Progetto sistemi distribuiti
TuCSoN on Cloud (Web Service)

Luca Guerra

January 18, 2015

1 Introduzione

TuCSoN on Cloud nasce con lo scopo di portare al middleware TuCSoN i benefici del Cloud Computing. In particolare:

- Facile scalabilità: rendere più robusto il sistema in risposta a cambiamenti di carico di lavoro o presenza di failure.
- Facile distribuzione: testare, con costi contenuti, TuCSoN in un ambiente globale distribuendo i nodi in diverse server farm ognuna con un certo numero di nodi.
- Distacco dalla macchina fisica: non è necessario preoccuparsi della macchina o insieme di macchine fisiche sulle quali girano i nostri nodi.

Il primo prototipo di TuCSoN on Cloud prevedeva la distribuzione dei nodi sul cloud tramite il servizio Cloudify. Ogni client doveva registrare un account attraverso il servizio NodeManager, il quale creava un nuova istanza di nodo TuCSoN in ascolto ad una certa porta; il client poteva coordinarsi tramite l'ACC-Cloud, un'estensione dell'ACC con la capacità di richiedere la porta di uno specifico account.
Questo approccio aveva problemi di sicurezza: per prima cosa tutti conoscevano l'IP del cloud, quindi bastava provare tutte le possibili porte per entrare e utilizzare nodi di altri account.
Il sistema mostrava anche problemi di apertura, in quanto il client comunicava con il nodo tramite il suo ACC Cloud, questo accoppiamento legava le applicazioni client al jar di TuCSoN e quindi al linguaggio Java.

2 Nuova versione del servizio

Per risolvere i problemi della vecchia versione è stato necessario isolare i nodi TuCSoN dai client. La separazione di queste entità rende sicura e gestibile la

coordinazione, incapsulando l'accesso diretto ai nodi e controllando le richieste di coordinazione dei client. Questa separazione ci consente anche di slegare TuCSoN on Cloud dalla tecnologia con cui vengono sviluppate le applicazioni client.
Tutto questo viene svolto da un livello di astrazione in grado di gestire la fase iniziale di creazione account e di smistare la coordinazione distribuita nei vari nodi cloud. Tale ruolo viene realizzato da un Web Service RESTful, il quale rende accessibile il servizio TuCSoN on Cloud tramite chiamate HTTP.

3 Architettura

TuCSoN on Cloud è un sistema distribuito composto da un insieme di client coordinati e da un Web Service che gestisce N Account collegati a N nodi TuCSoN.

Ogni client è un processo nella rete che vuole coordinarsi tramite il Web Service, per fare questo deve loggandosi ad uno degli account registrati nel sistema. Il Web Service gestisce N account e, grazie ad un unico ACC, intermedia l'interazione con i vari nodi. Nel sistema è presente sempre un unico agente TuCSoN istanziato staticamente dal Web Service. Il legame fra un utente (identificato da username e password) e il suo unico nodo è realizzato da un account.
NB: Ogni account collega un unico utente ad un unico nodo.

L'architettura è suddivisa nei seguenti livelli:

- Client: utilizza la coordinazione offerta dal livello Web Service.
- Web Service: gestisce l'interazione con i nodi.
- Nodi TuCSoN: classici nodi TuCSoN.

Per ogni sistema che sfrutta TuCSoN on Cloud è presente un account registrato sul Web Service e vari client che si coordinano grazie a questo.
I nodi sono dei classici nodi TuCSoN e vengono programmati tramite i propri centri di tuple. 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.

4 Interfaccia REST

Il modello scelto per il Web Service è quello RESTful (Reppresentational state transfer), tramite questo modello è possibile aumentare l'apertura del servizio, rendendolo interrogabile da qualsiasi dispositivo connesso alla rete tramite chiamate HTTP.
Le operazioni rese disponibili dal servizio sono:

- Richiedere un nuovo account
- Loggarsi al servizio

- Richiedere primitive TuCSoN

Possiamo notare una netta separazione in due famiglie di operazioni: operazioni di gestione account e operazioni di coordinazione con il nodo TuCSoN. Distinguendo ogni categoria con uno specifico url, se realizziamo un POST all'url "/manager" comunicheremo operazioni di gestione, mentre all'url "/proxy" andremo a interagire con il nodo TuCSoN associato al nostro account
NB: in questa versione avremo solo le primitive TuCSoN, non saranno presenti le primitive per caricare le tuple Respect.

4.1 Richiedere un nuovo account

Quando si vuole creare un nuovo nodo Cloud è necessario registrare un nuovo account. Per fare ciò è necessario passare al Web Service un file xml con la seguente struttura:

```
<new-node>
   <username>username prova</username>
   <password>password prova</password>
</new-node>
```

Alla ricezione di questo file il servizio controlla se lo username è disponibile, in seguito salva il dato su disco (in un file xml di nome Registry). Salvato il dato si richiede la creazione di una nuova istanza del servizio Node a Cloudify, la recipe 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.
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. Per gestire la comunicazione con Cloudify ho creato un livello di astrazione chiamato CloudifyAccessLayer con il compito di gestire l'interazione con Cloudify.
NB: Sgravando i client dell'operazione di chiedere la porta per parlare direttamente con un Nodo TuCSoN ho aumentato la sicurezza, infatti il proxy per le primitive di TuCSoN rende trasparente l'accesso al nodo (il client può ignorare il fatto che il nodo è distribuito in un'infrastruttura cloud).

4.2 Loggarsi al servizio

Per rendere la comunicazione minimale e sicura ho 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.

Per loggarsi al servizio è necessario inviare un xml con il seguente tracciato:

```
<log-in>
   <username>username prova</username>
   <password>password prova</password>
</log-in>
```

4.3 Richiedere primitive TuCSoN

Le primitive TuCSoN si distinguono in due grandi famiglie:

- Non bloccanti
- Bloccanti

Per l'accesso ai vari nodi ho creato un Layer di astrazione (**NodeAccessLayer**). Questo livello, per accedere al nodo TuCSoN, utilizza la composizione con la classe **WsACC**. La classe WsACC incapsula un AsynchACC, lo inizializza e lo espone. La classe è statica, in questo modo è sempre presente un'unica istanza di ACC per tutte le primitive richieste al Web Service.
La scelta di rendere statica la classe è particolarmente strategica, ricordando che, per ottenere un ACC devo avere un Agente TuCSoN e che questo è caratterizzato da un nome univoco, se per ogni chiamata inizializzo un nuovo ACC devo creare ogni volta nomi diversi, altrimenti in casi di concorrenza potrei avere più primitive parallele richieste dallo stesso agente TuCSoN che manderebbero in stallo in nodo.
Inoltre, se questa classe non fosse statica porterebbe ad una gestione non efficiente della creazione degli ACC, inducendo il Web Service ad occupare inutilmente memoria per mantenere più istanze parallele di ACC.

4.4 Primitive non bloccanti

Fra queste riconosciamo:

- out: inserimento di una tupla

```
<out tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</out>
```

- inp: se trova la tupla la legge e la rimuove

```
<inp tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</inp>
```

- rdp: se trova la tupla la legge

```
<rdp tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</rdp>
```

- get: lettura di tutte le tuple

```
<get tc="tuplecentreID"></get>
```

- set: sovrascrittura di tutte le tuple

```
<set tc="tuplecentreID">
   <tuple>tuple-set</tuple>
</set>
```

Tutte queste primitive sono caratterizzate da una risposta immediata. Quando chiediamo una primitiva questa ha subito un esito che viene comunicato al client, senza incorrere in timeout di richiesta.

4.5 Primitive bloccanti

Fra queste riconosciamo:

- in : legge e rimuove la tupla, se questa non è presente aspetta la sua comparsa nel centro

```
<in tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</in>
```

```
<in-result tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</in-result>
```

- rd : legge la tupla, se questa non è presente aspetta la sua comparsa nel centro

```
<rd tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</rd>
```

```
<rd-result tc="tuplecentreID">
   <tuple>template-tupla</tuple>
</rd-result>
```

La gestione di questa categoria risulta più complessa. La pagina può andare in timeout e, in linea di massima, non è consigliato mantenere una connessione aperta per un tempo indefinito.
Per risolvere il problema ho pensato ad un workflow diviso in archi temporali differenti:

- il client comunica l'intenzione di una primitiva bloccante e gli viene subito restituito un ack di risposta
- il client a polling richiede il risultato

L'idea è quella di simulare il comportamento di un SynchACC, una primitiva bloccante blocca, appunto, la chiamata finché non viene risolta la primitiva. Per far ciò ogni richiesta è associata ad una certa sessione, il Web Service non consente nessuna altra richiesta dalla stessa sessione finché la primitiva bloccante non è stata risolta. Come detto in precedenza, dobbiamo rispondere subito con un ack al client, quindi non possiamo evocare la primitiva di coordinazione dal thread che si occupa della richiesta, altrimenti questo si bloccherebbe e di conseguenza andrebbe in timeout la connessione.
Per evitare ciò, il thread della richiesta crea un nuovo thread rappresentato dalla classe **AsynchAgent** che si occuperà di eseguire la primitiva tramite il NodeAccessLayer. Quando il dato sarà disponibile verrà inserito nella sessione corrispondente e alla successiva richiesta del client verrà servito.

Schema sequenziale:

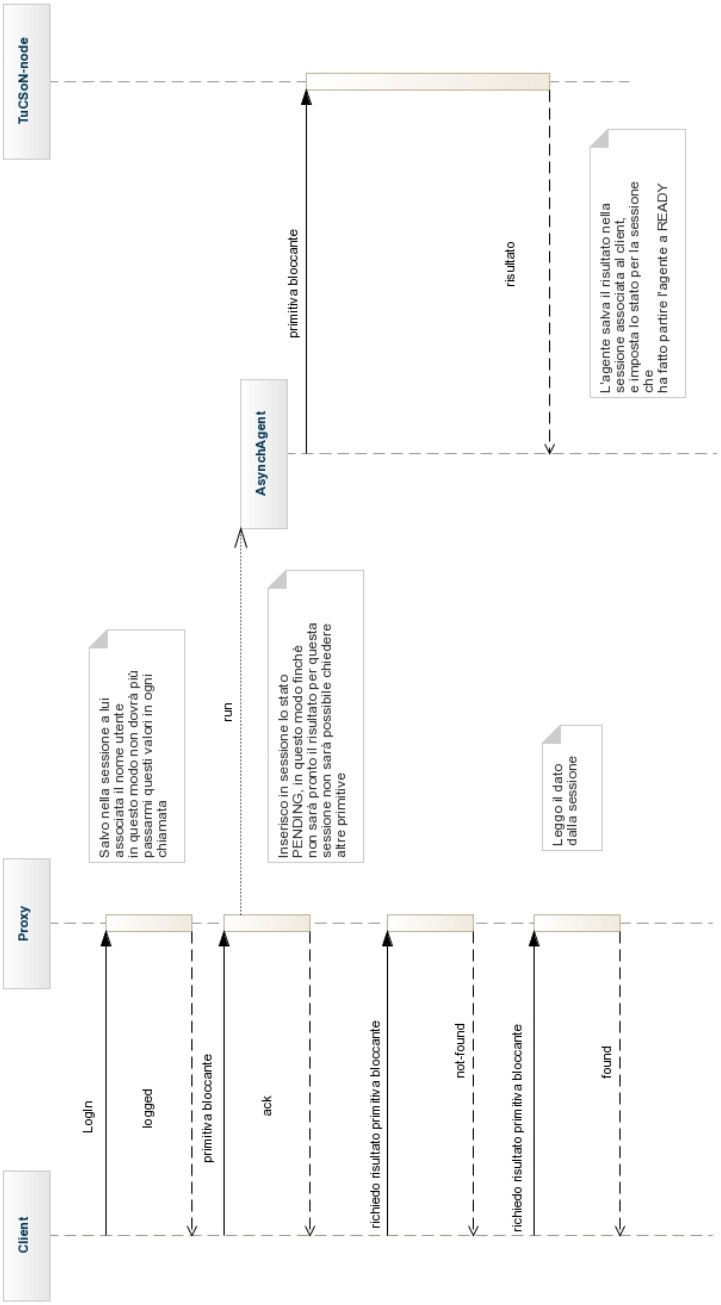

Figure 1: sequenziale primitiva bloccante

5 accesso al file Registry

Per rendere trasparente l'accesso e disaccoppiare la parte di storage delle informazioni, ho deciso di sfruttare un'astrazione di nome RegistryAccessLayer. Questa ha il compito di accedere direttamente al file fisico sulla macchina, quando sarà necessario cambiare il metodo o il mezzo di salvataggio dei dati basterà modificare questa classe, il Web Service rimarrà trasparente al cambio.

6 Caso d'uso

Per testare il Web Service ho creato un progetto client di testing.
Questo contiene sia il caso d'uso specifico che delle classi di utilità per la comunicazione con il Web Service.
Nel caso d'uso ho voluto simulare la coordinazione per l'accesso ad un file.
Ho creato due tipi di agenti, Scrittore e Lettore, il primo deve avere accesso mutualmente esclusivo alla risorsa, mentre gli altri possono leggere in parallelo fra loro, ma non possono leggere quando uno scrittore sta scrivendo.
Nella versione attuale del Web Service non è possibile caricare primitive Respect nei vari centri di tuple, quindi la coordinazione è basata sulle sole primitive TuCSoN, in particolare:
Uno scrittore prima di iniziare a scrivere sulla risorsa richiede una "in" di "res(testo)", dove *res* identifica una risorsa e *testo* il nome di quest'ultima, grazie alla "in", viene eliminata la tupla che rappresenta la risorsa, in questo modo vengono bloccate eventuali "rd" o nuove "in". Terminato il suo compito, lo scrittore deve reinserire la tupla con una "out". I vari lettori devo realizzare delle "rd" della stessa tupla, queste ritorneranno quando il template sarà presente nel centro di tuple.
Purtroppo per lo scrittore non è possibile disaccoppiare la gestione della coordinazione, questo è possibile solo sfruttando le tuple Respect di programmazione del centro di tuple.

Schema sequenziale:

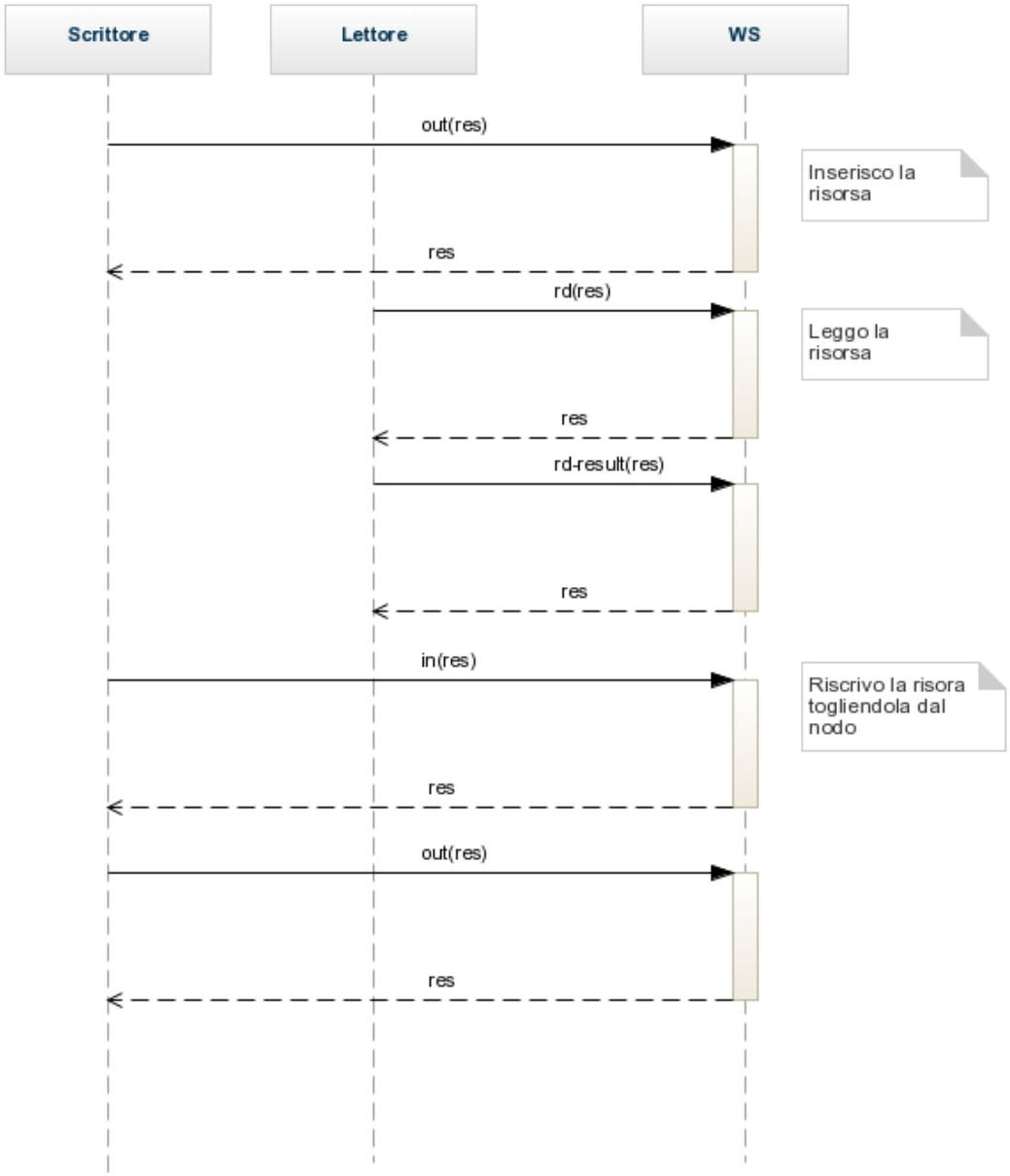

Figure 2: sequenziale caso d'uso

6.1 Esempio passo passo

Per prima cosa è necessario simulare un cloud locale, per fare questo apriamo la console di Cloudify; se siamo in ambiente windows la cartella di Cloudify deve essere posta nella root, ad esempio, nel mio caso in C.

1. apriamo la console di Cloudify e digitiamo il comando: *bootstrap-localcloud*, questo crea un ambiente cloud simulato nella macchina.
2. creiamo un file Registry.xml per ospitare le informazioni dei vari account. La struttura iniziale deve essere la seguente:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><
    accounts></accounts>
```

3. settiamo il nuovo Path del file Registry nel RegitryAccessLayer.
4. eseguiamo il Web Service TucsonOnCloudRESTful.
5. lanciamo il TestingEnv del progetto Test per vedere in azione la coordinazione as a service.

7 Sviluppi futuri

7.1 Mantenimento dati account

In questa versione viene sfruttato un semplice file xml per mantenere traccia dei vari account, uno sviluppo futuro potrebbe essere quello di creare un DB per avere una gestione dei dati in maniera più sicura e strutturata.

7.2 Timeout primitive bloccanti

Si potrebbe pensare di estendere il tracciato delle primitive bloccanti con la proprietà timeout, questa possibilità è già presente per gli ACC.

7.3 Programmazione centri di tuple

Dare anche la possibilità di programmare i centri di tuple caricando gli script Respect.

7.4 Eliminazione account

Trovare un modo per eliminare account e deallocare istanze di nodi distribuiti sul cloud.

7.5 Distribuire in un cloud reale

Non sfruttare il localcloud ma agganciare Cloudify ad un provider come Amazon o Azure per distribuire e testare il sistema in un ambiente reale.

7.6 Piano di acquisto

Estendere il tracciato di creazione "nuovo account" con varie proprietà che specificano il piano di acquisti scelto per l'account.

7.7 Sito pubblico TuCSoN

Il Web Service potrebbe offrire in GET pagine web in cui tramite log-in, mostrare ai vari user lo stato dei propri nodi, le tuple presenti e lo stato del proprio abbonamento TuCSoN on Cloud. Data l'apertura di cui gode TuCSoN è possibile far eseguire primitive anche da un pannello di comandi remoto mostrato nelle pagine web.

7.8 Migliorare la sicurezza

Si può pensare di sfruttare la crittografia per mascherare il contenuto della request passata.