

Progetto di Sistemi Distribuiti A.A. 2012/2013,
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Caso di studio: **Mapping TOTA on TuCSoN**

Membri

Fabio Paolucci, Roberto Togni

Sommario

[1. Introduzione](#)

[2. Il modello TOTA](#)

[3. Analisi del problema e Piano di Lavoro](#)

[3.1 Topologia](#)

[3.2 Propagazione](#) [Sintassi Astratta](#)

[3.3 Mantenimento](#) [Sintassi Astratta](#)

[4. Progettazione del Mapping](#)

[4.1 Definizione di una topologia in TuCSoN](#)

[4.2 Definizione della sintassi TOTA in TuCSoN](#) [Tupla TOTA](#)

[4.3 Definizione della semantica TOTA in TuCSoN](#)

[5. TOTA Testbed: Routing di messaggi mediante feromone](#)

[5.1 Lo scenario applicativo](#)

[5.2 Analisi e Definizione della tuple TOTA](#)

[5.2.1 TOTA Pheromone](#)

[5.2.2 TOTA FollowPheromone](#)

[5.3 Configurazione della topologia TOTA ed esecuzione del testbed](#)

[Prima Fase di Esecuzione \(Rilascio e Decadimento del Feromone\)](#)

[Seconda Fase di Esecuzione \(Invio del Messaggio in tupla FollowPheromone\)](#)

[6. Conclusioni e Sviluppi Futuri](#)

[7. Appendici](#)

[8. Riferimenti bibliografici](#)

1. Introduzione

La stigmergia, traendo ispirazione dalla natura, mira a gestire indirettamente e in forma decentralizzata la coordinazione all'interno di un sistema multi-agente complesso. Al di là dei numerosi studi a riguardo, il problema di definire strumenti *general-purpose* per programmare sistemi multi-agente basati su stigmergia è ancora aperto. In questo contesto, il presente elaborato si pone l'obiettivo di affrontare il problema del mapping del modello di coordinazione stigmergica TOTA (*Tuples On The Air*) sul middleware TuCSoN, formalizzando la sintassi della tupla TOTA e la sua semantica.

2. Il modello TOTA

Nel modello TOTA [1] l'entità chiave è la tupla, sulla quale ci si appoggia sia per rappresentare e conservare l'informazione, che per gestire la coordinazione stigmergica tra gli agenti. Questi ultimi possono muoversi nell'ambiente in cui sono situati, costituito da una rete dinamica e distribuita di nodi TOTA, e interagire indirettamente tra loro semplicemente *iniettando* tuple nello spazio locale e percependo, sempre localmente, le tuple propagate. In questo senso, la tupla incapsula delle regole che le consentono, una volta inserita all'interno della rete di nodi, di propagarsi autonomamente e di evolversi nel tempo in base alle proprietà dell'ambiente e alle sue variazioni.

La tupla TOTA risulta dunque così definita:

$$T = (C,P,M)$$

dove C definisce il contenuto informativo immagazzinato dalla tupla, P la regola di propagazione e M la regola di mantenimento.

3. Analisi del problema e Piano di Lavoro

Il problema richiede di delimitare con chiarezza il modello TOTA, per il quale non è presente una formalizzazione, e di specificarne, se possibile, un mapping su un middleware di coordinazione già esistente, TuCSoN, il quale consente una programmazione dinamica dei centri di tuple distribuiti mediante il linguaggio ReSpecT.

Prima di tutto, è necessario concentrarsi sulla fattorizzazione delle astrazioni legate alla nuova tupla TOTA:

- Com'è definita la topologia della rete di nodi TOTA
- Cos'è la propagazione in TOTA.
- Cos'è il mantenimento in TOTA.

Fatto ciò, è possibile procedere alla definizione del mapping, ovvero come tali concetti e la loro semantica siano rappresentabili in TuCSoN.

3.1 Topologia

Ciascun nodo della rete mantiene il riferimento a un insieme di nodi vicini, di conseguenza la topologia della rete TOTA è definita in termini di relazioni di vicinato.

3.2 Propagazione

La regola di propagazione specifica come deve essere distribuita la tupla TOTA nella rete di nodi. In particolare, ciò include determinare lo *scope* di propagazione della tupla, consistente nella definizione di un raggio di propagazione (e.g. in termini di salti tra nodi) e di un'eventuale direzione spaziale in cui essa deve avvenire, se la topologia è stata definita in questi termini (e.g. punti cardinali). In aggiunta, la propagazione può essere regolata da particolari condizioni (guardie), che specificano se la tupla può entrare o meno in un nodo, e contenere una regola che specifica il cambiamento (eventualmente nullo) a cui è sottoposto il contenuto della tupla, in termini di una funzione di variazione.

Sintassi Astratta:

```
<propagation_rule> ::= (
    <propagation_scope>,
    <propagation_guard>,
    <modify_content_function> )

<propagation_scope> ::= ( <distance> ,
    <direction> )

<distance> ::= ( <distance_type>, <distance_value> )
<direction> ::= ( <direction_type>, <direction_value> )
```

3.3 Mantenimento

La regola di mantenimento definisce come una tupla TOTA deve reagire agli eventi che avvengono nell'ambiente in cui si trova. In questo caso, quindi, si ragiona tramite reazioni (*Evento*, *Azione*). I tipi di eventi si possono classificare in: eventi temporali, eventi relativi a modifiche nella struttura della rete (e.g. inserimento di un nuovo nodo nel vicinato) o modifiche allo spazio di tuple, e possono essere causa (*Azione*) di un aggiornamento del contenuto della tupla TOTA o di una nuova propagazione di quest'ultima.

Sintassi Astratta:

```
<maintenance_rule> ::= ( <reaction> * )

<reaction> ::= ( <event>, <action> )

<event> ::= <time> |
    <change_neighbourhood> |
    <change_tuple_space>
```

```
<action> ::= <propagate> |
           <modify_content_function>
```

4. Progettazione del Mapping

L'approccio che si intende seguire prevede di definire un *mapping*, quindi di prendere le astrazioni TOTA e di specificarle all'interno del middleware TuCSoN mediante la definizione di un linguaggio che definisce cos'è la tupla TOTA, per poi realizzare un sistema di coordinazione TOTA, basato su centri di tuple programmati tramite reazioni ReSpecT che incapsulano la semantica del modello TOTA.

Si cercherà di proporre una risoluzione estendibile e del tutto *general-purpose*, dunque non specifica di un particolare caso applicativo preso in esame.

4.1 Definizione di una topologia in TuCSoN

I nodi TuCSoN non possiedono un concetto di localizzazione di nodi vicini (nell'implementazione attuale non è possibile modellare topologie di nodi). Per ovviare a questa mancanza, è possibile incapsulare le informazioni relative alle relazioni di vicinato sottoforma di tuple logiche inserite al boot del nodo. Si possono considerare diverse topologie di vicinato; nel nostro caso applicativo si tratterà un vicinato di tipo generico espresso nella forma:

```
neighbourhood( <neighbour> * )
<neighbour> ::= neighbour(Tc,NodeIp,NodePort)
```

L'astrazione di nodo TOTA è così mappata su un nodo TuCSoN, considerando un solo centro di tuple (che denomineremo *totatc*).

4.2 Definizione della sintassi TOTA in TuCSoN

Tupla TOTA

- `<tota_tuple> ::= tota_tuple(<ID> ,
 <tuple_content> ,
 <propagation_syntax> ,
 <maintenance_syntax>)`
- `<ID> ::= tota_uuid( <UUID> )`
Identificatore UUID della tupla
- `<tuple_content> ::= content( <logic_tuple> )`
Dalle specifiche del modello TOTA la tupla contenuto è una tupla logica classica che descrive il contenuto informativo

La sintassi che definisce la regola di propagazione:

- `<propagation_syntax> ::= propagation_rule(`

`<propagation_scope> ,`
`<propagation_guard> ,`
`<modify_content> )`

 - `<propagation_guard> ::= guard( <condition> )`
 - `<condition> ::=`
`prolog_solve(<file_path>,<prolog_goal>) |`
`java_exec( <file_path> )`
Per mantenersi general-purpose, il codice della guardia, viene definito a livello applicativo all'interno di una teoria Prolog o di un file Java.

- `<propagation_scope> ::= scope( <distance_term> ,`

`<direction_term> )`

 - `<distance_term> ::= distance( <distance_type> ,`

`<distance_value> )`

A fini di un successivo esempio applicativo si specifica come tipo di distanza solo l'hop (ovvero il salto fra i nodi) lasciando aperto a possibili estensioni

```
<distance_type> ::= hop
<distance_value> ::= #hop | inf
```

#hop è il numero di salti, inf rappresenta il caso con salti infiniti

- `<direction_term> ::= direction(<direction_type> ,`

`<direction_value> )`

A fini di un successivo esempio applicativo si specifica come tipo di direzione broadcast (ovvero a tutto il vicinato) lasciando aperto a possibili estensioni

```
<direction_type> ::= broadcast
<direction_value> ::= null
```

- `<modify_content> ::= modify_content(<function_code>)`
 - `<function_code> ::=`
`prolog_solve(<source>,<prolog_goal> ) |`
`java_exec( <source> )`
Per mantenersi general-purpose, il codice delle funzione di modifica, viene definito a livello applicativo all'interno di una teoria Prolog o di un file Java.

La sintassi che definisce la regola di mantenimento:

- `<maintenance_syntax> ::= maintenance_rule ( <reaction> * )`
 - `<reaction> ::= reaction( <event>, <action> )`
 - `<action> ::= propagate | modify_content(<function_code>)`

Alcuni tipi di eventi che possono scatenare un'azione (la sintassi può essere estesa con ulteriori eventi)

- ```

o <event> ::=
 eventi temporali
 timer(<number_timer>) |

 cyclic_timer(<number_timer>) |

 evento immediato, all'ingresso della tupla nel nodo
 node_entrance |

 evento di modifica del vicinato
 change_neighbourhood(<changing_type>,
 <direction_term>)

 evento di modifica nel centro di tupla
 change_tuple_space(<environment> ,
 <changing_type> , <tuple_template>)

```
- <number\_timer> ::= #time  
*#time, numero intero dopo il quale viene generato un time alarm*
  - <changing\_type> ::= new | delete  
*indica il tipo di cambiamento subito nell'ambiente ovvero inserimento di un nuovo nodo o eliminazione di uno esistente*
  - <environment> ::= local | neighbourhood
  - <tuple template> ::= template della tupla

### 4.3 Definizione della semantica TOTA in TuCSoN

L'intera gestione della tupla TOTA , ovvero la semantica del linguaggio sopra definito, è specificata in termini di reazioni ReSpecT, definite all'interno di un file `.rsp` caricato al boot del nodo. Per mantenere tale semantica il più generale possibile, le componenti *application-specific* della tupla TOTA (e.g. le particolari condizioni di guardia, o le regole di modifica del contenuto) sono definite all'interno di codice eseguibile mediante la primitiva `spawn` di TuCSoN.

La propagazione della tupla tra i nodi, invece, viene resa tramite l'uso delle primitive TuCSoN di *linking*. In questo caso, è però necessario fare la seguente assunzione: al momento della configurazione della rete di nodi TOTA, tutti i file relativi alle componenti applicative devono essere presenti in ogni nodo, poiché tali file non possono essere propagati insieme alla tupla. Per i dettagli sulle reazioni TOTA implementate si rimanda al file `TOTA_Reactions.rsp`, presente in appendice.

## 5. TOTA Testbed: Routing di messaggi mediante feromone

*Coordinazione di agenti basata su feromone*

### 5.1 Lo scenario applicativo

Un agente *PheromoneAgent* si muove all'interno della rete di nodi TOTA emettendo in ciascun nodo del suo tragitto una tupla TOTA *Pheromone*, il cui contenuto ne specifica l'intensità che decade nel tempo. Una volta completato il suo percorso, l'agente si mette in attesa di un messaggio nell'ultimo nodo in cui si trova.

Un secondo agente *MessageAgent* emette una tupla TOTA *FollowPheromone*, che incapsula il messaggio da inviare come contenuto. Questa particolare tupla è programmata in modo tale da propagarsi solo nei nodi in cui il valore di feromone risulta superiore a quello dell'ultimo nodo in cui si trovava; ciò consente alla tupla di seguire la "traccia di feromone" lasciata dall'agente *PheromoneAgent* (la quale è sempre più forte man mano che ci si avvicina ad esso), in modo da fare arrivare il messaggio a destinazione senza avere alcuna informazione a priori su di essa. La tupla *FollowPheromone* è, così, inserita all'interno del centro di tuple in cui *PheromoneAgent* sta attendendo il messaggio, che quindi viene recepito dall' agente.

### 5.2 Analisi e Definizione della tuple TOTA

Il caso di studio presentato prevede la specifica delle seguenti tuple TOTA.

#### 5.2.1 TOTA Pheromone

```
tota_tuple(tota_uuid(<pheromUUID>) ,
```

```

 content(pheromone(<#n_initial_level>)),

 null,

 maintenance_rule([

 reaction(cyclic_timer(<#ms_time>),

 modify_content(prolog_solve('tota/config/ModifyContentMaintenancePheromone.pl', dec(pheromUUID)))

])

)

```

- **Contenuto:** valore numerico di intensità del feromone.
- **Regola Propagazione:** nel caso di questa tupla non è definita (`null`) in quanto la tupla feromone viene rilasciata dall' agente localmente al singolo nodo in cui è situato e non si propaga.
- **Regola Mantenimento:** Il valore numerico di intensità decrementa ciclicamente (evento `cyclic_timer`) nel tempo fino a raggiungere il valore 0, dopodiché cessa di decrementare (evaporazione del feromone).

Di seguito, si riporta la teoria Prolog `ModifyContentMaintenancePheromone.pl` che implementa il decadimento del feromone.

`ModifyContentMaintenancePheromone.pl`

```

dec(Id) :- rd(tota_tuple(Id, content(pheromone(X)), P, M)),
 X1 is X-1, X1>=0,
 in(tota_tuple(Id, content(pheromone(X)), P, M)),
 out(tota_tuple(Id, content(pheromone(X1)), P, M)).

```

### 5.2.2 TOTA FollowPheromone

```

tota_tuple(tota_uuid(<msgUUID>) ,

 content(msg('Hello, world!')),

 propagation_rule(
 scope(distance(hop,inf), direction(broadcast,null)),

 guard(prolog_solve('tota/config/PropagationGuardMessage.pl', decide_enter(<msgUUID>))),

 modify_content(prolog_solve('tota/config/ModifyContentPropagationMessage.pl', mod(<msgUUID>)))

)

```

```

 maintenance_rule([
 reaction(node_entrance, propagate)
])
)

```

- **Contenuto:** messaggio da inviare.
- **Regola Propagazione:** la tupla si propaga in maniera broadcast a tutti i vicini del nodo corrente per un numero infinito di hop. Nel momento in cui viene propagata, il suo contenuto viene modificato aggiungendo l'informazione sul valore di intensità del feromone nel nodo corrente, come specificato nella teoria `ModifyContentPropagationMessage.pl`.  
La guardia, attivata nel momento in cui la tupla propagata entra in un nodo, controlla che il valore di feromone contenuto nella tupla sia superiore a quello del nodo attuale, come specificato nella teoria `PropagationGuardMessage.pl`.
- **Regola Mantenimento:** in questo caso l'unica azione di mantenimento è la propagazione, scatenata non appena la tupla entra nel nodo (`node_entrance`).

Di seguito, si riportano le teorie Prolog `ModifyContentPropagationMessage.pl` che implementano la propagazione della tupla `FollowPheromone` contenente il messaggio e `PropagationGuardMessage.pl` che implementa la guardia di propagazione.

#### ModifyContentPropagationMessage.pl

```

mod(Id) :- rd(to_modify(tota_tuple(Id, content(msg(X)), P,M))),
 rdp(tota_tuple(Id1, content(pheromone(V)), P1,M1)),
 !,
 in(to_modify(tota_tuple(Id, content(msg(X)), P, M))),
 out(to_prop(tota_tuple(Id, content(msg(X), V), P, M))).

mod(Id) :- in(to_modify(tota_tuple(Id, content(msg(X)), P,M))),
 out(to_prop(tota_tuple(Id, content(msg(X), 0), P,M))).

```

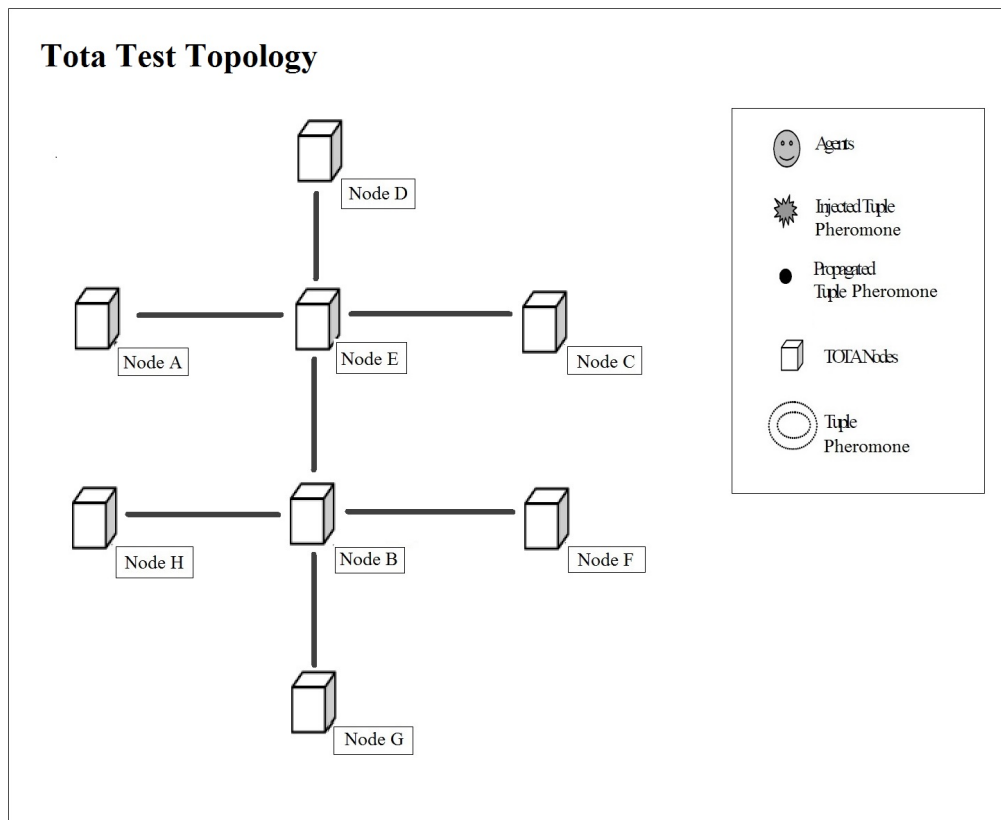
#### PropagationGuardMessage.pl

```

decide_enter(Id) :-
 in(prop(tota_tuple(Id,content(msg(X),Pheromone),P, M))),
 rdp(tota_tuple(Id1, content(pheromone(V)), P1, M1)),
 V>Pheromone,
 out(guard_success(tota_tuple(Id, content(msg(X)), P, M))).

```

### 5.3 Configurazione della topologia TOTA ed esecuzione del testbed



La topologia adottata per il test consiste in 8 nodi TuCSon avviati all'interno della macchina (*Netid = localhost*) con le seguenti porte:

- Nodo A: 20504
- Nodo B: 20505
- Nodo C: 20506
- Nodo D: 20507
- Nodo E: 20508
- Nodo F: 20509
- Nodo G: 20510
- Nodo H: 20511

Si riportano anche le relazioni di vicinato di ogni nodo in forma di tupla logica:

#### *Nodo A*

```
neighbourhood([neighbour(totatc,localhost,20508)])
```

#### *Nodo B*

```
neighbourhood([neighbour(totatc,localhost,20508),neighbour(totatc,localhost,20509),neighbour(totatc,localhost,20510),neighbour(totatc,localhost,20511)])
```

### Nodo C

```
neighbourhood([neighbour(totatc,localhost,20508)])
```

### Nodo D

```
neighbourhood([neighbour(totatc,localhost,20508)])
```

### Nodo E

```
neighbourhood([neighbour(totatc,localhost,20504),neighbour(totatc,localhost,20505),neighbour(totatc,localhost,20506),neighbour(totatc,localhost,20507)])
```

### Nodo F

```
neighbourhood([neighbour(totatc,localhost,20505)])
```

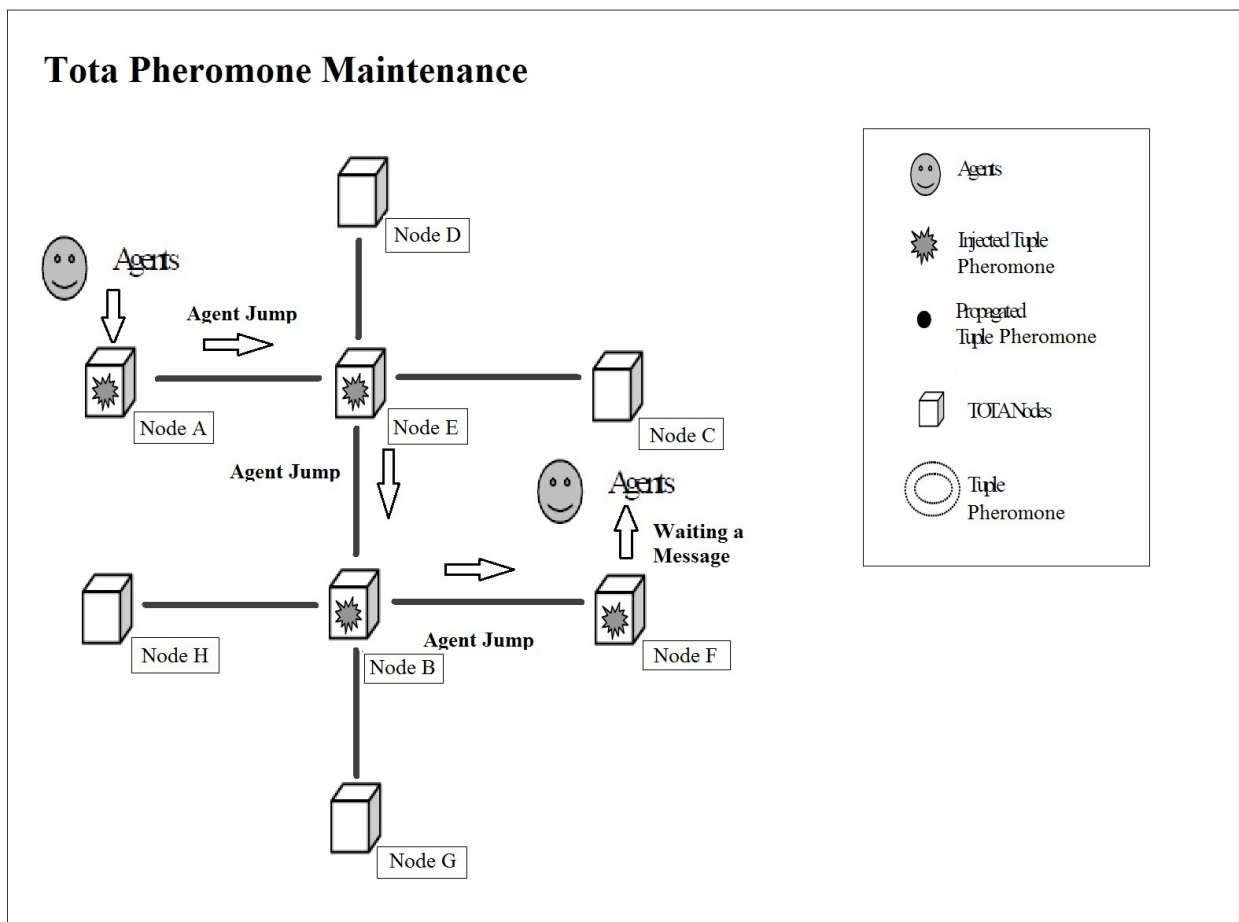
### Nodo G

```
neighbourhood([neighbour(totatc,localhost,20505)])
```

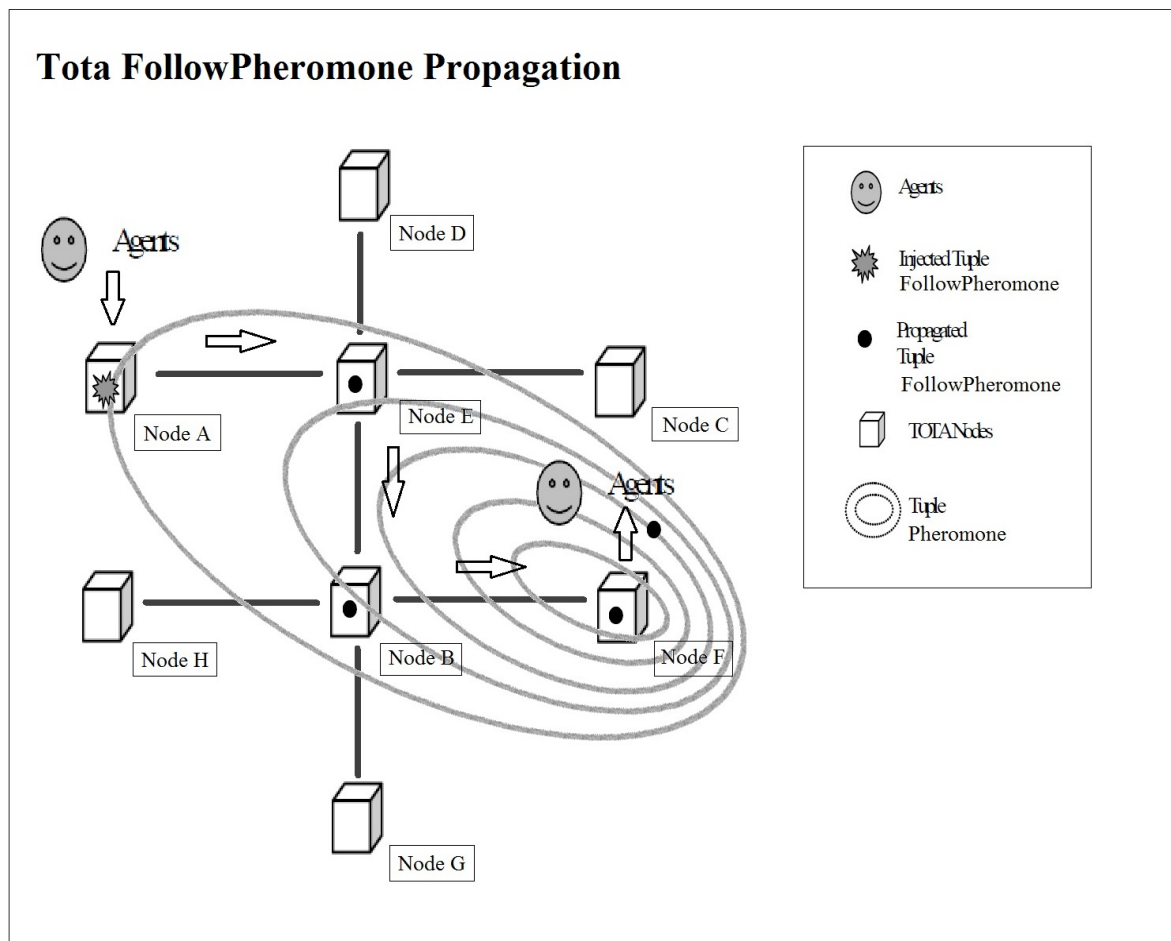
### Nodo H

```
neighbourhood([neighbour(totatc,localhost,20505)])
```

## Prima Fase di Esecuzione (Rilascio e Decadimento del Feromone):



## Seconda Fase di Esecuzione (Invio del Messaggio in tupla FollowPheromone):



## 6. Conclusioni e Sviluppi Futuri

Il modello TOTA propone un approccio *general-purpose* per la definizione programmabile e *application-level* di forme di coordinazione stigmergica in sistemi multiagenti complessi. Grazie a una specifica formale della sua sintassi e semantica è stato possibile delimitare tale modello, e progettare un mapping sul middleware di coordinazione TuCSoN. Tuttavia, per incrementare la sua usabilità in innumerevoli contesti, risulta importante arricchire il primo prototipo rilasciato al termine di questo elaborato, con molti altri elementi. In particolare, l'implementazione della semantica TOTA in ReSpecT riguarda solamente gli elementi essenziali per proporre un primo esempio applicativo, lasciando aperto a future estensioni un'implementazione esaustiva: per esempio, per ciò che riguarda il mantenimento, si sono gestiti solamente gli eventi `node_entrance`, `timer` e `cyclic_timer`, (`change_neighbourhood` e `change_tuple_space`, riguardanti la variazione dinamica dell'ambiente, non sono stati implementati) mentre la definizione di guardie e regole di modifica di contenuto *application-specific* può attualmente avvenire solo in linguaggio Prolog (nonostante la sintassi presenti anche la *signature* Java). Infine, la sintassi del linguaggio definito, può essere a sua volta estesa per formalizzare ulteriori astrazioni, in quanto per ora contempla solo una direzione di propagazione tipo `broadcast` a tutto il vicinato e una metrica basata su `hop`, salti tra nodi.

## 7. Appendici

TOTA\_Reactions.rsp

```
%Reazione triggerata quando viene inserita una tupla TOTA da un agente
reaction(
 out(tota_tuple(tota_uuid(Uuid) , Content , Propagation_rule ,
 maintenance_rule(Reaction_list)
)),
 (exo, completion),
 (
 spawn(solve('tota/config/TOTA_Function_Utility.pl',
 iter_list(Uuid , reaction_rule , Reaction_list)))
)
).

%Reazione per programmare il comportamento del TC in base alla maintenance rule
%(evento Timer)
reaction(
 out(reaction_rule(Uuid, reaction(timer(T), Action))),
 internal,
 (
 in(reaction_rule(Uuid, reaction(timer(T), Action))),
 current_time(NOW),
 CHECK is NOW + T,
 out_s(
 time(CHECK),
 internal,
 (
 out(act(Uuid, Action))
)
)
)
).

%Reazione per programmare il comportamento del TC in base alla maintenance rule
%(evento Cyclic Timer)
reaction(
 out(reaction_rule(Uuid, reaction(cyclic_timer(T), Action))),
```

```

 internal,
 (
 in(reaction_rule(Uuid, reaction(cyclic_timer(T), Action))),
 current_time(NOW),
 CHECK is NOW + T,
 out_s(
 time(CHECK),
 internal,
 (
 out(act(Uuid, Action)),
 out(reaction_rule(Uuid, reaction(cyclic_timer(T),
Action)))
)
)
)
).

%Reazione per programmare il comportamento del TC in base alla maintenance rule
%(evento Node Entrance)
reaction(
 out(reaction_rule(Uuid, reaction(node_entrance, Action))),
 internal,
 (
 in(reaction_rule(Uuid, reaction(node_entrance, Action))),
 out(act(Uuid, Action))
)
).

%Reazione triggerata all'inserimento di una tupla act(Uuid, Action) ->Esecuzione di
%una azione di mantenimento
%Action = modify_content
reaction(
 out(act(Uuid, modify_content(prolog_solve(Path, Goal)))),
 internal,
 (
 in(act(Uuid, modify_content(prolog_solve(Path, Goal)))),
 spawn(solve(Path, Goal))
)
).

```

```

%Reazione triggerata all'inserimento di una tupla
%act(Uuid, Action) -> Esecuzione di una azione di mantenimento
>Action = propagate

reaction(

 out(act(Uuid, propagate)),

 internal,

 (

 in(act(Uuid, propagate)),
 out(start_propagate(Uuid))

)

).

%Reazione triggerata all'inserimento di una tupla start_propagate
%(caso hop finito)

reaction(

 out(start_propagate(Uuid)),

 internal,

 (

 rd(tota_tuple(tota_uuid(Uuid) , Content ,
 propagation_rule(scope(distance(hop,N), Direction),
 G,
 M) ,
 Maintenance_rule)
),
 N1 is N-1,
 N1>=0,
 (
 out(to_modify(
 tota_tuple(tota_uuid(Uuid) , Content ,
 propagation_rule(
 scope(distance(hop,N1),Direction),
 G,
 M) ,
 Maintenance_rule))
),
 in(start_propagate(Uuid))
)
 ;
 (
 in(start_propagate(Uuid))
)
)

).

```

```

%Reazione triggerata all'inserimento di una tupla start_propagate
%(caso hop infinito)

reaction(

 out(start_propagate(Uuid)),

 internal,

 (

 rd(tota_tuple(tota_uuid(Uuid) , Content ,
 propagation_rule(scope(distance(hop,inf), Direction),
 G,
 M) ,
 Maintenance_rule)),

 out(to_modify(tota_tuple(tota_uuid(Uuid) , Content ,
 propagation_rule(scope(distance(hop,inf),Direction),
 G,
 M) ,
 Maintenance_rule))),

 in(start_propagate(Uuid))

)

).

%Reazione triggerata all'inserimento di una tupla to_modify
%(modifica del contenuto prima della propagazione)

reaction(

 out(to_modify(tota_tuple(Id, Content ,
 propagation_rule(scope(Distance , Direction),
 G,
 modify_content(prolog_solve(Path, Goal))
) ,
 Maintenance_rule))),

 internal,

 (
 spawn(solve(Path, Goal))
)

).

%Reazione triggerata all'inserimento di una tupla to_prop

reaction(

 out(to_prop(tota_tuple(tota_uuid(Uuid), C , P, M))),

 internal,

 (

 in(to_prop(tota_tuple(tota_uuid(Uuid), C , P , M))),

 rd(neighbourhood(List)),

```

```

 spawn(solve('tota/config/TOTA_Function_Utility.pl',
 iter_list(prop(tota_tuple(tota_uuid(Uuid), C , P , M)),
 link_op , List)))
)
).

%Reazione triggerata all'inserimento di una tupla link_op
reaction(
 out(link_op(Totatuple, neighbour(Tc, Netid, Port))),
 internal,
 (
 in(link_op(Totatuple, neighbour(Tc, Netid, Port))),
 Tc@Netid:Port ? out(Totatuple)
)
).

%Reazione triggerata all'inserimento di una tupla prop (triggera la guardia della
%propagazione quando la tupla TOTA entra in un nodo)
reaction(
 out(prop(tota_tuple(Totauuid, Content,
 propagation_rule(S,
 guard(prolog_solve(Path, Goal)),
 M),
 Maintenance))),
 (exo, completion),
 (
 spawn(solve(Path, Goal))
)
).

%Guardia success, inserimento della tupla TOTA nel nodo
reaction(
 out(guard_success(tota_tuple(tota_uuid(Uuid) , Content ,
 Propagation_rule ,
 maintenance_rule(Reaction_list))
)),
 internal,
 (
 in(guard_success(tota_tuple(tota_uuid(Uuid) , Content ,
 Propagation_rule ,
 maintenance_rule(Reaction_list))
)),
 out(tota_tuple(tota_uuid(Uuid) , Content ,
 Propagation_rule ,

```

```

 maintenance_rule(Reaction_list))),
 spawn(solve('tota/config/TOTA_Function_Utility.pl',
 iter_list(Uuid , reaction_rule , Reaction_list
)))
)
).

```

TOTA\_Function\_Utility.pl

```

iter_list(El , Functor , []).
iter_list(El , Functor , [X|Xs]) :- Term =..[Functor, El , X] , out(Term),
 iter_list(El, Functor, Xs).

```

## 8. Riferimenti bibliografici

- [1] M.Mamei,F.Zambonelli, “*Programming Stigmergic Coordination with the TOTA Middleware*”, Università di Modena e Reggio Emilia, IT