Distributed and fault-tolerant Web Crawler with Raspberry Pi

Benedetti Riccardo, Ramilli Elisabetta

Laurea magistrale in Ingegneria e Scienze Informatiche
Anno accademico: 2015 - 2016
Alma Mater Studiorum – Università di Bologna
via Sacchi 3, 47521 Cesena, Italia
{riccardo.benedetti3, elisabetta.ramilli }@studio.unibo.it

Keywords: WebCrawler, MAS, Distributed, Fault-tolerance, Resource-efficient, Raspberry Pi, Tucson, Jade, T4J

Table of Contents

1 Introduction

In questo documento viene descritto il software prodotto come progetto per l'esame di Sistemi Distribuiti.

2 Goals

Il goal del progetto è il seguente:
building a distributed web crawling infrastructure, whose functionality is to fetch publications' metadata from the APICe online repository, provided a set of keywords given for searching.

3 Vision

La prospettiva è la creazione di un sistema multi-agente (MAS) che soddisfi certi requisiti. Tale sistema dovrà sfruttare i meccanismi di coordinazione del middleware Tucson e gli agenti dovranno essere sviluppati con il framework Jade.

3.1 MAS

Un MAS prevede la molteplicità degli agenti all'interno del sistema. Un agente è un'entità autonoma che ha lo scopo di portare a termine un compito interagendo con gli altri agenti (society) e con l'ambiente circostante (environment). Le sue caratteristiche principali sono:

Autonomia Gli agenti sono autonomi in quanto incapsulano il flusso di controllo. Il controllo non oltrepassa i confini dell'agente, solo i dati (conoscenza, informazioni) possono farlo. Gli agenti non hanno un'interfaccia, non possono essere controllati o invocati. Quindi un MAS può essere considerato come un'aggregazione di multipli punti di controllo distinti che interagiscono tra loro scambiandosi informazioni.

Proattività Gli agenti sono proattivi in quanto fanno accadere qualcosa, non aspettano che qualcosa accada.

Interattività L'autonomia ha senso solo se l'individuo è immerso in una società. Quindi gli agenti vivono e interagiscono all'interno di società di agenti, ovvero i MAS. Gli agenti vengono considerati quindi come sottosistemi dei MAS e interagiscono all'interno del sistema globale scambiandosi informazioni e conoscenza

Reattività Gli agenti possono essere reattivi semplicemente per il fatto che reagiscono a eventi esterni o aspettano che accada qualcosa. Possono però anche essere reattivi al cambiamento, nel senso di avere una percezione dell'ambiente (ad esempio controllano pre-condizioni allo svolgimento di azioni o verificano gli effetti delle azioni sull'ambiente).

Situatedness Gli agenti sono intrinsecamente situati in quanto sono strettamente collegati all'ambiente in cui risiedono e agiscono.

3.2 Interaction and Coordination in a MAS

In un MAS è necessario un modello di coordinazione che svolga il ruolo di "collante" per collegare attività separate in un "ensamble". Un modello di coordinazione viene visto anche come framework per l'interazione tra agenti e deve coprire questioni come la creazione e distruzione degli agenti, la loro comunicazione, la loro distribuzione spaziale, la sincronizzazione e la distribuzione delle loro azioni nel tempo.
Gli agenti sono immersi in uno spazio delle interazioni che viene quindi "riempito" dal mezzo di coordinazione (coordination medium), il quale abilita, promuove e governa le interazioni ammissibili, desiderabili e richieste tra le entità interagenti in base a leggi di coordinazione.

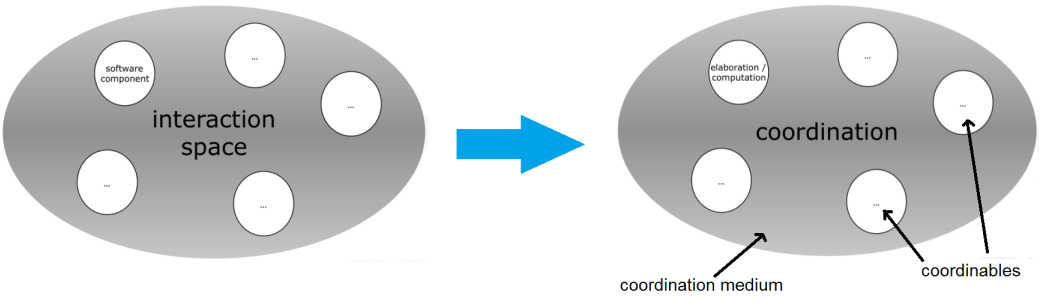


Fig. 1. Dallo spazio delle interazioni al mezzo di coordinazione

3.3 TuCSoN

Il modello di coordinazione utilizzato nel sistema oggetto di questo report è TuCSoN (Tuple Centres Spread over the Network).
È un modello per la coordinazione di processi distribuiti e di agenti autonomi, intelligenti e mobili ed è basato su tuple: collezioni ordinate di elementi informativi possibilmente eterogenei.
In breve, il sistema TuCSoN è definibile come una collezione di **agenti** e **centri delle tuple** che collaborano all'interno di un insieme di **nodi** possibilmente distribuiti.

- Gli agenti di TuCSoN sono i *coordinables*; essi vivono in una qualunque parte della rete e interagiscono con il centro delle tuple.
- I centri delle tuple ReSpecT sono i mezzi di coordinazione; ogni centro delle tuple appartiene a un nodo.
- I nodi di TuCSoN rappresentano l'astrazione topologica di base; ogni nodo è identificato da una coppia <NetworkId, PortNo> dove NetworkId è l'indirizzo IP del dispositivo che ospita il nodo, mentre PortNo è il numero di porta

dove il "Tucson Coordination Service" è in attesa di richieste. La porta di default per tucson è 20504.

3.3.1 TuCSoN Coordination operations Gli agenti, per interagire con il centro delle tuple, usano il *coordination language* che è basato sulle primitive dello spazio delle tuple ovvero un insieme di operazioni per inserire, navigare e recuperare le tuple dallo spazio.
Le operazioni di coordinazione sono costruite in base alle suddette primitive e ai due linguaggi di comunicazione: quello delle tuple e quello dei tuple template. Entrambi i linguaggi sono logic-based, così come il centro delle tuple di ReSpecT. Più precisamente, ogni atom di Prolg è una tupla TuCSoN ammissibile e ogni atom di Prolog è un tuple template TuCSoN ammissibile.
Le operazioni di coordinazione sono invocate da un agente sorgente su di un centro delle tuple target e si compongono di 2 fasi: invocazione (la richiesta dell'agente con tutte le informazioni) e completamento (risposta del centro delle tuple all'agente con le informazioni sull'esito dell'operazione).
Alcune operazioni sono:

- in(TupleTemplate): cerca nel centro delle tuple indicato una tupla che faccia matching con il TupleTemplate; se la trova, la ritorna rimuovendola dal centro delle tuple, altrimenti l'esecuzione viene sopsesa e viene ripresa e completata quando viene trovata una tupla che faccia matching.
- out(Tuple): scrive la tupla indicata come parametro nel centro delle tuple indicato come parametro; se l'operazione ha successo, la tupla viene ritornata come completamento.
- rd(TupleTemplate): cerca nel centro delle tuple indicato una tupla che faccia matching con il TupleTemplate; se la trova, la ritorna senza rimuoverla, altrimenti l'esecuzione viene sopsesa e viene ripresa e completata quando viene trovata una tupla che faccia matching.
- inp(TupleTemplate): identica alla in ma se non si trova un matching l'esecuzione non viene sospesa bensì fallisce: nessuna tupla viene rimossa dal centro delle tuple e viene ritornato il TupleTemplate.
- rdp(TupleTemplate): identica alla rd ma se non si trova un matching l'esecuzione non viene sospesa bensì fallisce e viene ritornato il TupleTemplate.

3.4 Jade

Jade è un framework per lo sviluppo di appicazioni ad agenti.
La piattaforma ad agenti può essere suddivisa tra diversi host, ognuno dei quali funge da container di agenti. Tale container fornisce un completo ambiente runtime per l'esecuzione di agenti Jade. Uno di questi container è il "main" ed è il primo a partire. È responsabile di mantenere un registro di tutti i container della piattaforma. Tramite il main container gli agenti possono vedersi gli uni gli altri.
Nel main container troviamo: l'AMS (agent management system) che tiene traccia di tutti gli agenti nella piattaforma; il DF (directory facilitator) che tiene

traccia di tutti i servizi dichiarati da ogni agente nella piattaforma; l'RMA (remote monitoring agent) per il controllo del ciclo di vita della piattaforma e i relativi agenti.

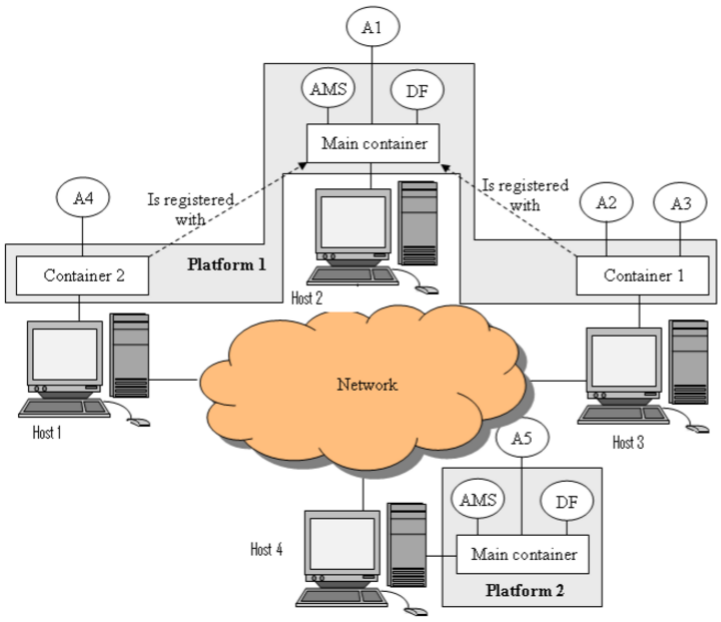


Fig. 2. Jade Architecture

Normalmente gli agenti Jade comunicano mediante scambio di messaggi asincrono usando l'ACC (agent communication channel).
Ogni agente ha una mailbox dove l'ACC consegna i messaggi inviati da altri agenti. Il ricevente viene notificato di una nuova entry (non si blocca nè fa richieste cicliche) ed è l'agente stesso (o il programmatore) a decidere quando processare il messaggio.

3.4.1 Agents and behaviours In Jade un **agente** è un oggetto java eseguito da un singolo thread ed estende jade.core.Agent (con metodi pronti all'uso). Ogni agente ha la sua autonomia e ha un *aid*, che funge da nome globale unico e viene usato per le comunicazioni: <local-name>@<jade-platform-name>. La logica di un agente è espressa in termini di *behaviour*.
Il ciclo di vita di un agente può essere rappresentato con una Finite State Machine le cui transizioni corrispondono a metodi (doActivate, doDelete, ecc.) e i cui stati possono essere: initiated, active, waiting, ecc.
Ogni agente ha un metodo setup() che serve a creare istanze dei behaviour e col-

legarli al loro agente. In questo modo l'agente ha un pool iniziale di behaviour pronti allo scheduling.

I **behaviour** vengono definiti come attività da compiere, con l'obiettivo di portare a termine un task. Vengono implementati come oggetti java eseguiti con uno scheduler round-robin non preventivo (i processi ricevono il controllo della cpu a turno, sulla base di un ordine circolare).

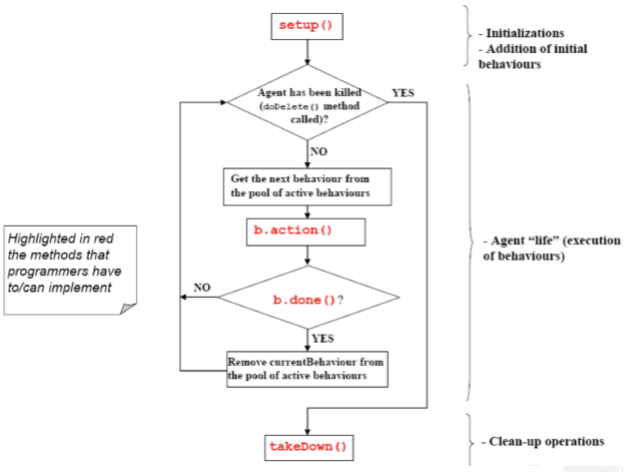


Fig. 3. La politica di scheduling non-preemptive di Jade

Ogni behaviour è istanza di jade.core.behaviours.Behaviour e ha due metodi fondamentali: action(), per eseguire il task specifico e done(), che controlla la condizione di terminazione del task.
Ci sono vari tipi di behaviour: simpleBehaviour (OneShotBehaviour o CyclicBehaviour), compositeBehaviour (SequentialBehaviour o ParallelBehaviour) ed FSMBehaviour. Quest'ultimo presenta i metodi registerState(), per aggiungere un figlio behaviour (che rappresenta l'attività da eseguire all'interno di uno stato della FSM) e registerTransition(), per aggiungere una transizione da uno stato all'altro.
È importante sapere che i behaviour vengono eseguiti uno alla volta. La rimozione di un behaviour avviene solo quando done() torna true; se torna false, il behaviour è rischedulato per il giro successivo. Non si può stoppare e poi far riprendere un behaviour: ci vuole una variabile istanza che tenga traccia dello stato computazionale. Se si vuole che un agente attenda l'arrivo di un messaggio, si deve usare block() che sospende solo il behaviour chiamante, altrimenti si può usare doWait() che mette in pausa tutti gli agenti.

L'esecuzione di un agente avviene seguendo questi step:

1. viene eseguito il suo costruttore
2. la piattaforma gli dà un aid
3. register() all'AMS
4. stato dell'agente messo su ACTIVE
5. esecuzione di setup()
6. scheduling dei behaviour

Se un behaviour chiama doDelete(), l'agente si ferma. A quel punto la piattaforma chiama il metodo astratto takedown() e avviene la de-registrazione dall'AMS; lo stato dell'agente viene messo su UNKNOWN e il thread che eseguiva l'agente viene distrutto.

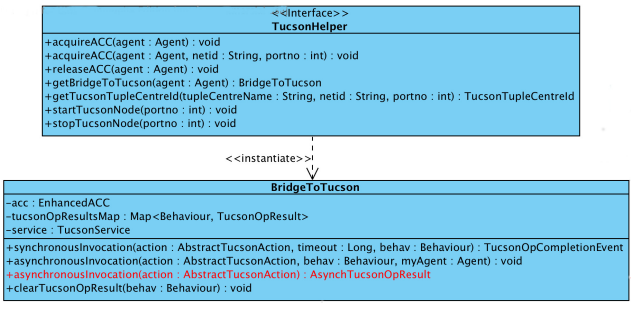
3.4.2 Jade Agents and Java Swing Per sviluppare agenti Jade che abbiano una GUI, è necessario estendere la classe GuiAgent nel package jade.gui. Dovrenno essere implementati due metodi:

onGuiEvent(GuiEvent e) può essere visto come l'equivalente del metodo actionPerformed() in Java Swing; è una callback invocata dalla piattaforma Jade non appena il GuiEvent viene generato.

postGuiEvent(GuiEvent e) è usato dalla GUI per mettere in coda i GuiEvent per un'elaborazione successiva; è simile a mettere in coda i messaggi ACL nella mailbox dell'agente.

3.5 T4J

Nel sistema oggetto di questo report, affinchè gli agenti Jade potessero sfruttare la coordinazione attraverso i centri delle tuple, ci siamo avvalsi di Tucson4Jade, un middleware che implementa TuCSoN come un servizio Jade. Questo avviene grazie al fatto che la classe BaseService di Jade è stata estesa con la classe TucsonService, che rappresenta il punto di accesso del servizio TuCSoN. Quest'ultima classe deve essere usata per ottenere una classe helper, che estende l'interfaccia ServiceHelper di Jade e che funge da mediatore tra i client e il servizio TuCSoN: in t4j, il ruolo di helper è svolto dall'interfaccia TucsonHelper.



BridgeToTucson è la classe a cui il TucsonHelper delega l'invocazione delle operazioni di coordinazione di TuCSoN. BridgeToTucson espone le seguenti API:

synchronousInvocation() permette ai client di invocare le operazioni di coordinazione di TuCSoN in modo sincrono. Ciò signifca che il behaviour chiamante viene eventualmente sospeso e automaticamente fatto ripartire da t4j non appena l'operazione richiesta si completa, ritornando l'evento di completamento, ovvero l'oggetto TucsonOpCompletionEvent.
La ricezione dei messaggi avviene in uno stile simile a quello utilizzato in Jade:

1. prima viene chiamato il metodo di comunicazione: synchronousInvocation() in t4j, receive() in JADE
2. poi il risultato viene controllato e se è disponibile viene gestito, altrimenti viene chiamato il metodo block()

Listing 1.1. Jade

```
@Override
public void action () {
	// field 'mt' stores the ACL message template
	final ACLMessage msg = myAgent.receive(mt);
	if (msg != null) { // message received: process it
		...
	} else { // message not received yet: wait
		block();
	}
}
```

Listing 1.2. t4j (estratto dal nostro codice)

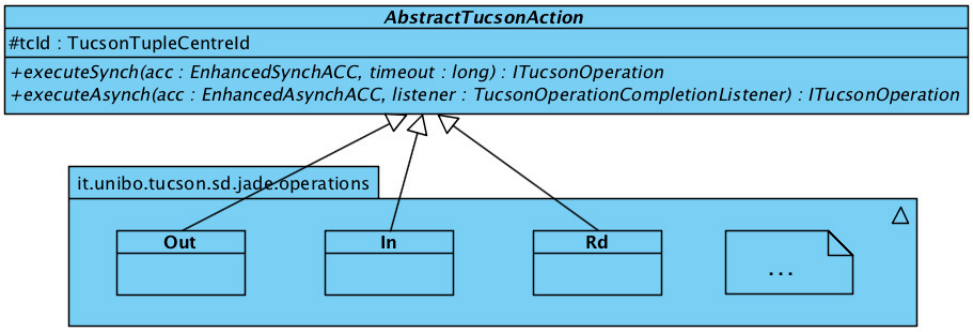
```
public void action() {
	// field 'keyword' stores the TuCSoN keyword tuple template
	TucsonOpCompletionEvent res = null;
	final In in = new In(tcid, keyword);

	res = bridge.synchronousInvocation(in, Long.MAX_VALUE, this);

	if(res != null){ //tuple found: process it
		...
	}else{ //tuple not found yet: wait
		this.block();
	}
}
```

asynchronousInvocation() permette ai client di invocare le operazioni di coordinazione di TuCSoN in modo asincrono. In questo caso, l'operazione di coordinazione può sospendersi invece l'agente no, quindi il behaviour chiamante continua l'esecuzione.

Come ultima nota, ecco la gerarchia dei tipi che rappresenta le operazioni di coordinazione di TuCSoN presenti nel package it.unibo.tucson.sd.jade.operations.



Per sfruttare i servizi di TuCSoN sono necessari alcuni passaggi, che sono stati implementati nel setup() degli agenti Master e Worker.

1. recuperare la classe helper dall'istanza del TucsonService

```
ITucsonHelper helper = (TucsonHelper) this.getHelper(TucsonService.NAME);
```

2. far partire il nodo TuCSoN sul quale si desidera operare

```
if (!this.helper.isActive(20504)) this.helper.startTucsonNode(20504);
```

3. recuperare un ACC (che in realtà è associato all'oggetto BridgeToTucson)

```
this.helper.acquireACC(this);
```

4. recuperare il bridge attraverso il quale passeranno tutte le operazioni

```
TuCSoN.BridgeToTucson bridge = this.helper.getBridgeToTucson(this);
```

5. costruire l'identificatore del centro delle tuple che si vuole usare

```
TucsonTupleCentreId tcid = this.helper.buildTucsonTupleCentreId("default",
"localhost", 20504);
```

Dopo aver fatto il setup dell'agente, si potrà procedere, in base alle esigenze, a costruire le tuple. Si dovrà poi costruire l'azione, rappresentante l'operazione

di coordinazione TuCSoN, per poi mandarla in esecuzione in base alla semantica di invocazione scelta (sincrona o asincrona).

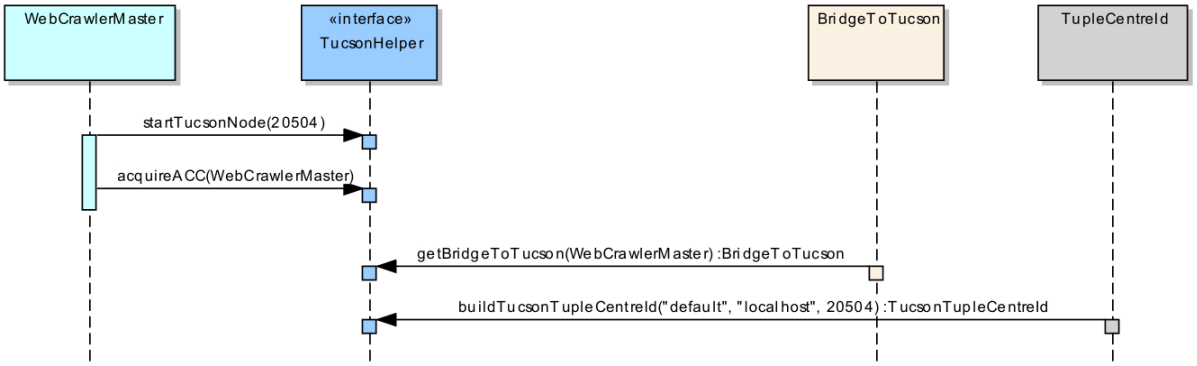


Fig. 4. Diagramma UML che mostra il setup di T4J

4 Requirements

L'infrastuttura deve essere:

Distributed and open Any number of web crawlers may be deployed on any number of networked machines, possibly even at run-time.

Fault-tolerant Both disconnections and crashes should be (i) detected as soon as possible and (ii) properly managed. E.g.: crawling tasks of the disconnected/crashed crawler re-assigned.

Resource-efficient The infrastructure should be able to execute smoothly on resource-constrained devices. E.g.: RasPi systems.

Based on Tucson and Jade Usage of either (i) the TuCSoN middleware for coordinating crawlers, or (ii) the JADE framework for programming crawlers is mandatory. Usage of both is considered a plus.

5 Problem approach and analysis

5.1 Web Crawler

Un crawler è un software che analizza i contenuti di una rete basandosi su una lista di URL da visitare fornita dal motore di ricerca. Durante l'analisi di un URL, identifica tutti gli hyperlink presenti nella pagina e li aggiunge alla lista di URL da visitare in seguito. In questo modo viene a crearsi una vera e propria "ragnatela" di pagine Internet, legate le une alle altre attraverso hyperlink. Il contenuto di questi indirizzi viene analizzato e salvato in memoria per essere poi indicizzato dal software di catalogazione associato al motore di ricerca.
Nel nostro caso, il crawler analizzerà i contenuti del sito apice.unibo.it: il suo scopo sarà quello di fornire una lista di pubblicazioni e i relativi indirizzi URL basandosi su una lista di parole chiave (titolo, autore, parole chiave della pubblicazione) fornite dall'utente tramite GUI.

5.2 Logic architecture

Il sistema è modellabile con un'architettura di tipo master/worker.
Un insieme di master prende le parole chiave inserite dall'utente nella gui e le mette nello spazio delle tuple a disposizione di un insieme di worker;
i worker si occupano di andare a cercare le occorrenze delle parole chiave nelle pubblicazioni accademiche sul sito apice.unibo.it.
La gestione della fault tolerance è delegata agli agenti WatchdogAgent e PingAgent, che controllano periodicamente, con scambio di tuple, che tutti gli agenti del sistema siano attivi.

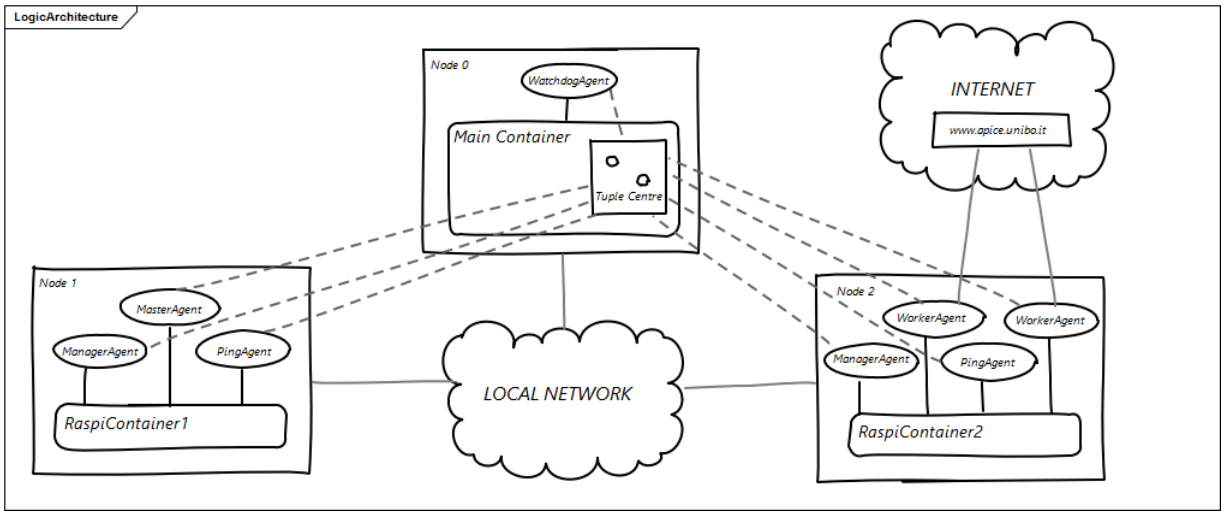


Fig. 5. Logic Architecture

6 Project

6.1 Structure

Di seguito la struttura delle classi del sistema. Ogni agente contiene vari behaviour e fa uso di classi di utilità per svolgere i vari compiti.

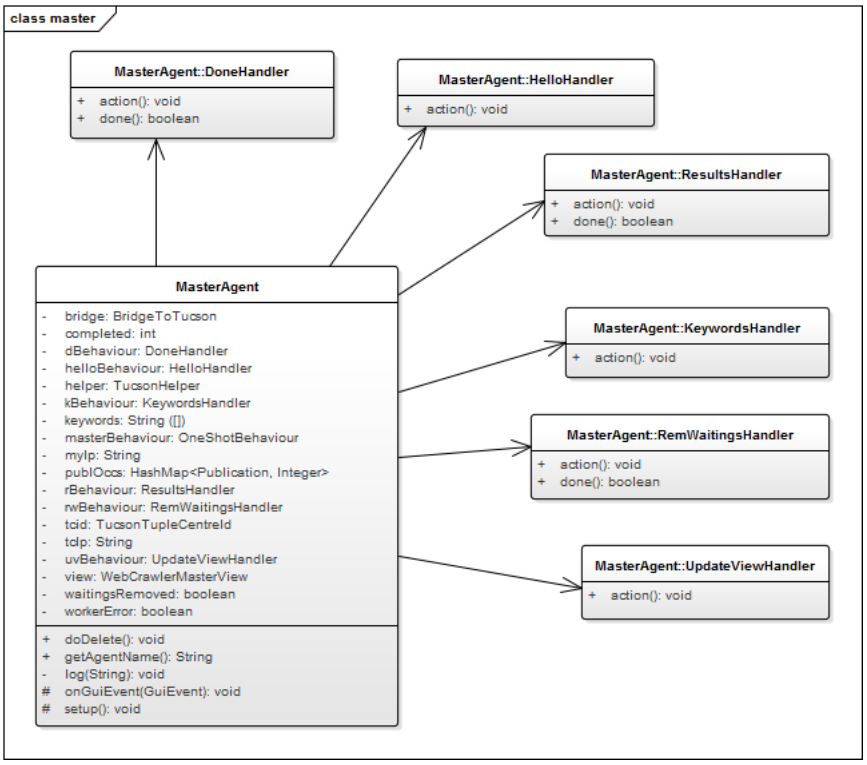


Fig. 6. MasterAgent

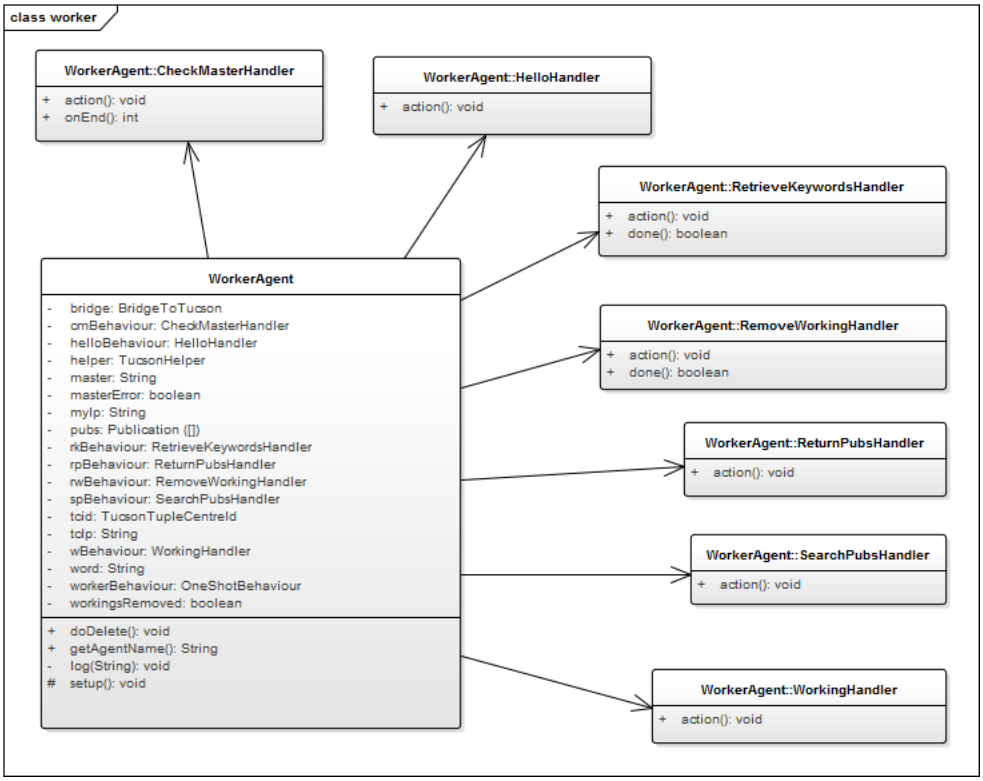


Fig. 7. WorkerAgent

Il ManagerAgent ha il compito di gestire tutti gli agenti presenti nel sistema; esso è fornito di un'interfaccia che permette di aggiungere o rimuovere master o worker con gli appositi pulsanti.

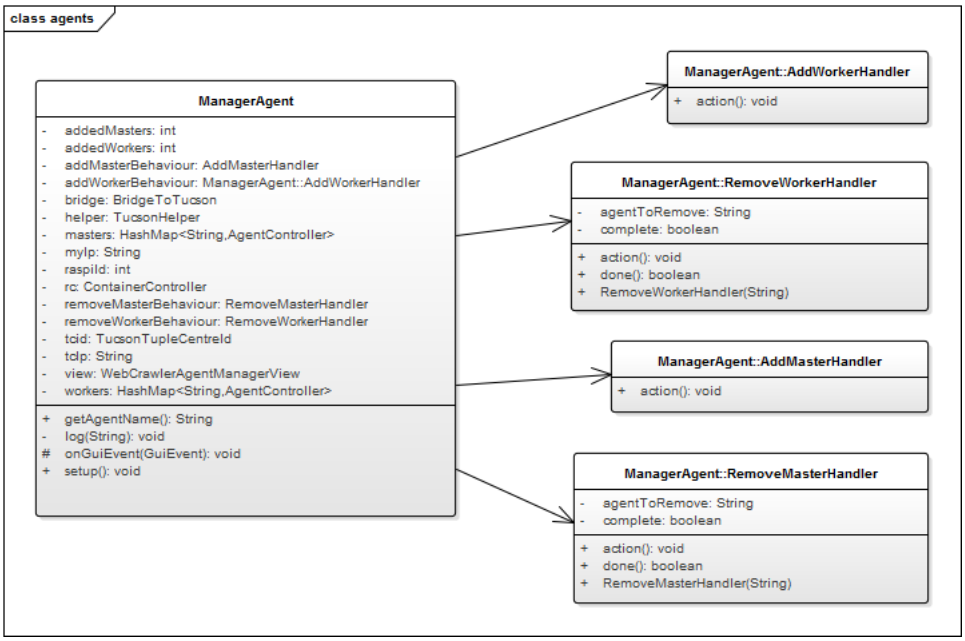


Fig. 8. ManagerAgent

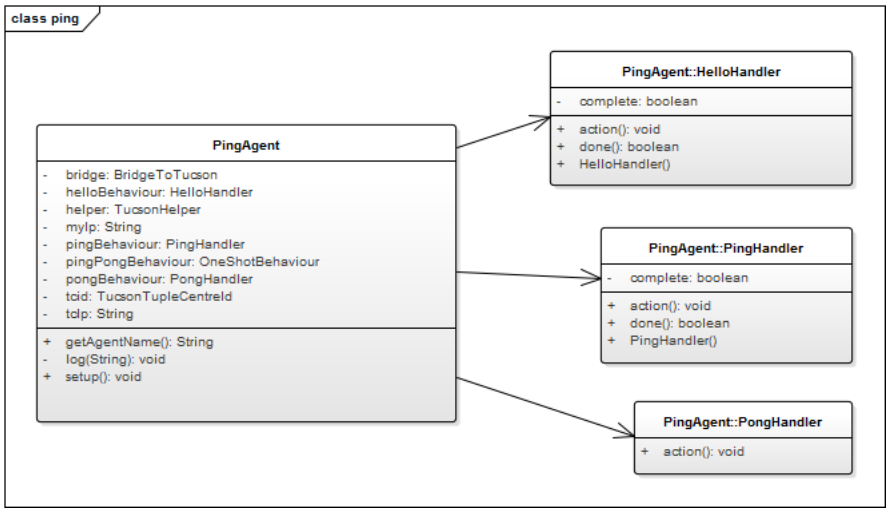


Fig. 9. PingAgent

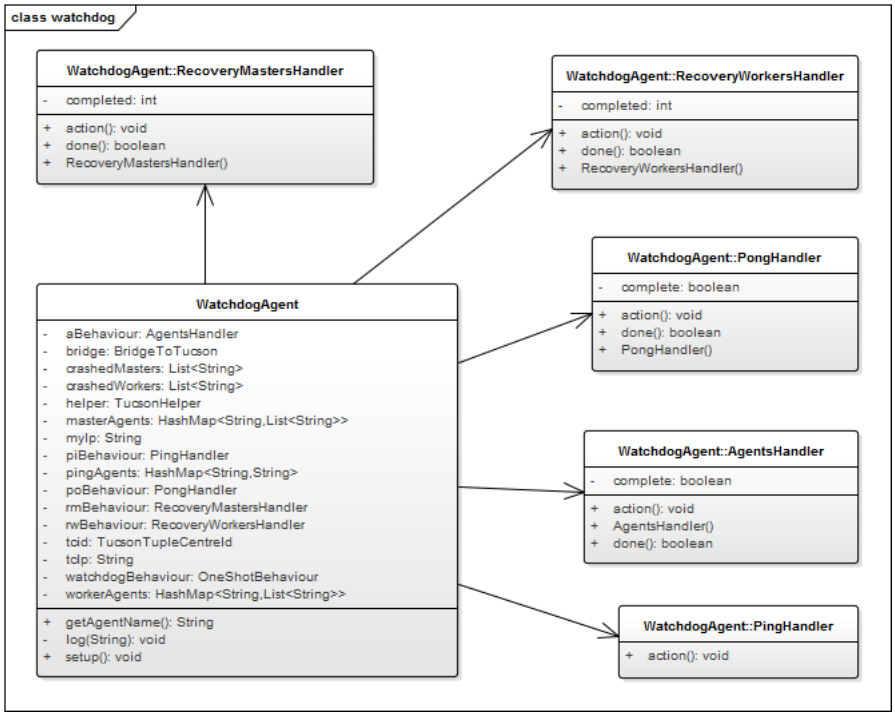


Fig. 10. WatchdogAgent

Il PubsCrawler è il cuore del sistema. Grazie a Jsoup, una libreria per lavorare con codice HTML, il crawler si connette al sito apice.unibo.it e va ad analizzare tutte le pubblicazioni, a partire dal loro indirizzo url. In particolare, di ogni pubblicazione viene recuperato il tag html "pre" (un testo pre-formattato) che contiene informazioni utili per la ricerca come titolo, parole chiave e autori della pubblicazione.
Tra le classi di utilità troviamo il PreParser, ovvero un parser del testo pre-formattato; la classe Publication, che contiene i campi title e url di una pubblicazione; la classe ValidTermFactory, che serve a creare tuple logiche valide da inserire nel centro delle tuple di Tucson.

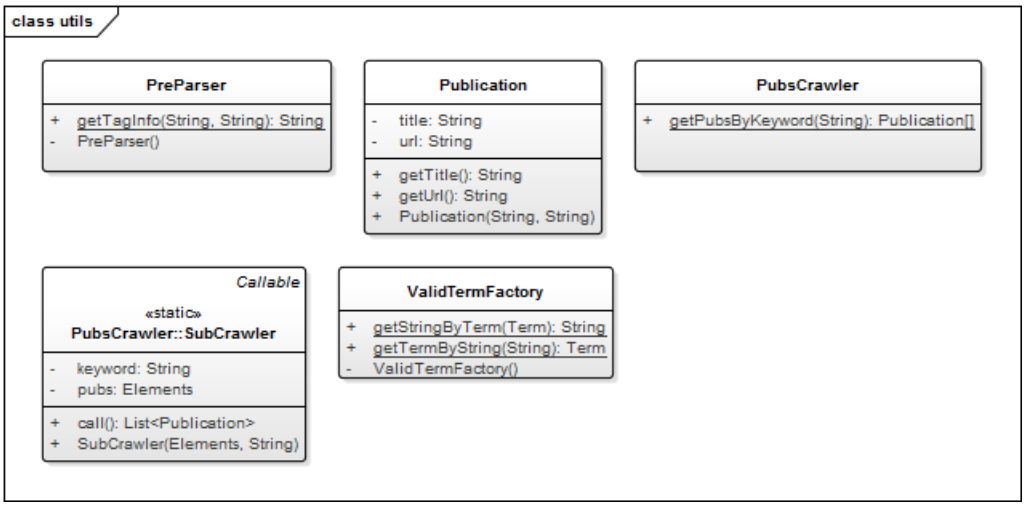


Fig. 11. Utils

6.2 Interaction

Il master e il worker comunicano grazie al centro delle tuple di Tucson. Dato che usiamo agenti di Jade ma il servizio è di Tucson, l'infrastuttura si basa su T4J. BridgeToTucson è la classe che si occupa dell'invocazione delle operazioni di coordinazione. Le tuple vengono immesse nel/recuperate dal centro delle tuple con invocazioni sincrone o asincrone.
In particolare: il master, con un'invocazione asincrona sul bridge, mette (out) nel centro delle tuple una tupla *keyword(value(V), from(M))*, che contiene la parola chiave recuperata dalla GUI e il nome del master mittente.
Il worker, con un'invocazione sincrona sul bridge, recupera (in) la parola chiave e la salva in un apposito campo *word* e la passa al PubsCrawler. Una volta terminata la computazione e ottenuti i risultati, il worker, con un'invocazione

asincrona sul bridge, mette (out) nello spazio delle tuple una tupla *publication(master(M), pub(P))* per ogni risultato trovato; essa contiene il master destinatario e la pubblicazione. Infine il worker mette, con un'altra invocazione asincrona sul bridge, nello spazio delle tuple una tupla *done(master(M))* per segnalare allo specifico master M che il worker ha finito di inviare tutte le tuple publication.
Il master, quindi, con un'invocazione sincrona sul bridge, recupera (in) la tupla *done(master(M))*. Quando ne ha ricevute tante quante il numero di keywords date in ingresso (ogni worker prende in gestione una keyword), va a recuperare (in), con un'altra invocazione sincrona sul bridge, anche le tuple publication con i risultati. Un'apposita hashmap tiene conto di tutte le pubblicazioni (key) e delle loro occorrenze (value). Infine, il master itera sulla hashmap e mostra i risultati sulla GUI.

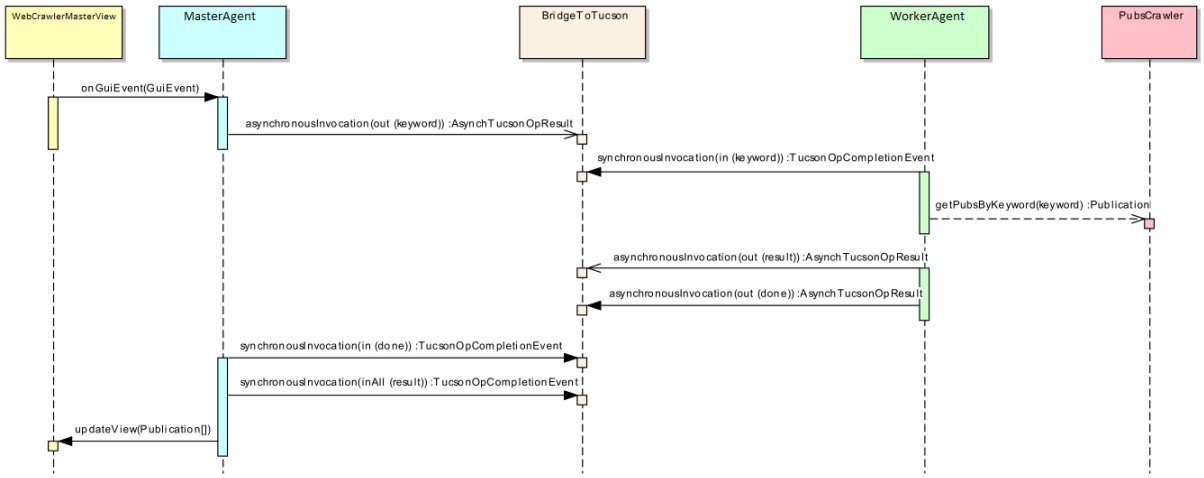


Fig. 12. Diagramma UML delle interazioni

6.2.1 Fault tolerance La gestione della fault tolerance prevede l'interazione tra gli agenti WatchdogAgent e PingAgent con tutti gli altri agenti del sistema. Ogni master e ogni worker, al loro avvio, mettono una tupla "hello" nel tuple centre per manifestare la loro presenza all'interno del sistema; il watchdog recupera tutte le tuple "hello" e salva le informazioni dei nodi e dei relativi agenti in una hashmap *nodeAgents*. In questo modo si ha una panoramica di quali e quanti agenti stanno operando nel sistema.
I vari master e worker risiedono su nodi sparsi in rete e per ogni nodo c'è un PingAgent; per verificare che, durante il funzionamento del sistema, un nodo non sia crashato avviene la seguente interazione: il watchdog mette nel tuple centre delle

tuple "ping" per ogni PingAgent attivo; quest'ultimo "risponde" recuperando la tupla "ping" e mettendo nel tuple centre una tupla "pong" che viene recuperata dal watchdog. Avviene poi un confronto tra gli agenti che hanno risposto al pong e quelli che sono nella hashmap *nodeAgents*.
Quelli che non hanno risposto al "pong" (e che quindi si suppone siano crashati) vengono posti in delle apposite hashmap, una per i master e i worker, ognuno col suo nome e l'indirizzo ip del nodo su cui risiedeva.
Per gestire il crash di agenti master, il watchdog rimuove dal tuple centre le tuple "waiting" lasciate dai master crashati.
Per gestire il crash di agenti worker, il watchdog rimuove dal tuple centre le tuple "working" lasciate dai worker crashati e invia ai master corrispondenti le tuple "done" al loro posto. Il master, vedendo arrivare la tupla "done" dal watchdog e non da un worker, mostrerà un alert.

6.3 Behaviour

Di seguito il comportamento degli agenti presenti nel sistema: MasterAgent, WorkerAgent, WatchdogAgent e PingAgent. Per rappresentare il ciclo di vita di un agente vengono utilizzate delle Finite State Machine.

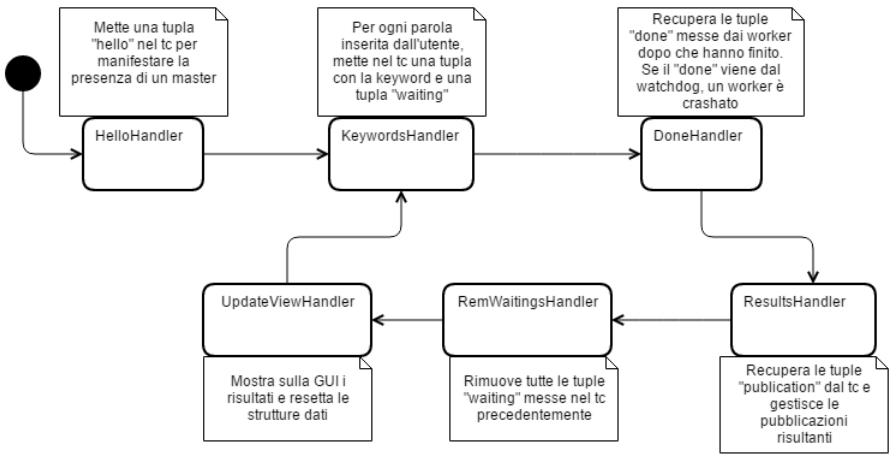


Fig. 13. MasterAgent

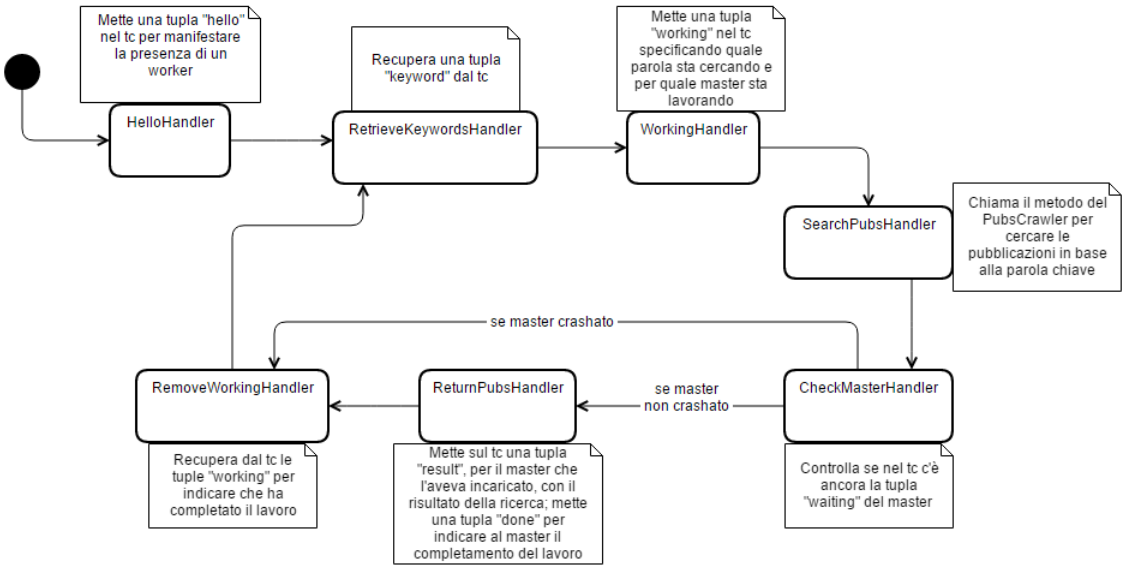


Fig. 14. WorkerAgent

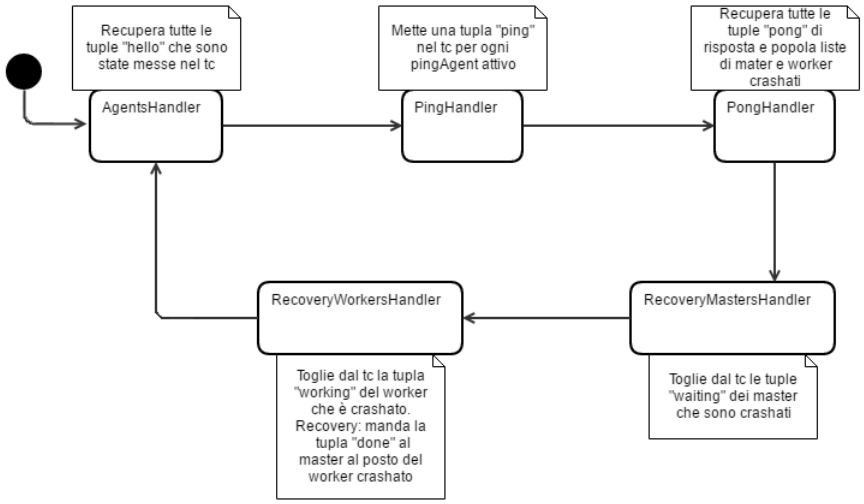


Fig. 15. WatchdogAgent

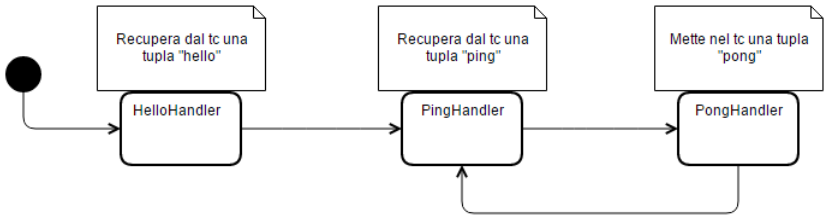


Fig. 16. PingAgent

7 Deployment

7.1 Configuration

Il sistema è stato testato in distribuito utilizzando pc e raspberry.
Un pc, considerato il punto di centralizzazione, contiene il MainContainer di JADE e il centro delle tuple di TuCSoN. Si suppone che tale pc non vada mai in crash. In questo container mandiamo in esecuzione la classe WebCrawlerPcMain.java che crea l'agente Watchdog, dedicato alla fault tolerance. Tale classe viene lanciata fornendo in ingresso due parametri: l'indirizzo IP del pc in cui viene lanciata e l'indirizzo IP del pc che ospita il centro delle tuple.

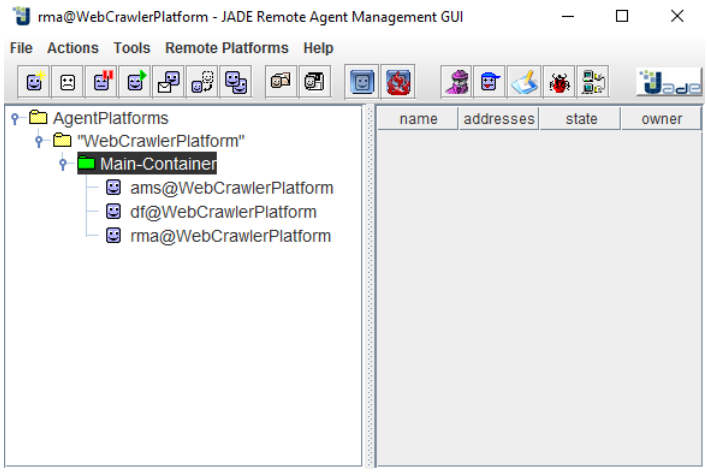


Fig. 17. Jade Main Container

In ogni raspberry mandiamo in esecuzione (servendoci di putty e TightVnc) la classe WebCrawlerRaspiMain.java che crea un ManagerAgent che è munito

di interfaccia e permette di aggiungere/rimuovere agenti Master e Worker sullo specifico raspberry. La classe WebCrawlerRaspiMain viene lanciata fornendo 3 parametri: un numero identificativo univoco all'interno del sistema, l'indirizzo IP del raspberry e l'indirizzo IP del pc che ospita il centro delle tuple.

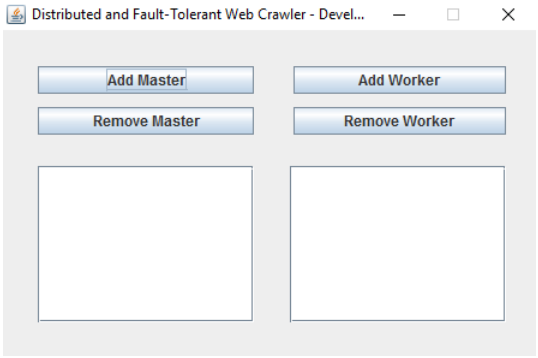


Fig. 18. ManagerAgent GUI

Fig. 19. MasterAgent GUI

Per ogni ManagerAgent in esecuzione, c'è anche un PingAgent. Il Watchdog interagisce con tale PingAgent per verificare che quello specifico nodo non sia crashato (si suppone che se crasha un agente master o worker in un raspberry, anche il PingAgent vada in crash perchè vengono mandati in esecuzione dallo stesso jar).
Per controllare che lo scambio di tuple sia corretto, usiamo il tool Inspector di Tucson; per avviarlo è sufficiente posizionarsi con il prompt dei comandi

all'interno della cartella libs del progetto e mandare in esecuzione il comando: java -cp tucson.jar;2p.jar alice.tucson.introspection.tools.InspectorGUI.

7.2 Starting the system

Dopo aver opportunamente configurato raspberry e pc e aver mandato in esecuzione i vari agenti (Watchdog sul pc principale, ManagerAgent su ogni raspberry, Master e Worker aggiunti dal ManagerAgent su ogni raspberry, PingAgent in esecuzione parallelamente al ManagerAgent) si può utilizzare il sistema. Inserendo delle parole chiave nella GUI del Master e facendo click sul pulsante **Search** inizierà la ricerca degli articoli contenenti le parole chiave. La ricerca è di tipo "AND": infatti, se vengono inserite le parole *fuzzy* e *logic*, gli articoli risultanti saranno solo quelli che contengono contemporaneamente entrambe le parole.
È importante specificare che ogni singola parola chiave viene presa in carico da un singolo worker, quindi per rendere la ricerca più veloce dovranno essere istanziati tanti Worker quante parole chiave. Nel caso in cui i Worker siano in numero inferiore rispetto alle parole chiave, il primo Worker che si libera prenderà in carico le parole successive.

7.3 Testing the system

A seguito di alcuni test del sistema su pc e raspberry abbiamo fatto alcune considerazioni sul tasso di completamento dei task dal punto di vista dell'ottenimento delle pubblicazioni-risultato.
Si suppone che il WatchdogAgent non vada in crash. Quando il sistema è impegnato in una ricerca di pubblicazioni:

- Se crasha un master, il tasso di completamento del task è molto basso perchè i worker non avranno più un corrispondente a cui restituire i risultati;
- Se crasha un worker, il tasso di completamento del task è mediamente alto: il task viene completato ma ha come risultato un messaggio d'errore e quindi le pubblicazioni trovate non vengono mostrate. Sarà però possibile dare inizio a una nuova ricerca con lo stesso master;
- Se crasha il ManagerAgent, il tasso di completamento del task è alto perchè il nodo continua a essere attivo nel sistema ma non c'è modo di gestire master e worker (aggiungerli o rimuoverli).
- Se crasha il PingAgent, il tasso di completamento del task dipende da 2 casi:
  - Se il PingAgent crasha perchè tutto il nodo è crashato il tasso di completamento del task è molto basso.
  - Se il nodo viene escluso dal sistema a causa di problemi di comunicazione PingAgent-Watchdog (falso-positivo), il tasso di completamento di eventuali task è alto se e solo se sia i master sia i worker si trovano tutti all'interno di quel container e non sparsi in rete.

Nella tabella seguente abbiamo stilato una classifica decrescente che mostra la gravità del crash di un agente ovvero quanto il crash di quell'agente influisce sul funzionamento del sistema.

Gravità	Agente
1	WatchdogAgent
2	PingAgent
3	ManagerAgent
4	MasterAgent
5	WorkerAgent

Dal punto di vista dei tempi di completamento possiamo dire che il tempo ottimale si ottiene se viene assegnata una parola chiave per ogni worker; se ci sono più worker che parole chiave, si verifica un inutile appesentimento del sistema perchè i raspberry dovranno gestire più agenti; se invece ci sono più parole chiave che worker, i tempi di completamento aumentano perchè si dovrà attendere la terminazione dei worker per prendere in carico le restanti parole.

8 Maintenance and future developments

Durante lo sviluppo del sistema abbiamo riscontrato alcuni problemi:

– Il parsing degli articoli di ApiCe: ogni pubblicazione ha un tag html detto "pre" (pre-formatted text), dove sono contenuti informazioni bibliografiche quali titolo, autore, parole chiave, anno, titolo del libro, ecc.

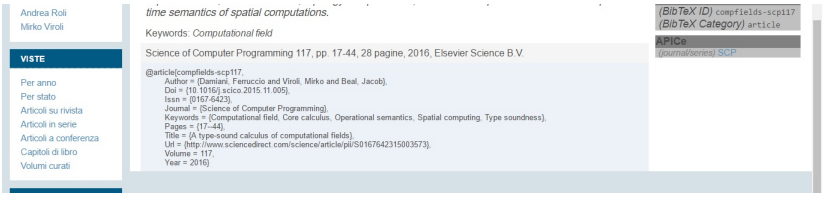


Fig. 20. Esempio di "pre" di una pubblicazione

Nonostante ci fosse un template generale comune a tutti gli articoli, le informazioni contenute in questo testo tuttavia erano formattate in modo diverso per ogni articolo, spesso con all'interno caratteri particolari come apici, parentesi graffe, barre oblique(derivanti da compilazioni in LaTex). Per il nostro scopo, cioè recuperare le informazioni salienti dagli articoli, queste difformità hanno reso oneroso il lavoro di creare regole di parsing del testo valide e riusabili per tutti gli articoli.
Per questo motivo riteniamo che tra gli sviluppi futuri ci possa essere quello di affinare il parser in modo da ottenere i risultati più giusti possibile dalle ricerche.

- La scarsa documentazione di TuCSoN e Jade: durante lo sviluppo ci siamo trovati spesso nella situazione in cui non sapevamo trovare una spiegazione a certi errori/eccezioni date dal codice quindi ci rivolgevamo a Internet con opportune query; tuttavia, i risultati delle ricerche erano spesso poco numerosi e aridi di contenuto, per cui la risoluzione dei problemi si dilungava notevolmente.
- Contestualmente al precedente problema, abbiamo riscontrato la mancanza di metodi per gestire le *pendingOperations*, ovvero le azioni prese in carico da un agente. In particolare, in caso di crash di un WorkerAgent, la sua pendingOp rimane nel tuple space e se arriva una tupla keyword rischia di essere "catturata" da quella pendingOp ma non c'è alcun agente esistente che possa gestirla. Sarebbe necessario un metodo che, in caso di crash di un agente, permettesse di rimuovere una pendingOp in base a un suo identificativo e senza che questa abbia restituito un risultato.
- In fase di test del sistema, abbiamo trovato difficoltà nella connessione al sito di ApiCe poichè il momento in cui effettuavamo i test è venuto a coincidere con l'operazione di migrazione del sito da parte degli amministratori. Questo causava enormi latenze nelle connessioni, spesso fallite, mettendoci nel dubbio su quale fosse il problema (se la rete o il nostro software)e costringendoci a rimandare i test. Ci è stato poi segnalato dall'amministratore che stavamo saturando il sito di richieste quindi ci è stato consigliato di introdurre un delay (ad esempio, 300ms) tra una richiesta e l'altra. Situazioni del genere sono spesso risolte dall'amministratore stesso di un sito, impedendo a chi si connette di effettuare richieste troppo frequentemente, situazione che causerebbe rallentamenti nell'accessibilità del sito.

Tra gli sviluppi futuri, prevediamo anche il fatto di poter applicare la funzionalità di WebCrawler ad altri siti e per altre tipologie di ricerche.
All'interno della classe PubsCrawler sarà necessario modificare l'url del sito a cui connettersi; dovrà inoltre essere modificata la parte in cui abbiamo fatto uso di Jsoup: infatti dovranno essere scelti gli opportuni tag html con cui fare *select* sul documento a cui ci si connette.
Il PreParser dovrà essere adattato in base alla struttura e sintassi delle informazioni da recuperare.
Importante da modificare è anche la struttura delle tuple TuCSoN che dovranno rispondere alle esigenze: questo deve essere fatto nelle classi WebCrawlerMaster e WebCrawlerWorker.

References

1. Lucidi del corso di Sistemi Distributi 2015-2016. *Middleware: astrazioni e tecnologie. Fondamenti.* http://apice.unibo.it/xwiki/bin/view/Courses/Sd1516Slides
2. Lucidi del laboratorio di Sistemi Distibuiti 2015-2016. *Jade. Tucson.* http://apice.unibo.it/xwiki/bin/view/Courses/Sd1516Lab
3. Sito ufficiale di Jade. http://jade.tilab.com
   Tutorial Starting JADE. http://jade.tilab.com/doc/tutorials/JADEAdmin/startJade.html
   Tutorial Creating a distributed platform. http://jade.tilab.com/doc/tutorials/JADEAdmin/JadeContainerTutorial.html
   Tutorial Running multiple JADE platforms. http://jade.tilab.com/doc/tutorials/JADEAdmin/JadePlatformTutorial.html
4. Wiki di Tucson4Jade. https://bitbucket.org/smariani/tucson4jade/wiki/Home
5. Wikipedia. *WebCrawler.* https://it.wikipedia.org/wiki/Crawler
6. Documentazione di Jade, nel materiale del laboratorio e online. http://jade.tilab.com/doc/api/overview-summary.html
7. Documentazione di TuCSoN, nel materiale del laboratorio.