BitPantarei
Incarnazione del modello $\mathcal{M}o\mathcal{K}$

Matteo Francia «matteo.francia2@studio.unibo.it»
Giovanni Ciatto «giovanni.ciatto@studio.unibo.it»

Sommario

L'idea di questo progetto è la realizzazione di un sistema di file sharing che incarni anche solo parzialmente il modello $\mathcal{M}o\mathcal{K}$ sfruttando il middleware TuCSoN ed il linguaggio ReSpecT, con la finalità di meglio comprendere come coordinare il flusso dell'informazione del primo, la robustezza ed effettiva praticità di utilizzo del secondo e, soprattutto, sperimentare l'impiego combinato di design pattern che traggano ispirazione dalla biologia.

Indice

1 Visione

Un sistema BitPantarei a regime sfrutta una rete di nodi logici detti *compartimenti*, almeno uno per ogni utilizzatore, ognuno capace di memorizzare una quantità indefinita di dati. A quest'ultimi, che dal punto di vista utente corrispondono a file e, in generale, anche a parte di essi, sono associati dei tag, assegnatigli da chi li ha inseriti nel sistema e/o dal sistema stesso mediante tecniche di aggregazione, e vengono riverberati di nodo in nodo con un meccanismo che tenga conto dei tag "desiderabili e non" da ogni compartimento. Tale riverbero *non* implica una migrazione dei dati bensì un loro moltiplicarsi da un compartimento all'altro o all'interno dello stesso, il che, particolarmente in questo secondo caso, evidenzia la necessità del concetto di *concentrazione* di un dato. Per evitare una saturazione del sistema è previsto un meccanismo di *evaporazione* che spontaneamente diminuisca la concentrazione dei dati. La particolarità del sistema è che ognuna delle caratteristiche descritte agisce in *probabilità*: è quindi possibile che l'informazione fluisca in più direzioni, non prevedibili a priori (da cui il nome del sistema). Scopo dell'esperimento è anche capire se sia possibile per un sistema siffatto raggiungere uno stato di "equilibrio dinamico" in cui le informazioni siano globalmente contenute all'interno del sistema, seppur in maniera volutamente frammentata, facilmente reperibili e difficilmente deteriorabili. L'utente ha la possibilità di condividere file pubblicamente inserendoli nel sistema e di specificare tag di interesse affinché il sistema possa reperire, ovvero attrarre nel compartimento dell'utente, tutte le informazioni correlate: i tag inseriti in un compartimento sono dunque quelli che in un sistema ACO si chiamerebbe "attrattiva a posteriori". Un meccanismo di "attrattiva a priori" potrebbe essere la quantità di dati inseriti dall'utente, la frequenza degli inserimenti o in generale un qualunque parametro che stimoli la partecipazione collettiva.

2 Meccanismi di base

Nel modellare i concetti impiegati nel progetto si cercherà di utilizzare un formalismo il più vicino possibile a quello matematico. Ciò avverrà senza la pretesa di completezza o coerenza ma comunque col preciso intento di definire in maniera non fraintendibile le *specifiche* che hanno guidato poi le implementazioni.

2.1 Glossario

Nella seguente sezione si analizzano i concetti necessari ed i meccanismi di base per il sistema e se ne elencano le rispettive possibilità implementative per mezzo di TuCSoN, non stupisce dunque che la nomenclatura di riferimento sia coincidente con quella di tale tecnologia. Nel seguito tuttavia, causa l'ambizione di generalità, si utilizzeranno anche termini che tentano di avvicinare lettori e autori al modello $\mathcal{M}o\mathcal{K}$, onde per cui è riportato un glossario informale che coadiuvi la comprensione:

Atomo è un vettore [qui inteso come portatore] di informazione. Per i TuCSoN-user si tratta di una *tupla* nella forma `tuple(A)`, dove `A` è informazione potenzialmente utile agli utenti. Ciò è importante per distingere gli atomi dalle tuple strutturali che contengono invece dati necessari al corretto funzionamento del sistema. Con abuso della parola *«tupla»*, l'atomo è spesso chiamato così in quanto unica tupla a trasportare effettivamente payload.

Molecola si intende una collezione di atomi. Per semplicità e dato che l'aggregazione non è il focus del progetto, nel seguito si considerano solo molecole composte da un solo atomo, quindi i due termini possono essere considerati sinonimi seppur a scapito della generalità e le rappresentazioni sono le stesse.

Specie [di molecole in un compartimento] continuando la metafora chimica, si tratta di un insieme di molecole uguali tra loro cui è associata, in un dato istante, una concentrazione ossia un numero che rappresenta la quantità di molecole dello stesso tipo presenti in un compartimento in quel momento. Implementativamente si tratta di una tupla nella forma `extTuple(C,tuple(A))`. È sinonimo l'espressione *«tupla estesa»*.

Compartimento è un insieme di specie di molecole comunicante con altri compartimenti. Dal punto di vista logico è una coppia insieme di specie-insieme di vicini. Implementativamente è un centro di tuple (nel seguito t.c.) contente tuple informative e strutturali, con associata una intrinseca nozione di vicinato: ogni compartimento ha un numero arbitrario e dinamico di vicini ad esso collegati tramite uno scambio reciproco di riferimenti. Analogamente al t.c., un compartimento è identificato da una terna nome-indirizzo-porta nella forma `Name@Address:Port`. Di nuovo con abuso di notazione, sono sinonimi le espressioni *«centro di tuple»* e *«nodo logico»*.

Servizio è un nodo TuCSoN identificato dalla coppia indirizzo-porta nella forma `Address:Port`

Inserimento [di molecole in un compartimento] è l'operazione attraverso cui si aumenta la concentrazione di una specie all'interno di un compartimento. Implemetativamente corrisponde alla `out` di una tupla o alla `out_all` di una lista nel t.c. corrispondente al compartimento.

Rimozione [di molecole in un compartimento] è l'operazione complementare all'inserimento. Implementativamente corrisponde ad una `in` oppure una `in_all`.

2.2 Concentrazione

La trattazione sulla concentrazione è la prima in quanto le scelte relative alla sua semantica ed implementazione condizionano pesantemente tutti gli altri ragionamenti.

La concentrazione di un'informazione in un determinato contesto è definibile informalmente come una misura della rilevanza dell'informazione stessa. Nella fattispecie ci si è focalizzati su concetrazioni a valori interi la cui modellazione e implementazione in TuCSoN viene molto naturale. Segue una definizione più rigorosa del concetto di concentrazione.

Data una specie s in un compartimento C, siano:

$ins_C(s,\, t)$ il numero di volte in cui una molecola di specie s è stata inserita in C dalla sua creazione al tempo t;

$rem_C(s,\, t)$ il numero di rimozioni di molecole di specie s da C al tempo t;

$con_C(s,\, t) \stackrel{def}{=} ins_C(s,t) - rem_C(s,t)$ la concentrazione in C della specie s al tempo t, invariante di base comune a tutte le casistiche che seguono;

$count_C(s,\, t)$ il numero di molecole di specie s effettivamente presenti in C al tempo t. Seppur concettualmente irrilevante, tale parametro, neanche a dirlo dipendente dall'implementazione, è assai importante per la complessità spaziale e computazionale.

2.2.1 Concentrazione come numero di tuple uguali

Il t.c. è un multiinsieme di tuple, non stupisce quindi che possa contenere più volte la stessa tupla: la concentrazione coincide col numero di volte in cui essa è presente nel t.c.. Questa è la modellazione più semplice ed intuitiva del concetto di concentrazione la cui implementazione è quella adottata da TuCSoN: essa porta l'ulteriore vantaggio di poter sfruttare le primitive stocastiche ma lo svantaggio di occupare una quantità di spazio che cresce linearmente con la dimensione della tupla e la sua concentrazione. Altro svantaggio è il seguente: a livello concettuale, considere il t.c. come un multi-set implica una complicazione nel far emergere il concetto di specie, cosa che si traduce, a livello impletetativo, in una maggiore complessità [computazionale] quando serve manipolare un intero insieme di tuple uguali.

In questa variante della concentrazione è definito un solo invariante:

$$count_C(s,\, t) = con_C(s,\, t)$$

che sancisce che in ogni momento la concentrazione di un'informazione è pari al numero di volte in cui è presente nel t.c..

2.2.2 Concentrazione come contatore

In questo caso il centro di tuple si può pensare come un insieme propriamente detto [e non più un multiinsieme] in quanto esiste una sola "copia" per ogni tupla. Il sistema sfrutta delle coppie tuplacontatore, che nel seguito chiameremo tuple estese, e opportune reazioni che rendano trasparente questo dettaglio ai coordinabili. Ovviamente sono possibili più implementazioni che riconducano la concentrazione ad un contatore; nel seguito ne è proposta una concepita per

richiedere poche reazioni possibilmente molto concise: per ogni specie s concettualmente presente nel compartimento, il t.c. che lo implementa contiene una tupla estesa nella forma `extTuple(C,tuple(A))` e vale il seguente invariante:

$$count_C(s,\, t) = \begin{cases} 1 & \textit{se } conc_C(s,t) > 0 \\ 0 & \textit{altrimenti} \end{cases}$$

che sostanzialmente fa sì che, se la concentrazione di una tupla è positiva, allora ne esiste un'unica copia all'interno del t.c., altrimenti non ne esiste nessuna: ciò permette di non dover scrivere reazioni per le primitive `rd`, `rdp`, `no` e `nop`.

In figura 1 nella pagina seguente è rappresentata l'implementazione delle reazioni che garantiscono il rispetto degli invarianti. Da notare che non è prevista la rimozione della tupla quando la concentrazione raggiunge lo 0 ed è ammesso che le tuple abbiano concentrazione negativa per far sì che questa implementazione della concentrazione sia trasparente agli agenti [leggasi: per far funzionare anche la `in` e varianti). Da notare inoltre che non sono state trattati i casi delle primitive bulk e stocastiche.

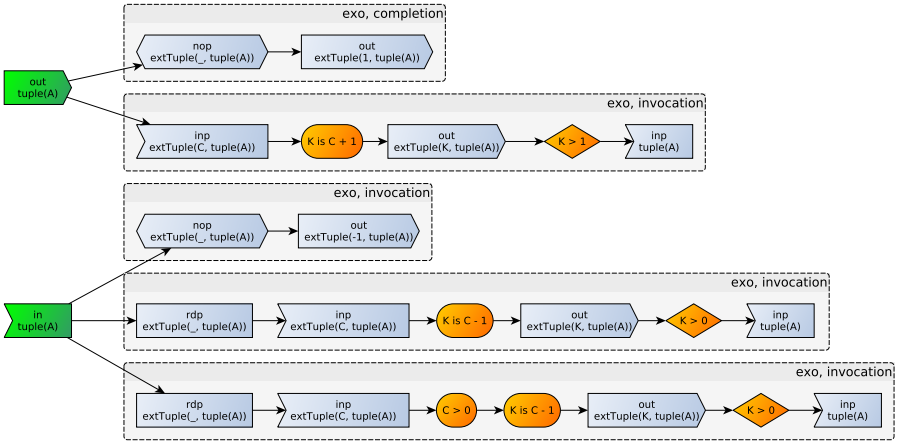
2.3 Attività periodica

Spesso nasce l'esigenza che il t.c. esegua un'azione più volte ad intervalli di tempo regolari in maniera trasparente agli agenti. Questa sezione descrive le reazioni che permettono di ottenere tale risultato. Le motivazioni dell'importanza di una definizione così articolata saranno chiare più avanti. Partiamo con un rapido accenno al Timed-ReSpecT: fissato un numero naturale P di quanti di tempo [nel seguito q.d.t., implementativamente parliamo di millisecondi] è possibile registrare una reazione all'evento `time(P)` che si verifica quando il tempo locale (ovvero il numero di q.d.t. dall'avvio del nodo TuCSoN) raggiunge P; è possibile anche, in ogni reazione, conoscere il valore del q.d.t. locale al momento dell'avvio della reazione stessa (quindi quello più vicino al q.d.t. attuale) tramite il predicato `current_time(T)`. La natura di ReSpecT impone un approccio asincrono: non è pensabile che la VM si blocchi in attesa che trascorra un periodo; pertanto è necessario ordinare con perizia gli eventi [i.e. le reazioni] affinché creino in ciclo potenzialmente infinito descritto informalmente nel seguito:

1. registrare l'esecuzione dell'operazione nell'istante in cui il tempo raggiungerà $T_{n+1} = T_n + P$, dove T_n è il q.d.t. corrente e P è il periodo;
2. quanto il tempo locale raggiunge T_{n+1} eseguire l'operazione;
3. tornare al punto 1.

In pratica ogni «iterazione del ciclo» deve registrare l'esecuzione futura della prossima iterazione con un ritardo pari ad un periodo.

Figura 1: Rappresentazione delle reazioni ReSpecT che implementano la concentrazione come contatore.



Legenda:

- Concavità a sinistra → consumo di tuple (`in` e varianti)
- Convessità a destra → inserimento di tuple (`out` e varianti)
- Rettangolo → lettura (`rd` e varianti)
- Convessità su ambo i lati → assenza (`no` e varianti)
- In verde → operazioni che generano reazioni
- Rettangolo con bordi arrotondati → predicato Prolog il cui fallimento provoca il fallimento della reazione
- Rombo → predicato Prolog il cui fallimento provoca semplicemente il non proseguimento del flusso d'esecuzione
- Contenitore → reazione
- Titolo del contenitore → guardia/e

2.3.1 Modellazione matematica

Si parte dalle seguenti assunzioni:

- è definita la struttura di una tupla detta *«di avvio»*, il cui inserimento innesca l'inizio di una sequenza di esecuzioni ad intervalli regolari di una medesima azione;
- è definita la struttura di una tupla detta *«di stop»*, il cui inserimento determina la fine della suddetta sequenza, ovvero, la cui assenza abilità il prosieguo della stessa;
- è definita la struttura di una dupla detta «d'azione», il cui inserimento rappresenta l'azione da eseguirsi periodicamente.

In questa ipotesi, siano:

p una costante che rappresenta il periodo;

t_0 l'istante in cui è stata inserita l'ultima tupla di avvio;

$start(t) : \mathbb{R} \to \mathbb{N}$ il numero di tuple di avvio inserite nel sistema dal tempo 0 al tempo t;
se si immaginasse di conoscere gli istanti di inserimento delle tuple di avvio, questa funzione sarebbe una sommatoria di scalini di di Heaviside centrati nei vari istanti e la sua derivata, $start'(t)$, una sommatoria di delta di Dirac centrate nei medesimi istanti (vale banalmente $start'(t_0) \neq 0$);

$stop(t) : \mathbb{R} \to \mathbb{N}$ il numero di tuple di stop inserite nel sistema al tempo t (valgono considerazioni analoghe);

$cstart(t) : \mathbb{R} \to \{0, 1\}$ il numero di tuple di avvio presenti nel t.c. all'istante t che per definizione può essere solo 0 o 1;

$cstop(t) : \mathbb{R} \to \{0, 1\}$ il numero di tuple di stop presenti nel t.c. all'istante t che per definizione può essere solo 0 o 1;

$enable(t) : \mathbb{R} \to \{0, 1\}$ funzione di abilitazione che indica, se positiva, che l'esecuzione dell'azione periodica può avvenire;

$action_{t_0}(t) : \mathbb{R} \to \mathbb{N}$ il numero di volte in cui la tupla d'azione è stata inserita nel t.c. dall'istante t_0 all'istante t.

Oltre a quelli banali, che regolano il numero di tuple si avvio e stop simultaneamente presenti nel t.c., ci sono due invarianti. Uno regola l'abilitazione:

$$\forall\, t_1 > t_0 \;:\; stop'(t_1) = 0$$
$$enable(t) = \begin{cases} 1 & se\ t \in [t_0,\, t_1] \\ 0 & altrimenti \end{cases}$$

e si legge «l'esecuzione dell'azione è abilitata negli intervalli di tempo compresi tra gli istanti di inserimento di una tupla di avvio e di una successiva di stop». L'altro regola la periodicità:

$$action_{t_0}(t) = \begin{cases} \left\lfloor \frac{t-t_0}{p} \right\rfloor & se\ enable(t) = 1 \\ indefinita & altrimenti \end{cases}$$

e si legge «il numero di volte in cui è eseguita l'azione a partire da t_0è pari al numero di periodi interamente trascorsi da quel momento».

2.3.2 L'implementazione in ReSpecT

L'implementazione proposta è un mapping dei concetti espressi precedentemente negli strumenti messi a disposizione dalla tecnologia: alla ricezione [dall'esterno] di una tupla d'avvio nella forma `start_periodic(P)`, il t.c. reagisce inserendo una tupla `period(P)` e innescando una reazione a catena, descritta nel dettaglio dal diagramma in figura 2 nella pagina successiva, che ha come esito l'inserimento della tupla d'azione `periodic_operation(T,N)` dove `T` è un intero che rappresenta il tempo locale ed `N` è un ordinale, detto *«tick»* che indica a quale periodo corrisponde la tupla stessa. Ovviamente, per poter tenere traccia di quest'ulma informazione, il t.c. mantiene aggiornata ad ogni step una tupla che tenga il conto dei periodi: nella fattispecie questa ha la forma `count_tick(T, N)`. Il ciclo procede sino alla ricezione [dall'esterno] di una tupla di stop nella forma `stop_periodic`.

L'utente che voglia far eseguire al t.c. un'azione con periodicità può inserire una reazione alla `out` della tupla d'azione ed inserirvi la propria *business logic*.

Viene da sé che, se si ponesse la necessità di realizzare più attività periodiche all'interno dello stesso t.c., nel caso peggiore con periodi tra loro diversi, non sarebbe necessario moltiplicare le copie delle reazioni, bensì sarebbe sufficiente prevedere un singolo loop con un periodo che sia il massimo comun divisore dei singoli e sfruttare, con apposite reazioni di supporto, il contatore presente nella tupla d'azione.

2.4 Rete logica e vicinato

Un altro concetto fondamentale per compredere i ragionamenti delle prossime sezioni è il *vicinato*: i t.c., che sono usati per implementare i compartimenti, non supportano nativamente una nozione di vicinato che quindi va definita e implementata. Nel seguito viene descritta solo la struttura logica, la conseguente implementazione e il funzionamento di un semplice meccanismo di connessione basato sullo scambio dei nomi: come facciano i compartimenti a giungere a conoscenza della reciproca esisteza è un problema che verrà affrontato in un secondo momento.

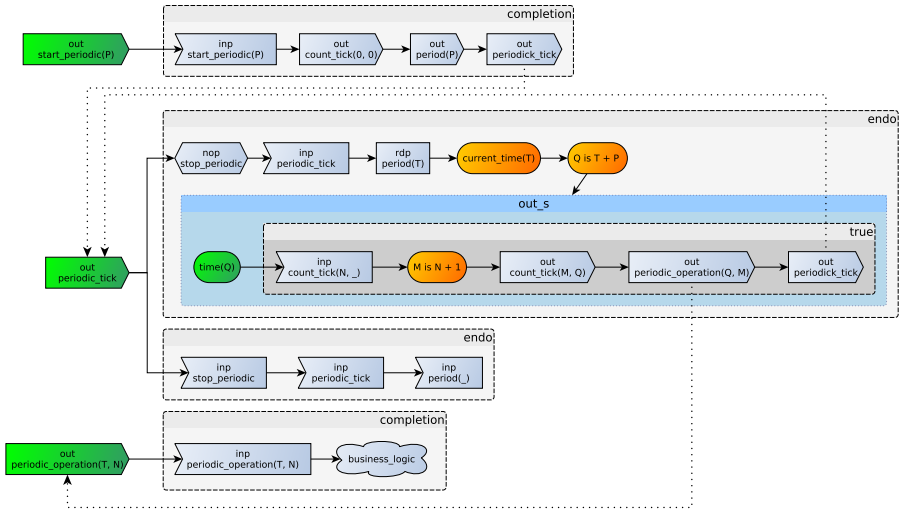


Figura 2: Reazioni ReSpecT che permettono l'esecuzione di un'operazione periodica.

2.4.1 Modellazione matematica

Ad ogni compartimento, a livello logico, in un certo istante t, è associato un *vicinato* V, ossia un insieme di riferimenti ai vicini. Il riferimento ad un compartimento, ovvero il suo identificatore, è una tripletta `(N, A, P)`, analogamente a quanto succede per un t.c..

Un compartimento c_1 è *connesso* con un compartimento c_2 in un lasso di tempo T se e solo se

$$\forall t \in T$$
$$ref(c_1) \in V_2 \wedge ref(c_2) \in V_1$$

essendo V_1 e V_2 i vicinati rispettivamente di c_1 e c_2 e $ref(\cdot)$ la funzione che, dato un compartimento, ne restituisce il riferimento. Il tal caso è lecito dire con abuso di notazione che c_1 è nel vicinato di c_2 e viceversa. La relazione di vicinanza è evidentemente simmetrica: ciò implica che il grafo che rappresenta la *rete dei vicini* non sia orientato.

2.4.2 L'implementazione in ReSpecT

Nell'implementazione proposta il vicinato di un t.c. è costituito dalle tuple in esso contenute aventi la struttura `neighbor(N, A, P)`. Sono introdotte reazioni che garantiscono che per ogni vicino la tupla sia presente al più una volta.

In realtà tutto ciò sarebbe sufficiente per molti dei ragionamenti successivi, tuttavia implicherebbe che, per ogni istanza del sistema, la rete venga costruita

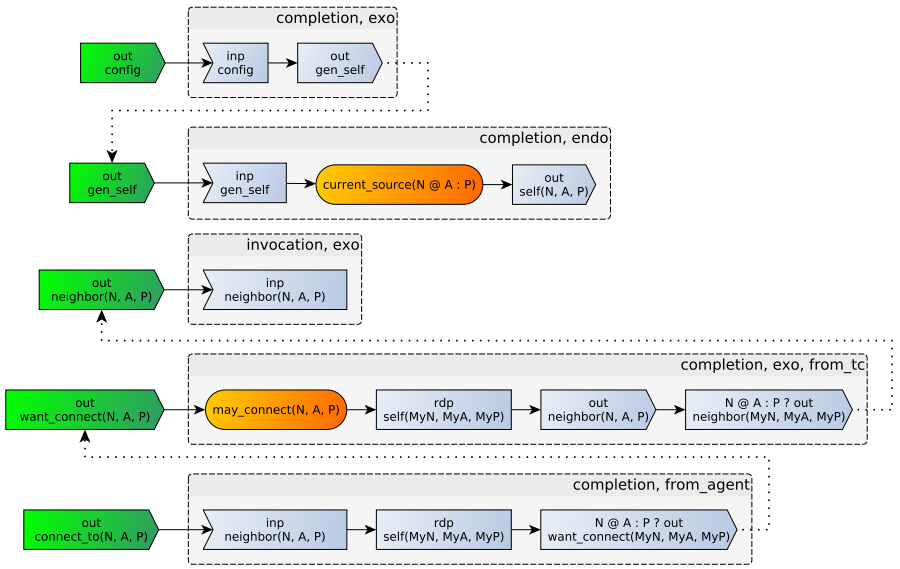


Figura 3: Rappresentazione della reazioni che permettono ai t.c. di formare un vicinato

«dall'esterno» tramite almeno un agente che faccia almeno due `out` per ogni connessione. L'approccio si è rivelato da subito inadeguato, onde per cui si è pensato di fare in modo che ogni t.c. potesse, previo comando esterno, chiedere ad un altro di connettersi e che quest'ultimo potesse confermare la richiesta, stabilendo la connessione, o ignorarla.

In figura 3 sono mostrate le reazioni inerenti il vicinato e la connessione: in buona sostanza un t.c. c_1, identificato dalla terna `(N1,A1,P1)` ha sempre la facoltà di chidere ad un altro t.c. c_2, con id `(N1,A1,P1)`, di potersi «connettere con lui» facendo la `out` su di lui della tupla `want_connect(N1,A1,P1)`. Il ricevente, c_2, può scegliere se accettare la connessione, creando quindi dentro di sé una tupla `neighbor(N1,A1,P1)` e rispondendo al mittente con una `out(neighbor(N2,A2,P2))`, o rifiutarla ignorando la richiesta. La «decisione» è presa dal predicato `may_connect(N,A,P)`, presente nella specifica di tutti i t.c., che, applicato all'id di un t.c. esterno, fallisce se è necessario impedire la conessione. L'implementazione attuale di tale predicato non fallisce mai, infatti esso ha solo la funzione di *placeholder* per politiche che un domani si potrebbero voler adottare. Un'altra reazione è stata pensata per far in modo che l'agente [configuratore] che voglia stimolare la connessione tra c_1 e c_2 debba solo preoccuparsi di fare una `out` su c_1 della tupla `connect_to(N2,A2,P2)` (o viceversa): la reazione scatenerà l'invio di una `want_connect` come descritto precedentemente. La figura 4 nella pagina successiva mostra un diagramma che descrive il tipico scenario di connessione.

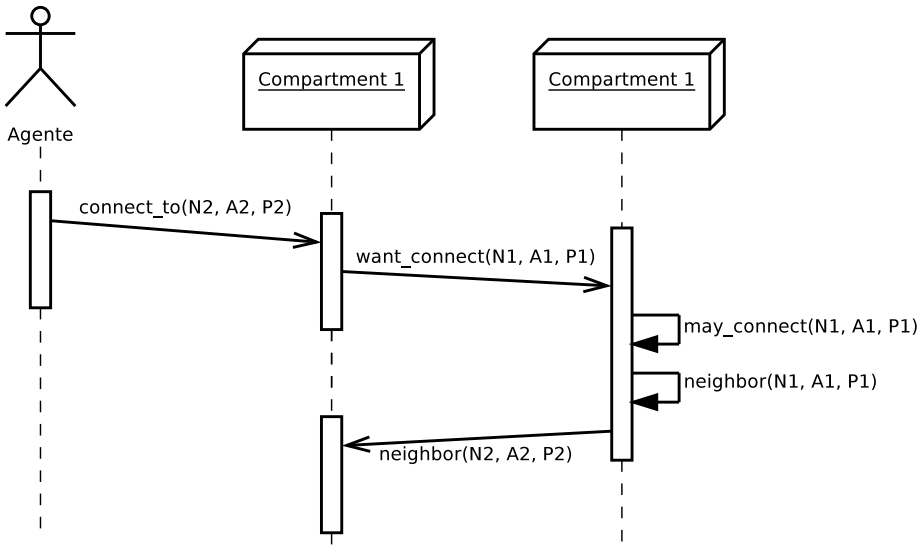


Figura 4: Rappresentazione della sequenza di `out` che permette a due t.c. di connettersi.
`may_connect` è un predicato.

A questo punto il lettore pratico di TuCSoN/ReSpecT potrebbe chiedersi:

1. come fa un t.c. a conoscere il proprio identificatore [che richiede di conoscere il proprio indirizzo sulla rete]?
2. che bisogno c'è di inviare, insieme alla richiesta di connessione, anche il proprio identificatore dato che con il predicato `start_source(S)` il ricevente potrebbe risalire all'identità del mittente?

La convenzione è la seguente: un t.c. si è *correttamente configurato* ovvero pronto a funzionare come membro di una rete di vicini, se e solo se possiede al suo interno una tupla detta *«self»* con struttura `self(N,A,P)` contenente il proprio id. Nell'implementazione proposta ogni t.c., in seguito alla `out(config)` perpetrata da un agente, genera un reazione interna che, in quanto tale, potrà recuperare l'id locale del t.c. mediante il predicato `current_source(S)` e crare così la tupla self. L'impiego della tupla self rende quindi esplicito - e modificabile - l'id di un t.c. con il vantaggio che, qualora l'indirizzo di rete del t.c. cambi (eventualità non improbabile in contesti reali), la configurazione del t.c. possa variare dinamicamente di conseguenza.

2.4.3 Alcune considerazioni su periodicità e (a)sincronia

Un t.c. è asincrono, nel senso che non deve di per sé sottostare a vincoli di natura temporale. Ciò significa che i t.c. hanno una visione (quasi) continua del tempo.

Nel seguito, grazie a quanto descritto, sarà possibile, se necessario, considerare il tempo locale di un t.c. come discreto e non più continuo, a patto che la sua specifica contenga le reazioni descritte nella sezione 2.4.2 a pagina 11. Si potrà dunque parlare, con abuso di notazione e con chiaro riferimento all'elettronica, di *evento/segnale di sincronia*, riferendosi al'inserimento della tupla d'azione, cui è possibile reagire per realizzare i comportamenti tempo-dipendenti più disparati. È sempre importante tenere a mente che i t.c., nel caso più generale, compongono un sistema distribuito e che quindi l'assunzione di sincronia globale è sbagliata a meno dell'implementazione di opportuni meccanismi.

2.4.4 Semplice DSL per istanziare reti statiche

Nelle fasi iniziali dello sviluppo capitava spesso che si dovesse istanzare una rete logica per analizzare il flusso delle informazioni ed il corretto funzionamento delle reazioni. La cosa era complicata dall'ulteriore requisito di impiegare un nodo per ogni t.c. per simulare l'effettiva indipendenza dei partecipanti alla rete logica. L'operazione si rivelò ben preso ripetitiva, noiosa e dispendiosa di tempo, motivo per cui si è optato per la creazione di un domain specific language che permettesse di scrivere piccoli script che se eseguiti permettessero di avviare i vari `TucsonNodeService` e `Inspector`. Il risultato di tale sforzo è stata la creazione di un linguaggio e di un programma Scala[1] che lo interpretasse.

Segue una descrizione semi-formale della grammatica del linguaggio chiamato operativamente `Network` (tra virgolette sequenze di simboli terminali):

```
<network> ::= <nodes>?  <arcs>

<nodes> ::= 'nodes' '{' <node> (',' <node>)* '}'

<node> ::= <name> ('@' <address> (':'  <port>)?)?

<arcs> ::= arcs '{' <arc> (',' <arc>)* '}'

<arc> ::= <node> '->' <node>
```

La semantica è molto semplice: dentro lo scope `arcs` vanno gli archi dei t.c. che si vuole interconnettere mentre dentro `nodes` vanno i t.c. che si vuole istanziare e basta. Sarà il programma-interprete a capire da solo quanti `TucsonNodeService` avviare ed a quali t.c. inviare le specifiche contenenti [almeno] le reazioni per gestire la rete.

L'interprete [2] accetta 4 argomenti:

`<net_conf>` il path del file contente la descrizione della rete, nel linguaggio appena descritto

`<net_spec>` il path del file contiene le specifice ReSpecT

[1] si veda il progetto `NetConfig`

[2] È possibile trovare il `main` nella classe `it.unibo.bitpantarei.NetConfig`

`[-start]` parametro che se presente indica che l'interprete deve anche avviare tanti `TucsonNodeService` quanti sono necessari

`[-inspect]` parametro che se presente indica che l'interprete deve anche avviare un `Inspector` per ogni t.c. istanziato

Questa parte del progetto è stata considerata sin dall'inizio una utility e per tanto non si è avuta alcuna accortezza legata all'ingegneria del software durante lo sviluppo di questo modulo.

3 Design pattern di ispirazione biologica

3.1 Evaporazione

L'evaporazione è nel seguito intesa come quel comportamento per cui un compartimento decrementa *periodicamente* la concentrazione di una specie per motivazioni qualsiasi (solitamente una diminuita rilevanza causata dall'«invecchiamento» dell'informazione): viene da sé che la sua modellazione e implementazione dipendono entrambe dalla scelta effettuata per quanto riguarda la concentrazione. Ancor prima della riflessione su «come» evaporano le specie, è importante definire «quali» specie evaporano e «quando».

Usando una metafora si può immaginare quanto segue: in natura i liquidi evaporano spontaneamente a patto di trovarsi in adeguate condizioni di temperatura e pressione; se dovessimo trasferire in un t.c. questa metafora potremmo fare in modo che, dato un insieme di tuple che rispettino il template `tuple(A)`, ogni T q.d.t. venga consumata non deterministicamente una tupla qualsiasi di tale insieme. Ciò può andar bene in certi contesti, ma è bene avere in mente che così facendo l'evaporazione colpirebbe indiscriminatamente ogni tipo di tupla che rispetti il template stesso. Ciò significa che, se si volesse far evaporare indipendentemente n tipi di tuple sarebbe necessario imporre n reazioni apposite con altrettanti template specifici.

In altri contesti può essere utile associare all'evaporazione un'azione equilibratrice che tenda a diminuire la concentrazione dei dati più concentrati: anche in questa ipotesi sarebbe necessario, come nella precedente, scegliere come modello di concentrazione il «numero di tuple uguali» ma anche consumare le tuple mediante primitive stocastiche.

È inoltre possibile definire un meccanismo di evaporazione che si basi sul modello di concentrazione come «contatore» e che colpisca uniformemente tutte le specie: ciò è facilmente realizzabile in quanto, grazie alle tuple estese, si può reperire rapidamente una lista di tutte le tipoligie di dati presenti nel t.c.; tornando nella metafora, ciò significa considerare le sostanze disgiunte tra loro nell'ambiente e quindi soggette ad evaporazione in maniera indipendente. Tuttavia non difficile immaginare come un meccanismo del genere possa facilmente diventare troppo punitivo nei confronti di informazioni poco concentrate: è per questo e per altri motivi che il focus si è infine concentrato su un tipo di evaporazione che colpisca le specie con probabilità uniforme.

Questo è in sintesi il flusso di ragionamenti che saranno formalizzati nel seguito.

3.1.1 Modellazione matematica

In [Fernandez-Marquez et al., 2013] l'evaporazione viene modellata come una *regola di transizione* istantanea che, nel caso più generale di una specie qualsiasi s con concentrazione C_s al momento del *firing* della transizione, in un compartimento L, ha la forma

$$Evaporation :: \langle L, C_s \rangle \to \langle L, evap_\lambda(C_s) \rangle$$

che può essere letta: «il compartimento transita atomicamente da uno stato in cui la specie s ha concentrazione C_s in uno in cui ha concentrazione $evap_\lambda(C_s)$» dove $evap_\lambda(\cdot)$ è una funzione, detta *«d'evaporazione»*, non crescente parametrica in λ. Notare che con questa modellazione l'evaporazione di una specie è indipendente da quella delle altre: ciò permette di semplificare di gran lunga i ragionamenti e le modellazioni. In generale $evap_\lambda : \mathbb{R} \to \mathbb{R}$ è una funzione a valori continui, utile nei casi in cui la concentrazione abbia valori dello stesso tipo, di cui la forma generale è

$$evap_\lambda(x) = \begin{cases} x - \lambda \cdot \Delta(x) & x > 0 \\ x & x \leqslant 0 \end{cases}$$

Nel seguito ci si concentrerà su funzioni $\mathbb{Z} \to \mathbb{Z}$ in quanto il modello di concentrazione di riferimento usa valori discreti. Ad esempio nel caso in cui si volesse far evaporare ogni specie di un'unità per periodio si userebbero $\lambda = 1$ e $\Delta(x) = 1$.

Nel progetto del sistema si è impiegato un semplice modello di evaporazione probabilistico tale per cui $\Delta(x)$ è la funzione costante 1 e λ è una *variabile aleatoria* con distribuzione $\mathcal{B}(1, p_{ev})$. Quindi, in sintesi, in ogni periodo T, ogni specie s, che abbia concentrazione positiva, può essere soggetta ad un decremento unitario di concentrazione con probabilità p_{ev}. Le motivazioni di tale scelta sono semplici: facilità di implementazione e maggiore malleabilità rispetto al modello non probabilistico.

3.1.2 L'implementazione in ReSpecT

L'evaporazione impiega le reazioni necessarie ad implementare un'azione periodica e quelle relative alla concentrazione. Il t.c. reagisce all'inserimento di una tupla d'azione generando una tupla-segnale che concettualmente avvia un «ciclo di evaporazione» e praticamente avvia un'iterazione su tutte le specie in cui, ad ogni passo, quindi per ogni tupla estesa `extTuple(_,tuple(A))`, viene generata una tupla `evaporate(tuple(A))`: la reazione a tale evento estrarrà un numero *float* in [0, 1) tramite il predicato `rand_float/1`[3] che, se minore

[3] supposto implementato tramite la funzione `Math::random()` di Java, che, stando alla documentazione, realizza una distribuzione uniforme di probabilità nell'intervallo [0, 1)

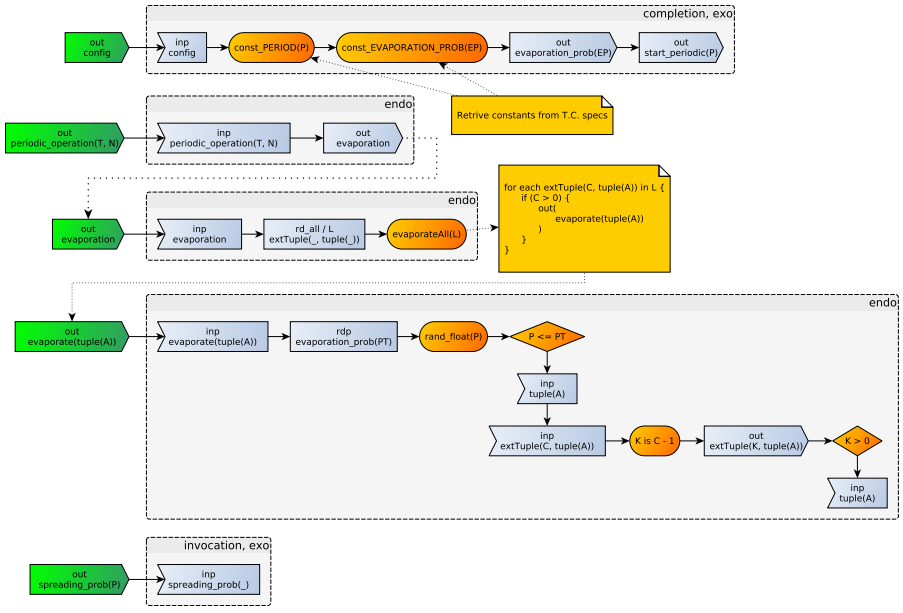


Figura 5: Rappresentazione delle reazioni che governano l'evaporazione. Sono omesse le reazioni che regolano la concentrazione e l'esecuzione di un'azione periodica per le quali bisogna fare riferimento alle immagini 1 a pagina 8 e 2 a pagina 11.

della probabilità di evaporazione, provocherà la riduzione di 1 della concentrazione della tupla corrente. L'informazione sulla probabilità di evaporazione è contenuta in un'apposita tupla con struttura `evaporation_prob(EP)` che è protetta da apposite reazioni in maniera tale che l'inserimento di una tupla con la stessa struttura sovrascriva la precedente: questo approccio fa sì che la probabilità di evaporazione dell'intero compartimento possa essere modificata a run-time. Un meccanismo simile è previsto ma non implementato per il periodo di evaporazione che dovrebbe essere espresso in numero di tick.

In figura 5 sono descritte nel dettaglio le reazioni di cui sopra.

3.2 Diffusione

La diffusione, detta anche *«spreading»*, è nel seguito intesa come quel comportamento per cui un compartimento invia una *copia* di un sottoinsieme delle specie in esso contenute ad un sottoinsieme dei suoi vicini. È importante sottolineare come lo spreading di per sé implichi non una migrazione delle informazioni bensì un moltiplicarsi delle stesse con conseguente aumento (da un punto di vista globale) della loro concentrazione. Il senso di questa scelta, si ritiene, è molto lungimirante: l'idea è di creare un *«equilibrio dinamico»* tale per cui diffusione

ed evaporazione si bilancino a vicenda e l'informazione continui a «vagare» all'intereno del sistema nel suo complesso senza appesantire troppo i singoli nodi. Come sarà chiaro a breve, realizzando una combinazione evaporazione-diffusione in cui entrambe sono regolate da leggi probabilistiche, sarà sufficiente far variare le probabilità per rompere l'equilibrio e creare così impliciti meccanismi di attrazione/repulsione.

3.2.1 Modellazione matematica

Rielaborando Fernandez-Marquez et al. [2013], lo spreading è definito come una regola di transizione che, nel caso generale di un compartimento $c \in C = \{compartimenti\ esistenti\}$ e di una specie $s \in S = \{specie\ possibili\}$, data una funzione $neigh : C \to C^n$ con n qualsiasi ed un'altra $f_c : (S \times C) \to S$, ha la forma

$$Spreading :: \langle c,\, s\rangle \to \langle c_1,\, f_c(s,\, c_1)\rangle, \ldots, \langle c_n,\, f_C(s,\, c_n)\rangle : \\ neigh(c) = (c_1,\, \ldots,\, c_n)$$

che può essere letta: «il sistema transita atomicamente da uno stato in cui una specie s è contenuta in un compartimento c ad uno in cui, previa selezione di n compartimenti $(c_1, \ldots, c_n)$ vicini a c tramite la funzione $neigh(\cdot)$, ognuno possiede una copia potenzialmente modificata della specie s tramite la funzione $f_c(s,\, c_i)$». La funzione $neigh(\cdot)$ ha quindi lo scopo di selezionare i vicini cui inviare informazione ed è quindi detta *«di selezione»* mentre la funzione $f_c(s,\, c_i)$, che può essere diversa da un compartimento all'altro, ha il compito di restituire la forma della specie in uscita noti una specie da inviare ed un compartimento cui inviarla ed è per tanto detta *«d'elaborazione»*.

Entrando nello specifico caso del progetto, si è considerata come funzione d'elaborazione la funzione identità $f_c(s,\, c_i) = s$, mentre è stata modellata come aleatoria la funzione $neigh(\cdot)$.

La modellazione di basa su assunzioni molto forti, che successivamente si cercherà di rilassare. Queste sono:

1. ipotizzando di avere un sistema a regime, in cui l'inforamazione sia già diffusa e gli utenti non ne stiano introducendo di nuova né rimuovendola;
2. ipotizzando che i compartimenti abbiano tutti lo stesso periodo di evaporazione e spreading T e che siano tra loro sincroni;
3. ipotizzando di poter per tanto ragionare in termini di n-esimo «passo del sistema» e che si possa fare la semplificazione $conc_{C,\,n}(s) \stackrel{def}{=} conc_C(s,\, nT)$ per ogni compartimento C;

allora possiamo assimilare un compartimento ad una coppia $C_n = \langle S_n,\, N_n\rangle$ in cui S_n è l'insieme delle specie presenti in C al passo n mentre N_n è l'insieme dei vicini. Come spiegato nella sezione 3.1.1 a pagina 16, data una qualunque specie $s \in S_n$ la probabilità che essa evapori è p_{ev}, definiziamo quindi un v.a.

$$evap_n(s) = \begin{cases} 1 & \to P(1) = p_{ev} \\ 0 & \to P(0) = 1 - p_{ev} \end{cases} \sim \mathcal{B}(1,\, p_{ev})$$

che esprima l'eventuale evaporazione della specie s al passo n.

Fissata una specie s, per ogni $v \in N_n$, definiamo anche una v.a. che espima la probabilità che v diffonda s verso C come

$$spread_n(s,\, v) = \begin{cases} 1 & \to P(1) = p_{sp} \\ 0 & \to P(0) = 1 - p_{sp} \end{cases} \sim \mathcal{B}(1,\, p_{sp})$$

rappresentando quindi il fatto che tutti i vicini debbano diffondere con la medesima probabilità.

A questo punto è possibile calcolare, per ogni tupla s nel compartimento C, la concentrazione al passo $n+1$ in funzione di quella al passo n:

$$conc_{n+1}(s) = conc_n(s) + \sum_{v \in N_n} spread_n(s,\, v) - evap_n(s)$$

il che permette di calcolare $I_n(s)$, ovvero l'incremento di concentrazione tra il passo n ed il successivo:

$$I_n(s) = conc_{n+1}(s) - conc_n(s) = \sum_{v \in N_n} spread_n(s,\, v) - evap_n(s)$$

di cui è particolarmente atteso il valore atteso:

$$\mathbb{E}\left[I_n(s)\right] = N_n \cdot p_{sp} - p_{ev}$$

in quanto, se confrontato con un numero qualsiasi k, permette di modellare, in maniera generale, meccanismi di attrazione, repulsione o equilibrio, esprimendo una probabilità in funzione dell'altra:

$$p_{ev} \lessgtr N_n \cdot p_{sp} - k \iff p_{sp} \gtrless \frac{p_{ev} + k}{N_n}$$

nella fattispecie, fissando $k = 0$ e scegliendo come operatore di confronto l'uguaglianza, si impone che evaporazione e diffusione si equivalgano completamente, in quanto si sta faceondo in modo che *il valore atteso dell'incremento di concentrazione sia zero*. A questo punto ci sono due strade:

1. fissare p_{sp} e far variare p_{ev}, in tal caso ogni compartimento dovrebbe interrogare i suoi vicini per conoscere la loro prob. di spreading e calibrare la propria prob. di evaporazione di conseguenza;
2. fissare p_{ev} e far variare p_{sp}, nel qual caso ogni compartimento dovrebbe informare i vicini della propria prob. di evaporazione e del proprio numero di vicini e questi dovrebbero calibrare la propria prob. di diffusione di conseguenza.

Ovviamente bisogna aver cura, nello scegliere k, di far in modo che la probabilità libera che si andrà poi a calcolare rimanga all'interno del range [0, 1].

3.2.2 L'implementazione in ReSpecT

A livello implementativo si è scelta l'opzione 2 in quanto la notifica *fire-and-forget* da un compartimento ai suoi vicini è sembrata più semplice da gestire ed implementare della controparte. Per questo motivo si è dovuto implementare un qualche meccanismo che permettesse ai t.c. di tenere traccia della probabilità di spreading dei vicini: in un primo momento si era pensato di estendere la struttura `neighbor(N, A, P)` aggiungendo un campo apposito ma

- non si voleva rischiare di compromettere il codice ReSpecT sin qui sviluppato;
- non si è ritenuto opportuno che, ogniqualvolta fosse stato necessario aggiungere un attributo d'interesse dei vicini [eventualità che poi si è effettivamente verificata], si sarebbe dovuto estendere la struttura della tupla di riferimento.

Per questi motivi si è introdotto il concetto di *«info»*, ovvero un'informazione opzionale relativa ad un oggetto: ogni info presente in un t.c. ha la forma generale `info(Object, Information)`. Nel caso di un vicino `(N, A, P)`, ad esempio, la tupla `info(neighbor(N, A, P), spread_prob(SP))` rappersenta il fatto che la prob. che il vicino diffonda verso il t.c. corrente è pari a `SP`.

A questo punto è stato sufficiente far in modo che il t.c. corrente, ad ogni variazione del vicinato, informi tutti i vicini della nuova prob. di spreading pari a $\frac{prob.\ di\ evaporazione}{numero\ di\ vicini}$ [quindi con $k = 0$, a garanzia dell'equilibrio] con la `out` di una tupla di `info` in ricezione della quale, questi ultimi sovrascrivano il precedente valore. La cosa è facilmente ottenibile così:

- ogni reazione che apporti modifiche al vicinato deve depositare nel t.c. una tupla-flag `neighbor_changed` (e.g. l'inserimento di un nuovo vicino);
- tra le varia attività svolte periodicamente c'è l'inserimento di una tupla `neighborhood` la cui reazione ha il compito, previa ricerca del flag, di iterare sui vicini inviando a ognuno il nuovo valore della probabilità di spreading.

La diffusione vera e propria è invece implementata da una cascata di reazioni molto simile a quella dell'evaporazione: la tupla d'azione genera una tupla-segnale che segna l'avvio di un ciclo di spreading. La reazione all'inserimento della tupla `spreading` itera su tutte le tuple estese `extTuple(C, tuple(X))` con concentrazione positiva e, per ognuna, genera una tupla `spread(tuple(X))`. La reazione all'inserimento di quest'ultima itera su tutti i vicini `neighbor(N, A, P)` e per ognuno genera una tupla `spread_to(tuple(X), neighbor(N, A, P))`. La reazione a quest'ultimo inserimento si occupa di recuperare la prob. di spreading connessa al t.c. destinatario (leggendo la corrispondente `info(neighbor(N, A, P), spread_prob(SP))`) e di inoltrargli una copia della `tuple(X)` previa estrazione di numero casuale inferiore a `SP`.

In figura 6 nella pagina successiva è presente una descrizione grafica di quanto appena descritto.

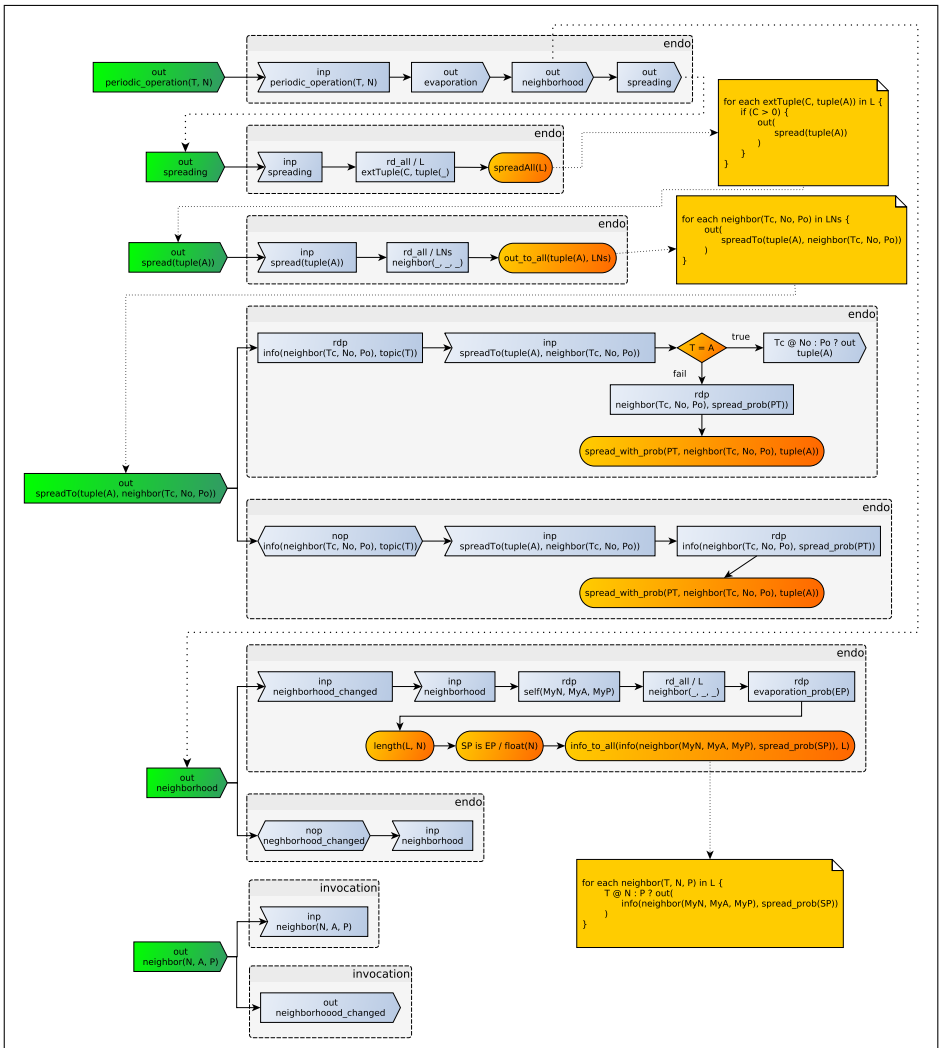


Figura 6: Rappresentazione delle reazioni che governano lo spreading. `spread_with_prob(P, N, T)` è un predicato che invia la tupla `T` al vicino `N` con probabilità `P`.

3.3 Attrattività: topic e interesse

L'attrattività nei confronti di una specie è ottenuta, come anticipato, modificando le probabilità di evaporazione e spreading. Nella fattispecie si è fissata la convenzione che ogni compartimento possa avere un *«topic»*, ovvero un insieme di specie, possibilmente vuoto, tale per cui, per ogni specie facente parte del topic, la prob. di evaporazione sia molto inferiore al normale e la prob. di diffusione molto superiore.

3.3.1 Termine Prolog VS Namespace

Il primo problema da affrontare è stato: come rappresentare sinteticamente un insieme potenzialmente infinito di concetti? La rispsota naturale è stata sfruttare il potente template matching del Prolog: l'idea era quella di permettere all'utente, nella fase di inserimento di file nel sistema, di taggarli tramite tuple Prolog che, con una sintassi annidata, rappresentassero macro-categoria, categoria, sotto-categoria, ..., identificatore, secondo una struttura lineare. Così facendo sarebbe possibile, per gli utenti interessati a scaricare, attrarre tutti i file con la stessa macro-categoria, categoria, ... oppure uno specifico file, semplicemente specificando il topic opportuno per il loro compartimento.

Così facendo un template di topic sarebbe una struttura del genere `macro_category(category(sub_category(...(id))))` ovvero un termine che ricorsivamente abbia sempre arità massima pari ad 1. Il termine più nested è l'unico che possa avere arità pari a 0, nel qual caso parleremmo propriamente di identificatore in quanto la tupla-topic nel suo complesso identificherebbe un insieme di concetti con cardinalità di al più 1; oppure essere una variabile, la cui semantica sarebbe «tutti i concetti che abbiano stuttura che *matchi* quella del topic stesso». La grammatica di riferimento è la seguente:

```
<concept> ::= <atom> | <variable> | <atom>'('<concept>')'
```

In queste ipotesi la struttura di un topic può essere *flattizzata* riducendola alla sequenza di namespaces separati da punti tipica dei package Java, di cui, eventualmente, l'ultimo possa essere un asterisco: `macro_category.category.sub_category.[...].id`. L'espressione regolare di riferimento è la seguente:

```
<concept> = <atom> ('.'<atom>)* ('.*')?
```

Le motivazioni di questa molteplicità di rappresentazioni sono molteplici:

- la rappresentazione come termine rende l'implementazione del matching semplice e naturale;
- la rappresentazione come namespace è più semplice da digitare e da comprendere per l'utente;
- inoltre il mapping dalla seconda alla prima garantisce il rispetto delle convenzioni con cui è stato definito il concetto di topic.

Nel seguito ci si riferità ai topic privi di variabili come «specifici» mentre quelli che presentino una variabile al livello più interno sono detti «generici».

3.3.2 Formulazione matematica

Concettualmente, dato un compartimento $C = \langle S, N \rangle$, un topic $T \subseteq S$ è un sottoinsieme, eventualmente improprio, di S definito per mezzo di una funzione $match_T(s) : S \to \{0, 1\}$ che vale 1 se e solo se il matching è verificato.

Nel caso dei termini Prolog, la funzione di matching coincide con l'unificazione.

3.3.3 Implementazione in ReSpecT

Il passo successivo è stato estendere la specifica dei t.c. affinché fossero capaci di gestire i topic. Il requisito era che ogni t.c. nascesse privo di topic, nel qual caso il comportamento sarebbe stato quello sin qui descritto, ma, qualora il topic fosse iniettato dall'utente a run-time, il sistema ne avrebbe localmente risentito attirando l'informazione inerente il topic.

Nel sistema sviluppato, l'inserimento del topic in un t.c. provoca:

- l'annullamento dell'evaporazione per tutte le tuple che metchino il topic nel t.c.;
- la certezza di ricevere dai vicini una copia delle informazioni che matchino il topic qualora essi ne siano in possesso;
- la «presa di coscienza» da parte dei vicini dei vicini dell'interesse dei vicini per le informazioni relative al topic, tale per cui anch'essi abbiano un'aumentata probabilità di diffondere eventuali informazioni interessanti verso gli interessati.

Per descrivere la cascata di eventi, conviene separe sia i ragionamenti che le relative spiegazioni sui tre livelli appena citati: il t.c., i suoi vicini ed i vicini dei suoi vicini (nella spiegazione si considera il caso ideale, quindi è possibile immagine la topologia di rete come un albero). Fare riferimento alle figure 7 nella pagina seguente e 8 a pagina 25 per meglio comprendere quanto segue.

Il centro di tuple

Tra le varie attività periodiche che compie ogni t.c. c'è anche quella di controllare se ci sono state variazioni del topic. Affinché la risposta sia affermativa è necassario che una reazione apposita abbia inserito una tupla-flag `topic_changed`: in tal caso tutti i vicini verrano informati del cambiamento tramite la `out` di una info dalla forma `info(neighbor(MyA,MyN,MyP),topic(T))` previa lettura della tupla `self(MyA,MyN,MyP)` e del topic `T`. Gli eventi cui il t.c. reagisce generando un flag sono ad esempio la modifica/inserimento/rimozione del topic oppure di un vicino.

La presenza nel t.c. della tupla `topic(T)` viene intercettata da una variante della reazione `evaporate(tuple(A))` che fallisce certamente se `T = A`, ovvero se `T` unifica con `A`, nel caso opposto tutto procede normalmente.

Il comportamento delle reazioni relative alla diffusione è invariato.

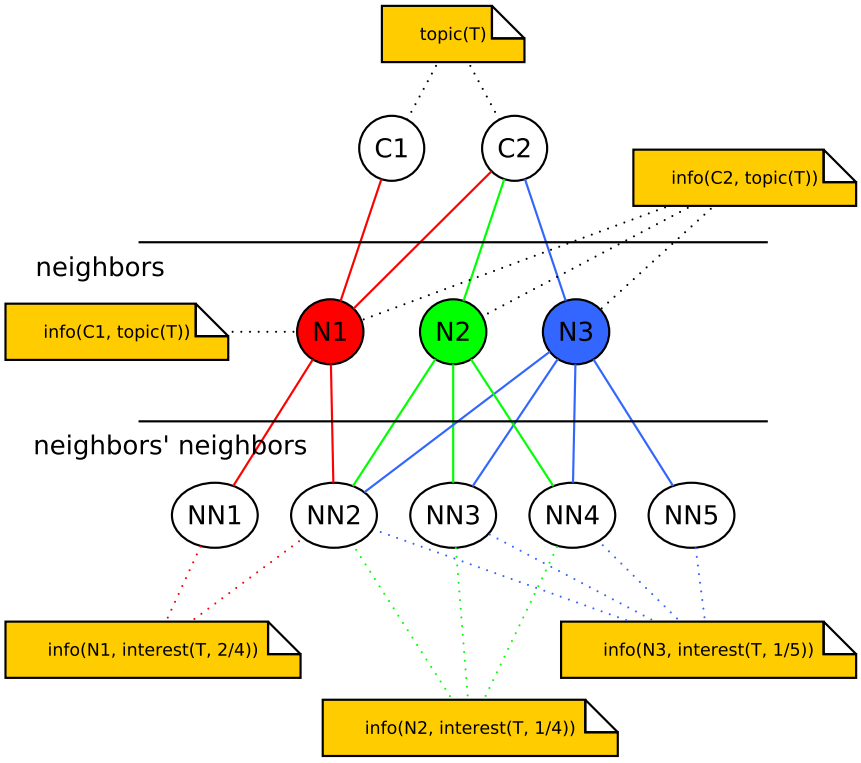


Figura 7: Rappresentazione, su una topologia di rete, di un semplice *«gradiente di interesse»* che si espande fino ad una profondità di livello 3 a partire dai compartimenti con topic.

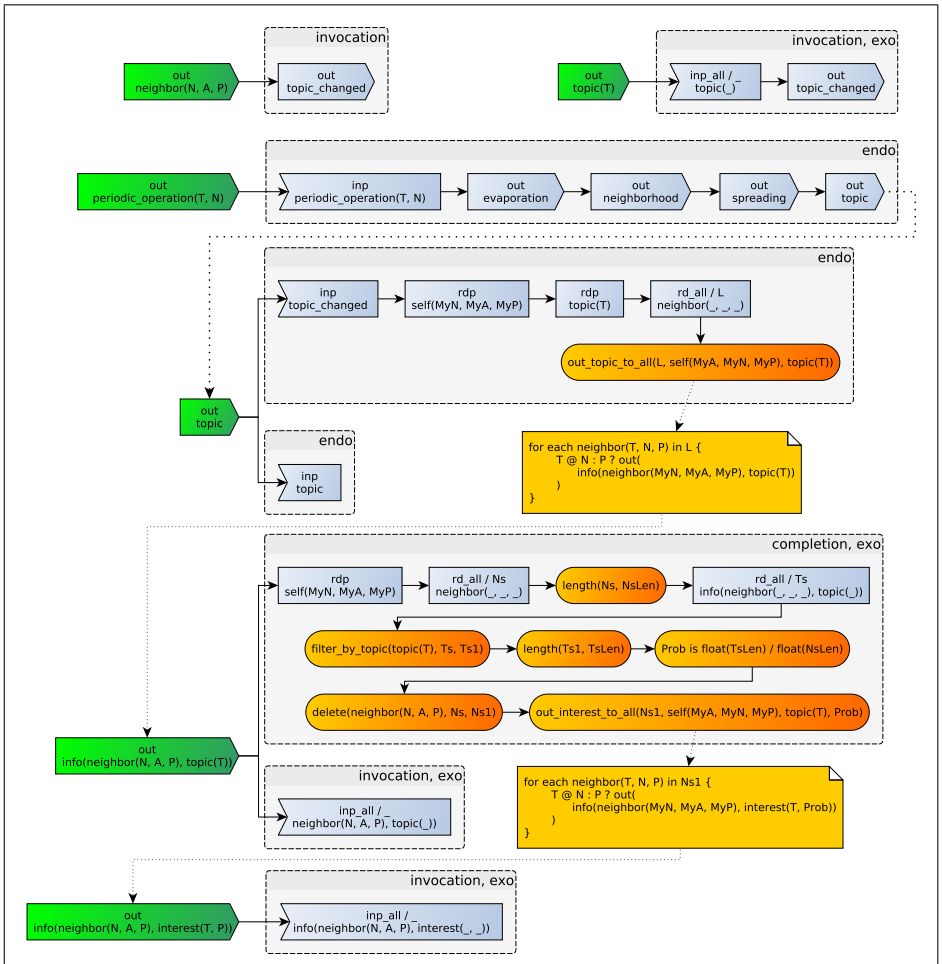


Figura 8: Rappresentazione delle reazioni che gestiscono l'effetto dell'inserimento di un topic in un compartimento.

I vicini del t.c.

I vicini che si vedono arrivare una tupla `info(neighbor(A,N,P),topic(T))` sovrascrivono eventuali info di topic precedenti, modificano la propria politica di spreading e informano tutti i propri vicini[4], mittente escluso, del loro interesse nei confronti del topic `T` tramite una tupla `info(neighbor(MyA,MyN,MyP), interest(T,GdI))`, anche qui previa lettura della `self(MyA,MyN,MyP)` e calcolo del «grado di interesse» `GdI`.

L'«interesse» è un concetto strettamente legato al topic, o, meglio, la sua generalizzazione: un topic è un interesse al 100% e, viceversa, un interesse è un argomento con una certa percentuale di rilevanza. Il grado di interesse di un topic `T` è calcolato come

$$\frac{n.\ vicini\ con\ topic\ \texttt{T}}{n.\ vicini}$$

ed è pertanto un numero in $[0, 1]$. Notare che concettualmente, pur essendo omogeneo ad una probabilità, non è tale.

Per questo riguarda la diffusione, il comportamento cambia nella misura in cui una variante della `spread_to(tuple(X),neighbor(N,A,P))`, prima di inviare, controlla se il vicino `(N,A,P)` abbia un topic e se questo faccia match con `X`, nel qual caso diffonderebbe con probabilità 1, altrimenti si comporterebbe normalmente.

I vicini dei vicini del t.c.

I «vicini dei vicini» sono quei t.c. che si vedono arrivare una `info(neighbor(A,N,P),interest(T,GdI))`. L'effetto sul loro comportamento è più blando: aumenta semplicemente la probabilità di diffondere tuple che facciano match con `T` di una quantità proporzionale a `GdI`. Nella fattispecie la reazione alla `spread_to(tuple(X),neighbor(N,A,P))`, se si accorge che il destinatario ha un interesse `interest(T,GdI)` e `T = X`, allora diffonde con probabilità `P+GdI*Q`, dove `P` è la consueta probabilità di spreading e `Q=1-P`. In tutti gli altri casi la diffusione avviene normalmente.

4 I parametri liberi

Questa sezione ha lo scopo di ricapitolare i parametri «liberi», concettuali ed implementativi, che sarebbe opportuno sottoporre a tuning in questa e qualunque successiva versione del software.

4.1 Del sistema

Questi parametri sono stati previsti dal modello e trovano una controparte a livello implementativo per cui sono modificabili. Essi sono:

Periodo fondamentale modificabile per ogni t.c. tramite la sostituzione della tupla `period(T)`, con `T` intero.

[4] per questo si parla di «vicini dei vicini»

Il valore di default è 7000 *ms*: si è osservato che, per periodi troppo corti, t.c. sullo stesso nodo si rallentano a vicenda sensibilmente.

Prob. di evaporazione modificabile per ogni t.c. tramite la sostituzione della tupla `evaporation_prob(EP)`, con `EP` di tipo float.

Il valore di default è 0.6: si voleva ottenere una certa variabilità del fenomeno senza penalizzarlo troppo.

Concentrazione iniziale ovvero la concentrazione che ha una tupla dopo essere stata inserita *per la prima volta* nel t.c.. È modificabile per ogni t.c. mediante sostituzione della tupla `initial_concentration(IC)`, con `IC` intero.

Il valore di default è 5: nelle fasi iniziali del testing, dopo aver inserito una tupla in un t.c., era necessario del tempo per «trovare» la finestra dell'inspector corrispondente. Questo parametro con valori maggiori di 1 era necessario per evitare che la tupla evaporasse prima di poter essere osservata.

4.2 Del modello

Questi parametri sono solo stati previsti nella modellazione teorica ma non sono modificabili nella pratica. Le future versioni potrebbero introdurli esplicitamente. Essi sono:

Periodo di evaporazione (espresso in numero di periodi fondamentali) si potrebbe desiderare che l'evaporazione abbia un periodo a sé stante. Attualmente concide con il periodo fondamentale.

Lambda (λ) ovvero la quantità di cui diminuisce la concentrazione ad ogni ciclo di evaporazione. Attualmente è 1.

Periodo di diffusione (espresso in numero di periodi fondamentali) valo lo stesso discorso fatto per l'evaporazione.

Incremento atteso (k) facendolo variare si possono creare molteplici effetti. Attualmente vale 0.

5 Discovery di centri di tuple TuCSoN

Questa sezione presenta alcune modalità di scoperta (*discovery*) di centri di tuple Tucson *spreaded* in rete. Le linee guida in queste fasi del progetto sono state:

- Rapid prototyping
- Scalabilità
- Indipendenza tecnologica
- Supporto cloud e web dell'architettura

5.1 Centro di tuple: definizione

Un centro di tuple (*tuple center (tc), tuple centers (tcs)*) è un contenitore di tuple con le seguenti caratteristiche: nome (*name, n*), indirizzo (*address, a*), porta (*port, p*).

La rappresentazione breve di un tc generico è: `(n@a:p)`. Questo concetto deve essere esteso per connettere tra loro centri di tuple diversi. Ogni centro di tuple ha un id univoco, un topic, una locazione e una informazione temporale (potenzialmente general purpose). Un tuple center è quindi una tupla definita come segue: `(id,name,address,port,topic,location, timestamp)`.

L'informazione legata all'indirizzo non è di particolare rilevanza al centro di tuple stesso, quanto è fondamentale per lo scambio di informazioni tra tcs.

È importante capire la semantica legata alle singole proprietà del centro di tuple:

id: tuple center identifier (statico, non cambia nel tempo)

- UUID

n: nome del nodo (statico, non cambia nel tempo)

- Identificatore univoco locale

a: indirizzo (dinamico, può cambiare nel tempo)

- IP del nodo in rete (locale o remoto)

p: porta (statico, non cambia nel tempo)

- Porta universale del servizio, 20504

topic: (dinamico, può cambiare nel tempo)

- Informazione legata alla contenuto delle tuple

location: (dinamico, può cambiare nel tempo)

- Posizione geografica del centro di tuple

timestamp: (dinamico, può cambiare nel tempo)

- Informazione general purpose temporale

5.2 Vicinanza: semantica

Due centri sono vicini (neighbourg) in base a:

- Topic: due tcs si connettono tra loro se sono interessati allo stesso topic
  - Fuzzy topic: è prevista anche la gestione di * topic
    - *: qualsiasi topic
    - *a.**: qualsiasi topic che specializzi il topic a
    - *a*: topic specifico

    Due tc sono vicini se i loro topic concidono o se uno di essi specializza l'altro:
    - *A.topic = *, B.topic = a*: A accetta B come neighborg, B rifiuta A come neighborg (poichè troppo generico)
    - *A.topic = a.*, B.topic = a*: A accetta B come neighborg (anche se B è più generico rispetto ad A, le tuple non interessanti evaporeranno nel tempo), B accetta A come neighborg
    - *A.topic = a.b, B.topic = a*: A accetta B come neighborg (anche se B è più generico rispetto ad A, le tuple non interessanti evaporeranno nel tempo), B accetta A come neighborg
    - *A.topic = b, B.topic = a*: A non accetta B come neighborg, B non accetta A come neighborg

- Location: due tcs si connettono tra loro se sono "vicini" o appartenenti alla stessa "organizzazione"
  - In questo report non si fornisce la semantica di vicinanza e organizzazione, queste sono lasciate a sviluppi futuri

5.3 Scenario

Quando un tc è creato, esso non ha conoscenza riguardo all'*environment* circostante, sia esso locale o remoto. I due ambienti operano secondo assunzioni differenti:

- LAN: ricerca di centri esistenti sulla base di broadcasting di rete
- Remoto: il broadcasting di informazioni non è disponibile in questo scenario. Un naming service è richiesto.

Diversi pattern di comunicazione sono possibili, alcuni di essi sono qui approfonditi.

Algoritmo 1 Astrazione del demone pinger

```
while (!stop) {
        // set the right condition here
        if (isAlive)
                // if the tc is alive then ping the server
                ping
        else
                failures += 1

        if (failures == retriesNumber)
                stop = true

        sleep (millisPing)
}
```

Algoritmo 2 Esempio di interazione nello scenario 1

```
Alice: "Is anybody in there?"
Bob: "Yes, I'm!"
Charlie : "Yes, I'm!"
        -> Connect Bob and Charlie to Alice
```

5.4 Discovery

Quando un centro di tuple è creato, con esso è istanziato un demone pinger (Listato 1), il cui compito è quello di indicare al sistema l'esistenza del centro di tuple stesso.

L'operazione di ping è implementata in due modalità:

1. Quando un tc (o meglio il relativo pinger *discoverer*) è creato inizia la ricerca dei vicini esistenti: esso invia una richiesta ad un indirizzo well-known tramite cui reperire le informazioni dei tc attivi (*discovered*). Discoverer raccoglie quindi tutte le informazioni utili in modo attivo

2. Il demone pinger invia un ping broadcast *stateful*, i centri attivi ricevono l'informazione e decidono se connettersi o meno al discoverer

C'è una differenza sostanziale tra queste modalità. Nel primo scenario è il tc a decidere quando ricercare i nuovi neighbourg. Nel secondo scenario il controllo è inverito (Hollywood Principle: "Don't call us, we'll call you")

5.4.1 Scenario 1

- Le interazioni di rete non sono minimizzate: per collegare due tcs almeno due interazioni sono richieste (listato 2).

- Ogni tc richiede la lista dei tcs attivi a un servizio well known

Algoritmo 3 Stub dello scenario 1

```
class Daemon_TCNeighbours_LAN(tc: List [TupleCenter]) [...] {
        val http = /* [...] */

        override def run =
        while (!isStopped) {
                tc.foreach(t => t.addNeighbour(getNeighbours(t)))
                Thread.sleep(sleepMillis)
        }

        def getNeighbours(tc: iTupleCenter): Seq[iTupleCenter] = {
                /* get request of neighbours with specified
                    parameters */
                val json = http.sendGet(/* [...] */)
                /* response deserialization */
                val list: Seq[TupleCenter] =
                    TupleCenter.fromJsonList(json)
                list
        }
}
```

Algoritmo 4 Esempio di interazione nello scenario 2

```
Alice : "Hi, I'm Alice and I want to connect"
        -> Connect Alice to Bob and other interested centers
Bob : "Hi, I 'm Bob and I want to connect"
        -> Connect Bob to Alice and other interested centers
```

- Semplice da realizzare (listato 3)

5.4.2 Scenario 2

- Le interazioni di rete sono minizzate: una sola interazione è richiesta (listato 4).
- Facile da realizzare (listato 5 nella pagina seguente)

5.5 Remote discovery

5.5.1 REST

Secondo [Fielding]: "REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I described the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles."

Algoritmo 5 Stub dello scenario 2

```
class Daemon TCNeighbours LAN(tc: List [TupleCenter]) [...] {
        val serverSocket: DatagramSocket
        val receiveData: Array [Byte]
        val receivePacket : DatagramPacket

        override def run () =
                while (true) {
                        /* waiting for the new tuple center */
                        val ntc = getNeighbour ()
                        /* add the new tuple center to existing
                            ones */
                        tc.foreach (t => t.addNeighbour(ntc))
                }
        def getNeighbour (): TupleCenter = {
                /* waiting for a ping */
                serverSocket.receive (receivePacket)
                /* ping received [............] */
        }
}
```

REST si basa su hypertext [Fielding, 2008], technologia in grado di scalare in modo efficiente e che permette di disaccoppiare client e server. Con REST, il server è libero di cambiare le risorse esposte purché l'interfaccia rimanga la stessa. Il client necessita unicamente di conoscere l'URI iniziale. Alla luce dei fatti, REST è funzionale quando è presente *Human in the loop*. Tale tecnologia consente una rapida evoluzione del server. Basandoci su queste assunzioni, l'interazione tra i componenti del sistema è stateless. Ogni scambio di messaggi deve contenere tutti i dati richiesti, riducendo così la complessità dell'interazione. Facendo uso di tale architettura un web service RESTful è facile da creare.

5.5.2 Perchè REST?

- Le interazioni remote sono basate sul protocollo HTML;
- *Technology decoupling*
    - Un client bash può interagire con il servizio remoto in modo indipendente dalla sua tecnologia (listato 6 nella pagina successiva).

5.5.3 Document database

(Fonti [Stackoverflow, 2010])

- Il termine NoSQL è diventato ubiquo e *overused*. Esso è utilizzato per indicare data storage non relazionale e che quindi non richiedono SQL per

Algoritmo 6 Esempio di client scritto in bash

```
curl -G http://127.0.0.1:3000/tuplecenters/
[
        {
        "_id" : "94fe91bc-bd53-48ac-97bc-bc4bb218d45a",
        "name" : "d",
        "address" : "127.0.0.1",
        "port" : 20504,
        "topic" : "blabla",
        "location" : "berlin",
        "timestamp" : 1445869674458
        } ,
        [...]
]
```

accesso ai dati. Tra i target principali di tali architetture si trovano velocità e scalabilità. Essendo schema-less, quesi data base non sono vincolati dallo schema stesso

- NoSQL è un'alternativa ai database relazionale, non è un rimpiazzo. Fattori importanti per la scelta sono: consistenza, disponibilità e tolleranza al partizionamento. In sistemi complessi è necessario scegliere due dei tre fattori. Se la consistenza è il fattore principale, allora il modello relazionale rimane una scelta valida

- La disponibilità è un fattore di rilievo, essa è connessa alle performance di sistema, benchmark dimostrano che i document database sono altamente performanti. La tolleranza al partizione è importante per la possibilità di scalabilità orizzontale

- RESTful HTTP API. I database NoSQL sono tipicamente accessibili in maniera RESTful. La connessione al database avviene tramite URI e le query sono HTTP call. MongoDB rappresenta un'eccezione. Di default esso si basa su interazioni tramite protocollo TCP nonostante API HTTP siano disponibili

- Incorrettamente si presume che i database non relazional siano file flat. I documenti possono contenere dati strutturati come alberi con foglie. Ogni record è un documento e può essere un insieme autonomo di dati

- JSON è utilizzato per memorizzare dati. MongoDB fa uso di Binary JSON (BSON) in grado di eseguire serializzazione binaria. BSON è la rappresentazione interna dei deti, dal punto di vista della programmazione non si nota alcuna differenza. Tramite apposite librerie (GSON, ...) è possibile serializzare gli oggetti direttamente all'interno del database evitando l'utilizzo di object-relational mapper (ORM)

- Possibilità di applicare meccanismi Map/Reduce

5.6 MongoDB

(Fonti: [Mytton, 2013, Stackoverflow, 2011])

MongoDB è un database document-based in grado di fonire perfomance elevate, elevata disponibilità e facilmente scalabile.

- Document Database
  - I documenti si mappano facilmente sui tipi di dato dei linguaggi di programmazione
  - Embedded document e array riducono il bisogno dei join
  - Schemi dinamici consentono una prototipazione rapida
- Performance elevate
- Elevata availability, server replicati con *automatic master failover*
- Facile scalabilità, sharing automatico in grado di distribuire collezione di dati tra le macchine

5.6.1 Vantaggi

- Schema-less, tale flessibilità è difficile da implementare in un RDBMS
- Scaling e autobilanciameto del carico di lavoro;
- Autoconfigurazione di macchine repplicate (aggiungere macchine = aggiungere RAM)
- Gratis e indipendente dalla piattaforma
- Livello di consistenza personalizzabile
  - *Higher performance = fire and forget inserts to MongoDB*
  - *Slower performance = wait until insert has been replicated to multiple nodes*

5.6.2 Svantaggi

- Ridondanza, non sono possibili operazioni di join
- No supporto per le transazioni, alcune operazioni atomiche sono supportate a livello di singolo documento. Nel caso in cui maccanismi map/reduce siano richiesti, architetture come Hadoop possono essere richieste

5.6.3 Perchè MongoDB?

High availability e easy scalability non sono caratteristiche richieste al momento (in quando quello presentato è un prototipo), tuttavia esse potrebbe consistere nella chiave di volta per sviluppi futuri. Una base di dati schemaless è fortemente richiesta per lo sviluppo flessibile di prototipi incrementali.

5.7 Node.js

(Fonti: [Snyder, 2015, Wayner, 2015])

Node.js è un ambiente JavaScript runtime basato su Chrome's V8 JavaScript engine. Node.js usa un modello event driven con un modello non-blocking I/O che lo rende leggero ed efficiente.

5.7.1 Why Node.js?

1. Fare pratica con JavaScript. JavaScript è un linguaggio fortemente utilizzato in ambito web. Utilizzando un database come Mongo è possibile estendere JS con un layer di persistenza;

2. Node.js consente di ottenere performance elevate. V8 compila ed esegue JS efficientemente, in quanto V8 compila JS producendo codice macchina nativo. La "magia" dietro a Node.js è l'event loop. Quest'ultimo è un single thread in grado di servire tutte le richieste I/O in modo asincrono (e una alla volta). Tradizionalmente, le operazioni IO possono essere eseguito in modo sincrono (e quindi bloccante) o in modo asincrono creando thread concorrenti. Quest'ultimo è un approccio obsoleto che consuma grandi quantità di memoria, è difficile da programmare ed è computazionalmente oneroso (basti pensare al context switch tra thread). Quando un'applicazione Node.js deve servire un'operazione asincrona, questa invia un task asincrono all'event loop insieme alla callback. Quando l'operazione asincrona è completa, l'event loop esegue la callback specificata

3. Node consente di creare in modo rapido applicazioni scalabili in grado di gestire un elevato numero di connessioni simultanee con un grande throughput

4. npm è il package manager di Node.js, è veloce, robusto e consistente. Esso è in grado di gestire e installare le dipendenze specificate nel progetto. Esso mantiene i package isolati dagli altri progetti evitando conflitti di versioni

5. Real-time made easy. L'event loop è in grado di gestire in modo efficiente applicazioni web multi-user in modalità real time utilizzando protocollo websocket. Le websocket sono un semplice canale di comunicazione bidirizezionale tra client e server. Quest'ultimo puà eseguire la push di data verso il client in modo attivo e autonomo. Websock si basa su TCP, evitando quindi l'overhead introdotto da HTTP

6. Corporate Caretaker. Node è correntemente sponsorizzato da Joyent che ha assunto un project leader e altre figure che contribuiscono al progetto. Il futuro di Node è quindi "assicurato" dalla compagnia. Aziende quali Walmart, Microsoft, Yahoo, Paypal, Voxer (...) si sono mostrate fortemente interessate in tale tecnologia

7. Hosting. Platform-as-a-Service (PaaS) provider supportano forlmamente il deployment di applicazioni Node.js

8. Node.js dispone di una plethora di database adapter disponibile nella Sails.js community per MySQL, MongoDB, PostgreSQL, Redis, e Microsoft SQL Server

9. Sviluppo rapido e agile. E' facile e veloce sviluppare e eseguire nuove applicazioni. Node.js e i relativi package forniscono inoltre funzionalità di test

5.7.2 Java vs Node.js

E' possibile scrivere web service facendo uso di tecnologia Java, segue un elenco di pro e contro:

- Rock-solid foundation, Java ha *bug*, ma la JVM è consolidata da 20 anni
- Migliori IDE di sviluppo: Eclipse, NetBeans, IntelliJ (utilizzato in questo progetto)
- Debugging remoto: esistono tool di profiling per l'identification di bottleneck e failures
- JavaScript può essere utilizzato per scrivere codice client/server in modo quasi trasparente (migliore code migration)
- Database query: Java richiedere di scrivere query in SQL da richiamare tramite appositi connettori. Mongodb consente un'integrazione ottima con JavaScript e Node.js
- Molti web service e database ritornano dati in formato JSON, parte costituente di JavaScript
- JavaScript consente la gestione agile di progetti "semplici", ma Java consente di applicare un livello maggiore di ingegneria del software;
- Node.js non richiede utilizzo di multithreading
- Java consente exploting di multithreading, ciò consente di aggirare problemi di lock dell'event loop di Node.js, se una richiesta Node.js è lenta, tutto il sistema rallenta
- Cross compiling tra i due linguaggi, engine come Rhine consentono di eseguire JavaScript all'iterno di applicazioni Java

Considerando i pro e contro (e soprattutto per interesse personale), Node.js è risultata la soluzione migliore. Si veda anche [Apache, 2015]

Algoritmo 7 Ping nello scenario remoto

```
class DaemonTCPingRemote(tc: TupleCenter, [...]) [...] {
        override def ping = {
                println(tc.toString + "pinging")
                tc.address = http.sendPost(dns address, data)
                /* post method returns the tuple center address */
        }
        /* [...] */
}
```

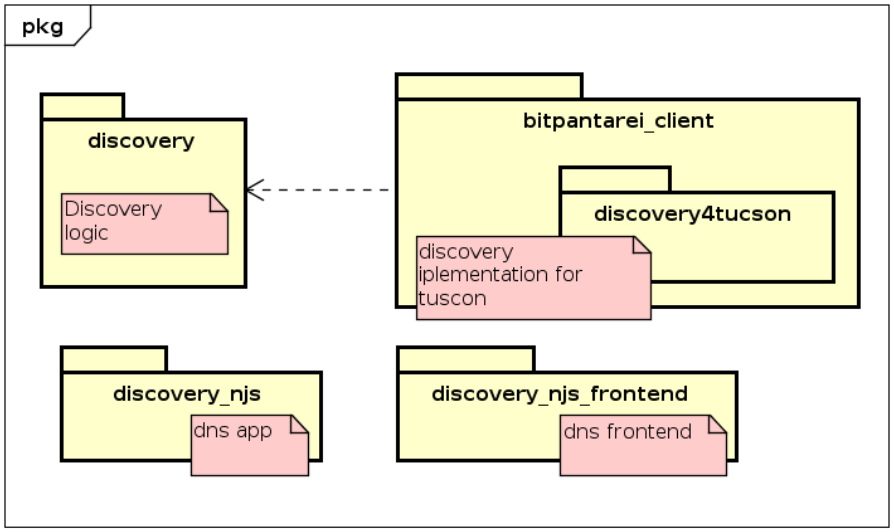


Figura 9: Struttura dei package

5.8 Local and remote addresses

Nello scenario remoto l'indirizzo del tuple center è reperibile tramite il DNS server che ritorna tale informazione al pinging tuple center (listato 7).

Nello scenario locale un paradigma differente è utilizzato seguendo il principio dell'inversione di controllo. Il pinger tuple center non necessità di conoscere il proprio indirizzo, in quando il receiver del ping estrapola tale informazione dal pacchetto in ingresso (listato 8 nella pagina successiva).

5.9 Deployment del progetto

In figura 9 è presenta una descrizione della struttura dei package:

1. `discovery`: discovery business logic;
2. `discovery_njs`: name space web application;

Algoritmo 8 Ping nello scenario remoto

```
class Daemon_TCNeighbours_LAN(tc: List [TupleCenter]) [...] {
        val serverSocket: DatagramSocket
        val receiveData: Array [Byte]
        val receivePacket: DatagramPacket

        override def run () =
                while (true) {
                        /* waiting for the new tuple center */
                        val ntc = getNeighbour()
                        /* add the new tuple center to existing
                            ones */
                        tc.foreach(t => t.addNeighbour(ntc))
                }

        def getNeighbour() : TupleCenter = {
                /* waiting for a ping */
                serverSocket.receive(receivePacket)
                /* ping received */
                val neighbour = [...] /* getting information about
                    the neighbour */
                neighbour.address =
                    receivePacket.getAddress.toString
                neighbour
        }
}
```

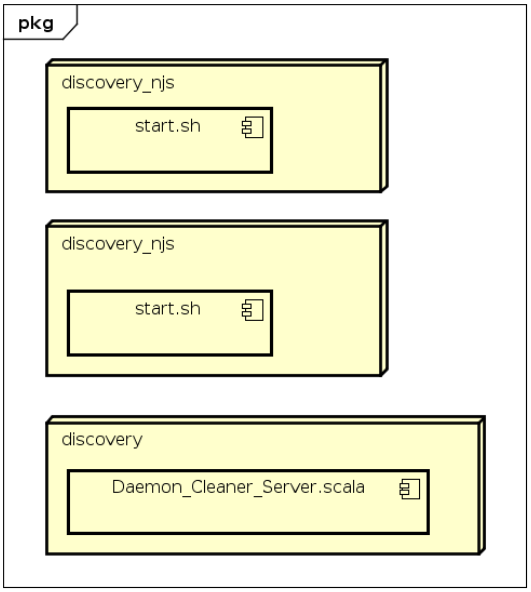


Figura 10: Deployment

3. `discovery_njs_frontend`: web application per scopo di debug del server dns;
4. `bitpantarei_client/discovery4tucson`: implementazione di componenti per tucson.

Deployment del progetto (figura 10):

1. `discovery/Daemon_Cleaner_Server.scala`: componente responsabile dell'eliminazione di elementi obsoleti dalla lista del server dns. Da attivare una *sola* volta;
2. `discovery_njs/start.sh`: file responsabile dell'avvio del servizio dns;
3. `discovery_njs_frontend/start.sh`: file responsabile dell'avvio del front end del server dns.

5.10 Debug

Per verificare l'effettivo funzionamento del sistema remoto è stata sviluppata una semplice applicazione web (basata su Node.js [Elizondo, 2012]). Essa è raggiungibile presso l'indirizzo `http://127.0.0.1:8080`. Il debug del pinging locale è invece facilmente gestibile tramite l'inserimento di breakpoint nel codice.

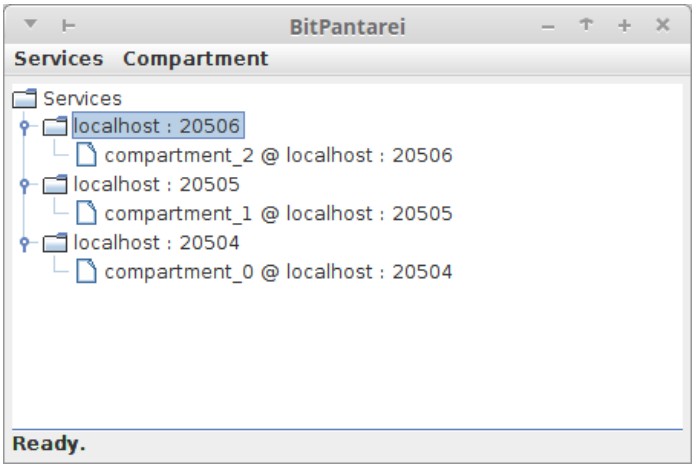


Figura 11: Schermata principale della GUI chiamata `ClientMainFrame`: visualizza servizi e compartimenti attivi e permette di crearne di nuovi, visualizzare i dati ivi presenti, terminarli, etc.

6 Assunzioni e problemi noti del sistema

- Tutti i tuple center operano sulla stessa porta globale;
- Il servizio di discovery, lan o remoto, di centri di tuple può essere utilizzato in mutua esclusione;
- Il concetto di location è predisposto ma non implementato;
- Il sistema non opera con indirizzi logici;
- Come indicatore di tempo sono utilizzati i millisecondi correnti nella macchina (tempo per ping);
- L'informazione `(n@a:p)` individua univocamente un tuple center.
- Questo progetto manca di robustezza in quanto è mantenuta attiva una singola copia del server dns;
- Le interazioni tra centri di tuple non sono sicure, in quando il protocollo di comunicazione non si basa su https.

7 L'interfaccia grafica

Per poter simulare l'interazione tra compartimenti in un sistema BitPantarei è stato creato un «client» con apposita interfaccia grafica. In figura 11 è visibile

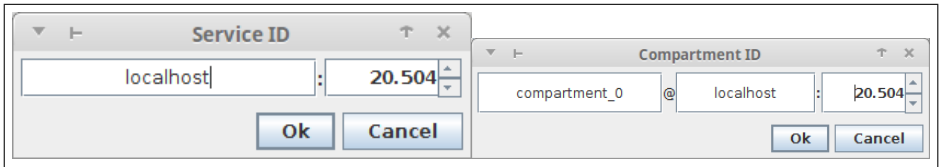


Figura 12: Il cosiddetto `IdDialog` nella sua duplice veste di generatore di ID di servizi (a sinistra) e compartimenti (a destra).

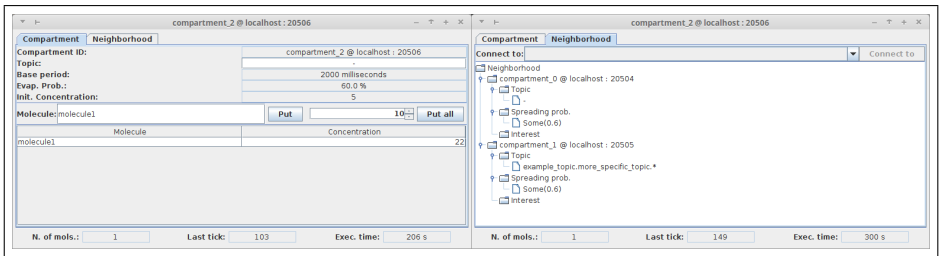


Figura 13: I due tab del `CompartmentFrame`: il primo mostra informazioni relative ad un compartimento, il secondo riguardo il suo vicinato.

un'istantanea della finestra principale. Le azioni che è in grado di svolgere sono le seguenti:

Avvio di servizi (doppio click su `Sercives` oppure da menù `Services > Start/Connect...`) avvia o si connette ad un servizio (TuCSoN) previa richiesta all'utente dell'inserimento di indirizzo e porta tramite l'apposito `IdDialog` in figura 12. Se l'indirizzo è `localhost` e non è già in esecuzione un `TucsonNodeService` con quella porta, ne viene avviato uno, altrimenti il programma prova ad interagire con il servizio remoto supposto già avviato. Se il servizio è attivo e funzionante, il t.c. di default viene configurato / si suppone configurato (a seconda che il servizio sia stato o meno avviato on-the-fly) affinché risponda ad un ping e successivamente «pingato». Dal momento in cui il ping ritorna al programma, il servizio si considera funzionante e viene visualizzato nella schermata principale come figlio di `Sercives`.

Stop di servizi (doppio click destro su un servizio opure da menù `Services > Stop/Disconnect`) provoca in generale la rimozione del servizio dalla memoria del programma. Se è stato quest'ultimo ad avviare il servizio, si occuperà anche di terminarlo.

Creazione compartimenti (doppio click su un servizio oppure da menù `Compartment > Create...`) configura un t.c. affinché si comporti come un compartimento, inviadogli le specifice ReSpecTprevia selezione di un servizio e previa richiesta all'utente dell'inserimento di nome, indirizzo e porta tramite

l'apposito `IdDialog` in figura 12 nella pagina precedente. Al termine di quest'operazione il compartimento viene aggiunto come figlio del rispettivo servizio nella GUI e può essere ispezionato.

Lista dei compartimenti attivi su un servizio noto (da menù `Services > Compartments...`) richiede ad un servizio la lista dei compartimenti attivi su di esso. Ciò è facilmente ottenibile fissando la convenzione che il t.c. di default sia debba solo contenere tuple relative ai compartimenti attivi sul suo servizio: questi ultimi sanno, grazie alla loro specifica, che prima di «partire» devono registrare la propria esistenza sul t.c. di default.

Ispezione di un compartimento avvia un `Inspector` modificato che si occupa di aggiornare il `CompartmentFrame` in figura 13 nella pagina precedente. Il buon esito dell'operazione è presentato all'utente tramite l'apparizione del frame stesso.

Presentazione/modifica dello stato del compartimento avviene, previo avvio di un'ispezione, tramite il tab `Compartment` del `CompartmentFrame`. Nella parte superiore vengono mostrati i parametri che regolano il comportamento del compartimento. L'unico editabile alla varsione attuale è il topic. Nella parte inferiore, comune a tutti i tab, sono visualizzati i valori che variano con maggiore frequenza, per avere un feedback visivo del corrento funzionamento del programma.

Presentazione delle molecole del compartimento nella parte centrale del tab `Compartment` è possibile visionare le molecole presenti nel compartimento ed eventualmente inserirne di nuove. Si noti che, alla versione attuale, l'inserimento di molecole *non* sfrutta la primitiva `out_all`, onde per cui l'inserimento di n molecole comporta l'esecuzione parallela di n primitive `out`.

Presentazione/modifica del vicinato del compartimento avviene tramite il tab `Neighborhood` del `CompartmentFrame` che mostra, con una struttura ad albero, il vicinato del compartimento corrente. Per ogni vicino sono mostrati topic, probabilità di spreading (verso il compartimento corrente) ed info varie. In alto è possibile connettere il compartimento con altri presenti nella memoria del programma.

Distruzione compartimenti comporta la rimozione del compartimento dalla memoria del programma, la cancellazione di tutte le sue molecole e specifice.

Alcune note implementative:

- lo stile è CSP (continuation style passing) ovvero «a callback» onde per cui il codice è relativamente frammentato e impiega pervasivamente le l'ACC asincrono;
- si è cercato di impedire all'utente di inserire stringhe mal formate validando tutti gli input;

- è comunque possibile effettuare sequenze di operazioni non previse;
- eccezioni non gestite dovrebbero generare un feedback grafico (alert dialog);
- le well-practices di ingegneria del software sono state applicate più alla parte back-end (progetto `bitpantarei_client`) scritta in Scala che alla parte di front-end (progetto `bitpantarei_client_gui`) scritta in Java.

8 Dipendenze e avvio del sistema

Per poter eseguire tutti i moduli sviluppati sono necessarie le seguenti tecnologie:

- Java 8
- Scala 2.11.7
- Node.js, con i moduli npm:
  - express
  - mongodb
  - mongojs
  - bson
- MongoDB
- Apache Maven (non necessario per l'esecuzione)

Per poter eseguire il sistema nel suo complesso è necessario avviare:

1. Il name server:
   `discovery_njs/start.sh`
2. Eventualmente il suo front-end:
   `discovery_njs_frontend/start.sh`
3. L'interfaccia grafica (descritta nel seguito):
   `java it.unibo.bitpantarei.client.gui.ClientMainFrame`

9 Conclusioni

Il progetto è stato fonte di riflessioni interessanti e sfide appaganti dal punto di vista implementativo relative in generale alla complessità dei sistemi distribuiti. Particolarmente gradevole è stato acquisire praticità col Prolog e con i sistemi LINDA-like come TuCSoN-ReSpecT. Segue una serie di considerazioni *a posteriori*.

9.1 Perché reazioni *timed*

A questo punto della trattazione il lettore potrebbe essersi chiesto il motivo di un uso così pervasivo di reazioni temporizzate: infatti dalla descrizione delle spefiche è evidente che le reazioni che gestiscono i principali meccanismi del sistema abbiano un avanzamento tempo-discreto e solo indirettamente legato alle azioni degli utenti. Evaporazione a parte, non era necessario implementare anche la diffusione come attività periodica: uno dei primi tantativi, ad esempio, prevedeva che il «ciclo di diffusione» avvenisse in maniera reattiva rispetto all'immissione di informazioni da parte dell'utente. Questo approccio soffriva, si ritiene, di una serie di pecche che hanno portato al suo rifiuto, di cui le principali sono:

- se la diffusione avviene solo in seguito all'immissione, manca un meccanismo che naturalmente controbilanci l'evaporazione [il rinforzo] che quindi va previsto esplicitamente. Nell'attuale concezione della diffusione, invece, essa è intrinsecamente legata all'evaporazione e nasce come sua naturale controparte: meccanismi di rinforzo possono essere creati «semplicemente» alterando l'equilibrio teorico che si viene a creare tra le due;

- soprattutto nei casi di rete scarsamente connessa, è lecito presumere, che, in caso di non meglio precisati problemi nello spreading da un nodo `A` ad uno `B` tali per cui quest'ultimo non riceva l'informazione, tutti i nodi che dipendono da `B` avranno una diminuita probabilità di ricevere l'informazione stessa. Se la diffusione è per sua natura perioca, la politica di *retry* è implicita nel funzionamento. Se, viceversa, la diffusione funsionasse in maniera meramente reattiva, tali politiche andrebbero previste esplicitamente.

Queste considerazioni hanno portato a preferire l'approccio *timed* descritto nelle sezioni precedenti.

9.2 Sviluppi futuri

Allo stato attuale il sistema è solo un prototipo, come si è visto, non privo di parti incomplete. Nel seguito vengono passate in rassegna alcune idee e proposte per migliorare il sistema.

Per quanto riguarda l'evaporazione Sarebbe possibile introdurre a livello implementativo una nozione di periodo/frequenza di evaporazione anche modificabile a run-time, analogamente a quanto succede per la probabilità di evaporazione. La maniera più naturale di realizzare la cosa sarebbe introducendo un periodo di evaporazione spresso in numero di *tick* tale per cui ogni T_{ev} periodi fondamentali di svolga un ciclo di evaporazione.

Per quanto riguarda la diffusione Vale lo stesso ragionamento fatto per l'evaporazione. Inoltre sarebbe d'uopo impiegare simulatori e/o tester per effettuare un tuning più fine dei parametri legati a questi due meccanismi. Si potrebbe poi pensare di avere una probabilità di spreading per ogni specie e non più comune a tutte; si potrebbe considerare di rompere l'assunzione di indipendenza elaborando tecniche più fini per valutare di volta in volta la probabilità di diffuzione e perché no, anche quella di evaporazione.

Per quanto riguarda topic ed interesse Anche qui le possibilità sono innumerevoli. Variazioni meno invasive potrebbero essere legate ad una diversa valorizzazione di topic e interessi riguardo al loro effetto sugli altri meccanismi. Un'idea riguarda il gradiente di interesse che, nell'implementazione proposta si ferma ad una profondità di 3 nodi: sarebbe interessante trovare un modo per estenderlo a profondità maggiori e, perché no, non limitate a priori, in tal caso bisgonerebbe prevedere un meccanismo per terminare la diffusione del gradiente d'interesse, come ad esempio fissare una soglia sotto la quale la info di un interesse venga ignorata.

Per quanto riguarda la GUI Essa è sicuramente la parte più distante dal focus del progetto, onde per cui le migliore applicabili sono innumerevoli:

- nel `CompartmentFrame`, l'`Inspector` cancella e ridisegna le informazioni ogni volta che ne arrivano di nuove: una implementazione più accorta potrebbe gestire meglio le strutture dati ed il ridisegno evitando così il fastidioso flickering delle informazioni che variano più spesso;
- nel tab `Compartment` sarebbe opportuno poter modificare tutti i campi così da poter effettivamente influire su tutti i parametri di funzionamento del compartimento, analizzando gli effetti delle variazioni;
- sarebbe opportuno un generale refactory del codice in un'ottica più vicina all'ingegneria del software.

Riferimenti bibliografici

Apache. Pojo web services using apache axis2, June 2015. URL https://axis.apache.org/axis2/java/core/docs/pojoguide.html.

Luis Elizondo. Mongoose vs mongojs on node.js, August 2012. URL http://www.luiselizondo.net/mongoose-vs-mongojs-on-node-js/.

Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluís Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1): 43–67, 2013.

Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Roy Thomas Fielding. Rest apis must be hypertext-driven, October 2008. URL http://roy.gbiv.com/untangled/2008/REST-apis-must-be-hypertext-driven.

David Mytton. Checking if a document exists - mongodb slow findone vs find, March 2013. URL https://blog.serverdensity.com/checking-if-a-document-exists-mongodb-slow-findone-vs-find/.

Rick Snyder. Why node.js beats java and .net for web, mobile, and iot apps, August 2015. URL http://www.infoworld.com/article/2975233/javascript/why-node-js-beats-java-net-for-web-mobile-iot-apps.html.

Stackoverflow. What's the attraction of schemaless database systems?, October 2010. URL http://stackoverflow.com/questions/3856222/whats-the-attraction-of-schemaless-database-systems.

Stackoverflow. Pros and cons of mongodb?, March 2011. URL http://stackoverflow.com/questions/5244437/pros-and-cons-of-mongodb.

Peter Wayner. Java vs. node.js: An epic battle for developer mind share, February 2015. URL http://www.infoworld.com/article/2883328/java/java-vs-nodejs-an-epic-battle-for-developer-mindshare.html.