Sistema multiagente replicato per la gestione di feed RSS

Reti di Calcolatori LS
Prof. A. Corradi
aa. 2003/2004

Alessandro Zanarini
0000170061

1 Abstract

Sistemi complessi che offrono una varietà di servizi a entità esterne si stanno sempre più muovendo verso un approccio ad agenti. Tale approccio, oltre a essere un vero e proprio cambiamento di paradigma, può essere visto come una possibile soluzione alle necessità di adattività all'ambiente nel quale i servizi sono immersi. L'interazione con l'ambiente, l'interazione tra gli agenti stessi e il relativo modello basato su centri di coordinazione assumono dunque un ruolo fondamentale per lo sviluppo di sistemi multiagente. La centralità del centro di coordinazione tuttavia può diventare un problema in sistemi nei quali vi siano requisiti di disponibilità del servizio.
Il progetto parte quindi dalla progettazione di un sistema multiagente e propone una possibile soluzione per la replicazione del centro di coordinazione.

2 Introduzione

Il modello ad attori prima e ad agenti poi, nasce dalla necessità di incapsulare all'interno di un'entità, non solo lo stato e il comportamento di una data entità (come avviene nella OO), ma anche il controllo autonomo delle proprie attività. Da questo punto di vista gli agenti possono essere visti come oggetti attivi, ovvero oggetti che sono totalmente autonomi nel gestire il flusso di controllo, al contrario degli oggetti passivi che subiscono il controllo da entità esterne.
L'astrazione d'agente va oltre e può essere definita come: "qualsiasi entità immersa in un ambiente, con cui ed in cui interagisce al fine di perseguire un determinato obiettivo, svolgendo una o più attività".
In sistemi mediamente complessi è naturale la presenza di più agenti per lo svolgimento di attività collettive e nasce quindi la necessità di interazione fra di essi.
In particolare nel progetto si è preso in considerazione il modello di interazione basato su spazio di tuple (Linda) e si è utilizzata un'implementazione java (Tucson) che aggiunge le leggi di coordinazione (sottoforma di reazioni) allo spazio di tuple, tali da modellare le cosidette *social rules.*
Il progetto prende in considerazione un caso di studio di un sistema multiagente per la gestione di feed RSS. Sono stati realizzati agenti per le comuni operazioni su feed RSS quali il recupero di canali RSS, la ricerca di blog basata su parole chiave e l'eventuale iscrizione a blog. Intorno a questo sistema è stato successivamente sviluppato un client che si appoggia al centro di tuple, per realizzare un'applicazione che fornisca all'utente le funzionalità di feed aggregator.
Infine è stata presa in considerazione la problematica relativa alla centralità del centro di coordinazione ed è stata proposta una soluzione per la replicazione del nodo tucson. Lo studio del problema (effettuato in maniera ortogonale e scorrelata al dominio dell'applicazione e ai particolari protocolli di interazione) insieme all'utilizzo della programmazione orientata agli aspetti, ha permesso di realizzare una soluzione generica e riutilizzabile in altri sistemi basati su tucson, senza necessità alcuna di modifiche al codice preesistente.
La relazione è organizzata come segue. La sezione 3 analizza il sistema di agenti e ne illustra l'architettura, la sezione 4 presenterà le scelte progettuali e le interazioni dei vari agenti con i centri di tuple. La replicazione verrà presentata nella sezione 5. La sezione 6 è dedicata alle conclusioni e agli sviluppi futuri.

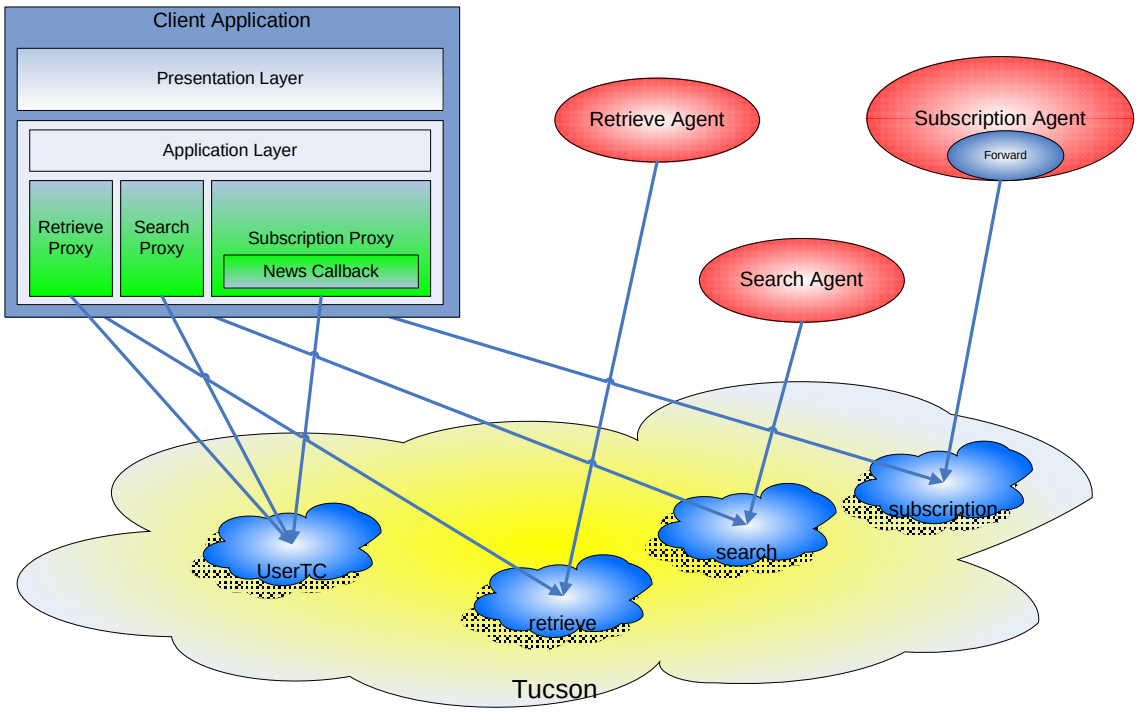
3 Architettura del sistema

Le comuni operazioni per la realizzazione di un'applicazione di feed aggregator sono:

1. Il recupero di feed RSS dalla rete dato un'url che li localizza
2. L'iscrizione (disiscrizione) a feed RSS per permettere un costante aggiornamento dell'applicazione in caso di pubblicazione di nuove notizie
3. La ricerca basata su parole chiave di feed RSS.

E' stato scelto di utilizzare un agente diverso per ogni singola operazione. Tale scelta ha come fine la semplificazione dell'architettura del singolo agente e inoltre è vista nell'ottica di una possibile integrazione di questa serie di servizi in un sistema multiagente più vasto e con più servizi già attivi.

L'architettura può essere schematizzata nel modo seguente:



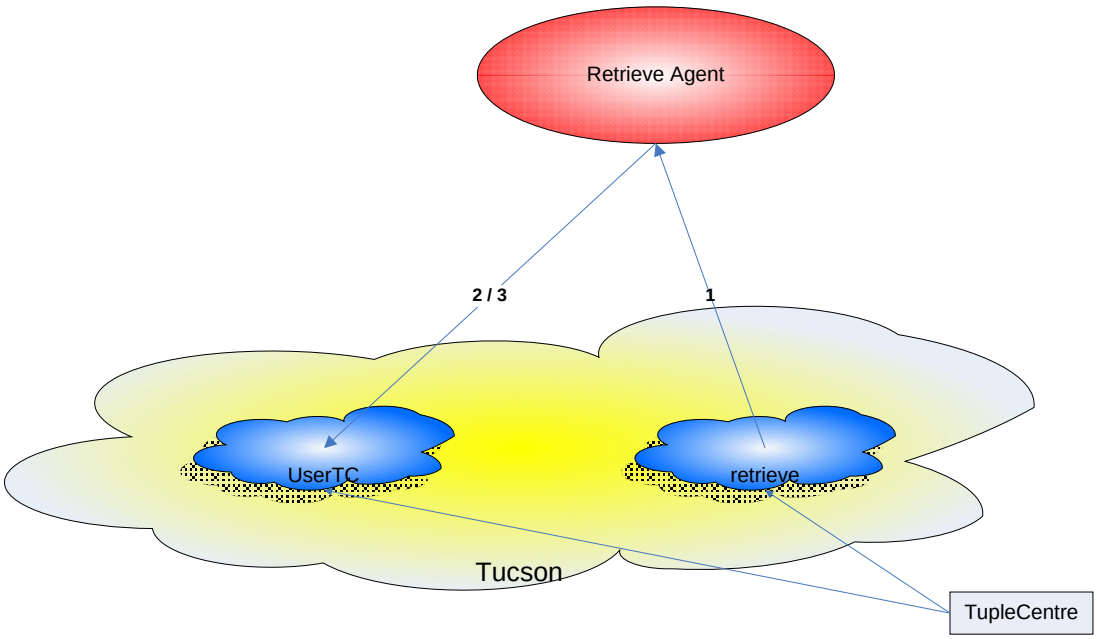
Nell'applicazione utente sono presenti opportuni componenti proxy che si occupano dell'interazione con i centri di tuple.

Lato agente è stata sfruttata una libreria (Informa) per la gestione dei feed RSS. Per le ricerche invece è stato scelto di interfacciarsi al motore di ricerca per feed RSS www.feedster.com e realizzare un componente wrapper all'uopo per l'estrazione dei dati da esso.

4 Interazioni con il centro di coordinazione

4.1 Retrieve Agent

Si occupa di recuperare un feed RSS da Internet a partire da un Url che lo localizza e presenta le notizie in esso contenute all'utente che ha effettuato la ricerca.



Il Retrieve Agent riceve le richieste nel centro di tuple "retrieve" e risponde nel tuple centre associato all'utente che ha effettuato la richiesta. In particolare:

1. *retrieve ? in(retrieve(user(Name), Url))*
   - retrieve - il tuple centre dal quale l'agente riceve le richieste
   - Name - il nome dell'utente che effettua la richiesta
   - Url - url del blog da recuperare

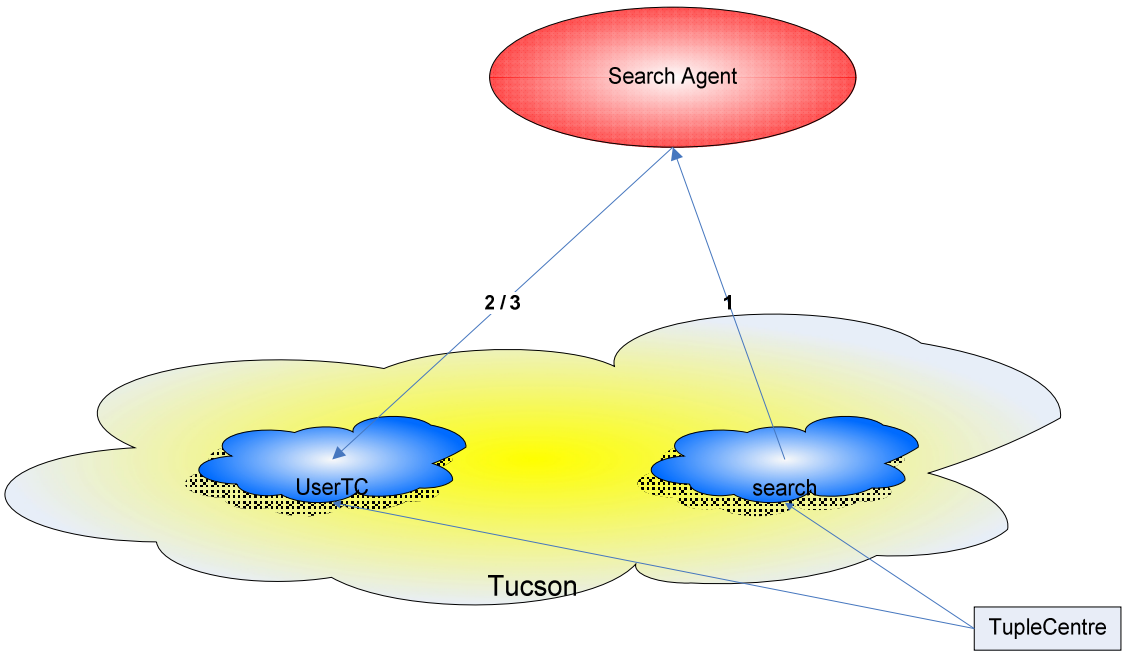
2. *Name ? out(items(ChannelUrl, ChannelTitle, NumItems))*
   - Name – il tuple centre associato all'utente che ha effettuato la richiesta
   - ChannelUrl - url del blog
   - ChannelTitle - titolo del blog
   - NumItems - numero di notizie appartenenti al blog che corrisponde al numero di successive tuple che seguono il pattern (3)

3. *Name ? out(item(ChannelUrl, Title, Date, Description, Creator, Link))*
   - ChannelUrl - url del blog al quale la singola notizia è associato
   - Title - titolo della singola notizia
   - Date - data della singola notizia
   - Description - descrizione della singola notizia
   - Creator - l'autore della singola notizia
   - Link - link associato alla singola notizia

In caso di errore interno dell'agente le tuple 2 / 3 non verranno mandate ma l'agente risponderà con una tupla d'errore.

4.2 Search Agent

Si occupa di effettuare le ricerche di feed RSS a partire da parole chiave. Presenta successivamente i risultati della ricerca all'utente che ha effettuato la richiesta.



Il Search Agent riceve le richieste nel centro di tuple "search", effettua la ricerca collegandosi al motore di ricerca www.feedster.com e risponde nel centro di tuple associato all'utente. In particolare:

1. *search ? in(search(user(Name),Keyword,opt([X])))*
   - search - il tuple centre dal quale l'agente riceve le richieste
   - Name - il nome dell'utente che effettua la richiesta
   - Keyword - le parole chiave da ricercare
   - opt(X) - le eventuali opzioni di ricerca (lingua, moreResults, ecc...)

2. *Name ? out (results(Keyword,Num))*
   - Name – il tuple centre associato all'utente che ha effettuato la richiesta
   - Keyword - la parola chiave ricercata
   - Num - il numero di elementi trovati che corrisponde al numero di successive tuple che seguono il pattern (3)

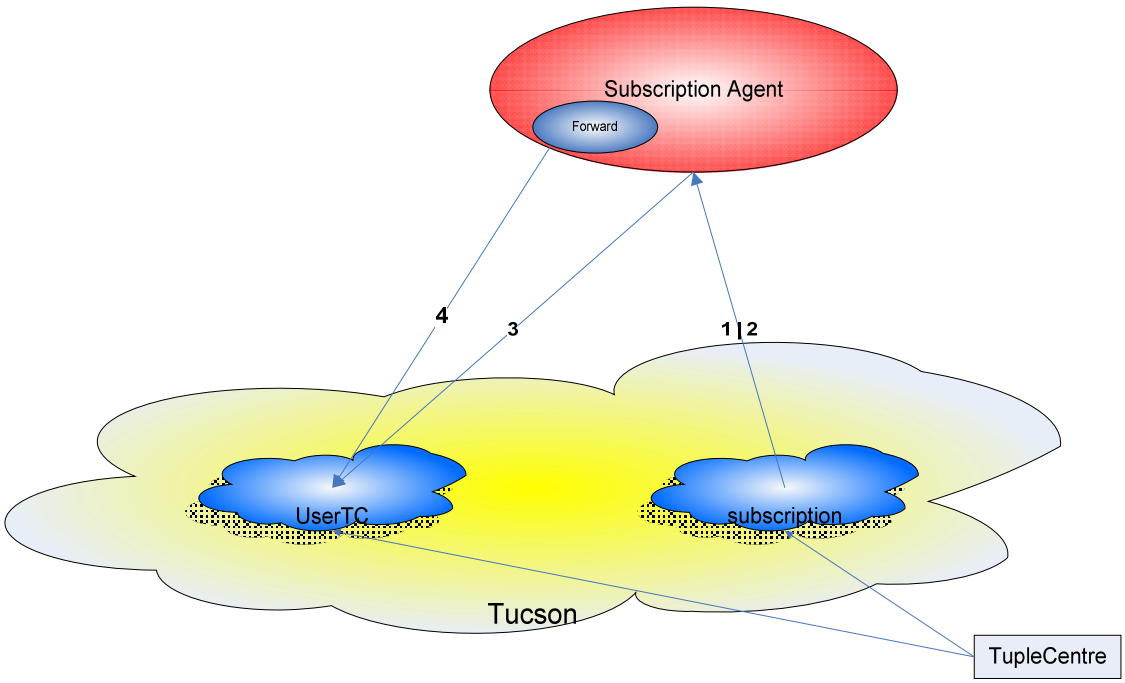
3. *Name ? out (result(Keyword,Item,Link,Description))*
   - Keyword - la parola chiave ricercata
   - Num - il numero di elementi trovati
   - Item - il titolo riassuntivo del contenuto del blog
   - Link - url del blog

- Description - descrizione (estratto) del contenuto del blog

In caso di errore nell'esecuzione della richiesta le tuple 2 / 3 non verranno mandate ma l'agente risponderà con una tupla di errore.

4.3 Subscription Agent

Si occupa di gestire le iscrizioni a feed RSS con un meccanismo del tipo publish-subscribe. Raramente i feed RSS offrono servizi di tipo publish-subscribe perciò è stato scelto di mantenere un costante aggiornamento tramite un meccanismo a polling. Il componente che effettua il polling è già fornito nella libreria Informa. Infine è presente un sottocomponente che si occupa di inoltrare all'utente opportuno le nuove notizie arrivate.



Il Subscription Agent riceve le richieste dal centro di tuple "subscription" e risponde nel centro di tuple associato all'utente che ha effettuato la richiesta. In particolare le interazioni procedono nel seguente modo:

1. *subscription ? in(subscribe(user(Name), Url))*
   - subscription - il tuple centre dal quale l'agente riceve le richieste
   - Name - il nome dell'utente che effettua la richiesta
   - Url - url del blog a cui l'utente vuole iscriversi

2. *subscription ? in(unsubscribe(user(Name), Url))*
   - subscription - il tuple centre dal quale l'agente riceve le richieste
   - Name - il nome dell'utente che effettua la richiesta
   - Url - url del blog a cui l'utente vuole disiscriversi

3. *Name ? out(subscribed(Url, ChannelTitle))*
    - Name – il tuple centre associato all'utente che ha effettuato la richiesta
    - Url - url del blog a cui l'utente si è iscritto
    - ChannelTitle - titolo del blog a cui l'utente si è iscritto

4. *Name ? out(publishedItem(ChannelUrl, Title, Date, Description, Creator, Link))*
    - ChannelUrl - url del blog a cui è associata la notizia
    - Title - titolo della singola notizia
    - Date - data della singola notizia
    - Description - descrizione della singola notizia
    - Creator - l'autore della singola notizia
    - Link - link associato alla singola notizia

5 Replicazione

5.1 Analisi

Il problema dei sistemi basati su centri coordinazione risiede nell'eccessiva centralità del media di coordinazione stesso. Questo in caso di guasto può portare a disservizi e problemi di disponibilità. L'analisi trasversale, indipendente dal dominio e completamente slegata dai particolari protocolli di interazione, unita all'utilizzo della programmazione orientata agli aspetti ha portato una soluzione che può essere facilmente riutilizzata in altri sistemi basati su tucson.

La replicazione, nel tentativo di mantenere il sistema semplice, parte da queste ipotesi semplificative:

- Guasto singolo
- Conoscenza a priori dei nodi che parteciperanno al cluster

Non sono state fatte ipotesi sull'organizzazione gerarchica dei vari nodi (Master – Slave, alberi, ecc...), ma si è considerato il gruppo di nodi come un insieme di pari.

Si possono quindi individuare due problematiche essenziali che verranno trattate in maniera indipendente:

1. L'aggiornamento dei vari nodi che concorrono alla replicazione
    a. Mantenimento della consistenza all'interno del cluster
    b. Recovery in caso di guasto
2. L'accesso al cluster di nodi da parte degli agenti

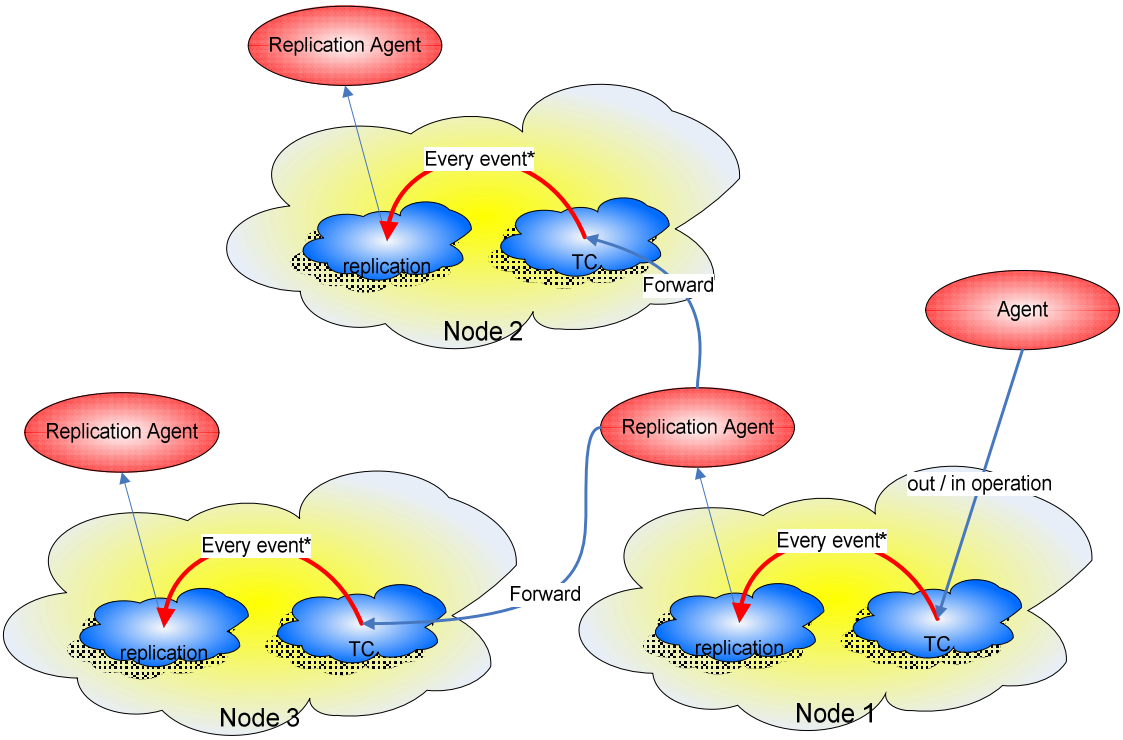
Questi aspetti verranno trattati nei successivi paragrafi

5.2 Protocollo di replicazione

Dato un nodo, le uniche operazioni che cambiano l'insieme delle tuple in esso contenute sono: *in[p]* e *out*.

L'idea di base consiste nel catturare tutti gli eventi che modificano lo stato del nodo e riproporli nei nodi che partecipano al cluster.
Sono state quindi definite delle reazioni in ogni singolo centro di tuple in modo da notificare ogni cambiamento al centro di tuple "*replication*". Un agente si occupa successivamente di inoltrare i cambiamenti a tutti i nodi del cluster.

Questo è lo schema di funzionamento di base:



* Every event that is caused by an agent that is not the replication Agent

Le reazioni definite su tutti i centri di tuple (fuorché "*replication*" stesso) sono le seguenti:

% Replication specification

1. *reaction(out(X),*
   (current_agent(A),
   \\+(A is replicationManagementAgent),
   out_tc(replication,tuple(TC,out,X)))).

2. *reaction(in(X),*
   (post,
   current_agent(A),
   \\+(A is replicationManagementAgent),
   out_tc(replication,tuple(TC,in,X)))).

3. *reaction(inp(X),*
   (post,
   success,
   current_agent(A),
   \\+(A is replicationManagementAgent),
   out_tc(replication,tuple(TC,in,X)))).

Tutte le reazioni unificano se e solo se l'agente che effettua l'operazione non è l'agente che si occupa della replicazione. Questo per evitare che l'inoltro di tuple da parte del Replication Agent scateni successive reazioni a catena per replicare nuovamente la tupla.

In particolare:

1. all'occorrenza di una operazione di *out* viene inoltrata al centro di tuple "*replication*" una tupla nella forma di:
   tuple(TC,out,X)

2. all'occorrenza di una operazione di *in* viene inoltrata al centro di tuple "replication" una tupla nella forma di:
   tuple(TC,in,X)

3. all'occorrenza di una operazione di *inp* andata a buon fine, viene inoltrata al centro di tuple "replication" una tupla nella forma di:
   tuple(TC,in,X)

dove:

- TC – il centro di tuple originale dove la tupla è stata depositata
- out | in – l'operazione che l'agente replication dovrà effettuare
- X – la tupla da inoltrare

Non vi è esplicito riferimento ad un nodo master o nodi slave in quanto qualunque nodo può modificare il suo stato e inoltrarlo agli altri nodi del cluster.

5.3 Procedura di recupero dal guasto

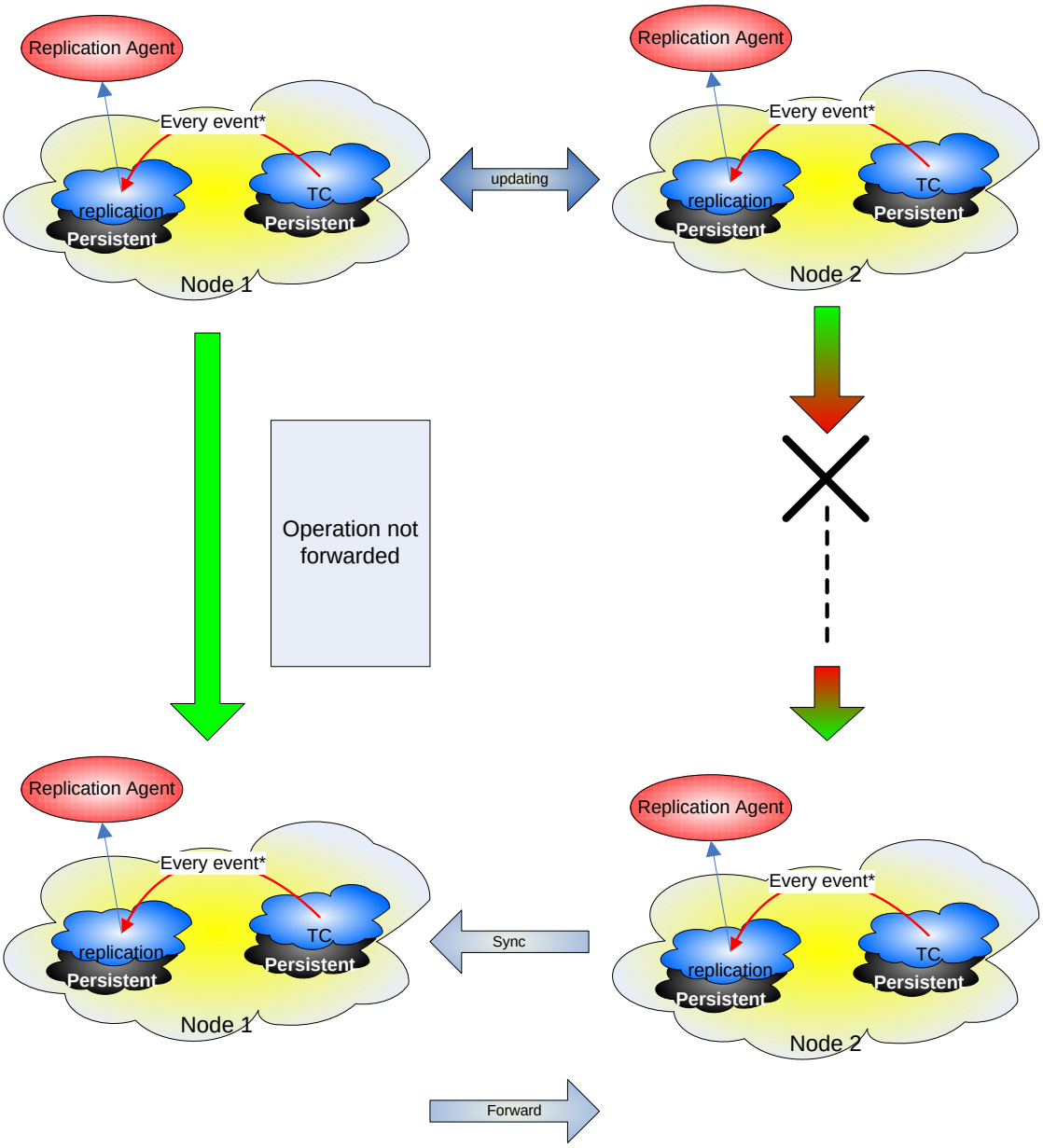
Nella procedura proposta per il recupero da guasti si possono individuare due sottoprocedure fondamentali:

1. Il salvataggio dello stato di un nodo fino al momento del guasto
2. L'aggiornamento dello stato per riflettere i cambiamenti avvenuti durante il downtime

Per poter mantenere lo stato salvato, ogni centro di tuple sfrutta la persistenza fornita da tucson, ovvero ogni singola tupla depositata o prelevata da un nodo viene registrata su disco. Al riavvio del nodo, in seguito ad un guasto, il centro di tuple ricarica automaticamente lo stato salvato.

Nel periodo di downtime il cluster può evidentemente avere subito modifiche ed è necessario quindi inoltrare le modifiche al nodo tornato attivo.
La procedura di aggiornamento segue questo schema:

1. Ogni nodo tiene traccia degli eventi che non è riuscito ad inoltrare al nodo caduto.
2. Alla riattivazione, il nodo che ha subito il guasto, manda una tupla di sincronizzazione al centro di tuple "*replication*" di tutti i nodi del cluster. In particolare la tupla segue questo pattern:
   tuple(LocalName,"sync","")
   dove:
   LocalName – è il nome del nodo che ha subito il guasto
3. I nodi presenti nel cluster inoltrano tutti gli eventi registrati.

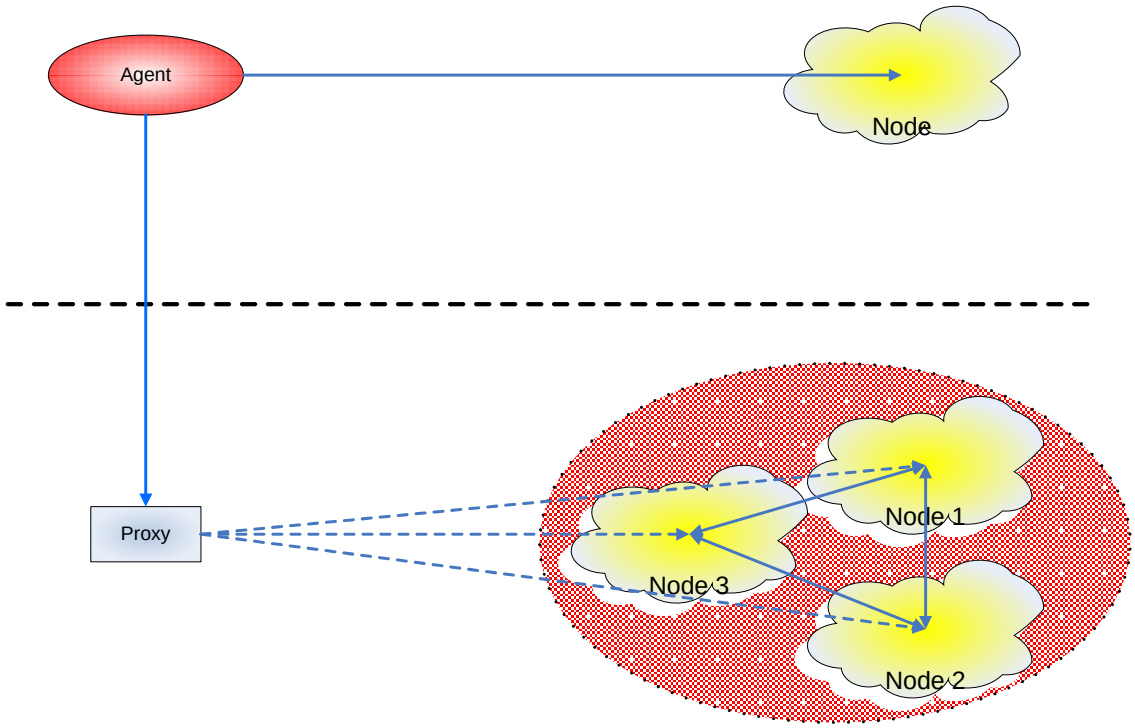


5.4 Accesso al cluster da parte degli agenti

Si ricorda che la soluzione proposta si basa su due forti assunzioni:

- Conoscenza a priori dei centri di tuple che partecipano alla replicazione
- Nessuna presenza di nodi master

L'accesso al cluster di nodi da parte degli agenti avviene in maniera totalmente trasparente, a livello di modello, ovvero l'agente vede, accede e percepisce sempre un solo nodo. Tuttavia a livello implementativo uno strato software provvede a smistare le richieste al primo nodo che trova disponibile.



La soluzione proposta sfrutta la programmazione orientata agli aspetti per catturare ogni chiamata a metodi che accedono ai centri di coordinazione. Tali chiamate vengono sostituite con un blocco di codice che provvede a ciclare sulla lista di nodi e su di essi inoltra la chiamata fermandosi al primo nodo che trova disponibile.
Da notare i relativi pointcut:

```
pointcut out(TupleCentreId TCId, LogicTuple tuple):
        call(void TucsonContext.out(TupleCentreId, LogicTuple))
        && args(TCId, tuple)
        && !within(rss.replication.*);

pointcut in(TupleCentreId TCId, LogicTuple tuple):
        (call(LogicTuple TucsonContext.in*(TupleCentreId, LogicTuple))
        && args(TCId, tuple)
        && !within(rss.replication.*));
```

Il primo cattura tutte le chiamate a out mentre, il secondo cattura tutte le chiamate a in (sia bloccante che non bloccante). Da evidenziare inoltre che non vengono catturate le chiamate che avvengono dentro il codice del Replication Agent: questo difatti deve

avere una visione granulare del cluster e poter fare delle chiamate puntuali ai singoli nodi.

La soluzione permette facilmente un'integrazione con progetti preesistenti: una ricompilazione del codice con il supporto ad AspectJ (implementazione di AOP) permette immediatamente di sfruttare la replicazione.

6 Conclusioni e sviluppi futuri

Il progetto ha proposto una soluzione facilmente riutilizzabile per la replicazione dei centri di tuple. Tra le caratteristiche più significative è da evidenziare l'assenza totale di gerarchie all'interno del cluster di nodi, questo permette di accedere in maniera totalmente trasparente ad ogni singolo nodo con una facile opportunità di effettuare *load balancing*.
La replicazione effettua un'aggiornamento lazy dello stato per poter migliorare il tempo di risposta al cliente. Tuttavia questo può causare inconsistenze: l'assenza di un meccanismo, quale l'*atomic broadcast*, per l'aggiornamento delle copie può portare, in casi limite, ad un disallineamento delle copie.
Gli sviluppi devono quindi andare verso la realizzazione di un meccanismo di atomic broadcast; con tale supporto può essere inoltre presa in considerazione una facile estensione del protocollo per la gestione di guasti multipli.

Riferimenti

- McKinley et al., "Composing Adaptive Software"
- A. Omicini e F. Zambonelli, "Tuples Centres for the Coordination of Internet Agents"
- A. Omicini, E. Denti, "Formal ReSpecT"
- A. Ricci, A. Omicini e M. Viroli, "Extending ReSpecT for Multiple Coordination Flows"
- A. Ricci, "Tucson guide" (version 1.4)
- Ramnivas Laddad, "AspectJ In Action", cap.3 pp. 64-99