde l'infrastruttura sottostante portando l'interazione con l'utente ad un livello più alto. Sarebbe quindi più corretto collocarlo nel livello del *PaaS* poiché infatti, come già detto, offre dei servizi che possono essere utilizzati tramite apposite API per lo sviluppo di applicazioni nel Cloud. Tuttavia può anche essere usato come servizio autonomo, non necessariamente integrato in un'applicazione, grazie alla possibilità di interfacciarsi ad esso tramite dei particolari tool, sia da linea di comando che dotati di interfaccia grafica, i quali quindi sono più orientati al livello del *SaaS*. Considerazioni analoghe valgono anche per Google Cloud SQL e Google BigQuery.

5.4.2 Google Compute Engine

Google Compute Engine, al momento della stesura di questo lavoro, è il più recente insieme di servizi aggiunto a Google Cloud Platform ed attualmente è disponibile solo attraverso un'anteprima con funzionalità ancora abbastanza limitate.

I casi d'uso per cui esso è pensato, come viene suggerito dalla documentazione relativa a tale servizio, sono l'esecuzione di compiti pesanti in modalità batch, l'elaborazione di grandi quantità di dati e la computazione ad alte performance.

Le funzionalità supportate da Google Compute Engine permettono di lanciare ed eseguire macchine virtuali su richiesta, di gestire le loro interconnessioni di rete e di accedere a diversi servizi di storage.

Configurazione delle macchine virtuali

Le macchine virtuali possono essere configurate in modo da avere 1, 2, 4 o 8 core virtuali, ciascuno dei quali con 3.75 GB di memoria RAM. I sistemi operativi per ora supportati sono Ubuntu[21] e CentOS[22].

[21] http://www.ubuntu.com
[22] http://www.centos.org

Storage

Le opzioni di storage tra le quali è possibile scegliere includono la possibilità di usufruire direttamente dalle macchine virtuali del servizio Google Cloud Storage descritto precedentemente, oppure l'utilizzo di particolari dischi virtuali. Questi ultimi si distinguono in effimeri e persistenti. La prima categoria consiste di dischi legati al ciclo di vita della macchina virtuale: quando essa viene spenta infatti il loro contenuto è perduto. I dischi persistenti invece rappresentano dispositivi di storage, collegati tramite rete ed automaticamente replicati, che dal punto di vista delle prestazioni e delle latenze sono comparabili ai classici hard disk locali. Essi possono essere collegati a più macchine virtuali facendo in modo che li condividano in lettura e supportano facilmente la possibilità di catturarne degli *snapshot*.

Rete

Le funzionalità di rete offerte da Google Compute Engine includono la possibilità di collegare le macchine virtuali tra di loro oppure a Internet tramite indirizzi IP esterni. In quest'ultimo caso è possibile assegnare loro degli indirizzi statici, ottenuti tramite Google, oppure degli indirizzi dinamici. Inoltre è possibile configurare semplicemente dei firewall per avere un controllo aggiuntivo sul traffico in entrata ed in uscita dalle macchine virtuali, nonostante la protezione da accessi non autorizzati sia già garantita dai molteplici livelli di sicurezza adottati da Google Compute Engine.

5.4.3 Google Cloud SQL

Google Cloud SQL permette di creare Database MySQL che risiedono all'interno del Cloud e che vengono completamente mantenuti dalla piattaforma. Grazie alla familiarità ed alla diffusione di MySQL è possibile raggiungere un ottimo livello di portabilità delle applicazioni che utilizzano questo servizio, permettendo loro di muoversi con relativa facilità verso il Cloud o al di fuori di esso. Questo servizio è tuttavia indicato solamente per applicazioni di piccole o medie dimensioni, coerentemente con tutto ciò che è stato detto anche nei capitoli precedenti riguardo alla scalabilità dei dati e dei modelli relazionali. Vengono infatti posti dei limiti sulle dimensioni dei Database, i quali non possono superare i 100GB.

La gestione ed il monitoraggio dei Database possono essere effettuati tramite dei tool dotati di interfaccia grafica, mentre la sicurezza, la replicazione e l'affidabilità vengono garantite automaticamente, in maniera trasparente all'utente. Inoltre, è possibile interfacciarsi a questi Database da qualunque applicazione di Google App Engine tramite le apposite librerie, garantendo un'ottima integrazione tra i due servizi.

Tra le varie caratteristiche di questo servizio, c'è la possibilità di scegliere se ospitare il Database nell'Unione Europea o negli Stati Uniti d'America, per le stesse ragioni di cui sopra, e la possibilità di importare o esportare automaticamente i Database tramite alcuni tool.

Esistono anche alcune caratteristiche, tipiche dei Database MySQL, che non sono invece supportate da Google Cloud SQL, tra cui la possibilità di definire funzioni e di eseguire operazioni che coinvolgono il file system locale, il quale viene completamente nascosto, come nel caso di Google App Engine.

5.4.4 Google BigQuery

Google BigQuery è un servizio orientato all'analisi di grandi quantità di dati, che possono avere dimensioni perfino dell'ordine di svariati Terabyte. Questi dati possono essere memorizzati ed elaborati sfruttando le funzionalità di storage e la potenza di calcolo offerte dall'infrastruttura di Google. Applicazioni tipiche di *Data Warehousing* possono così essere eseguite velocemente ed in maniera interattiva, generando report ed analisi nell'arco di pochi secondi, piuttosto che richiedere intere ore di lavoro in modalità batch.

La sicurezza è garantita grazie all'adozione di vari livelli di protezione ed alla possibilità di specificare delle *Access Control List* (ACL) molto dettagliate. Anche la tolleranza ai guasti è assicurata, tramite meccanismi e politiche di replicazione, trasparenti all'utente.

L'accesso a Google BigQuery può essere effettuato tramite apposite applicazioni dotate di interfaccia grafica, oppure tramite l'interfaccia REST. Vengono inoltre fornite librerie per vari linguaggi, tra cui Java e Python, per permettere di integrare le funzionalità di questo servizio in altre applicazioni.

I dati su cui lavora Google BigQuery sono organizzati in tabelle e le interrogazioni possono essere effettuate in modalità sincrona, tramite un

linguaggio sviluppato appositamente per essere il più possibile simile all'SQL.

É possibile infine programmare comunque l'esecuzione di particolari lavori in modalità batch, permettendo così ad alcune interrogazioni di essere eseguite in modalità asincrona.

5.5 Conclusioni

Come conclusione di questo capitolo dedicato al *PaaS*, riportiamo ora alcune riflessioni legate allo sviluppo di applicazioni in ambiente Cloud.

Come abbiamo visto l'introduzione del *PaaS* porta numerosi vantaggi, permettendo agli sviluppatori di astrarre da molte questioni non legate al dominio applicativo che il livello dell'*IaaS* costringeva invece a gestire personalmente.

Tuttavia non è automatica, appoggiandosi al livello *PaaS*, la realizzazione di applicazioni realmente orientate all'ambiente Cloud; ad esempio, il fatto che Google App Engine sia in grado di scalare qualunque applicazione gli venga fornita non implica che tale applicazione sia stata progettata per scalare. Questo si traduce in una serie di problematiche che finora non sono state esplicitate: ad esempio, potrebbe verificarsi che un'applicazione, nonostante venga fatta scalare automaticamente, non ricavi da tale scalatura i vantaggi che ci si aspettava. Oppure potrebbe succedere che alcune applicazioni scalando portino ad un notevole incremento dei costi che non è assolutamente necessario ed è legato ad uno spreco di risorse.

É necessario quindi, anche dotandosi di un provider che fornisca servizi secondo il modello del *PaaS*, conoscere dei principi e delle tecniche che permettano di realizzare applicazioni opportunamente progettate per il Cloud. Queste sono le tematiche che verranno affrontate nel prossimo capitolo.

Capitolo 6

SaaS - Sviluppo di applicazioni Cloud

6.1 Definizione di applicazioni Cloud

Nell'informatica sono molto ricorrenti oggi termini come *Cloud Application*, *Software as a Service*, *Web Application*, *Cloud App*, *Web App* ed altri ancora, spesso purtroppo usati senza sapere esattamente cosa significhino e senza nemmeno curarsi del loro significato, cercando più che altro di fare appello alla risonanza che oggi hanno, soprattutto nel mondo del mercato. Volendo in questo capitolo approfondire particolarmente lo sviluppo di applicazioni nell'ambiente Cloud, dobbiamo per prima cosa cercare di fare chiarezza e definire, se non sempre in maniera rigorosa almeno intuitivamente, ma con precisione, il significato dei concetti sopra esposti.

Innanzitutto, come già abbiamo detto nella sezione introduttiva, il *Software as a Service* è un modello di distribuzione del software, secondo cui esso è interpretato come un servizio ed è tipicamente acceduto da remoto. Per quest'ultimo motivo in particolare spesso si parla di *Web Service*, come nel caso di Amazon. Il mondo del Cloud Computing è fortemente permeato ed influenzato dal concetto di servizio e dalle architetture orientate ai servizi (SOA), infatti tutte le sue risorse e funzionalità, da quelle di infrastruttura a quelle di più alto livello, sono rese disponibili come servizio. É naturale quindi pensare e realizzare anche i componenti software e le applicazioni in maniera orientata ai servizi, da cui discende il modello *SaaS*.

Per comprendere al meglio cosa sia invece una *Web Application* occorre

tornare indietro nel tempo e fare riferimento ai primi anni di diffusione del Web: esso inizialmente era infatti composto principalmente da siti Web, dal contenuto esclusivamente statico. Successivamente è stato esteso grazie alla possibilità di gestire un contenuto dinamico, in grado di cambiare in funzione degli utenti che vi accedevano e delle azioni compiute da tali utenti. Questo meccanismo ha innescato un processo di arricchimento che ha portato le pagine Web dinamiche ad aumentare sempre di più le loro potenzialità, oltre alla loro complessità. Seguendo questa strada ci si è trovati quindi di fronte a pagine in grado di svolgere precisi compiti e di interagire pienamente ed in maniera molto complessa con gli utenti, per cui il termine stesso di pagina non era più stato adatto ad identificare ciò che tale entità effettivamente rappresentava, ovvero un'applicazione completa, eseguita, a differenza delle applicazioni desktop classiche, su un server remoto ed alla quale gli utenti accedevano tramite un comune browser Web. Tutt'ora il mondo delle *Web Application* è in grande evoluzione e si cercano sempre migliori soluzioni e modelli che permettano di sviluppare al meglio tali applicazioni.

É importante notare che le applicazioni Web ed il *SaaS* non identificano lo stesso concetto, per quanto tuttavia, nella maggior parte dei casi, applicazioni realizzate secondo il modello del *SaaS* siano poi distribuite come *Web Application*, ed anche queste ultime siano quasi sempre realizzate secondo il modello di cui sopra. Per quanto si tratti quindi di concetti differenti, in realtà essi sono strettamente accoppiati.

Per quanto riguarda le *Cloud Application*, ancora oggi non esiste una definizione che sia condivisa ed accettata all'interno delle comunità informatiche. Ciò che è importante sottolineare dal punto di vista dell'ingegneria del software tuttavia è proprio ciò che spesso alcuni Cloud provider smentiscono o cercano di nascondere, ovvero che una *Cloud Application* non è una qualunque applicazione eseguita in ambiente Cloud. Al di là delle definizioni che ciascuno potrà dare riguardo a questo termine, nell'ambito di questo lavoro con *Cloud Application* intenderemo un'applicazione progettata, ingegnerizzata e realizzata per essere eseguita su una piattaforma Cloud.

Per quanto riguarda infine il termine *App* esso nasce dalla grande diffusione dei dispositivi *mobile*, ma, per i nostri scopi, non è rilevante individuare sostanziali distinzioni tra applicazioni ed App.

6.2 Principi generali per sviluppare applicazioni in ambiente Cloud

Procediamo ora in questa sezione nell'intento di delineare, in maniera abbastanza generica e descrittiva, quali sono i principi che sarebbe bene tenere in considerazione per lo sviluppo di applicazioni in grado di sfruttare appieno le proprietà e le potenzialità offerte dai sistemi Cloud.

Tali principi nascono dall'esperienza e dallo studio di molti progettisti e sviluppatori di software che ormai da qualche anno hanno cominciato ad interfacciarsi al mondo del Cloud Computing realizzando le prime applicazioni ed analizzandone criticamente la qualità (in particolare, alcuni articoli[1] [2] del blog presente all'indirizzo *http://horicky.blogspot.it* presentano una sintesi molto interessante, da cui è stata tratta una parte del materiale di questa sezione).

Facciamo notare fin da subito che alcuni di questi principi potrebbero non essere applicabili nell'ambiente fornito da un determinato Cloud provider. Ad esempio, verrà più volte detto che può essere necessario conoscere e decidere la localizzazione, sia logica che geografica, di determinati componenti dell'applicazione; scegliendo come Cloud provider Google App Engine tuttavia ciò non sarà possibile. A causa di ciò si è deciso, in questo capitolo, di affrontare anche uno studio legato all'applicazione concreta di vari principi presso le piattaforme precedentemente analizzate.

6.2.1 Processo di sviluppo del software

É bene cominciare questa sezione con alcune precisazioni per quanto riguarda il processo di sviluppo del software: sappiamo infatti che la fase di analisi dei requisiti e la conseguente modellazione iniziale delle entità del dominio dovrebbero essere fatte astraendo il più possibile dalla piattaforma sulla quale poi si andrà a realizzare l'applicazione, anche perché tale piattaforma non dovrebbe essere ancora nota bensì dovrebbe essere scelta nelle fasi successive. Ciò che verrà studiato quindi in questo capitolo non dovrebbe mai influenzare la fase di analisi dei requisiti ma deve casomai

[1] http://horicky.blogspot.it/2009/08/skinny-straw-in-cloud-shake.html
[2] http://horicky.blogspot.it/2009/08/traditional-saas-vs-cloud-enabled-saas.html

essere collocato nelle fasi successive, in particolare in quella di analisi del problema e di progettazione.

La prima di queste due fasi è quella in cui entrano in gioco le piattaforme, infatti una buona analisi del problema è in parte determinata anche da una buona conoscenza delle tecnologie disponibili, che nel caso del Cloud Computing si traduce almeno in una conoscenza delle sue caratteristiche fondamentali già più volte esposte. É in questa fase che gli analisti dovrebbero decidere se avvalersi o meno di una piattaforma Cloud, poiché, lungi dagli obbiettivi di questo lavoro, non si vuole qui affermare che il Cloud sia la soluzione migliore ad ogni problema, ma solo un'opportunità che deve essere attentamente valutata.

Nella fase di progettazione invece trova la propria collocazione quanto sarà esposto in questo capitolo poiché ciò riguarda strettamente le architetture delle applicazioni, le quali vengono effettivamente modellate in tale fase.

6.2.2 Influenza della piattaforma sulle applicazioni

L'adozione di una piattaforma Cloud ha, o può avere, un notevole impatto sulle applicazioni che su di essa verranno eseguite (vedi [9]). La possibilità di un certo software di essere dotato di determinate qualità infatti discende anche dalle proprietà architetturali della piattaforma sottostante e da come tali proprietà sono state sfruttate o mitigate nella fase di progettazione dell'architettura del sistema. Di conseguenza, le proprietà della piattaforma influenzano l'architettura delle applicazioni che su tale piattaforma sono eseguite.

Ruolo del progettista

Questo è il motivo per cui un progettista software dovrebbe sempre conoscere le proprietà della piattaforma sulla quale svilupperà le proprie applicazioni. Egli dovrebbe seguire gli usuali metodi di progettazione per arrivare ad un'architettura in grado di sfruttare le proprietà caratteristiche della piattaforma in modo da garantire che i requisiti funzionali e non funzionali siano propriamente soddisfatti.

Consapevolezza nelle applicazioni

É importante notare che spesso le applicazioni in sé stesse non sono consapevoli di essere eseguite su un'infrastruttura virtualizzata ed elastica e potrebbero quindi non sfruttare appieno le potenzialità della piattaforma. Ad esempio, rilasciare un'applicazione sviluppata secondo i metodi tradizionali tramite una piattaforma altamente scalabile ed elastica come il Cloud non rende tale applicazione automaticamente scalabile. In altri termini, come già detto nella parte introduttiva di questo capitolo, eseguire un'applicazione nel Cloud non la rende automaticamente un'applicazione Cloud.

Migrazione di applicazioni nel Cloud

Quanto detto sopra smentisce in parte un'altra tesi sostenuta da molti fornitori di servizi Cloud nel tentativo di diffondere il più possibile l'utilizzo delle loro piattaforme. Essi infatti ribadiscono spesso che è molto semplice migrare le proprie applicazioni già esistenti nell'ambiente Cloud. Tuttavia tali applicazioni quasi sempre sono state progettate e realizzate per essere eseguite in un ambiente locale, o solo parzialmente distribuito. Di conseguenza, quando esse vengono distribuite su una piattaforma Cloud, spesso portano ad notevole incremento dei costi e ad un drastico calo nelle performance rispetto a quanto era stato promesso. Vedremo ora di seguito come evitare di incorrere in problemi di questo tipo.

6.2.3 Consapevolezza di caratteristiche fondamentali

Tra le proprietà delle piattaforme virtualizzate e Cloud elencate nel capitolo introduttivo, ve ne sono alcune che risultano molto più importanti di altre, per il fatto che non dipendono dall'attuale stato dell'arte, come potrebbe essere per lo scarso isolamento nelle performance, che possiamo immaginare verrà risolto sempre meglio nel futuro, ma che concorrono a plasmare la natura intrinseca di tali piattaforme, come l'elasticità ed il modello di pagamento. Tali proprietà possono essere fatte convergere in una serie di linee guida per lo sviluppo di applicazioni, che verranno ora descritte e spiegate confrontando, per ciascuna di esse, uno scenario tipico in cui si può trovare ad essere eseguita una qualunque applicazione Web, sviluppata secondo il modello *Saas* e lo stesso scenario traslato nel mondo del Cloud.

Consapevolezza della latenza

Le applicazioni tradizionali sviluppate secondo il modello *SaaS* lavorano solitamente su un unico centro di dati, supponendo una latenza trascurabile tra le varie componenti server. Ad esempio, in una tipica applicazione Web, si tende sempre a considerare rilevante solo il tempo che intercorre tra la richiesta del client e la risposta del server, mentre si pensa sia quasi istantanea, o quantomeno di vari ordini di grandezza più rapida, l'interazione che intercorre tra il server Web ed il server della base di dati.

Tale assunzione non è più valida, poiché tipicamente nel Cloud si ha a che fare con un'infrastruttura spesso molto distribuita geograficamente. Occorre quindi prestare attenzione, quando possibile, a dove l'applicazione ed i suoi vari componenti vengono collocati per evitare frequenti comunicazioni tra parti distanti del sistema.

Conoscenza dei costi

Le applicazioni *SaaS* tradizionali sono eseguite su piattaforme hardware dedicate, in cui l'uso efficiente delle risorse non è considerato eccessivamente importante, entro certi limiti, chiaramente.

Nell'ambiente Cloud invece, poiché i costi sono strettamente legati all'utilizzo delle risorse, occorre prestare molta più attenzione a come vengono impiegate. Inoltre occorre conoscere il modello dei costi per risorse differenti, in modo da poter adottare la giusta strategia di utilizzo, che permetta di minimizzare i costi delle operazioni. Ad esempio, si potrebbe valutare più economico mantenere in memoria dei risultati parziali piuttosto che ricalcolarli quando sono richiesti, oppure condividerli tra vari componenti del sistema inviandoli sulla rete, piuttosto che farli ricalcolare da ogni componente individualmente.

Sicurezza

Tipicamente le applicazioni tradizionali lavorano su centri di dati affidabili, in cui si conosce bene anche il perimetro tra essi e l'ambiente esterno.

Nel Cloud invece tale perimetro può essere molto più confuso e mutevole nel tempo, soprattutto nel caso di una Hybrid Cloud. Occorre prestare una particolare attenzione a dove i dati vengono memorizzati, a quali provider rivolgersi ed eventualmente crittare i dati più sensibili.

Elasticità

Le applicazioni tradizionali solitamente non si trovano a dover affrontare crescite o riduzioni su larga scala delle risorse computazionali e non sono progettate per gestire propriamente la distribuzione dei dati su macchine appena aggiunte al sistema, né la loro ridistribuzione sulle macchine rimaste. Questo si traduce spesso in un grande spreco di risorse, in particolare in un uso molto inefficiente della banda, che porta ad elevati costi e scarse performance.

Per applicazioni orientate alle piattaforme Cloud sono necessari protocolli più sofisticati per la distribuzione dei dati, che siano in grado di gestire efficacemente la crescita e la riduzione, dinamica o su richiesta, del sistema.

6.2.4 Gestione e riduzione del traffico

É stato osservato, in questi primi anni di diffusione del Cloud Computing, che il problema principale delle applicazioni che si avvalgono di tale tecnologia è l'uso della banda. In particolare ciò si traduce in:

- scarse performance dovute a latenze troppo elevate
- costi eccessivi per eseguire l'applicazione nel Cloud

Rispettando i dovuti accorgimenti è tuttavia possibile ridurre la quantità di dati trasferiti, quando non è strettamente necessario, riducendo quindi i costi d'esecuzione ed aumentando le performance.

Componenti *stateless*

In fase di inizializzazione occorre fornire all'applicazione tutte le informazioni necessarie per poter essere lanciata. Tuttavia non sempre tali informazioni sono tutte indispensabili ed alcune di esse potrebbero addirittura rendersi necessarie solo a causa di una particolare scelta architetturale. É bene quindi, in fase di progetto, modellare i componenti applicativi in modo che abbiano bisogno del minor numero di informazioni per essere creati: accentuando questa caratteristica, si converge al caso di componenti *stateless*.

Creazione dei dati

Chiaramente, non è sempre possibile modellare tutti i componenti in modo che siano *stateless*: talvolta le informazioni di inizializzazione sono indispensabili a causa della loro natura e di quella del componente.

Tuttavia è bene valutare se tali informazioni devono necessariamente essere inviate all'applicazione tramite la rete, eventualmente persino dall'esterno dei confini del Cloud, oppure se non sia possibile spostare il processo di creazione di tali informazioni direttamente nel Cloud, minimizzando in questo modo il traffico.

Anche questa soluzione non è sempre applicabile, perché possono esistere informazioni che per loro natura non possono essere computate o dedotte da altre, ma è necessario che vengano reperite dall'esterno. Anche quando fosse applicabile tuttavia è bene valutare anche questioni legate alla sicurezza.

Mobilità del codice

Tipicamente siamo abituati a modellare le applicazioni in maniera tale che quando il codice necessita di informazioni si rivolge ad una base di dati, o a qualunque componente che sia incaricato di gestire e mantenere i dati, chiedendo, esplicitamente o implicitamente, che questi ultimi gli siano inviati.

É opportuno valutare se invece non sia il caso di ribaltare la situazione, facendo in modo che, quando determinate informazioni sono richieste da una certa porzione di codice, sia quest'ultima a spostarsi verso i dati e non il contrario. Occorre comunque sincerarsi che la macchina dove il codice è stato spostato abbia la potenza di calcolo necessaria ad eseguirlo.

Partizionamento e ridistribuzione dei dati

L'ambiente Cloud è caratterizzato da un forte parallelismo che sarebbe bene sfruttare per migliorare le performance dell'applicazione, soprattutto nel caso di operazioni computazionalmente pesanti. Ciò implica la necessità di partizionare le informazioni per assegnarle ai componenti che si occuperanno di processarle parallelamente. Chiaramente il partizionamento deve essere eseguito in funzione della computazione che è necessario eseguire su tali informazioni e può succedere che fasi successive della computazione possano

rendere necessario un ripartizionamento dei dati, anche lungo dimensioni diverse.

Poiché l'operazione di partizionamento e distribuzione dei dati può essere molto costosa in termini di traffico, è bene cercare di eseguire tutte le computazioni possibili su un certo partizionamento prima di cambiarlo, in modo da minimizzare il numero di operazioni di ripartizione e ridistribuzione dei dati.

Inoltre, considerando la grande elasticità delle piattaforme Cloud, può succedere che nuove macchine vengano aggiunte a quelle già assegnate all'utente corrente, oppure che alcune di esse gli vengano sottratte. Questa eventualità implica un ulteriore fase di ripartizionamento e ridistribuzione dei dati, che andrebbe effettuata cercando di minimizzare il traffico di rete. Questo è possibile utilizzando particolari algoritmi che riescono a ridurre la quantità di dati che devono essere spostati, come il *consistent hashing*.

Sempre a causa della grande dinamicità del Cloud, possono concretizzarsi degli scenari in cui continuamente vengono allocate o rimosse risorse computazionali. In questo caso è necessario fare in modo che le operazioni di partizionamento e ridistribuzione dei dati non blocchino l'esecuzione dei task principali dell'applicazione, bensì vengano eseguite parallelamente ad essi. La distribuzione quindi deve essere interpretata come un processo di miglioramento delle performance sempre in esecuzione, non come una fase di riorganizzazione che segue in maniera sequenziale il cambiamento dell'architettura del sistema, precedendo e bloccando l'esecuzione dell'applicazione.

Posizionamento componenti

I Cloud provider tipicamente fanno pagare il traffico che attraversa i confini della loro infrastruttura Cloud più di quello che resta interno. Quando si collocano i componenti è bene quindi analizzare quali sono i flussi comunicativi più intensi e fare in modo che siano il più possibile interni all'infrastruttura Cloud.

Variazione dei *pattern* di comunicazione

I pattern di comunicazione possono cambiare mentre l'applicazione è in esecuzione, in base alla natura dei componenti e dell'applicazione stessa e agli

scenari in cui si trova ad essere eseguita. É bene quindi monitorare costantemente tali pattern e prendere in considerazione l'ipotesi di migrare i dati verso i componenti che li usano più frequentemente in modo da minimizzare il traffico necessario al loro accesso. In particolare occorre analizzare i costi di tale migrazione, valutando quali dati migrare, e stimare i costi legati al traffico lungo i pattern comunicativi in questione, sapendo comunque che in generale non è possibile conoscere con assoluta certezza per quanto tempo tali pattern manterranno i picchi di traffico rilevati.

Inoltre se i dati vengono effettivamente spostati nasceranno probabilmente nuovi pattern, o vecchi pattern vedranno aumentare il proprio costo, mentre se vengono copiati, si genererà del traffico legato ai processi necessari a mantenere la consistenza tra le repliche, secondo il modello adottato.

Uso della *cache*

Le tecniche di *caching* possono aiutare notevolmente ad abbattere i costi ed aumentare le performance legate alle latenze di rete, soprattutto in presenza di dati relativamente statici.

Accesso diretto ai dati

Quando è opportuno, in base anche alle politiche di sicurezza, l'accesso diretto ai dati può minimizzare il traffico necessario. Secondo il modello suggerito dal SOA, i dati dovrebbero essere incapsulati in componenti dei quali rappresentano lo stato, ed essere accessibili solo tramite determinate API. Questo implica tuttavia lo scambio di almeno quattro messaggi: richiesta verso il componente, richiesta dal componente alla base dei dati e relative risposte. Abilitando invece l'accesso diretto ai dati il numero di messaggi e di conseguenza il traffico, si riducono: si hanno in questo caso infatti solamente la richiesta e la risposta direttamente verso e dalla base di dati, o la parte di sistema che si occupa della memorizzazione e persistenza delle informazioni, qualunque essa sia.

Informazioni sulla latenza

Una tecnica avanzata per la gestione del traffico consiste nel creare una mappa delle latenze tra i vari componenti del sistema, aggiornata dinamicamente dal sistema stesso, che può essere utilizzata dagli amministratori

per apportare modifiche che ritengono opportune oppure può essere fornita all'applicazione, arricchendola della capacità di adattarsi automaticamente alle variazioni dei pattern di comunicazione.

6.3 Pattern architetturali

In questa sezione introdurremo il tema dei pattern architetturali, definendo innanzitutto cosa sono e quale ragione ci ha indotto a prenderli in considerazione. Successivamente mostreremo alcuni dei pattern più importanti per lo sviluppo di applicazioni Cloud. Il contenuto di questa sezione fa riferimento agli articoli [4] e [5].

6.3.1 Definizione e scopo dei pattern

I pattern possono essere definiti come porzioni di conoscenza in grado di descrivere stili architetturali e soluzioni riusabili a problemi ricorrenti, fornendone un'astrazione dai dettagli implementativi e dalle caratteristiche proprie dei singoli casi d'uso. Molto spesso nell'informatica, ma anche al di fuori di essa (infatti i pattern nascono nell'ambito dell'architettura), capita di incontrare problemi che si ripresentano frequentemente, talvolta anche in contesti alquanto diversi. Quando possibile, sarebbe bene, applicando i principi dell'ingegneria del software, realizzare soluzioni che siano direttamente riutilizzabili, ad esempio come librerie o moduli applicativi. Ciò però spesso non è possibile, tra i vari motivi, a causa della differenza dei contesti applicativi, delle tecnologie, dei linguaggi e delle piattaforme disponibili. Sarebbe impensabile, ad esempio, pensare di realizzare un'unica libreria in grado di affrontare il problema dello scheduling sia per un dispositivo mobile, sia per un sistema operativo desktop, che per un Cloud OS. Esistono tuttavia molti aspetti in comune tra le varie soluzioni che possono essere messe in campo ed i pattern servono proprio ad esprimere questa conoscenza: permettono quindi di non dover più volte risolvere lo stesso problema limitandosi solo casomai a reimplementarne la soluzione.

6.3.2 Necessità di pattern nel Cloud Computing

Il mondo del Cloud Computing, essendo ancora molto giovane ma ricco di risorse ed assai promettente anche in campo industriale, sta subendo al

giorno d'oggi un processo evolutivo ancora molto intenso e dinamico. Il problema è che tale processo subisce una pesante spinta da parte di forze provenienti dal mercato che, se da una parte gli forniscono le risorse, anche economiche, per potersi alimentare, dall'altra lo spogliano del rigore e della formalità che sarebbero necessari per portarlo ad un maggiore livello di maturazione. Questo in particolare si traduce spesso in un mascheramento degli aspetti e dei concetti architetturali del Cloud e delle loro implicazioni sulle applicazioni, a favore di caratteristiche da un punto di vista ingegneristico meno rilevanti ma spesso più accattivanti per un pubblico di potenziali tenant del Cloud. Le applicazioni vengono quindi spesso realizzate applicando in maniera implicita, o non applicando per nulla, i pattern architetturali che sarebbero di volta in volta opportuni. I principi architetturali comuni spesso vengono nascosti da una terminologia e da componenti specifici dell'applicazione, e questo le rende spesso anche difficili da confrontare. É quindi importante l'esplicitazione dei pattern usati, per rendere anche le applicazioni e le loro architetture maggiormente comprensibili.

Di seguito riportiamo alcuni dei pattern più diffusi ed importanti che permettono di sviluppare applicazioni Cloud di qualità.

6.3.3 Applicazioni modulari

Le applicazioni monolitiche sono difficili da integrare le une con le altre, inoltre presentano una flessibilità molto scarsa e può essere molto arduo cambiarne alcune funzionalità e comportamenti durante l'esecuzione. Per quanto riguarda l'ambiente Cloud in aggiunta esse sono scalabili in maniera molto poco efficiente, in quanto sarebbe bene poterne scalare le singole funzionalità, non l'applicazione intera.

La soluzione consiste nel suddividere le funzionalità dell'applicazione in più componenti indipendenti, che vengono successivamente integrati per realizzare l'applicazione di partenza. In questo modo l'applicazione è sin da subito estendibile e l'integrazione con altre applicazioni diviene più semplice: in questo modo infatti i confini stessi dell'applicazione vengono determinati dai componenti che ne fanno parte e possono essere quindi modificati facilmente.

Una scelta architetturale molto importante riguarda la suddivisione delle funzionalità in componenti. Se i componenti sono troppo pochi infatti, non si godono dei vantaggi tipici dell'adozione di questo pattern e l'applicazione

torna ad essere poco flessibile e difficilmente integrabile. Viceversa se vengono realizzati troppi componenti adottando un'eccessiva suddivisione delle funzionalità, si perviene ad un sovraccarico legato alle interazioni e comunicazioni necessarie che intercorrono tra i vari componenti, portando l'applicazione a fornire scarse prestazioni e rendendo molto difficoltoso il compito di comprenderne il comportamento e di seguirne i flussi comunicativi.

Per realizzare l'integrazione dei vari componenti può essere utilizzato un linguaggio specifico, come nel caso del *Business Process Execution Language (BPEL)*.

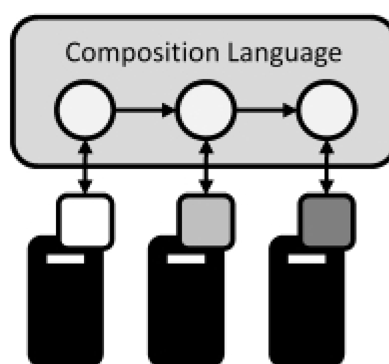

Figura 6.1: Applicazioni modulari

6.3.4 Accoppiamento debole

In un'applicazione realizzata a componenti può essere più semplice occuparsi di determinati aspetti, come la scalabilità, la gestione degli errori, la propagazione degli aggiornamenti o il mantenimento della consistenza, se le dipendenze esistenti tra tali componenti sono sufficientemente ridotte, o persino assenti. In tal modo l'aggiornamento, la sostituzione, la rimozione o il fallimento di un componente hanno un impatto minimo sugli altri.

Il disaccoppiamento, o almeno l'accoppiamento debole, vengono raggiunti riducendo il numero di assunzioni che un componente fa sugli altri quando scambia con essi informazioni; in tal modo si aumenta la robustezza del servizio da essi fornito. La soluzione migliore consiste nella comunicazione persistente asincrona, in cui si fa uso di un *middleware* affidabile che

si occupa di gestirne tutti gli aspetti, dal formato dei messaggi scambiati alla loro consegna ai destinatari corretti. In questo modo un componente non necessita di conoscere l'indirizzo della controparte con la quale vuole comunicare, e non richiede nemmeno che essa sia in esecuzione al momento dell'invio del messaggio. In questo caso viene raggiunto il massimo grado di indipendenza tra i componenti.

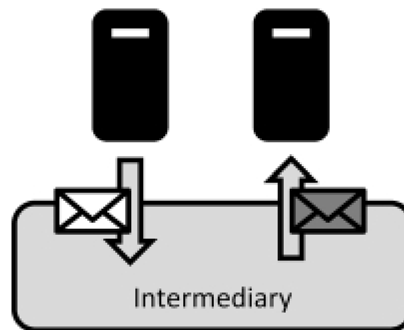

Figura 6.2: Accoppiamento debole

6.3.5 Componenti *stateless*

Quando ci si trova in contesti altamente distribuiti la probabilità di fallimenti aumenta notevolmente. L'applicazione deve quindi tenere in considerazione questo fatto assumendo che i suoi componenti possano fallire in qualunque momento. Inoltre in un contesto Cloud le istanze di un componente potrebbero essere aggiunte o rimosse in qualunque momento, in virtù della proprietà di elasticità.

I componenti dovrebbero essere quindi realizzati in modo da non contenere uno stato interno ma da appoggiarsi completamente a servizi di memorizzazione persistente esterni. In tal modo, se un componente subisce un fallimento, non si verifica alcuna perdita di dati; inoltre vengono notevolmente migliorate le capacità di un'applicazione di scalare poiché più istanze di uno stesso componente possono condividere la sorgente dei dati ed agire quindi come se avessero tutti lo stesso stato interno. Viene infine notevolmente semplificato anche il processo di aggiunta o rimozione delle istanze

di un componente, poiché non è necessario preoccuparsi delle informazioni sul suo stato.

Occorre prestare attenzione però che non si creino dei colli di bottiglia dovuti alla centralizzazione delle informazioni. Potrebbe rendersi in tal caso necessario replicare anche i dati per raggiungere i livelli di scalabilità desiderati.

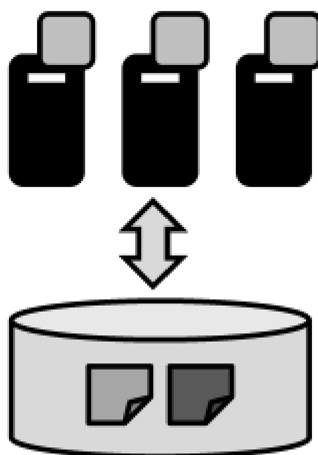

Figura 6.3: Componenti *stateless*

6.3.6 Componenti a singola istanza

Le applicazioni Cloud possono essere ottimizzate attraverso la condivisione di componenti, in particolare di quelli che non richiedono configurazioni particolari e personali tra i vari tenant. Se ciò avviene, le risorse vengono sfruttate in modo migliore, portando ad un abbattimento dei costi necessari all'esecuzione dell'applicazione. Perché ciò sia possibile è necessario sviluppare ed effettuare il deployment dei singoli componenti in maniera tale che siano condivisibili tra le varie istanze di un'applicazione o, eventualmente, anche di applicazioni diverse. In questo modo si semplifica anche lo sviluppo della scalabilità dell'applicazione perché, una volta che tali componenti siano stati progettati scalabili ed elastici, essi rappresentano una porzio-

ne dell'applicazione già in grado di scalare ed è sufficiente concentrarsi sui rimanenti aspetti.

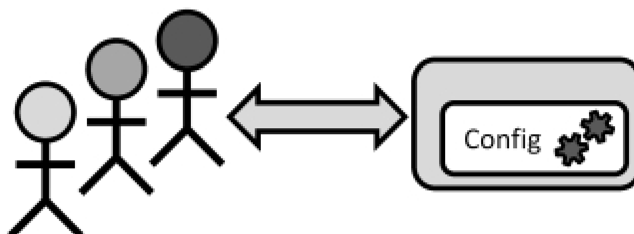

Figura 6.4: Componenti a singola istanza

6.3.7 Componenti a singola istanza configurabili

Questo pattern estende il precedente nel caso in cui i vari tenant o utenti richiedano una configurazione personalizzata dei componenti ed offre una soluzione per non dover creare più istanze di uno stesso componente, una per ogni diversa configurazione.

La soluzione viene raggiunta non innestando le configurazioni all'interno dei componenti ma memorizzandole altrove, ad esempio presso i servizi di storage del sistema. Tali configurazioni vengono poi accedute dalla singola istanza del componente, che determina così il proprio comportamento in funzione del tenant con cui si trova ad interagire. In tal modo, anche di fronte a comportamenti e configurazioni personalizzate differenti, si riesce ad ottenere un'ottimizzazione nell'uso delle risorse dovuta alla condivisione dei componenti tra i vari tenant.

Per completezza è bene sottolineare che questi due pattern non coprono tutti gli scenari di fronte ai quali è possibile trovarsi. Potrebbe infatti essere necessario, ad esempio per ragioni legali, riservare ad ogni tenant la propria istanza di un componente, impedendo la condivisione ottimale di risorse. Tuttavia, quando ciò è indispensabile se ne può trarre comunque un vantaggio cercando di aumentare la configurabilità e la personalizzazione dei componenti che in questo caso possono raggiungere livelli più elevati e con maggior semplicità rispetto alle due situazioni precedenti.

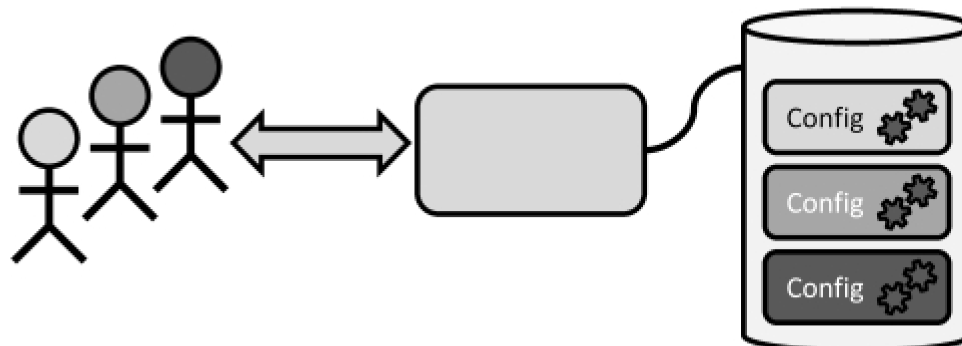

Figura 6.5: Componenti a singola istanza configurabili

6.3.8 Componenti idempotenti

Il problema che si vuole risolvere tramite l'adozione di questo pattern è la gestione dei messaggi quando essi potrebbero eventualmente anche essere replicati. Ciò può avvenire ad esempio nel caso in cui il middleware sottostante non garantisca un protocollo in grado di effettuare consegne *exactly-once*. In generale tuttavia è possibile beneficiare dei vantaggi di questo pattern anche in altre situazioni.

In questo caso la soluzione al problema può essere ottenuta attraverso due strade differenti: da una parte è possibile implementare un filtro in grado di riconoscere e scartare i messaggi ricevuti, dall'altra invece si tende ad agire per modificare la semantica dei messaggi, raggiungendo gli stessi risultati tramite richieste idempotenti.

Nel primo caso, per permettere al filtro di funzionare, è necessario che ogni messaggio sia identificato in maniera univoca. Tale identificatore può quindi essere memorizzato dal filtro a fronte della ricezione di un nuovo messaggio permettendogli così di capire, all'arrivo di successivi messaggi uguali, se essi sono effettivamente delle repliche oppure dei messaggi nuovi ma con contenuto uguale. É necessario in questo caso stabilire quanto mantenere in memoria gli identificatori dei messaggi ricevuti e quando è possibile e sicuro cancellarli.

Nel secondo caso invece occorre modificare le funzionalità offerte dal componente, quindi la semantica dei messaggi ricevuti, facendo in modo

che l'esecuzione delle operazioni abbia gli stessi effetti anche se ripetuta più volte. Ad esempio, volendo modificare una proprietà del sistema, piuttosto che specificarne la variazione in funzione del suo attuale valore, occorre invece specificarne direttamente il nuovo valore.

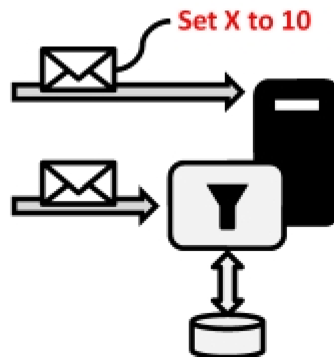

Figura 6.6: Componenti idempotenti

6.3.9 Map Reduce

Avendo già incontrato più volte nel corso di questo lavoro la tecnica chiamata MapReduce, ne forniamo ora una descrizione rappresentandola come pattern architetturale.

Il contesto in cui tale pattern nasce è la presenza di sistemi di memorizzazione persistente che non sono in grado di eseguire interrogazioni sui dati complesse, e ne restituiscono di conseguenza porzioni più grandi che devono essere ulteriormente elaborate e raffinate. Per implementare la scalabilità nell'applicazione in sviluppo occorre fare in modo che anche queste operazioni siano affrontate in modo da garantirne una forma di elasticità, ad esempio attraverso il calcolo parallelo.

La soluzione viene ottenuta mappando (*Map*) i dati in sottoinsiemi di dimensioni più ridotte, distribuendoli attraverso i nodi computazionali, ciascuno dei quali esegue parallelamente l'elaborazione necessaria producendo dei risultati parziali. Successivamente tali risultati devono essere sintetizzati e ridotti (*Reduce*) ad un unico insieme, eseguendo eventualmen-

te delle elaborazioni finali, in modo tale che esso rappresenti il risultato dell'interrogazione complessa iniziale.

Questo pattern viene utilizzato spesso per eseguire elaborazioni complesse sui dati col fine di svolgere analisi di vario tipo, dal controllo delle azioni di un utente per determinarne statisticamente le preferenze, all'analisi di mercato per individuare i prodotti più o meno venduti e popolari.

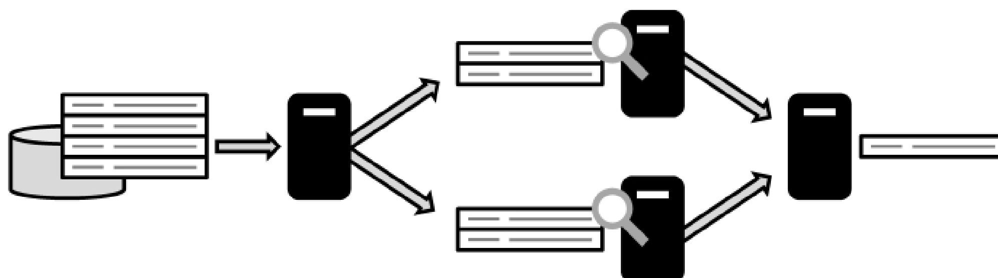

Figura 6.7: MapReduce

6.3.10 Tipi di Cloud e modelli di servizi

Per concludere questa sezione sui pattern architetturali, mettiamo in evidenza come anche i tipi di Cloud già analizzati nel capitolo introduttivo (*Public*, *Private*, *Protected* e *Community Cloud*) ed i modelli di servizi (*IaaS*, *PaaS* e *SaaS*) possono essere rappresentati ed interpretati come dei pattern. Essi infatti sono delle soluzioni generiche a dei problemi ricorrenti che affrontato, ad esempio, i Cloud provider quando devono realizzare un'infrastruttura elastica e fornirne i servizi ai propri tenant, o le aziende quando devono condividere le proprie risorse con altre organizzazioni ottimizzandone anche l'utilizzo e le prestazioni.

6.4 Sviluppo di applicazioni in AWS

In questa sezione viene riportato un insieme di concetti e pratiche, tratti dalla documentazione ufficiale di *Amazon Web Services*, in particolare [10], che permettono di sviluppare applicazioni realmente Cloud. Come potremo

osservare, alcuni di questi concetti si riallacciano perfettamente ai principi esposti nella prima sezione di questo capitolo e ad alcuni dei pattern architetturali descritti nella sezione precedente, confermando che quanto finora abbiamo esposto solo teoricamente trova reale applicazione. Per ogni concetto esposto inoltre verrà anche fornito un insieme di indicazioni su come implementare concretamente tale concetto tramite i servizi e le tecnologie offerte da AWS.

6.4.1 Considerare la possibilità di errori

Gli ingegneri del software di AWS raccomandano di tenere in considerazione la possibilità di fallimenti ed errori sin dalla fase di progetto. In questo modo si può pervenire ad un software in cui il recupero da tali errori sia il più possibile automatizzato e che non si faccia cogliere di sorpresa da eventuali eventi più o meno catastrofici.

In generale occorre rimuovere le ipotesi implicite fatte abitualmente quando si sviluppano applicazioni locali o moderatamente distribuite, sostituendole con ipotesi più adeguate alla piattaforma Cloud. In particolare, tra queste le più importanti sono:

- assumere che l'hardware sottostante possa fallire;
- assumere che periodi di non disponibilità delle risorse (corrente elettrica, nodi computazionali, dispositivi di memorizzazione) possano avvenire, indipendentemente dal fatto che esse siano virtuali o reali;
- assumere che disastri di varia natura possano verificarsi durante il ciclo di vita dell'applicazione;
- assumere che picchi di traffico e sovraccarichi nelle richieste in certi momenti possano verificarsi;
- assumere che perfino il software prodotto e la logica applicativa possano incontrare degli errori che non sono stati previsti.

L'applicazione di quella che chiamano una progettazione pessimista permette di realizzare sistemi più sicuri ed affidabili che supportino nativamente strategie di recupero e risoluzione degli errori. In altre parole le ipotesi sopra elencate permettono di progettare ed implementare sistemi che siano *fault-tolerant*.

É importante in fase di progettazione chiedersi che cosa possa accadere se uno specifico nodo fallisce, come sia possibile individuare e riconoscere tale fallimento (questione tutt'altro che scontata, se si considerano le dimensioni in termini di infrastruttura che può raggiungere un sistema altamente scalabile ed elastico come quelli Cloud) e come sostituire un nodo nel quale si è verificato un errore. Occorre poi tenere in considerazione i vari scenari per i quali è bene essere preparati, ad esempio, se ci si trova in presenza di *load balancer* come reagire nel caso siano proprio questi ultimi a fallire, oppure nel caso di architetture master-slave come comportarsi se fallisce il nodo master. Inoltre è importante considerare anche problematiche software di diversa natura, come ad esempio la possibilità che un servizio esterno dal quale l'applicazione dipende possa cambiare la propria interfaccia, oppure possa tardare eccessivamente a rispondere o generare un'eccezione. Infine, una buona architettura Cloud dovrebbe essere il più possibile indifferente ad eventuali riavvii, sia di parti del proprio sistema che globali, e dovrebbe sempre poter ripristinare il proprio stato precedente al riavvio.

Per raggiungere questi obbiettivi è possibile una serie di strategie, tra le quali:

1. realizzare buoni piani di backup e ripristino dei dati ed automatizzarli il più possibile

2. realizzare componenti in grado di ripristinare il proprio stato al momento del riavvio

3. permettere allo stato del sistema di essere completamente ripristinato, ricaricando le richieste precedenti tramite una coda

4. utilizzare opportune AMI in grado di ottimizzare e supportare i due punti precedenti

5. evitare di mantenere componenti dello stato in memoria volatile, usando il più possibile i servizi di memorizzazione persistente

In particolare, l'implementazione di tali strategie può essere fatta tramite *Amazon Web Services* grazie alle seguenti tecniche:

- Utilizzare *Elastic IP*, che permette di riassegnare dinamicamente un indirizzo IP statico. In particolare esso è utile in caso di fallimenti che avvengono a livello hardware rendendo inutilizzabili alcuni server

oppure in concomitanza di aggiornamenti nel software. Tale servizio permette di reindirizzare il traffico verso nuovi server.

- Utilizzare più *Availability Zone*, aumentando la disponibilità *availability* dei propri servizi. Esistono in particolare alcune funzionalità di cui precedentemente non abbiamo parlato, come *Amazon RDS Multi-AZ* che permettono di replicare automaticamente gli aggiornamenti fatti su un database a tutte le sue repliche situate in *Availability Zone* diverse.

- Mantenere delle specifiche AMI in modo da poter replicare e clonare facilmente e velocemente determinati ambienti, sia di sviluppo e di testing che di esecuzione.

- Utilizzare il servizio *Amazon CloudWatch* per aumentare la visibilità interna del proprio sistema e prendere opportuni provvedimenti in caso di problemi hardware o degradamento delle prestazioni tramite il servizio *auto-scaling*, in modo da poter sostituire automaticamente istanze di *Amazon EC2* nelle quali si riscontrano malfunzionamenti.

- Utilizzare *Amazon EBS* per creare degli *snapshot* delle macchine virtuali e caricare automaticamente tali *snapshot* in *Amazon S3*, in modo che siano indipendenti e slegati dalla singole istanze.

- Utilizzare il servizio *Amazon RDS* che permette di specificare la frequenza con cui effettuare dei backup automatici.

6.4.2 Favorire il disaccoppiamento dei componenti

L'architettura Cloud rinforza i principi del modello SOA secondo i quali più i componenti di un sistema sono debolmente accoppiati più facilmente e grandemente tale sistema è in grado di scalare. Il punto fondamentale risiede nel realizzare componenti che non abbiano forti dipendenze gli uni dagli altri, in modo tale che se uno qualunque di essi dovesse mostrare qualche problema, come un'assenza o un ritardo nella risposta, tutti gli altri restino comunque in grado di procedere con il proprio lavoro, come se non si fosse verificato alcun errore. I componenti devono ridurre al minimo le ipotesi che fanno gli uni sugli altri, limitandosi il più possibile a trattarsi come delle 'scatole nere' (*black boxes*) delle quali non conoscono il funzionamento

e l'implementazione interna e con le quali comunicano in maniera asincrona. Così facendo sia dal punto di vista del codice che da quello funzionale non si verificano accoppiamenti che possono risultare pericolosi per l'intero sistema.

Gli aspetti a cui occorre prestare attenzione si possono riassumere nei seguenti punti:

- individuare quali componenti o funzionalità è possibile isolare dall'applicazione monolitica, facendo in modo che possano essere eseguiti separatamente;
- capire come è possibile aggiungere più istanze di uno stesso componente senza danneggiare il sistema ma anzi facendo in modo che esso sia in grado di servire più utenti;
- incapsulare correttamente i componenti in modo che siano in grado di interagire gli uni con gli altri secondo un protocollo di comunicazione asincrono.

Il disaccoppiamento dei componenti, l'asincronicità e la scalabilità orizzontale sono degli aspetti strettamente legati tra loro, e tutti fondamentali nel contesto delle applicazioni Cloud. Oltre a garantire una migliore capacità di scalare al sistema essi permettono anche di realizzare applicazioni secondo dei modelli ibridi nei quali alcuni componenti vengono eseguiti all'esterno del Cloud mentre altri ne sfruttano le potenzialità, in alcuni casi rivolgendosi ad esso anche solo per ottenere maggiori potenza di calcolo e larghezza di banda. Diventa così possibile, con uno sforzo minimo, realizzare applicazioni che stiano sul confine tra il Cloud e l'ambiente esterno.

I componenti possono essere disaccoppiati tramite l'adozione di code o buffer di messaggi che permettono di gestire molto più facilmente la concorrenza e i picchi di traffico e permettono al sistema di raggiungere una maggiore affidabilità. Grazie ad essi infatti, se un componente si trova momentaneamente non disponibile, le richieste ad esso inviate non vengono perse e potranno essere processate non appena tale componente tornerà in funzione o verrà rimpiazzato. La figura 6.8 è una rappresentazione di come i componenti possono mostrare un accoppiamento forte tramite l'esempio dell'invocazione di metodi classica della programmazione *Object-Oriented*. In figura 6.9 invece viene mostrato come raggiungere un disaccoppiamento, o accoppiamento debole, tramite l'uso di code di messaggi.

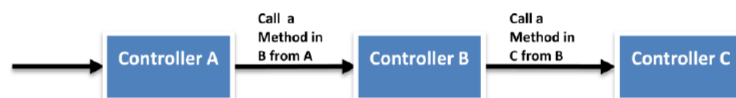

Figura 6.8: Un esempio di come i componenti possano essere fortemente accoppiati a causa del meccanismo dell'invocazione di metodi su un riferimento diretto ad un oggetto.

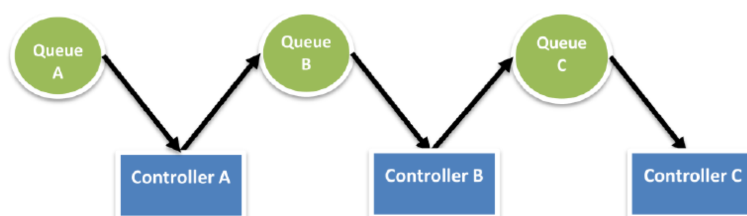

Figura 6.9: I componenti in questo caso vengono invece disaccoppiati poiché ognuno di essi accede, in scrittura o in lettura, ad una determinata coda di messaggi, che garantisce l'asincronicità mantenendo in memoria le richieste.

Il disaccoppiamento dei componenti può essere raggiunto tramite le seguenti tecniche di AWS:

- Utilizzare *Amazon SQS* per isolare i componenti e realizzare un buffer di messaggi.
- Progettare ogni componente in modo che maniera tale che esponga l'interfaccia tipica di un servizio, sia responsabile della propria scalabilità secondo le dimensioni appropriate e interagisca con gli altri componenti secondo una modalità di comunicazione asincrona.
- Incapsulare ogni componente in una propria AMI in modo che possa essere distribuito e gestito separatamente.
- Rendere l'applicazione il più possibile *stateless*, memorizzando le informazioni sullo stato all'esterno del componente, ad esempio tramite il servizio *Amazon SimpleDB*. Questo semplifica e garantisce una migliore scalabilità.

6.4.3 Implementare l'elasticità

L'elasticità può essere implementata sostanzialmente in tre modi:

- scalando periodicamente l'infrastruttura a intervalli fissi;
- scalando l'infrastruttura sulla base di previsioni di picchi di traffico legati a particolari eventi di mercato, come campagne pubblicitarie o lanci di nuovi prodotti;
- scalando l'infrastruttura automaticamente in base alla quantità di richieste; per far ciò è necessario utilizzare un servizio di monitoraggio di determinate metriche.

Qualunque modalità si scelga per l'implementazione dell'elasticità è tuttavia necessario innanzitutto automatizzare ed ottimizzare i processi di deployment e configurazione degli ambienti. In questo modo il sistema può diventare in grado di scalare automaticamente, senza necessità di un intervento umano, raggiungendo subito dei benefici economici, poiché le risorse utilizzate sono strettamente allineate alla domanda e non si corre il rischio di allocarne in quantità eccessiva lasciandone alcune inutilizzate.

Automatizzazione dell'infrastruttura

La creazione di processi di deployment automatizzati è possibile grazie alla disponibilità di API che permettono di interagire e gestire l'infrastruttura. É importante automatizzare tali processi sin dalle prime fasi dello sviluppo piuttosto che aspettare quelle finali, poiché essi forniscono la possibilità di ridurre gli errori, soprattutto quelli dovuti ad un intervento umano, e facilitano e rendono più efficiente il processo di aggiornamento del software. Per automatizzare i processi di deployment è possibile utilizzare le seguenti tecniche:

- creare una libreria contenente gli script più frequentemente utilizzati per l'installazione e configurazione delle macchine virtuali;
- gestire il processo di configurazione e deployment tramite degli agenti appositamente installati sulle macchine virtuali;
- inizializzare le singole istanze di macchine virtuali in funzione del proprio ruolo.

Inizializzazione delle istanze

Ogni istanza dovrebbe avere un ruolo specifico da interpretare all'interno dell'architettura e dell'ambiente dell'applicazione. Questo ruolo può essere passato come argomento durante la fase di avvio dell'istanza e le permette di determinare quali azioni compiere dopo essere stata completamente avviata. Essa infatti, una volta partita, ha necessariamente bisogno di determinate risorse, incluso il codice da eseguire, per poter svolgere il suo compito. L'inizializzazione delle istanze in funzione del loro ruolo porta con sé vari benefici, tra cui la possibilità di ricreare gli ambienti, sia di sviluppo e testing che di esecuzione, con il minimo sforzo, la garanzia di un maggior controllo sulle risorse computazionali, la riduzione degli errori introdotti dall'intervento umano e la possibilità di creare un ambiente in grado di descriversi, conoscersi e guarirsi in caso di errori di varia natura.

Le tecniche ed i servizi che AWS mette a disposizione per automatizzare l'infrastruttura e il processo di deployment e configurazione sono le seguenti:

- Utilizzare il servizio *Amazon Auto-scaling* per definire dei gruppi o dei cluster all'interno di *Amazon EC2* in grado di scalare automaticamente.

- Monitorare le metriche del sistema utilizzando il servizio *Amazon CloudWatch* e prendere i provvedimenti necessari, ad esempio lanciando nuove istanze di AMI tramite *Amazon Auto-scaling* oppure inviando le opportune notifiche.

- Memorizzare ed accedere alle informazioni di configurazione delle macchine virtuali dinamicamente. Ciò può essere fatto utlizzando il servizio *Amazon SimpleDB* per accedere, ad esempio, alle connessioni necessarie durante la fase di avvio delle istanze o per memorizzare informazioni come l'indirizzo IP o il ruolo.

- Realizzare un processo di deployment tale che carichi l'ultima versione dell'applicazione in un *bucket* di *Amazon S3*; accedere successivamente a tale versione durante la fase di inizializzazione del sistema.

- Realizzare strumenti di configurazione, come script automatici o immagini preconfigurate, oppure avvalersi di strumenti *open source* disponibili come Chef[3], Puppet[4], CFEngine[5], o Genome[6].

- Utilizzare versioni minimali di sistemi operativi (*Just enough Operating System*, o *JeOS*[7]; in tal modo le istanze sono più semplici e veloci da avviare, gestire e mantenere.

- Non fare supposizioni sulla disponibilità dell'hardware sul quale un componente sta venendo eseguito. Ad esempio, ciò si traduce nell'assegnazione automatica degli indirizzi IP ai nuovi nodi che si aggiungono alla rete. Essi possono così essere rimpiazzati facilmente riassegnando tali indirizzi ai nodi destinati a sostituirli in caso di guasti.

6.4.4 Sfruttare il parallelismo

É bene fare in modo che il concetto di parallelismo sia introdotto all'interno dell'architettura stessa dell'applicazione, in qualunque tipo di operazione possa supportarlo, dall'accesso ai dati, alla loro memorizzazione o all'elaborazione. Inoltre è importante non solo implementare il parallelismo quando possibile, ma anche automatizzarlo, poiché il Cloud permette di creare istanze di processi molto facilmente.

Per quanto riguarda l'accesso ai dati, sia in scrittura che in lettura, il Cloud è progettato in modo da gestire in maniera ottimale l'esecuzione di operazioni con un grande livello di parallelismo. Per raggiungere le massime prestazioni è perciò fondamentale parallelizzare le richieste, poiché in tal modo esse verranno servite più velocemente di quanto non sarebbe se esse fossero state effettuate sequenzialmente.

Inoltre, per quanto riguarda invece l'elaborazione dei dati, può persino risultare più importante l'adozione del parallelismo di quanto già non lo sia nel caso del loro accesso. Un principio generalmente valido consiste nel processare più richieste in maniera concorrente utilizzando più server, i quali vengono coordinati da un *load balancer*. Nel caso di operazioni *batch* invece

[3]http://wiki.opscode.com/display/chef/Home
[4]http://puppetlabs.com/
[5]http://cfengine.com/
[6]http://genome.et.redhat.com/
[7]Per ulteriori informazioni: http://en.wikipedia.org/wiki/Just_enough_operating_system

è possibile utilizzare un'architettura *master-slave* in grado di eseguire le computazioni in parallelo, ad esempio tramite framework come il già citato *Hadoop*[8].

Per implementare il parallelismo tramite AWS è possibile utilizzare le seguenti tecniche:

- Parallelizzare le richieste ad *Amazon S3* e *Amazon SimpleDB* sfruttando il multi-threading.
- Utilizzare il servizio *Amazon Elastic MapReduce Service* per l'esecuzione di operazioni *batch* parallele.
- Utilizzare il servizio *Elastic Load Balancing* per distribuire le richieste su più server dinamicamente.

6.4.5 Posizionare i dati opportunamente

Dati altamente dinamici

In generale è buona norma mantenere i dati che hanno bisogno di essere elaborati vicino ai componenti che li devono processare per ridurre le latenze che, nell'ambito del Cloud Computing, possono essere assai rilevanti, dovendo spesso muovere i dati attraverso Internet. Ciò diventa ancora più importante se si pensa che il trasferimento dei dati da e verso il Cloud ha dei costi spesso non trascurabili.

Se una grande quantità di dati che devono essere elaborati si trovano all'esterno del Cloud, potrebbe essere più efficiente ed economicamente conveniente trasferirli all'interno del Cloud prima di effettuare le computazioni. Ad esempio, nel caso di applicazioni di *data warehousing* è preferibile spostare tutto l'insieme di dati da elaborare nel Cloud, come anche per applicazioni Web che fanno uso di Database sarebbe più conveniente trasferirvi quest'ultimo.

Viceversa se ci sono dati che vengono generati all'interno del Cloud, è bene che anche le applicazioni che ne fanno uso siano spostate nel Cloud, in modo da usufruire dei vantaggi come una minore latenza ed un minor costo di banda, che solitamente caratterizzano il traffico che resta interno. Ad esempio, nel caso di applicazioni Web che generano delle grandi quantità di

[8] http://hadoop.apache.org/

dati che devono poi essere analizzati ed elaborati da altre applicazioni, il sistema otterrebbe dei vantaggi se queste ultime venissero spostate all'interno del Cloud.

Dati principalmente statici

Nel caso di dati statici che non subiscono modifiche frequenti è invece consigliabile usufruire dei servizi di un componente che sia in grado di gestire la consegna e la distribuzione dei dati in modo che essi vengano mantenuti in *cache* presso i confini del Cloud, nei punti più vicini all'utente finale, minimizzando in tal modo le latenze dovute all'accesso.

Le tecniche che AWS mette a disposizione per gestire il posizionamento corretto dei dati sono le seguenti:

- Utilizzare il servizio *Amazon Import/Export* che permette di muovere grandi quantità di dati utilizzando dispositivi di memorizzazione di massa spediti fisicamente, ad esempio tramite posta, in un centro Amazon. Può sembrare paradossale, ma nel caso ci sia bisogno di spostare quantità veramente imponenti di dati, ad esempio diverse decine di TeraByte, i costi di banda diventerebbero proibitivi e questa soluzione può essere molto più economica rimanendo competitiva in termini di velocità.

- Utilizzare la stessa *Availability Zone* per istanziare macchine che devono comunicare frequentemente.

- Nel caso si utilizzi il servizio *Amazon S3*, creare una distribuzione dei suoi *bucket*, avvalendosi del servizio *Amazon CloudFront*, per poi gestirli tramite *cache* attraverso le 14 locazioni di confine del Cloud sparse nel mondo.

6.5 Sviluppo di applicazioni con Google App Engine

6.5.1 Differenze nella documentazione

Google App Engine, a differenza di Amazon Web Services, non dispone di una documentazione completa e dettagliata per quanto riguarda le pra-

tiche migliori per lo sviluppo di applicazioni Cloud. Mentre infatti AWS fornisce molti documenti che illustrano pattern e principi architetturali che dovrebbero essere adottati per arricchire le applicazioni di determinate caratteristiche, Google App Engine fornisce invece principalmente un insieme di guide e *tutorial* che permettono ai suoi utenti di imparare velocemente a sviluppare applicazioni.

Questo fatto potrebbe sembrare una limitazione o una mancanza di Google App Engine rispetto ad Amazon Web Services, ma in realtà tale mancanza è solo apparente. AWS infatti, fornendo servizi di tipo *IaaS*, è, in un certo senso, costretto a fornire una documentazione più ricca per quanto riguarda lo sviluppo di applicazioni, poiché, come abbiamo visto nel capitolo precedente, tale livello non è orientato a questo aspetto e rende quindi questo compito molto più difficile. Google App Engine invece, collocandosi a livello del *PaaS*, rende lo sviluppo di applicazioni molto più semplice ed accessibile, rivolgendosi quindi non solo ad ingegneri ed architetti del software, ma anche a programmatori e tecnici informatici. Per questo motivo infatti tipicamente i servizi di AWS vengono utilizzati da imprese medie o grandi, che possono disporre di un personale informatico dotato di un certo livello di competenza, mentre i servizi di Google App Engine possono essere utilizzati anche da piccole aziende ed organizzazioni.

6.5.2 Sviluppo di applicazioni Cloud

Quanto detto nella sottosezione precedente riguardo alla documentazione fornita da Google App Engine non deve indurre a pensare che qualunque applicazione sviluppata tramite di esso sia automaticamente un'applicazione Cloud. Il fatto che sia possibile programmare in maniera abbastanza semplice non comporta infatti che un approccio elementare allo sviluppo di applicazioni sia equivalente ad uno ingegneristico.

Vedremo di seguito come alcune indicazioni generali vengano fornite comunque anche dalla documentazione di Google App Engine, nonostante essa si rivolga ad un pubblico con competenze minori rispetto a quello di AWS, e, soprattutto,

Indicazioni generali

La documentazione di Google App Engine, nonostante si rivolga ad un pubblico con competenze minori rispetto a quello di AWS, riporta comunque alcune indicazioni che è sempre bene considerare quando si realizza un'applicazione in ambiente Cloud. Queste sono state dedotte in particolare leggendo ed analizzando i *tutorial*[9] che vengono forniti per lo sviluppo di applicazioni tramite la tecnologia *JSP* e trovano riscontro anche con quanto già esposto nelle sezioni precedenti. Tali indicazioni infatti consistono in una serie di consigli che vengono rivolti allo sviluppatore e si possono così riassumere:

- sviluppare applicazioni cercando di suddividerle in componenti, raggruppando le funzionalità comuni e separando quelle appartenenti a contesti diversi;
- sviluppare i singoli componenti in un'ottica orientata ai servizi.

Realizzare applicazioni tenendo in considerazione tali indicazioni porta alla produzione di un software dotato di un buon livello di qualità, in grado di sfruttare le caratteristiche di scalabilità automatica che vengono fornite dalla piattaforma di Google App Engine.

Applicazione di pattern

Per raggiungere un livello di qualità superiore e trarre una maggiore efficacia dai meccanismi e dalle politiche di scalabilità è sufficiente in realtà applicare i principi ed i pattern architetturali discussi nelle sezioni precedenti di questo capitolo. Mentre con Amazon Web Services infatti la conoscenza di tali pattern poteva da sola non essere sufficiente ed occorreva quindi anche uno studio ulteriore per capire come applicarli concretamente e correttamente, in Google App Engine, proprio per il fatto che esso offre servizi di tipo *PaaS* e permette agli sviluppatori di occuparsi solamente del software, non esistono barriere ulteriori tra la conoscenza ed una buona pratica nell'applicazione dei pattern e la loro effettiva applicazione.

Concludendo quindi, possiamo osservare come lo sviluppo di applicazioni Cloud, anche dal punto di vista dell'utilizzo di pattern e principi architetturali, sia molto più semplice tramite Google App Engine che non con Amazon

[9] https://developers.google.com/appengine/docs/java/gettingstarted

Web Services. Questo, coerentemente anche con quanto detto nel capitolo precedente, è legato al fatto che i servizi di Google App Engine, essendo di tipo *PaaS*, sono strettamente orientati proprio allo sviluppo di applicazioni. Per quanto riguarda AWS invece, volendo utilizzare i suoi servizi di *IaaS* sempre per sviluppare applicazioni, si riscontra la presenza di un *gap* che deve essere colmato da parte del progettista, aumentando la quantità e la complessità del suo lavoro.

6.6 Conclusioni

Come abbiamo potuto capire da questo capitolo, una questione fondamentale degli ambienti Cloud è la realizzazione di applicazioni Cloud, intese, come definito all'inizio, nel senso di applicazioni in grado di sfruttare appieno le caratteristiche di tali ambienti. Purtroppo spesso non vengono forniti strumenti o supporti tecnologici in grado di aiutare lo sviluppatore a scrivere applicazioni di questo tipo: tutto si basa sulle sue capacità e sull'uso che egli fa dei principi e dei pattern architetturali esposti in questo capitolo, la cui applicazione talvolta può risultare anche molto complessa. Se infatti il modello che discende da essi è molto diverso da quello indotto dalla tecnologia sottostante, mappare il primo sul secondo può risultare molto complicato e dispendioso, in termini di energie e di tempo da parte di progettisti e sviluppatori.

In particolare, il modello che viene indotto dalla tecnologia è quasi sempre quello a oggetti, poiché nella maggior parte dei casi i linguaggi adottati sono proprio quelli *Object-Oriented*, al giorno d'oggi decisamente i più diffusi. Tuttavia essi presentano però dei grossi limiti che, se non sono percepiti nel contesto di applicazioni desktop o comunque lievemente distribuite, per applicazioni Cloud diventano molto rilevanti. Il paradigma *Object-Oriented* infatti, nella versione che oggi viene implementata comunemente dai linguaggi e che in realtà è leggermente diversa da come era stata originariamente pensata, non modella in alcun modo il flusso di controllo associato agli oggetti e realizza la comunicazione tra di essi tramite invocazione di metodi per riferimento diretto. Questo rappresenta una grande limitazione nel contesto di ambienti altamente distribuiti poiché introduce due livelli di accoppiamento: quello tra la comunicazione ed il flusso di controllo e quello tra qualunque coppia di oggetti che necessitino di comunicare, uno dei quali

deve necessariamente avere un riferimento diretto all'altro. Di conseguenza, i linguaggi ad oggetti ostacolano lo sfruttamento del parallelismo e dell'asincronicità, poiché richiedono che vengano impiegate specifiche soluzioni di livello piuttosto basso sia per slegare il flusso di controllo dalla comunicazione, tipicamente implementando quest'ultima con dei meccanismi diversi da quelli nativi dei linguaggi (ad esempio tramite *socket*, piuttosto che tramite invocazione di metodi), sia per poter usufruire di più flussi di controllo all'interno di una stessa applicazione (ad esempio usando specifiche librerie per il multi-threading). Inoltre rendono anche assai complicato gestire la distribuzione degli oggetti poiché non possono esistere riferimenti diretti che attraversino i confini della singola macchina, quindi le applicazioni restano comunque sempre coscienti, ad un certo livello, della localizzazione e della mobilità dei propri componenti.

Si rende ora quindi necessario abbandonare il paradigma *Object-Oriented* per passare a qualcosa che si trovi ad uno stato più avanzato nell'evoluzione: questo è il tema fondamentale che verrà affrontato nel prossimo capitolo.

Capitolo 7

Nuovi modelli e paradigmi - Orleans

Come già detto nel capitolo precedente, il paradigma *Object-Oriented* non è pienamente in grado di dominare la complessità degli ambienti altamente distribuiti come quelli Cloud. Per poterlo utilizzare è quindi necessario dotarsi di librerie ed estensioni dei linguaggi che permettano di risolvere molti dei problemi che tipicamente sorgono in tali ambienti, legati ad aspetti come ad esempio il multi-threading, la concorrenza, la consistenza degli stati, la sincronizzazione e la coordinazione. Così facendo però spesso si perviene ad un software molto complesso, che finisce col porre un accento maggiore sui problemi di cui sopra piuttosto che sul contesto applicativo.

Una soluzione a queste problematiche può essere ottenuta adottando dei diversi paradigmi, i quali si adattano molto meglio agli ambienti distribuiti e concorrenti. Uno di questi paradigmi è quello ad attori, che verrà introdotto nella prima sezione di questo capitolo, basandosi sulla descrizione fornita in [6]. Successivamente prenderemo in esame Orleans, un framework tuttora in fase di sviluppo da parte di Microsoft che, adottando un modello molto simile a quello ad attori, permette di scrivere applicazioni affidabili, scalabili ed elastiche. Esso si basa sulla tecnologia .NET ed è reso disponibile ai programmatori tramite delle librerie per i linguaggi supportati da tale tecnologia, come *C#* o *F#*. Le informazioni riguardanti Orleans presenti in questo capitolo sono state tratte da [1].

7.1 Il modello ad attori

7.1.1 Introduzione

Il modello ad attori è un modello di programmazione concorrente per lo sviluppo di sistemi paralleli, distribuiti e mobili. L'entità fondamentale di questo modello è l'attore, un oggetto autonomo in grado di operare concorrentemente ed in maniera asincrona rispetto agli altri attori, con i quali comunica solo tramite scambio di messaggi, senza condividerne lo stato. Un sistema realizzato tramite questo modello è una collezione di attori, alcuni dei quali scambiano messaggi con attori o altre entità esterne al sistema.

Nel modello Object-Oriented gli oggetti incapsulano informazioni e comportamento, separando così la propria interfaccia dalla propria implementazione e permettendo una modularizzazione delle applicazioni. Il modello ad attori estende queste caratteristiche alla programmazione concorrente, incapsulando anche il flusso di controllo.

7.1.2 Gli attori

Ogni attore ha un nome in grado di identificarlo univocamente all'interno del sistema, ed un comportamento che ne determina le azioni svolte. Essi possono inviare e ricevere messaggi in maniera asincrona. Per poter inviare un messaggio ad un altro attore, il mittente deve conoscerne il nome, mentre per quanto riguarda la ricezione dei messaggi, essa avviene tramite una *mailbox*, ovvero una coda di messaggi. Quando un attore prende in carico un messaggio, esegue le operazioni ad esso associate e definite dal suo comportamento, espresso tramite dei metodi, analoghi a quelli del paradigma ad oggetti. Il ciclo di vita di un attore prevede che esso, quando ha terminato di eseguire le operazioni associate ad un messaggio e si trova in stato di *idle*, se è presente un nuovo messaggio nella propria *mailbox*, lo accetti ed esegua la relativa computazione. Se invece, trovandosi in stato di *idle*, non è presente alcun messaggio nella *mailbox*, allora l'attore si blocca, entrando in stato di *waiting* fino all'arrivo di un nuovo messaggio. É importante notare che l'arrivo di un nuovo messaggio non può mai interrompere l'elaborazione che un attore sta svolgendo: essa è sempre portata a termine prima di accettare il nuovo messaggio.

Gli attori non possono comunicare in altro modo se non con lo scambio di messaggi; in particolare, ciò significa che non c'è condivisione di stato tra gli attori, quindi le computazioni parallele non presentano problemi di concorrenza, non potendo interferire tra loro in alcun modo.

Le classi di operazioni che un attore può compiere durante la computazione sono solo tre:

1. invio di un messaggio;
2. modifica del proprio stato;
3. creazione di un nuovo attore.

Il modello ad attori introduce il non determinismo: i messaggi infatti vengono scambiati lungo la rete, la quale ha dei ritardi che non sono noti a priori, e possono seguire percorsi diversi; inoltre il modello adotta il parallelismo e la concorrenza. Questi fattori portano alla non conoscenza dell'ordine di arrivo dei messaggi. Un errore comune quando si programma con il modello ad attori infatti consiste nel supporre che messaggi inviati sequenzialmente da uno stesso attore con un determinato ordine, mantengano tale ordine anche nella ricezione. Questo non è mai vero, nemmeno se tutti i messaggi vengono inviati allo stesso destinatario: nulla infatti impedisce ad un qualunque messaggio di arrivare prima di un altro che era stato spedito precedentemente; ricordiamo infatti che l'invio di messaggi è eseguito in modalità asincrona, ovvero non bloccante, e non si attende nessun tipo di conferma né dall'ambiente di esecuzione né dal destinatario.

7.1.3 Programmi ad attori

All'inizio dell'esecuzione di un programma ad attori, tutte le *mailbox* degli attori presenti sono vuote e nessuno degli attori sta eseguendo della computazione. Perché l'esecuzione abbia effettivamente inizio è necessario che almeno un attore riceva un messaggio da parte dell'ambiente esterno: questo rappresenta l'*entry point* del programma.

La terminazione di un programma ad attori avviene invece quando tutti gli attori creati sono in stato di *idle* e l'ambiente non è più abilitato ad inviare nuovi messaggi al sistema. Tuttavia è anche possibile che un programma ad attori non termini mai, come può succedere per molte applicazioni interattive, ad esempio sistemi operativi o server di vario genere.

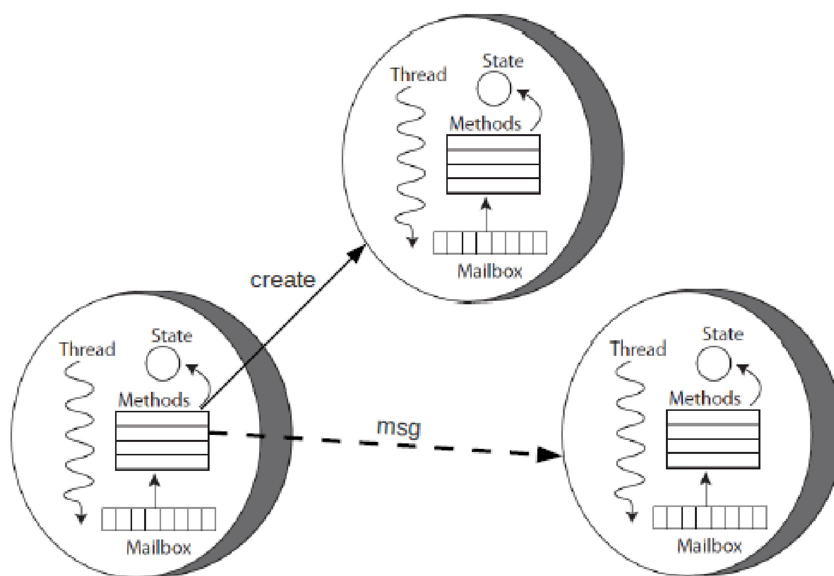

Figura 7.1: Rappresentazione degli attori, che possono essere considerati come degli oggetti potenziati aggiungendo loro un proprio flusso di controllo ed una *mailbox*. Vengono qui messe in risalto le tre operazioni caratteristiche: modifica dello stato, invio di messaggi e creazione di un attore.

7.1.4 Sincronizzazione

A causa della semantica non bloccante dell'operazione di invio dei messaggi, è necessario introdurre dei meccanismi che permettano di gestire la sincronizzazione. Ciò è fatto sempre tramite la comunicazione.

Remote Procedure Call

Adottando uno stile di comunicazione simile a quello del Remote Procedure Call, si introduce un'operazione di invio di messaggi bloccante. Nello scenario tipico dell'RPC infatti il chiamante invia una richiesta e rimane in attesa di una risposta dal destinatario, senza eseguire altre operazioni. Mentre non è corretto rendere tale semantica l'unica disponibile poiché sarebbe in forte contrasto con il parallelismo degli ambienti distribuiti, impedendo

a quest'ultimo di essere sfruttato, talvolta si rende però necessario come meccanismo di sincronizzazione in base allo specifico contesto applicativo.

I due casi d'uso più tipici nei quali è utile utilizzare la semantica bloccante dell'RPC sono i seguenti:

- quando un attore vuole inviare una sequenza di messaggi ordinata ad un certo destinatario, oppure quando vuole essere certo che il suo primo destinatario abbia ricevuto un messaggio prima di inviarne altri a destinatari diversi;
- quando lo stato del mittente varia in funzione della risposta ottenuta, rendendogli impossibile elaborare altre richieste significativamente finché non ottiene il risultato richiesto.

Occorre fare attenzione quando si adotta questo tipo di comunicazione, poiché un suo abuso, oltre ad influire negativamente sulle prestazioni del programma, può generare anche pericolose dipendenze tra gli attori che potrebbero rimanere bloccati, ad esempio a causa di *deadlock*.

Vincoli di sincronizzazione locali

A causa del non determinismo di cui abbiamo già parlato precedentemente, i mittenti, quando inviano i loro messaggi potrebbero non conoscere lo stato dei destinatari. In particolare, potrebbero non sapere se il destinatario si trova in uno stato che gli permette di elaborare il messaggio, oppure se tale elaborazione al momento non è per lui possibile. Un primo modo per risolvere questo problema consiste nel cosiddetto *busy waiting*: poiché il destinatario non è in grado di processare la richiesta, egli la rifiuta; il mittente quindi è costretto a continuare a inviare richieste analoghe finché non vengono accettate, secondo il meccanismo chiamato *polling*. Questa soluzione però presenta due grossi svantaggi: il mittente non può proseguire con l'esecuzione del proprio programma poiché si trova impegnato nell'inoltro continuo di richieste sempre uguali, quando invece potrebbe sfruttare il tempo in cui resta in attesa del risultato per eseguire ulteriore lavoro; inoltre viene generata una notevole quantità di traffico, che rischia di saturare ed intasare la rete del sistema.

Per ovviare a questi problemi si introduce negli attori un *buffer*, ovvero una struttura che permetta loro di memorizzare i messaggi ricevuti ed eventualmente rimandarne l'esecuzione, se ad esempio attualmente non

sono in grado di portarla a termine. Si contrasta in questo modo il non determinismo che impedisce di conoscere l'ordine di arrivo dei messaggi con l'introduzione di vincoli che ne specificano invece un ordinamento parziale nell'elaborazione.

Un'ultima questione riguarda come è possibile specificare l'ordinamento di esecuzione dei messaggi. Se ciò è fatto all'interno del codice dell'attore infatti si verifica una violazione del principio di separazione degli aspetti (*separation of concerns*) poiché vengono mischiate la logica delle funzionalità dell'attore con quella di ordinamento dell'esecuzione dei messaggi. Una soluzione migliore che talvolta viene offerta è quella di avvalersi di predicati che permettano di governare l'elaborazione di tali messaggi tramite espressioni logiche riguardanti lo stato dell'attore ed il tipo di messaggio ricevuto.

7.1.5 Pattern di programmazione parallela

Avendo già precedentemente introdotto e spiegato alcuni pattern architetturali, è opportuno ora far notare che ne esistono alcuni che presentano una forte affinità con il modello ad attori, in particolare il pattern *pipeline* ed il *divide et impera*.

Pipeline

Questo pattern aiuta a modellare le situazioni in cui è necessario eseguire su dei dati un'elaborazione che viene suddivisa in più fasi, o *stage*. Ogni fase può essere gestita come un componente a sé stante, permettendone in tal modo anche il riuso in elaborazioni diverse da quella corrente. Una volta eseguita questa suddivisione è quindi possibile costruire il processo di elaborazione necessario, chiamato in questo contesto appunto *pipeline*, semplicemente combinando i componenti corretti in un particolare ordine. Un'applicazione classica di questo pattern, mostrata anche in figura 7.2, può essere trovata nell'elaborazione delle immagini.

Questo pattern è particolarmente legato al modello ad attori perché permette di sfruttarne il parallelismo in maniera assai semplice ed efficace. Nelle situazioni infatti in cui sia presente non una sola struttura di dati da elaborare, ma un intero insieme, come potrebbe essere nel caso di uno stream di immagini, realizzando ogni *stage* tramite un attore è possibile

eseguire contemporaneamente fasi diverse dell'elaborazione su diversi dati, aumentando notevolmente il *throughput* del sistema.

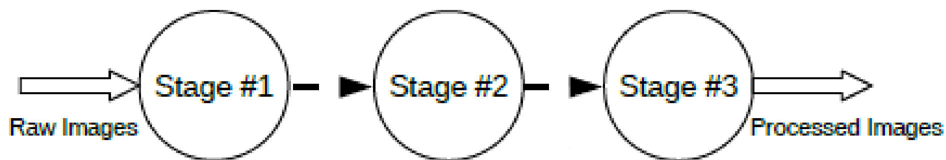

Figura 7.2: Esempio del pattern *pipeline* applicato all'elaborazione delle immagini.

Divide et impera

Un altro pattern molto comune e generale è il *Divide et impera*. Esso consiste in un approccio *top-down* ai problemi complessi i quali, secondo questo pattern, devono essere divisi in sottoproblemi più semplici da risolvere, eventualmente anche in maniera ricorsiva. Una volta raggiunto un livello di complessità sufficientemente ridotto, è possibile quindi risolvere i vari sottoproblemi generati, ricombinando infine le soluzioni parziali in modo da ricostruire la soluzione finale.

A causa della sua grande generalità ed astrazione, talvolta il *Divide et impera* non viene nemmeno considerato un pattern architetturale, quanto piuttosto, come già detto sopra, un approccio ai problemi, o un principio generale valido in molti altri ambiti oltre a quello informatico. Ad esempio, lo stesso pattern *pipeline* può essere considerato un'applicazione di questo principio, poiché suddivide il problema iniziale lungo la dimensione dell'elaborazione. Applicando invece la suddivisione lungo la dimensione dei dati, i quali vengono quindi partizionati, si perviene ad un altro pattern architetturale già studiato in precedenza, come viene mostrato nella figura 7.3: il Map Reduce. Anche esso, come si può vedere sempre da tale figura, trova una sua naturale realizzazione nel paradigma che stiamo studiando, poiché permette di modellare i singoli nodi computazionali, sia i *master* che i *worker*, proprio come attori.

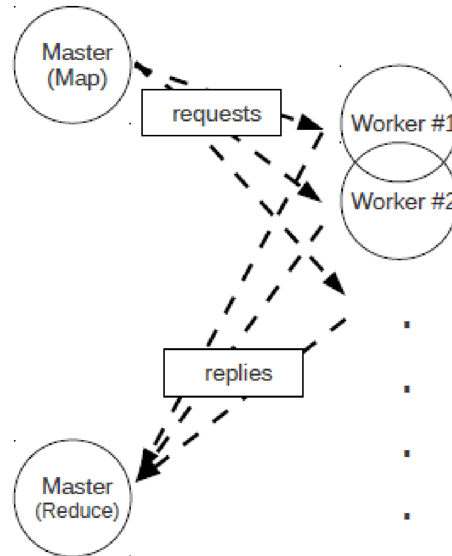

Figura 7.3: Esempio di come il *divide et impera* trovi applicazione nel pattern Map Reduce, in una sua implementazione ad attori.

7.1.6 Proprietà del modello ad attori

Incapsulamento ed atomicità

Fairness

Trasparenza alla locazione

7.2 Orleans

Come già detto all'inizio del capitolo, Orleans è un framework, che Microsoft sta tuttora sviluppando, il quale, grazie all'adozione di un modello di programmazione simile a quello ad attori studiato nella sezione precedente e che ora analizzeremo, induce nelle applicazioni un'architettura in grado renderle adeguate all'ambiente Cloud. Gli obbiettivi principali che si pone tale framework sono due:

- permettere, anche agli sviluppatori che non hanno familiarità con i sistemi distribuiti, di realizzare applicazioni su larga scala;

- assicurarsi che tali applicazioni siano in grado di gestire scalature di vari ordini di grandezza senza che sia necessario riprogettarle, cambiandone le architetture.

Per raggiungere questi obbiettivi il modello di programmazione di Orleans è stato vincolato in modo da condurre gli sviluppatori verso la realizzazione di applicazioni in grado di scalare facilmente. Quando possibile, ciò è stato fatto tramite l'introduzione di meccanismi dichiarativi, che permettano di esprimere il comportamento desiderato di un determinato componente, lasciando al sistema di esecuzione di Orleans la responsabilità di come ottenere tali comportamenti.

Il sistema di programmazione di Orleans si basa su componenti asincroni, isolati e distribuiti, chiamati *grain*, i quali hanno molte caratteristiche in comune con gli attori. In aggiunta essi però possono essere automaticamente replicati per ottenere la scalabilità, il loro stato può essere reso persistente su un sistema di storage condiviso, ed esistono meccanismi in grado di riunificare le modifiche dello stato quando repliche diverse di uno stesso *grain* apportano ciascuna i propri aggiornamenti.

In questa sezione verranno utilizzati esempi di codice tratti dalla documentazione ufficiale di Orleans fornita da Microsoft, i quali riguardano l'implementazione di un *social network* molto semplice i cui utenti, modellati tramite la classe *Chirper* possono specificare di seguire (*follow*) altri utenti ricevendone i messaggi da questi pubblicati, similmente a quanto avviene per il noto social network *Twitter*.

7.2.1 *Grain*

I *grain* possono essere visti come l'unità base di programmazione: tutto il codice scritto per il framework Orleans viene infatti eseguito all'interno di un *grain*. Il sistema mantiene in esecuzione concorrentemente più *grain*; essi non condividono memoria o porzioni di stato, sono internamente single-threaded e portano a compimento l'esecuzione di ogni richiesta, prima di passare alla successiva, come succede anche per gli attori nel relativo modello.

Le dimensioni dei *grain* non hanno limiti, tuttavia sta al progettista trovare una buona granularità per il proprio sistema, data dal compromesso tra la concorrenza e la quantità di stato necessarie per garantire una computazione efficiente. *Grain* troppo grandi non permettono di sfruttare appieno

il parallelismo dell'ambiente costringendo computazioni che potrebbero essere eseguite concorrentemente ad essere invece eseguite sequenzialmente. Al contrario, *grain* troppo piccoli e numerosi porteranno ad un sovraccarico nelle comunicazioni necessarie per eseguire le richieste e per mantenere coerente lo stato interno del sistema, il quale risulterà eccessivamente frammentato.

I *grain* comunicano tra loro tramite lo scambio di messaggi asincrono, che viene implementato tramite l'invocazione di metodi e che, come per il modello ad attori, presenta una semantica non bloccante. L'eventuale valore di ritorno di tali metodi viene gestito da Orleans tramite il concetto di *promise*, o promessa, che rappresenta un risultato il cui arrivo è atteso nel futuro. Le *promise* sono sono il meccanismo che permettono di coordinare la computazione concorrente del sistema. Ad esse può essere assegnata una porzione di codice, detta *handler*, la quale viene eseguita all'arrivo del risultato mentre l'esecuzione del *grain* continua. Se invece è necessario si può decidere di adottare esplicitamente un comportamento bloccante, scegliendo di rimanere in attesa dell'arrivo di tale risultato.

L'impredicibilità è una caratteristica intrinseca dei programmi realizzati tramite il framework Orleans, come d'altra parte lo è per ogni sistema distribuito sufficientemente complesso. Essa si esplicita nell'impossibilità di determinare l'ordine di esecuzione degli *handler*, poiché non è possibile conoscere anticipatamente con certezza il tempo necessario a trasmettere il messaggio attraverso la rete né quello richiesto per l'esecuzione di ogni metodo invocato dal *grain* chiamante. Tuttavia, grazie al meccanismo delle *promise*, l'impredicibilità resta limitata a questo aspetto e risulta quindi facile da gestire e comprendere, riducendo la possibilità che essa divenga una fonte di errori.

7.2.2 *Activation*

Come già detto all'inizio del capitolo, un *grain* può essere replicato automaticamente da Orleans per gestire aumenti nel carico di lavoro. Le singole istanze di un *grain* che sono in esecuzione nel sistema sono dette *activation*, o attivazioni. Esse sono in grado di elaborare richieste, indipendenti tra loro, che sono state rivolte ad uno stesso *grain*, eventualmente venendo anche eseguite su server diversi. Il processo di richiedere ulteriori macchine all'infrastruttura sottostante viene quindi gestito automaticamente da Orleans, e

tale infrastruttura viene completamente nascosta allo sviluppatore, che non deve in alcun modo farsene carico.

Mentre i *grain* quindi sono l'astrazione logica a livello di programmazione, le attivazioni rappresentano l'unità di esecuzione a runtime. Ogni attivazione di uno stesso *grain* viene eseguita indipendentemente ed in maniera isolata dalle altre ed esse non possono condividere memoria tra loro o invocarsi metodi reciprocamente. Esse sono quasi completamente trasparenti le une alle altre; la loro unica interazione, nascosta comunque allo sviluppatore, consiste nella fase di riconciliazione delle modifiche allo stato persistente del *grain*.

7.2.3 *Promise*

Il ciclo di vita di una promessa è abbastanza semplice e comprende pochi stati. Inizialmente essa è detta *unresolved* e rappresenta, finché si trova in questo stato, l'attesa di ricevere un risultato in futuro. Quando tale risultato arriva, la promessa diviene *fulfilled* ed il suo valore diventa il risultato stesso. Se durante l'esecuzione della richiesta si è verificato invece qualche errore, la promessa si dice diventata *broken* e non ha valore. Sia che si trovi in stato di *fulfilled* che in stato di *broken* la promessa si dice *resolved*.

Le promesse vengono implementate tramite due classi .NET:

- *AsyncCompletion* che rappresenta il futuro completamento di un'operazione

- *AsyncValue⟨T⟩* che rappresenta invece il futuro valore ottenuto come risultato di un'operazione

Gli *handler* associati alla risoluzione di una promessa sono invece implementati tramite delagati, passati come argomento al metodo *ContinueWith()* della promessa stessa. Tale metodo a sua volta restituisce una promessa.

Se una determinata *promise* diventa *broken*, allora il delegato non viene eseguito ed anche la promessa restituita dal suo metodo *ContinueWith()* diviene *broken*, a meno che non sia stato predisposto un delegato appositamente associato al fallimento della *promise* iniziale. Questo meccanismo permette di realizzare la propagazione degli errori, come permette di fare anche il meccanismo delle eccezioni, solitamente implementato dai linguaggi *Object Oriented* moderni.

Le promesse mettono a disposizione anche i metodi *Wait()*, che sospende l'esecuzione di un *grain* fino all'arrivo del risultato, e *GetValue()*, che fornisce il valore di tale risultato, bloccando anch'esso l'esecuzione.

L'esecuzione dei delegati all'interno delle *activation* è sempre single-threaded, quindi non può mai esserci più di un delegato in esecuzione all'interno di ogni singola attivazione.

La figura 7.4 mostra un esempio di quanto detto finora, in particolare tramite il codice riportato viene creata una promessa invocando un metodo su *grainA* e ad essa viene associato un delegato invocando il metodo *ContinueWith()* sulla promessa stessa. Il delegato restituisce a sua volta una *promise*, sulla quale il *grain* corrente resta in attesa.

```
(1) AsyncCompletion p1 = grainA.MethodA();
(2) AsyncCompletion p2 = p1.ContinueWith(() =>
(3) {
(4)     return grainB.MethodB();
(5) });
(6) p2.Wait();
```

Figura 7.4: Creazione di una promessa, associazione di un delegato il quale a sua volta restituisce un'altra promessa, e comportamento bloccante specificato tramite l'invocazione del metodo *Wait()*

7.2.4 Esecuzione di *Activation*

Quando un'attivazione riceve una richiesta, la serve in unità di lavoro discrete, chiamate turni. Tutta l'esecuzione del codice di un *grain*, sia che si tratti di gestire un messaggio proveniente da un altro *grain* o da un client esterno, sia che si tratti dell'esecuzione di un delegato, avviene in un turno. Un turno arriva sempre alla sua conclusione senza essere interrotto da altri turni della stessa attivazione.

Nel complesso i sistemi Orleans possono eseguire più turni di attivazioni diverse in parallelo, ma ogni attivazione esegue i propri turni sequenzialmente; questo è il meccanismo che permette di rendere l'esecuzione di un *activation* single-threaded. A livello di scheduling tuttavia non c'è un thread

dedicato per ogni attivazione, ma lo scheduler di Orleans si occupa di allocare di volta in volta i thread disponibili a turni di attivazioni diverse. I thread vengono gestiti internamente come un *pool*.

Grazie al modello di esecuzione single-threaded, non c'è bisogno di adottare strumenti per la sincronizzazione, come lock, semafori o monitor, per garantire l'assenza di corse critiche, tuttavia esso limita il parallelismo solo ad insiemi di *grain* ed attivazioni, impedendone l'utilizzo invece all'interno di uno stesso *grain*, come potrebbe essere necessario ad esempio per eseguire computazioni più veloci partizionando determinati dati e parallelizzandone il lavoro. Questa scelta è stata fatta per semplificare lo sviluppo di applicazioni e ridurre notevolmente la probabilità di commettere errori, i quali solitamente si presentano numerosi quando gli sviluppatori si trovano a gestire questioni di sincronizzazione e concorrenza, molto più di quando si affrontanto problematiche di altro tipo.

Come già precedentemente affermato, esiste un certo grado di non determinismo nelle applicazioni sviluppate tramite Orleans, dovuto all'impredicibilità nella risoluzione delle promesse. Esso non genera comunque corse critiche, tuttavia richiede una certa attenzione poiché occorre tener presente che lo stato di un'attivazione quando un delegato viene eseguito potrebbe essere diverso dallo stato che sussisteva quando il delegato è stato creato.

Generalmente è necessario che un'attivazione finisca completamente di servire una richiesta prima di accettarne un'altra. In particolare, questo significa che non verranno accettate nuove richieste finché ci sono ancora promesse in stato *unresolved*, con relativi delegati non ancora eseguiti. Tuttavia è possibile intervenire per modificare questa politica in due maniere:

- marcando la classe del *grain* con l'attributo *Reentrant*, che permette ai turni di interferire liberamente;

- marcando i singoli metodi con l'attributo *ReadOnly*, che da a tali metodi la possibilità di interferire tra loro.

7.2.5 Persistenza dello stato

Ogni sistema Cloud generalmente è dotato di uno stato che viene mantenuto persistente tramite funzionalità e servizi di storage. Questo avviene anche in Orleans: la persistenza è integrata nei *grain*, il cui stato è infatti (quasi)

sempre mantenuto persistente e che contengono parte dello stato dell'applicazione. Questo permette anche di rendere visibili e disponibili le modifiche che vengono effettuate da una specifica attivazione anche alle altre di uno stesso *grain*. Oltre alla porzione di stato persistente è tuttavia possibile averne anche una transiente, la quale viene mantenuta all'interno di una specifica attivazione ed esiste solo durante il tempo di vita dell'attivazione stessa.

É possibile che un *grain* in un determinato momento esista solo in memoria persistente, se non ci sono richieste che lo riguardano; ciò significa che non esiste alcuna *activation* associata ad esso.

Per ragioni di prestazioni Orleans può anche decidere di non riportare subito in memoria persistente le modifiche fatte allo stato di un *grain*, ma di mantenerle in memoria volatile, rendendole comunque disponibili a tutte le attivazioni. Ciò tuttavia aumenta la vulnerabilità del sistema: lo spegnimento delle macchine su cui queste modifiche erano state memorizzate causa infatti la loro perdita.

7.2.6 Transazioni

Orleans fornisce un modello di consistenza basato su transazioni che permette di semplificare la gestione della concorrenza di un dato sistema. Tutti i *grain* che stanno processando una richiesta esterna vengono infatti inclusi in una transazione: ciò permette alla richiesta di essere eseguita atomicamente ed in maniera isolata. Le attivazioni che vengono eseguite all'interno di una data transazione sono completamente isolate dalle attivazioni eseguite in altre transazioni e non possono accedere a dati modificati da transazioni non ancora completate. Le transazioni possono terminare con successo oppure fallire, e le loro modifiche vengono rese persistenti e visibili atomicamente solo in caso di successo, altrimenti vengono completamente annullate.

Le transazioni implementano anche un semplice meccanismo per la gestione automatica degli errori: se un *grain* infatti fallisce o non è più disponibile, ad esempio per un problema di rete o del server sul quale era in esecuzione, la transazione per la quale esso stava eseguendo del codice viene fatta fallire e Orleans cerca automaticamente di rimandarla in esecuzione. In tal modo gli sviluppatori vengono liberati dalla responsabilità di gestire gli errori più comuni legati a problemi hardware o software.

7.2.7 Meccanismi e politiche di scalabilità

Le tecniche più comuni per raggiungere la scalabilità sono l'asicronicità, il partizionamento e la replicazione. Essi vengono tutti inclusi nel modello di programmazione ed in quello di esecuzione di Orleans. I *grain* infatti si basano su meccanismi di comunicazione asincrona e promuovono il partizionamento dello stato e della computazione. Inoltre Orleans può decidere a runtime di replicare le *activation* per aumentare le capacità del sistema, distribuire e bilanciare il carico di lavoro tra più server, automatizzando quindi la scalabilità.

Tuttavia questi meccanismi non garantiscono automaticamente la soluzione di tutti i problemi di scalabilità. Essi devono essere infatti iniettati nell'applicazione scegliendo le corrette politiche, le quali non possono essere automaticamente adottate da Orleans poiché sono strettamente legate alla semantica del problema. Ad esempio, se un *grain* diventa eccessivamente grande, la sua replicazione non può portare i benefici sperati e sarebbe quindi necessario suddividerlo in più *grain*. Come ciò debba esser fatto tuttavia dipende dalla semantica del *grain* che Orleans non può conoscere, di conseguenza tale operazione non può essere fatta automaticamente. In sintesi quindi Orleans fornisce i meccanismi, ma le politiche corrette per la loro utilizzazione devono essere introdotte da chi conosce la semantica dell'applicazione.

7.2.8 Interfacce dei *grain*

Piuttosto che sviluppare uno specifico linguaggio per la definizione di interfacce, Orleans utilizza le interfacce standard di .NET per definire quelle dei *grain*. Esse tuttavia devono seguire delle regole particolari:

- un'interfaccia di *grain* deve ereditare, direttamente o indirettamente, dall'interfaccia *IGrain*;
- tutti i *getter* devono restituire una *promise*, mentre non devono esserci *setter*, poiché in ambiente .NET essi non hanno valore di ritorno;
- gli argomenti dei metodi devono essere interfacce di *grain* oppure di un qualunque tipo serializzabile, in maniera tale da poter essere passati per valore.

7.2.9 Riferimenti ai *grain*

I riferimenti ai *grain* sono in realtà dei *proxy* i quali implementano anch'essi la stessa interfaccia del *grain* a cui sono associati e ai quali forniscono l'accesso. Essi sono l'unico modo che ha un client per accedere ad un *grain*.

Come per le promesse, anche i riferimenti ai *grain* possono trovarsi nei tre stati *unresolved*, *fulfilled* o *broken*. Il chiamante crea il riferimento ad un *grain* allocandone uno nuovo oppure richiedendone uno specifico già esistente. Se tale riferimento è ancora in stato *unresolved*, e su di esso vengono invocati dei metodi, questi vengono automaticamente messi in coda in attesa di poter essere eseguiti, in maniera completamente trasparente allo sviluppatore.

7.2.10 Creazione ed uso dei *grain*

Per ogni interfaccia di *grain* Orleans genera automaticamente due classi:

- Una *factory* statica che permette di creare, eliminare o cercare determinati *grain*
- Un proxy, usato internamente dal framework per convertire le invocazioni di metodi in messaggi

La *factory*, nel caso più semplice contiene i metodi per creare ed eliminare un *grain* e per effettuarne il cast ad un riferimento di altro tipo. É possibile specificare delle annotazioni ai membri dell'interfaccia del *grain*, come ad esempio *Queryable*, per generare dei metodi aggiuntivi nella *factory* per la ricerca di *grain* che soddisfino determinate condizioni.

Coerentemente con quanto affermato nella sezione precedente, invocando il metodo *CreateGrain* di una *factory*, viene immediatamente restituito il riferimento ad un *grain*, che può essere usato fin da subito per invocare determinate operazioni su di esso. Queste verranno messe quindi in una coda fino al momento in cui la creazione viene effettivamente completata. Se essa fallisce, allora tutte le promesse associate ai metodi invocati sul riferimento di quel *grain* falliranno automaticamente di conseguenza, come ci si aspetterebbe dai meccanismi di propagazione degli errori.

Nella figura 7.5 è mostrato un esempio di *factory*, in particolare quella necessaria alla gestione dei *ChirperAccount*. In essa compaiono i metodi necessari alla creazione, cancellazione e casting del *grain* oltre a quello in

grado di svolgere la ricerca basandosi sulla proprietà *UserName*, la quale è stata precedentemente dichiarata *Queryable* nell'interfaccia di tale *grain*.

```
(1)  public class ChirperAccountFactory
(2)  {
(3)    public static IChirperAccount
(4)       CreateGrain(string name);
(5)    public static void
(6)       Delete(IChirperAccount grain);
(7)    public static IChirperAccount
(8)       Cast(IGrain grainRef);
(9)    public static IChirperAccount
(10)      LookupUserName(String userName);
(11) }
```

Figura 7.5: *Factory* per la gestione del *grain ChirperAccount*: i primi due metodi si occupano della creazione e cancellazione di un *ChirperAccount*, il terzo di effettuare il casting esplicito da un *grain* generico ad un *ChirperAccount* (necessario in caso di *down-casting*) ed il quarto in grado di effettuare la ricerca di un *ChirperAccount* all'interno del sistema, dato il suo *UserName*.

7.2.11 Classi dei *grain*

Come già detto, la classe di un *grain* deve necessariamente implementare una o più interfacce di *grain*, ed ogni suo metodo deve restituire una *promise*. É tuttavia possibile, in fase di implementazione, specificare direttamente un valore di ritorno che non sia una promessa: esso verrà convertito a runtime in una *promise* da Orleans. Ciò permette una maggiore semplicità e chiarezza nella scrittura ed analisi del codice.

Come valore di ritorno è anche possibile specificare una promessa ottenuta dalla chiamata ad un altro *grain* oppure mettere in scheduling l'esecuzione di un delegato. Quest'ultimo caso è quello mostrato dalla figura 7.6.

7.2.12 Confronto con il modello ad attori

Come abbiamo potuto constatare analizzando il framework Orleans, esso adotta un modello che è molto simile a quello ad attori, introducendo però alcune differenze con lo scopo di portare un valore aggiunto nell'ambito del Cloud Computing. Esse nascono tutte dalla dissociazione presente tra il concetto di *grain*, che rappresenta l'attore in fase di programmazione, e quello di *activation*, a cui corrisponde invece la specifica istanza in esecuzione di un certo attore.

Per quanto riguarda le quattro caratteristiche fondamentali del modello ad attori, possiamo affermare che sicuramente Orleans implementa l'incapsulamento e la trasparenza alla locazione. Inoltre, nonostante non ci siano specifici riferimenti al concetto di *fairness*, possiamo supporre che la raffinata e complessa politica di scheduling adottata da Orleans, che deve, tra le altre cose, decidere anche quante *activation* assegnare ad un certo *grain* in ogni momento, sia in grado di garantire che nessun *grain* venga bloccato semplicemente perché il suo codice non viene più messo in esecuzione, da un certo momento in poi. L'unica proprietà che non viene quindi rispettata è quella di atomicità: questo discende da una precisa scelta effettuata tenendo conto delle prestazioni del sistema e dell'obbiettivo di raggiungere scalabilità automatica. Proprio per questo infatti viene introdotto il concet-

```
 (1)  AsyncCompletion FollowUser (string name)
 (2)  {
 (3)    IChirperPublisher user =
 (4)       ChirperPublisherFactory.LookupUserName(name);
 (5)
 (6)    IChirperSubscriber me = this.AsReference();
 (7)
 (8)    AsyncCompletion p = user.AddFollower(myName, me);
 (9)    return p.ContinueWith(() =>
(10)    {
(11)      this.Subscriptions[name] = user;
(12)    });
(13)  }
```

Figura 7.6: Questo esempio mostra come sia possibile specificare al posto di un comune valore di ritorno l'esecuzione di un delegato.

to di *activation* e viene deciso che ad ogni *grain* possano esserne associate più di una. Voler rispettare l'atomicità sarebbe quindi assurdo poiché implicherebbe che in ogni istante, per ciascun *grain*, sia in esecuzione solo un *activation*. Per questo vengono infatti introdotte le complesse politiche di riconciliazione dello stato, che sono state solamente accennate nel corso di questa trattazione. Un'ulteriore opposizione alla proprietà di atomicità può essere riscontrata nella possibilità, per ogni *activation*, di servire concorrentemente più richieste, se, come già visto, i relativi metodi sono marcati come *ReadOnly* o la classe del *grain* come *Reentrant*. Ciò tuttavia non comporta ulteriori problemi poiché queste marcature possono essere fatte solo se effettivamente l'esecuzione concorrente di più metodi non può tradursi in corse critiche con relativi problemi di consistenza dello stato.

7.3 Conclusioni

Come anticipato nell'introduzione di questo capitolo, l'analisi del modello ad attori e del framework Orleans dovrebbero averci fatto comprendere come essi siano effettivamente molto più adatti alla realizzazione di applicazioni distribuite, in particolare applicazioni Cloud, di quanto non lo siano i modelli più tradizionali, come l'*Object-Oriented*. Questi nuovi modelli tuttavia, perché possano effettivamente comportare i benefici sperati, necessitano di essere ben compresi e studiati da parte di coloro che decidono di adottarli; in particolare occorre comprendere veramente il significato dei vari concetti offerti dai modelli e dei costrutti messi a disposizione dai linguaggi di programmazione che li implementano.

Per comprendere questo fatto, possiamo considerare l'analogia con il modello *Object-Oriented*. Esso infatti ha permesso di realizzare applicazioni molto più strutturate, dotando i linguaggi di una semantica in grado di supportare nativamente alcuni principi architetturali, come l'incapsulamento e la componentizzazione, che altrimenti avrebbero dovuto essere innestati esplicitamente dagli sviluppatori. Tuttavia non ha vincolato questi ultimi al rispetto ed all'adozione di tali principi, impedendogli di realizzare applicazioni mal progettate: chi infatti per anni fosse stato abituato a realizzare programmi secondo un approccio sequenziale e non avesse compreso i concetti fondamentali dell'*Object-Oriented*, probabilmente avrebbe continuato a lavorare secondo il suo metodo, ad esempio, realizzando applicazioni com-

poste da un unico oggetto monolitico, nel cui metodo che funge da punto di ingresso del programma avrebbe trovato posto tutto il codice necessario alla sua esecuzione, realizzato come una lunga sequenza di istruzioni.

Questo ci aiuta a capire che anche l'adozione del modello ad attori (o simili) non introduce automaticamente nel programma la capacità di sfruttare il parallelismo e, nel caso specifico di Orleans, la scalabilità. Esso permette di raggiungere tali capacità, a condizione però che le entità fondamentali di tali modelli e linguaggi siano opportunamente utilizzate.

Concludendo quindi, è importante oggi, per poter sviluppare applicazioni Cloud e distribuite, conoscere ed adottare nuovi paradigmi più adatti a tali ambienti d'esecuzione, ma anche conoscere i modelli, le architetture ed i pattern che sono in grado di sfruttarne le potenzialità e risolverne i problemi.

Capitolo 8

Conclusioni

Giunti alla conclusione di questo lavoro possiamo sicuramente affermare che l'introduzione del *Cloud Computing* ha segnato una svolta nel mondo dell'informatica, svolta che sta tutt'ora progredendo e che deve essere tuttavia accompagnata da un cambiamento allo scopo di adattarsi alle novità ed alle differenze emergenti.

Tale cambiamento deve coinvolgere tutti coloro che operano in questo settore, i quali devono essere formati ai concetti fondamentali del *Cloud Computing*, arrivando a conoscere le caratteristiche peculiari di queste nuove piattaforme. In particolare sarebbe importante arrivare a comprendere le proprietà generali del Cloud, ovvero quelle che trascendono i singoli servizi o le implementazioni che ne forniscono i vari Cloud provider. Come infatti è importante conoscere le proprietà fondamentali di un calcolatore generico o l'architettura tipica di un dispositivo *mobile* per poterne sviluppare applicazioni, allo stesso modo è importante conoscere gli aspetti fondamentali delle piattaforme Cloud, affinché le applicazioni che vi saranno realizzate siano in grado di adattarsi ad esso sfruttandone le potenzialità.

Inoltre, alla luce del fatto che presto probabilmente il *Cloud Computing* arriverà a permeare ogni settore dell'*Information Technology*, sarebbe molto importante che esso divenisse accessibile ad una larga porzione della comunità informatica, che include non solo ingegneri ma anche tecnici e programmatori, i quali dispongono tipicamente di un livello di formazione inferiore. Perché ciò avvenga è necessario che il cambiamento di cui sopra coinvolga anche gli strumenti e le tecnologie, in modo che ne vengano sviluppati e diffusi taluni in grado di supportare il lavoro che queste figure

professionali andranno a svolgere nell'ambito del *Cloud Computing*. Come abbiamo visto nel sesto capitolo infatti, lo stato attuale della tecnologia permette di realizzare applicazioni Cloud, come sono state da noi definite precedentemente, tuttavia ad un costo notevole in termini di conoscenza necessaria, poiché al momento i modelli adottati non sono i più indicati agli ambienti altamente distribuiti.

In conclusione di questo lavoro e a seguito di mie personali riflessioni quindi, ritengo di poter indicare come ambito principale nel quale concentrare gli studi futuri concernenti il *Cloud Computing* proprio nelle tecnologie e negli strumenti di cui sopra, con particolare rilievo ai modelli ed ai framework come quelli illustrati nel settimo capitolo.

Bibliografia

[1] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *ACM Symposium on Cloud Computing*, October 2011.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: a distributed storage system for structured data, November 2006.

[3] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson. Toward an open cloud standard. *Internet Computing, IEEE*, August 2012.

[4] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rutschlin, and D. Schumm. Capturing cloud computing knowledge and experience in patterns. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, 2012.

[5] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck. A collection of patterns for cloud types, cloud service models, and cloud-based application architectures, May 2011.

[6] R. Karmani and G. Agha. Actors, 2011.

[7] W. Kim. Cloud architecture: A preliminary look. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services.*

[8] R. Moreno-Vozmediano, R. Montero, and I. Llorente. Iaas cloud architecture: From virtualized data centers to federated cloud infrastructures, February 2011.

[9] B. Sodhi and T. Prabhakar. Cloud-oriented platforms: bearing on application architecture and design patterns. In *Proceedings of the 2012 IEEE Eighth World Congress on Services*, 2012.

[10] J. Varia. Architecting for the cloud: best practices, January 2010.

Elenco delle figure

1.1 Architettura WebApp 12

2.1 Architettura di OpenStack 37
2.2 Struttura di Eucalyptus 38
2.3 OpenNebula 40

3.1 Elasticità 48
3.2 Servizi di Amazon 54

4.1 Webtable - righe 58
4.2 Webtable - colonne 60
4.3 Webtable - codice modifica 62
4.4 Webtable - codice lettura 63

6.1 Applicazioni modulari 91
6.2 Accoppiamento debole 92
6.3 Componenti *stateless* 93
6.4 Componenti a singola istanza 94
6.5 Componenti a singola istanza configurabili 95
6.6 Componenti idempotenti 96
6.7 MapReduce 97
6.8 Accoppiamento forte 102
6.9 Accoppiamento debole 102

7.1 Attori 116
7.2 *Pipeline* 119
7.3 *Divide et impera* 120
7.4 Promesse 124
7.5 Factory 129

7.6 Scheduling di delegati 130