Alma Mater Studiorum
Università di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura

Corso di laurea in INGEGNERIA INFORMATICA, ELETTRONICA E TELECOMUNICAZIONI

Titolo dell'elaborato:

Coordinazione embodied vs. disembodied: TuCSoN on Cloud

Elaborato in:
Sistemi Distribuiti

Relatore
Prof. Andrea Omicini
Correlatore
Dott. Stefano Mariani

Presentata da
Luca Guerra

Sessione II - Anno Accademico 2012/2013

Alla mia famiglia, vicina in ogni momento.
Alla mia ragazza, inseparabile compagna di viaggio.
A mio cugino Majkol, guida e ispirazione.

Contents

Introduzione

1 Visione

La continua diminuzione del costo dei computer, ha portato ad un aumento esponenziale dei suoi utilizzatori. Questi sviluppano e sfruttano sempre nuove tecnologie e costrutti, portando ad una veloce evoluzione dei sistemi software. Questi, da semplici algoritmi, diventano un insieme di varie tecnologie che devono cooperare fra loro al fine di raggiungere un determinato scopo.

La mente umana ha bisogno di vedere i problemi al giusto livello di astrazione, quindi per attaccare dei sistemi software di una certa complessità, che vedono l'accesso alla rete e la coorperazione di varie entità, anche costruite con tecnologie differenti, ha la necessità di passare da strutture layerizzate chiamate middleware. Queste, aiutano l'essere umano a comprendere ed astrarre meglio un problema applicando il principio della *separation of concern*; in questo modo è possibile concentrarsi solo sulla problematica in sè, tralasciando dettagli tecnici non utili. Il cloud computing, così come la tecnologia TuCSoN, nasce proprio da questo principio, avere una struttura che risolva svariati compiti, facendo preocccupare l'utilizzatore solo dello scopo che deve raggiungere.

2 Obiettivi

L'obiettivo posto per la tesi è quello di portare TuCSoN su un'infrastruttura di Cloud computing, offrendo, in questo modo, la sua coordinazione non più come servizio fra normali host, ma come servizio cloud. Tramite questa

tecnologia, infatti, sarà possibile offrire a TuCSoN una maggiore affidabilità, scalabilità ed efficienza, senza considerare che sarà più semplice distribuirlo ed offrirlo a possibili Utenti.

3 Struttura tesi

La tesi vuole essere così divisa:

- **Chapter 1 *Cloud Computing***: Parlerò di questa nuova tecnologia, spiegando come nasce e perché sta avendo così tanto successo. Darò una panoramica sui modelli proncipali presenti nel Cloud Compunting e presenterò il consumer e il provider, le due figure principali in questo ambito.
- **Chapter 2 *TuCSoN***: Questo capitolo ha lo scopo principale di introdurre alla tecnologia TuCSoN, spiegando qual è il suo scopo e il suo ambito di utilizzo.
- **Chapter 3 *TuCSoN nel Cloud***: Si parlerà del modello pensato per attaccare il problema di una distribuzione nel Cloud di TuCSoN.
- **Chapter 4 *Cloudify***: Analizzerò la tecnologia Cloudify, in particolare cosa è stato utilizzato per lo svolgimento della tesi.
- **Chapter 5 *Progetto e caso di studio***: Proseguo logico e punto di incontro dei capitoli 3 e 4. Qui si vedrà lo sviluppo del sistema più in particolare ed infine si proverà a mostrare un possibile caso di utilizzo dell'infrastruttura creata.

Chapter 1

Cloud Computing

"L'informatica non riguarda i computer
più di quanto l'astronomia riguardi i telescopi."
- cit Edsger Dijkstra -

1 Visione

Il NIST (National Institute of Standards and Technology) definisce il Cloud Computing come:
Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Fondamentalmente il Cloud computing non inventa nulla di nuovo, cambia però il modo di approcciarsi ai problemi in termini di sviluppo, distribuzione, pagamento e utilizzo delle risorse computazionali.
I servizi Cloud hanno delle differenze sostanziali con i normali servizi di hosting, sono molto più dinamici e reattivi, più semplici da gestire e manutenere, grazie all'infrastruttura Cloud possiamo essere molto più elastici nella pianificazione di progetto e avere le giuste risorse per raggiungere i nostri scopi. Possiamo decidere di scalare in dipendenza di fattori non direttamente controllabili come il successo o meno del sistema, il tutto in maniera veloce e

semplice.
Questo scenario, ancora in pieno sviluppo, vede, all'interno del suo processo di creazione, fornitura, distribuzione e utilizzo, vari ruoli. Fondamentalmente, quando parliamo di Cloud, parliamo di un servizio e in quanto tale, vede la presenza di due figure principali, i *Cloud provider*, ovvero chi offre il Cloud e i *Cloud consumer* chi il Cloud lo utilizza. Più precisamente, quando parliamo di Cloud provider intendiamo chi, tramite abbonamenti o altre forme di pagamento mette a disposizione le proprie risorse computazionali sulla rete. Tra i principali Cloud provider troviamo: IBM, Microsot, Rackspace.
Chi invece il Cloud lo utilizza, traendo un qualche tipo di beneficio (economico, personale, ecc..) viene chiamato *Cloud consumer*, questi sulla base di un piano tariffario "consumano" le risorse, il tutto tramite una semplice connessione internet. In questo scenario, fondamentale è la connessione internet fra i provider e i consumer, la figura che si occupa di questo viene definita con il nome di *Cloud carrier*, esso abilita, gestisce e manutiene la comunicazione e la normale fruizione dei servizi Cloud.
Anche se il termine Cloud è utilizzato in svariati contesti, con significati anche diversi fra loro, possiamo, a seconda del grado di astrazione richiesto, distinguere tre tipologie di servizi di Cloud computing: *Infrastructure-as-a-Service (IaaS) , Platform-as-a-Service(PaaS) e Software-as-a-Service (SaaS).* Questi, offriranno ai *Cloud consumer* più o meno visibilità dell'infrastruttura Cloud utilizzata. Come sempre in informatica, avere un grado di visibilità molto approfondito della struttura, porta, si una maggiore potenza di gestione, ma può anche distogliere l'attenzione dallo scopo prefissato. Se una azienda vuole semplicemente distribuire un servizio sul Cloud, non è interessata, su quante macchine poggierà, sull'hardware che montano, o sapere come è il carico sulla rete, ma vorrà solo avere un certo grado di affidabilità del servizio, e se gli sarà necessario, incrementare la potenza della struttura su cui poggia.

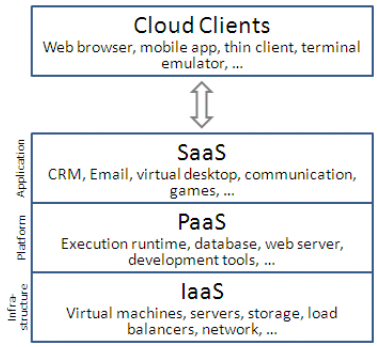

Figure 1.1: visione distribuzione Cloud

Nel Cloud le tre figure (provider, consumer e carrier) non sempre sono distinte, ma a seconda del tipo di distribuzione scelta, possono in qualche misura convergere anche ad una sola società. In particolare distinguiamo *Cloud pubblico*, *Cloud privato*, *Cloud ibrido*, le varie differenze fra queste figure verranno analizzate di seguito.

2 Modelli di servizio

2.1 Infrastructure-as-a-Service IaaS

Immaginando, i servizi Cloud disposti su vari livelli, a seconda del grado di astrazione, IaaS si posiziona sicuramente alla base della struttura. In questo modello viene data ai *consumer* una visione approfondita della struttura portante del Cloud. I *provider* devono garantire una certa affidabilità del servizio, che in questa ottica si traduce in una affidabilità dell'infrastruttura fisica delle risorse computazioni offerte sulla rete. I Consumers quindi non devono preoccuparsi di gestire o controllare l'infratruttura fisica offerta come servizio. Essi, avranno solamente l'onere di gestire le varie istanze di server virtuali assegnati (l'hardware dedicato, la velocità di connesione ecc..), i sistemi operativi che li popolano, ed in fine le applicazioni che sfruttano l'infrastruttura acquisita. Grazie a questo servizio, i *consumer* non devono preoccuparsi di acquistare, gestire e mantenere all'interno della propria impresa i vari server, le loro risorse computazionali saranno infatti servite tramite la rete da un provider, che garantirà un certo livello di affidabilità. La scelta di acquistare

un servizio IaaS risulta molto interessante per quelle aziende che non possono contare su grandi capitali, ma preferiscono magari, tramite un abbonamento pay per use, pagare a terzi per le risorse utilizzate.
Riassumendo l'obbiettivo del servizio è quello di provvedere una maggiore flessibilità, standardizzazione e virtualizzazione all'ambiente operativo che può diventare, anche una base per un PaaS o per un SaaS.

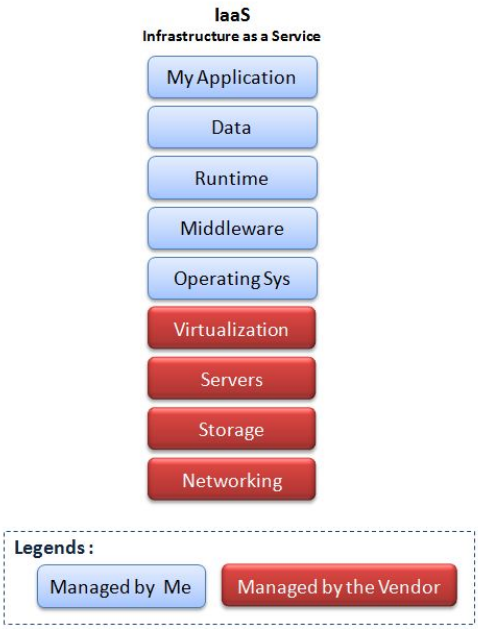

Figure 1.2: Infrastructure-as-a-Service (IaaS)

2.2 Platform-as-a-Service PaaS

Un servizio PaaS offre una piattaforma che poggia su un'infrastruttura virtualizzata. I consumer, ignorano completamente le tecnologie portanti il servizio, non hanno cognizione di quante e quali macchine, fisiche o virtuali, stiano sfruttando, di quale hardware o di quale sistema operativo regge il tutto. Il lato positivo di questo approccio è dato dalla quasi totale gestione dell'infrastruttura da parte dei *provider*. Questi, come nel caso precedente, garantiscono soglie di affidabilità del servizio ai consumer, i quali, possono concentrare le proprie

forze solo nello sviluppo delle applicazioni.
Più in particolare, la piattaforma offre servizi che vanno dallo storage, all'elaborazione dati, oltre che un insieme di tools, grazie ai quali i provider possono modificare le configurazioni della piattaforma stessa, adattandola così alle proprie esigenze.
Grazie a questi tools, i consumer possono avere controllo, mediato dai provider, sull'infrastruttura reggente; ad esempio sarà possibile, scalare la propria applicazione, richiedendo più o meno risorse computazionali. Il servizio PaaS è adatto ad aziende di piccola media dimensione, le quali non avendo grandi disponibilità di budget, personale e spazi, traggono il massimo vantaggio da questo tipo di servizio, potendosi così concentrare solamente sul loro obiettivo finale.
Un ramo di questo tipo di servizi, sono le piattaforme OPaaS. Queste piattaforme, sfruttate per portare sistemi non scritti per il mondo Cloud, oppure scritti per diverse tipologie di Cloud, sopra una qualsiasi PaaS, permettono di testare e sviluppare a costo zero, le proprie applicazioni, prima di inserirle nel Cloud.

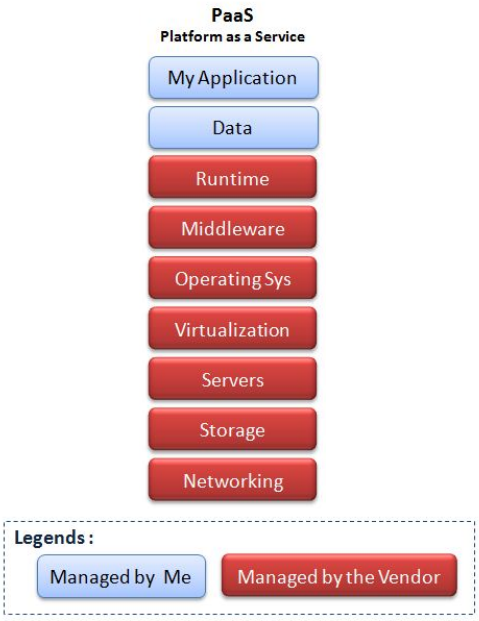

Figure 1.3: Platform-as-a-Service (PaaS)

2.3 Software-as-a-Service SaaS

In questa tipologia di servizio, i *provider* offrono direttamente degli applicativi software distribuiti sul Cloud. I *consumer*, attraverso una qualche forma di pagamento, hanno la capacità di utilizzare il software offerto come servizio Cloud, sfruttandone le varie caratteristiche, senza avere nessuna percezione o controllo dell'infrastruttura reggente il software.
Questo tipo di software è raggiungibile da svariati client. Esempi possono essere: servizi di mailing, servizi di desktop virtuale, servizi di storeage, videogames o anche semplici applicativi di comunicazione. Per quanto riguarda la gestione, l'unica cosa che i consumer possono gestire sono le varie configurazioni, permesse dai provider, inerenti il software utilizzato.

Figure 1.4: Software-as-a-Service (SaaS)

3 Modelli di distribuzione

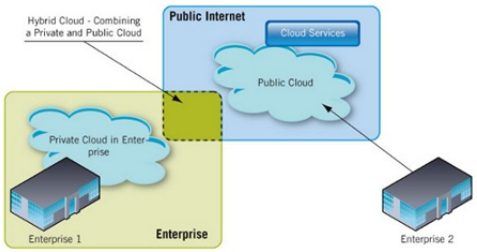

Figure 1.5: Struttura di distribuzione del Cloud

3.1 Public Cloud

Nell'ambito di Cloud pubblico, vediamo i *Cloud provider* offrire, in una determinata modalità, un servizio ai *Cloud consumer*, questi pagano per avere accesso a un certo numero di risorse computazionali offerte con una certa qualità di servizio.
Fra i principali provider di servizi troviamo Microsoft, IBM, Amazon. Per utilizzare questi servizi è sufficiente una connessione internet, e una volta registrato un account, solitamente tramite carta di credito, è possibile sfruttare i servizi e la varie modalità ad esempio con un abbonamento *pay per use*.
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. Inoltre, il Cloud pubblico viene criticato per il livello di privacy offerto, effettivamente, i dati inviati ad un Cloud pubblico, non saranno custoditi fisicamente in un luogo controllabile, ma saranno esterni all'impresa o comunque non sotto il controllo diretto del consumer. Tutto questo, anche se i provider garantisco una certa soglia di sicurezza nella gestione dei dati, limita la sicurezza sia dei sistemi, sia dei dati sensibili

contenuti in essi.

Per aziende che possono permettersi investimenti maggiori, una soluzione migliore sarebbe sicuramente quella del Cloud privato, nel quale mantenere dati sensibili, applicazioni critiche e applicazioni che devono essere controllate con grande efficienza, magari affiancato ad un Cloud pubblico, che rende il tutto molto più flessibile, manutenibile e garantito.

3.2 Private Cloud

Nel Cloud privato i servizi sono forniti da elaboratori che si trovano all'interno dell'azienda. In questo scenario le figure di *Cloud provider* e *Cloud consumer* si fondono, l'azienda stessa, offre e sfrutta il servizio Cloud, il quale sarà sotto il suo totale controllo bypassando abbonanmenti o qualsiasi forma di contratto con terzi.
Attravero questa distribuzione, è possibile avere pieno controllo sulle macchine all'interno delle quali vengono conservati i dati ed eseguiti i processi, avendo un elevato grado di sicurezza e gestione.
Nel momento in cui un'impresa decide di adottare un'infrastruttura di Cloud privato, trasforma le proprie risorse IT, in servizi, le rende efficienti, flessibili e dinamiche, ottenendo oltre che un vantaggio economico, anche un vantaggio in termini di manutenibilità e affidabilità.
Grazie al Cloud privato, a differenza del Cloud pubblico, è possibile sfruttare applicazioni aziendali che richiedono tecnologie o servizi presenti all'interno dell'azienda e che hanno bisogno di funzionare anche in assenza di connettività verso l'esterno. Inoltre, dato che certe risorse hanno necessità di essere scalabili ed elastiche appena i servizi lo richiedono, il Cloud privato consente di avere elasticità immediata per supportare le attività aziendali.
Il Cloud privato, è da preferirsi a quello publico, sia per l'ammortizzamento dei costi, per l'efficienza e la sicurezza del sistema. Tutto questo però ha un costo, l'impresa deve avere alte disponibilità economiche e di spazio per i vari server e deve avere personale dedicato all'infrastruttura, inoltre non può avere certezze di scalabilità elevate, ovvero, le risorse computazionali da lei acquisite saranno comunque limitate.
Come detto in precedenza, per avere una massima efficienza, sia economica

sia computazionale, si dovrebbe ricorrere ad un tipo di Cloud denominato ibrido.

3.3 Hybrid Cloud

Come dice il nome stesso, i Cloud ibridi sono costituiti da servizi Cloud privati (interni all'azienda) e pubblici (esterni). Si cerca in questa visione, di unire i lati positivi di entrambe le ditribuzioni, per trarre il massimo guadagno. Nello specifico, il Cloud pubblico è utilizzato per gestire i picchi di utilizzo, potendo sfruttare un'infrastruttura composta sicuramente di svariate macchine fisiche, mentre per quelle applicazioni che richiedono livelli superiori di sicurezza o privacy è opportuno sfruttare il Cloud privato, mantenendo il tutto all'interno dell'azienda.
La visione più sofisticata del Cloud computing ibrido, prevede un ambiente unico, i cui confini di Cloud pubblico e privato si confondono e in cui applicazioni e attività si muovono liberamente nel sistema, in base ai carichi di lavoro e alle esigenze economiche.

Chapter 2

Coordinazione come servizio

"Mentre in fisica devi capire come è fatto il mondo,
in informatica sei tu a crearlo.
Dentro i confini del computer, sei tu il creatore.
Controlli, almeno potenzialmente, tutto ciò che vi succede.
Se sei abbastanza bravo, puoi essere un dio.
Su piccola scala."
- Linus Torvalds -

In riferimento all'articolo *Coordination as a Service: Ontological and Formal Foundation* [1], in questa sezione verrà presentato il modello di coordinazione offerta come servizio, dando prima un modello formale che ne descriva le componenti fondamentali, poi descrivendo la sua applicazione in TuCSoN.

1 Visione

La coordinazine vede la luce in un periodo storico nel quale è stata naturalmente concepita come linguaggio. Prendendo ad esempio il modello LINDA, questa nella sua prima versione era concepita per sistemi chiusi, costituiti da applicazioni parallele, in cui l'obbiettivo era quello di ottimizzare la velocità di elaborazione.
In questo tipo di sistemi le entità coordinate erano conosciute a design time, il media di coordinazione era intrinsecamente legato all'applicazione coordinata.

Inoltre data la staticità dei sistemi, le entità di coordinazione, potevano essere costruite una volta per tutte in fase di compilazione in maniera specifica. Questo tipo di scenario viene descritto con il termine di *Coordination as a Language*.
Al giorno d'oggi con l'aumentare della complessità ed eterogeneità dei sistemi, si richiede, che il coordination media diventi un'astrazione di prima classe, interessandosi del processo di ingegnerizzazione, portando alla distribuzione di infrastrutture che provvedono a fornire coordinazione, al fine di gestire features come gestione a run-time, adattamento dinamico e un miglioramento progressivo.
Nella pratica, tutto questo viene tradotto in un Middleware, che fornendo il giusto livello di astrazione permette di parlare non più di una coordinazione as a language, ma di una coordinazione offerta come servizio.

2 Modello sistemi coordinati

Un sistema che adotta un modello di coordinazione viene chiamato *sistema coordinato*. Esso è un agglomerato di varie tecnologie ed entità che cooperano al fine di raggiungere uno scopo comune.
In questa tipologia di sistemi possiamo distinguere tre spazi separati: *spazio coordinato*, *spazio di coordinazione* e *spazio di interazione*, questi rappresentano concettualmente, tutte le entità e le astrazioni che ricoprono un qualche ruolo nel processo di coordinazione.

Un sistema coordinato, viene per sua natura diviso in tre spazi, essi vanno a comporre l'architettura logica del sistema, offrendo una base concettuale da cui partire per sviluppare sistemi di coordinazione reali.
In particolare definiamo:

Spazio coordinato: In questo spazio, sono concettualmente presenti tutte le entità coordinate del sistema. Quando parliamo di entità coordinate, ci riferiamo a tutte quelle attività computazionali governate da un modello di coordinazione.
Il comportamento di ogni entità può essere descritto come, l'invio di richi-

este (*coordination request*), l'apertura verso il sistema all'ascolto di risposte (*coordination reply*) e da un proprio calcolo interno (*silent action*). In generale, queste entità vengono viste dal sistema all'interno dello stesso spazio coordinato, ma esse non hanno, almeno per quanto riguarda la formulazione del modello, coscienza diretta delle altre. Ogni agente vede il sistema come un luogo in cui rilasciare richieste per ricevere poi coordinazione, quindi la comunicazine diretta non è contemplata nel modello, ma solo una forma di comunicazione indiretta mediata dallo spazio di interazione. Verrà chiamata *situated entity*, una entità coordinata indentitfica in maniera univoca, in modo che possa accedere ai vari servizi di coordinazione offerti dal sistema. Ovviamente, dato che il sistema cooordinato è per propria natura aperto, non è dato sapere a priori il numero di coordinabili, i quali potranno entrare a far parte e uscire dal sistema in qualsiasi momento. Per finire, quindi, lo spazio di coordinazione viene visto come un'insieme di entità coordinate, ogniuna con una propria identità, sfruttato dalle attività di coordinazione al fine di tener traccia degli agenti che hanno emesso richieste o che aspettano risposte.

Spazio di interazione: Lo spazio in questione, viene visto dai coordinabili come l'ambiente, esso ha il compito di gestire, mediare, le richieste e le risposte delle varie entità coordinabili. In pratica, rende possibile la comunicazione fra spazio coordinato e spazio di coordinazione, materializzando (o reificando) le varie azioni compiute dalle entità nel sistema, in eventi di comunicazione (*request events* o *reply events*), che verranno poi consumati dalle varie attività di coordinazione presenti nel sistema. Questo spazio è composto da un insiseme di *coordination media*, essi vengono visti come astrazioni che trasportano attività di coordinazione. Quando i coordination media vengono associati ad un certo sotto-insieme di coordinabili, essi rappresentano una certa topologia a run-time. Inoltre, è assunto un certo meccanismo di filtraggio il quale permette al medium di coordinazione di consumare eventi in accordo con una certa *matching condition*. Una volta consumato l'evento il medium di coordinazione produrrà uno o più eventi, i quali verranno reificati nello spazio di interazione. Questi eventi potranno essere *coordination reply* dirette ad un certo coordinabile, oppure potranno essere altri *coordination request* che interesseranno altri media.

Grazie a questo spazio, è possibile avere un disaccoppiamento temporale, si possono inserire richieste, che verranno elaborate appena il destinatario è libero, inoltre si disaccoppia mittente e destinatario, i quali, comunicheranno sempre con lo spazio di interazione. Mittente e destinatario non sono obbligati a conoscersi, devono solo preoccuparsi di svolgere i propri compiti, questa proprietà è fondamentale all'interno di un sistema aperto.

Spazio di coordinazione: L'elaborazione delle *coordination request*, è gestita in questo spazio. Esso è costituito da un certo numero di attività di coordinazione, le quali nell'insieme costituiscono il medium di coordinazione. Questo, ricevere attraverso i *coordination media* le varie richieste dallo spazio di interazione, le consuma emettendo sullo spazio di interazione un certo numero di eventi. Teoricamente, questi eventi possono essere ulteriori richieste per altri coodination media, formando una catena di chiamate, vista agli occhi dei coordinabili come un'azione atomica.

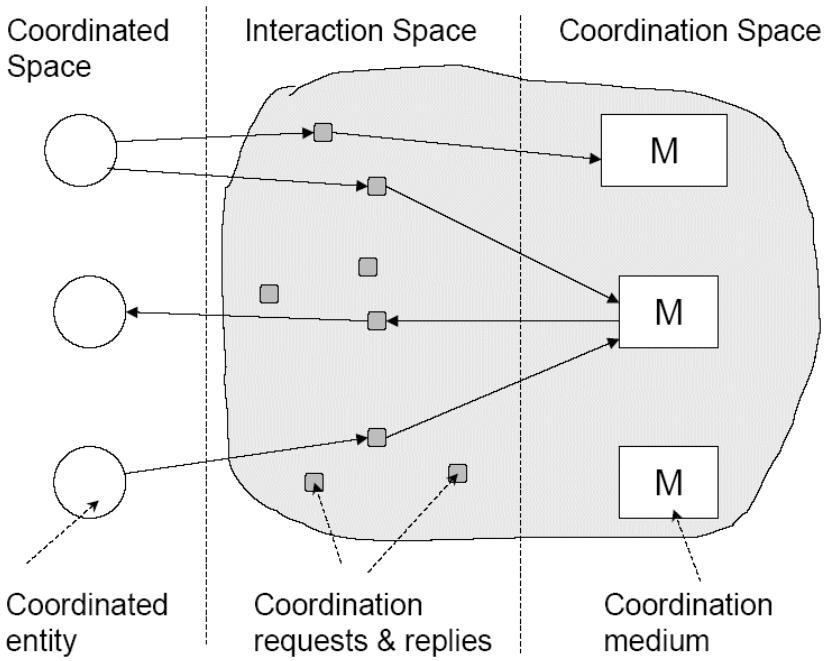

Figure 2.1: Architettura logica

3 Modello di coordinazione TuCSoN

TuCSoN (Tuples Centers Spread over the Network) è un modello e una tecnologia [2], realizzata dal gruppo di ricerca aliCE di Bologna e Cesena. Questa, estendendo ed ereditando aspetti del modello di coordinazione *Linda*, coordina processi distribuiti sulla rete e agenti autonomi, intelligenti e mobili.
Il modello fonda le sue radici sull'astrazione di *centro di tuple*, ovvero uno spazio di tuple programmabile, dove la programmazione avviene tramite apposite tuple, chiamate di specifica, scritte in linguaggio ReSpecT.
Questi centri di tuple sono distribuiti all'interno di nodi sparsi sulla rete, utilizzati dagli agenti per coordinarsi. In pratica, un sistema TuCSoN può essere visto come un insieme di agenti e centri di tuple che interagiscono in un possibile scenario di nodi distribuiti sulla rete.

Definiamo, specificatamente gli elementi principali, costituenti il modello TuCSoN:

Agente TuCSoN: Rappresenta, in un sistema coordinato, l'entità coordinata, descritta come attività pro-attiva ed intelligente, la cui interazione è gestita sfruttando l'invio e la ricezione di tuple, attraverso delle primitive TuCSoN. Questa, come detto in precedenza, ha determinate caratteristiche, in particolare: ogni agente gode di un certo livello di *autonomia*, infatti lavora senza un controllo continuo umano, ha cognizione del proprio stato e può prendere decisioni sulle proprie sue azioni. Ogni agente ha *abilità di comunicazione*, tramite la quale comunica con il mondo esterno, è in grado di *reagire* in risposta a stimoli esterni, ha la capacità di *prendere iniziative* sulle proprie azioni. In oltre, ogni agente gode di un certo grado di mobilità, non è legato ad uno specifico device, ma nel corso della sua vita può passare da device a device.

Per quanto riguarda il naming, come detto in precedenza, quando un coordinabile entra in un sistema coordinato, per essere considerato situato, gli deve essere associato un identificativo univoco, in TuCSoN questo è chiamato uuid (universally unique identifier). Questo completerà il nome assegnato all'agente secondo la logica Prolog *aname:uuid*.

Centro di tuple ReSpecT: Elemento concettuale fondamentale per TuCSoN, grazie ad esso è possibile separare il comportamento di un agente dalla sua coordinazione, rendendo il sistema più elegante e semplice da capire. Fondamentalmente un centro di tuple, funge sia da spazio di interazione, sia da spazio di coordinazione. I centri infatti possono essere programmati tramite ReSpecT e reagire a determinati eventi. La mobilità di ogni centro è legata allo specifico device a cui è collegato.
Nodi TuCSoN: Ogni sistema TuCSoN è caratterizzato prima di tutto da un insieme di nodi, possibilmente anche sparsi sulla rete, che ospitano un servizio TuCSoN. Questa astrazione di tipo topologico, funge da contenitore per i vari centri di tuple. Ogni nodo può essere raggiunto tramite l'IP dell'host che lo ospita alla porta a lui assegnata, la quale, nel caso non venga dichiarata, è di default la 20504.

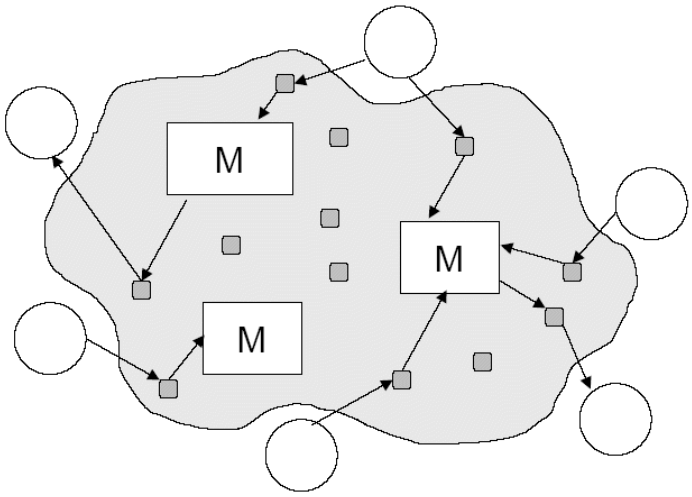

Figure 2.2: Topologia

4 Linguaggio di coordinazione

TuCSoN, fornisce un linguaggio di coordinazione[2] definito da un insieme di primitive di coordinazione, tramite le quali gli agenti possono interagire con i vari centri di tuple.
Ogni agente, viene fornito con un certo set di primitive. Tramite esse, può leggere, scrivere, consumare le tuple del centro e sincronizzarsi con esso. Il linguaggio di comunicazione è costituito da un *tuple language* e da un *template language*. Per realizzare la coordinazione, viene invece sfruttato il centro di tuple ReSpecT . Ogni operazione di coordinazione è divisa in due fasi:

- *invocazione*: Invio della richiesta, contenente tutte le informazioni dell'invocazione verso il centro di tuple, da parte dell'agente.
- *completamento*: Il risultato dell'operazione richiesta al centro di tuple ritorna all'agente, includendo tutte le informazioni riguardanti l'operazione eseguita.

Vi sono primitive sincrone o asincrone, differenziate con un comportamento bloccante oppure no nell'attesa delle completition. La sintassi astratta per definire un'operatione *op*, su un certo tuple center *tcid* è la seguente: *tcid ? op* in cui tcid è il nome completo del centro di tuple, formato dall'aggregazione di più informazioni. La sintassi completa per richiamare una determinata operazione su un certo nodo nella rete, è la seguente:

tname @ netid : portno ? op

La frase ha il seguente significato: realizza l'operazione *op* (una primitiva TuCSoN) sul centro di tuple di nome *tname*, presente all'interno del nodo in ascolto sull host *netid*, alla porta *portno*. Nel caso in cui non venga specificata la porta in ascolto, si ipotizza che si voglia parlare con il nodo default di un certo host, che per convenzione giace alla porta 20504.

5 Architettura

5.1 Topologia

Nel suo più alto livello di astrazione, un sistema TuCSoN è visto come insieme di *Nodi*. Questi Nodi offrono i loro servizi tramite un host collegato alla rete, sono quindi identificati da un certo IP (quello del device ospitante) e porta ai quali offrono i loro servizi. Nulla vieta che sulla stessa macchina siano presenti più Nodi TuCSoN, ovviamente collegati a porte diverse.
All'interno di ogni Nodo, troviamo una collezione di vari *centri di tuple*, questi rappresentano lo spazio di coordinazione del sistema TuCSoN. Essi sono caratterizzati da un certo *tname* univoco. In questo caso, appena avviene una richiesta di coordinazione verso un certo nodo, senza indicare il nome di centro di tuple, in automatico viene creato un centro di tuple di nome default. Ogni centro conterrà le varie tuple utilizzate per la coordinazione dei vari agenti TuCSoN appartenenti al sistema. Ogni agente, facente parte del sistema può comunicare tramite un linguaggio di coordinazione composto da varie primitive.

5.2 Spazio di coordinazione

Lo spazio di coordinazione in TuCSoN viene definito, a seconda del tipo di visione, in due diversi modi[2]:
Spazio di coordinazine globale: definito in ogni instate dall'insieme di tutti i centri di tuple disponibili sulla rete, ospitati sopra un nodo e identificati dal loro nome completo.
Spazio di coordindazione Locale: definito in ogni istante come l'insieme di tutti i centri di tuple, ospitati nei nodi di un certo host.

5.3 Mezzo di coordinazione

I centri di tuple, sono visti dall'esterno come normali spazi di tuple, i vari agenti comunicano direttamente con loro e grazie a loro si coordinano, sfruttando quindi i centri come *media di coordinazione*. Il comportamento di ogni singolo centro di tuple può essere definito separatamente e indipendentemente da qualsiasi altro centro di tuple, in base agli specifici compiti di

coordinazione scelti. Il comportamento di uno spazio di tuple è naturalmente definito come la transizione di stato osservabile dopo un evento di comunicazione. La definizione di un nuovo comportamento per un centro di tuple, fondamentalmente equivale a specificare una nuova transizione di stato in risposta ad un evento di comunicazione standard. Ciò si ottiene consentendo la definizione di reazioni (reaction) di comunicazione attraverso un linguaggio di specifica [3].
Più precisamente, un linguaggio di reazione è associabile a qualsiasi delle primitive di base di TuCSoN (out, in, rd, inp, rdp) per specificare attività computazionali, queste saranno chiamate *reaction*. Esse sono definite come un insieme di operazioni non bloccanti. Una reaction eseguita con successo può atomicamente produrre effetti sullo stato del centro di tuple, mentre il fallimento di una reaction non produce alcun risultato.
In linea di principio, ogni reazione può far scatenare altre reazioni, generando in questo modo una catena di eventi che agli occhi del chiamante, è vista in maniera atomica. Nel caso in cui la catena di eventi dovesse fallire in uno qualsiasi dei suoi passaggi, tutte le modifiche effettuate nello spazio di tuple dovranno essere eliminate, in modo da portare il sistema nello stato post-reaction. Il chiamante quindi vedrà semplicemente fallire la sua chiamata, mantenendo il sistema consistente.

5.4 Agent Coordination Context

L'Agent Coordination Context (ACC) può essere visto come l'astrazione con la quale si modella un agente "agli occhi" del coordination medium [2].
Ricopre lo stesso ruolo dell'interfaccia nei sistemi ad oggetti, fornendo una disciplina per l'interazione e qualche modello per la gestione di controllo nei sistemi ad agenti. Un ACC fornisce un disaccoppiamento tra gli agenti e il loro ambiente, grazie a ciò ogni agente può essere progettato e sviluppato in mariera del tutto indipendente dagli altri elementi del MAS (MutilAgent System).
Dal punto di vista dell'agente, esso dovrebbe fornire aspetti di interazione e comunicazione tramite un'astrazione che racchiuda anche nozioni di località nello spazio e nel tempo, visti gli scenari applicativi di oggi dovrebbe anche consentire agli agenti di percepire lo spazio in cui interagiscono in ter-

mini di effetti delle loro azioni e comunicazioni ed eventualmente, influenzare l'ambiente per raggiungere i propri obiettivi.
Dal punto di vista del progettista un ACC dovrebbe fornire un quadro complessivo per esprimere l'interazione all'interno di un MAS, definendo l'insieme delle interazioni ammissibili, gli agenti coinvolti e l'ambiente MAS. Dovrebbe anche incapsulare le regole per le applicazioni che disciplinano i sistemi ad agenti, e mediare le interazioni tra gli agenti e l'ambiente.

Nella sua implementazione attuale, l'astrazione dell'ACC si divide fra Node side e Agent side, rispettivamente:
Node side: si occupa di gestire le varie comunicazioni provenienti dall'esterno, comportandosi da Proxy nei confronti del Nodo.
Agent Side: può essere descritta sfruttando la metafora della sala comandi, ovvero: quando un agente entra in un nuovo ambiente, gli viene assegnata una sua sala comandi, questa rappresenta il suo unico modo per percepire e interagire con il nuovo ambiente. All'interno di questa sala, vi saranno *luci* per input discreti presi dall'ambiente e *schermi* per input continui, mentre per quanto riguarda l'interazione, ogni agente avrà a disposizione *bottoni* e *telecamere* rispettivamente per, output discreti e continui. Quali comandi siano presenti nella sala comandi e come sia possibile utilizzarli, fa parte della configurazione della sala comandi, che appare agli occhi dell'agente come una sorta di interfaccia. Questa configurazione, nasce da una negoziazione preliminare l'assegnazione della sala, questa poi nel corso della vita di un agente potrebbe anche cambiare a seconda delle necessità o delle interazioni del sistema.

Il rapporto che nasce fra i due lati dell'ACC, è un rapporto 1-n, ogni nodo può avere n agenti con cui interagire, mentre ogni agente deve avere un solo ACC per nodo, anche se nulla vieta che possa avere più ACC per comunicare con diversi nodi. Parlando di permessi e diverse tipologie di contratti stipulati fra Nodo e Agente, in TuCSoN sono presenti varie astrazioni di ACC, ogniuna delle quali rappresenta un diverso set di interazioni possibili.

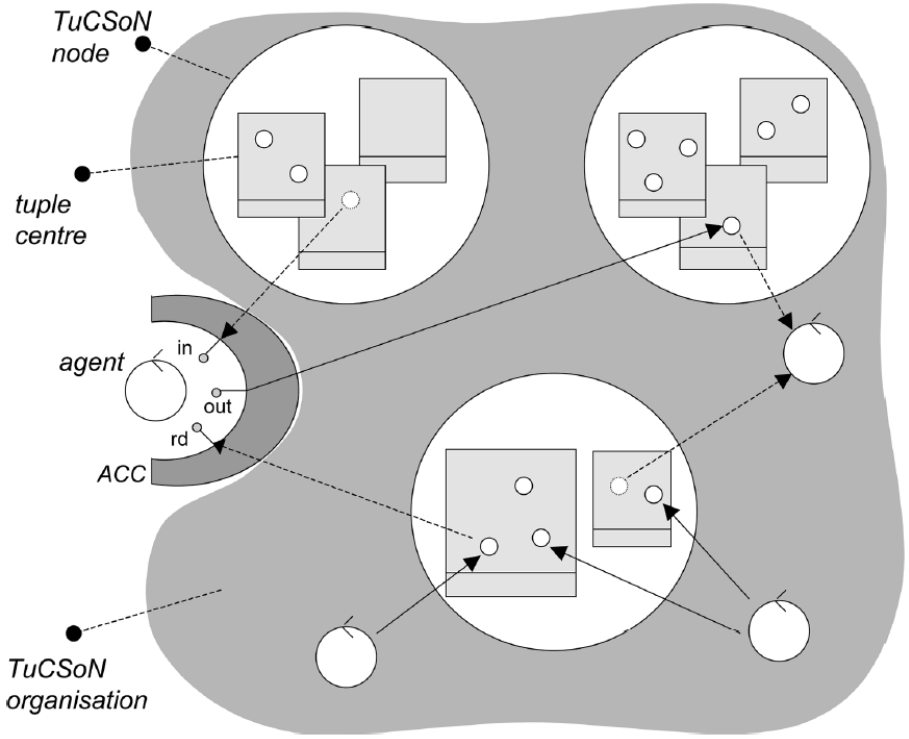

Figure 2.3: Agent Coordination Context

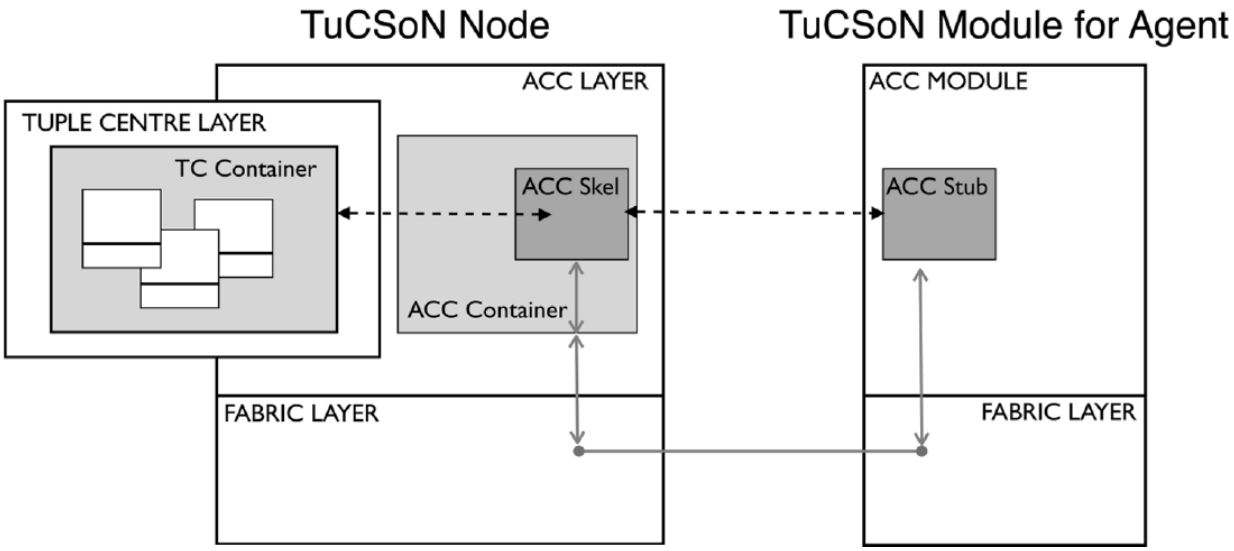

Figure 2.4: Topologia ACC

Chapter 3

TuCSoN on Cloud

"Il computer non è una macchina intelligente
che aiuta le persone stupide, anzi è una macchina stupida
che funziona solo nelle mani delle persone intelligenti."
- Umberto Eco -

1 Visione

Viste le caratteristiche di un sistema coordinato, si può pensare di portarle all'interno di un servizio Cloud. In questo modo diventa possibile distribuire la coordinazione in un ambiente gestito da terzi, il consumer non deve aver installato sulle proprie macchine, nessun software particolare dedicato, basta che abbia una connessione alla rete. Viene definito quindi un nuovo modello paragonabile a IaaS, PaaS e SaaS, che verrà identificato con il nome di CaaS (Coordination-as-a-Service). In quest'ottica, attraverso ad esempio un'abbonamento pay-per-use, sarà possibile comprare il servizio di coordinazione, e sfruttarlo nelle modalità richieste dai vari consumer, limitando e offrendo più o meno risorse, sia computazioni che di coordinazione.
Il modello TuCSoN, sembra sposarsi alla perfezione con l'idea di Cloud, esso offre già un servizio sulla rete, questo però deve essere manutenuto e gestito, solitamente questa è una cosa che i consumer preferiscono evitare. Inoltre offrendo questo tipo di servizio, si crea un luogo concettualmente disembodied, staccato dalla fisicità di un server fisico e situato, consegnando nelle mani dei consumer, un ambiente potenzialmente onnipresente di coordinazione, al

quale un agente TuCSoN può fare riferimento per compiti di coordinazione di alto livello, non direttamente collegati con l'ambiente nel quale si viene a trovare.

2 Modello di base

Gli spazi presenti in un ambiente coordinato, devono essere in quest'ottica divisi fra *on Cloud* e *out Cloud*. Le componenti "out Cloud" vengono viste come parti proprie dei *consumer*, mentre le parti "on Cloud" devono essere create, gestite e manutenute, con un certo grado di affidabilità dai *provider*. Prima di tutto, separando le entità già presenti in TuCSoN, notiamo come, lo spazio coordinabile sia competenza dei vari consumer, mentre lo spazio di interazione e coordinazione, rimangano competenza dei provider.
Più in particolare, il consumer si dovrà occupare di gestire i propri agenti TuCSoN, essi subiranno la coordinazione da parte dei nodi comprati, tramite un qualche tipo di contratto, sul Cloud. I provider dovranno, gestire, manutenere e organizzare tutte le risorse di coordinazione dei vari user, offrendo a seconda del tipo di abbonamento, un certo livello di servizio. Tradotto in TuCSoN, i Nodi e i relativi centri di tuple dovranno essere provider side.

Per completare il quadro generale, come risorsa offerta ai consumer dai provider, vedremo il nodo TuCSoN al centro della scena. Esso infatti, funge da astrazione topologica contenente i vari centri di tuple. Anche se nel Cloud, si perde di concretezza topologica, rimane comunque l'astrazione concettuale di contenitore di tuple center, studiando poi, l'infrastruttura TuCSoN si è notato che mantenendo questa entità il trasporto del modello sul Cloud risulta più veloce e compatibile con la versione di TuCSoN distribuita normalmente nella rete.

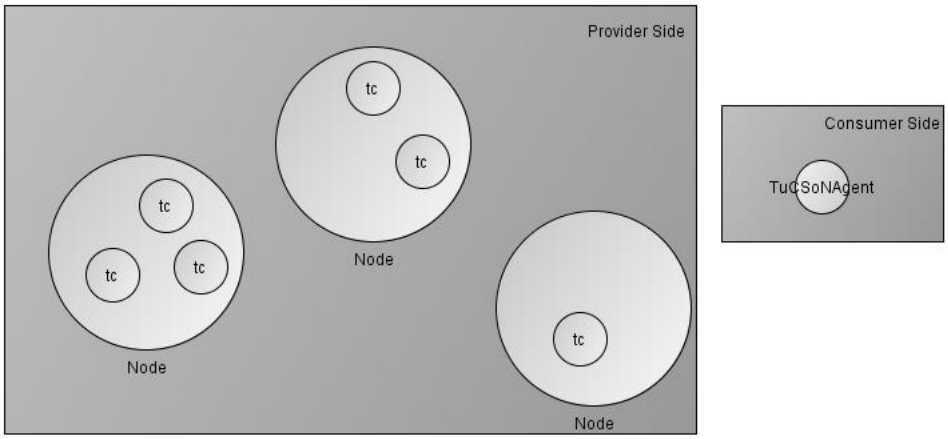

Figure 3.1: Visione base

3 Consumer Side

Il consumer, nell'ottica del CaaS, viene visto come l'ente che compra, attraverso un qualche tipo di contratto, nodi on Cloud e tramite essi coordina un certo numero di agenti TuCSoN. Gli unici compiti, che un cosumer deve svolgere, sono:

richiedere un account questa operazione dovrà essere realizzata tramite un CLI che consenta, ad utenti umani, di interagire con il provider, il quale tramite la registrazione può identificare lo user.

richiedere risorse Cloud sempre tramite CLI, il consumer, richiederà le varie risorse al Cloud.

utilizzare le proprie risorse questa fase, avviente tramite gli agenti TuCSoN del consumer. I quali, un volta ottenuto il giusto ACC, si comporteranno come una normalissima distribuzione TuCSoN.

3.1 TucsonCLICloud

Il consumer, deve poter comunicare con il proprio node on Cloud, questa comunicazione avviene sulla rete, secondo un preciso protocollo di messaggi. Per dare un accesso veloce e controllato al proprio nodo sul Cloud, verrà sfruttata un'entità che chiameremo *TucsonCLICloud*. Questa, fungerà da Proxy

per il consumer, nascondendo la complessità della comunicazione e offrendo un'interfaccia di comandi.

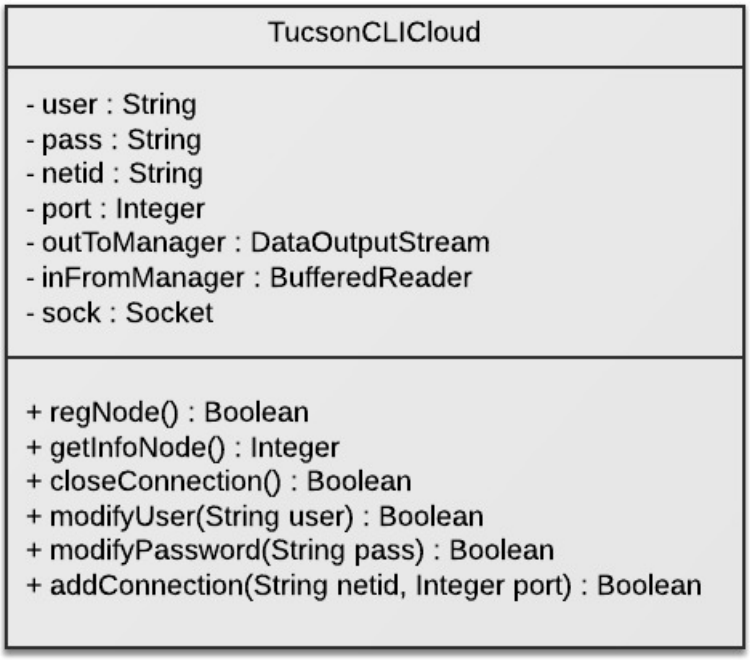

Figure 3.2: TucsonCLICloud

4 Provider Side

Il provider in quest'ottica è visto come il gestore delle risorse nel Cloud.
Esso avrà dei compiti principali:

creare nodi quando i vari consumer richiedono di voler prendere parte, secondo una certa modalità, all'ambiente di coordinazione Cloud.
accogliere gli agenti quando un agente vuole prendere parte delle risorse di coordinazione, comunica con il provider identificandosi, provando di essere un agente di un certo consumer. Il provider, o meglio delle sue entità avranno il compito di accertare l'appartenenza e di fornire le giuste informazioni per poter accedere alle risorse di coordinazione.
Manutenere i nodi il provider ha il dovere di manutenere ed organizare nel migliore dei modi, tutte le risorse presenti sul Cloud del sistema, garantendo una buona soglia di garanzia del servizio. Tutto questo controllando che i server virtuali siano sempre attivi, duplicando nel caso di failure l'istanza di un certo nodo, in modo che questo rimanga, anche se virtualmente, sempre attivo.

4.1 Node Manager

A supporto del provider di servizio, servirà un'entità specifica, che chiameremo *Node Manager*. Questa, fungerà da server, prenderà tutte le richieste dei vari consumer, seguendo un certo protocollo di comunicazione e svolgendo a seconda del tipo di richiesta, e del tipo di utente i compiti descritti in precedenza.
La figura del Node Manager, vista in maniera astratta, può essere paragonata a quella di un receptionist, il quale accoglie i vari consumer nell'atto della registrazione dell'account, e poi conduce i relativi agenti, dando le giuste informazioni sul nodo cercato.

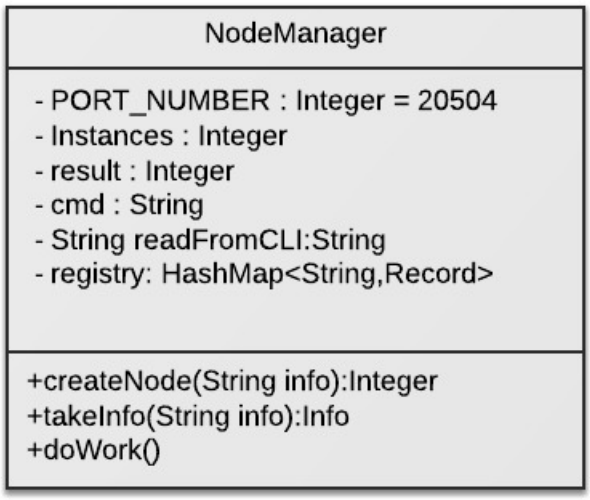

Figure 3.3: NodeManager

5 Dinamica

La dinamica di un sistema CaaS, si divide in due parti principali. La prima fase, sarà inerente l'acquisto, da parte di un certo consumer, delle risorse a lui necessarie per la coordinazione del suo sistema. Nel nostro modello, vediamo il nodo TuCSoN come elemento di base della vendita. Sarà questo l'ambiente in cui far "vivere" i vari centri di tuple. A seconda del tipo di abbonamento/contratto, che viene stipulato, un agente potrà avere a disposizione, più o meno risorse nel suo nodo. In pratica, ogni consumer, potrà far interagire un numero limitato di agenti con le risorse Cloud e un numero limitato di tuple center inseribili e utilizzabili nel nodo.
La struttura Cloud, ci offre la possibilità aggiuntiva di offrire, su richesta, più o meno risorse computazionali al nodo. In questo modo, un consumer che per un certo periodo dovrà sfruttare il nodo in modo intenso, potrà richiedere più risorse (più processori dedicati, più memoria RAM, più spazio fisico) scalando il sistema in maniera veloce e "indolore". Questa prima fase, verrà realizzata tramite un programma CLI, il quale, sfruttando l'entità *TucsonCLICloud*, potrà comunicare, sfruttando un certo protocollo, con la figura del NodeManager, per determinare il proprio abbonamento, e le proprie credenziali di account. Il NodeManager, avrà l'onere di verificare l'unicità delle credenziali account e di registrare il contatto, creando e mettendo a disposizione le risorse Cloud di coordinazione richieste nell'abbonamento.

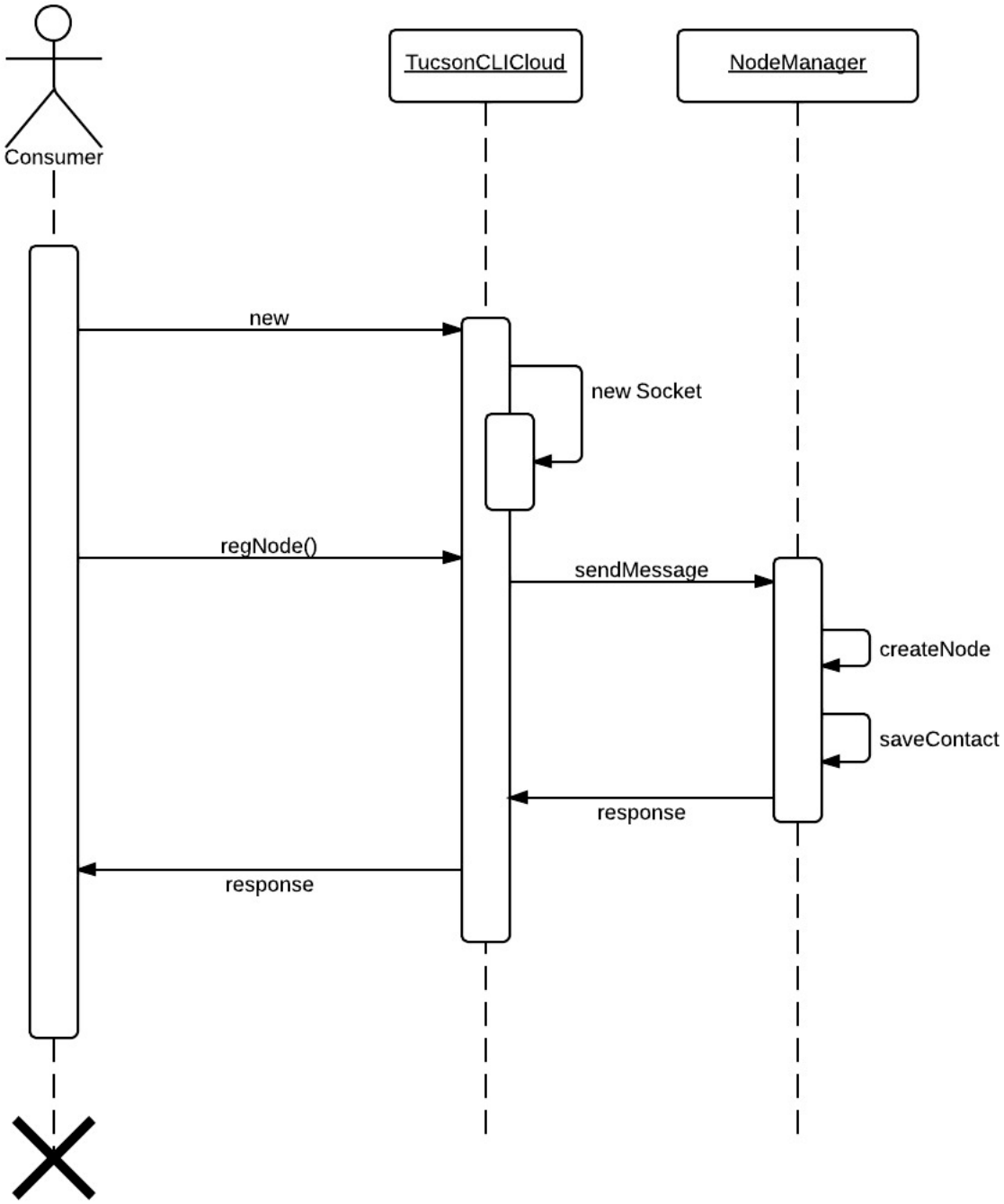

Figure 3.4: creazione account

Terminata questa prima fase, (realizzata dalla persona fisica che sfrutterà, tramite un sistema da lui realizzato, la coordinazione) il consumer, può decidere di creare dei propri agenti TuCSoN e farli coordinare dal nodo Cloud comprato. Per far ciò, ogni agente TuCSoN deve prendere un determinato ACC direttamente collegato al nodo. Per creare questo particolare ACC, prima di richiamare la creazione dell'agente, sarà necessario sfruttare il *TucsonCLICloud*, tramite esso, ricavare tutte le informazinoi necessarie per collegarsi alle proprie risorse sul Cloud.
Questa fase di inizializzazione alla coordinazione, può avere anche una dinamica leggermente diversa. Infatti, è stata data, ad un agente TuCSoN la capacità di avere più riferimenti alla volta di ACC, in questo modo, potrà coordinarsi anche con più Nodi alla volta. In questo scenario, il consumer assegnerà ad ogni ACC un nome univoco per agente, questo identificherà ogni ACC all'interno di una lista interna all'agente. Per aggiungere un determinato ACC, sarà necessario sfruttare la funzione addContext, passando le info necesssarie per agganciarsi ad un certo Nodo, come IP o url, porta, e il nome dell'ACC. Dopodiché tramite un setContext, sarà possibile settare l'ACC che l'agente potrà utilizzare.
Fatto ciò avremo un contesto Cloud, utilizzabile in tutto e per tutto, salvo limitazioni di abbonamento, come un normalissimo ACC.
Una volta eseguite queste fasi, l'agente potrà utilizzare il nodo Cloud, in maniera trasparente, senza preoccuparsi, di dove o come il nodo gli offre coordinazione, il tutto continuerà a funzionare come sempre.

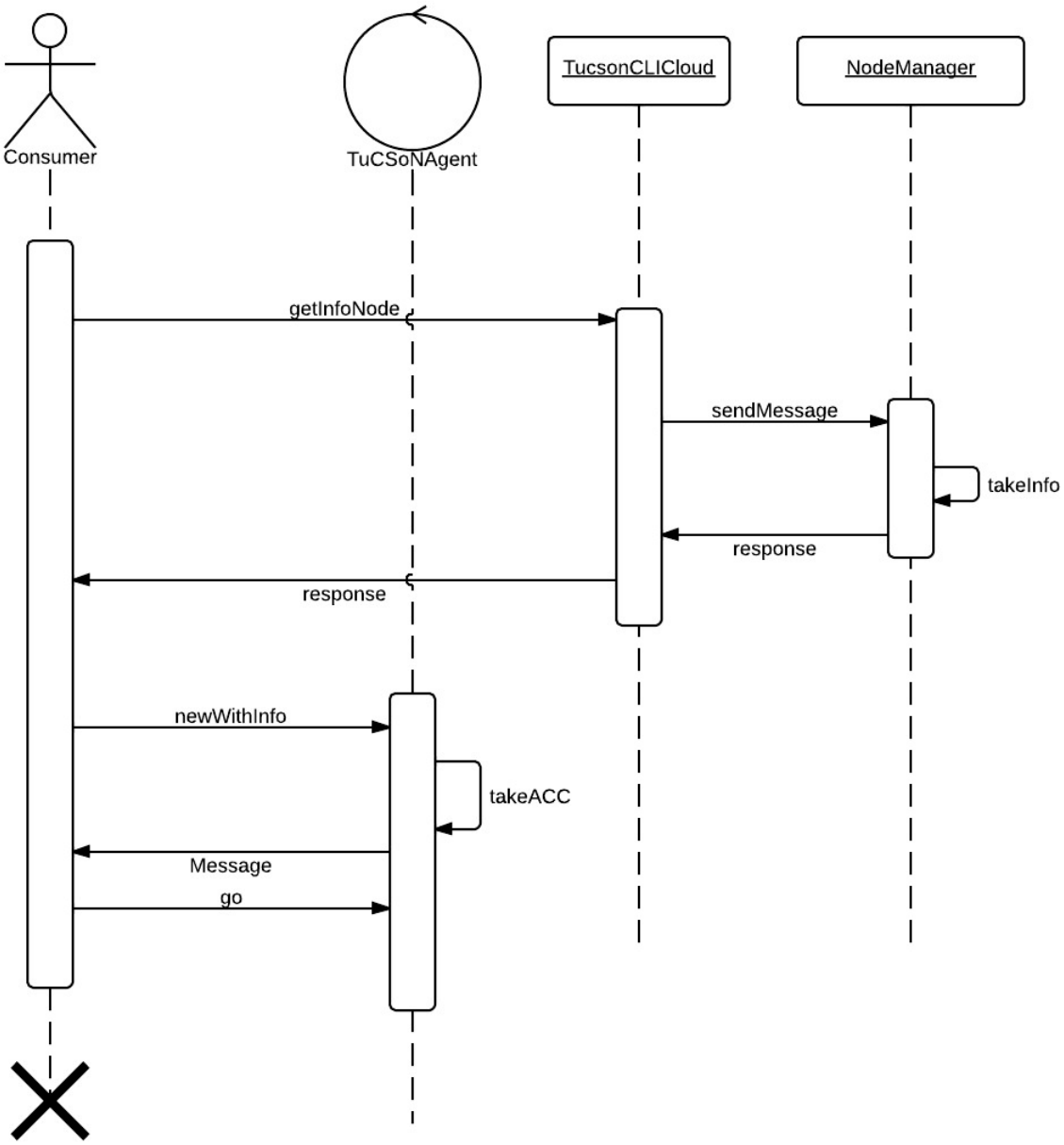

Figure 3.5: inizializzazione coordinazione

6 Sicurezza

Il NodeManager, offre i propri servizi ai vari consumer, questi, pagando "accendono" dei nodi TuCSoN che si metteranno in ascolto a determinate porte. Potenzialmente, se non viene costruito un modello di sicurezza, qualsiasi agente, anche casualmente, che inizi una cumunicazione alle porte in cui sono in ascolto i vari nodi, li può utilizzare.
Questa, anche se remota, possibilità deve essere eliminata. Una possibile soluzione potrebbe essere quella di un protocollo SRP(*secure remote password*). In pratica, quando un agente si identifica presso il provider, questo gli passa una chiave. Questa chiave può essere conosciuta solamente da quegli agenti che si sono autenticati. Creando in questo modo una comunicazione protetta, dando ai vari nodi la capacità di respingere tutte quelle richieste con chiave non corretta, provenienti quindi da agenti non identificati e potenzialmente dannosi per il sistema.

Chapter 4

Cloudify

" I computer danno esattamente quello che gli è stato immesso;
se futilità immettiamo, futilità otterremo,
ma gli uomini non sono molto diversi."
- Richard Bandler -

1 Introduzione

Sul sito ufficiale di cloudify [4] esso viene definito come *Cloudify is designed to bring any app to any cloud enabling enterprises, ISVs, and managed service providers alike to quickly benefit from the cloud automation and elasticity organizations today need. Cloudify helps you maximize application onboarding and automation by externally orchestrating the application deployment and runtime. Cloudify's DevOps approach treats infrastructure as code, enabling you to describe deployment and post-deployment steps for any application through an external blueprint - AKA, a recipe, which you can then take from cloud to cloud, unchanged.*

Cloudify, fa parte di una famiglia di freamwork chiamati OPaaS nati a supporto della modalità di servizio Platform as a Service. Grazie a Cloudify è possibile portare sul cloud qualsiasi tipo di sistema già esistente, anche se non progettato per il cloud. Essendo inoltre aperto a molti fornitori di cloud, esso può, nel caso si volesse cambiare provider, con un minimo dispendio di energie, migrare facilmente il sistema. Si può persino pensare, di sfruttare Cloudify in fase di progettazione del sistema, sfruttando il localcloud da lui creato e gestito.

2 Interprete comandi

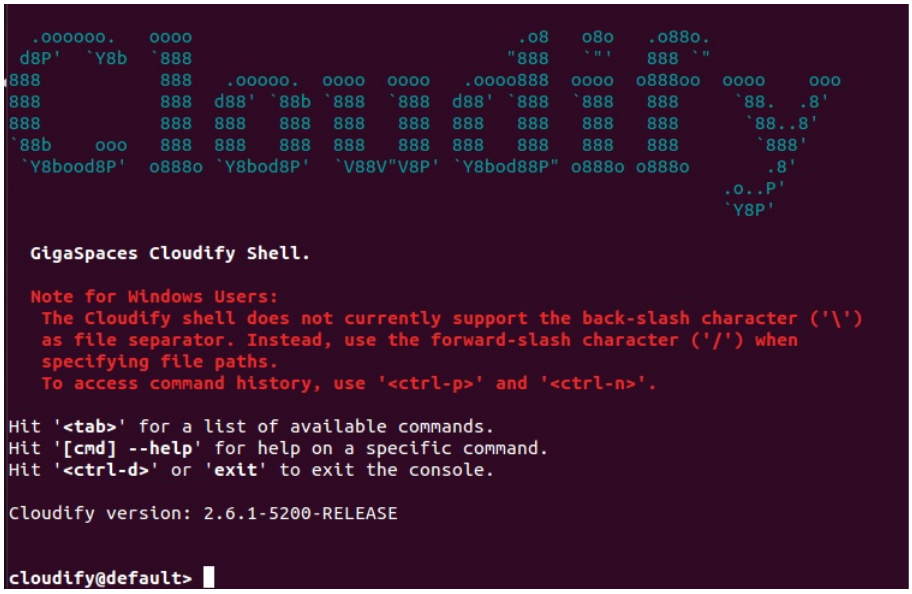

Figure 4.1: Interprete dei comandi Cloudify

Cloudify, offre ai propri utilizzatori un'interprete comandi, tramite il quale, sarà possibile installare e gestire tutte le applicazioni distribuite nel Cloud. Di seguito vengono lasciati i principali comandi, utilizzati anche nello svolgimento della tesi, con le relative caratteristiche:

- **add-template**:comando per aggiungere un template al cloud riferendo ad un file template o ad una cartella template contenente una serie di file template;
- **bootstrap-cloud**: richiama il Cloudify Agent ed il processo di gestione Cloudify sul cloud specificato come argomento;
- **bootstrap-cloud**: richiama il Cloudify Agent ed il processo di gestione Cloudify sulla macchina locale, questo processo rimane comunque isolato da altri processi di localcloud presenti su altre macchine;
- **bye/exit/quit**: tali comandi terminano la shell;
- **clear**: pulisce la consolle;
- **connect/disconnect**: permette di connettersi/disconnettersi al/dal server REST amministratore specificato come argomento sottoforma di URL o IP;
- **help**: visualizza la lista completa dei comandi disponibili;
- **install-application**: installa l'applicazione specificata, o dal file path o dalla cartella o dal archivio specificato come argomento;
- **install-service**: installa il servizio specificato, o dal file path o dalla cartella o dal archivio specificato come argomento;
- **list-application**: visualizza la lista delle applicazioni inserite nel cloud;
- **list-attributes**: visualizza la lista degli attributi dello store di controllo degli attributi di Cloudify;
- **list-instances**: visualizza la lista di tutte le istanze di un certo servizio passato come argomento;
- **list-services**: visualizza la lista di tutti i servizi inseriti nell'applicazione corrente;
- **set-instances**: setta il numero di servizi di un servizio scalabile per un certo numero di istanze inserite come argomento; macchine di gestione;
- **start-management**: mette in esecuzione l'agente Cloudify all'interno della specifica zona di gestione e nel processo di gestione sulla macchina locale, esso comunica con l'agente generale di Cloudify e con le macchine di gestione, entrambi i comandi sono eseguibili solo per un uso interno;
- **tail**: recupera le ultime n righe di un log di uno specifico servizio;
- **teardown-cloud**: termina le macchine di gestione del Cloud passatogli come argomento;

- **teardown-localcloud**: termina e disinstalla il Cloud locale installato sulla macchina;
- **test-recipes**: testa la correttezza della recipe passatagli come argomento;
- **uninstall-application**: disinstalla l'applicazione specificata come argomento;
- **uninstall-service**: rimuove un servizio specificato;
- **use-application**: setta l'applicazione da utilizzare;
- **version**: visualizza la versione di Cloudify;

2.1 Modalità Non-Interactive Shell

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, bastera lanciare il file bash contenente la seguente righa:
/bin/cloudify.sh "connect 127.0.0.1;use-application AppProva;install-service ServizioProva"
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 Funzionalità

Cloudify, può essere definito come un software che a run-time gestisce il ciclo di vita di un sistema software distribuito sopra una piattaforma Cloud.
Esso, fornisce ai suoi utilizzatori diverse funzionalità, alcune delle quali, dopo un'attenta ricerca sul web e ricordando che si tratta di un software opensource, risultano uniche nel suo genere. Prima di tutto, 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, dato che i servizi cloud sono per la maggiore pay-per-use. Cloudify inoltre, provvede *all'installazione e alla configurazine di servizi sopra l'infrastruttura Cloud* collegata, sia essa di simulazione o reale. Snellisce il processo di deploy di un'applicazione sull'ambiente cloud, ponendosi fra il consumer e il provider. Una volta realizzato il deploy delle varie applicazioni, offre strumenti agli sviluppatori per *monitorarne lo stato*. Si possono controllare molte caratteristiche, dalla memoria occupata, ai log che rilasciano i vari servizi, alle risorse computazionali sfruttate come l'impiego di RAM o CPU. Infine provvede anche, in fase di disinstallazione delle applicazioni, a liberare lo spazio utilizzato rilasciando correttamente le risorse cloud utilizzate.
Grazie a Cloudify è possibile agganciare le applicazioni a qualsiasi modello di Cloud, sia publico o privato. Esso basa i deploy delle varie applicazioni sulle recipe, queste difiniscono i parametri e il ciclo di vita delle applicazioni distribuite, offrendo un pieno controllo sia sulla piattaforma, sia sugli elementi che la popolano. Possono essere definiti comandi customizzati tramite file groovy, grazie ai quali è possibile interagire direttamente con i servizi via shell o via web browser. Si possono dichiarare probes, attraverso le quali sondare l'environment e i servizi distribuiti sopra. Di base Cloudify offre una dashboard, consultabile via web browser, grazie alla quale è possibile monitorare, in maniera intuitiva, i vari servizi.
Tramite la piattaforma offerta, è possibile anche programmare procedure di auto-scaling del sistema, eliminando in questo modo il controllo diretto umano, per operazioni di routine.

4 Recipes

Le recipes [5] sono il mezzo tramite il quale, cloudify installa, distribuisce e gestisce tutte le varie applicazioni. Esse vengono scritte in Groovy, un python-like JVM-hosted language.
Riprendendo la documentazione ufficiale, definiamo, le recipes come un piano di esecuzione per installare, eseguire, orchestrare e monitorare il livello dell'applicazione, senza cambiare il codice dell'applicazione o l'architettatura.
All'interno di cloudify possiamo individuare due tipi principali di recipes:

- **application recipes**: questa comprende sia il file descriptor dell'applicazione, sia le recipes dei vari servizi utilizzati.
- **service recipes**: questa comprende: service descriptor file, handler scripts, opzionali plugins di monitoraggio, il nome opzionale di un'icona rappresentante il servizio e un opzionale file di parametri per recipes.

Tutti i file che comprendono le varie recipes, devono essere inseriti all'interno della cartella recipes nella cartella relativa l'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:

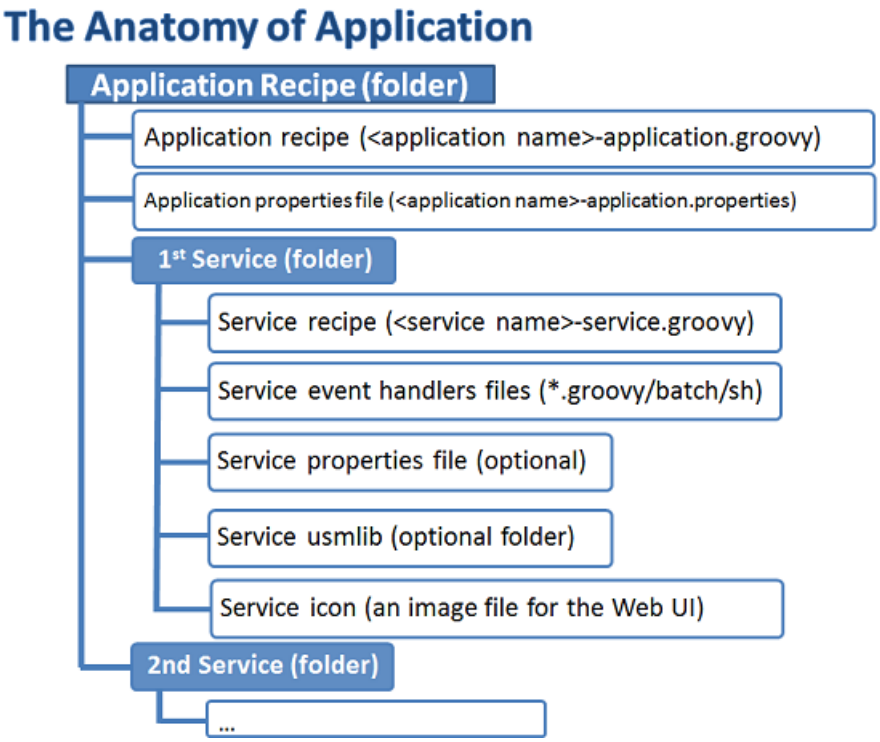

Figure 4.2: Recipes

4.1 Ciclo di vita instanza di servizio

Ogni ciclo di vita 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 events, service events e service instance events***. In questa sezione ci occuperemo degli eventi di istanza di servizio. Mentre per le prime due famiglie di cicli di vita, gli eventi e le operazioni agganciabili sono richiamabili tramite linea di comando, per gli eventi di istanza di servizio la cosa è diversa. All'interno della recipe di ogni servizio, è possibile definire un ciclo di vita seguendo la seguente sintassi:

```
1 lifecycle{
2   init "mysql_install.groovy"
3   start "mysql_start.groovy"
4   postStart "mysql_poststart.groovy"
5   preStop "mysql_preStop.groovy"
6 }
```

Figure 4.3: life cycle

questa aggancierà ai vari eventi definiti, un certo insieme di comandi, che in questo caso, sono inseriti nei vari file .groovy presenti nella stessa cartella del groovy del servizio.

4.2 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

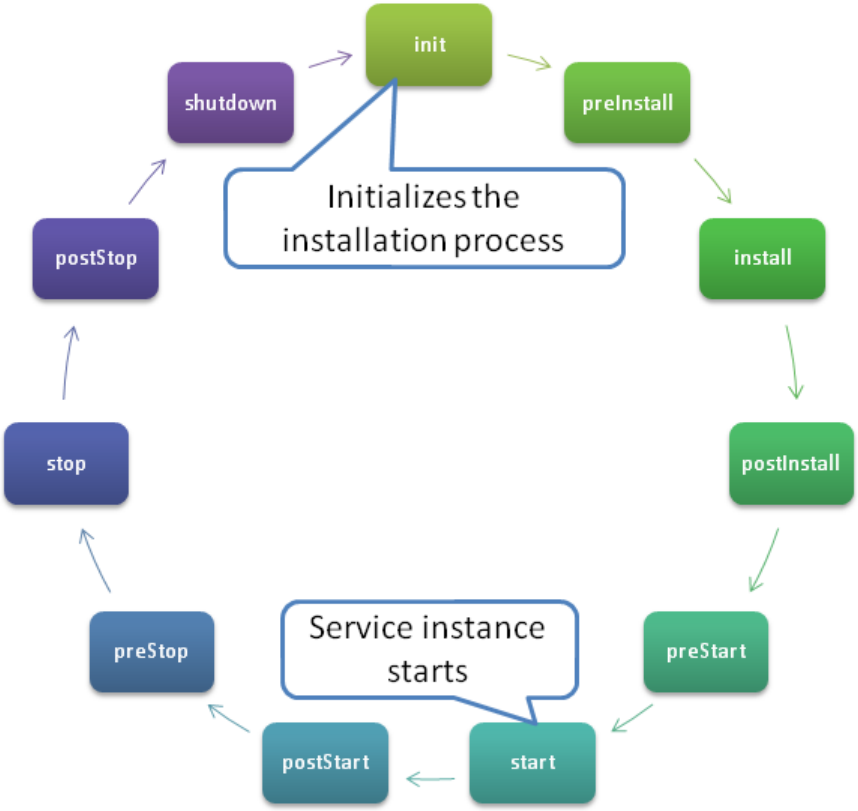

Figure 4.4: service instance lifecycle

essere presenti nella stessa cartella del file descriptor del servizio. I Formati utilizzabili sono ***Groovy, batch or shell***. Teoricamente è possibile lanciare qualsiasi script scritto in qualsisi linguaggio, questo però deve essere lanciato e gestito da uno script di un formato concesso.
Lascio di seguito la lista completa degli handler presenti in cloudify:

- preServiceStart
- init
- preInstall
- install
- postInstall
- preStart
- start

- startDetection
- locator
- postStart
- preStop
- stop
- postStop
- shutdown
- preServiceStop

Figure 4.5: Recipes

4.3 service context API

Durante il deploy e il normale funzinamento dei vari servizi sulla piattaforma Cloudify, può essere utile accedere a delle informazioni inerenti le varie istanze di un certo servizio. Per fare ciò possiamo, all'interno dei vari file groovy richiamare tramite un import la classe:
org.cloudifysource.dsl.context.ServiceContextFactory;
questa ci da modo di accedere al contesto del servizio corrente:
def context = com.gigaspaces.cloudify.dsl.context.ServiceContextFactory.getServiceContext()
e tramite esso possiamo leggere i seguenti attributi:

```
1  public int getInstanceID();
2  public String getHostAddress();
3  public String getHostName();
4
5  //Invoke a custom command
6  public void invoke(String commandName);
7
8  //get the service instances detail
9  public Object getDetails(String serviceDetailsKey);
10
11 //get the service instance monitor
12 public Object getMonitors(String serviceMonitorsKey);
```

Figure 4.6: Recipes

e richiamare i seguenti metodi:

```
1 public String getName();
2 public int getNumberOfPlannedInstances();
3 public int getNumberOfActualInstances();
4 public ServiceInstance[] waitForInstances(final int howmany,
5                         final long timeout, final TimeUnit timeUnit);
6 public ServiceInstance[] getInstances();
```

Figure 4.7: Recipes

4.4 Attributes API

A volte, può risultare utile scambiare informazini fra le varie istanze di servizio, ad esempio per sapere a quale porta un certo servizio è in ascolto. Per realizzare ciò, Cloudify mette a disposizione gli attributes API. Questi, inseribili nei vari file groovy che compongono l'esecuzione di un servizio, vengono richiamati attravverso l'import di:
org.cloudifysource.dsl.context.ServiceContextFactory;
tramite il ServiceContextFactory sarà possibile ottenere il contesto di una

certa istanza tramite:
def context = com.gigaspaces.cloudify.dsl.context.ServiceContextFactory.getServiceContext()
dichiariamo la variabile context e le assegnamo il constesto. Sfruttando context sarà possibile scrivere e leggere i vari attributi seguendo la seguente sintassi:
context.attributes.thisApplication["myKey"] = "myValue": istanzia l'attributo myKey e gli assegna il valore MyValue.
def appAttribute = context.attributes.thisApplication["myKey"]: istanzio la variabile appAttribute e gli passo il valore di myKey definito sopra.

5 Cloud Driver

Cloud driver [6], 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 pubbblici 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.

Chapter 5

Progetto e caso di studio

"Tutti sanno che una cosa è impossibile da realizzare,
finchè arriva uno sprovveduto che non lo sa e la inventa."
- Albert Einstein -

1 Visione

L'obbiettivo posto per la tesi è quello di portare il sistema TuCSoN su un'infrastruttura Cloud, al fine di sfruttare i vantaggi di questa tecnologia. Per raggiungere questo scopo ho utilizzato Cloudify, un OPaaS, che nel corso dello svolgimento del progetto, si è dimostrato un'ottima piattaforma di sviluppo e testing. La "migrazione" di TuCSoN sul Cloud è andata per piccoli passi, dato che la tecnologia di Cloudify era ancora inesplorata.
Il primo step è stato quello di portare un semplice nodo TuCSoN sul cloud, trasformandolo in un servizio dell'applicazione TuCSoNCloud creata in Cloudify. Le due possibili soluzioni erano: sfruttare un web service, il quale avrebbe

preso tutte le richieste dei vari agenti e le avrebbe indirizzate al relativo nodo sul Cloud, oppure cercare di portare l'infrastruttura sul cloud senza toccarla, ovvero far ascoltare ogni nodo ad una certa porta, come è già in TucSoN, senza sfruttare un web server da "filtro". La scelta è caduta sulla seconda alternativa, in quanto un web service avrebbe, con l'aumentare dei nodi gestiti, portato a colli di bottiglia nelle richieste e nelle risposte. Avrei dovuto modificare direttamente, o tramite wrapper, il nodo TuCSoN, appesantendo il codice. Infine ho deciso di utilizzare Cloudify, il quale, come primo scope ha quello di portare qualsiasi tipo di sistema sul cloud, senza modificarne l'infrastruttura. Mi è quindi sembrato giusto sfruttare questa sua caratteristica, allegerendo il carico di lavoro. Portato il Nodo sul Cloud, sono passato a progettare un'infrastruttura che permettesse di offrire questo Cloud a diversi consumer. A tal proposito, come ho già spiegato nel capito 3, ho sviluppato il servizio Cloud del NodeManager, creando un'infrastruttura di distribuzione di TuCSoN attraverso il Cloud.

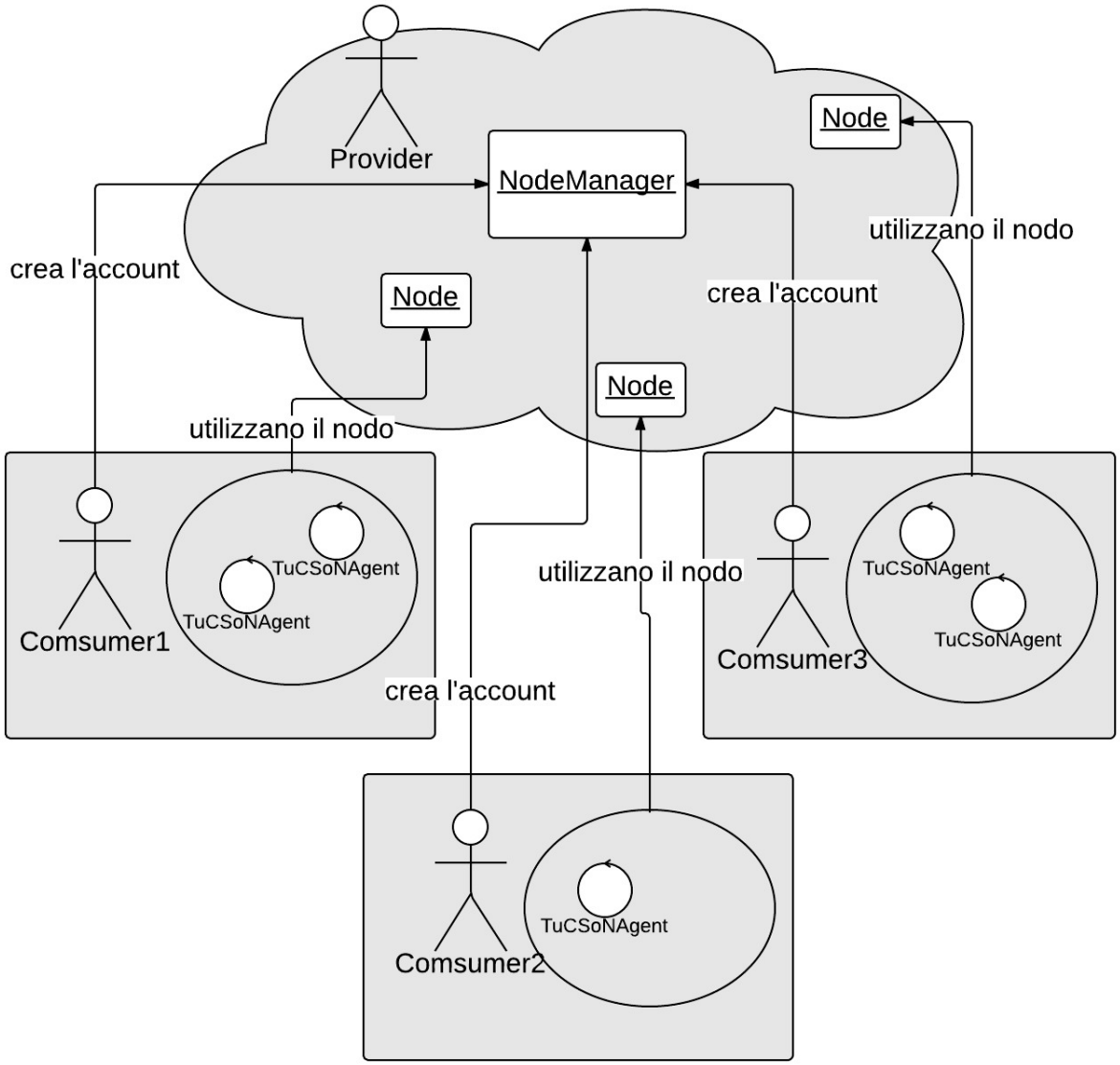

Figure 5.1: Visione sistema

2 TuCSoN Node on Cloud

Per sviluppare un nodo TuCSoN come servizio Cloudify, ho creato l'applicazione TuCSoNCloud. Come primo passaggio ho creato la cartella TuCSoNCloud all'interno di Recipes/Apps di Cloudify. Questa, a sua volta, contiene la cartella del servizio TuCSoN Node chiamata Node. La cartella Node rappresenta la recipes del servizio e contiene tutti i file a lui utili, in particolare troveremo:

node-service.groovy: file descriptor del servizio, tramite questo sarà definito:

- il nome del servizio, che deve coincidere con il nome della cartella,
- l'attributo **elastic** del servizio a true, grazie al quale, dichiariamo a Cloudify che il servizio node potrà avere più di una sola istanza, caratterista indispensabile se pensiamo di distribuire i nodi TuCSoN a più consumer
- maxIstancesAllowed il tetto massimo di istanze del servizio concesse,
- l'icona sfruttata per identificare il servizio
- il suo ciclo di vita, nel quale, in corrispondenza del event handler start, richiameremo il file Node-start.groovy.

node-Start.groovy: groovy richiamato dall'evento start del servizio Node, prenderà dal contesto del servizio Node il suo ID, assegnato da Cloudfy in maniera univoca, e gli sommerà 20504. Questo, dopo aver controllato la sua disponobilità, verrà utilizzato come numero di porta sul quale mettere in ascolto il servizio Node. Nel caso in cui un numero di porta non risultasse libero, il numero verrà incrementato di uno e il test sarà rieffettuato. Trovato un numero libero, verrà lanciato il file shell launch.sh passandogli come parametro il numero di porta appena trovato. In questa manierà possiamo avere n servizi Node che asccoltano tutti a porte diverse, partedo da una radice comune(20504).

launch.sh: Il seguente script bash, inserirà in un file Registry.txt il numero di porta passato e farà partire il jar contenente tutti i file necessari per eseguire un nodo TuCSoN, passandogli come paramentro il numero otteneuto

dal file groovy.
Inseriti tutti questi file, basterà far partire l'interprete comandi di Cloudify, instaurare il localcloud, e installare l'applicazione TuCSoNCloud. Questo scenario ancora grezzo, non da modo di sapere ad un agente a quale porta andare a parlare, inoltre, rende possibile distribuire un solo nodo TuCSoN, mentre l'obbiettivo era quello di creare un'infrastruttura di distribuzione di più nodi TuCSoN a più User.

TuCSoN.jar questo file, conterrà il jar con tutte le classi indispensabili per un Nodo TuCSoN.

In uno sviluppo futuro, quando il nodo verrà distribuito in un Cloud reale, il file Registry.txt dovrà essere inserito in una zona di memoria accessibile al NodeManager. Potrebbero a tal proposito, essere utilizzati degli storage cloud, oppure, nel caso di un Cloud privato, basterà inserire il file nella macchina virtuale reggente Cloudify, rendendo raggiungibile tramite percorso assoluto o relativo il file testuale.

3 NodeManager on Cloud

Il servizio NodeManager, sarà il responsabile della creazione e gestione delle varie istanze dei servizi Node. La sua recipes sarà cosi composta:

NodeManager-service.groovy: file descriptor del servizio, questo lancerà, allo scatenarsi dell'evento start, il file shell launchNodeManager.sh.
launchNodemanager.sh: responsabile dello start del NodeManager.jar.
NodeManager.jar: questo file jar contiene la classe NodeManager tramite la quale sarà possibile ricevere messaggi dalla classe TucsonCLICloud utilizzata dai consumer. A seconda del messaggio ricevuto il NodeManager richiamerà lo start di una nuova istanza del servizio Node, lanciando lo script bash launchNode.sh, oppure leggerà il proprio registro interno per dare le informazioni richieste dai vari consumer.
launchNode.sh responsabile, a seconda del parametro passato, dell'installazione oppure dell'aumento del numero di istanze, del servizio Node. Per richiamare

questo script viene sfruttata la modalità non Interactive di Cloudify (vedi capitolo 4).

4 Sistema TuCSoNCloud

Il sistema TuCSoNCloud comprende due servizi principali, il Node e il NodeManager. Nel contesto Cloudify, questo di traduce in un'applicazione TuCSoNCloud che comprende entrambi i servizi sopra descritti, in particolare nella recipe dell'applicazione avremo:

TuCSoNCloud-application.groovy: file descriptor dell'applicazione, questo dichiarerà il servizio NodeManager, in questo modo quando verrà installata l'applicazione automaticamente Cloudify installerà anche il servizio NodeManager.

5 TucsonCLICloud

Questa classe abiliterà, come detto in precedenza, la comunicazione, fra consumer e NodeManager. Il codice della classe, lasciato in fondo al capitolo, instaura la comunicazione tramite socket con il nodeManager, dopodiché seguendo un certo protocollo, permette ad un consumer, di registrare nuovi nodi, oppure ricavare le informazioni per accedere alle risorse Cloud collegate al proprio account.
Questa classe può essere vista come l'interfacciamento con il Cloud. Nella mia visione, questo TucsonCLICloud, è consegnato ai vari consumer dal provider, ad esempio, dando la possibilità di scaricarlo dal proprio sito, se non indicato diversamente, conterrà già le informazioni per collegarsi con il provider, come l'url e la porta a cui comunicare.

6 UserCloud

Dato che, la registrazione del cloud, non verrà mai fatta da automi ma sempre da persone umane, mi è sembrato opportuno realizzare un'applicazione lato

consumer, che rendesse la registrazione più user friendly. Questa comunicherà direttamente con il TucsonCLICloud, richiamando i metodi opportuni.

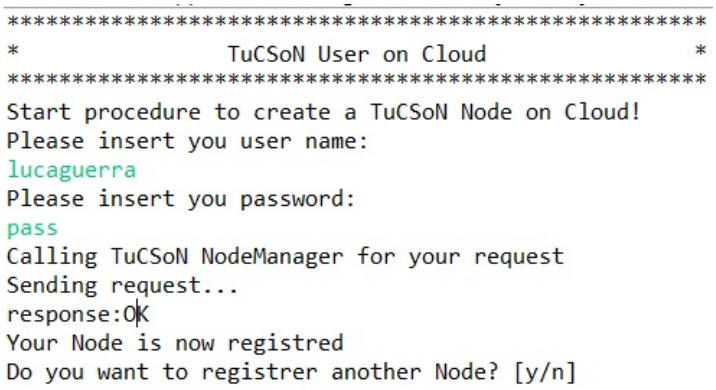

Figure 5.2: UserCloud

7 Caso di studio (Dining Pholosopher On Cloud)

Nel seguito, sarà presentato un caso di studio, al fine di presentare un esempio, e dare la possibiltà ad altri di replicare i risultati da me trovati per poterli sfruttare come punto di partenza per ulteriori studi.

7.1 Visione (filosofi affamati on Cloud)

Ipotizzando che un'azienda voglia sfruttare dei nodi TuCSoN on Cloud, al fine di sviluppare un'applicazione che utilizzi la coordinazione offerta, prendiamo ad esempio il problema dei filosofi affamati.
Nell'esempio dei filosofi affamati, ritroviamo il classico scenario di applicazione distribuita nella quale n processi sono interessati ad accedere a n risorse condivise limitate. Un buon metodo per attaccare questo problema è sfruttare TuCSoN e le reaction. Nella sua versione standard, abbiamo, n centri di tuple chiamati *seat*, i quali identificano idealmente i posti a sedere dei vari filosofi e un centro di tuple *table*, responsabile di gestire e mantenere la

risorsa condivisa. La dinamica della coordinazine è la seguente: ogni filosofo per poter mangiare, avrà bisogno di ottenere due *chop*, questi gli permetteranno di mangiare gli spaghetti idealmente posti sul tavolo. I chop sono rappresentati da tuple, queste tuple saranno contenute nel centro di tuple *table*. Ogni filosofo, che altro non è che un Agente TuCSoN, quando vuole iniziare a mangiare, rilascia una richiesta di wanna-eat nel suo *seat*, questo lancierà una reaction, la quale richiederà i chop, a lui vicini, al centro di tuple *table*. Questo, appena disponibili ritornerà la coppia di chops, le quali potranno essere utilizzate per mangiare dal filosofo. In questo caso nel suo seat comparirà la tupla eating a indicare il suo stato, dopo un certo periodo il filosofo rilascerà i chops i quali torneranno disponibili all'interno del centro di tuple table, per un altra richiesta.
Quello che vogliamo ottenere, partendo da questo scenario, è quello di inserire nel Cloud il centro di tuple *table*, il quale simula il gestore di risorse condivisive, mentre tutti i filosofi e i loro centri di tuple *seat*, lasciarli in un nodo "normale" non Cloud. Pensando ad un sistema realmente distribuito, portando il table sul cloud, forniamo una maggiore robustezza e velocità in fase di accesso alla risorsa condivisa. Infatti ogni filosofo, idealmente sarà presente su un certo device, e comunicherà con il table per le sue risorse, nel momento in cui il numero dei filosofi dovesse aumentare, il table si ritroverebbe a dover scalare le proprie risorse computazionali, al fine di rimanere efficiente, sicuramente lo strumento del Cloud viene in aiuto. Inoltre, grazie al Cloud, possiamo avere la quasi certezza della onnipresenza del table, sfruttando meccanismi di duplicazione di macchine virtuali. Al fine di avere una visione più chiara, separiamo fin da subito, i compiti di consumer e provider di servizio.

7.2 Provider Side

Per simulare la figura del provider, ho volutamente sfruttato, tramite l'ausilio di un software per la virtualizzazione di macchine, il sistema operativo Ubuntu. La scelta è ricaduta su questo sistema, perché Cloudify nascendo su OpenStack, trova la sua massima compatibilità con l'ambiente Unix. Una volta

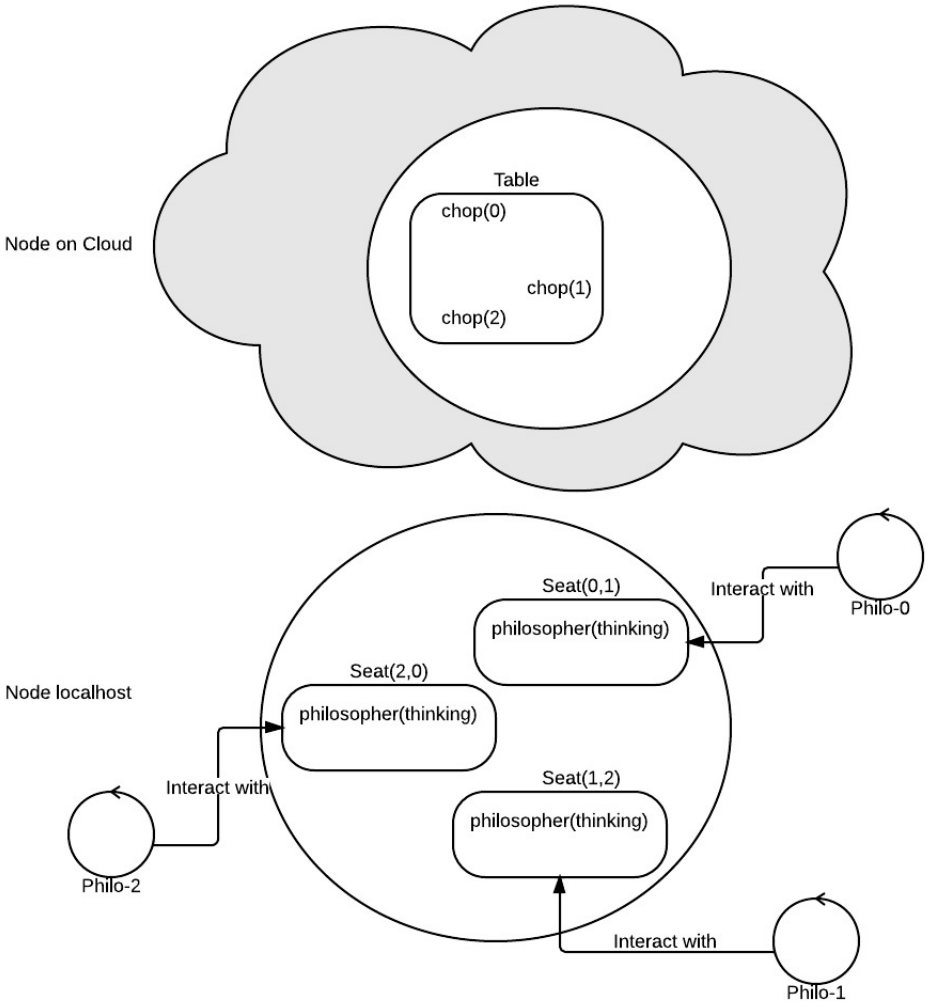

Figure 5.3: DiningPhilopherOnCloud

preparata la macchina, ho scaricato dal sito di cloudify, www.cloudifysource.org, l'ultima versione di GA della piattaforma. Non vi è installer, appena scaricato è possibile fin da subito utilizzare la shell di Cloudify, basta spostarsi nella cartella bin e far partire il bash cloudify.sh (sh ./cloudify.sh). Per poter installare l'applicazione TuCSoNCloud, bisogna inserire la cartella relativa all'interno di Recipes/Apps. Fatto ciò, il provider, può collegarsi al Cloud, nel nostro esempio con bootstrap-localcloud, dopodiché con il comando install-

application TuCSoNCloud installerà l'applicazione, la quale farà partire il NodeManager. Terminati questi passaggi la figura del provider smette di lavorare, sarà il NodeManager che gestirà il tutto.

7.3 Consumer Side

Lato consumer, l'interazione si divide in due fasi:

- ***Creazione account***: in questa fase tramite l'applicazione *CloudUser*, il provider, crea un account TuCSoNCloud, passando le informazioni di autenticazione, username e password. Grazie a questo passaggio sarà possibile avere l'accesso ad un nodo Cloud. Se tutto è andato a buon fine verrà comunicato dall'applicazione, e sarà possibile utilizzare il nodo.
- ***Utilizzo risorse Cloud***: il consumer, nel caso dei filosofi affammati, dovrà dividere l'elaborazione in *in Cloud* e *out Cloud* sfruttando la possibilità data al TuCSoN Agent di utilizzare più ACC. In particolare un ACC per il nodo out Cloud e un'altro per il Nodo on Cloud.

7.4 Dinamica filosofi affamati on cloud

Per prima cosa creiamo il nodo sul Cloud, questo conterrà il centro di tuple *table*. Per far ciò installiamo TuCSoNCloud su Cloudify e tramite l'applicazione UserCloud creiamo un account. Eseguita questa operazione, sarà necessario creare un nodo localhost sulla nostra macchina fisica al fine di contenere i centri di tuple *seat*. Realizziamo questa operazione seguendo la normale procedura.
Creati i due nodi, facciamo partire la classe DiningPhilosopherOnCloudTest, questa per prima cosa crea un Agente TuCSoN collegato al nodo localhost e lo manda in esecuzione. L'agente, all'interno del suo main, prende le informazioni necessarie per accedere al nodo Cloud tramite il TucsonCLICloud e si aggiunge il contesto Cloud, tramite addContext. A questo punto crea gli n *seat* in localhost e gli inserisce la tupla *philosopher(thinking)*, dopodiché setta il contesto Cloud a principale e crea sul Cloud il table con all'interno gli n chop. Come ultima operazione, l'agente richiama il metodo go() dei vari filosofi mettendoli in esecuzione nell'ambiente preparato.

8 Sviluppi futuri

Il sistema sviluppato è ancora molto grezzo, la sua funzione principale è quella di mostrare un modello per distribuire TuCSoN sul Cloud e come sia possibile farlo attraverso la piattaforma Cloudify.
Sicuramente in futuro, al fine di rendere il tutto davvero utilizzabile, sarà indispensabile inserire un protocollo criptato fra nodi Cloud e agenti, in modo da evitare attacchi informatici e dare una maggiore robustezza in termini di sicurezza ed affidabilitá al sistema. Si potrebbe pensare di offrire la possibilità ai consumer di poter scalare le proprie applicazioni on Cloud estendendo la classe UserCloud. Un'ultriore proprietà da poter aggiungere potrebbe essere la possibilità di self-managing dei vari nodi, in modo da aumentare l'affidabilità del sistema. Per concludere, dal lato commerciale, sarebbe interessante agganciare un qualche meccanismo di abbonamenti, ognuno dei quali fornisce una visione più o meno aperta dell'utilizzo del nodo, applicando un meccanismo di filtri.

9 Codice

9.1 NodeManager

CREATENODE (lascio solo le parti principali)

```
//Controllo se es gia presente un contatto con quel nome
if(registry.get(user)==null)
{
  if(registry.isEmpty())
  {
        try
        {
          System.out.println(
          "Start service & create Node for:"+user
          );
          //Eseguo il file di bash che
          //carica il nuovo nodo sul cloud
          result = Runtime.getRuntime()
                    .exec("sh launchNode.sh InstallNode")
                                                .waitFor();
        }catch(Exception e)
        {
          e.printStackTrace();
          return -2;
        }
  }
  else
  {
    try
    {
      System.out.println("I'm creating Node for:"+user);
      int numOfServices = registry.size()+1;
          //Eseguo il file di bash che carica
          //il nuovo nodo sul cloud
          result = Runtime.getRuntime()
```

```
                .exec("sh launchNode.sh AddNode "+numOfServices)
                                                        .waitFor();
        }catch(Exception e)
        {
          e.printStackTrace();
          return -3;
        }
  }
  if(result==0)
  {
    FileReader f=null;
    int port=0;
    //Apro il file
        try {
          f = new FileReader("percorso/Registry.txt");
        } catch (FileNotFoundException e)
        {
      e.printStackTrace();
      return -4;
        }
        //Leggo il numero di porta
        try {
      port = Integer.parseInt(new BufferedReader(f).readLine());
        } catch (NumberFormatException | IOException e) {
          e.printStackTrace();
          return -4;
        }
        System.out.println("I'm registring a User..");
        Record infoRead = new Record();
        infoRead.password = password;
        infoRead.port = port;
        System.out.println("Node listening on:"+infoRead.port);
        //Aggiungo al registro il nuovo user
        registry.put(user, infoRead);
```

```
        //Aumento il numero di istanze
        Instances ++;
        return 0;
  }
  return -3;
}
return -1;
```

TAKEINFO

```
private Info takeInfo(String info)
{
  Info inf = new Info();
  String user = info.substring(0,info.indexOf(';'));
  String password = info.substring(info.indexOf(';')+1);
  Record recordFind = registry.get(user);
  if(recordFind != null && recordFind.password.equals(password))
  {
        System.out.println("record find..");
        inf.port=recordFind.port;
        return inf;
  }
  System.out.println("record not find..");
  inf.port=-1;
  return inf;
}
```

DOWORK

```
public void doWork()
{
  ServerSocket serverSock = null;
  boolean exit = false;
  try {
    //Listen on PORT_NUMBER
    serverSock = new ServerSocket(PORT_NUMBER);
  }catch(Exception e)
```

```
{
  e.printStackTrace();
}
while(!exit) {
  try{
    System.out.println("Listening a Tucson Agent...");
    //Get connection
    Socket clientSock = serverSock.accept();
    System.out.println(
    "Connected client"+clientSock.getInetAddress()
    );
    //Get input
    BufferedReader in = new BufferedReader(
              new InputStreamReader(clientSock.getInputStream()
                                    ));
    DataOutputStream out = new DataOutputStream(
              clientSock.getOutputStream()
                                    );
    readFromCLI = in.readLine();
    cmd = readFromCLI.substring(0,readFromCLI.indexOf(':'));
    //Leggo da client
    System.out.println("I read this:"+readFromCLI);
    if(cmd.equals("close"))
    exit=true;
    switch(cmd)
    {
      case "reg":
        int result = createNode(
        readFromCLI.substring(readFromCLI.indexOf(':')+1)
          );
        switch(result)
        {
          case 0:
            out.writeBytes("OK"+"\n");
```

```
            break;
          case -1:
            out.writeBytes("user used yet.."+"\n");
            break;
          case -2:
            out.writeBytes("Problems node launch.."+"\n");
            break;
          case -3:
            out.writeBytes("Problems add node instance.."+"\n");
            break;
          case -4:
            out.writeBytes("Problems read port number.."+"\n");
            break;
        }
        break;
      case "info":
        //Ritorno la porta al CLI
        out.writeBytes(""+takeInfo(
          readFromCLI.substring(
            readFromCLI.indexOf(':')+1)
          ).port
        );
        break;
    }
    in.close();
    out.flush();
    clientSock.close();
  } catch(Exception e) {
    e.printStackTrace();
  }
}
try {
  //Chiudo la connessione e rilascio la porta
  serverSock.close();
```

```
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

9.2 TucsonCLICloud

regNode

```
public boolean regNode()
{
  System.out.println("Sending request...");
  //Registro un nuovo nodo
  try {
    outToManager.writeBytes("reg"+ ":" + user + ";" + pass + '\n');
    //Leggo il risultato
    String result = inFromManager.readLine();
    System.out.println("response:"+result);
    if(result.equals("OK"))
  {
    return true;
  }
return false;

  } catch (IOException e) {
        e.printStackTrace();
  }
  return false;
}
```

getInfoNode

```
public int getInfoNode()
{
  //Registro un nuovo nodo
  try {
```

```
        outToManager.writeBytes("info"+ ":" + user + ";" + pass + '\n');
        //Leggo il risultato
        String result = inFromManager.readLine();
        if(result != "-1")
        {
          int myNodePort = Integer.parseInt(result);
          System.out.println("Your node is on portno:"+myNodePort);
          return myNodePort;
        }
        return -1;
      } catch (IOException e) {
            e.printStackTrace();
      }
      return -1;
}
```

Bibliography

[1] Andrea Omicini Mirko Viroli, Coordination as a Service: ontological and Formal Foundation;

[2] Andrea Omicini, Stefano Mariani "The TuCSoN Coordination Model and Technology. A Guide", TuCSoN v. 1.10.3.0206, Guide v. 1.0.2, 9 January 2013;

[3] Enrico Denti, Antonio Natali, Andrea Omicini "On the expressive power of a language for programming coordination media", in: Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98), Atlanta(USA) 27 February - 1 March 1998;

[4] *www.cloudifysource.org/guide*;

[5] *www.cloudifysource.org/guide*/2.6/*developing/recipes_overview*;

[6] *www.cloudifysource.org/guide*/2.6/*clouddrivers/cloud_driver*;