ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Campus di Cesena - Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

PATTERN E TECNOLOGIE PER LO SVILUPPO DI APPLICAZIONI CLOUD: ORLEANS COME CASO DI STUDIO.

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
ALBERTO MODIGLIANI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2013–2014
SESSIONE I

PAROLE CHIAVE

Cloud Computing

Pattern

Piattaforme

Attori

Orleans

Alla mia famiglia e ai miei amici

Indice

Introduzione

Uno dei temi più discussi ed interessanti nel mondo dell'informatica al giorno d'oggi è sicuramente il Cloud Computing. Nuove organizzazioni 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.

Lo scopo che mi propongo con questa tesi è quello quindi di cercare di fare chiarezza in quello che è il mondo del Cloud computing, focalizzandomi particolarmente su quelli che sono i design pattern più utilizzati nello sviluppo di applicazioni di tipo cloud e presentando quelle che oggi rappresentano le principali tecnologie che vengono utilizzate sia in ambito professionale, che in ambito di ricerca, per realizzare le applicazioni cloud, concentrandomi in maniera particolare su Microsoft Orleans.

La tesi risulta così strutturata: nel primo capitolo vengono sostanzialmente date le nozioni di base per il cloud computing. Verranno di conseguenza illustrati i concetti principali che saranno poi alla base dei successivi

capitoli. Nel secondo capitolo invece si mostreranno quelli che oggi sono i design pattern più utilizzati nello sviluppo di applicazioni cloud. Si partirà di conseguenza con una breve introduzione alla filosofia dei design pattern e sulle motivazioni del loro utilizzo nell' ambito dell' ingegneria del software. In seguito si daranno alcune linee guida per lo sviluppo di applicazioni cloud che sfruttino efficacemente tutte le potenzialità che questo ambiente offre e poi verranno spiegati (anche con l' ausilio di figure) i design pattern. Il terzo capitolo invece offre una panoramica sulle tecnologie oggi più utilizzate in ambito professionale per lo sviluppo di applicazioni cloud, ed in particolare verranno presentate due Platform as a Service (PaaS) che sono Apprenda e Microsoft Windows Azure e un middleware, Cloudify. Per quel che riguarda la prima è stata scelta in quanto piattaforma fortemente in ascesa in ambito professionale, grazie alle sue caratteristiche di flessibilità e portabilità. Azure invece viene presentato sia perché risulta un ottima piattaforma, sia perché ci verrà poi utile averne i concetti base nel capitolo sei dove si vedranno due semplici esempi applicativi basati su Orleans. Cloudify invece viene introdotto in quanto è un middleware free, ovvero che non richiede costi per essere utilizzato, che si adatta bene a tutte le piattaforme e che, per questo, è molto utilizzato. Nel quarto capitolo si parla invece del modello di programmazione ad attori. Questo è un capito che serve sostanzialmente per dare i concetti base per capire la filosofia che sta dietro ad Orleans, poiché quest' ultimo si basa proprio sul modello ad attori. Nel quinto capitolo parliamo invece di Orleans, un framework che la Microsoft ha sviluppato per rendere più facile e intuitivo lo sviluppo di applicazioni cloud anche per chi non è un esperto in questo settore. E' stato deciso di prendere proprio Orleans come caso di studio perché laMicrosoft sta investendo molto su questo progetto (basti pensare che è stato utilizzato per la realizzazione dei servizi Cloud di Halo 4), anche se ad oggi rimane ancora a livello di ricerca, e quindi molto probabilmente in futuro se ne sentirà parlare con insistenza. Infine il sesto capitolo illustra due semplici esempi applicativi nei quali sostanzialmente vengono applicati i concetti spiegati all' interno del capito precedente su due esempi reali.

Capitolo 1

Introduzione al Cloud Computing

Il Cloud Computing è un modo di pensare infrastrutture, piattaforme e software come un servizio (as a service). Di norma in questa visione vi è un Cloud provider che offre i suoi servizi e le sue risorse computazionali, di norma a pagamento, ad un pubblico di Cloud consumer o Tenant i quali, in questo caso, non possiedono assolutamente nulla ma si devono semplicemente loggare tramite un client ad un server contenente il servizio che hanno acquistato e che vogliono utilizzare. Questo è solo uno dei tanti vantaggi che il Cloud Computing offre ai propri clienti fra gli altri vi sono:

- il fatto di non dover più gestire e mantenere le macchine server fisiche. Infatti in questo sistema è il Provider che si occupa di questi aspetti, rendendo di conseguenza l' utente del tutto immune a problemi come guasti o spese dovute all' acquisto di nuove macchine o alla manutenzione di queste ultime.

- 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. Questo concetto presuppone il fatto che il consumer possa utilizzare le risorse messe a disposizione On Demand, ovvero su richiesta. Questo è uno degli altri motivi che rendono il Cloud Computing un modello vincente, perché ci consente di utilizzare solo ciò che esattamente ci

serve, quando ci serve e senza alcun vincolo, risparmiando così non solo soldi rispetto alla configurazione classica nella quale si comprano macchine server che la maggior parte delle volte non vengono utilizzate completamente, ma ci si ripara anche da picchi inaspettati di utenza che nel modello appena citato avrebbero causato non pochi problemi e disagi ai fruitori della nostra applicazione.

- metodo di pagamento pay-per-use, che consente al tenant di pagare solo in base alle risorse computazionali che usa. Questo è uno dei punti più controversi del cloud perché dal lato del provider non è così facile definire dei metodi di pagamento che si basino sull' utilizzo delle risorse usate (bisogna pensare infatti che non è che ad ogni utente viene affidata una macchina fisica quando esso ne richiede una, ma vi è dietro una forte virtualizzazione). D'altro canto obbliga il tenant a progettare le proprie applicazioni in maniera tale da utilizzare solo le risorse strettamente necessarie.

- 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.

- indipendenza dai dispositivi e dalle loro locazioni; questo modello svincola totalmente il Consumer dal doversi preoccupare di quei due fattori. Infatti esso può usufruire dei servizi offerti dal provider senza sapere effettivamente dove essi siani collocati, ma anche senza sapere quale macchina effettivamente li stia fornendo; tutto quello che deve fare è semplicemente connettersi tramite un programma Client (che può anche essere un web browser) ai servizi da lui acquistati.

Va inoltre sottolineata la presenza di un altra figura all' interno del Cloud Computing ovvero quella del Cloud Career, che è colui che si occupa di attivare, mantenere attiva e mantenere la connessione fra consumer e provider al fine di potergli consentire l' interazione. Ad esempio in uno scenario in cui il tenant si connette tramite internet il Cloud Career dovrà occuparsi di fornire le giuste caratteristiche di connettività.

1.1 Modelli di servizio

I Cloud Provider di norma organizzano i servizi da loro forniti organizzandoli in tre categorie: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) e infine Software as a Service (SaaS). Queste tre classi di servizi spesso vengono strutturati secondo una architettura a livelli:

- il livello inferiore (IaaS) è quello che 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.

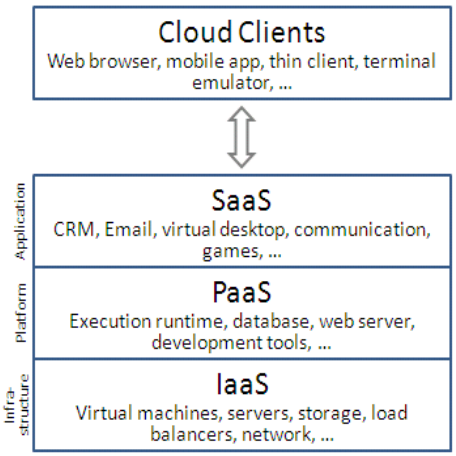


Figura 1.1: I livelli del cloud computing

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.

1.1.1 Infrastructure as a Service (IaaS)

In questa tipologia di servizio il provider mette a disposizione quelle che sono le risorse hardware, fornendo all' utente quella che ai suoi occhi viene vista come una macchina singola tutta per se. Ovviamente non è così, quello è solamente ciò che vede l' utilizzatore finale, in realtà ciò che veramente gli viene fornito è una macchina virtuale che si basa su un hardware reale. I vantaggi di questo tipo di servizio sono senza dubbio il fatto che l' utente non si deve preoccupare di quella che è la gestione della macchina e inoltre non la ha effettivamente acquistata, quindi, qualora non gli servisse più è sempre libero di rescindere il contratto. Sul quel computer il tenant può installare il sistema operativo che più gli aggrada, le applicazioni che vuole e tutto quello di cui ha bisogno, è lui che si occupa di gestire la parte Software. Uno dei problemi principali di questo servizio è il fatto di poter garantire un giusto livello di isolamento a tutti i vari utilizzatori. Infatti l' utente deve avere la percezione di essere l' unico utilizzatore della macchina, quando noi sappiamo che in realtà non è così perché in realtà l' hardware fisico è condiviso con altre macchine virtuali. Quindi la sfida sta nel fatto che la richiesta di un aumento di prestazioni su una macchina virtuale non vada ad incidere sulle prestazioni delle altre che stanno sulla stessa macchina fisica. Per risolvere questo problema occorre potenziare i componenti del sistema che si occupano della gestione delle risorse e, 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. In figura 1.2 si mostra ciò che gestisce l' utente e ciò che gestisce il provider in un IaaS.

1.1.2 Platform as a Service (PAAS)

In questo modello di servizio, differentemente dall' IaaS, il provider offre al consumer una piattaforma vera e propria, rendendolo quindi del tutto trasparente a quelli che sono i problemi legati all' Hardware. In questa configurazione l' utilizzatore del servizio non può scegliere nulla del sistema operativo da utilizzare o di quali strumenti di sviluppo mettere sulla propria piattaforma, può solamente svilupparci sopra le proprie applicazioni e metterci i propri dati. Inoltre l' utente non ha nemmeno coscienza di quali siano le risorse utilizzate dalla propria applicazione ne su quale macchina essa stia effettivamente girando. Ovviamente la piattaforma non è statica;

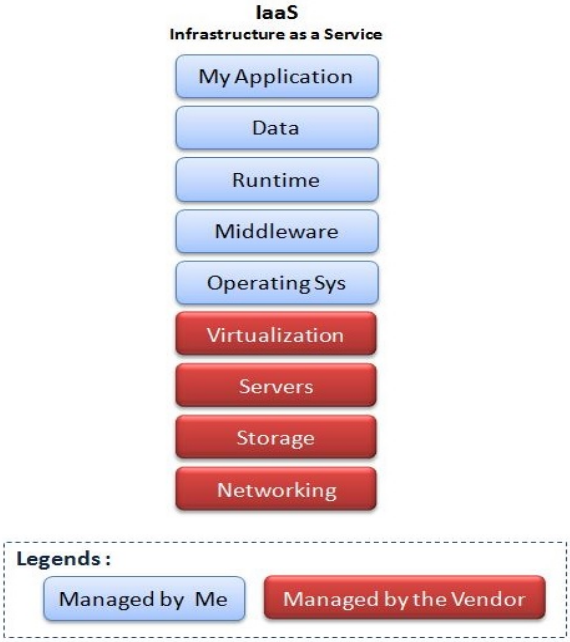


Figura 1.2: Gestione nell' IaaS

i provider infatti mettono a disposizione una serie di tool per poterla configurare secondo le proprie esigenze. In questo modo gli utilizzatori hanno un modo, mediato dal provider, per poter agire sulla piattaforma reggente per, ad esempio, richiedere più risorse per la propria applicazione che deve scalare. In questo particolare modello di servizio il problema dell' isolamento diventa ancora più rilevante, infatti esso 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. In figura 1.3 si mostra ciò

che gestisce l' utente e ciò che gestisce il provider in un PaaS.

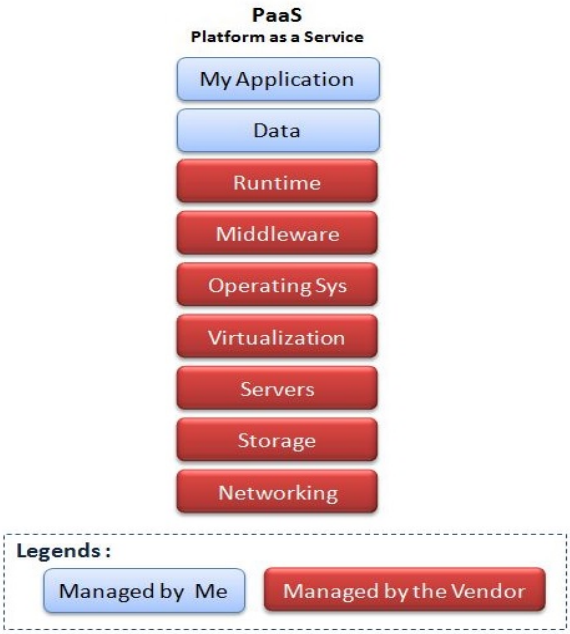


Figura 1.3: Gestione nel PaaS

1.1.3 Software as a Service (SAAS)

In questo modello di servizio il provider offre direttamente degli applicativi software ai propri clienti che poi saranno in grado di poterli utilizzare, sotto una certa forma di pagamento del servizio, loggandosi attraverso un programma client che può essere di qualsiasi tipo. Qui il cliente non ha alcun tipo di potere ne sulla piattaforma ne sulla infrastruttura hardware, si limita solamente ad utilizzare l' applicazione che gli viene offerta, configurandola magari in base a quelle che sono le sue preferenze o esigenze. Anche in questa situazione l' utente deve avere la percezione di essere il solo utilizzatore dell' applicazione, quindi non sono molto tollerati ritardi dovuti a latenza o un calo delle performance dovuto ad un picco di accessi temporaneo. Per risolvere questo problema o si replicano le istanza dell' applicazione

per ogni utente (approccio sconsigliato e spesso infattibile) oppure si rende l' applicazione consapevole di avere più utenti attivi contemporaneamente, permettendo così il giusto isolamento e la corretta condivisione dei componenti. I vantaggi di questo modello sono molteplici sia per il consumer che per il fornitore, dal punto di vista del primo il vantaggio più grande è quello di possedere una applicazione senza in realtà averla installata sulla propria macchina, ciò implica il fatto che esso non utilizza le proprie risorse computazionali, tranne quelle necessarie per far funzionare il programma client e quelle per lo scambio di dati. Dal punto di vista del secondo il vantaggio fondamentale riguarda sicuramente quello che è il mantenimento dell' applicazione; infatti qualora si renda necessario un aggiornamento o per sistemare bug o per aggiungere funzionalità, il provider non dovrà fare altro che andare a cambiare il codice sorgente della sua applicazione e automaticamente tutti i tenant vedranno le modifiche apportate. Questo migliora sicuramente quanto avviene in un modello tradizionale nel quale si rilasciano delle patch che poi tutti gli utilizzatori devono in qualche modo scaricare e installare. In figura 1.4 si mostra ciò che gestisce l' utente e ciò che gestisce il provider in un SaaS.

1.2 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

1.2.1 Public cloud

Nel modello del public cloud, che è quello che si adatta meglio alla descrizione iniziale di cloud, abbiamo di norma un provider che si occupa delle

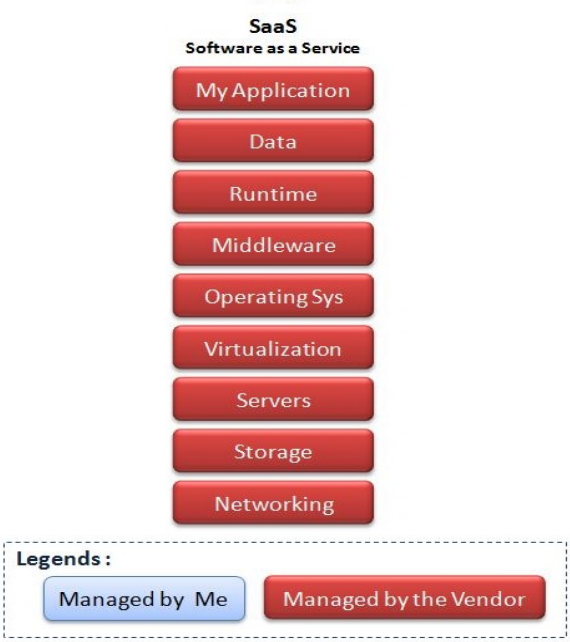


Figura 1.4: Gestione nel SaaS

gestione delle risorse sia hardware (come server e componenti di rete) ma anche software (come esempio un load balancer) e poi una serie di utenti che, tramite i già noti metodi di pagamento pay-per-use, sfruttano queste risorse. Questa modalità di distribuzione è adatta ad aziende di piccole medie dimensioni le quali, in questa maniera, non si devono più preoccupare di affittare o comprare cluster fisici, ma possono richiedere di pagare solo per quello che usano.

Tutto questo però ha un lato negativo, questo genere di costi non vengono ammortizzati con il tempo, ma lungo tutta la vita del prodotto dovranno essere pagati i costi di contratto con il provider, limitando, da un certo punto di vista, l'efficacia stessa del sistema. Una delle critiche più grandi che viene però fatta al cloud pubblico è senza dubbio quella che riguarda la sicurezza. Con questo modello infatti l' utente non ha idea di dove siano fisicamente allocati i propri dati e, inoltre, non è neanche responsabile delle politiche di sicurezza, che vengono invece gestite dal provider, che comun-

que garantisce una soglia minima di sicurezza. Qualora si abbiano dati che non si vogliono assolutamente condividere o che si preferisce mantenere al sicuro occorrerebbe optare, qualora si avessero le disponibilità economiche, ad un modello di cloud privato che, dal punto di vista della sicurezza, è assolutamente preferibile.

1.2.2 Private cloud

Nel modello di private cloud abbiamo una infrastruttura dedicata interamente ad una sola azienda, questa viene ad essere gestita interamente dalla azienda stessa, senza alcuna forma di contratto con terzi. Ovviamente i costi per una azienda sono di gran lunga superiori rispetto a quelli che si avrebbero con l' utilizzo di un cloud pubblico, in quanto in questa situazione tutti i costi di gestione e manutenzione vengono completamente addebitati all' azienda stessa in quanto possessore delle macchine server; non solo deve anche disporre di posti adeguati alla locazione delle macchine e personale adeguato in grado di fornire la giusta manutenzione. Il vantaggio che si ha con una architettura di questo tipo è senza dubbio legato, come si diceva nella sezione al public cloud, alla sicurezza; infatti i servizi offerti dal cloud vengono sfruttati solamente dalla azienda, inoltre in questo modello è l' azienda che è responsabile delle politiche di sicurezza da adottare per i propri dati ed è anche pienamente consapevole del luogo fisico nel quale i propri dati sono ubicati. Se questo è un vantaggio lo svantaggio più grande è sicuramente quello relativo alla scalabilità. Infatti, come è facile intuire, in una situazione di cloud privato si ha un numero limitato di risorse, diversamente da quanto invece si ha nel cloud pubblico che ha risorse illimitate (questo solamente dal punto di vista concettuale), quindi qualora una applicazione avesse bisogno di scalare richiedendo un aumento delle risorse, ma queste, data la loro limitatezza, fossero tutte occupate, si andrebbe incontro ad una serie di problemi e di ritardi che andrebbero sicuramente ad infastidire i nostri utenti. Un modo per tentare di risolvere questo problema è, qualora le politiche di privacy e sicurezza lo permettano, utilizzare quello che è un private virtual cloud. Tramite questa soluzione praticamente non si realizza un cloud privato utilizzando quelle che sono le risorse della azienda, ma lo si costruisce su un cloud pubblico realizzando, per l' appunto, quello che è un private cloud virtuale. In questo modo si risolvono quelli che sono i problemi legati alla scalabilità, in quanto si ha la possibilità di utilizzare le

risorse di un cloud pubblico, rinunciando però a quella che era la sicurezza garantita da un cloud privato puro.

1.2.3 Commumity 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.

1.2.4 Hybrid cloud

Un Hybrid Cloud è una composizione dei tre tipi di cloud precedenti dove le caratteristiche di ognuno rimangono ben distinte, ma questi sono collegati insieme per poter avere maggiori benefici e per adeguarsi al meglio a quelle che sono le esigenze di chi li utilizza. 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. Un esempio classico di Hybrid Cloud è quello nel quale una azienda mantiene tutti i dati di interesse e le applicazioni su un cloud privato ma, qualora si abbiano picchi di accesso inattesi e le risorse interne siano finite, si scala utilizzando quelli che sono i servizi di un cloud pubblico. Quindi in situazioni normali di utilizzo vengono utilizzate le sole risorse computazionali che sono interne all' azienda, senza aggiungere costi, e si va sul cloud pubblico solo quando

ce ne è l' effettivo bisogno. 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.

1.3 Virtualizzazione

Essendo la virtualizzazione uno dei concetti chiave del Cloud Computing cercherò qui di elencare quelle che sono le caratteristiche e le componenti principali di questa tecnologia. Innanzi tutto con il termine virtualizzazione si intende quel processo attraverso il quale si crea una versione virtuale di qualcosa; questo include, ma non si limita solamente, la virtualizzazione di piattaforme hardware (che però è quella che a noi interessa) e la virtualizzazione di sistemi operativi. Mediante il primo processo è possibile creare delle risorse hardware simulate che poi possono venire utilizzate come delle macchine vere e proprie dagli utenti senza che questi ultimi siano effettivamente a conoscenza di quali dispositivi fisici stiano effettivamente utilizzando. In questo ambiente viene chiamata macchina host la macchina sulla quale sta avvenendo la virtualizzazione e macchina guest (ospite) la macchina virtuale vera e propria. Le parole host e guest sono inoltre utilizzate per distinguere il software in esecuzione sulla macchina fisica rispetto a quella virtuale. 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.

Infine per concludere questa mia sezione sulla virtualizzazione passo in rassegna quelli che sono i principali modi di fare virtualizzazione.

1.3.1 Virtualizzazione Completa

Attraverso questo tipo di virtualizzazione si simula completamente quella che è la architettura hardware, di conseguenza tutte le chiamate vengono intercettate dall' Hypervisor e mappate in opportune interazioni con l' hardware sottostante. Il vantaggio fondamentale di questo tipo di virtualizzazione è che il sistema operativo non sa di essere eseguito su una macchina virtuale e, di conseguenza, non ne va cambiato il codice. D'altro canto però questa metodologia non ha delle buone performance in quanto il sistema viene molto sovraccaricato.

1.3.2 Paravirtualization

Mediante questo tipo di virtualizzazione si cerca di incrementare quelle che sono le performance rispetto a quelle della virtualizzazione completa. Per fare questo pero si rinuncia a virtualizzare parte dell' hardware e quindi bisogna appoggiarsi direttamente a quello fisico. A questo punto il sistema operativo ospite è conscio di trovarsi su una architettura virtualizzata e di conseguenza ne va cambiato il codice poiché le chiamate di sistema devono venire sostituite con chiamate all' hypervisor che ne offre un interfaccia simile. I problemi relativi a questo tipo di virtualizzazione sono fondamentalmente due : in primo luogo, la sua compatibilità e portabilità potrebbe essere in dubbio, poiché deve supportare anche il sistema operativo immodificato. In secondo luogo, il costo di manutenzione di sistemi operativi paravirtualizzati è alto, perché possono richiedere modifiche profonde al kernel del sistema operativo. Ovviamente però il vantaggio in termini di performance è molto alto e questo la rende una forma di virtualizzazione ampiamente utilizzata.

1.3.3 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 dalla 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 VT10) 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.

Capitolo 2

Pattern per applicazioni cloud

In questo capitolo andrò ad elencare quello che è un insieme di possibili pattern per la realizzazione di cloud app. I pattern verranno presentati nella forma : problema da risolvere/contesto, soluzione cercando così di dare una spiegazione il più possibile chiara e schematica. Prima di iniziare a parlare di pattern farò una introduzione sul ruolo dei pattern e il loro scopo all' interno dell' ingegneria del software e dei principi fondamentali dei quali bisogna essere a conoscenza nel momento in cui si decide di realizzare una cloud app.

2.1 Design Pattern

Innanzi tutto i design pattern vengono definiti come una soluzione generale progettuale ad un problema ricorrente; essi non possono essere subito tradotti in codice da far compilare alla macchina, ma forniscono un modello, o danno dei suggerimenti, su come risolvere quel particolare problema per il quale sono stati creati. Esistono diverse tipologie di design pattern, esse sono:

- **Algorithm strategy patterns**: servono per affrontare i problemi legati alle strategie di alto livello che descrivono come sfruttare le caratteristiche delle applicazioni su una piattaforma di calcolo.

- **Computational design patterns**: servono per affrontare i problemi relativi alla computazione in generale.

- **Execution patterns**: affrontano i problemi relativi al supporto delle applicazioni in esecuzione; questi includono anche strategie per l'esecuzione di stream di task e per la creazione di blocchi che supportino la sincronizzazione fra task.

- **Implementation strategy patterns**: affrontano i problemi relativi all' implementazione di codice sorgente per il supporto dell' organizzazione del programma e per le strutture dati comuni specifiche della programmazione parallela.

- **Structural design patterns**: rispondono alle preoccupazioni relative alle strutture di alto livello delle applicazioni in fase di sviluppo.

L' utilizzo di pattern nella programmazione è sicuramente un fattore cruciale. Essi infatti non solo consentono al programmatore o al progettista di risparmiare tempo, poiché non dovrà stare ogni volta a trovarsi la soluzione al problema che sta correntemente affrontando essendo che ce l'ha già tra le mani, ma avrà sicuramente anche una soluzione di qualità. Ovviamente l' utilizzo dei pattern non garantisce che il nostro software soddisferà i requisiti richiesti; essi infatti vanno utilizzati con coscienza e, nelle varie fasi di progettazione, va tenuto conto di quelli che possono essere i pro e i contro nell' utilizzo di un determinato pattern e vedere se questi sono conformi coi requisiti espressi dal committente. Tutto questo per dire che i pattern rappresentano un ottima linea guida, ma che non sono la soluzione a tutti i nostri problemi e che, soprattutto, non bisogna abusarne.

Un altro problema dell' utilizzo dei pattern è che essi non promuovono nuove idee; infatti, come viene logico pensare, se tutti utilizzano soluzioni pre-confezionate nessuno potrà mai avere l' opportunità di implementarne di nuove, magari anche migliori.

Infine va detto che un pattern non va preso e utilizzato in maniera automatica, ma bisogna prenderlo ed adattarlo a quello che è il nostro problema. Possono esistere infatti mille sfaccettature di un problema, tutte fra di loro simili, ma che hanno caratteristiche diverse che vanno prese in considerazione per avere un prodotto software di qualità.

2.2 Principi per poter sviluppare cloud app

Partendo dal presupposto che non vi è ancora una definizione chiara e precisa di ciò che possiamo definire come una applicazione cloud, definiremo quest' ultima come una applicazione che è stata progettata per essere messa in esecuzione su una piattaforma cloud qualsiasi. Questo ci pone subito un problema rilevante ovvero il problema della piattaforma. Il progettista infatti dovrebbe essere il più consapevole possibile di quelle che sono le caratteristiche fondamentali delle varie piattaforme cloud che sono sul mercato e scegliere quella con le caratteristiche più adatte alle esigenze della sua applicazione. Scegliendo quindi una piattaforma a caso potrebbe capitare che essa non offra tutti i servizi di cui l' applicazione ha bisogno, e ciò porterebbe sicuramente ad una progettazione più difficile. In questo contesto la fase dell' analisi del problema diventa fondamentale. Una buona analisi del problema infatti sarà in grado di mettere in luce quelle che sono i punti critici del sistema che vogliamo andare a costruire, di conseguenza il progettista, leggendone l' architettura logica, sarà in grado di poter scegliere la piattaforma che permetterà di risolvere i problemi nella maniera più agevole possibile.

Il fatto che ogni piattaforma abbia le sue caratteristiche e le sue proprietà mette anche in discussione la facilità di trasporto di una applicazione cloud da un provider all' altro. Se infatti una applicazione è stata progettata per essere mandata in esecuzione su una piattaforma, facendo di conseguenza affidamento su tutte quelle che sono le caratteristiche che quella offre, poi ad un certo punto, per motivi di natura economica o di altra natura, si sceglie di cambiare provider, la nuova piattaforma potrebbe non offrire le stesse proprietà. Questo potrebbe mandare in crisi la nostra applicazione costringendoci a cambiarne l' implementazione in maniera massiccia e di conseguenza a perdere tempo e denaro. Una buona progettazione e una buona analisi del problema possono mitigare questi svantaggi, se si persegue anche lo scopo di essere il più possibile technology independent, ma comunque il passaggio da un provider all' altro rimane una operazione complessa e, il più delle volte, dispendiosa.

A questo punto bisogna dire che eseguire una applicazione su una piattaforma cloud (indipendentemente da quanto detto sopra) non la rende automaticamente una cloud app. Infatti, se progettiamo la nostra applicazione come una applicazione standard essa non avrà, molto probabilmente,

le proprietà tipiche di una cloud application, come ad esempio la grande scalabilità ed elasticità. Per facilitare questo passaggio, ovvero far passare una applicazione qualsiasi a cloud app, i vari provider mettono a disposizione diverse caratteristiche alla loro piattaforma, che possono essere più o meno efficaci; comunque, come sempre in linea di massima, una buona analisi del problema e un progetto attento riducono sicuramente questo sforzo.

Un altro fattore di cui bisogna essere in qualche modo consapevoli quando si sviluppa una applicazione cloud è la latenza. Nelle applicazioni SaaS tradizionali infatti si tende a considerare che i componenti lavorino tutti su una stessa macchina e che di conseguenza il tempo per interazioni fra di essi sia trascurabile. Conseguentemente a questo fatto si tende a tenere in considerazione solo del tempo che intercorre fra le richieste dei client e le risposte dei server. Questo non può più essere considerato vero quando stiamo parlando di applicazioni cloud. Infatti in questo modello oltre a tener conto della latenza classica bisogna anche considerare il fatto che componenti appartenenti alla stessa applicazione potrebbero essere in esecuzione su macchine diverse per i più disparati motivi e di conseguenza il tempo che intercorre fra le loro interazioni non può più essere considerato trascurabile. Bisogna essere consapevoli di questo fatto perché altrimenti si può incorrere nel rischio di creare una applicazione che può essere reattiva in alcuni casi, ovvero quando i componenti sono tutti sulla stessa macchina, ma meno in altri e questo senza alcun controllo da parte del programma perché di norma la distribuzione fisica dei componenti è scelta dalla piattaforma.

Un altro fattore importante di cui sicuramente tener conto, come già sottolineato precedentemente, è quello della sicurezza. Infatti nelle applicazioni tradizionali si va a lavorare sui dati in centri sicuri e affidabili, di cui si conoscono bene i confini con l' ambiente esterno, quindi non ci sono problemi rilevanti di sicurezza, almeno per quel che riguarda il salvataggio dei dati. Nell' ambiente cloud invece, sopratutto se stiamo parlando di un Hybrid Cloud, questi confini sono molto meno definiti e persistenti, perciò bisogna prestare attenzione a dove salviamo i dati rilevanti per la nostra applicazione; in generale vanno sicuramente aumentate le politiche di sicurezza rispetto ad applicazioni tradizionali.

Un altra proprietà che una applicazione cloud possiede è senza dubbio quella dell' elasticità, ovvero la capacità di cresce e ridursi di dimensione senza calare nelle performance. Se si progetta una applicazione non per il cloud difficilmente essa possiederà questa proprietà. Inoltre di norma essa

non sarà progettata per gestire dati allocati su più macchine e che possono anche cambiare la loro posizione nel corso dell' esecuzione. Tutto questo si tradurrà, quando la nostra applicazione verrà messa sul cloud, in uno spreco della banda che porterà ad un calo nelle performance e anche ad un aumento dei costi. Per questo motivo 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.

Quanto detto sopra ci introduce alla consapevolezza dei costi. Come già detto infatti i provider utilizzano un metodo di pagamento pay- per-use che consente di pagare solo in base alle risorse che effettivamente vengono utilizzate. In questo scenario diventa molto importante progettare una applicazione in maniera tale da poter utilizzare le risorse delle quali si dispone nel miglior modo possibile; un uso inappropriato infatti porterebbe sicuramente ad un aumento dei costi. Bisognerebbe inoltre conoscere il costo per tipologia di risorsa, in maniera tale da poter prendere la decisione che, in termini di utilizzo delle risorse, abbia il minor costo. 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.

Quest' ultimo paragrafo ci ha fatto capire quanto possa essere importante all' interno di una applicazione cloud tenere il traffico ridotto al minimo ed è per questo che, prima di passare ad illustrare alcuni dei pattern più utilizzati nell' ambito cloud, andrò ad elencare alcune soluzioni che permettono, se correttamente utilizzate, di poter ridurre l' utilizzo delle risorse.

2.2.1 Componenti stateless

Per poter essere eseguita una applicazione ha bisogno di informazioni che gli vengono fornite dall' esterno. Alcune di essere però risultano non essere fondamentali o, addirittura, necessarie solo a causa di una scelta architetturale specifica. Per cercare di ridurre il traffico che viene dall' esterno è bene cercare di progettare i componenti che fanno parte della nostra applicazione in maniera tale che abbiano bisogno della minor quantità di informazioni possibili per poter essere mandati in esecuzione. Portando quanto detto all' estremo si arriva alla definizione di componenti stateless ovvero componenti

che, per poter funzionare, non hanno bisogno di tenere traccia di quanto avvenuto precedentemente nel corso dell' esecuzione poiché trattano ogni comunicazione come una interazione a sé stante.

2.2.2 Creazione dei dati

Essendo chiaro che non potrà essere possibile realizzare tutti i componenti come stateless, poiché delle informazioni a volte sono assolutamente necessarie per far partire dei componenti , occorre a questo punto prendere in esame quanto sia effettivamente necessario il fatto che i dati vengano inviati dall' esterno. Una soluzione alternativa infatti potrebbe essere quella di creare i dati che sono necessari ai componenti all' interno del cloud, in maniera tale da ridurre notevolmente quello che è il traffico verso l' esterno.

Ovviamente anche questa soluzione non potrà essere sempre applicabile in quanto non tutti i dati possono essere ricavati internamente ai confini del cloud e di conseguenza vanno inviati dall' esterno. In questi casi occorre fare però attenzione alla sicurezza, rafforzandone quelle che sono le politiche.

2.2.3 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.

E' 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.

2.2.4 Posizionamento dei componenti

Qualora la piattaforma sulla quale abbiamo scelto di mandare in esecuzione la nostra applicazione ci dia la possibilità di scegliere dove poter posizionare i componenti, occorre posizionarli con cura. Analizzando infatti le interazioni e il flusso dei dati, bisognerebbe cercare un posizionamento che

consenta di mantenere le interazioni il più possibile limitate all' interno dell' infrastruttura cloud e il meno frequenti possibile.

2.2.5 Pattern di comunicazione

Durante l' esecuzione del nostro programma i pattern di comunicazione possono cambiare, ad esempio, in base alla natura dei componenti e dell'applicazione stessa o agli scenari in cui si trova ad essere eseguita. Occorre di conseguenza monitorare questi pattern e prendere in considerazione l' idea di far migrare i dati verso chi li sta utilizzando di più in quel momento. Ovviamente anche la migrazione dei dati ha un costo, perciò occorrerà valutare bene se i vantaggi dovuti a questo spostamento siano effettivamente tali da poterlo giustificare, anche perché è difficile stimare con certezza per quanto tempo un pattern manterrà un picco di accesso elevato. Inoltre spostando i dati si verranno a creare nuovi pattern e componenti che prima magari impiegavano poco tempo per accedere al dato adesso se lo vedranno aumentare. In poche parole la migrazione dei dati è una operazione che può e deve essere fatta in certi casi, ma della quale bisogna essere ben consci.

In alternativa, in presenza di dati sufficientemente statici, si può prendere in considerazione di usare delle politiche di caching che possono aiutare notevolmente ad abbattere i costi ed aumentare le performance legate alle latenze di rete.

2.2.6 Partizionamento e ridistribuzione dei dati

Quando si ha una grande mole di dati da dover processare può aver senso pensare, per alleggerire quella che è la computazione, di partizionare questo blocco e di distribuire le varie partizioni ad entità diverse che si occuperanno di fare le elaborazioni su di essi in maniera parallela. Questo processo ha un costo sufficientemente elevato in termini di consumo di banda, perciò occorre, anche in questa situazione, saper scegliere bene quando utilizzare questa tecnica e quando invece non risulta conveniente farlo; inoltre sarebbe opportuno fare tutte le operazioni possibili su una determinata partizione prima di cambiarla, in maniera tale da ridurre al minimo le operazioni di partizionamento e ridistribuzione dei dati.

Un' altra situazione nella quale avvengono operazioni di questo genere è quando vengono assegnate o tolte macchine alla nostra applicazione

dalla piattaforma sottostante. Anche in questo caso le operazioni di partizionamento e ridistribuzione dei dati andrebbero fatte in maniera tale da minimizzare l'utilizzo della 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.

2.3 Pattern

In questa sezione andrò a fare un elenco di quelli che sono i pattern più utilizzati per progettare applicazioni Cloud. Suddividerò questa sezione in due parti, una prima parte nella quale andrò a proporre due metodi che servono per dare un idea in quali parti suddividere la nostra applicazione e quali ruoli esse debbano svolgere , questi sono il two tier pattern e la sua forma più evoluta il three tier pattern. Fatto ciò passerò ai pattern architetturali.

2.3.1 Two tier pattern

Attraverso questo pattern si vuole suddividere quelle che sono le funzionalità dell' applicazione da quella che è la gestione dei dati, in maniera tale da potergli permettere di scalare in maniera indipendente l' una dall' altra. Di norma infatti le applicazioni vengono suddivise in componenti applicativi che consentono di far scalare indipendentemente funzioni specifiche dell' applicazione. In questa situazione però le funzionalità di gestioni dei dati sono significativamente più difficili da far scalare rispetto ai componenti stateless, poiché i componenti statefull devono coordinare le informazioni relative allo stato fra le varie istanze. Conseguentemente a quanto detto l'

applicazione dovrebbe essere pensata in maniera tale da suddividere le parti più facili da far scalare da quelle che lo sono meno.

Per fare ciò si suddividono le funzionalità dell' applicazione in due parti: la prima di gestione dei dati, che viene fornita mediante uno o più servizi di salvataggio dei dati, la seconda di componenti applicativi che gestiscono la presentazione e la businnes logic. Questa separazione rende in grado i due livelli di scalare indipendentemente in base ai loro livelli di carico. Di seguito viene lasciata un immagine esplicativa di quanto appena enunciato.

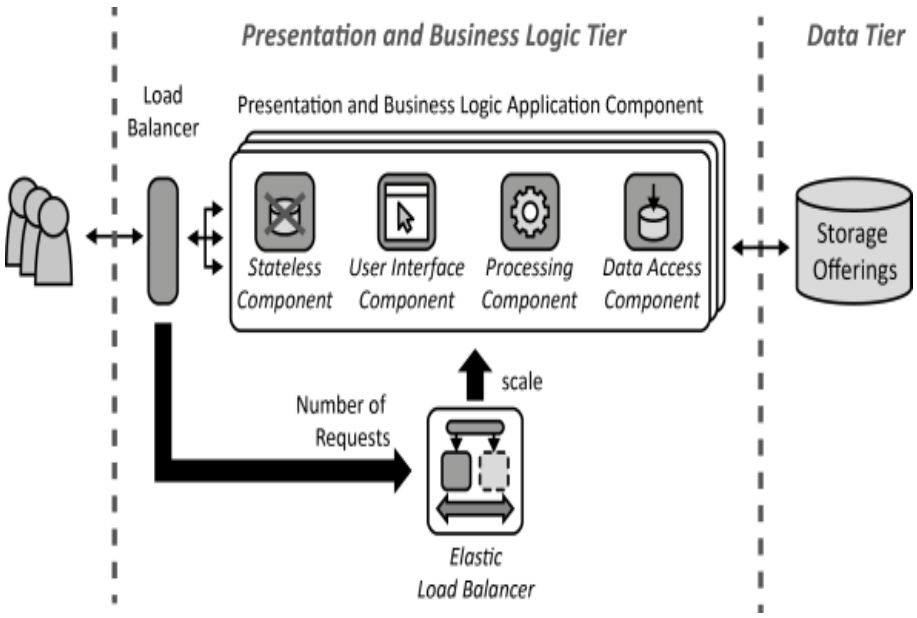


Figura 2.1: Two tier pattern

2.3.2 Three tier pattern

Questo modello è molto simile al two tier pattern solo che in questo caso si va ad introdurre un ulteriore livello, per separare quella che è la presentazione dalla buisness logic. Questo serve per migliorare l' efficienza della nostra applicazione; infatti se i componenti di elaborazione sono computazionalmente più attivi o, più semplicemente, vengono usati più frequentemente

di quelli per l' interfaccia utente, allineare la scalabilità di questi due componenti unendo le loro implementazioni in un solo livello potrebbe essere inefficiente. Questo problema si verifica ogni volta che i componenti hanno carichi di lavoro differenti, impendendo di fornire il giusto numero di istanze ai componenti in presenza di picchi essendo che sono stati pensati livelli con una granularità troppo grossa.

Per risolvere il problema, come già detto, si suddivide l' applicazione in tre livelli: uno che realizza l' interfacciamento con l' utente e che implementa lo Stateless Component Pattern e lo User interface pattern, un secondo che realizza la businnes logic e che implementa lo Stateless Component Pattern in aggiunta al Processing component pattern, infine il terzo livello che è il livello di storage dei dati.

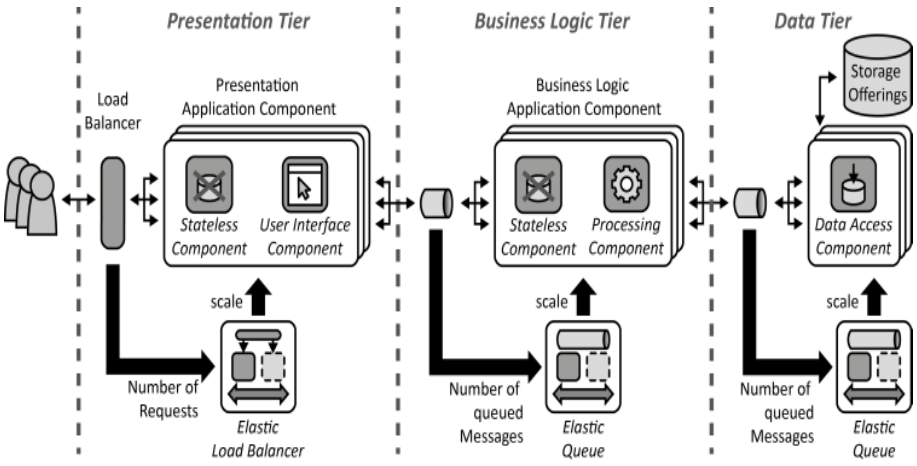


Figura 2.2: Three tier pattern

2.3.3 Stateless Component

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.

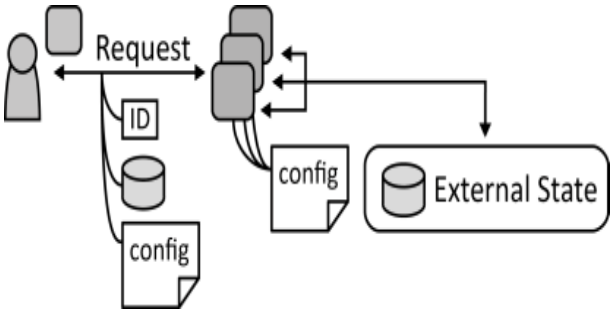


Figura 2.3: Stateless Componet pattern

2.3.4 User Interface component

Attraverso questo pattern si cerca di fornire una soluzione al problema di come mantenere l' interfaccia utente interattiva, ma allo stesso tempo personalizzabile e disaccoppiata dal resto dell' applicazione. Infatti i componenti relativi all' interfaccia con l' utente devono essere aggiunti e rimossi facilmente dall' applicazione senza intaccare l' esperienza dell' utente; di conseguenza le dipendenze con gli altri elementi dell' applicazione dovrebbero essere ridotte il più possibile. Per risolvere il problema si mantiene lo stato esternamente al sistema, come descritto precedentemente nello stateless pattern, e si rilascia la parte di gestione delle richieste sul dispositivo

posseduto dall' utente o in una memoria esterna dalal quale può essere facilmente ricavato. Inoltre i componenti dell' interfaccia utente saranno scalati sulla base delle richieste sincrone ricevute, tramite un elastic balancer.

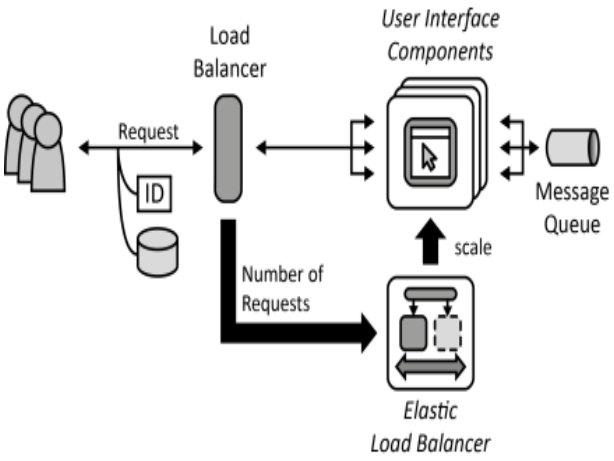


Figura 2.4: User Interface Componet pattern

2.3.5 Processing Component

Mediante questo pattern si cerca di risolvere il problema di come un componente di elaborazione possa essere fatto scalare elasticamente fra le risorse distribuite e allo stesso tempo essere configurabile per quanto riguarda le funzioni supportate per soddisfare le diverse esigenze dei clienti. Inoltre le funzionalità di elaborazione offerte da una applicazione dovrebbero essere gestite da diverse istanze di componenti applicativi che operano in maniera indipendente. Le istanze di questi componenti dovrebbero essere aggiunte e rimosse facilmente dall' applicazione.

Per risolvere il problema si dividono le funzionalità di elaborazione in blocchi funzione separati che vengono assegnati a componenti di elaborazione indipendenti. Ognuno di questi viene fatto scalare in maniera indipendente e implementa lo stateless pattern. Le procedure di scale in e scale out sono gestite da una elastic queue.

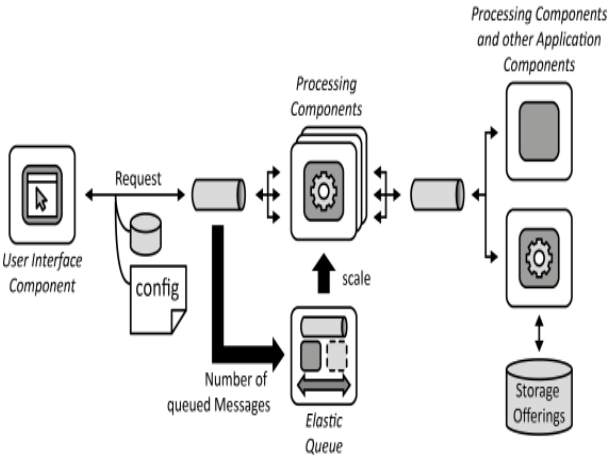


Figura 2.5: Processing Componet pattern

2.3.6 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 message oriented middleware (broker) 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.

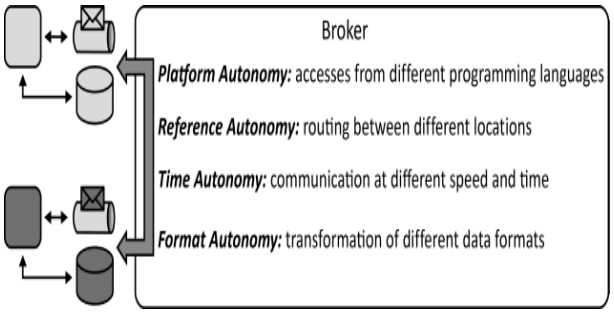


Figura 2.6: Pattern accoppiamento debole

2.3.7 Applicazioni modulari

Le applicazioni monolitiche, ovvero quelle applicazione che riassumono tutte le funzionalità di interazione con l' utente, di processamento e di gestione dei dati in un unico componente e che sono rilasciate su un singolo componente IT, non sono adatte all' ambiente cloud principalmente per due motivi.

- Prima di tutto le risorse disponibili e le performance di una singolo componente IT potrebbero non risultare sufficienti per la nostra applicazione. Questo è particolarmente vero se il provider ci assicura solamente l' environment based avialability; ovvero la disponibilità di tutto l' environment e non delle singole risorse IT contenute in esso. Ad esempio il provider potrebbe garantire solamente il fatto che i clienti possano avere dei server, ma non garantire affatto la disponibilità di server attivi.

- In secondo luogo è il fatto di non essere adatte alla scalabilità. Infatti in questi casi si dovrebbero fornire nuove istanze di tutta l' applicazione e non solo delle singole funzionalità che stanno avendo un carico di lavoro inaspettato; questo rende ovviamente problematiche operazioni di scale out e scale in.

Per risolvere questa problematica si dividono le funzionalità dell' applicazione in componenti indipendenti, ognuno dei quali provvederà a fornire una funzione specifica; in seguito questi elementi verranno integrati per formare l' applicazione distribuita nel suo tutto. Questo ridurre in componenti le varie funzionalità dell' applicazione introduce una decomposizione logica

dell' applicazione. Questi componenti logici verranno poi assegnati, in base alle funzionalità che svolgono, ad uno dei tre tier dei quali abbiamo parlato precedentemente parlato. Questi tier saranno poi rilasciati su diverse risorse IT fornite dal provider. Esistono di tre tipi di decomposizione, ognuno più orientato a certe caratteristiche essi sono :

- **Layer based decomposition**: In questo approccio si suddivide l' applicazioni in livelli logici. Di solito sono quelli che abbiamo introdotto nel three tier pattern ma altri livelli possono essere aggiunti. Ogni livello è composto da componenti fra di loro indipendenti, ma che possono interagire sia con componenti dello stesso livello, sia con componenti di un livello inferiore. Questo impedisce complicate interdipendenze fra i componenti ed incoraggia una descrizione ben definita per l' interfaccia di comunicazione fra i layer.

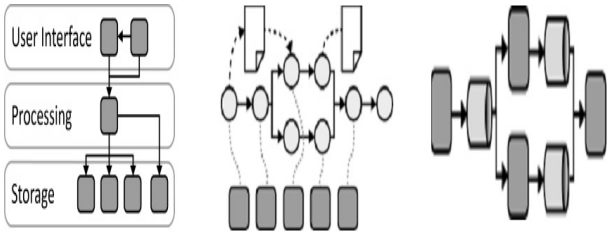
- **Process based decomposition**: Questo processo di decomposizione si concentra sui businnes process supportati dall' applicazione. Questi processi sono formati da una serie di activities che sono eseguite in un ordine specifico. Le activities il loro ordine di esecuzione e i dati gestiti dal processo sono descritti in una serie di file eseguibili. La funzionalità acceduta dalle diverse activities dei processi è decomposta in vari componenti. Le activities interagiscono quindi con questi componenti che costituiscono l' applicazione distribuita.

- **Pipes-and-filters-based decomposition**:Questo modello si concentra sul processamento per il centro dati di una applicazione. Ogni filtro fornisce una certa funzione che è eseguita su dati di input e che produce in uscita un output. I filtri sono interconnessi fra di loro attraverso delle pipe che assicurano che l' output di un filtro sia dato in ingresso ad un altro in una catena di processamento. In una applicazione distribuita i filtri mappano i componenti che forniscono una certa funzione e sono interconnessi utilizzando le comunicazioni offerte dal cloud.

In fondo alla sottosezione lascio una foto che, partendo da destra, riassume in ordine quelle che sono le caratteristiche di ognuna di queste divisioni.

Infine 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.



2.3.8 Componenti Statefull

Con questo pattern si cerca di risolvere il problema di come fare in maniera tale che i componenti della nostra applicazioni che hanno fatto scale out possano mantenere uno stato interno sincronizzato. Per beneficiare di quello che è l' ambiente distribuito di esecuzione cloud, i vari componenti che la compongono sono rilasciati su risorse cloud differenti. L' accoppiamento debole fra questi componenti assicura che questi possano essere istanziati più volte per consentirgli di poter scalare. Alcuni di questi componenti applicativi però possono avere bisogno di mantenere uno stato interno. Questo stato potrebbe essere, per esempio, una lista di oggetti che un utente ha aggiunto al suo carrello in un negozio on-line. Quando un componente ha fatto scale out il problema che insorge è quello che tutte le istanze dovrebbero mantenere lo stesso stato, cosicchè possano avere il medesimo comportamento.

Per risolvere il problema si replica lo stato mantenuto dalle istanze del componente applicativo fra tutte le istanze di quel componente. Le informazioni condivise vengono utilizzate in maniera molto ridotta; per esempio tutte le istanze di un componente stafull potrebbero condividere un file di configurazione allocato in una memoria, oppure la loro configurazione potrebbe essere inviata dall' utilizzatore ad ogni richiesta. Lo sviluppatore

si trova in questo caso a dover affrontare il problema di quando replicare o meno lo stato per poter garantire un adeguato livello di consistenza di quest' ultimo. Infine ogni richiesta di un cliente deve essere identificata da un identificatore, che servirà al componente statefull per recuperare in maniera corretta i dati associati ad una particolare richiesta. Di norma si tende ad aggiornare lo stato dopo ogni operazione di manipolazione sui dati che riguarda un componente statefull. Di seguito una foto che mostra il comportamento di questi componenti.

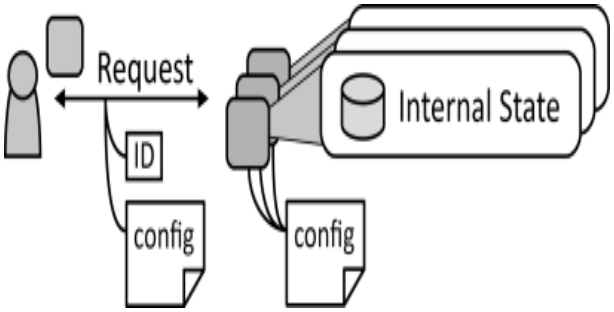


Figura 2.7: Pattern componenti statefull

2.3.9 Data acess component

Mediante questo pattern si cerca di fornire un meccanismo per nascondere e isolare quella che è la complessità delle operazioni di accesso ai dati, dovuta ai protocolli di accesso e alla consistenza dei dati, e allo stesso tempo garantire la configurabilità della struttura dati.

Un applicazione distribuita può utilizzare diversi metodi per il salvataggio dei dati come ad esempio database relazionali o key value storages. Alternativamente queste applicazioni potrebbero anche utilizzare componenti statefull sviluppati individualmente. Occuparsi delle complessità relative a questi dati, come per esempio la gestione delle autorizzazioni, le query su di essi o la gestione dei fallimenti a livello di interfaccia utente o a livello di componenti di elaborazione accoppia strettamente questi componenti alla maniera che si è scelta per salvare i dati, rendendone più difficile l' implementazione. Inoltre, un cambiamento nel metodo di salvataggio utilizzato comporterebbe anche il dover cambiare il componente nel quale esso veniva

gestito. Contrariamente a quanto detto invece bisognerebbe che i diversi modi di accedere ai dati fossero integrati per fornire un accesso ai dati uniforme agli altri componenti dell' applicazione. Inoltre i dati potrebbero essere salvati su cloud di provider differenti, situazione anche essa che va gestita.

Per risolvere il problema l' accesso a fonti di dati differenti viene integrato da un data access component. Questo componente coordina la manipolazione dei dati se vengono utilizzati servizi di storage differenti. Nel caso uno di questi servizi debba essere cambiato o cambi l' interfaccia di un servizio di salvataggio dati di un cloud provider l'unico componente che deve essere cambiato è il data access component; questo assicura l' accoppiamento debole fra i dati e il resto dell' applicazione.

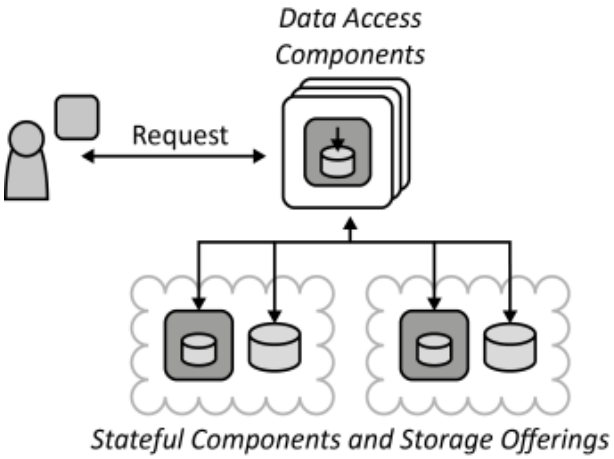


Figura 2.8: Pattern componenti statefull

2.3.10 Shared Component

Tramite questo pattern si cerca di risolvere il problema di come un componente applicativo possa essere condiviso fra più tenant pur mantenendo caratteristiche di configurabilità.

In una applicazione Cloud di norma i vari tenant condividono le risorse di una macchina in base a quante ne richiede l' applicazione che stanno

correntemente utilizzando; ma le performance di un componente applicativo possono essere migliorate se non solo si condividono le risorse IT, ma anche le istanze del componente stesso. La condivisione di componenti è in particolare consigliabile qualora si abbia un componente che offre la stessa funzione a tutti gli utenti. Se i tenant potessero condividere la stessa istanza di questo componente applicativo le risorse sottostanti sarebbero usate più efficientemente e si eviterebbe una forma di ridondanza nel deployment.

La proprietà principale di uno shared component è che esso fornisca la stessa funzionalità a tutti i tenant che vi accedono. In particolare, se il componente fornisce solo dati ai tenant ma non salva i dati specifici di ognuno, tutti i tenant potrebbero essere trattati come un gruppo di utenti uniformi ai quali viene garantita la stessa user experience e livello di servizio. Lo shared component gestisce quindi le richieste di tutti i tenant, come spiegato nella figura che verrà messa alla fine della sezione. Questi componenti si comportano allo stesso modo per tutti i tenant, ma mostrano, processano e salvano i dati di utenti specifici. Di conseguenza, i corrispondenti user interface component, processing component o data access component, sono configurati allo stesso modo per tutti i tenant e potrebbero non essere consapevoli del fatto che gestiscono carichi di lavoro provenienti da utenti differenti. Solo configurazioni minori del comportamento dei componenti potrebbero essere passate all' istanza del componente applicativo con ogni richiesta, per esempio per specificare diverse risoluzioni del display specificate dall' interfaccia utente.

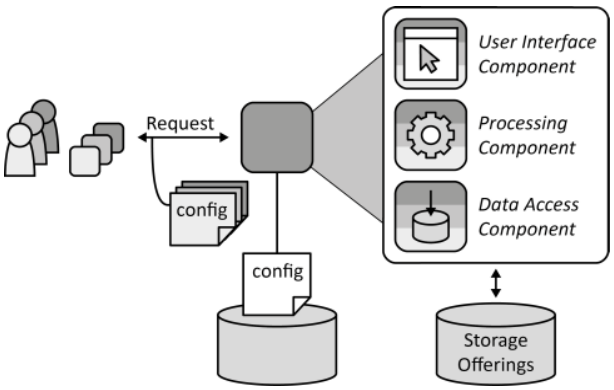


Figura 2.9: Shared Component pattern

2.3.11 Tenant Isolated Component

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.

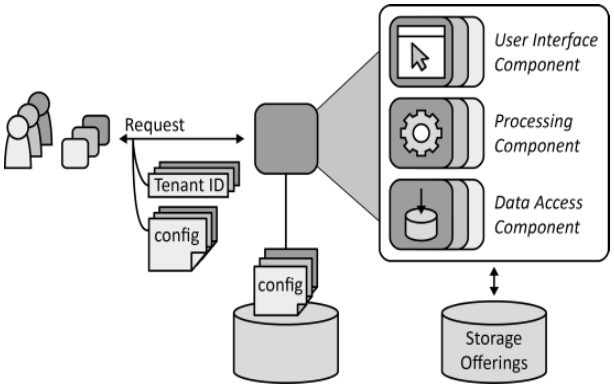


Figura 2.10: Tenant Isolated Component pattern

Capitolo 3

Tecnologie per lo sviluppo di applicazioni cloud

Nelle seguenti sezioni andrò ad analizzare quelle che sono i principali framework, PaaS e middleware che consentono di sviluppare applicazioni Cloud. In quest' ottica cercherò di dare una visione il più possibile chiara di quelle che sono le caratteristiche fondamentali degli strumenti che analizzerò. In primis verrà presentato Apprenda, che è una PaaS tra le più utilizzate nell' anno 2013. Subito dopo presenterò invece Microsoft Windows Azure, la piattaforma Cloud di Microsoft. Questa viene introdotta perché risulterà utile per capire quello che sarà l'esempio di applicazione che farò alla fine utilizzando il Framework Orleans. In seguito presenterò Cloudify, un middleware che si interpone fra la PaaS scelta e la nostra applicazione. Infine parlerò, per l'appunto, di Orleans che è un framework introdotto da Microsoft per lo sviluppo di applicazioni Cloud, e qui mi soffermerò anche a parlare del modello di programmazione ad attori, essendo che Orleans si basa su di esso.

Cercando di capire quali sono le differenze che sussistono fra una PaaS, un middleware e un framework. La prima abbiamo già spiegato all' inizio della tesi di che cosa si tratta per ciò non ci spenderei ulteriori parole. Un middleware è invece un programma che si interpone fra il nostro programma e l' infrastruttura sottostante, consentendoci di scrivere la nostra applicazione basandoci sui servizi forniti dallo stesso, rendendoci così molto più platform-independet e fornendoci un livello superiore di astrazione. Infine per framework si intende una serie di librerie e funzioni che permettono di

risolvere problemi ricorrenti e di conseguenza di progettare applicazioni in maniera più semplice e veloce qualora lo si utilizzi. Su questi ultimi va però precisato che essi risolvono specifici problemi e non è detto quindi che sia sempre utile farne uso, bisogna vedere se si adattano a ciò che noi stiamo cercando.

3.1 Caretteristiche comuni ai PaaS

Tutte le PaaS offrono più o meno le stesse caratteristiche, anche se magari vengono poste in modo diverso. Questo fa si che la scelta di una PaaS per la propria azienda a discapito di un altra venga fatta più che altro in termini di facilità di utilizzo dei servizi che essa offre, compatibilità con il software già esistente o tipologie di linguaggi supportati. Ho scelto quindi di approfondire apprenda, a discapito di altre piattaforme, come ad esempio google app engine, per la sua predisposizione ad applicazioni java e .NET, che sono le due a me più famigliari. Inoltre apprenda sta continuando a venire sempre più utilizzata all' interno di aziende, quindi mi è sembrato opportuno focalizzarmi su di essa piuttosto che su altre PaaS magari già mainstream e che di conseguenza avranno già avuto i loro approfondimenti all' interno di altre tesi.

Detto questo passerò adesso ad elencare in maniera breve quelle che sono le caratteristiche che tutte le PaaS che si vogliono definire tali dovrebbero fornire :

- Servizi per sviluppare, testare, implementare e mantenere applicazioni host nello stesso ambiente di sviluppo integrato.
- Strumenti per la creazione di interfaccia utente basata su web.
- Un' architettura multi-tenant.
- Integrazione con i servizi web e i database
- Supporto per lo sviluppo della collaborazione di squadra
- Strumenti per il controllo delle applicazioni

3.2 Apprenda

Apprenda è una piattaforma PaaS per le aziende che è in grado di mandare in esecuzione applicazioni .NET e java. In particolare, Apprenda è una PaaS che può essere inserita come livello al di sopra di una qualsiasi infrastruttura che abbia come sistema operativo istanze di Linux o Windows. Le applicazioni che Apprenda accetta come applicazioni ospite, oltre a dover essere scritte in uno dei due linguaggi sopracitati, possono essere composte di servizi web, data base (e.g. SQL Server, Oracle) e front ends. Le caratteristiche che rendono Apprenda un' ottima Paas sono le seguenti :

- E' focalizzato sui casi d' uso .NET su Windows e Java su Linux, con una facile integrazione di sistemi IT legacy e la capacita di rendere più agevoli processi aziendali tradizionalmente critici, come ad esempio il cablaggio dell' autenticazione nelle applicazioni.
- Fornisce un modello lock-in-free, assicurando la libertà di scelta su quale delle infrastruttura utilizzare. Apprenda supporta anche la federazione delle infrastrutture, banalizzando le architetture infrastrutturali come il multi-datacenter o l' hybrid cloud.
- Consente di poter cambiare poco o niente del codice di una applicazione esistente per poterla mandare in esecuzione sulla Apprenda Paas.
- Ha una pletora di API e framework per attingere ai servizi che la piattaforma offre.
- Può essere stratificato su infrastrutture esistenti, consentendo un continuo sfruttamento degli investimenti esistenti.

Detto questo passerò adesso a enunciare nel dettaglio quali sono tutte le funzionalità che Apprenda offre e che hanno sicuramente contribuito a farla diventare una delle migliori PaaS che ci sono sul mercato.

3.2.1 Service Broker

Apprenda rileva automaticamente le dipendenze dei servizi all'interno dell'applicazione. Utilizzando queste informazioni apprenda collegherà dinamicamente le richieste dei clienti ai servizi di SOAP e REST appropriati,

gestendo ogni aspetto, dal rilevamento automatico all' orchestrazione della chiamata. Le applicazioni multi-tier in stile SOA possono essere difficili da gestire su larga scala. I client hanno bisogno di una istanza di servizio che, a run time, sia in grado di gestire quelle che sono le loro richieste. Inoltre, all' evolvere dell' applicazione in nuove versioni, assicurare una versione consistente in accordo con le richieste del cliente potrebbe risultare problematico.

Apprenda rileva automaticamente i confini del servizio e le dipendenze che fanno parte dell' applicazione. I servizi SOAP e REST vengono automaticamente portati alla giusta versione e registrati nel registro Apprenda. Il Service Broker di Apprenda, assicura che le richieste a runtime dei client vengano risolte in modo dinamico, e che le chiamate vengano orchestrate al servizio target appropriato. Essendo strettamente integrato con la distribuzione del carico di Apprenda e il sottosistema HA, se le istanze del servizio corrispondenti alle richieste dei client non fossero disponibili, Apprenda distribuirebbe automaticamente le istanze per garantirne la disponibilità. Grazie alla capacità di Apprenda di risolvere automaticamente i servizi e orchestrare le chiamate costruire applicazioni scalabili e dinamiche in stile SOA è molto più facile per il programmattore, poiché parte del lavoro viene fatto dalla piattaforma stessa.

3.2.2 API based transaction metering

Le metering APIs di apprenda consentono agli sviluppatori di definire i contatori delle transazioni, i contatori delle quantità di blocco, caratteristiche di toggle, e di imporre dei limiti per poter controllare la logica dell'applicazione. Ciò significa che l'accesso dell'utente finale alle funzionalità dell' applicazione può essere misurato e controllato.

I requisiti applicativi spesso richiedono dei livelli di accesso che garantiscano ad utenti differenti un diverso insieme di caratteristiche. In aggiunta a questo l' utilizzo delle funzionalità dell' applicazione da parte di un utente deve essere monitorato e, se necessario, ridotto in base a quello che è il ruolo dell' utente e alle risorse correnti. Un esempio potrebbe essere quello in cui una applicazione possa servire sia utenti paganti che utenti con una licenza free, dove questi ultimi hanno a disposizione, giustamente, un numero di funzionalità inferiori rispetto ai primi. Consentire questa sorta di controllo in un applicazione richiederebbe che essa tenesse traccia delle transazioni e

fornisse uno strumento configurabile per il controllo delle quantità di risorse utilizzate.

Per risolvere questo problema in Apprenda gli sviluppatori marcano nel codice sorgente le API con dei meta-dati che definiscono cosa un utente è autorizzato a fare all' interno di quella applicazione. In fase di esecuzione, Apprenda valuta i diritti nei confronti delle chiamate a tali API per determinare ciò che un utente finale è o non è autorizzato a fare. Grazie a questi meccanismi gli sviluppatori possono gestire il tracking delle risorse e le problematiche di accesso relative a diversi utenti senza bisogno di dover inventare nulla, in quanto è già tutto compreso a livello di API.

3.2.3 Cache distribuita

Apprenda fornisce una cache distribuita che suddivide il lavoro in modo automatico fra server diversi in maniera ottimizzata per i PaaS, inoltre fornisce una visione di questa cache come un unico insieme agli sviluppatori, grazie ad una serie di API facili da utilizzare.

Il salvataggio dello stato nelle applicazioni Cloud è una scelta architetturale importante. Tipicamente queste applicazioni richiedono che non sia salvato nessuno stato nella memoria standard poiché questo metterebbe a repentaglio la capacità di una applicazione di essere scalabile ed elastica. Questo non significa che gli sviluppatori sono lasciati senza scelta. Apprenda fornisce una cache distribuita, che è nativa alla piattaforma, che permette agli sviluppatori di salvare dati e stato dei componenti in maniera tale che siano accessibili all' applicazione senza doversi preoccupare di dove essa sia in esecuzione sulla piattaforma. Questo assicura che le applicazioni possano archiviare e recuperare lo stato, consentendo ai dati di essere persistenti evitando l' insidia, tipica delle architetture cloud, della memorizzazione dello stato nella memoria dell' applicazione.

3.2.4 Coda multi-tenant

Il supporto On-platform per la coda multi-tenant di Apprenda assicura che gli sviluppatori aziendali possono sfruttare modelli di messaggistica asincrona attraverso le loro code preferite senza doversi preoccupare dell' effettiva implementazione a livello di sistema.

Le applicazioni moderne difficilmente sono in esecuzione su un singolo server e di norma richiedono di poter scalare in base a quelle che sono le loro necessità; tutto ciò rende lo scambio di messaggi asincrono una parte fondamentale dello sviluppo del nostro software. Apprenda consente agli operatori delle piattaforme di configurare un' implementazione di coda a livello di sistema e agli sviluppatori di utilizzare modelli di messaggi asincroni attraverso un livello di astrazione comune, che si basa su quella che è l'implementazione specifica della coda. Inoltre, le code sulla piattaforma sono ottimizzate per le applicazioni multi-tenant, rendendo in questo modo banale affrontare problemi relativi al multi-tenancy quando si utilizzano modelli di chiamate asincrone.

3.2.5 Catalogo dei servizi

La piattaforma applicativa di Apprenda rende la condivisione di servizi in tutta l'azienda semplice, veloce e sicuro. Consentendo la creazione di un archivio centralizzato e ad elenco dei servizi disponibili, gli sviluppatori di tutta l'organizzazione possono sfruttare il lavoro altrui per migliorare il time to market delle applicazioni.

Uno dei vantaggi della PaaS di Apprenda è la possibilità di condividere applicazioni e servizi in tutta l'azienda. Apprenda consente agli amministratori della piattaforma di curare i servizi in un catalogo di servizio centralizzato. L'amministratore può impostare politiche e controlli di utilizzo, mentre gli sviluppatori ottengono un modo semplice per accedere ai servizi e alle capacità comuni dell' azienda.

Una volta costruita la lista dei servizi costruire una applicazione non diventa niente altro che una operazione di assemblaggio per gli sviluppatori. Questo ovviamente consente di ridurre notevolmente i tempi di sviluppo e allo stesso tempo di ampliare la riusabilità dei servizi. Nell' immagine un esempio di catalogo dei servizi.

Figura 3.1: Catalogo dei servizi apprenda

3.2.6 End User Onboarding/Provisioning

Apprenda fornisce un potente servizio di provisionig self-service che può essere strettamente integrato con i servizi o la directory di autenticazione dell' azienda.

Il sistema di provisioning Apprenda permette ai nuovi utenti finali di fornire informazioni all' atto della registrazione, che in seguito verranno utilizzate per allocare le risorse, distribuire i componenti applicativi necessari, e concedere loro un accesso immediato alle applicazioni dando, in sostanza, la possibilità di attivare un servizio per i clienti con nessuna codifica da parte degli sviluppatori. Questo consente un on-boarding degli utenti finali a basso costo e rimuove gli eventuali ostacoli che potrebbero scoraggiarne l'ingresso di nuovi.

Questa serie di servizi offerti dalla piattaforma Apprenda riducono il numero di errori dovuto ad un provisioning manuale dei nuovi servizi. Inoltre questo processo automatizzato non solo riduce i costi, ma consente anche di fornire i nuovi servizi in maniera più fluida e veloce, favorendo la crescita dell' azienda.

3.2.7 Definizione delle autorizzazioni

Apprenda fornisce un Entitlment Engine all' interno dell' infrastruttura attraverso la quale si possono marcare i vari servizi con un livello di accesso attraverso delle chiamate a delle API. Questi diritti individuali possono essere

raggruppati in degli insiemi di autorizzazioni che istruiscono la piattaforma su quali caratteristiche concedere a ogni utente, permettendo esperienze applicative specifiche per ogni categoria.

Il Multi-tenancy è una architettura applicativa che consente agli utenti raggruppati in unità organizzative di accedere a una singola istanza dell' applicazione condivisa attraverso le unità organizzative, dando a ciascun utente un'esperienza univoca dell' applicazione. Alcuni utenti potrebbero aver bisogno di accedere ad alcune caratteristiche dell' applicazione che ad altri non servono, come ad esempio un amministratore che necessita del controllo totale sull' applicazione mentre gli utenti normali no. L' Entitlment Engine fornisce delle API che rendono facile la marcatura di parte di codice come riservato ad utenti con certi privilegi, questi vengono settati attraverso dei meta-dati specifici per ogni gruppo che rendono facile sapere quali sono i servizi ai quali quel determinato gruppo può accedere. Di seguito un immagine che mostra quanto sia facile in apprenda aggiungere nuovi gruppi e settarne le rispettive operazioni concesse (l' interfaccia è relativa alla versione precedente ma i concetti sono quelli).

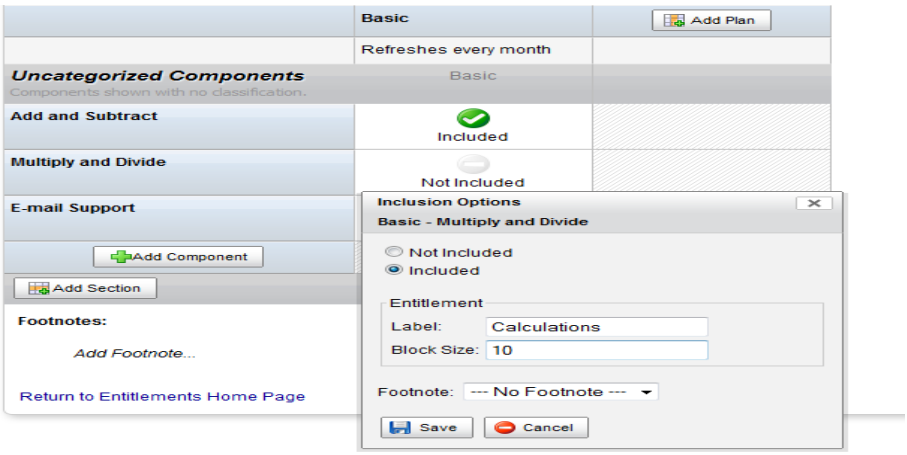


Figura 3.2: Catalogo dei servizi apprenda

3.2.8 Politiche per il deployment dell' applicazione

Le politiche di Apprenda per il deployment delle applicazioni consentono di controllare come le applicazioni vengano armonizzate con l' infrastruttura in relazione ai vari clouds.

Mentre Apprenda consente alle aziende di gestire un gran numero di server come una singola risorsa logica, ci sono spesso situazioni in cui le applicazioni devono essere mappate su infrastrutture specifiche. In molti casi ciò è dovuto alla sicurezza o a requisiti di conformità e ottimizzazione delle risorse. Le politiche per il deployment di Apprenda consentono un mapping delle applicazioni o di componenti di esse verso le infrastrutture basandosi su proprietà specifiche che possono essere configurate.

3.2.9 Inventario delle applicazioni

Apprenda fornisce un unica piattaforma per mandare in esecuzione e gestire la totalità delle applicazioni che sono presenti all' interno di una azienda. Questo assicura che quest' ultima sarà sempre in grado di conoscere di quali componenti è composta una particolare applicazione e di dove essi siano correntemente in esecuzione.

La maggior parte delle aziende hanno centinaia di applicazioni diverse che cambiano significativamente nel corso del tempo. Tenere traccia di tutte queste applicazioni e sapere in ogni momento dove i componenti di ciascuna risiedono è una operazione che può risultare parecchio complessa. Per risolvere questo problema la PaaS di Apprenda fornisce un servizio di inventario real-time che tiene costantemente traccia di dove tutti i componenti risiedono all' interno dell' infrastruttura, senza dover costringere i programmatori a mantenere modelli complicati o a sviluppare tecniche di scoperta. Questo semplifica notevolmente la condivisione, il sostegno, il controllo, la manutenzione e la gestione del portfolio applicativo. Nell' immagine un esempio di Portfolio delle applicazioni.

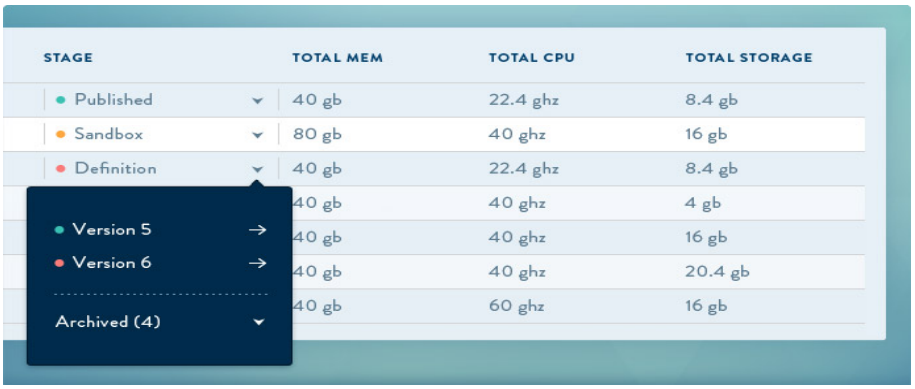


Figura 3.3: Inventario delle applicazioni apprenda

3.2.10 Gestione del ciclo di vita di una applicazione

Apprenda gestisce il deployment di tutte le applicazioni attraverso un processo di controllo della versione fornendo un modo facile e veloce per applicare patch che apportino piccoli cambiamenti, sistemino bug o per fare dei veri e propri aggiornamenti di versione. In questa maniera si potranno effettuare aggiornamenti alle applicazioni ogni volta che lo si desidera essendo che la piattaforma garantisce che la patch venga applicata attraverso tutta la rete, senza preoccuparsi del fatto che i componenti siano tutti allocati su uno stesso server oppure distribuiti su più macchine. Il rilascio delle applicazioni diventa di conseguenza facile quanto cliccare pochi pulsanti e decidere dove fare il deploy.

Apprenda fornisce inoltra la possibilità di definire politiche specifiche per ogni applicazione, che consentono di decidere dove effettivamente verranno rilasciati i componenti all' interno del nostro environment. Per esempio si può scegliere che un certo componente venga rilasciato sulla piattaforma hardware più performante oppure solo su un certo tipo di cloud pubblico.

Questi meccanismi permettono di risparmiare molto tempo per quel che riguarda il deploy delle applicazioni. Infatti se questa operazione dovesse venire effettuata manualmente ad ogni patch, il numero di questi aggiornamenti verrebbe sicuramente ridotto perché troppo costosi e lunghi. Questo ridurrebbe notevolmente quella che è la competitività di un azienda sul mercato poiché i clienti vogliono sempre nuove caratteristiche e funziona-

lità. Grazie ad Apprenda invece tutte queste operazioni vengono fatte in automatico consentendo di poter effettuare un aggiornamento ogni volta che lo si vuole. Di seguito l' immagine di una schermata che mostra l' utilizzo delle risorse di una applicazione e di come esse possano essere gestite a run time.

Figura 3.4: Visualizzazione info sulle applicazioni in apprenda

3.2.11 Politiche per le risorse

Le politiche per la gestione delle risorse danno la possibilità agli operatori di piattaforma di usare dei meta-dati per pubblicare delle configurazioni di CPU, RAM e spazio di salvataggio dati che poi i programmatori potranno associare alle loro applicazioni. Apprenda allocherà poi le risorse in base a quello che i programmatori hanno scelto, dividendo l' infrastruttura sulla base di queste politiche.

La PaaS di Apprenda unisce sotto un fabbrica di host logica qualsiasi numero di istanze (fisiche o virtuali) di sistemi operativi Windows e Linux. Piuttosto che basarsi sulla macchina virtuale come strumento per garantire l' isolamento fra le applicazioni Apprenda utilizza un modello di contenitore custom che viene mandato in esecuzione all' interno delle istanze dei sistemi operativi per isolare le applicazioni l' una dall' altra. Questo tipo di isolamento viene utilizzato per suddividere ogni istanza del sistema operativo in segmenti multipli, incrementando l' utilizzo globale delle risorse e riducendo i costi dell' hardware e delle licenze.

Grazie a questi strumenti gli operatori della piattaforma hanno il controllo totale su come le risorse vengono utilizzate e allo stesso tempo gli sviluppatori hanno la possibilità di prendere decisioni sulle risorse a livello di applicazione, riducendo i costi attraverso una maggiore conformità e utilizzo delle risorse.

Figura 3.5: Gestione delle risorse in apprenda

3.2.12 Servizi per applicazioni cloud estendibili

Apprenda offre con la sua PaaS un modello per aggiungere servizi alla piattaforma in maniera dinamica, come ad esempio servizi per lo storage NoSQL o code; questi servizi potranno poi essere gestiti, configurati e messi a disposizione degli sviluppatori che li potranno utilizzare nelle proprie applicazioni.

Nessuna piattaforma applicativa potrebbe nutrire l' idea di fornire ogni possibile funzionalità necessaria agli sviluppatori. Una piattaforma adeguata permette di estendere le funzionalità che questa offre personalizzando quelle già esistenti o dando la possibilità di installare servizi creati da terze parti, che poi possano in seguito essere facilmente utilizzati dagli sviluppatori e dalle loro applicazioni.

Il modello add-on di Apprenda rende banale estendere la piattaforma con nuove funzionalità e servizi. Gli sviluppatori possono scegliere un servizio

dalla lista di quelli correntemente attivi e collegare le loro applicazioni a questi. Apprenda metterà poi automaticamente a disposizione le risorse del add-on quando necessario, legando la risorsa e l'accesso a tale risorsa al ciclo di vita dell'applicazione dello sviluppatore.

3.2.13 Rest API e interfaccia a linea di comando

Apprenda fornisce agli sviluppatori delle potenti REST API e un interfaccia a linea di comando chiamata Apprenda Cloud Shell (ACS). ACS e l'API possono essere sfruttate per personalizzare l'esperienza degli sviluppatori nel distribuire e gestire le applicazioni ospiti sulla piattaforma Apprenda.

Mentre il Developer Apprenda Portal fornisce un'interfaccia utente per gli sviluppatori che mostra la distribuzione e la gestione delle applicazioni in modo semplice, le esigenze di automazione e la convenienza richiedono delle API e una shell che espongano la stessa funzionalità. Grazie a questi strumenti è possibile:

- connettere e gestire istanze diverse di Apprenda tramite l' ACS
- costruire interfacce grafiche personalizzate utilizzando le REST API
- scrivere script basati sull' ACS per lo sviluppo automatico
- integrare la gestione dei flussi di lavoro di Apprenda con sistemi scelti dagli sviluppatori

3.2.14 Capacità di supportare cloud ibridi

Apprenda consente di posizionare i server su cloud diversi e di creare politiche per decidere dove mettere le applicazioni e di conseguenza dove esse consumano risorse.

La maggior parte delle imprese hanno molteplici ambienti in cui eseguire e gestire le loro applicazioni. Tipicamente questo include diverse aree geografiche insieme a combinazioni di risorse private, pubbliche o ospitate. Spesso poi dividono ciascuno di questi cloud per caso d'uso come ad esempio uno per il testing, uno per lo staging e un altro per la produzione. Apprenda consente ai clienti di combinare le risorse provenienti da tutti questi clouds in un unico pool di risorse logiche e applicare criteri di deployment delle

applicazioni flessibili e potenti. Queste politiche possono essere utilizzate per mappare automaticamente le applicazioni sulle infrastrutture basandosi su molteplici fattori tra cui casi d'uso, sicurezza, conformità,geografia ecc.

In poche parole le funzionalità per i cloud ibridi di Apprenda consentono di mappare in modo efficiente e flessibile le risorse sia di cloud pubblici che privati sul portfolio delle applicazioni dell' azienda.

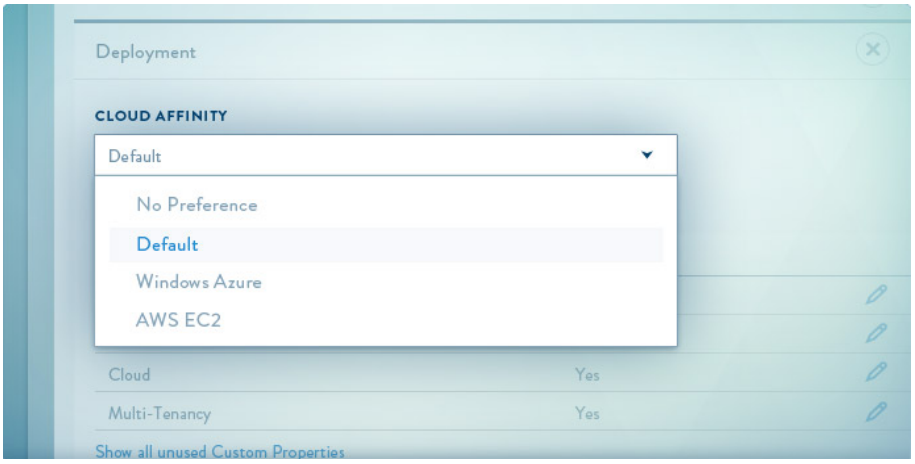


Figura 3.6: Gestione delle affinità con la piattaforma in apprenda

3.3 Microsoft Windows Azure

Microsoft Windows Azure (MWA) è la soluzione che Microsoft fornisce per i servizi relativi al cloud computing. Lanciata nel 2010, è una piattaforma che essenzialmente fornisce le risorse per eseguire delle applicazioni e per salvare dati. Oltre a questo però, essa fornisce anche una serie di servizi ulteriori che assistono l' utilizzatore finale nel rilascio delle sue applicazioni, dandogli la garanzia di rilasciare (qualora sia stata ben progettata) un applicazione facilmente scalabile e facilmente gestibile. MWA adotta sia il modello di tipo IaaS che di tipo PaaS. I tre componenti principali di quest' ultima vengono illustrati in figura 3.7 e sono compute, storage e fabric controller.

Le tecnologie e le potenzialità che la MWA offre possono essere riassunte come segue. All' inizio i servizi di computazione forniscono la capacità di

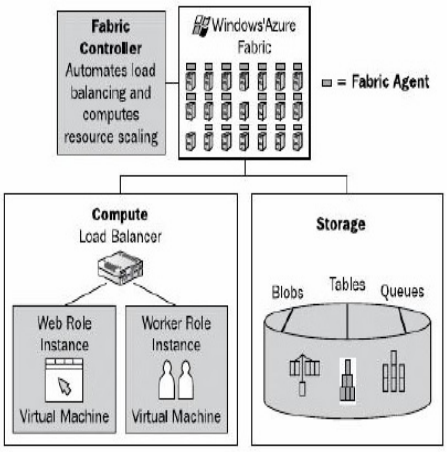


Figura 3.7: Componenti base Windows Azure

elaborazione per poter eseguire le applicazioni che stanno su diverse macchine virtuali, ognuna delle quali ha in esecuzione il proprio sistema operativo scelto precedentemente dallo sviluppatore (le macchine virtuali stesse sono però in esecuzione sul Windows Azure OS). In seguito i servizi di storage forniscono lo spazio necessario, su server di proprietà della Microsoft, per salvare le informazioni necessarie. Infine il Fabric Controllor gestirà l' interazione fra i due componenti precedentemente citati, in quanto elemento predisposto a farlo.

3.3.1 Cloud Services

I Cloud Services sono quei componenti che forniscono la potenza di elaborazione per poter eseguire l' applicazione e che ne gestisce allo stesso tempo anche il codice. In MWA le applicazioni sono suddivise in 3 componenti, chiamati ruoli, ed essi sono i ruoli Web, i ruoli Worker e i ruoli VM.

- **Ruoli Web** Rappresentano l' interfaccia dell' applicazione con la quale l' utente può interagire. Il loro scopo è quello di rendere più facile la creazione di applicazioni Web-based. Ogni istanza di servizio web ha già configurato dentro di esso un IIS (internet information service), di conseguenza creare applicazioni usando ASP.NET , Windows communication foundation (WCF) o altre tecnologie web risulta semplice.

Gli sviluppatori possono inoltre creare applicazioni in codice nativo; non è richiesto l' utilizzo del framework .NET. Questo significa che è possibile installare e mandare in esecuzione tecnologie non Microsoft come PHP e Java.

- **Ruoli Worker** Essi gestiscono quella parte dell' applicazione che viene mandata in esecuzione in background e che non è visibile all' utente finale. La differenza maggiore fra questi e i ruoli web è che non possiedono al loro interno un IIS configurato, quindi il codice che mandano in esecuzione non viene gestito da un IIS; un ruolo worker potrebbe gestire una simulazione o gestire il processamento di un video. E' per questo motivo che l' interazione con l' utente avviene mediante i ruoli web e la computazione viene spostata sui worker. Anche in questo caso gli sviluppatori sono liberi di poter utilizzare sia il framework .NET che tecnologie non Microsoft per il rilascio di questo tipo di ruoli.

- **Ruoli VM** Sono i ruoli che si occupano delle macchine virtuali. Fra le altre cose possono essere utili per il passaggio di una applicazione in esecuzione su un server Microsoft verso una piattaforma Windows Azure.

La figura 3.8 mostra i vari tipi di ruoli di cui una applicazione può essere composta e come viene gestita la comunicazione fra i ruoli e la piattaforma MWA.

Per rilasciare un applicazione su Windows Azure uno sviluppatore può utilizzare il portale che la piattaforma fornisce. Oltre all' applicazione stessa, esso invia anche delle informazioni di configurazione alla piattaforma, che la informano di quante istanze vuole che siano mandate in esecuzione per ogni ruolo. A questo punto il Fabric Controller crea una macchina virtuale(VM) per ogni istanza e vi manda in esecuzione il codice del ruolo scelto per quella VM.

Come anche la figura mostra, le richieste fatte dall' applicazione dell' utente possono essere inviate utilizzando i protocolli HTTP, HTTPS o TCP. Comunque sia, all' arrivo, per ogni richiesta viene fatto una operazione di distribuzione del carico (load-balancing) automatica, che tiene in considerazione tutte le istanze del ruolo al quale si riferisce. Il componente che si occupa di fare questa operazione viene definito load balancer. poiché quest' ultimo non consente la creazione di affinità con una particolare istanza di

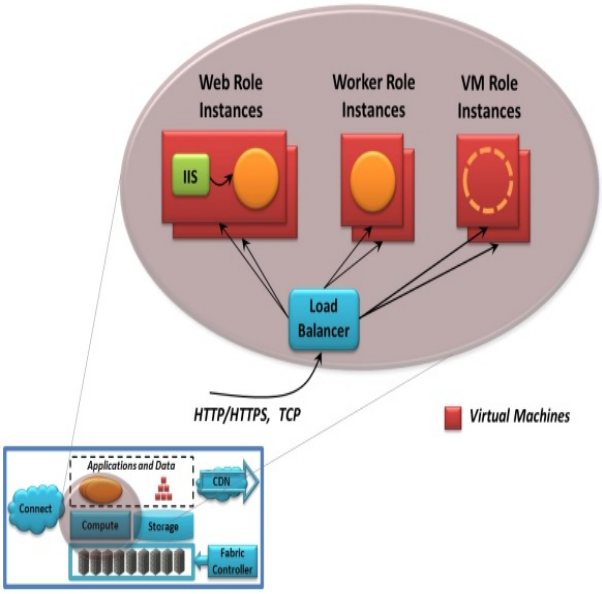


Figura 3.8: Gestione dei vari tipi di rouli in Windows Azure

un ruolo, non c'è modo di garantire che richieste successive inviate dallo stesso utente vengano gestite dalla stessa istanza di un ruolo. Questo significa che le istanze dei ruoli non dovrebbero mantenere lo stato solo al loro interno durante le richieste, ma che ogni stato che possa essere influenzato in qualche modo dalle operazioni del utente debba essere salvato nei servizi di storage forniti da Windows Azure o in qualche servizio esterno.

Uno sviluppatore può usare qualsiasi combinazione di istanze di ruoli Web, ruoli Worker e ruoli VM per creare una applicazione Windows Azure. Se il carico sull' applicazione dovesse crescere, si può utilizzare il portale MWA per richiedere più istanze di un qualsiasi ruolo che faccia parte dell' applicazione. D' altro canto, se il carico cala, si può sempre ridurre il numero di istanze in esecuzione. Windows Azure mette a disposizione delle API che consentono che queste cose possano venire fatte in maniera programmatica (cambiare il numero di istanze non richiederà di conseguenza l' intervento manuale), ma la piattaforma non scala automaticamente le applicazioni in base al loro carico.

Per consentire il monitoraggio e il debug di applicazioni Windows Azure, ogni istanza può chiamare un'API di registrazione che scrive le informazioni in un registro comune a livello di applicazione. Uno sviluppatore può anche configurare il sistema per raccogliere i contatori delle prestazioni di un'applicazione, misurare il suo utilizzo della CPU, salvare dei crash dump in caso di fallimento della stessa, e molto altro. Queste informazioni sono conservate in uno spazio di archiviazione di Windows Azure e uno sviluppatore è libero di scrivere codice per esaminarlo. Ad esempio, se una istanza di un ruolo si blocca per tre volte nel giro di un'ora, del codice scritto appositamente dal programmatore potrebbe inviare un'e-mail all'amministratore dell'applicazione.

3.3.2 Storage

Le applicazioni lavorano con i dati in molti modi differenti; in accordo con questo Azure fornisce diverse opzioni; la figura 3.9 mostra quelle che sono le scelte che ci vengono messe a disposizione.

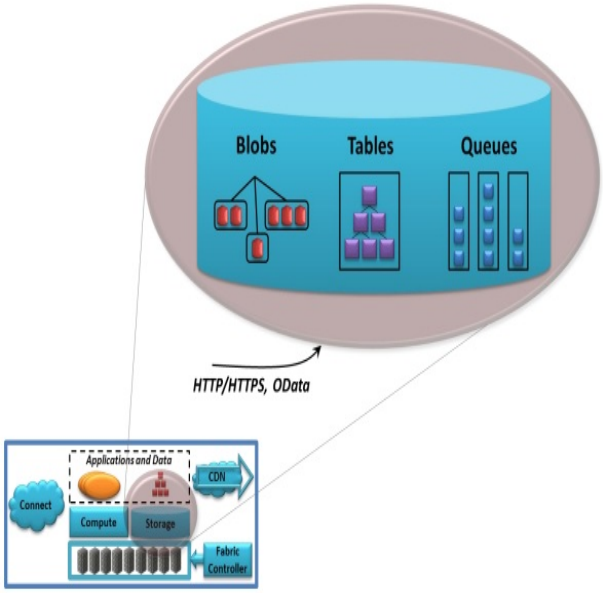


Figura 3.9: Tipi di archivi in Windows Azure

Il modo più semplice per salvare dati in Windows Azure è l' utilizzo dei blobs. Un blob contiene dati binari e, come la figura suggerisce, c'è una gerarchia semplice: ogni container può contenere uno o più blob. I blob posso essere anche di grandi dimensioni (fino ad 1 Terabyte) e possono anche avere associati dei metadati, come ad esempio informazioni su dove una fotografia JPEG è stata scattata o chi è il cantante di un file MP3. I blobs forniscono anche il meccanismo per lo storage sottostante ai Windows Azure drives, un meccanismo che consente ad un istanza di un ruolo di interagire con un servizio di storage persistente come se fosse un file system locale di tipo NTFS.

I blobs sono adeguati per certe situazioni, ma sono troppo non strutturati per altre. Per consentire alle applicazioni di lavorare con i dati con una grana più fine, Azure mette a disposizione le tabelle. Diversamente da quanto potrebbe far intendere il nome, queste non sono tabelle di tipo relazionale. Infatti, i dati che ogni tabella possiede, vengono in realtà salvati in gruppi di entità che hanno certe proprietà. Inoltre una applicazione, piuttosto che usare SQL, può effettuare una query su una tabella usando quelle che sono le convenzioni definite da OData. Questo consente al nostro servizio di salvataggio dei dati di crescere molto più efficientemente di un database relazionale (l' operazione di crescita (scale-out) viene gestita diffondendo i dati su più macchine). Bisogna infatti tenere presente che una tabella di Windows Azure può contenere miliardi di entità che gestiscono terabytes di dati.

Blob e tabelle si concentrano entrambi sul salvataggio e sull' accesso ai dati. La terza opzione che Azure offre, le code, hanno uno scopo differente. Una funzione primaria delle code è quella di consentire ad istanze di ruoli web di poter comunicare in maniera asincrona con istanze di ruoli worker. Per esempio un utente potrebbe inviare una richiesta per una operazione computazionalmente pesante attraverso un interfaccia web costruita mediante un ruolo web. L' istanza del ruolo che riceve la richiesta può scrivere il lavoro che deve essere svolto all' intero di una coda. Un istanza di worker che era in attesa su quella coda può leggere il messaggio e di conseguenza eseguire il lavoro richiesto; il risultato della computazione potrà essere ritornato mediante un altra coda o in un qualsiasi altro modo.

Indipendentemente da come i dati vengono salvati, tutte le informazioni mantenute nei servizi di storage di Windows Azure vengono replicati tre volte. In questa maniera si aumenta la tolleranza ai fallimenti, in quanto

la perdita di una copia non risulta fatale. Il sistema garantisce anche una forte consistenza, cosicché una applicazione che va a leggere un dato che ha appena scritto ottiene esattamente ciò che aveva appena scritto e non quello che c'era prima di tale scrittura. Azure mantiene anche una copia di backup di tutti i dati in un altro data center nella regione geografica selezionata per i dati dei quali si fa la copia. Se il data center che mantiene la copia principale non fosse disponibile, questo backup rimarrebbe comunque disponibile. I servizi di archiviazione di Windows Azure possono essere acceduti da applicazioni Windows Azure, da applicazioni on-premise, da applicazioni in esecuzione su un altra piattaforma o da qualsiasi applicazione in esecuzione su un host esterno. In tutte queste situazioni tutti e tre i servizi di archiviazione utilizzano la convenzione definita dal REST per identificare ed esporre i dati, come la figura suggerisce. Blobs, tabelle e code sono nominati attraverso gli URI e sono accessibili mediante normali richieste HTTP. Una applicazione .NET può utilizzare le librerie fornite da Azure che fanno questo in maniera automatica, ma non è necessario,si possono utilizzare direttamente anche le richieste HTTP.

Creare applicazioni che utilizzano blob, tabelle e code risulta sicuramente utile. Applicazioni che invece fanno uso di database relazionali come servizi di archiviazione possono utilizzare SQLAzure, che è un altro componente fornito dalla piattaforma. Le applicazioni che sono in esecuzione su Windows Azure, ma anche fuori da esso, possono utilizzare questo tipo di servizio per avere un accesso di tipo SQL ai database relazionali sul Cloud.

3.3.3 Fabric controller

Tutte le applicazioni Windows Azure e tutti i dati di archiviazione di Windows Azure risiedono in un qualche data center Microsoft. In quel centro dati, l'insieme di macchine dedicate a Windows Azure e il software che gira su di esse vengono gestiti dal fabric controller. La Figura 3.10 illustra questa idea.

Il fabric controller è esso stesso una applicazione distribuita che è replicata su un gruppo di macchine. Esso è il possessore di tutte le risorse nel suo ambiente: computer, swithces, load balancers e altro. Essendo che può comunicare con un fabric agent presente su ogni computer, è anche consapevole di qualsiasi applicazione Windows Azure presente nel suo gruppo. E' interessante notare che il fabric controller vede i servizi di archiviazione

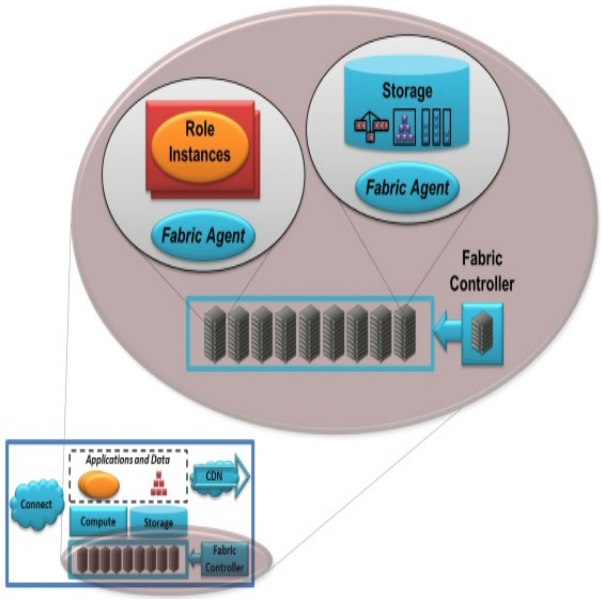


Figura 3.10: Fabric controller di Windows Azure

Windows Azure semplicemente come un altra applicazione e così i dettagli della gestione dei dati e della loro replica non gli sono visibili.

Questa ampia conoscenza permette al Fabric controller di fare un certo numero di cose utili. Esso monitora, per esempio, tutte le applicazioni in esecuzione fornendo una rappresentazione precisa al minuto di quello che sta accadendo. Decide anche dove le nuove applicazioni dovrebbero essere mandate in esecuzione, scegliendo i server fisici che ottimizzano l' utilizzo dell' hardware. Per fare questo il fabric controller ha bisogno delle informazioni di configurazione che vengono caricate con ogni applicazione Windows Azure. Questi file forniscono una descrizione basata su XML di ciò che l' applicazione ha bisogno: quante istanze di ruoli web, quante istanze di ruoli worker e altro. Quando il fabric controller rilascia una nuova applicazione utilizza il file di configurazione per determinare quante macchine virtuali creare.

Una volta create queste VM, il fabric controller inizia a monitorare ciascuna di esse. Se una applicazione richiede cinque istanze di un ruolo web ed una di esse fallisce, il fabric controller provvederà a ricrearne una in maniera

automatica. Allo stesso modo se una macchina virtuale sulla quale è in esecuzione un ruolo muore, il fabric controller inizializzerà una nuova istanza del ruolo su una macchina virtuale attiva, andando a modificare le impostazioni del load balancer per fargli indirizzare quella macchina. Windows Azure fornisce agli sviluppatori la possibilità di poter scegliere fra cinque taglie standard per dimensionare le proprie macchine virtuali, esse sono:

- Extra-small, con una CPU single-core da 1.0 GHz , 768MB di memoria, e 20GB di spazio di archiviazione.
- Small, con una CPU single-core da 1.6 GHz , 1.75GB di memoria, e 225GB di spazio di archiviazione.
- Medium, con una CPU dual-core da 1.6 GHz , 3.5GB di memoria, e 490GB di spazio di archiviazione.
- Large, con una CPU quad-core da 1.6 GHz , 7GB di memoria, e 1000GB di spazio di archiviazione.
- Extra-large, con una CPU eight-core da 1.6 GHz , 14GB di memoria, e 2040GB di spazio di archiviazione.

Un istanza extra-small condivide il core di un processore con altre entità extra-small, tutte le altre istanze hanno invece uno o più core dedicati. Questo significa che le prestazioni dell' applicazione sono prevedibili, senza alcun limite arbitrario su quanto tempo un'istanza può eseguire. Un istanza di ruolo web, per esempio, può prendersi tutto il tempo di cui necessita per gestire una richiesta di un utente, o l' istanza di un Worker potrebbe computare il valore di pigreco fino ad un milione di cifre decimali.

Per i ruoli web e worker (non per i VM), il fabric controller gestisce anche il sistema operativo in ciascuna istanza. Questo include cose come l' applicazione di patch al sistema operativo e l' aggiornamento di altro software di sistema. Questo consente agli sviluppatori di concentrarsi solamente sullo sviluppo di applicazioni, non devono preoccuparsi infatti di gestire la piattaforma stessa. E' importante sottolineare che il fabric controller assume sempre che ci siano due istanze per ogni ruolo siano in esecuzione. Questo gli consente di spegnerne una per aggiornare il proprio software senza interrompere l' esecuzione dell' applicazione. Per questa e altre ragioni mandare in esecuzione una sola istanza di un qualsiasi ruolo è di norma una cattiva idea.

3.3.4 Rete di distribuzione dei contenuti (CDN)

Un uso comune dei blobs è quello di utilizzarli per archiviare informazioni che poi potranno essere accedute da molti posti differenti. Un esempio può essere quello di una applicazione che fornisce video a client Flash, Silverlight o HTML5 sparsi in tutto il mondo. Per migliorare le prestazioni in situazioni come questa, Windows Azure fornisce una rete di distribuzione di contenuti; che semplicemente non fa altro che archiviare le informazioni in maniera tale che siano le più vicine possibili al client che le sta utilizzando. La figura 3.11 mostra questa idea:

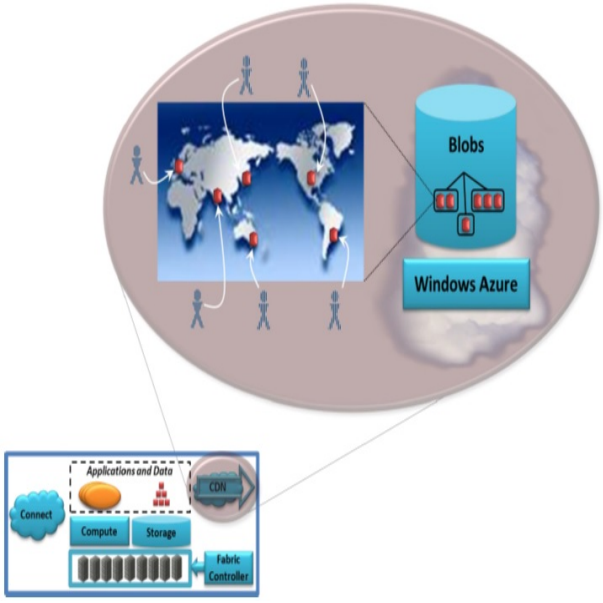


Figura 3.11: Rete di distribuzione dei contenuti Windows Azure

La figura non va presa troppo alla lettera, infatti il CDN di Windows Azure ha molte più locazioni per i propri server di quanto mostrato in figura, ma il ragionamento è corretto. La prima volta che un certo blob viene acceduto da un utente, il CDN archivia una copia in una locazione geograficamente vicina a quel determinato utente. La prossima volta che il blob sarà acceduto i dati verranno forniti dalla copia che è stata appena fatta e non dall' originale che risulterebbe più lontana.

Per esempio, supponiamo che Windows Azure venga utilizzato per fornire video di eventi sportivi di una particolare giornata ad un pubblico lontano. Il primo utente che accede ad un determinato video non otterrà il beneficio del CDN, dal momento che blob non è ancora stato copiato in una posizione più vicina. Tutti gli altri utenti della stessa regione geografica vedranno però migliori prestazioni poiché, l'utilizzo della copia appena fatta, consentirà loro di caricare il video più velocemente.

3.3.5 Connessione

L'esecuzione di applicazioni nel cloud di Microsoft è sicuramente utile. Ma le applicazioni e i dati che vengono utilizzati all' interno delle organizzazioni non andranno via in tempi brevi. Detto questo risulterà molto importante connettere efficacemente gli ambienti locali con Windows Azure.

La connessione di Windows Azure è stata progettata per aiutare a realizzare quanto appena detto. Fornire una connettività a livello IP fra le applicazioni Windows Azure e le macchine in esecuzione fuori dal Cloud Microsoft, rende questa configurazione più facile da utilizzare. La figura 3.12 mostra quella che è l' idea.

Come mostra la figura, usare la connessione Windows Azure richiede l' installazione di un agente endpoint in ogni computer locale che è connesso all' applicazione Windows Azure. Poiché la tecnologia si basa su IP v6 l' agente endpoint è oggi disponibile solo per Windows Server 2008, Windows Server 2008 R2, Windows Vista, Windows 7 e Windows 8. L' applicazione Windows Azure ha anche bisogno di essere configurata per poter lavorare con il servizio di connessione. Una volta fatto ciò, l'agente potrà utilizzare IPsec per interagire con un qualsiasi ruolo all' interno dell' applicazione.

Si noti che questa non è una rete privata virtuale a tutti gli effetti (VPN); Windows Azure Connect è una soluzione più semplice. La sua creazione non necessita di contattare l'amministratore di rete, per esempio. Tutto ciò che è richiesto è la possibilità di installare l'agente endpoint sulla macchina locale. Questo approccio nasconde anche la potenziale complessità di configurazione di IPSec, che viene gestita in automatico dal servizio di connessione Winows Azure.

Una volta che la tecnologia è stata configurata i ruoli in una applicazione Windows Azure appariranno come se fossero sulla stessa rete IP della macchina locale; questo permette le seguenti cose:

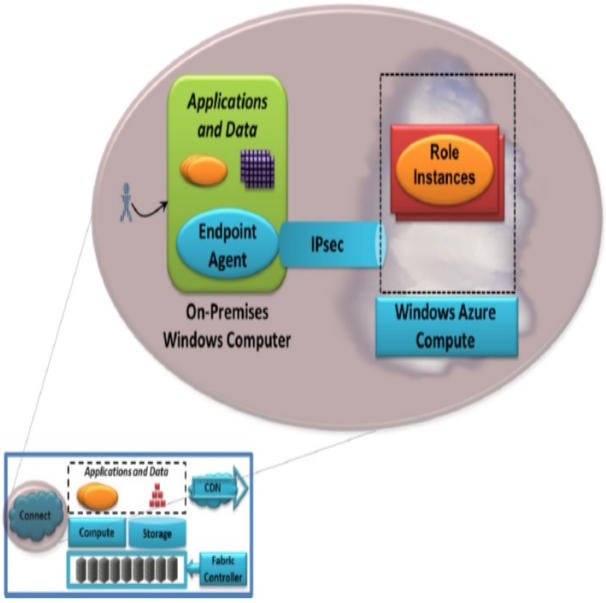


Figura 3.12: Connessione in Windows Azure

- Una applicazione su WA potrà accedere ad un database locale direttamente. Per esempio si supponga che una azienda trasformi una applicazione esistente costruita mediante ASP.NET e che si appoggia su Windows Server, in una applicazione ruolo web di Windows Azure. Se il data base che questa applicazione usa ha bisogno di rimanere in locale, una connessione realizzata mediante Windows Azure Connect consente all' applicazione, che ora è in esecuzione su WA, di poter accedere al database come ha sempre fatto; neanche le stringhe di connessione necessitano di essere cambiate.

- Un applicazione Windows Azure può essere domain-joined, all' ambiente locale. Facendo questo si consente il Single Sign-On all' applicazione per gli utenti locali. Questo consente inoltre all' applicazione di utilizzare account Active Directory esistenti e gruppi per il controllo dell' accesso.

Far si che il cloud si adatti bene con gli ambienti locali di oggi è importante. Consentendo una connettività diretta a livello IP, Windows Azure

Connect rende tutto questo più facile per le applicazioni Windows Azure.

3.4 Caratteristiche comuni ai middleware

Le caratteristiche che possono offrire i middleware sono le più svariate. Quando però si parla di middleware per il cloud la caratteristica comune a tutti è quella di rendere facile il passare da una piattaforma ad un altra e rendere il più facile il portare sul cloud una applicazione concentrata. La scelta di un determinato middleware dipende perciò dal costo che esso ha in proporzione ai benefici che esso è in grado di offrirci, oltre ai linguaggi che esso supporta e alla facilità con la quale possiamo usufruire dei servizi offerti.

Ho scelto di trattate Cloudify in quanto è uno dei più noti middleware free per Cloud app. Esistono anche middleware a pagamento (come ad esempio quello di IBM), anche essi molto validi, anzi probabilmente migliori in molti aspetti; però il fatto che Cloudify fosse free mi ha spinto ad approfondirlo anche per un interesse personale, in quanto chiunque, anche senza risorse, può iniziare ad approcciarsi con il mondo del cloud e a costruire le sue applicazioni senza vincoli monetari, ma con uno strumento comunque di qualità.

3.5 Cloudify

Cloudify è un middleware che si interpone fra la nostra applicazione e la piattaforma cloud(PaaS) sulla quale abbiamo scelto di mandarla in esecuzione. La sua caratteristica principale è quella di consentirci di poterci concentrare esclusivamente sulla nostra applicazione, senza doverci preoccupare della piattaforma cloud in quanto è Cloudify che si preoccupa che le risorse delle quali ha bisogno gli siano assegnate , alleggerendo quindi di gran lunga il lavoro del programmatore. Cloudify si propone sostanzialmente tre obiettivi:

- **No code change**, ovvero la necessità di non dover cambiare il codice della nostra applicazione nel momento in cui decidiamo di trasferirla sul cloud. Questa operazione è già di per sè difficoltosa se l'applicazione non è stata ben progettata, ma lo può diventare ancora di

più qualora le piattaforme cloud non forniscano i servizi sui quali la nostra applicazione si basa o se, più semplicemente, non forniscono il giusto ambiente di sviluppo. Per risolvere questi problemi Cloudify ha introdotto i Percipes, entità che vedremo in seguito.

- **No lock in**,ovvero la necessità di poter cambiare il proprio cloud provider facilmente qualora insorga la necessità. Questo, come già detto, non è sempre facile perché ogni piattaforma ha le sue caratteristiche e offre i suoi servizi. Per offrire questa caratteristica, Cloudify nasconde all' applicazione le API tipiche di ogni piattaforma, disaccoppiandola così completamente da quest' ultima e rendendo la migrazione da una piattaforma all' altra estremamente semplice. Cloudify supporta tutti i principali cloud sia pubblici (Amazon EC2, Windows Azure, Rackspace, ...) che privati (OpenStack, CloudStack, VMWare vCloud, Citrix XenServer, ...).

- **Full controll**, ovvero il non voler rinunciare al controllo sull' ambiente nel quale la nostra applicazione è in esecuzione, configurare le risorse in base alle nostre necessità o più semplicemente monitorarne l' utilizzo (ricordando la modalità di pagamento pay-per-use dei cloud provider). Cloudify rende possibile tutto ciò perché possiede l' accesso all' infrastruttura e di conseguenza ci fornisce questo livello di controllo; ma solo se ne abbiamo necessità: qualora infatti non fossimo interessati a tali proprietà il sistema caricherà automaticamente delle configurazioni predefinite che utenti meno esperti possono trovare comunque soddisfacenti. Qualora fossimo interessati invece è possibile monitorare e configurare molte caratteristiche, dalla memoria occupata, ai log che rilasciano i vari servizi, alle risorse computazionali sfruttate come l'impiego di RAM o CPU. Per poter gestire e visualizzare questi dati si può usare o una console Online o la shell di Cloudify, entrambe personalizzabili andando ad aggiungere comandi o visualizzatori semplicemente modificando opportunamente un Recipe (argomento trattato in seguito) di servizio.

3.5.1 Interprete comandi

Cloudify offre ai propri utilizzatori un'interprete comandi, tramite il quale,sarà possibile installare e gestire tutte le applicazioni distribuite nel Cloud.

La lista completa dei comandi, utilizzabili nel CLI di Cloudify, può essere richiamata tramite un file di bash. In questa maniera risulta possibile utilizzare la piattaforma, non solo da un gestore umano, ma anche da un qualunque processo che possa richiamare file bash; si rende così possibile un'automazione della gestione. Per fare tutto ciò, basta richiamare il file cloudify.sh passandogli come parametro tutti i comandi, uno dopo l'altro nella giusta sequenza di utilizzo, separati dal punto e virgola. Ad esempio per installare il servizio ServizioProva, per l'applicazione AppProva, installato sul localcloud di un certo host, basterà lanciare il file bash contenente la seguente riga: */bin/cloudify.sh connect 127.0.0.1;use-application AppProva;install-serviceServizioProva*

Tramite questo parametro, il sistema operativo, in questo caso Ubuntu, cercherà il file cloudify.sh nella cartella di root bin e lo eseguirà. Questo, tramite i parametri passati, si connetterà al server REST local-host, dichiarerà di utilizzare l'applicazione passata e di voler installare il servizio passato su di essa.

3.5.2 Funzionalità

Cercando di approfondire un pò quelle che sono le principali funzionalità e innovazioni offerte da Cloudify non possiamo non citare il fatto che permette di creare un ambiente di simulazione cloud sulla propria macchina, offrendo un'infrastruttura cloud che risponde in local-host. Questo ambiente permette di testare le applicazioni prima di metterle in un ambiente reale, risparmiando molto tempo e denaro. Un' altra caratteristica è inoltre quella di rendere estremamente agile il deploy della nostra applicazione essendo che, come già citato, Cloudify si interpone fra la nostra applicazione e quello che è la piattaforma sottostante; inoltre provvede all'installazione e alla configurazione di servizi sopra l'infrastruttura Cloud collegata, sia essa di simulazione o reale. Inoltre provvede anche, in fase di disinstallazione delle applicazioni, a liberare lo spazio utilizzato rilasciando correttamente le risorse cloud utilizzate. Cloudify basa il deploy delle proprie applicazioni sul concetto di Recipes che, a differenza di quanto avviene normalmente, sono entità che non solo descrivono l'applicazione, ma anche tutti i servizi che essa offre e le dipendenze rispetto ad altre entità. Grazie a questo meccanismo Cloudify riesce a fornire i meccanismi per poter gestire la propria applicazione non solo durante l' installazione (come avviene spesso in altri

cloud) ma per tutto il suo ciclo di vita. Cloudify da infine la possibilità di poter creare procedure di auto-scaling e probes (sonde) che permettono di sondare, per l'appunto, l' ambiente di esecuzione e i servizi distribuiti sopra ad essa.

3.5.3 Recipes

Riprendendo direttamente quanto scritto sulla documentazione ufficiale definiamo un Recipe come un piano di esecuzione per installare, eseguire, orchestrare e monitorare il livello dell'applicazione, senza cambiare il codice dell'applicazione o l'architettura. Su cloudify esistono due tipi di Recipes :

- **application recipes**: che sono quei recipes che descrivono il nome dell' applicazione i servizi di cui hanno bisogno e le loro interdipendenze. Questi sono composti dal file descriptor dell' applicazione più dai Recipes degli altri servizi dei quali hanno bisogno.

- **service recipes**: sono quei recipes che descrivono i servizi sui quali si basa la nostra applicazione; questi comprendono: service descriptor file, handlerscripts, opzionali plugins di monitoraggio, il nome opzionale di un'icona rappresentante il servizio e un opzionale file di parametri per recipes.

Pensando ora che Cloudify vede un' applicazione come un insieme di servizi e un servizio come un insieme di istanze di servizio che formano un livello applicativo, è importante fare una distinzione fra i servizi e le istanze di un servizio. Infatti un servizio deve essere composto da almeno un istanza di servizio e ogni istanza può risiedere su un solo host. Essendo però che un servizio è l'aggregazione di tutte le istanze del servizio, come tale può estendersi su più macchine mentre queste ultime no. Infine per fare il deploy della nostra applicazione occorre organizzare i file nella seguente maniera: tutti i file che formano l' application recipe devono trovarsi nella cartella recipe relativa all' applicazione. Se l'applicazione sfrutta dei servizi, anche questi dovranno essere inseriti all'interno della cartella dell'applicazione in una cartella propria. Lascio di seguito un'immagine che spiega la convenzione di divisione in sotto-cartelle:

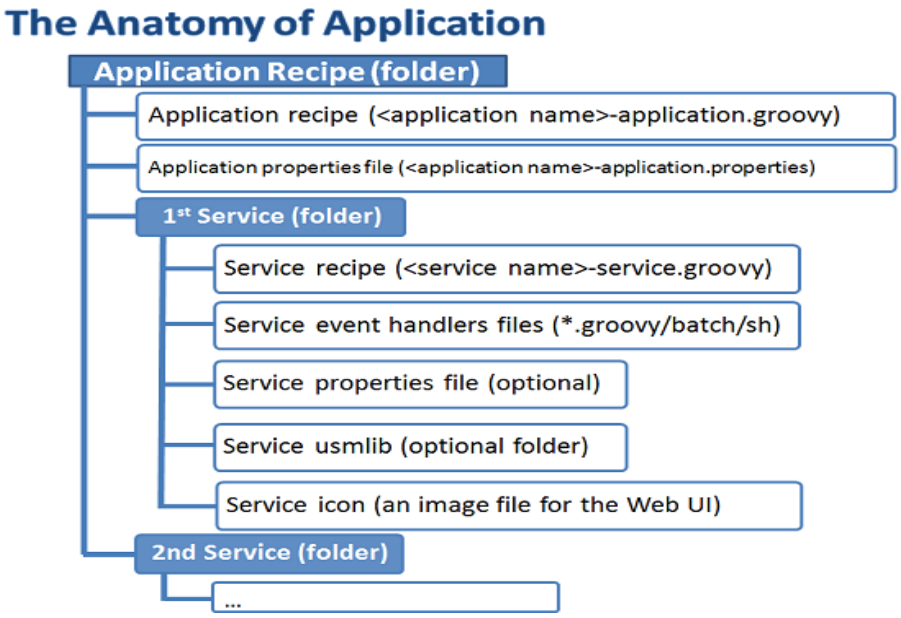


Figura 3.13: Gestione in cartelle dell' applicazione con Cloudify

3.5.4 Application Recipes

Cercando qui di approfondire il discorso su quello che è il descriptor file dell' applicazione ricordiamo che quest' ultimo ne definisce il nome, i servizi di cui ha bisogno per funzionare e le loro interdipendenze. La dipendenza significa che Cloudify non manderà in esecuzione un particolare servizio finché tutti i servizi dai quali esso dipende non saranno stati a loro volta attivati con successo. Il nome dell' applicazione poi deve essere univoco nell' insieme delle applicazioni che sono gestite dallo suo stesso Cloudify controller; questo perché esso viene utilizzato dal sistema per identificare l' applicazione agli occhi dei vari tools di gestione e per la visualizzazione delle risorse utilizzate. Un esempio di descriptor file di una applicazione potrebbe essere quello riportato in figura 3.14.

3.5.5 Service Recipes

Anche in questa sezione cercheremo di approfondire la composizione di quello che è il service descriptor file; esso è composto da:

```
application {
  name="petclinic-mongo"

  service {
    name = "mongod"
  }

  service {
    name = "mongoConfig"
  }

  service {
    name = "mongos"
    dependsOn = ["mongoConfig", "mongod"]
  }

  service {
    name = "tomcat"
    dependsOn = ["mongos"]
  }
}
```

Figura 3.14: File descriptor di una applicazione

- **General**: sezione nella quale sono contenuti il nome del servizio, l'immagine relativa all' icona, il numero massimo di istanze e il tipo del servizio (quest' ultimo campo è usato dalla console di gestione Online di Cloudify per determinarne la posizione all' interno della Application Map).

- **Compute**: specifica il nome del modello da utilizzare durante la richiesta di macchine. Questo nome fa parte di un insieme di modelli presenti nel Cloud driver che descrivono i profili hardware sull' insieme di macchine che possono essere utilizzate quando richiediamo nuove istanze di servizio. Tutto ciò permette un grande disaccoppiamento tra l' applicazione e l'architettura Cloud sottostante; maggiori approfondimenti verranno forniti nella sezione Cloud Driver.

- **Lifecycle events**: in questa sezione associamo gli handler ai relativi eventi e come tale è la sezione più importante del Service descriptor file. Gli handlers devono risiedere nella service folder e possono essere

o un Groovy script esterno o uno script shell o un file batch esterno (a seconda del sistema operativo sul quale stiamo operando).

- **Custom Commands**: è una mappa di comandi personalizzati per la gestione degli script che possono essere invocati come comandi usando la shell Cloudify.
- **Probes**: sono le sonde che monitorano la configurazione del servizio, le performance e la disponibilità.
- **UI Layout**: il layout da utilizzare per mostrare i dati relativi al servizio sulla console di gestione Online di Cloudify.

Cloudify permette l'implementazione di alcuni servizi sullo stesso host. Lo scopo di questa funzionalità è quello di consentire il massimo utilizzo della macchina sottostante al fine di non sprecare risorse hardware. Sono supportate quattro modalità:

- **dedicated**(default) : l' host sarà dedicato solo a questa istanza di servizio; nessun altro servizio potrà essere rilasciato su questo host.
- **global**: l' host può essere condiviso da tutte le applicazioni e cloud tenant basandosi sulle regole scritte nella sezione Isolation SLA (sezione nella quale sono descritti, oltre al tipo di condivisione, altri parametri come ad esempio il numero di CPU richieste e la quantità di memoria necessaria per essere mandata in esecuzione)
- **appShared**: i servizi che appartengono alla stessa applicazione possono condividere l' host.
- **tenantShared**: i servizi che appartengono allo stesso gruppo di autenticazione possono condividere l' host.

3.5.6 Lifecycle events

Il ciclo di vita di una applicazione si compone di vari eventi, esempi possono essere l'installazione o lo start di un processo. Questi eventi Cloudify li distingue in tre categorie: application event, service events e service instance events. Quando un evento viene rilevato uno script Handler, che incapsula le operazioni da eseguire una volta verificato tale evento, viene mandato in esecuzione.

3.5.7 Application events

Sono eventi Built-in all' interno di Cloudify, possono essere richiamati digitando opportuni comandi sulla shell di quest' ultimo ma non possono essere personalizzati. Ci sono due application events:

- **application install**: per installare una applicazione scrivere sulla shell il seguente comando: *install-application <applicationName>* . Questo comando fornisce i servizi applicativi richiesti, inizializza i servizi in ordine di dipendenze, e quindi installa l'applicazione.
- **application uninstall**:per disinstallare una applicazione scrivere sulla shell il seguente comando: *uninstall-application <applicationName>* . Questo comando arresta e rimuove tutti i servizi dell'applicazione e agenti Cloudify, e poi termina le macchine su cui i servizi erano in esecuzione.

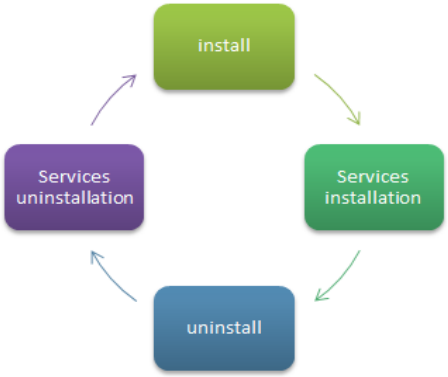


Figura 3.15: Ciclo di vita degli application events

3.5.8 Service events e Service instance events

Entrambi questi tipi di eventi, a differenza degli Application Events, possono essere personalizzati all' interno del Service Recipe, quindi non vengono chiamati in seguito alla digitazione di un comando sulla shell ma vengono

attivati direttamente dal sistema. In particolare i Service Events sono particolarmente utili per eseguire azioni subito prima che le istanze del servizio vengano fornite o subito dopo la rimozione dell' ultima istanza del servizio ma prima che il cluster sia arrestato. Il seguente schema mostra il ciclo di vita dei Service Events.

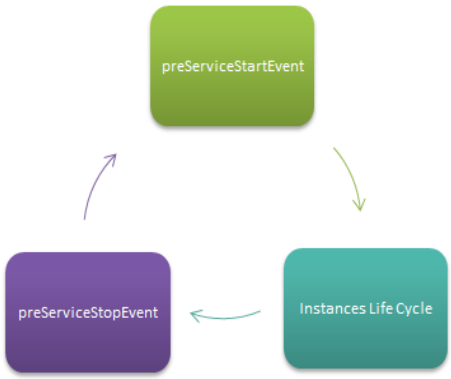


Figura 3.16: Ciclo di vita dei service event

3.5.9 Event Handler

Una volta individuati quali eventi il nostro servizio deve intercettare, si deve dichiarare un' apposito handler nella definizione di ciclo di vita nel file descriptor. Ogni handler è collegato ad un certo evento, tramite esso permettiamo a Cloudify di lanciare un file script, oppure di eseguire direttamente qualche operazione, nell'istante prescelto. Tutti i file utilizzati e richiamati dovranno essere presenti nella stessa cartella del file descriptor del servizio. Teoricamente è possibile lanciare qualsiasi script scritto in qualsiasi linguaggio, questo però deve essere lanciato e gestito da uno script di un formato concesso. E' anche disponibile una funzionalità che permette di associare Handler diversi a seconda del particolare sistema operativo che stiamo utilizzando; nella figura 3.17 è lasciato un esempio di quanto appena detto:

Lascio anche un elenco con tutti gli eventi ai quali è possibile associare un handler all' interno di Cloudify:

- preServiceStart

```
lifecycle{
  init "tomcat_install.groovy"
  start ([
    "Win.*" : "tomcat_run.bat",
    "Linux" : "tomcat_run.sh",
    "Mac.*" : "tomcat_run.sh"
  ])

  preStop ([
    "Win.*" : "tomcat_stop.bat",
    "Linux" : "tomcat_stop.sh",
    "Mac.*" : "tomcat_stop.sh"
  ])
}
```

Figura 3.17: Gestione degli eventi su diversi OS

- init
- preInstall
- install
- postInstall
- preStart
- start
- startDetection
- stopDetection
- locator
- postStart
- preStop
- stop
- postStop

- shutdown
- preServiceStop

3.5.10 Estendere i service recipes

In molti scenari si può avere la necessità di voler creare un nuovo servizio che però non differisce molto da altri servizi precedentemente creati. Essendo che sarebbe scomodo ricopiare il codice del servizio interessato e cambiare o aggiungere solo le parti di codice che ci interessa modificare, Cloudify mette a nostra disposizione la possibilità di estendere i Service Recipe. Grazie a questo meccanismo è possibile dichiarare che un servizio ne estende un altro (in maniere tale che ne erediti tutte le proprietà) e poi andare a fare l' override (spiegato più avanti) o ad aggiungere solo gli Handler agli eventi che effettivamente ci servono. L'estensione è inoltre ricorsiva, ovvero un servizio può estenderne un altro e può essere esteso a sua volta senza alcun limite e si basa sul concetto di Inheritance (ereditarietà) ovvero un servizio che ne estende un altro ne eredità la proprietà solo se non vi è in già esso un elemento più specifico associato ad un particolare evento (l' ovverride di prima). Cercando di fornire un esempio si prenda il codice di sinistra della figura 3.18 come quello del servizio da estendere e quello di destra come quello che lo ha esteso:

In questa situazione il figlio eredita dal padre il comportamento per l' evento postStart, fa l' ovveride del comportamento su init e aggiunge un nuovo comportamento che il servizio padre non aveva per l'evento preStop.

```
1 service{
2   name "myService"
3   lifecycle {
4      postStart …
5      init …
6   }
7 }
```

```
1 service{
2   extend <myService path>
3   lifecycle {
4      init … <my new init behavior>
5      preStop …
6   }
7 }
```

Figura 3.18: Estensione dei recipes

3.5.11 Scaling rules

Essendo che, come ben noto, la scalabilità è uno dei principi cardine delle Cloud application e del Cloud in generale, Cloudify mette a disposizione due metodi per consentire la scalabilità di un servizio, uno automatico e l' altro manuale. Per quel che riguarda l' automating scaling Cloudify da la possibilità ad ogni servizio di definirsi le proprie regole per decidere quando aumentare le risorse da utilizzare (scale out) o quando diminuirle (scale in); per esempio si potrebbe definire una regola basata sul numero di thread correntemente occupati relativi ad un servizio e, in base a questo, scegliere se aumentare o meno il numero di istanze. Tutte le regole hanno un limite minimo e un limite basso che si basa su dei dati statistici raccolti dai plugin (sono dei particolare tipi di probes, argomento che non ho approfondito in questa tesi ma ampliamente descritto nella documentazione ufficiale di Cloudify). A questo punto la logica è molto semplice, quando i dati ricevuti superano il limite massimo il sistema crea nuove istanze del servizio, viceversa, qualora venga superato il limite inferiore, verranno eliminate istanze che non sono più utili. Per quel che riguarda invece il manual scaling basta digitare sulla console di Cloudify il seguente comando : *set-instances service-name number-of-required-instances* (dove ovviamente gli ultimi due parametri vanno sostituiti rispettivamente col nome dell' applicazione e il numero di istanze richieste). Bisogna sottolineare come questo comando venga utilizzato sia per aggiungere che per rimuovere istanze di servizio; infatti se il numero di istanze inserito è maggiore di quello corrente nuove istanze vengono create, mentre se è minore quelle in eccesso vengono rimosse.

3.5.12 Cloud driver

I Cloud driver vengono visti come un livello di astrazione fra la piattaforma e l'ambiente cloud sottostante. Essi sono i responsabili dell'interfacciamento con l'infrastruttura cloud, provvedendo a richiedere on-demand le risorse richieste dalle applicazioni installate sopra Cloudify. Questo layer viene sfruttato, quando si istanziano o rimuovono macchine virtuali di gestione, tramite i comandi bootstrap-cloud, teardown-cloud, quando si istanziano o rimuovono macchine virtuali per le applicazioni install-application,uninstall-application, oppure vengono sfruttati dall'ESM. L'ESM è il responsabile dell' auto-scaling, ne viene creato dall'infrastruttura uno per ogni istanza

di servizio, per sua natura dovrà interagire con il Cloud e quindi sfrutterà i Cloud driver. Tramite questi Cloud driver, sarà possibile richiedere ai vari provider di servizio, siano essi pubblici o privati, un certo quantitativo di risorse, siano esse computazionali o di gestione del sistema. Ad esempio, sarà possibile richiedere un certo range di porte per le varie applicazioni installate sulla piattaforma, oppure richiedere più o meno processori dedicati. Essendo che i Cloud provider e le API non sono perfetti può accadere che si generino degli errori che vanno gestiti. E' importante ricordare che l' infrastruttura del Cloud Driver non è transazionale quindi è compito del Cloud Driver gestire in maniera propria gli errori che possono scaturire dalla piattaforma cloud sottostante e, cosa ancora più importante, è sempre compito del Cloud Driver rilasciare propriamente le risorse qualora un errore avvenga durante la richiesta di una macchina. Ad esempio, se un Cloud Driver richiede una macchina dal Cloud e poi attende che la macchina diventi disponibile, la macchina potrebbe impiegare troppo tempo prima di diventare disponibile, superando così il timeout prefissato. In questo caso è compito del Cloud Driver spegnere la macchina richiesta prima di lanciare una TimeOutException all' infrastruttura.

Capitolo 4

Modello ad Attori

L' introduzione in questa sezione del modello di programmazione ad attori nasce dal fatto che esso rappresenta sicuramente un modello di programmazione valido per le applicazioni cloud come ad esempio vedremo in seguito per Orleans (un framework per sviluppare applicazioni cloud gestito dalla microsoft). Innanzi tutto diciamo che il modello ad attori è un modello nato nell' ambito della programmazione concorrente e non di quella distribuita. Le sue caratteristiche di atomicità, fairness e locazione lo hanno reso un modello molto apprezzato anche in altri ambiti. Cercando di fare un paragone per capire meglio le proprietà fondamentali di questo paradigma, prendiamo in considerazione quello che è un paradigma a noi ben noto, quello ad oggetti. In questa visione un oggetto incapsula sia i dati che il suo comportamento; questo separa l'interfaccia di un oggetto (cosa fa) dalla sua rappresentazione (come lo fa). Ciò permette un ragionamento di tipo modulare durante la progettazione di sistemi object-based e ne facilità la loro evoluzione. Gli attori aggiungono ai vantaggi degli oggetti quelli della computazione concorrente separando il controllo (dove e quando) dalla logica della computazione.

Avendo fatto una sorta di paragone fra questi due modelli diamo ora una definizione più precisa di che cosa è un attore e un sistema ad attori. Un attore è un oggetto autonomo che opera in maniera concorrente e asincrona, ricevendo ed inviando messaggi ad altri attori, creando nuovi attori, ed aggiornando il suo stato locale. Un sistema ad attori consiste in una collezione di attori, alcuni dei quali possono ricevere o inviare messaggi da/ad attori esterni al sistema.

Infine nel corso dell' esposizione parlerò anche di quelle che sono i principali framework per la JVM, confrontandoli e mettendo in luce quelle che sono le caratteristiche che offrono.

4.1 Attori

Gli attori sono come, abbiamo già detto, entità autonome all' interno del nostro sistema. Un attore ha un nome che lo identifica univocamente all' interno del sistema e che gli permetterà quindi di essere raggiunto dai messaggi degli altri attori. Essendo che un attore, che vuole inviare un messaggio ad un altro, ne deve conoscere l' indirizzo, è importante cercare di capire come può fare per ottenerlo. Le modalità sono tre e sono o che era già presente all'interno dell' attore stesso, o che è dato come risultato di una computazione o, infine, che gli è stato inviato all' interno dell' ultimo messaggio ricevuto. Si può notare, da quanto appena scritto, che un indirizzo quindi non può mai essere suggerito da altri attori presenti all' interno del sistema se non tramite un messaggio. Quanto appena detto ci porta subito ad concetto fondamentale, ovvero che gli attori, come unico modo per interagire gli uni con gli altri, hanno lo scambio di messaggi; questo implica che gli attori non possono condividere lo stato, quindi le computazioni parallele non presentano problemi di concorrenza, non potendo interferire tra loro in alcun modo . Tutto ciò ci introduce a quello che è il concetto di task. I task non sono niente altro che il modo che esiste nei sistemi ad attori per rappresentare le comunicazioni; essi sono composti da una etichetta univoca che gli identifica, l' indirizzo del destinatario (target) del messaggio e da un insieme di parametri che l' attore dovrà elaborare. Come già sottolineato alla ricezione di un task un attore può fare solo tre cose:

- inviare un messaggio ad un altro attore
- cambiare il proprio stato
- creare un altro attore

Ovviamente queste tre non sono mutuamente esclusive, nel senso che un attore, una volta ricevuto un messaggio potrebbe eseguire solo una di queste azioni così come tutte e tre. Quindi man mano che la computazione procede il sistema evolve creando nuovi task e nuovi attori che sono creati

nel processare task già presenti all' interno del sistema. Tutti i task che sono già stati processati e che non sono più utilizzati possono essere rimossi dal sistema (garbage collection); citando direttamente Gul Agha *we can say that the unprocessed tasks in a system of actors are the driving force behind computation in the system* il cui significato è che i task non ancora processati, in un sistema ad attori, rappresentano la forza che guida la computazione dell' intero sistema.

L' attore ha, come strumento per ricevere i messaggi, una mailbox o coda. Qualora arrivi un messaggio nella mailbox e l' attore non sia già occupato a processarne un altro se ne prende carico chiamando dei metodi (similmente a quanto accade nel modello ad oggetti) che sono funzione sia del messaggio ricevuto ma anche nello stato corrente. Più nel dettaglio un attore una volta finito di processare un messaggio entra nello stato di idle e guarda nella mailbox se vi sono presenti altri messaggi. Qualora ve ne siano l' attore prenderà il primo messaggio all' interno della lista e lo processerà, altrimenti entrerà in uno stato di waiting aspettando l' arrivo di un nuovo messaggio. A questo punto è importante notare, come già accennato, che l' arrivo di un messaggio non può mai interrompere la computazione che un attore sta eseguendo, ma viene sempre portata a termine prima di processare un nuovo messaggio.

Un altra caratteristica fondamentale del modello ad attori è che essa introduce il non determinismo. Infatti i messaggi vengono consegnati, attraverso la rete, secondo una politica di tipo best effort, questo implica che, anche qualora inviassimo due messaggi l' uno di seguito all' altro allo stesso attore, non potremmo mai avere la certezza di quali dei due arrivi per primo a destinazione. Questo è dovuto al fatto che non è detto che i messaggi prendano la stessa strada per arrivare all' attore finale e di conseguenza i ritardi introdotti dalla rete non sono predicibili; senza poi considerare il fatto che il modello adotta il parallelismo e la concorrenza. Infine ricordiamo che i messaggi vengono inviati in maniera asincrona, di conseguenza una volta inviati dall' attore, quest' ultimo riprende immediatamente la propria esecuzione, senza aspettare alcun tipo di conferma né dal sistema né dal destinatario. Lascio di seguito un immagine che spero renda tutto più chiaro e che riassume quanto detto in questo paragrafo.

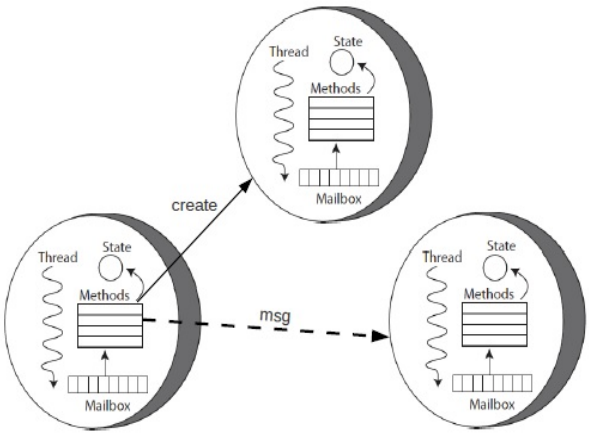


Figura 4.1: Rappresentazione grafica di attori

4.2 Astrazioni di programmazione

Andremo ora a discutere due astrazioni di programmazione utili per la comunicazione e la sincronizzazione nei programmi ad attori.

4.2.1 Remote procedure call

E' un pattern di comunicazione RPC-like che viene comunemente utilizzato per lo scambio di messaggi nella programmazione ad attori. In questo tipo di comunicazione il mittente invia un messaggio e poi aspetta la risposta del destinatario prima di poter continuare con la propria esecuzione, così facendo si introduce un elemento bloccante all' interno della comunicazione. Quanto appena detto può essere ben rappresentato dalla figura 4.2 dove è stato preso come esempio un attore che richiede il prezzo di un viaggio in treno e, prima se scegliere di acquistare il biglietto o chiedere ad un altro emittente aspetta l' arrivo della risposta.

Questo modello si dimostra particolarmente utile in due situazioni:

- la prima è quella situazione nella quale un attore vuole essere sicuro che il destinatario dall' altra parte riceva una sequenza ordinata di messaggi, di conseguenza vuole essere sicuro che un determinato messaggio sia arrivato a destinazione prima di procedere con l' invio del

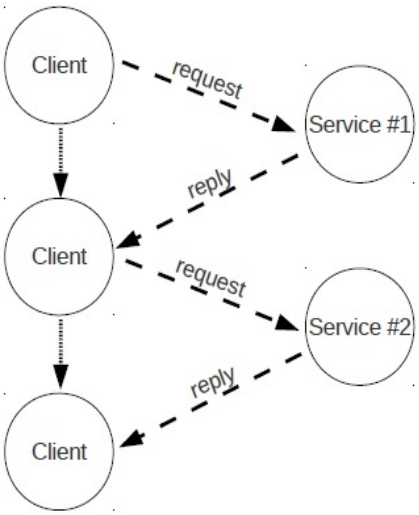


Figura 4.2: Modello di interazione RPC

successivo. Una variante può essere se il mittente vuole essere sicuro che il messaggio sia arrivato a destinazione prima di passare questa informazione ad un terzo attore.

- la seconda avviene quando lo stato dell' attore mittente dipende dalla risposta che riceverà dal destinatario. In questo caso non ha senso che l' attore richiedente processi nuovi messaggi fino a quando non ha ricevuto la risposta, perché a seconda dello stato in cui si troverebbe il messaggio da elaborare potrebbe scaturire comportamenti diversi.

Essendo che questo tipo di pattern è molto simile a quello delle chiamate di procedura nella programmazione concentrata molti programmatori tendono ad abusarne. Sfortunatamente un uso non corretto ed inappropriato di questo pattern porta a delle dipendenze non necessarie all' interno del programma che non solo ne diminuiscono le performance, ma possono anche causare situazioni spiacevoli quali deadlock e livelocks (questi ultimi avvengono quando un attore ignora o pospone il processamento di un messaggio in attesa di una risposta da un attore che non arriverà mai).

Questo tipo di pattern viene supportato da praticamente tutti i framework per la JVM, in particolar modo esso è fornito in maniera primitiva in Scala Actors, SALSA, Actor Architecture and ActorFoundry.

4.2.2 Vincoli di sincronizzazione locali

Una delle implicazioni introdotte dalla programmazione sincrona è quella che il numero di possibili ordini nei quali possono arrivare una serie di messaggi cresce in maniera esponenziale con il numero di messaggi che stiamo mandando. Questo porta al fatto che il mittente non può sapere a priori in che stato si trova colui che deve ricevere il messaggio nel momento in cui lo andrà a ricevere. Non sa dunque se esso sarà in uno stato che gli consentirà di processare il messaggio oppure no. Come esempio si può pensare quello di uno spooler (programma per la gestione di stampanti in ambiente distribuito) alla quale arriva da una stampante una richiesta di stampa ma lui al momento non ha nulla da far stampare. Un prima metodologia per tentare di ovviare a questa situazione è quella di far rifiutare allo spooler la richiesta. A questo punto il programma che gestisce la stampante farà del polling sullo spooler andandogli a chiedere di continuo se ha un lavoro disponibile da affidargli; questa tecnica viene chiamata busy waiting. Fare busy waiting non è mai una scelta saggia, in primo luogo per se si sprecano risorse computazionali e di rete per fare una operazione ripetitiva e sostanzialmente inutile, in secondo luogo perché il programma, invece di continuare ad andare avanti con quello che è il suo lavoro, si blocca ad effettuare delle richieste, inoltre se sono molte le stampanti che utilizzano questa metodologia e lo spooler è solo uno, si appesantisce anche quella che è la sua computazione, perché sarà sempre impegnato a mandare risposte negative a tutti i richiedenti.

Un metodo alternativo per cercare di andare incontro a quella che è questa situazione è quello di introdurre un buffer all' interno dello spooler. L'effetto di tale buffer è quello di cambiare l' ordine nel quale i messaggi vengono processati, in una maniera tale da garantire che il numero di messaggi di put ricevuti dallo spooler siano sempre in numero maggiore dei messaggi get processati. Fuori dall' esempio tramite l' inserimento di questo buffer si cerca di contrastare quello che è il non determinismo, che impedisce di conoscere l'ordine di arrivo dei messaggi, con l'introduzione di vincoli che invece ne specificano 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.

Fra i framework disponibili per la JVM solo le librerie di Salsa Actor forniscono vincoli di sincronizzazione locali. Il loro supporto per tali vincoli si basa sul pattern-matching dei messaggi in arrivo (come in Erlang). I messaggi che non corrispondono al modello di ricezione vengono posticipati e possono essere riscontrati da un modello diverso in un momento successivo dell' esecuzione.

4.2.3 Comparazione

Prima di tutto bisogna sottolineare che le astrazioni fornite dai linguaggi sono puramente sintattiche e come tali non cambiano la semantica sottostante degli attori. Ricerche hanno dimostrato formalmente come sia possibile traslare un programma ad attori con i vincoli appena descritti in uno che i vincoli non li ha.

In aggiunta al request-replay messagging e ai vincoli di sincronizzazione locali alcuni framework ad attori hanno fornito altre astrazioni per lo scambio di messaggi e la sincronizzazione. Per esempio, sia Scala che ActionFoundry forniscono le join-expressions: che permettono ad un attore di rispondere ad un set di messaggi come se fossero uno solo. SALSA fornisce altre tre astrazioni per la sincronizzazione: token-passing continuations, join blocks e first-class continuations. Il primo è stato progettato per poter fornire un ordinamento parziale ai messaggi; in aggiunta un token può essere passato come parametro ad una funzione per ulteriori processamenti. I join block possono essere utilizzati per specificare una barriera alle attività concorrenti, in maniera tale da rendere disponibile il risultato di tutte le attività in un messaggio seguente; queste sono molto simili alle join-expression. Infine le first-class continuation consentono la delegazione

della computazione a terze parti, abilitando la sostituzione dinamica o l' espansione dei messaggi raggruppati per token-passing continuations.

Sia Killim che Jetlang hanno un modello di comunicazione channel-based. Killim fornisce come canali delle caselle di posta tipizzate; inoltre nel suo modello c'è solo un ricevitore per ogni canale. D'altro canto Jetlang fornisce un paradigma di per lo scambio di messaggi di tipo publish/subscribe per ognuno di questi canali. Un canale può avere più abbonati, così come un attore può abbuonarsi a più canali. Questa ultima caratteristica può sfociare nel fatto che più messaggi vengano processati simultaneamente dallo stesso attore, il che va contro la proprietà di incapsulamento dello stato degli attori.

Citiamo ora altri due paradigmi per esprime la coordinazione fra più attori che possono essere utili qualora si vogliano costruire programmi ad attori su larga scala. Il primo richiede l' utilizzo di entità note come sincronizzatori, che vincolano l'arrivo dei messaggi per un attore o per un gruppo di essi. Il secondo invece è l' introduzione di un Protocol Description Language che è in grado di esprime protocolli per lo scambio di messaggi fra gli attori complessi e quindi potenzialmente di poter vincolare tutti i messaggi agli attori che fanno parte di un protocollo specifico.

La tabella sotto riportata riassume quelle che sono le proprietà che ogni framework offre:

	SALSA (v1.1.2)	Scala Actors (v2.7.3)	Kilim (v0.6)	Actor Architecture (v0.1.3)	JavAct (v1.5.3)	ActorFoundry (v1.0)	Jetlang (v0.1.7)
Request-reply messaging	Yes	Yes	No	Yes	No	Yes	No
LSC	No	Yes	No	No	No	Yes	No
Join expressions	Yes	Yes	No	No	No	Yes	No

Figura 4.3: Tabella riassuntiva delle varie forme di astrazione proposte dai vari framework

4.3 Proprietà semantiche

Come già precedentemente detto alcune importanti proprietà del modello ad attori puro sono: incapsulamento ed esecuzione atomica dei metodi,

fairness e trasparenza alla locazione. Sebbene queste proprietà siano fondamentali per il modello, non tutti i linguaggi o framework basati su attori le applicano tutte e tre alla lettera; questo perché molto spesso si ottiene una implementazione più performante. Questo modo di procedere è conveniente per gli sviluppatori del linguaggio o del framework, ma lascia al programmatore l' onere di scrivere il proprio codice in maniera tale da non andare a violare quelle proprietà. Di seguito andremo ad enunciare ciò che tali caratteristiche implicano sia a livello di progettazione sia a livello di programma.

4.3.1 Incapsulamento

Nel modello ad attori ci sono due importanti requisiti per l'incapsulamento, che sono l' incapsulamento dello stato e lo scambio di messaggi sicuro. Per quel che riguarda l' incapsulamento dello stato si intende che un attore non può accedere direttamente allo stato interno di un altro attore; un attore può modificare lo stato di un altro solo inviandogli un messaggio. Questo viene molto utile perché ci consente di pensare, così come nella programmazione ad oggetti, al comportamento futuro di un attore come funzione del messaggio ricevuto e dello stato corrente, cose già dette ma che vale la pena sottolineare. Nel esempio in figura 4.4 è mostrato un semaforo che conta utilizzando Scala Actors. L' attore principale, in aggiunta all' invio del messaggio di enter() esegue enter() nel proprio stack. A causa della mancanza del rafforzamento del incapsulamento nella libreria, il codice viola la proprietà degli attori di non poter accedere direttamente lo stato di un altro. Come conseguenza, in una implementazione multi-threaded e a memoria condivisa del modello ad attori, due attori possono entrare in maniera concorrente all' interno della sezione critica e questo viola la semantica del semaforo. In Killim gli attori hanno un riferimento alle caselle di posta degli altri attori; ancora questo viola l' incapsulamento. I modelli che rafforzano l' incapsulamento dello stato lo fanno attraverso l' indirezione. Per quel che riguarda lo scambio sicuro di messaggi bisogna che questo abbia una semantica di tipo call-by-value. Questo potrebbe richiedere il fare una copia del messaggio anche su piattaforme con memoria condivisa. In Scala Actor, Killim, JavAct e Jetlang il messaggio trasporta il riferimento al suo contenuto nella memoria condivisa, introducendo così lo stato condiviso fra gli attori. Questi framework incoraggiano il programmatore ad usare

oggetti immutabili all' interno dei propri oggetti o a copiare esplicitamente gli oggetti all' interno del programma per evitare la condivisione dello stato. Un altra caratteristica importante dello scambio di messaggi sicuro è l' atomicità; infatti grazie a lei possiamo dire che un messaggio può arrivare ad un attore mentre esso ne sta già processando un altro, ma che esso non ha il diritto di interromperne l' esecuzione. Se questo non fosse vero non potremmo più affermare quanto detto precedentemente, ovvero che il comportamento di un attore è funzione dello stato corrente, perché processare il secondo messaggio mentre già si stava processando il primo potrebbe portare l' attore a cambiare stato. Tutto ciò renderebbe molto difficile ragionare sul comportamento del sistema perché messaggi inviati potrebbero portare a stati inconsistenti o erronei.

```
import scala.actors.Actor
import scala.actors.Actor._

object semaphore {
  class SemaphoreActor() extends Actor {

    ...

    def enter() {
      if (num < MAX) {
        // critical section
        num = num + 1;
  } } }

  def main(args : Array[String]) : Unit = {
    var gate = new SemaphoreActor()
    gate.start
    gate ! "enter"
    gate.enter
  }
}
```

Figura 4.4: Codice in Scala Actors del semaforo che conta

4.3.2 Fairness (Fair Scheduling)

Il modello degli attori assume una nozione di fairness (equità) che significa che ogni messaggio è consegnato all' attore destinazione tranne nel caso in

cui sia permanentemente disabilitato e che un attore non potrà mai essere permanentemente in starvation (aspettare di poter far qualcosa). Grazie alla fairness è possibile applicare un ragionamento di tipo modulare sulle proprietà di liveness di un programma ad attori. Per esempio se un sistema ad attori A è composto da un sistema ad attori B dove B contiene attori che sono permanentemente occupati, la composizione non intacca il progresso degli attori in A. Un esempio comune di ambito nel quale la fairness sarebbe gradita è nei browser. Infatti i problemi qui sono causati molte volte dal fatto che alcuni componenti di un browser sono composti da plug-in di terzi che, in mancanza di fairness, quando causano problemi mandano in crash l' intero browser. Scala Actors, ActorFoundry, SALSA e Actor Architecture assicurano uno scheduling equo degli attori, ma questa garanzia è limitata dai vincoli imposti dalla JVM e dalla piattaforma sottostante.

4.3.3 Trasparenza alla locazione

Nel modello ad attori l' attuale locazione di uno di essi non influenza quello che è il suo nome. Gli attori comunicano scambiando messaggi; ogni attore ha il suo spazio di indirizzi, che potrebbe essere completamente differente da quello degli altri. Gli attori che un attore conosce potrebbero essere sullo stesso core, sulla stessa cpu o su un altro nodo all' interno della stessa rete. Il naming trasparente alla locazione permette ai programmatori di non doversi preoccupare di quella che sia, in un determinato momento, la locazione fisica effettiva di un attore.

Poiché un attore non conosce lo spazio degli indirizzi degli altri attori, una desiderabile conseguenza della trasparenza alla locazione è l' incapsulamento dello stato. Essa inoltre facilita anche la migrazione degli attori da un nodo a un altro a runtime. A sua volta, la migrazione può consentire ottimizzazioni runtime per il bilanciamento del carico e tolleranza ai guasti.

La trasparenza alla locazione è supportata da SALSA, Actor Architecture, JavAct, ActorFoundry e Jetlang mentre in Scala Actors e Kilim il nome di un attore è un riferimento in memoria, rispettivamente all' oggetto che rappresenta l' attore (Scala) e alla casella di posta dell' attore (Killim).

4.3.4 Mobilità

La mobilità è definita come l' abilità della computazione di spostarsi da un nodo ad un altro. Esistono poi due tipi di mobilità: quella forte che consente di spostare sia il codice che lo stato dell' esecuzione, e quella debole che consente di muovere solo il codice. Nei sistemi ad attori la mobilità debole è utile per spostare un attore in stato di idle (come per esempio un attore che si è bloccato poiché ha la mailbox vuota), mentre avere la mobilità forte significa che ad un attore può risultare utile migrare mentre sta ancora processando un messaggio.

Poiché gli attori forniscono la modularità del controllo e l'incapsulamento, la mobilità viene abbastanza naturale nel modello ad attori. I linguaggi orientati agli oggetti possono consentire la mobilità a livello di oggetti, ma tutti i thread che lavorano su di un oggetto devono essere consapevoli della sua migrazione. Inoltre, quando lo stack frame richiede di accedere ad un oggetto su un nodo remoto, lo stack su cui viene eseguita l'esecuzione deve essere migrato sul nodo remoto e poi, una volta terminata, riportato indietro. A livello di sistema la mobilità è importante per la ridistribuzione del carico, riconfigurazione e tolleranza ai guasti. La mobilità debole è supportata da SALSA, Actor Architecture, JavAct e ActorFoundry, mentre la mobilità forte può essere fornita in framework come ActorFoundry che consentono di catturare il contesto corrente dell' esecuzione (continuation) e di rafforzare l' incapsulamento degli attori.

4.3.5 Discussione

La tabella (in figura 4.5) riassume le proprietà semantiche supportate da alcuni dei più popolari Actor frameworks nella piattaforma JVM. Alcuni framefork migliorano l' efficienza dell' esecuzione ignorando aspetti della semantica degli attori. Per esempio come abbiamo menzionato in precedenza, lo scambio di messaggi fra attori comporta mandare il contenuto del messaggio per valore. Nei linguaggi come C, C++, o Java, che possono avere modelli di aliasing arbitrari, mandare il contenuto del messaggio per valore implica fare una copia del messaggio per evitare qualsiasi condivisione involontaria tra gli attori. Copiare è una operazione dispendiosa, anche quando eseguita al livello di istruzioni native.

Le implementazioni come quelle di Kilim e Scala Actors, forniscono una semantica by-reference per lo scambio di messaggi e, quando richiesto, la-

sciano la responsabilità di fare una copia del contenuto del messaggio ai programmatori. Tale approccio crea per essi un doppio pericolo. Per cominciare, devono pensare allo scambio di messaggi nel modello ad attori, poi hanno bisogno di rivedere il loro progetto allo scopo di capire quale messaggio effettivamente ha bisogno di essere copiato, e alla fine, loro hanno bisogno di garantire che il contenuto del messaggio sia effettivamente stato copiato.

La tentazione di ignorare l' incapsulamento è forte nel caso di un framework ad attori al contrario di un linguaggio ad attori. Per esempio, allo scopo di garantire che un attore sia incapace di accedere allo stato di un altro direttamente, un linguaggio potrebbe fornire un'astrazione come un indirizzo di posta elettronica o un canale ma attuarlo usando referenze dirette nel codice compilato per efficienza. Questo è simile a come Java attua riferimenti agli oggetti per astrarre i puntatori. In un Actor-based framework, una simile astrazione è da risolvere a runtime, cosa che è relativamente inefficiente.

Anche la nozione di Fair Scheduling è sottile nelle implementazioni dell' attore . Bisogna notare che l'esecuzione di programmi ad attori è message-driven e che essi si suppone siano cooperativi. Comunque, nulla impedisce a un attore di eseguire un loop infinito, o di bloccarsi indefinitamente in I/O o una chiamata di sistema .

	SALSA (v1.1.2)	**Scala Actors** (v2.7.3)	**Kîlim** (v0.6)	**Actor Architecture** (v0.1.3)	**JavAct** (v1.5.3)	**ActorFoundry** (v1.0)	**Jetlang** (v0.1.7)
State Encapsulation	Yes	No	No	Yes	Yes	Yes	Yes
Safe Message-passing	Yes	No	No	Yes	No	Yes	No
Fair Scheduling	Yes	Yes	No	Yes	No	Yes	No
Location Transparency	Yes	No	No	Yes	Yes	Yes	Yes
Mobility	Yes	No	No	Yes	Yes	Yes	No

Figura 4.5: Tabella riassuntiva delle propietà semantiche che i vari framework attuano

Capitolo 5

Orleans

Orleans è un framework progettato da Microsoft che ha come obiettivo quello di rendere lo sviluppo e la progettazione di applicazioni cloud più agevole e fattibile anche ad utenti non esperti in questo ambito. Orleans si basa sul modello ad attori discusso in precedenza ma lo estende, aggiungendo alcune caratteristiche importanti che consentono di risolvere alcuni problemi tipici dei sistemi distribuiti. Ad esempio, nella classica architettura three-tier con front-ends stateless, middle-tier stateless e un livello di raccolta dati, vi sono limiti alla scalabilità dovuti alla latenza, ma anche il throughput del sistema è limitato dal fatto che il layer di raccolta dati doveva essere consultato ad ogni richiesta; per risolvere quest' ultimo problema di norma si aggiungeva un ulteriore livello, detto di chaching, fra il middle-tier e il livello dei dati.

Il modello degli attori offre già delle soluzioni soddisfacenti a questo tipo di problematiche; il problema però risiede nel fatto che le piattaforme utilizzate non offrono un livello di astrazione adeguato, costringendo il programmatore a doversi preoccupare della locazione degli attori, di dover gestire i fallimenti e di tutte quelle tematiche tipiche dei sistemi distribuiti che lo costringono a perdere tempo o comunque ad essere un esperto del settore.

Orleans, come già detto, cerca di trovare una soluzione a questi problemi. Innanzi tutto Orleans tratta gli attori come entità virtuali (chiamate Grains) e non come entità fisiche; con questo si assume il fatto che un attore esiste sempre, almeno virtualmente. Esso non può essere esplicitamente creato o distrutto. La sua esistenza va oltre quello che è il tempo di vita di una qualsiasi delle sue istanze in memoria e di conseguenza va oltre anche

il tempo di vita di qualsiasi server. In secondo luogo gli attori in Orleans vengono instanziati automaticamente: se non c'è un istanza in memoria di un attore al quale viene mandato un messaggio questo causerà il fatto che verrà creata una nuova istanza di quell' attore su un server disponibile; inoltre un istanza di un attore non utilizzata viene automaticamente eliminata dal sistema. Un attore non fallisce mai : se un server S smette di funzionare e viene mandato un messaggio ad un attore A che era in esecuzione su quel server, questo causerà il fatto che il sistema automaticamente genererà una nuova istanza dell' attore A su un server disponibile inviando il messaggio a quella. Questo consente anche di alleggerire quelle che sono le responsabilità del programmatore; infatti in questo scenario non deve essere il suo programma che si occupa di dover creare nuovamente attori che hanno fallito, ma ci pensa il sistema automaticamente. Questo allude anche al fatto che non è necessario conoscere l'effettiva locazione delle istanze degli attori all' interno del programma e questo lo semplifica notevolmente. Infine Orleans può creare automaticamente più istanze dello stesso attore stateless, consentendo così alla nostra applicazione di scalare in maniera automatica qualora ce ne sia il bisogno.

Nel complesso, Orleans offre agli sviluppatori uno spazio attore virtuale che, analogamente alla memoria virtuale, consente loro di invocare ogni attore del sistema, anche se non è presente in memoria. La virtualizzazione si basa su un riferimento indiretto che mappa da attori virtuali alle loro istanze fisiche che sono attualmente in esecuzione. Questo livello di astrazione fornisce il runtime con la possibilità di risolvere molti problemi principali dei sistemi distribuiti che devono altrimenti essere affrontati dallo sviluppatore.

5.1 Grains

I Grains sono l'entità computazionale che sta alla base di Orleans; tutto il codice scritto all' interno di Orleans risiederà dentro un Grain. Un sistema può averne in esecuzione più di uno contemporaneamente e essi non condividono né lo stato né memoria per comunicare; ma per interagire utilizzano solamente lo scambio di messaggi asincrono. Internamente essi hanno un solo thread e, prima di gestire una nuova richiesta, finiscono di elaborare quella corrente. Ciò è una sorta di limite perché non consente di usare modelli di concorrenza all' interno dei Grain, che viene supportata solo fra

Grain diversi. Questo rende inutile all' interno di Orleans tutti i meccanismi di sincronizzazione come semafori o lock poiché la comunicazione fra Grain diversi è asincrona e all' interno essi hanno un solo thread.

Continuando a parlare di comunicazione abbiamo visto che lo scambio di messaggi viene implementato attraverso la chiamata a dei metodi che, diversamente da quello che avviene nei sistemi RPC tradizionali che forniscono una semantica bloccante, ritornano subito con la promessa (promise) di un risultato futuro. Le promise 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. Il sistema da anche la possibilità, se ritenuto necessario dal programmatore, di adottare esplicitamente un comportamento bloccante, scegliendo di rimanere in attesa dell'arrivo di tale risultato.

Il sistema non impone un limite massimo o minimo alla dimensione dei grain che dovrà essere scelta in maniera opportuna dal progettista per trovare un giusto compromesso fra la concorrenza è lo stato necessario per 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.

Il non determinismo è una caratteristica intrinseca dei programmi realizzati tramite il framework Orleans, come d'altra parte lo è per ogni sistema distribuito sufficientemente complesso. Esso si esplica 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.

5.2 Grain interface

Un qualsiasi Grain deve implementare una o più interfacce pubbliche di grain che ne andranno a definire il contratto. Un interfaccia di grain è un interfaccia che aderisce alle seguenti regole:

1. Un interfaccia di Grain deve ereditare direttamente o indirettamente dall' interfaccia marker IGrain.

2. Tutti i metodi devono essere asincroni, ovvero ritornare o un Task o un Task <T> . Questo rende esplicita la natura asincrona della comunicazione fra grain, fa si che client e server possano utilizzare la stessa interfaccia e consente di poter far ritornare ad un grain una promessa unresolved invece di un valore concreto.

3. Non ci devono essere dei property setter, ovvero dei metodi che settano delle proprietà del grain, e neanche metodi di registrazione/deregistrazione ad aventi (questi perché sono sincroni in .NET).

4. I campi all' interno dei metodi devo essere del tipo di un interfaccia di Grain o di un tipo serializzabile che logicamente possono essere passati per valore.

Nella figura sottostante riportiamo un esempio di interfaccia di grain.

```
public interface ISimpleGrain : IGrain {
  AsyncValue<int> A { get; }
  AsyncCompletion SetA(int a);
  AsyncCompletion SetB(int a);
  AsyncValue<int> GetAxB();
}
```

Figura 5.1: Esempio legacy di interfaccia di un grain dove le promise erano ancora implementate come AsyncCompletion, il loro equivalente attuale sono i Task

5.3 Grain References

Una Grain Reference è un oggetto mediante il quale noi possiamo accedere ad un grain; questo implementa la stessa interfaccia di grain del grain al quale si riferisce. L'unico modo che ha un client (di qualsiasi tipo, potrebbe anche essere,per esempio, un altro grain) di poter accedere ad un grain è mediante questa reference. Inoltre esse sono degli oggetti di prima classe che possono essere passati come argomenti ai metodi. Infine hanno alcune similitudini con le promise di cui parleremo in seguito:

- Una grain references può essere in uno dei tre stati: unresolved, fulfilled o broken.
- Un chiamate può programmare o mettere in pipeline operazioni su una grain references prima che questa sia risolta invocando o un metodo asincrono o il metodo continueWith(). Inoltre viene supportato anche il metodo wait() che attende in maniera sincrona per la sua risoluzione.
- Le condizioni di errore vengono propagate alle azioni di continuazione (rompendo le promesse relative alle operazioni) oppure possono essere gestite in modo sincrono attraverso una chiamata al metodo wait ().

5.4 Usare e creare grain

Per ogni interfaccia di Grain il ClientGenerator tool di Orleans genera automaticamente una classe factory pubblica e una classe proxy interna che converte le chiamate a metodi in messaggi; gli utilizzatori per creare,cancellare o per trovare dei Grain, utilizzano la factory associata all' interfaccia del Grain che gli interessa. Quindi nel caso più semplice una factory contiene metodi per creare, cancellare e fare il cast di Grain, come mostrato nella figura 5.2.

Aggiungere delle notazioni opzionali all' interfaccia di grain come Lookup o Queryable, fa si che il ClientGenerator aggiunga dei metodi alla classe factory per la ricerca di un grain e per la ricerca di un grain che corrisponda a determinate caratteristiche .La figura 5.3 riporta un esempio di codice che crea un grain sul quale verranno poi eseguite alcune operazioni.

Il metodo CreateGrain() ritorna immediatamente una grain reference dandoci così la possibilità di poter chiamare su di essa le operazioni asin-

```
public class SimpleGrainFactory {
   public static ISimpleGrain CreateGrain();
   public static void Delete(ISimpleGrain grain);
   public static ISimpleGrain Cast(IGrain grainRef);
}
```

Figura 5.2: Metodo factory per un grain

crone setA e setB anche prima che il grain sia completamente creato. Il client chiama poi getAxB prima che i metodi setA e setB rendano fulfilled le loro promesse; questa invocazione è messa in coda al grain che la eseguirà subito dopo aver eseguito i due metodi setter. Quando sarà risolta la funzione getPromise() verrà invocata una funzione delegato o di successo o di errore.

Poiché ogni operazione asincrona, come la chiamata al metodo di un grain, una chiamata al metodo continueWith() su una promise, ritorna una promise e, essendo che le promesse propagano gli errori attraverso la continuazione, la gestione degli errori può essere implementata in maniera molto semplice. Nell' esempio precedente un errore in un qualsiasi punto del programma avrebbe fatto passare resultPromise allo stato broken e di conseguenza la chiamata al metodo wait() su di essa avrebbe generato un eccezione con informazioni relative all' errore generato. Tutti gli errori vengono così gestiti in un unico punto del programma e questo semplifica enormemente quello che è il codice per la gestione degli errori.

5.5 Classi di grain

Come già sottolineato precedentemente una classe implementa una o più interfacce di grain. Poiché in esse ogni metodo è asincrono, la corrispondente implementazione del metodo all' interno della classe deve ritornare una promise. Ci sono due possibili casi : il metodo può ritornare un valore concreto (che viene automaticamente convertito dal sistema in una promessa risolta dal sistema) oppure può ritornare una promessa che ottiene chiamando un

```
ISimpleGrain grain = SimpleGrainFactory.CreateGrain();
AsyncCompletion setAPromise = grain.SetA(3);
AsyncCompletion setBPromise = grain.SetB(4);

// join the promises
AsyncCompletion setPromise =
  AsyncCompletion.Join(setAPromise, setBPromise);
AsyncValue<int> getPromise = setPromise.ContinueWith(
    () => {
          return grain.GetAxB();
    });

// schedule action when GetAxB returns actual result
AsyncCompletion resultPromise =
   getPromise.ContinueWith((int x) => {
        Console.WriteLine("Result: " + x.ToString());
     },
     (Exception exc) => {
        Console.WriteLine("Error: " + exc.Message);
        throw exc; // re-throw the exception
     });
// wait for operation to complete
try {
    resultPromise.Wait();
} catch(Exception exc) {
   // error at any stage will throw exception
   Console.WriteLine("Error: " + exc.Message);
}
```

Figura 5.3: Come creare un grain

metodo di un altro grain. Nella figura 5.4 è fornito, in ordine, un esempio di queste possibilità.

5.6 Modello di esecuzione dei grain

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 activation.

Mentre turni relativi a diverse activations possono essere eseguiti in parallelo da Orleans, turni relativi alla stessa activation vengono invece eseguiti

```
AsyncValue<int> GetAxB() {
   int x = this.a * this.b;
   return x;
}

AsyncValue<int> GetAxB() {
   AsyncValue<int> p = anotherGrain.GetAxB();

   return p;
}
```

Figura 5.4: Classi di grain

in maniera sequenziale; è grazie a questo meccanismo che si rende l' esecuzione delle 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.

Con questo modello di esecuzione non vi è più necessità di utilizzare meccanismi quali lock e semafori per risolvere problemi relativi alla sincronizzazione. Tuttavia questo modello limita l' esecuzione parallela solamente a livello di insiemi di grain, quindi esclude il parallelismo dovuto alla memoria condivisa. Questa restrizione è stata fatta per semplificare la stesura del codice e per prevenire tutti quelli errori, molto frequenti nella programmazione parallela, che possono avvenire nel momento in cui abbiamo una memoria condivisa.

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

Generalmente è necessario che una activation 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: o marcando la classe del grain con l'attributo Reentrant, che permette ai

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

5.7 Activation

Come già sottolineato in precedenza al fine di avere un throughput maggiore nella gestione di un carico crescente sul sistema Orleans crea in maniera automatica più istanze di uno stesso Grain occupato per poter gestire più richieste contemporaneamente; queste istanza sono chiamate activation. Queste sono in grado di poter elaborare richieste, tra loro indipendenti e rivolte allo stesso grain, in maniera concorrente e anche se esse si trovano su server diverse. Questo riduce il tempo di attesa dei Grain più richiesti e, di conseguenza, migliora la scalabilità del nostro sistema. Se necessario il framework può anche richiedere macchine al cloud sottostante per mandare in esecuzione le nuove activation, liberando così il programmatore da questo compito.

Mentre i Grain rappresentano quindi l' astrazione logica a livello di programmazione, le activations rappresentano le unità di esecuzione a runtime. Ogni activation 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.

5.8 Promise

Orleans usa le promise come primitive per la programmazione asincrona. Una promessa ha un ciclo di vita semplice che si suddivide in pochi stati. Inizialmente essa è nello stato di unresolved, che significa che è nell' attesa di un risultato in un qualsiasi tempo futuro, quando il risultato viene ricevuto la promise passa allo stato fulfilled e assume il valore del dato appena ricevuto. Se per un qualche motivo capita un errore nel processamento della richiesta la promise passa invece allo stato broken e non assume alcun valore. Una promise sia che sia nello stato broken che nello stato fulfilled viene

considerata comunque resolved. Le promise sono implementate attraverso due classi .NET :

- *System.Threading.Tasks.Task* che rappresenta il futuro completamento di un'operazione
- *System.Threading.Tasks.Task <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 promise vengono fornite da Orleans anche di un metodo Wait() che si può chiamare sulle promesse stesse e che blocca l' esecuzione del Grain fino a che la promessa non diventa resolved. Esiste poi un altro metodo, getValue(), che fa sostanzialmente la stessa cosa di Wait() solo che in più ritorna anche il valore della promessa.

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 5.5 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 (anche qui essendo un esempio legacy le promise erano ancora implementate come AsyncCompletion, il loro equivalente attuale sono i Task).

Va infine detto che Orleans da la possibilità di creare una promessa unendone delle altre attraverso il metodo join(). La promessa composta sarà resolved quando tutte le promesse che la compongono saranno a loro volta risolte; se però anche solo una delle promesse che la compongono diventa broken allora anche la promessa composita passa in quello stato.

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

Figura 5.5: Esempio di promise

5.9 Transazioni

Le transazioni in Orleans servono a tre ruoli:

- Isolare le operazioni concorrenti l' una dall'altra.
- Assicurare che una operazione veda uno stato consistente dell' applicazione nonostante la distribuzione e la replica dei grain.
- Ridurre la necessità di una specifica gestione degli errori e recupero di essi.

Le transazioni in Orleans rispettano quelle che sono le proprietà ACID, ovvero hanno caratteristiche di atomicità, consistenza, isolamento, durabilità. Durante l' esecuzione una transazione vede solamente una singola activation per ogni grain coinvolto nella transazione, così ogni transazione vede uno proprio stato consistente dell' applicazione. Questo stato è isolato dai cambiamenti fatti da esecuzioni concorrenti di transazioni. L' update di una transazione in un archivio durevole, anche se esso avviene su più grain, diventa automaticamente visibile alle transazioni successive quando la transazione termina. Le transazioni attraverso più grain vengono automaticamente scritte in archivi durevoli fornendo un meccanismo affidabile per la persistenza del risultato di una computazione.

Il sistema delle transazioni opera tracciando e controllando il flusso di esecuzione attraverso le activations dei grain. Una transazione è creata all' arrivo di una richiesta da un cliente esterno al sistema. La transazione contiene tutte le attivazioni dei grain invocate dal processo richiedente, a meno che lo sviluppatore non specifichi dei limiti transazionali. Una transazione si dice completed nel momento in cui il processo richiedente finisce l' esecuzione, si dice invece committed nel momento in cui i cambiamenti sono scritti

in un archivio durevole. Il programmatore può scegliere, prima di poter vedere i risultati della transazione, se dover aspettare che essa sia completed (transazione ottimistica) o che sia committed (transazione pessimistica).

In caso di fallimento una transazione che era in esecuzione o una in stato completed può essere eseguita nuovamente prima che passi allo stato committ. La riesecuzione è non deterministica e può produrre risultati differenti. Se questo non è accettabile dalla nostra applicazione bisogna marcare la transazione come pessimistica.

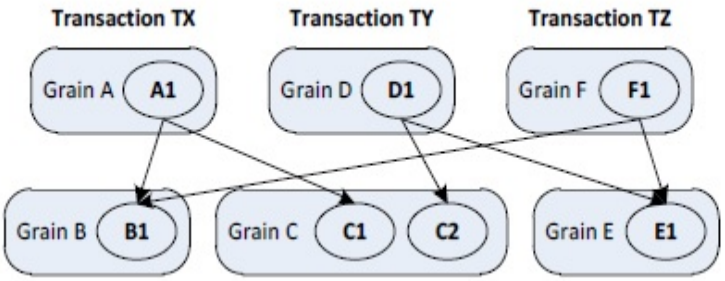
5.9.1 Isolamento

L' isolamento garantisce che una transazione non veda i cambiamenti apportati da altre transazioni in esecuzione concorrentemente ad essa fino a che queste non sono completate e viceversa. Per assicurare ciò Orleans mantiene una corrispondenza uno a uno fra le activations del grain e le transazioni read-write attive. Una activation parteciperà a non più di una transazione attiva, a meno che tutte le transazioni non siano di sola lettura. Si dice che una activation viene unita ad una transazione nel momento in cui riceve un primo messaggio all' interno della transazione. Una activation rimane unita alla transazione fino a che quest' ultima non completa.

5.9.2 Atomicità

Per conservare l' atomicità Orleans deve assicurare che l' aggiornamento di una transazione diventi visibile come un insieme completo o che non sia visibile per nulla. Per assicurare ciò l' ambiente mantiene la corrispondenza transazione/activation finché le transazioni non sono committed. Prima di unire una transazione ad una activation il sistema verifica che questa azione preservi l' atomicità. Ad esempio, in figura 5.6, una transazione completata TX ha modificato le activation A1, B1, C1 e, una transazione completata TY ha modificato D1, C2 e E1. Una transazione attiva TZ ha modificato le activation F1 e B1 e invia una richiesta al grain E. Se questo messaggio arriva all' activation E1, l' ambiente ha abbastanza informazioni per individuare una potenziale - ma non ancora effettiva - violazione del atomicità se TZ dovesse inviare un messaggio al grain C. Potrebbe scegliere di reindirizzare il messaggio a un altra activation del grain E. Oppure, se non è disponibile o è troppo costoso crearne una nuova, può andare avanti

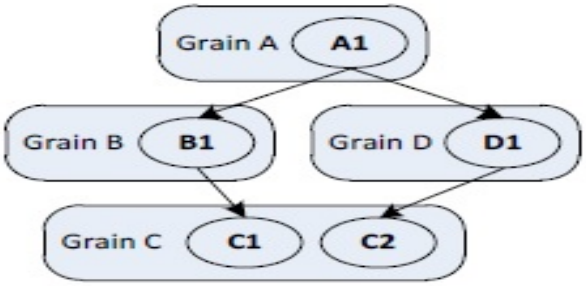
e unire l' activation E1 a TZ. Finora, l' atomicità è conservata. Tuttavia, se TZ successivamente invia un messaggio al grain C, l' ambiente non può scegliere nè l'attivazione C1 nè la C2 senza violare l'atomicità (di TY o TX, rispettivamente). Il sistema rileverà questo prima che il messaggio possa essere inviato al grain C e interrompa TZ, garantendo che nessun codice in esecuzione osservi una violazione di atomicità. Le transazioni TX e TY dovranno abortire e ripartire poichè i loro aggiornamenti a B1 ed E1 saranno persi quando TZ abortirà.



5.9.3 Consistenza

La consistenza viene specificata sia attraverso le transazioni che all' interno della transazione. In quest' ultimo caso la consistenza richiede che la sequenza degli stati delle activation osservate debba essere consistente con l' ordinamento parziale definito dal flusso di messaggi di risposta e di richiesta all' interno della transazione. Unire le activation alle transazioni assicura che ci sia una singola activation per grain in ogni transazione; questo consente di operare su una singola copia dello stato del grain. Mantenere questa proprietà è facile per applicazioni che eseguono in maniera seriale attraverso un insieme di gran. Ciascun messaggio di richiesta o di risposta contiene l'intero insieme di activation che sono state unite finora alla transazione. Ogni volta che all' interno di una transazione viene fatta una richiesta ad un grain X il sistema prende una activation presente nell' suddetto insieme. Qualora nessuna activation fosse stata unita finora alla transazione il sistema è libero di scegliere qualsiasi activation che non sia già coinvolta in nessun altra transazione. Comunque quando il sistema presenta richieste multiple e concorrenti è richiesto un meccanismo aggiuntivo.

Nella figura 5.7 l' activation A1 manda concorrentemente messaggi a D1 e B1 i quali a loro volta mandano concorrentemente messaggi al grain C. Orleans prova ad assicurare che entrambe le activation mandino i messaggi alla stessa activation di C senza usare un meccanismo di coordinazione distribuito, che sarebbe costoso e non scalabile. Se il meccanismo euristico fallisce e i grain scelgono activation diverse, per esempio C1 e C2, l' inconsistenza sarà scoperta quando arriveranno le risposte ad A1. A questo punto la transazione abortirà prima che qualsiasi parte dell' applicazione possa osservare l' inconsistenza fra lo stato di C1 e C2. Quando la transazione viene rieseguita viene notificata la causa del fallimento e il sistema selezione proattivamente una activation del grain C che unirà alla transazione prima di far ripartire il grain A. Questo previene il verificarsi della situazione precedente assicurando che D e B scelgano la stessa activation di C.



5.9.4 Durabilità

Orleans assicura anche che le transazioni committed siano scritte atomicamente in archivi persistenti. Il meccanismo di persistenza delle transazioni segue un approccio ottimistico, scrivendo in maniera asincrona i risultati modificati negli archivi senza cancellare una transazione in esecuzione. Quando la transazione finisce il server che ha inviato la richiesta iniziale manda una notifica di completamento all' archivio di sistema listando tutte le activations coinvolte nella transazione. Il committ di una transazione ha due fasi:

- l' archivio colleziona le rappresentazioni serializzate dello stato persistente di ogni activation nella transazione.

- gli stati dei grain vengono scritti negli archivi persistenti usando un commit a due fasi per assicurare che tutti gli aggiornamenti diventino visibili simultaneamente.

5.10 Gestione dello stato e persistenza

Lo stato di un grain viene gestito da Orleans per tutto il ciclo di vita dello stesso : inizializzazione, riconciliazione, replica e persistenza. Il programmatore identifica lo stato come persistente e Orleans gestisce il resto. Nessun codice è richiesto per rendere persistente o per caricare lo stato del grain. Orleans in sè non implementa alcun tipo di archiviazione; piuttosto si basa su un provider di persistenza esterno come il Windows Azure Storage di Microsoft.

Ogni grain dichiara le parti del suo stato che devono essere considerate persistenti usando le annotazioni .NET; queste devono essere in grado di supportare la serializzazione e possono comprendere dati, riferimenti a grain e promesse risolte. A livello di singolo grain queste forniscono un modello di persistenza semplice. Il sistema attiva un grain con le sue proprietà persistenti già inizializzate sia da parametri per l' inizializzazione del grain o dalla loro versione corrente in memoria persistente. In seguito viene chiamato il metodo Activate che permette di inizializzare la parte di stato non persistente. A questo punto il sistema può invocare i metodi per gestire le richieste inviate all' activation i quali possono operare liberamente sullo stato in memoria.

Capitolo 6

Esempi Applicativi

In questa sezione finale andrò ad analizzare quelli che sono due semplici esempi applicativi sviluppati dalla Microsoft di applicazioni sviluppate mediante il framework Orleans. La prima che analizzerò sarà una applicazione che realizza quello che è il gioco del tris e che verrà poi rilasciata sulla piattaforma Windows Azure. La seconda invece è una applicazione che, preso un account twitter, analizza quelli che sono gli Hashtag specificati e ne fa un analisi sull' opinione. In questo caso l' applicazione non verrà però rilasciata su alcuna piattaforma cloud ma verrà fatta girare in locale.

6.1 Tris

Questo esempio viene utilizzato per mostrare quanto possa essere semplice implementare un gioco a turni utilizzando Orleans, ma anche per mostrare come rilasciare un applicazione scritta utilizzando questo Framework all' interno della piattaforma Azure.

Ad un primo sguardo questo non sembrerebbe la classica applicazione realizzabile mediante Orleans, o per lo meno non sembrerebbe adattarsi a quelle che sono le sue caratteristiche. In realtà questo viene subito smentito nel momento in cui si pensa a questa applicazione in un contesto social, come può essere ad esempio un gioco per Facebook, dove centinaia di giocatori potrbbero potenzialmente giocare contemporaneamente dozzine di partite, alcune in attesa di un avversario, altre nel mezzo del gioco, in attesa della mossa del rispettivo avversario; tutto questo è un ambiente molto più congeniale e naturale per Orleans.

In questa applicazione vi sono tre grain. Uno che rappresenta il giocatore (IPlayer), uno la partita(IGame) e l' ultimo(IPairing) che serve per gestire una cache nella quale si vanno a salvare tutte quelle che sono le informazioni relative ai vari giochi ai quali un certo utente può partecipare.

Un istanza di PlayerGrain viene attivata ogni qualvolta un utente si logghi all' interno del sistema. Ad ognuno di esse viene assegnato un GUID (Globally Unique Identifier) che servirà per identificarli univocamente; questo poi verrà archiviato in un cookie dall' applicazione web MVC. Esso tiene traccia di tutte le informazioni relativa ad un giocatore quali ID e partite alle quali partecipa; le operazioni che un IPlayer può eseguire sono rappresentate in figura 6.1.

```
public interface IPlayerGrain : IGrain
{
    // get a list of all active games

    Task<PairingSummary[]> GetAvailableGames();

    Task<List<GameSummary>> GetGameSummaries();

    // create a new game and join it
    Task<Guid> CreateGame();

    // join an existing game
    Task<GameState> JoinGame(Guid gameId);

    Task LeaveGame(Guid gameId, GameOutcome outcome);

    Task SetUsername(string username);

    Task<string> GetUsername();
}
```

Figura 6.1: Interfaccia IPlayer

Mediante le prime due operazioni un Player quindi non fa altro che ottenere informazioni su quelle che sono le partite disponibili (attraverso la prima) e ottenere informazioni sui risultati delle partite da lui disputate (mediante la seconda), entrambe infatti ritornano una struttura dati contente tali informazioni. Un player può anche creare una nuova partita, ma anche entrare in una partita in attesa di altro giocatori, da notare come la prima operazione ritorni il GUID della partita (che potrà poi eventualmente essere passato ad un altro giocatore), mentre la seconda ritorni lo

stato della partita, con relative informazioni sul turno. Infine un player può lasciare una partita, quando essa sarà terminata, e può settare e ritornare quello che è il proprio nome.

Per quel che riguarda il grain associato alla partita, esso mantiene tutte le informazione relative alla partita stessa, quali i giocatori che vi partecipano, a chi sta di turno e se la partita è terminata o se invece vi sono ancora mosse da fare. Le operazione che esso può compiere sono rappresentate in figura 6.2.

```
public interface IGameGrain : Orleans.IGrain
{
    // add a player into a game
    Task<GameState> AddPlayerToGame(Guid player);
    Task<GameState> GetState();
    Task<List<GameMove>> GetMoves();
    Task<GameState> MakeMove(GameMove move);
    Task<GameSummary> GetSummary(Guid player);
    Task SetName(string name);
}
```

Figura 6.2: Interfaccia IGameGrain

Un game può aggiunge un giocatore (all' atto della creazione di una nuova partita un player si aggiunge automaticamente anche al game appena creato), può ritornare quello che è il suo stato e una lista di movimenti che sono stati effettuati. Può effettuare un movimento su quello che è il suo tavolo di gioco (in questa operazione si controlla anche se il movimento appena effettuato sia valido, se non abbia in qualche modo condotto alla fine della partita e , in caso lo fosse, si informano i giocatori mediante una chiamata al metodo LeaveGame()) , può ritornare un riassunto di quella che è stata la partita, quindi vincitore, perdente, mosse effettuate etc. e infine può settare il proprio nome.

Infine abbiamo in Pairing Grain, che come già detto, è un grain che svolge la funzione di accedere alla cache per salvare le partite visibili dai giocatori, l' interfaccia è quella in figura 6.3.

Le operazione che fornisce sono semplici: si può salvare un game nella cache salvandosi il proprio GUID e un nome, si può eliminare quella partita

```
public interface IPairingGrain : Orleans.IGrain
{
    Task AddGame(Guid gameId, string name);
    Task RemoveGame(Guid gameId);
    Task<PairingSummary[]> GetGames();
}
```

Figura 6.3: Interfaccia IPairing

(una volta terminata) utilizzando sempre il GUID della partita che si vuole eliminare. Infine si può ottenere quella che è una lista di tutti le partite correntemente disponibili. I dati presenti all' interno della cache hanno un expire time ti un ora, questo per limitare il numero di partite visibili, dall' utente che molto probabilmente non verranno mai iniziate.

Il programma funziona quindi come segue. Non appena un utente crea un gioco, quell' utente viene automaticamente aggiunto alla lista degli utenti che appartengono a quella partita e un' istanza che tiene traccia del fatto che ci sia una partita attiva viene salvata nella cache. Una volta che un secondo giocatore si aggiunge alla partita quest' ultima può iniziare e i due giocatori continueranno a giocare fino a che non terminerà o con un pareggio o con una vittoria. Una rappresentazione più dettagliata di quelle che sono le possibili interazioni è data dal diagramma UML in figura 6.4.

Le uniche cose da notare in questo diagramma UML sono il fatto che sui metodi per l' aggiunta di un giocatore ad una partita, del settaggio del nome della partita e per l'aggiunta o rimozione di un elemento alla cache si aspetti una risposta prima di poter proseguire con la computazione; in altre parole si blocca il grain tramite l' utilizzo del metodo wait. Questo ha senso nel primo caso poiché il task ritorna quello che è lo stato della partita, quindi non avrebbe senso salvare l' identificativo di una partita nella cache se poi in realtà non sono stato in grado di potermi registrare. Nel secondo caso si richiede questa cosa perché nella cache si salva anche il nome della partita, quindi prima di poterla salvare bisogna che il nome gli sia stato effettivamente assegnato per fare qualcosa di corretto. Nel terzo e ultimo caso invece ha senso perché, prima di poter eseguire un qualsiasi movimento sulla partita, è giusto aspettare che essa sia stata salvata correttamente in cache per non avere poi forme di inconsistenza.

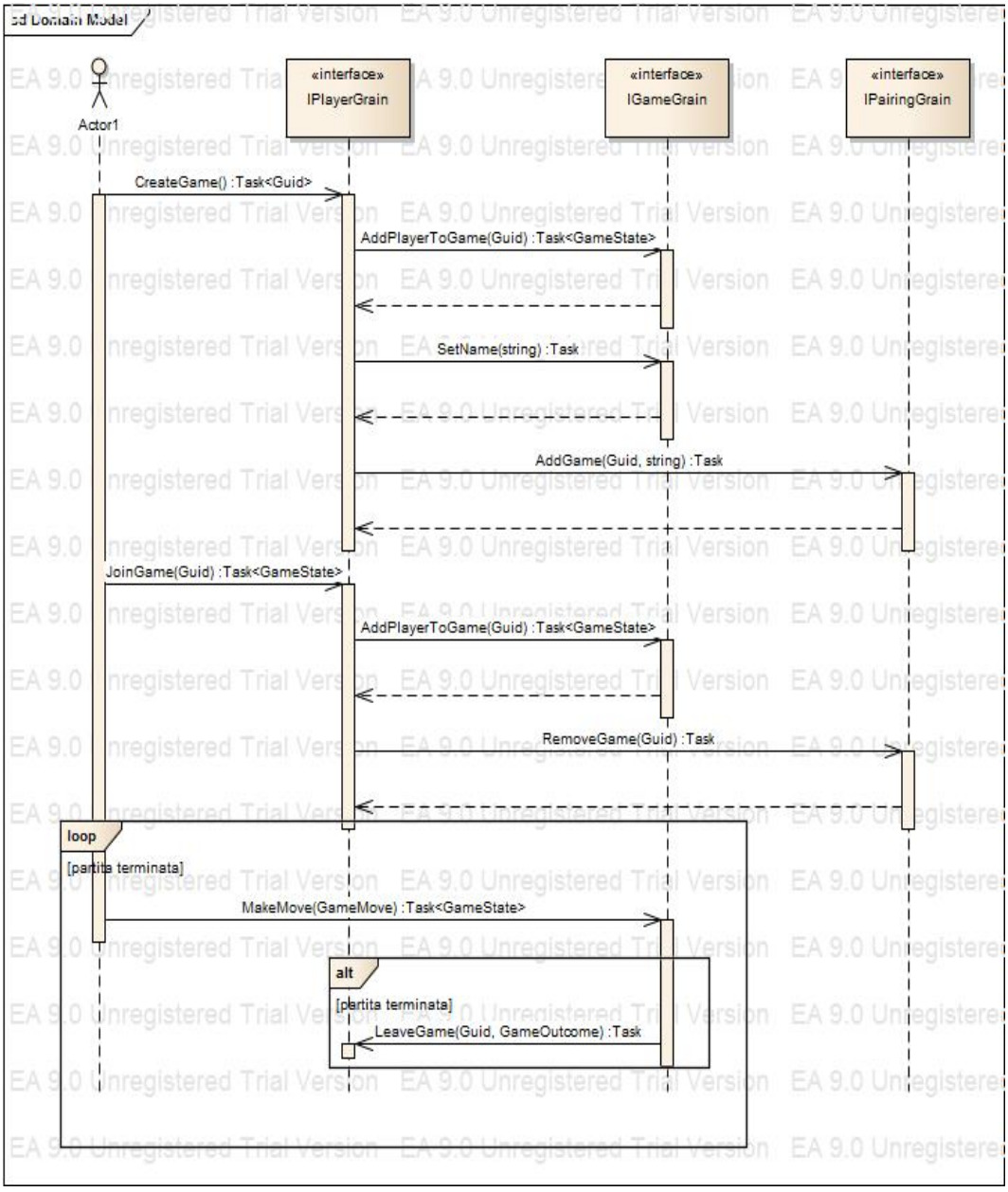


Figura 6.4: Diagramma UML di interazione Tris

Per quel che riguarda invece la distribuzione su Azure si è invece suddivisa l' applicazione in due ruoli. Uno un ruolo di tipo web che, come spiegato nel capitolo su Azure, si occupa della gestione dell' interfaccia grafica e del quale non mi preoccuperò, e un altro che invece è un ruolo worker, il cui compito è invece quello di eseguire la computazione e di andare a salvare tutti quelli che sono i dati importanti all' interno di un account di storage Windows Azure. Praticamente, una volta settata la stringa di accesso all' account di storage prescelto per ognuno dei due ruoli (stringa composta dal nome dell' account, relativa chiave di accesso e protocollo mediante il quale si fanno le richieste), si pubblica il progetto sulla piattaforma. Questa genererà automaticamente le macchine virtuali che gestiranno ognuna, come già detto, un solo ruolo e rilascerà un URL attraverso il quale ciascun utente potrà accedere all' applicazione, che viene di conseguenza rilasciata come servizio.

6.2 Twitter Sentiment

L' applicazione twitter sentiment registra le opinioni su Twitter consumando lo streaming fornito dalle API di Twitter, calcola un punteggio relativo all' opinione (ad esempio se viene utilizzato un linguaggio positivo, negativo o neutro), e poi registra questa opinione per ogni hashtag presente nel tweet usando un Grain Orleans per mantenere un punteggio totale per ogni singolo hashtag. I punteggi e tutte le altre informazioni utili all' applicazione vengono mantenuti all' interno di uno spazio di archiviazione Windows Azure, che viene gestito direttamente all' interno della stessa che, come già detto, non viene rilasciata come servizio cloud ma viene mandata in esecuzione localmente.

L' applicazione mostra come si possa utilizzare Orleans per manipolare dati con un grande traffico in input, dove ogni grain gestisce una riga in una tabella (o una chiave in uno spazio di archiviazione chiave/valore o un documento in uno a documenti) e tutti gli aggiornamenti passano attraverso un unico grain, riducendo la contesa per singole righe, chiavi o documenti.

Il calcolo dell' opinione per ogni tweet viene effettuato all' interno di una applicazione Node.js e usa una apposita libreria per fare il calcolo di quella che è l' opinione, per farlo si basa sulla presenza di certe parole chiave all' interno del tweet. Il punteggio relativo all' opinione e il tweet vengono

postati in un programma ASP.NET MVC per venire poi processati. Quest' ultimo ha due scopi fondamentali. Il primo è quello di agire come punto di arrivo per postare il punteggio dell' opinione su Orleans, e il secondo è quello di fornire l' interfaccia utente per visualizzare questi punteggi da Orleans. All' aggiornamento di un punteggio relativo a un opinione viene chiamato il metodo setScore nel GrainController che ottiene un riferimento ad un grain TweetDispatcher e chiama il suo metodo addScore. Il metodo getScore recupera il punteggio per un elenco arbitrario di hashtags, esprimendo il punteggio totale come una combinazione dei punteggi complessivi delle opinioni positive, negative e il numero totale di tweet processati contenti quell' hashtag. L' operazione di getScore viene effettuata ogni cinque secondi, o quando si inserisce un nuovo hashtag del quale ci interessa sapere l'opinione (si noti che se esso fosse già presente in tweet già esaminati si avrà subito un opinione come riscontro) oppure quando viene eliminato un hashtag dalla lista. Tenere traccia del numero di tweet totali contenenti un determinato hashtag serve perché l' analisi di un tweet spesso risulta avere un punteggio neutro e di conseguenza, senza questo dato, non si avrebbe una visione completa. Inoltre si tiene anche traccia del numero di hashtag distinti analizzati al fine di dare un idea della mole di dati processati.

Scendendo ora più nello specifico abbiamo che i grain presenti in questa applicazione, oltre al GrainController, che sostanzialmente fa da ponte fra l' applicazione Node.js e quella che è la logica applicativa e del quale ho già parlato prima, sono tre e sono un ITweetDispatcherGrain, un IHashtagGrain e un ICounter. Un ITweetDispatcherGrain può fare le operazioni riportate in figura 6.5.

```
[StatelessWorker]
public interface ITweetDispatcherGrain : Orleans.IGrain
{
    Task AddScore(int score, string[] hashtags, string tweet);

    Task<Totals[]> GetTotals(string[] hashtags);
}
```

Figura 6.5: Interfaccia ITweetDispatcherGrain

La prima cosa da notare di questo grain è il fatto che esso è un componente Stateless (lo si deduce dall' attributo StatelessWorker). Infatti

esso non deve tenere memorizzato nulla per poter procedere con la sua computazione, l' unico lavoro che fa è quello di smistare il lavoro verso IHashtagGrain. In particolare, mediante il metodo AddScore non fa altro che prendere un tweet in ingresso e smistare il lavoro ai grain che gestiscono gli hashtag contenuti al suo interno. Mediante il metodo GetTotals invece esso recupera tutte informazioni relative ad un insieme di hashtag (contenute all' interno del tipo di dato Totals che è stato definito dagli sviluppatori). Le informazioni contenute sono quelle che poi verranno mostrate all' utente e sono, oltre al numero di tweet positivi, negativi e neutri, anche l' ultimo update, l'ultimo tweet e anche il nome dell' hashtag stesso. Come detto questo grain scambia messaggi solo con gli IHashtagGrain che possono fare le operazioni riportate in figura 6.6.

```
[ExtendedPrimaryKey]
public interface IHashtagGrain : Orleans.IGrain
{
    Task AddScore(int score, string lastTweet);

    Task<Totals> GetTotals();
}
```

Figura 6.6: Interfaccia IHashtagGrain

Questa tipologia di grain viene utilizzata per tenere traccia di quello che è il punteggio di un determinato hashtag. Da notare inoltre l' attributo ExtendedPrimaryKey, che sta a significare che una stringa, in questo caso il nome dell' hashtag, viene utilizzata come chiave composita per il grain, al posto del più usuale GUID o long; ciò si adatta meglio a grain che, come in questo caso, rappresentano stringhe delle chiavi numeriche. Il metodo AddScore non fa nient'altro che aggiornare il numero di tweet positivi o negativi relativamente all' hashtag del quale si occupa, tenere traccia dell' ultimo tweet e di quando è stato tweettato. E' importante sottolineare che questi grain salvano sull' account di storage il loro stato solamente quando trovano un hashatg che ha un opinione positiva o negativa. Questo non influenza quanto fatto vedere a video durante l' esecuzione dell' applicazione, ma lo può influenzare al riavvio. Supponiamo ad esempio di aver appena individuato un hashtag, fra quelli che ci interessano, neutro e di non trovarne più nessun altro dopo di questo. Alla chiusura dell' applicazione, in

questo caso specifico, il dato andrebbe perso e ad una successiva riapertura avremmo che quell' hashtag è come se non lo avessimo mai trovato. Questo non è un particolare problema in quanto lo scopo della nostra applicazione è quello di fornire le opinioni positive e negative, è chiaro che però può essere una cosa fastidiosa. A favore di questa scelta va però detto che salvare lo stato di un grain in un database esterno è una operazione per lo meno dispendiosa quindi va fatta solo quando necessaria. Se poi consideriamo il numero di tweet esaminati dall' applicazione ci rendiamo conto che inizierebbe a diventare davvero un collo di bottiglia e quindi si è preferito scrivere su database solo quando necessario.

Il metodo getTotals invece non fa nient' altro che ritornare quelli che sono i dati relativi all' hashtag del quale si occupa nella struttura dati già definita in precedenza. Alla sua prima invocazione ogni IHashtagGrain manda un messaggio ad un ICounter che espone i metodi riportati in figura 6.7.

```
public interface ICounter : Orleans.IGrain
{
    Task IncrementCounter();
    Task ResetCounter();
    Task<int> GetTotalCounter();
}
```

Figura 6.7: Interfaccia ICounter

L' unico ruolo di questo componente è di tenere traccia del numero totale di hashtag. Anche qui per motivi di performance questo componente salva i sui dati sul database solamente ogni cento incrementi. Anche qui si da vita a tutti quelle problematiche di cui ho discusso precedentemente, ma tanto il numero di hashtag totali è solo una informazione in più che serve solo per dare un idea di quanti dati siano stati analizzati e come tale anche se non è assolutamente precisa non risulta essere un particolare problema. In figura 6.7 un diagramma UML che mostra l' interazione fra i vari componenti.

Questo diagramma mostra come su tutte le operazioni si applichi un wait, quindi prima di proseguire con la successiva si aspetti che quella precedente abbia terminato. Le interazioni dentro ai loop più interni in realtà non avvengono proprio come sono mostrare nel diagramma UML. Infatti

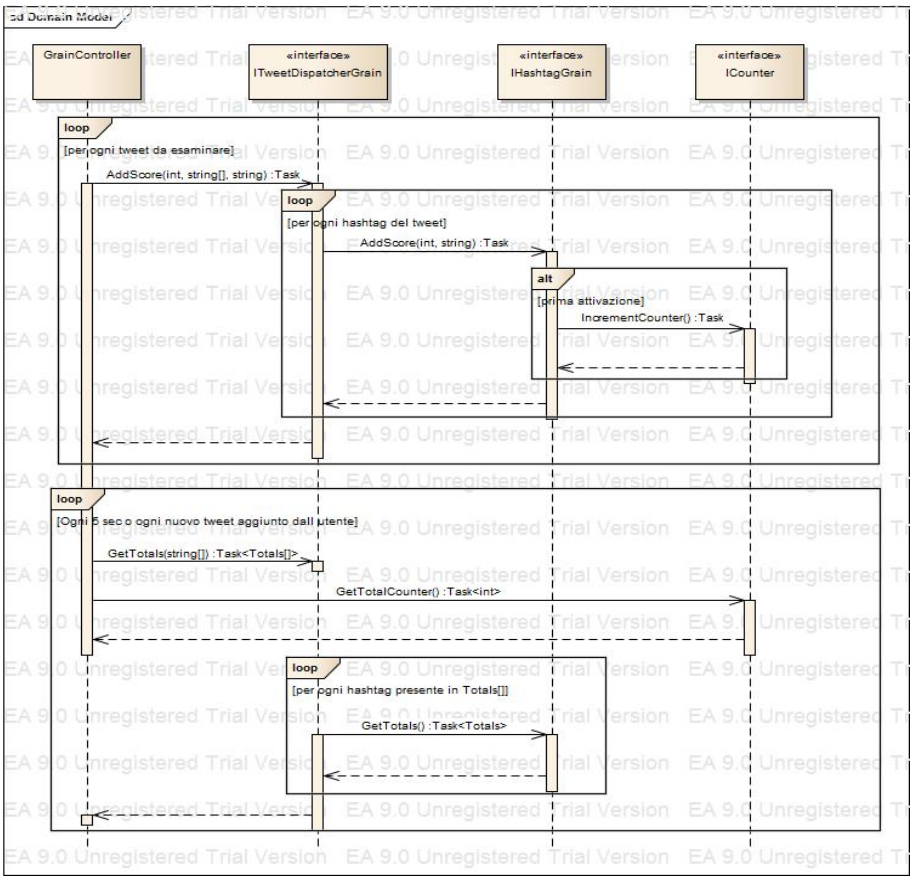


Figura 6.8: Diagramma UML di interazione

guardando il diagramma si evince che prima di poter eseguire l' operazione successiva si aspetti la fine di quella precedente. In realtà prima vengono tutte eseguite in maniera asincrona e poi, una volta eseguite tutte, si attende la loro risposta. Questo ottimizza le prestazioni rispetto alla situazione mostrata in figura. Purtroppo in UML rappresentare la situazione reale risulta complicato quindi, per far passare comunque il concetto di attesa, l' ho rappresentato come si vede in figura. La stessa considerazione la si può fare per le chiamate a getTotalCounter e getTotals, dove avviene la stessa identica situazione di prima. Applicare wait su tutte le operazioni riduce senza dubbio quelle che sono le performance, limitando la concorrenza, ma

ci da maggiore certezza su quello che avviene all' interno del programma e perciò si è scelto di andare in questa direzione quando si è progettata questa applicazione.

Capitolo 7

Conclusioni

Giunti al termine di questa tesi possiamo sicuramente concludere che, per le caratteristiche che mette a disposizione, il cloud computing acquisterà sempre una maggiore rilevanza in ambito informatico. In quest' ottica diventa quindi fondamentale per un informatico avere per lo meno delle basi su quelli che sono i principali pattern e piattaforme che al giorno d' oggi si utilizzano nel cloud; e questo è proprio quello che ho cercato di fare con questa tesi. Essendo che, come già citato, questo è un ambiente molto mutabile e che non gode ancora oggi di specifici standard, ho cercato di introdurre quelle che sono le metodologie e le piattaforme sulle quali, probabilmente, si andrà ad investire maggiormente negli anni futuri. Questo ovviamente potrebbe anche essere smentito, ma nel mondo dell' informatica bisogna sempre essere pronti al cambiamento.

Sicuramente questa incertezza porta anche a dei problemi, come ad esempio la portabilità di una applicazione da una piattaforma cloud ad un altra, operazione non sicuramente facile dato che, come visto anche nel corso della tesi, ogni piattaforma ha le proprie caratteristiche e i propri modi per rappresentare le cose. Fortunatamente questo problema viene mitigato dalla presenza di middleware come Cloudify, ma anche qui non è detto che questi supportino tutta la marea di piattaforme esistenti.

Parlando invece del caso di studio, ovvero Orleans, di esso si può sicuramente dire che è un framework abbastanza semplice, proprio come era negli obiettivi di Microsoft, che rende la programmazione di applicazioni cloud sicuramente più accessibile al grande pubblico. L' unica pecca negativa è il fatto che il codice scritto mediante Orleans è stato pensato per

essere eseguito sulla piattaforma Windows Azure di Microsoft, e questo ovviamente impone a coloro che vogliono utilizzare il framework di dotarsi di un account Azure. Tralasciando questa pecca l' ultima riflessione che faccio su Orleans è relativa a quello che secondo me lo rende veramente un framework vincente, ovvero il fatto che lascia veramente il programmatore libero di concentrarsi solamente sulla logica applicativa tralasciando gli aspetti più tecnici, che vengono gestiti in automatico. Un esempio lampante di ciò lo si può avere guardando il codice degli esempi applicativi (reperibile al sito *https://orleans.codeplex.com/SourceControl/latest*) dove non ci dobbiamo preoccupare di quale activation di un grain andiamo ad accedere, sceglie il sistema per noi, inoltre lo stato, qualora si tratti di un grain statefull, viene mantenuto consistente automaticamente dal sistema.

Quindi in conclusione l' ambiente cloud è un mondo sicuramente affascinante e che offre delle possibilità che lo rendono unico ma manca, almeno dal mio punto di vista, di una standardizzazione e di una definizione unica e condivisa che rende il tutto molto più confuso e meno fruibile a chi voglia approcciarsi ad esso. Inoltre mi sento di consigliare caldamente Orleans a coloro che, nonostante ciò che ho appena detto, vogliano iniziare a costruire applicazioni cloud in quanto, dal mio punto di vista, è veramente un ottimo framework.

Bibliografia

[1] Antonio Natali, Ambra Molesini, *Costruire Sistemi Software : dai modelli al codice*, Ottobre 2009. Esculapio.

[2] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, Peter Arbitter, *Cloud Computing Patterns*, Vienna, 2014. Springer.

[3] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, Jorgen Thelin, Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC 2011)*, ACM, Ottobre 2011.

[4] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, Jorgen Thelin, Orleans: A Framework for Cloud Computing, 30 Novembre 2010.

[5] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin, Orleans: Distributed Virtual Actors for Programmability and Scalability, 24 Marzo 2014.

[6] Rajesh K. Karmani, Gul Agha, Actor

[7] Rajesh K. Karmani, Amin Shali, Gul Agha, Actor Frameworks for the JVM Platform: A Comparative Analysis.

[8] David Chappell, Introducing Windows Azure, Ottobre 2010

[9] `http://www.getcloudify.org/guide/2.7`

[10] `http://www.apprenda.com/platform/features/`

[11] `http://www.en.wikipedia.org/wiki/Cloud_computing`