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:

TuCSoN on Cloud: Revisione dell'architettura

Elaborato in:

Sistemi Distribuiti

Relatore:
Prof. Andrea Omicini
CoRelatore:
Ing. Stefano Mariani

Candidato:
Fabio Ricca Rosellini

Sessione II 2014/2015

Dedica*:*

a chi mi è stato vicino

e a chi ha creduto in me

Indice

Capitolo 1

Introduzione

L'obietto di questa tesi è sì migliorare l'attuale architettura di TuCSoN on Cloud [1] ma è stata anche una prova per me poichè mi ha permesso di mettermi in gioco, di studiare cose nuove e di farmi vedere l'informatica con un altra ottica. Infatti se dapprima dovevo creare dal nulla delle applicazioni e dover rendere conto solo a me stesso ora invece mi sono dovuto mettere alla prova in un ambiente tutto nuovo. Ovvero mi sono dovuto relazionare con il lavoro di altre persone e lavorare in un ambiete appunto non creato da me e mi ci sono dovuto adattarvi dando il mio piccolo contributo.

1.1 Vision

L'idea con cui sono partito per scrivere questa tesi è stata quella di vedere cosa si poteva migliorare nell'attuale struttura di TuCSoN on Cloud per cercare di fare un altro piccolo passo avanti. Infatti cercando di porre soluzione o di apportare migliorie a questo sistema sarà possibile offrire a TuCSoN on Cloud una maggiore affdabilità, scalabilità ed effcienza, senza considerare che sarà più semplice distribuirlo ed offrirlo a possibili Utenti.

Capitolo 2

TuCSoN

2.1 Modello di TuCSoN

TuCSoN (Tuples Centers Spread over the Network) [3] è un modello e una tecnologia realizzata dal gruppo di ricerca APICe di Bologna e Cesena. Questa, estendendo ed ereditando aspetti del modello di coordinazione Linda [6], coordina processi distribuiti sulla rete e agenti autonomi, intelligenti e mo- bili. Il modello fonda le sue radici sull'astrazione di centro di tuple [4], ovvero uno spazio di tuple programmabile, dove la programmazione avviene tramite ap- posite tuple, chiamate di specifica, scritte in linguaggio ReSpecT [7]. 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. Qui gli elementi fondamentali:

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. 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.

2.2 Funzionamento

TuCSoN, fornisce un linguaggio di coordinazione 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 op- erazione 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.

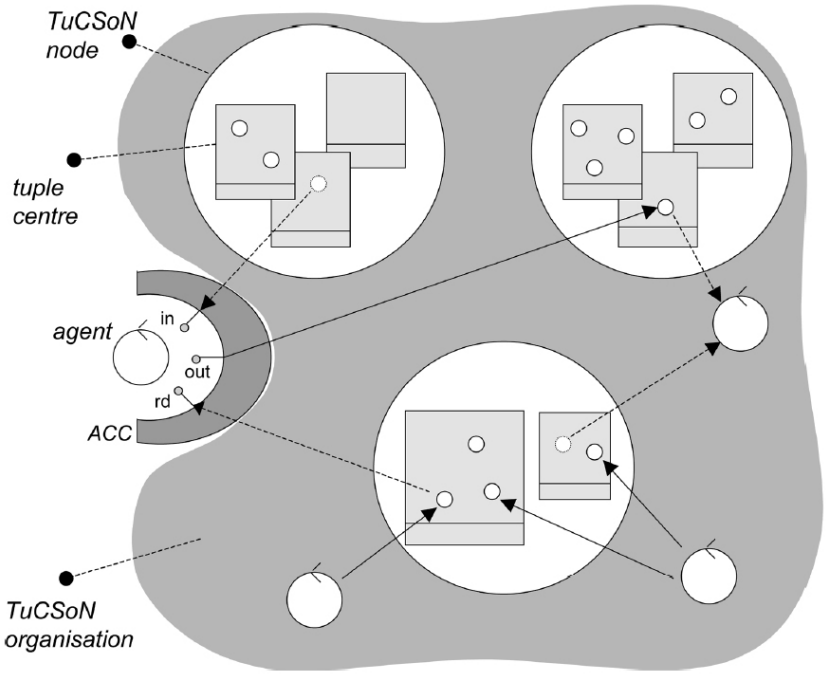

Figura 2.1: Schema TuCSoN

Capitolo 3

TuCSoN on Cloud

3.1 Vision

Viste le caratteristiche di un sistema coordinato, si può pensare di portarle all'interno di un servizio Cloud [2]. 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) [8]. Il modello TuCSoN, sembra sposarsi alla perfezione con l'idea di Cloud [1], 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 necessità 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.

3.2 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. Il modello di base sostanzialemente quindi si sviluppa in due sottogruppi, quello del lato Consumer e quello del lato Provider. Lato Consumner è possibile richiedere un account e richiedere tuple dal cloud quale nodi e spazi di tuple. Lato provider è necessario creare nodi e accogliere le richieste degli agenti. Come servizio cloud è anche necessario che possa offrire un servizio cloud, ovvero altamente scalabile e con ridondanza dei nodi se serve.

3.3 Funzionamento

L'attuale struttura di TuCSoN on Cloud si basa su un Web Service che gestisce gli account e i Nodi TuCSoN a lor collegati. Essa si basa su tre livelli fondamentali

- **Client**: utulizza il servizio.
- **Web Service**: gestisce l'interazione fra i componenti.
- **Nodi TuCSoN**: classici Nodi TuCSoN.

Ogni entità coordinata del sistema è un client del Web Service. La visione del nodo è mascherata ai client dal Web Service che si comporta da proxy per le richieste di coordinazione. Si è quindi scelto di utilizzare il REST [9] poichè così è stato possibile utilizzare il servizio tramite la rete con semplici interrogazioni HTTP.

3.3.1 Richiesta account

Per meglio comprendere come funziona tale servizio analizziamo il caso nel quale un utente voglia creare un proprio account. Innanzi tutto bisogna mandare al Web Service, nel nostro caso fatto girare in locale con Tomcat 8, un file xml con i dati dell'utente . Ricevuto questo file tramite la classe RegistryAccessLayer si controlla se lo username è disponibile e lo si aggiunge ad un file registry. Salvato il dato si richiede la creazione di una nuova istanza del servizio Node a Cloudify [5], uno script cercerà una porta libera sulla macchina, una volta individuata la invierà al Web Service tramite la classe SendPort e metterà in esecuzione un nuovo nodo TuCSoN. Per ora tutto è gestito con una busy wait Quando il Web Service riceve una nuova porta, prende il file Registry, cerca un account senza porta assegnata e gli

assegna quest'ultima, in questo modo nel file Registry avremo i dati di autenticazione e la porta a cui parlare. Come vedremo successivamente questo modo di salvare i dati crea alcuni problemi che tratterò in seguito. Per gestire la comunicazione con Cloudify si è creato un livello di astrazione chiamato CloudifyAccessLayer con il compito di gestire l'interazione con Cloudify.

3.3.2 Login al servizio

Per rendere la comunicazione minimale e sicura si è deciso di sfruttare la sessione del Web Service. La sessione altro non è che un file nel quale il server associa ad uno specifico id delle stringhe di testo, questo id viene creato per ogni nuova connessione che lo richieda. Per permettere al sistema di utilizzare un meccanismo di sessione basta che il client una volta ottenuto un sessionID dal server, lo reinserisca nell'header delle chiamate HTTP delle richieste successive, in questo modo il server potrà attingere alla stringa associata e potrà leggere il dato. Tramite questo meccanismo i vari client possono sfruttare un concetto di log-in al servizio. Il workflow è il seguente: alla prima connessione il Web Service restituisce un sessionID che viene salvato dal client e riutilizzato nelle connessioni successive, il web service, dopo una log-in associa lo username alla sessione, così nella fase di coordinazione basta inserire il sessionID nell'header per essere autenticati e passare nel body le minime informazioni per richiedere le primitive di coordinazione.

3.4 Interazione

Quindi per concludere la creazione di un nodo TuCSoN su un ambiete cloud simulato si deve, tramite un applicativo java, mandare un richiesta di tipo NewNode alla classe Operations inizializzata con l'indirizzo del WebSer-

vice (in questo caso localhost:8080). Successivamente si mostra la struttura del tale file che Operations invia al Manager per la creazione del nodo nell'ultimo capitolo. In seguito si interroga il sistema per un Login, ciò avviene sempre passando da Operations che, a sua volta crea come detto precedentemente,lancia un file http con nell'header un sessioID, che poi passa al Proxy per l'autenticazione. Il manager, ricevuto il file XML, chiama CloudifyAccessLayer passandogli tali informazioni affinchè esso possa interagire con Cloudify e creare un vero e proprio nodo. Una volta che il CloudifyAccessLayer ha ricevuto il comanda bisogna controllare che il nodo appena richiesto sia il primo da quando cloudify è partito, o meno, e inoltre controllare se l'utente che ha richiesto tale nodo sia un nuovo utente. In base a questo controlli il CloudifyAccessLayer deve inizializzare cloudify o richiedere una nuova istanza e delegare al RegistryAccessLayer le modifiche da effettuare sul file di registro. Infatti ora possiamo distriguere due casi, nello specifico trattati nell'ultimo capitolo. Se l'utente è la prima volta che si logga la servizio, verrà creata una struttura xml ad hoc contenente tutti dati utili ad identificarlo, ovvero l'username, la password e tutti i suoi tuplecentre annessi, più altre informazioni accessorie per tenere traccia di tali nodi. Se invece l'utente era già registrato nel servizio e aveva già un tuplecentre a lui assegnato e ne sta richiedendo uno nuovo, alla sua entry nel file XML dev'essere aggiunto il nuovo tuple centre, passato al Manager precedentemente. La nuova implementazione di tale funzione è svolta anch'essa nell'ultimo capitolo. Ora analizziamo la fase di vero e proprio utilizzo del nodo. Per operare all'interno di un nodo TuCSoN si possono usare le operazioni classiche di "in o "out", ovvero prendere la risorsa e rimetterla nel centro rispettivamente. In questo caso allora si torna ad interrogare Operations affichè lui effettui una connesione al Proxy inviando un file contente il comando "in" e la risorsa

di interesse. Il Proxy poi comunica al NodeAccessLayer il quale è colui che effettua questi comandi sul centro di tuple.

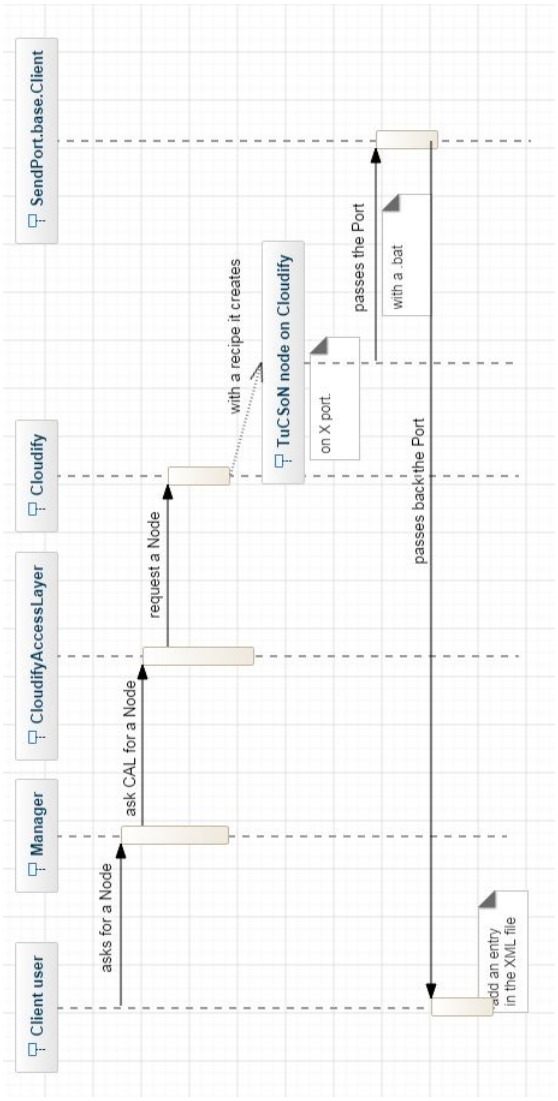

Figura 3.1: Interazioni

Capitolo 4

Cloudify

4.1 Introduzione

Sul sito uffciale di cloudify [5] 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 (On-Premise as a Service) 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. Qui di seguito vediamo una lista dei comandi utilizzabili su cloudify tramite la sua interfaccia shell a noi utili:

- 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;

- connect/disconnect: permette di connettersi/disconnettersi al/dal server REST amministratore specificato come argomento sottoforma di URL o IP;

- 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;

- set-instances: setta il numero di servizi di un servizio scalabile per un certo numero di istanze inserite come argomento; macchine di gestione;

- teardown-cloud: termina le macchine di gestione del Cloud passatogli come argomento;

- use-application: setta l'applicazione da utilizzare;

- version: visualizza la versione di Cloudify; enditemize

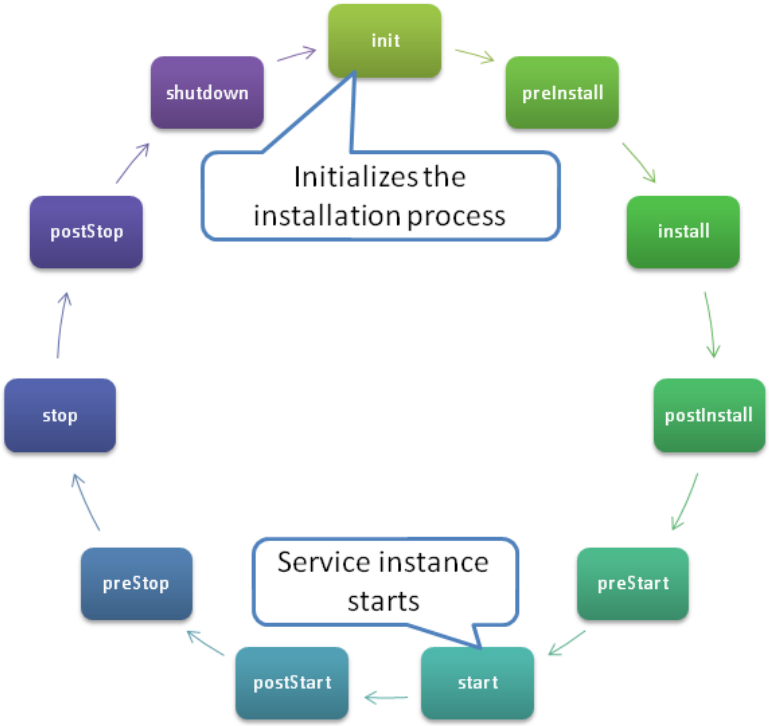

Figura 4.1: ciclo Cloudify

4.2 Utilizzo

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 è 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.

4.3 Funzionamento

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 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.

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 tramite script Groovy, grazie ai quali è possibile interagire direttamente con i servizi via shell o via web browser. Di base Cloudify offre una dashboard, consultabile via web browser, grazie alla quale è possibile moni- torare, in maniera intuitiva, i vari servizi.

Capitolo 5

Progetto

5.1 Vision

L'obiettivo di questa tesi è di continuare il lavoro su TuCSoN on Cloud, che ho brevemte ripreso qui sopra, per cercare di porre soluzione ad alcuni aspetti migliorabili di tale architettura. Per migliorare l'intero apparato si è dovuto operare su più fronti. Dapprima si è cercato di risolvere il problema relativo alla memorizzazione dei nuovi tuple centre associati ad un singolo utente, infatti questi venivano salvati come nuove entry nel file xml, creando una ridondanza non necessaria e possibilmente problematica in un ambiente distribuito come quello del cloud. Era poi impossibile recuperare altri tuple centre a lui associati, si restituiva sempre e comunque il primo. Un altro problema riscontrato era quello della sicurezza. Infatti qui i dati viaggiano completamente in chiaro, cosa discutibile in un vero ambiente distribuito. Qui di seguito propongo una semplice soluzione per il salvataggio della password utente nell'XML in forma criptata usando l'algoritmo SHA-256, data le sua

caratteristica non reversibilità. Non ultimo si è dovuto intervenire anche nella gestione delle performance. Difatti ad oggi alcuni meccanismi si basano su busy-wait, ovvero forme di polling. In un ambiente locale e circoscritto non sono un problema ma se si deve pensare a sviluppare questa applicazione in un vero cloud bisogna risolvere il problema con reali metodi di sincronizzazione fra i vari componenti. Ci si è quindi voluti soffermare su

- **Gestione Nodi**
- **Gestione Concorrenza**
- **Sicurezza**

5.2 Gestione Nodi

Si è dovouto riguardare il metodo di implementezione della funzione che si occupa di salvare i dati in un file xml. Infatti ora se un utente chiede un nuovo nodo noi lo aggiungiamo al file xml come fosse un utente nuovo, con l'evidente problema che se si va a richidere un nodo associato ad un utente viene sempre e solamente restituito il primo. Inoltre tutto ciò genera un problema di ridondanza che può portare a problemi di consistenza in un sistema software distributo. Inoltre si potrebbero creare problemi di spazio poichè se molti utenti richiedono spesso nuovi nodi si crea molto overhead nel dovere riportare molte informazioni molte volte che porterebbero ad un aumento considerevole delle dimensioni nel file.
Un altro problema che affliggeva tale metodo di salvataggio dei dati è che non era possibile richiamare altri tuple centre se non il primo, ren-

dendo vana quindi la possibilità di poterne aggiungere altri. Abbiamo poi aggiunto la possibilità in fase di richiesta dell'utente di un nuovo nodo di mandare come dato, oltre che l'uisername e la password, anche un nome logico del tuple centre che vuole. Ora il file xml inviato al Manager ha forma:

```
<new-node>
   <username>username</username>
   <password>password1</password>
   <tuple_centre_name>nome tuplecentre</tuple_centre_name>
</new-node>
```

Ora nella classe Manager tramite la funzione assignPort, che controlla se ci sono utenti senza porta identificati dal campo "port" inizializzato a "NP", si assegna la prima porta TCP disponibile. Tale porta, ricordiamo, è comunicata al Manager dalla classe SendPort. Una bozza del codice usato per questa aggiungere il tuple centre nuovo all'account già esistente è:

```
public boolean addTupleCentretoExistingUser(final String username, final String tuple_centre_name) {

final Document doc = this.getRegistry();
final Element root = doc.getDocumentElement();

 final NodeList accounts2 = root.getElementsByTagName("tuple_centre_name");

 for (int i = 0;accounts2.item(i) != null; i++) {
  if(!accounts2.item(i).getTextContent().equals(tuple_centre_name) && accounts2.item(i)
    .getParentNode().getParentNode().getTextContent().contains(username)){

     Element tuple_centre_name2 = doc.createElement("tuple_centre_name");

     tuple_centre_name2.setTextContent(tuple_centre_name);

     accounts2.item(i).getParentNode().insertBefore(tuple_centre_name2, accounts2.item(i));

}
```

Invece per assegnare la porta nel nuovo modello dell'XML:

```
public synchronized boolean assignPort(final String port) throws InterruptedException {

    final Document doc = this.getRegistry();

    final Element root = doc.getDocumentElement();

            final NodeList accounts = root.getElementsByTagName("port");

             for (int i = 0; accounts.item(i) != null; i++) {
                    if(accounts.item(i).getTextContent().equals("NP")){
                    accounts.item(i).setTextContent(port);
             }
     }
```

5.2.1 Soluzione

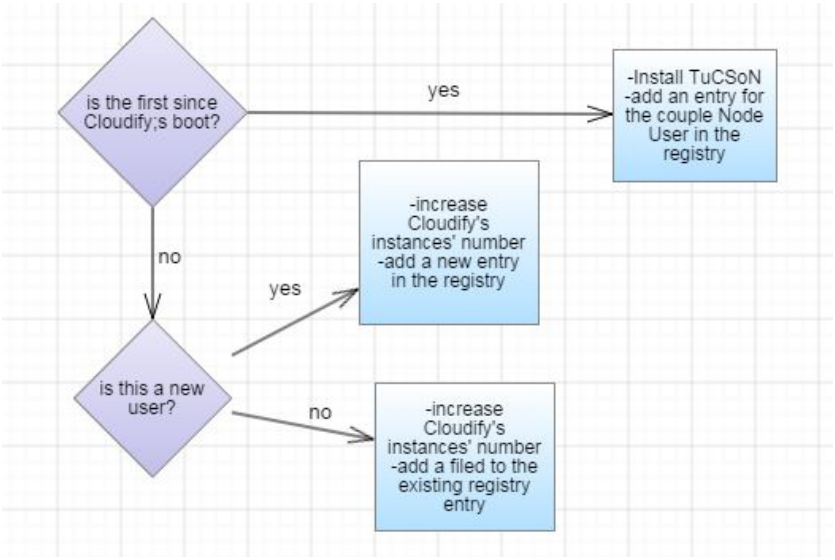

Figura 5.1: newuser

è stata aggiunta sotto la voce account, a fianco dell'username e della password, un campo node all'interno del quale sono salvati sia la porta del del nodo sia tutti i tuple centre associati all'utente.

Con questa modifica è possibile recuperare tutti i tuplecentre che l'utente ha richiesto con delle semplici interrogazioni di xml, riducendo

anche considerevolmente i tempi di accesso a tali dati, poichè non si dovrà ciclare all'interno di tutti gli account cercando il tuplcentre richiesto. Per implementare queste funzionalità ulteriori è bastato rivedere l'implementazione del metodo che scriveva nel file registry.

Il file xml ha ora la forma:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<accounts>
<account>
    <username>fabio</username>
    <password>123456</password>
        <node>
          <port>NP</port>
          <tuple_centre_name>commerciale</tuple_centre_name>
          <tuple_centre_name>amministrativo</tuple_centre_name>
    </node>
</account>
</accounts>
```

5.3 Gestione Concorrenza

All'interno dell'architettura era presente una busywait per chiedere al componente SendPort la porta TCP libera. Ovviamente una busywait, come ogni altra forma di polling, è sconsigliabile e bisogna cerca di rimuoverla e sostituirla con qualche meccanismo reale di sincronizzazione poichè in un ambiente locale i tempi possono essere circoscritti, ma in un vero scenario distribuito i tempi si possono dilatare ed è impenasibile procedere così.

5.3.1 Soluzione

Per risolvere questo problema si potrebbe creare un Semaforo Condiviso all'esterno dei package delle due calsse interessante, ovvero il Manager che aspetta la porta e la classe Client in SendPort, che l'ha ricevuta dalla recipe di Clodify, e deve segnalarla al Manager.

```
package SharedSemaphore;

public class SemaphoreClass {

    public static Semaphore;

}
```

Un altro possibile approccio protrebbe essere passare ad entrambe le classe un Object lock al momento della inizializzazione cosi le due classi non devono passarsi nulla di globale e statico.

5.4 Sicurezza

Allo stato attuale tutti i dati sono salvati in una struttura dati in xml totalmente in chiaro. E stato necessario quindi pensare ad un metodo, seppur basilare, di criptare almeno la password dell'utente. Inoltre anche tutti i dati viaggiano in chiaro, ma questo verrà risolto in futuro.

5.4.1 Soluzione

Si è pensato, come sistema di sicurezza embrionale, di utilizzare l'algoritmo di hashing SHA-256. Ovvero si è deciso di creare l'impronta della password dell'utente e di scrivere quella nel file registry cosicchè non sia intellegibile dall'esterno. Ogni volta che si dovrà riutilizzare la password per confronto non si dovrà fare altro che convertire nuovamente la password appena immessa dall'utente e controllare se l'impronta creata è uguale a quella nel database, se lo è allora l'utente può autenticarsi. Sebbene sia un controllo primitivo e non di reale criptaggio dei dati, è comunque un passo avanti verso la sicurezza quando questo sistema diventerà realmente distribuito. Qui un esempio del codice:

```
public static String SHA_256(String SHA_256) throws UnsupportedEncodingException {
try {
 java.security.MessageDigest md = java.security.MessageDigest.getInstance("SHA-256");
 byte[] array = md.digest(SHA_256.getBytes("UTF-8"));
 StringBuffer sb = new StringBuffer();
 for (int i = 0; i < array.length; ++i) {
  sb.append(Integer.toHexString((array[i] & 0xFF) | 0x100).substring(1,3));
  }
 return sb.toString();
 } catch (java.security.NoSuchAlgorithmException e) {
 }
 return null;
}
```

Il vantaggio di questa semplice modifica è che ora la password che appare nel database non è più in chiaro. Un altro vantaggio che la "codifca" SHA-256 porta è che crea un file di lunghezza fissa e finita, appunto 256 bit o 32 byte, che non può essere ristrasformata nella password originaria poichè è una funzione "a senso unico". Come accennato prima per controllare l'effettiva identità dell'utente mostro ora una esemplificazione di tale metodo, ovvero quello di far reimmettere la password

all'utente, codificarla nuovamente e controllare che essa sia la stessa nel database. Questo è possibile data l'unicità della funzione SHA-256.

```
String crypto = null;
try {
crypto = SHA_256("password");
 } catch (UnsupportedEncodingException e) {

  e.printStackTrace();
}
System.out.println("password e' " + crypto );
boolean true1 = false;
try {
 true1 = SHA_256("password").equals(crypto);
} catch (UnsupportedEncodingException e) {

 e.printStackTrace();
}
System.out.println(" e' "+ crypto + " == password? " + true1);

}

}
```

Quindi per salvare la password utente, al momento della creazione della struttura si dovrà operare nel seguente modo:

```
public boolean addNewUser(final String username, final String password,
                          final String tuple_centre_name) {


 final Document doc = this.getRegistry();
 final Element root = doc.getDocumentElement();
 String CryptedPassword =  SHA_256(password);
 final Element account = doc.createElement("account");
 final Element uname = doc.createElement("username");
 uname.setTextContent(username);
 final Element psw = doc.createElement("password");
 psw.setTextContent(CryptedPassword);
 final Element node = doc.createElement("node");
 final Element port = doc.createElement("port");
```

```
port.setTextContent("NP");
final Element tip= doc.createElement("tip");
tip.setTextContent("tip");
final Element tuplecentrename = doc.createElement("tuple_centre_name");
tuplecentrename.setTextContent(tuple_centre_name);

            root.appendChild(account);
            account.appendChild(uname);
            account.appendChild(psw);
            account.appendChild(node);
            node.appendChild(port);
           // node.appendChild(tip);
            node.appendChild(tuplecentrename);


}
```

Capitolo 6

Progetti per il futuro

Come progetto futuro si è pensato all'implementazione di un vero database per la gestione di tutti gli utenti, poichè un file xml non è performante quanto una vera e propria struttura di un database. Questo permetterà anche di centralizzare tali informazioni rendendole più facilmente fruibili, senza i problemi di consistenza che l'uso di un semplice file xml comporta nell'uso distribuito. Un altro aspetto sul quale ci si dovrà soffermare è quello della distruzione dei nodi creati e della cancellazione degli utenti una volta immessi nel registro/database. Infatti ad oggi non è possible rimuovere un nodo se non manualmente cancellandolo da cloudify con un >teardown-localcloud e per eliminare un utente si deve cancellare la sua entry manualmente dal xml. Successivamente sarà necessario prevedere una vera e propria forma di sicurezza, utilizzando forme di criptografia sia per il salvataggio dei dati nel database ma anche, se si riterrà necessario, criptare tutti i dati che viaggiano nella rete. Questa decisione sarà da valutare attentamente poich'e ogni sistema di sicurezza e di criptografia introduce un overhead considere-

vole e un aumento delle risorse richiesto, aspetto non trascurabile se i nostri agenti operano su dispositivi mobili dove le risorse sono limitate.

Capitolo 7

Conclusioni

L'obiettivo principale di questa tesi è stato quello di rivedere l'architettura attuale di TuCSoN on Cloud correggiendola e migliorandola laddove fosse necessario. Si è risolto il problema della memorizzazione di più tuple centre per singolo utente, quindi del salvataggio di tali dati nel datbase e del metodo usato; si è poi affrontato il problema della gestione della concorrenza per la sincronizzazione nel caso passaggio della porta TCP e infine si è avanzata una possibile soluzione per la questione della sicurezza, ovvero del salvataggio della password dell'utente nel database. Ovviamente il lavoro fin qui svolto va visto come una base da cui continurare per migliorare per rendere TuCSoN ancora più fruibile.

Bibliografia

[1] Mariani Stefano, Omicini Andrea, " TuCSoN on Cloud: An Event-driven Architecture for Embodied / Disembodied Coordination" in "Algorithms and Architectures for Parallel Processing", 2013

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia, " A View of Cloud Computing" in "Communications of the ACM", 2010

[3] Omicini Andrea, Zambonelli Franco,"Tuple Centres for the Coordination of Internet Agents" in "1999 ACM Symposium on Applied Computing (SAC'99)", 1999

[4] Omicini Andrea, Denti Enrico,"From Tuple Spaces to Tuple Centres" in "Science of Computer Programming", 2001

[5] www:cloudifysource:org=guide;

[6] Gelernter David,"Generative Communication in Linda" in "ACM Transactions on Programming Languages and Systems", 1985

[7] Omicini Andrea "Formal ReSpecT in the A&A Perspective" in "Electronic Notes in Theoretical Computer Science", 2007

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

[9] Richardson Leonard and Ruby Sam, "RESTful Web Services", 2007

Ringraziamenti

Voglio qui ringraziare la mia famiglia, insostituible sostegno, e tutti i miei amici che mi sono stati vicino in questo periodo.