ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

IL TEMPO NEI SISTEMI COORDINATI: ESPERIMENTI IN TUCSON

Tesi in
SISTEMI DISTRIBUITI

Relatore:
Chiar.mo Prof. Ing.
ANDREA OMICINI

Correlatore:
Dott. Ing.
STEFANO MARIANI

Presentata da:
RAFFAELE MAZZA

Sessione II

Anno Accademico 2013 – 2014

A mio Padre

Indice

Introduzione

Quando si parla di sistemi computazionali si intendono sistemi che hanno assunto sempre di più un'accezione pervasiva nella nostra vita quotidiana. È necessario pensare ad un sistema distribuito come ad un sistema artificiale che ha un cuore software ma una natura sostanzialmente fisica. Quest'ultima può variare dinamicamente, con coordinabili in grado di connettersi e disconnettersi a reti, anche di grandi dimensioni, che non richiedono necessariamente un'infrastruttura cablata. Le unità computazionali del sistema sono distribuite, così come la conoscenza. Il sistema non ha più un'unità spazio-temporale; ogni dispositivo avrà la sua nozione di tempo e le interazioni con gli altri agenti non richiedono più la compresenza nello spazio o nel tempo. Non è più pensabile ordinare cronologicamente gli eventi; tipicamente succede che due eventi non sono correlabili tra di loro a priori. Il sistema deve essere in grado di reagire a situazioni impreviste in maniera autonoma, senza il controllo da parte del sistemista o dell'utente, in un ambiente impredicibile e in continua evoluzione.

Di fronte a uno scenario così complesso è necessario definire modelli temporali che siano in grado di soddisfare le esigenze di coordinazione degli agenti.

La tesi intende esplorare i problemi relativi al tempo nei sistemi coordinati e costruire esperimenti sia applicativi che di estensione di middleware di coordinazione, concentrandosi in particolare sulla tecnologia di coordinazione TuCSoN.

Capitolo 1

Sincronizzazione nei sistemi distribuiti

> "Un sistema distribuito è una collezione di computer indipendenti che agli utenti appaiono come un sistema unico coerente"[1].

In un sistema centralizzato i processi condividono lo stesso spazio di indirizzamento e utilizzano un'unico riferimento temporale che è dato dal clock di sistema. In un sistema distribuito è necessario mettere d'accordo agenti con una propria capacità computazionale e con una propria "percezione" del tempo. È necessario stabilire un criterio univoco riguardo alla successione temporale degli eventi, al fine garantire coerenza al sistema. La macchina di Turing definisce in modo formale la dimensione computazionale di un elaboratore ma non dice nulla riguardo all'interazione. Quest'ultima per poter essere modellata necessita di coordinate temporali, dal momento che sono stati introdotti, mediante l'impiego di calcolatori indipendenti, molteplici riferimenti. "La computazione globale può essere vista come un ordine totale di eventi se si considera il tempo al quale sono stati generati"[2], a patto che il riferimento temporale utilizzato sia univoco e condiviso da tutti gli agenti. Per poter ordinare cronologicamente gli eventi in un sistema distribuito, è necessario che tutti gli agenti che ne facciano parte possiedano un clock sincronizzato. Il tempo digitale all'interno di un calcolatore viene scandito mediante i seguenti riferimenti temporali:

1. *Tempo globale assoluto (Global absolute time):* è un riferimento temporale assoluto (cioè una data e un'ora reale) ed è globale, ovvero è accessibile a tutti quelli che sono interessati a conoscerlo. Questo tempo è gestito dal *BIH (Bureau International de*

[1] *Sistemi Distribuiti*, A.S. Tanenbaum M.V. Steen, Ed. Pearson 2007 *[17]*
[2] *Sincronizzazione nei Sistemi Distribuiti*, Valeria Cardellini, 2009 [3]

l'Heure) di Parigi. Il tempo coordinato universale o *UTC* viene calcolato stimando una media tra i valori temporali rilevati tramite orologi atomici. Gli algoritmi di sincronizzazione che sfruttano questo parametro temporale si appoggiano ad un *UTC service*.

2. *Tempo globale relativo (Global relative time):* In un sistema distribuito la sola cosa che conta, in definitiva, è quella di poter stabilire un'ordine cronologico degli eventi. È sufficiente utilizzare un parametro temporale che abbia una valenza globale per quel che riguarda il sistema, pur essendo relativo per il mondo esterno. All'interno dei sistemi distribuiti tale concetto può essere affrontato tramite l'algoritmo di Berkeley (vedi oltre, sezione 1.1). In buona sostanza, affinchè possa essere garantita coerenza temporale all'interno di un sistema distribuito è sufficiente sfruttare un parametro condiviso (un tempo globale calcolato dal *time daemon* nell'algoritmo di Berkeley) da tutti gli agenti, indipendentemente dal fatto che abbia o meno una correlazione con una misurazione temporale reale.

3. *Tempo logico:* è un riferimento temporale non necessariamente correlato con il concetto di tempo fisico. Esso sfrutta la relazione di precedenza tra eventi per realizzare coerenza. All'interno dei sistemi distribuiti tale concetto può essere affrontato tramite l'algoritmo di Lamport (vedi oltre, sezione 1.2).

1.1 Sincronizzazione tramite orologi fisici

Il problema della sincronizzazione degli orologi tra elaboratori differenti è di natura squisitamente fisica. Il problema era già noto a Galileo Galilei [7]. Per poter correlare misurazioni temporali, effettuate mediante strumenti di misura differenti, è necessario che il movimento del meccanismo degli orologi (ipotizzando anche che gli orologi siano identici) non alteri il loro sincronismo. Il timer del computer è solitamente un cristallo di quarzo. Quest'ultimo oscilla a una frequenza che dipende dalle caratteristiche fisiche intrinseche della materia che lo compone e dal livello di tensione che gli viene applicata. Ogni cristallo ha le sue peculiarità che lo rendono unico e pertanto non è fisicamente possibile realizzare clock di sistema identici. Questo si traduce nel fatto che due orologi tendono ad andare lentamente fuori sincronismo (*clock skew*). Per ovviare a questo inconveniente sono sta-

ti creati algoritmi di sincronizzazione che riconducono i clock locali degli elaboratori ad un'unico riferimento temporale. Essi sfruttano i seguenti meccanismi:

- *Mediante time server passivo:* Ogni agente sincronizza periodicamente il proprio clock fisico con quello di un orologio di servizio. Un esempio è dato dal protocollo *NTP,* che permette ai client di contattare un *time server*. Il ruolo del time server è passivo, si limita a comunicare il tempo (*Il tempo globale universale,* pagina 3) ai client che lo richiedano, ovvero agli agenti del sistema.

- *Mediante time server attivo (algoritmo di Berkeley):* Ogni agente sincronizza il proprio clock fisico con quello di un orologio di servizio, un demone per la precisione. Un *time daemon*, a differenza di un time server, è attivo e si occupa di richiedere periodicamente ad ogni macchina l'orario locale. In base alle risposta ottenuta calcola un tempo medio e indica a tutte le altre macchine il nuovo riferimento temporale (*Il tempo globale relativo*, Intro. Cap. 1, pag. 2).

1.2 Sincronizzazione tramite orologi logici di Lamport

Finora si è visto che la sincronizzazione in un sistema distribuito può essere realizzata mediante un'unico riferimento temporale, indipendentemente dal fatto che possa essere relativo o assoluto. È possibile spingersi oltre. Per poter realizzare coerenza temporale all'interno di un sistema distribuito è sufficiente poter ordinare cronologicamente eventi correlati, dal momento che la mancanza di sincronizzazione tra eventi non connessi logicamente tra loro non genera conflitti. Mediante l'algoritmo introdotto da Lesley Lamport [9] è possibile ordinare gli eventi all'interno del sistema sfruttando il paradigma di precedenza. Tramite espressioni del tipo $a \rightarrow b$ (*a* precede *b*) è possibile esprimere il concetto che tutti gli agenti all'interno di un sistema, siano concordi riguardo alla precedenza cronologica dell'evento *a* rispetto all'evento *b*. Ai due eventi vengono associati due riferimenti temporali, rispettivamente $C(a)$ per *a* e $C(b)$ per *b*, per cui vale la regola che se $a \rightarrow b$ (*a* precede *b*), allora $C(a) < C(b)$. Ogni processo si preoccupa di correggere il proprio clock qualora si verificasse un'incoerenza rispetto alla relazione di precedenza. I nuovi riferimenti, eventualmente scorrelati dal tempo fisico, vengono definiti come *tempo logico* dei processi, scandito mediante *orologi logici di Lamport* (figura 1.1).

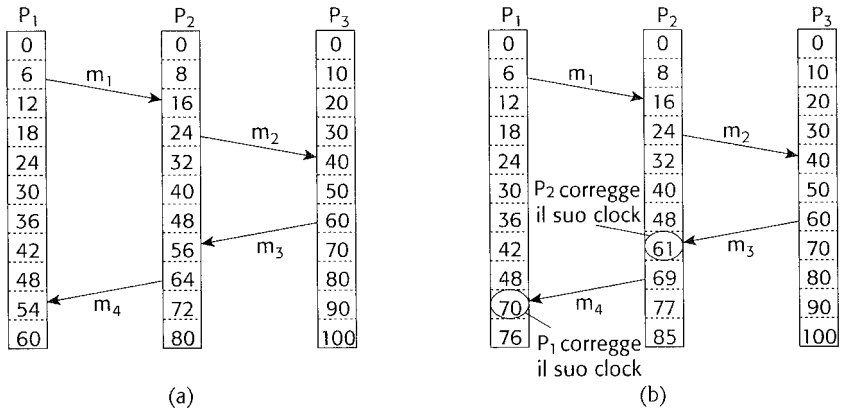

Figura 1.1: Orologi logici di Lamport [17]

1.3 Sincronizzazione tramite clock vettoriali

Con gli orologi di Lamport si ottiene che tutti gli eventi, all'interno di un sistema distribuito, siano ordinati tra loro mediante criteri di precedenza. Se $a \rightarrow b$ (*a* precede *b*), allora $C(a) < C(b)$, ma questo non permette di stabilire la precedenza tra *a* e *b* semplicemente osservando che $C(a) < C(b)$. Quindi la sincronizzazione non è ancora completa. Questo problema si può risolvere utilizzando i clock vettoriali. Se un clock vettoriale $VC(a)$ assegnato a un evento *a* è in relazione con il clock vettoriale $VC(b)$ di un evento *b* in modo che $VC(a) < VC(b)$, allora si avrà che l'evento *a* è precede l'evento *b*. Per implementare questo meccanismo i processi si scambiano tutto il vettore dei riferimenti logici (l'orologio logico locale) e non solamente il riferimento temporale relativo a due eventi correlati. La correzione di eventuali incongruenze tra tempi logici è analoga a quella che avviene negli orologi di Lamport, con la differenza che riguarda valori memorizzati all'interno di array temporali.

Nonostante sia stato affrontato in maniera organica il problema della sincronizzazione degli eventi, occorre far notare che la possibilità di tale sincronizzazione non è condizione sufficiente a poter garantire il corretto funzionamento di un sistema distribuito. La sincronizzazione si occupa solamente di ordinare cronologicamente gli eventi, ma l'interazione non consiste solamente nell'esecuzione di istruzioni distribuite. Gli eventi in un sistema distribuito hanno un significato interattivo che dipende dalla loro natura. Governare interazioni basate sul tempo e sulla natura degli eventi, al fine di realizzare un sistema coerente, è prerogativa specifica della coordinazione.

Capitolo 2

Modelli di coordinazione

Un modello di coordinazione partecipa in egual misura, insieme ad un modello computazionale, alla costituzione di un modello di programmazione completo. La parte computazionale può essere ricondotta al paradigma di programmazione strutturata, mediante la quale vengono create le singole entità computazionali. Queste ultime possono essere threads, processi o più genericamente agenti in grado di simulare un comportamento "Turing compatibile". La parte relativa al modello di coordinazione si occupa della comunicazione tra agenti, che consiste in interazioni dove vi è scambio di informazione, ovvero dove lo stato informativo del ricevente cambia. I sistemi distribuiti sono pervasivi, ovvero sono sistemi software in cui l'elaborazione dell'informazione è integrata in maniera trasparente all'interno degli agenti che li compongono. La topologia del sistema può variare dinamicamente, con coordinabili in grado di connettersi a reti, anche di grandi dimensioni, che non richiedono un'infrastruttura cablata. Il sistema deve essere in grado di reagire a situazioni impreviste in maniera autonoma, senza il controllo da parte del progettista o dell'utente, in un ambiente in continua evoluzione. Bisogna quindi accettare il fatto che l'ambiente in cui opera un sistema distribuito sia impredicibile. Ne consegue che non è possibile stabilire un'ordinamento cronologico degli eventi a priori, in quanto non è garantita, in virtù della natura stessa del sistema, la compresenza degli agenti nello spazio o nel tempo. Un modello di coordinazione si deve preoccupare di garantire lo scambio di informazione in un sistema in cui non fosse richiesta la compresenza temporale o spaziale dei coordinabili, implementando quindi la *comunicazione ortogonale* tra gli agenti. Una definizione di meta-modello di coordinazione è proposta in *[4]*:

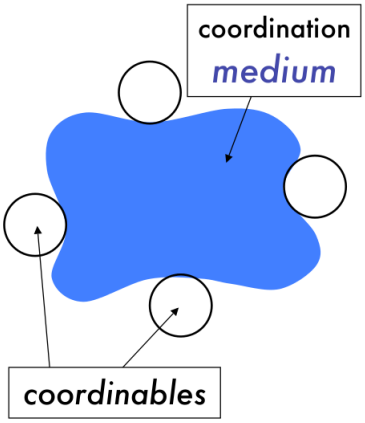

Figura 2.1: Meta-modello di coordinazione [12]

> "Un modello di coordinazione fornisce un framework concettuale in cui le interazioni tra entità attive ed indipendenti chiamate agenti possono essere espresse. Un modello di coordinazione dovrebbe occuparsi dei problemi inerenti alla distruzione e creazione di agenti, della comunicazione tra agenti, della distribuzione spaziale degli agenti, così come della sincronizzazione e distribuzione delle loro azioni nel tempo."

Il Professor Ciancarini individua le componenti di un meta-modello di coordinazione secondo le seguenti classi (figura 2.1):

- ***entità di coordinazione:*** entità la cui mutua interazione è regolata dal modello, dette anche coordinabili.

- ***media di coordinazione:*** astrazioni che permettono e regolano le interazioni tra agenti. Esse mediano la comunicazione, ovvero fanno in modo che la comunicazione tra agenti sia possibile. Inoltre un media di coordinazione può fornire servizi ad agenti aggregati che devono essere manipolati nel loro complesso.

- ***leggi di coordinazione:*** leggi che definiscono il comportamento dei media di coordinazione in risposta alle interazioni. Un modello di coordinazione definisce un'insieme di regole che descrivono il modo in cui si coordinano gli agenti sui media di coordinazione attraverso primitive di coordinazione.

2.1 Classi di coordinazione

I modelli di coordinazione possono essere suddivisi in due classi principali, a seconda di cosa si voglia coordinare, i dati o i componenti [16]:

1. *modelli di coordinazione Control-Driven/Control-Oriented:* I modelli di coordinamento appartenenti a questa categoria si sviluppano attorno al concetto di cambiamento di stato di un processo legato alla comunicazione di eventi. Sono caratterizzati da coordinatori (astrazioni per la coordinazione) che si occupano di manipolare i dati e di modificare la topologia della comunicazione. I processi (sia quelli di coordinamento che quelli computazionali) vengono trattati come *black-box* e comunicano con il loro ambiente tramite interfacce ben definite (porte di *input* o di *output*, a seconda che consumino o producano informazione). La comunicazione è di tipo *stateful* e *punto-punto*. I processi si coordinano comunicando al proprio ambiente di sviluppo eventuali cambiamenti di stato interno oltre a quelli di tipo informativo.

2. *modelli di coordinazione Data-Driven/Data-Oriented:* I modelli di coordinamento appartenenti a questa categoria si sviluppano attorno all'informazione che viene scambiata nella comunicazione. Quest'ultima è di tipo stateful ed avviene mediante l'astrazione di memoria condivisa (*dataspace*). I processi comunicano indirettamente, producendo e consumando informazione sul dataspace. La coordinazione avviene mediante la manipolazione dei dati condivisi. I modelli che appartengono a questa categoria sono quelli di tipo tuple-based (Linda e derivati).

2.2 Coordinazione tramite scambio di tuple

Il modello di coordinazione tramite tuple è stato quello che ha avuto maggior successo nel tempo. Esso appartiene ai modelli che sfruttano lo scambio di informazione mediante memoria condivisa. Per definire tale modello si è pensato di utilizzare una strategia di tipo bottom-up; ciò si traduce nell'elencarne le componenti, riconducendole alla classificazione del meta-modello del Professor Ciancarini (vedi Intro. Cap. 2), per poi descriverne il comportamento:

- ***processi, threads, oggetti concorrenti, utenti fisici:*** corrispondono ai coordinabili, la cui mutua interazione è regolata dal modello.

- ***spazio di tuple**:* corrisponde al medium di coordinazione, ovvero all'astrazione di spazio di indirizzamento condiviso. Corrisponde al luogo dove avviene la comunicazione tra coordinabili attraverso la produzione e il consumo di *tuple*.
- ***tuple e primitive Linda:*** definiscono il linguaggio di coordinazione, che modella il comportamento del medium di coordinazione in risposta agli eventi interattivi. Le prime si occupano della parte comunicativa del linguaggio, che consiste nella sintassi utilizzata per esprimere e modificare le strutture dati; le seconde della coordinazione, che consiste in un set di primitive di interazione e relativa semantica [5].

Lo spazio di tuple corrisponde all'astrazione di medium di coordinazione. Analogamente ad una blackboard ad accesso associativo permette ai coordinabili, mediando il contenuto informativo, di sincronizzarsi e cooperare tramite pattern di tipo *producer-consumer*. Le tuple, collezioni ordinate di possibili dati eterogenei, costituiscono il linguaggio con cui i processi interagiscono. Lo spazio delle tuple può essere acceduto mediante meccanismi di *tipo associativo*, con cui è possibile manipolarne il contenuto informativo.

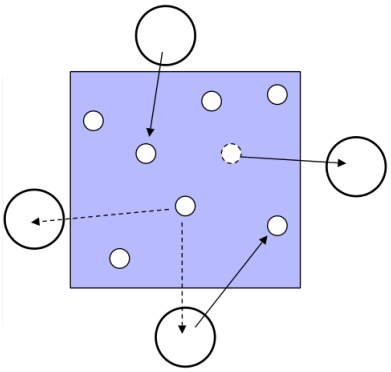

Figura 2.2: Rappresentazione dello spazio di tuple [12]

2.3 I modelli di coordinazione Linda

Il modello *Linda* è considerato l'antesignano del paradigma di coordinazione di tipo *space-based ad accesso associativo*, pertanto si considereranno tutti i modelli riconducibili ad esso come facenti parte dell'omonima classe di coordinamento. Linda sfrutta uno spazio di memoria condiviso, lo spazio di tuple, mediante il quale i processi possono comunicare tra loro. Lo scambio di informazione avviene tramite tuple, collezioni ordinate di possibili

informazioni eterogenee, che vengono prodotte e consumate dagli agenti all'interno del medium di coordinazione. Le tuple generate dai coordinabili, all'interno dello spazio di tuple, hanno un'esistenza che non dipende da chi li ha generati e sono accessibili a tutti i processi. Questa caratteristica viene definita con il termine di *comunicazione generativa* [8]. Quest'ultima è disaccoppiata nello spazio e nel tempo rispetto ai coordinabili, dal momento che avviene solamente tramite memoria condivisa. Non è necessario che i processi si conoscano tra loro, il fulcro del sistema è il contenuto informativo. Il disaccoppiamento secondo le coordinate spaziali, temporali e di naming definiscono il paradigma di *comunicazione ortogonale* [8]. Le tuple vengono manipolate mediante *accesso associativo* [8] con l'ausilio di primitive, definite a priori in Linda, che utilizzano meccanismi di matching attraverso dei *tuple template* [8]. Per ogni istanza di tupla che è compatibile con il tuple template viene fatto un confronto campo a campo. Due campi sono corrispondenti se entrambi hanno una copia dello stesso riferimento o se il campo del tupla template è nullo. L'update dello spazio di tuple avviene in maniera atomica. In Linda sono definite primitive preposte allo scambio di tuple. È possibile realizzare semplici protocolli di coordinazione tramite il loro impiego sfruttando le proprietà bloccanti di alcune di esse (*semantica sospensiva*) [8]:

- `out(T)`: inserisce una tupla T nello spazio di tuple;
- `in(TT)`: preleva la tupla dallo spazio di tuple compatibile con il tuple template. L'operazione è distruttiva nel senso che ogni tupla prelevata viene rimossa dal medium di coordinazione (*destructive reading*). Se non sono disponibili tuple compatibili con il tuple template il processo che ha invocato l'operazione viene sospeso fino a quando non viene inserita la giusta tupla nello spazio di tuple (semantica sospensiva). Qualora fossero presenti più tuple compatibili ne viene scelta una secondo un criterio *non deterministico*;
- `rd(TT)`: analoga alla `in(TT)` con la differenza che preleva una copia della tupla compatibile lasciando inalterato lo spazio di tuple (*no destructive reading*).

Le primitive appena descritte si preoccupano di manipolare una tupla alla volta. Al fine di migliorare l'espressività del modello in funzione delle esigenze relative alle problematiche presenti nei sistemi distribuiti, si è deciso di estendere il set di istruzioni native con primitive di tipo predicativo (`inp(TT)`, `rdp(TT)`) e di tipo bulk (`in_all(TT)`,

`rd_all(TT)`). Le prime due sono analoghe rispettivamente alla `in(TT)` e alla `rd(TT)`, con la differenza che non sono bloccanti. Le seconde restituiscono un insieme di tuple compatibili con quella del proprio template. Queste operazioni sono descritte qui di seguito:

- `inp(TT)`: preleva la tupla dallo spazio di tuple compatibile con il tuple template. L'operazione è distruttiva nel senso che ogni tupla prelevata viene rimossa dal medium di coordinazione (destructive reading). Se non sono disponibili tuple compatibili con il tuple template viene restituito al processo che ha invocato l'operazione un messaggio di fallimento dell'operazione (*semantica success/failure*). Qualora fossero presenti più tuple compatibili ne viene scelta una casualmente;

- `rdp(TT)`: analoga alla `inp(TT)` con la differenza che preleva copie di tuple compatibili lasciando inalterato lo spazio di tuple (no destructive reading);

- `in_all(TT)`: preleva tutte le tuple dallo spazio delle tuple compatibili con il tuple template. L'operazione è distruttiva nel senso che tutte le tuple prelevate vengono rimosse dal medium di coordinazione (destructive reading). Se non sono disponibili tuple compatibili con il tuple template viene restituito al processo che ha invocato l'operazione una collezione vuota;

- `rd_all(TT)`: analoga alla `rd_all(TT)` con la differenza che preleva copie di tuple compatibili lasciando inalterato lo spazio di tuple (no destructive reading).

Nonostante il successo di questo modello di coordinazione risultano evidenti i limiti di cui è affetto. In primo luogo il fatto che il comportamento del medium di coordinazione è limitato alle istruzioni e non è riconfigurabile nel tempo. Questo significa che la coordinazione di processi funziona bene nella misura in cui il loro comportamento si adatta alle regole di coordinazione prestabilite. Inoltre le primitive sono in grado di manipolare solamente una tupla alla volta e le estensioni di tipo bulk non sono *general purpose* ma sono specifiche per ogni singolo problema. Da ciò ne consegue che l'onere della coordinazione è totalmente a carico dei coordinabili.

2.4 Coordinazione tramite centri di tuple

I modelli di coordinazione Linda-like costituiscono una classe di modelli di coordinamento ibrida. Sfruttano un approccio di tipo data-driven per quel che riguarda l'interazione tra agenti e centri di tuple; inoltre offrono la possibilità di implementare meccanismi di tipo control-driven come le reaction, in modo da rendere il sistema più potente e flessibile. Per definire tale modello si è pensato di utilizzare la stessa strategia di tipo bottom-up utilizzata precedentemente per i modelli basati su scambio di tuple (vedi sez. 2.2); si andranno pertanto ad elencarne le componenti, riconducendole alla classificazione del meta-modello del Professor Ciancarini (vedi Intro. Cap. 2), per poi descriverne il comportamento:

- ***Gli agenti:*** corrispondono alle entità di coordinazione la cui mutua interazione è governata dal modello. Essi sono distribuiti nella rete, hanno una loro capacità computazionale autonoma, sono mobili indipendentemente dal dispositivo da cui vengono eseguiti, e hanno un comportamento pro-attivo;

- ***I centri di tuple:*** corrispondono al medium di coordinazione ovvero a una molteplicità di astrazioni di tipo tuple-based indipendenti distribuite nel sistema. Essi incorporano in se due caratteristiche:

  - analogamente agli spazi di tuple forniscono uno spazio di indirizzamento condiviso per la comunicazione basata sullo scambio di tuple;
  - forniscono uno spazio di comportamento programmabile per la coordinazione tramite tuple.

  I centri di tuple vengono raggruppati in nodi che insieme agli agenti costituiscono l'astrazione topologica del sistema;

- ***Tuple Linda:*** definiscono il linguaggio di comunicazione, che consiste nella sintassi utilizzata per esprimere e modificare le strutture dati;

- ***primitive Linda e reaction programmabili***: definiscono il linguaggio di coordinazione, le prime consistono in un set di primitive di interazione e relativa semantica che definiscono il comportamento pro-attivo degli agenti e servono a questi ultimi per

interfacciarsi ai centri di tuple al fine di manipolare i dati condivisi (*layer data/driven*); le seconde offrono la possibilità di poter implementare comportamenti all'interno del centro di tuple in risposta a cambiamenti di stato causati dalle interazioni tra/con gli agenti, definendo il comportamento reattivo del medium di coordinazione (*layer control/driven*).

La creazione di spazi di tuple programmabili, detti centri di tuple, deriva dalla necessità di poter adattare il comportamento del medium di coordinazione a specifici applicativi, senza modificare il modo in cui gli agenti si interfacciano allo spazio condiviso. Si desidera incapsulare le politiche di coordinazione all'interno dello spazio di tuple spostando l'onere della coordinazione dagli agenti all'astrazione di memoria condivisa. Il nuovo modello dovrà essere definito da caratteristiche tipiche di un modello control-driven affiancate a quelle native Linda di tipo information-driven. Si definisce pertanto un centro di tuple come "uno spazio di tuple con specifiche di comportamento che definiscono le reazioni in risposta agli eventi interativi" [12], dove le specifiche di comportamento vengono definite mediante un *reaction specification language*. Quest'ultimo definisce potenzialmente il layer Control-Driven del sistema ibrido, costituito da attività computazionali dette reaction eseguite in risposta agli eventi locali al centro di tuple. Ogni reazione può:

- accedere al centro di tuple e modificarne lo stato, aggiungendo o rimuovendo tuple
- accedere alle informazioni relative all'evento scatenante rendendolo completamente osservabile
- invocare primitive di collegamento su altri centri di tuple

L'update del centro di tuple avviene atomicamente, una reaction alla volta. Ne consegue che, qualora anche solo un'operazione fra quelle scatenate dalla reazioni non vada a buon fine, il centro di tuple viene riportato alle condizioni originali (*semantica success/failure transazionale*). Il tutto è stato implementato in maniera trasparente ai coordinabili, che si interfacciano al centro di tuple tramite primitive Linda come se fosse uno spazio di indirizzamento non reattivo.

2.5 I modelli di coordinazione Linda-like

Esistono diversi modelli di coordinazione che adottano il paradigma dei centri di tuple. Essi sono generalmente conosciuti come modelli di coordinazione Linda-like. Essi presentano diverse analogie con i modelli di tipo blackboard-based nonostante questi ultimi, accoppiando spazialmente i processi, non ne contemplino attività di tipo indipendente; gli agenti sono modellati in termini di operazioni protette scatenate in risposta ai cambiamenti di stato nello spazio di tuple.

LuCe (*Logic Tuple Centres*) e TuCSoN (*Tuple Centres Spread over the Network*) sono modelli di coordinazione multiagente che utilizzano ReSpecT come linguaggio di programmazione per i propri centri di tuple, ma differiscono nel modo in cui gli agenti sono abilitati ad accedere allo spazio di coordinazione. Nel primo modello come ad un singolo servizio internet globale mentre nel secondo come a una moltitudine di servizi internet distribuiti.

Il modello di coordinazione Mars (*Mobile Agent Reactive Spaces*) offre un singolo spazio di indirizzamento reattivo programmato in Java dove le tuple sono oggetti (*entry*) la cui classe implementa l'interfaccia *Entry*, mentre le reazioni sono metodi Java. I meccanismi di matching di Mars sono quelli di JavaSpace (vedi oltre sez. 3.1.1) così come le primitive e ricalcano in tutto e per tutto quelle del modello Linda (`read()` = `rd()`, `write()` = `out()` e `take()` = `in()`) [19].

Il modello di coordinazione LGL (*Law-governed Linda*) sfrutta un linguaggio di programmazione dei propri centri di tuple che è di tipo logic-based (come ReSpecT in TuCSoN). Esso si differenzia dagli altri in quanto implementa il livello control-driven mediante astrazioni di coordinazione (*controller*) piuttosto che attraverso spazi di tuple reattivi. ReSpecT conserva la semantica delle primitive Linda intatta, aggiungendo incrementalmente al comportamento delle primitive quello delle reazioni (*composizione sequenziale*), mentre LGL rende possibile ridefinire completamente l'effetto comunicativo di una primitiva (*composizione interattiva*) [11]. TuCSoN sfrutta un'astrazione di coordinamento globale, incorporando le reaction nei centri di tuple indipendentemente dalla locazione dell'agente nel sistema, mentre LGL associa proxy localmente agli agenti, spostando la programmazione dal medium di coordinazione ai coordinabili, affinchè intercettino le interazioni informative modificandone possibilmente la semantica [13]. Il modello di coordinazione TOTA (*Tuples On The Air*) si differenzia dagli altri modelli in quanto le

tuple non vengono associate ad uno spazio condiviso univoco. Esse vengono “iniettate” in uno specifico nodo della rete e si propagano autonomamente secondo un determinato pattern. I nodi TOTA sono in grado di memorizzare localmente le tuple e di permetterne la loro diffusione. Le tuple TOTA comprendono al loro interno sia il contenuto informativo che le regole di propagazione. Queste ultime definiscono sia la strategia di distribuzione che quella di variazione del contenuto informativo delle tuple durante la diffusione [10].

Capitolo 3

Il tempo nei modelli di coordinazione

Il metamodello di Ciancarini descritto precedentemente (vedi Intro. Cap. 2), individua tre classi di componenti in un sistema distribuito. Si può pensare di individuare eventuali implementazioni di modelli temporali attraverso l'analisi degli aspetti implementativi delle componenti. In particolare, a seconda della prospettiva scelta, un modello temporale dovrà essere in grado di soddisfare i seguenti requisiti:

- Dal punto di vista del medium di coordinazione, l'utilizzo del tempo è necessario a definire l'esistenza e la qualità di un servizio, come ad esempio definire la frequenza minima o massima con cui un processo può interagire con lo spazio di memoria condiviso.

- Dal punto di vista dei coordinabili, l'utilizzo del tempo è necessario a definire l'interazione con il medium di coordinazione, come ad esempio vincolare un processo ad eseguire un determinato compito entro un tempo prestabilito, o vincolare il tempo di risposta del medium di coordinazione entro un determinato istante.

- Dal punto di vista del linguaggio di comunicazione, l'utilizzo del tempo è necessario a limitare la permanenza dell'informazione sul medium di coordinazione, come ad esempio definire il tempo di vita dell'informazione nel medium di coordinazione.

Si analizzeranno in questo capitolo le due grandi famiglie di modelli di coordinazione descritte nel capitolo precedente, ovvero quella del capostipite Linda (scambio di tuple) e quella dei suoi discendenti Linda-like (centri di tuple), al fine di stabilire se esi-

ste la possibilità di supportare un'infrastuttura temporale, distribuita sugli elementi che le compongono.

3.1 Modello del tempo legato alla tupla

3.1.1 Il tempo legato alle tuple in Linda

Il modello di coordinazione Linda non si preoccupa di circoscrivere in intervalli temporali definiti la presenza di una tupla nel medium di coordinazione. Il paradigma della comunicazione generativa [8] stabilisce che l'esistenza di una tupla è indipendente dal processo che l'ha generata. La vita di una tupla si svolge secondo il pattern producer-consumer (Vedi Sez. 2.2) e può essere descritta in tre punti:

1. nasce nel momento in cui viene depositata da un processo nello spazio di indirizzamento condiviso (tramite primitiva di tipo `out()`);

2. permane temporalmente senza soluzione di continuità nel medium di coordinazione, indipendentemente dal fatto che il coordinabile generatore sopravviva o meno ad essa;

3. muore solamente nel momento in cui viene rimossa da un processo (tramite primitiva di tipo `in()`). Questo significa che non invecchia e che può potenzialmente avere vita eterna.

Esistono modelli di coordinazione appartenenti alla famiglia Linda, che prevedono la possibilità di poter assegnare una scadenza alla tupla; la rimozione dallo spazio di indirizzamento condiviso avviene automaticamente oltrepassato il limite temporale.

In *JavaSpace*, modello di coordinazione java-based, è possibile limitare la permanenza delle *entry* (istanze di tuple appartenenti alla classe *Entry*) nel repository remoto attraverso uno stile di programmazione chiamato leasing. Esso consiste nell'utilizzo di una risorsa per un periodo limitato di tempo, negoziato fra chi richiede la risorsa e chi la offre. L'implementazione viene effettuata definendo un tempo di lease assegnato alla tupla mediante la primitiva di inserimento [2]:

```
public Lease write(Entry e, Transaction txn, long lease)
```

L'operazione `write()` inserisce una copia della entry specificata all'interno del medium di coordinazione. La `write()` viene invocata passando come parametri un oggetto di tipo Entry, un oggetto di tipo *Transaction* (è diverso da `NULL` qualora la entry faccia parte di una transazione[1]) e il tempo di lease richiesto (espresso in millisecondi). Come è possibile notare dalla signature, la `write()` ritorna un oggetto Lease che esprime l'intervallo di tempo, espresso in millisecondi, entro il quale la entry verrà mantenuta all'interno del medium di coordinazione[2].

3.1.2 Il tempo legato alla tupla nei modelli Linda-like

I modelli di coordinazione Linda-like, similmente a quelli di classe Linda, non sempre si preoccupano di circoscrivere, in intervalli temporali definiti, la presenza di una tupla nel medium di coordinazione.

Una tupla in LGL vive per un periodo di tempo indefinito nella memoria condivisa; la sua nascita e la sua morte vengono decise in ultima istanza dai controller, che applicano le leggi di coordinazione del sistema ai processi [11].

Le tuple TOTA sono distribuite spazialmente nei nodi TOTA che definiscono la topologia del medium di coordinazione; la loro struttura è definita come segue:

$$T = (\mathcal{C}, \mathcal{P}, \mathcal{M})$$

dove $\mathcal{C}$ è il contenuto informativo, $\mathcal{P}$ sono le regole di propagazione ed $\mathcal{M}$ le regole di mantenimento. È possibile affermare che al suo interno vi siano tutte le componenti necessarie a definire il modello di coordinazione. Si è ritenuto pertanto utile riportare tale struttura, all'interno di tutte le sezioni di questo capitolo, considerando opportunamente i parametri relativi alla classificazione scelta. In questa sezione è stato inserito il parametro relativo al contenuto informativo, che corrisponde propriamente al linguaggio di comunicazione nel modello di Ciancarini e può essere analogamente accostato a una tupla Linda. Secondo la documentazione disponibile [10] non è previsto l'inserimento di un

[1] Una transazione è un insieme di istruzioni che verranno eseguite atomicamente, o tutte o nessuna.

[2] L'effettivo tempo di lease dipende anche da ciò che il medium di coordinazione è in grado di supportare. Il tempo di lease restituito dall'operazione `write()` dipende quindi dalla richiesta e dal medium.

riferimento temporale in $\mathcal{C}$ al fine di limitare la presenza delle tuple TOTA nel medium di coordinazione.

Il modello Mars è stato creato sulle specifiche di JavaSpace, pertanto ne eredita i meccanismi di leasing (Vedi Sez. 3.1.1).

3.2 Modello del tempo legato all'operazione

3.2.1 Il tempo legato alle operazioni in Linda

Linda non offre un controllo temporale esplicito sulle operazioni in quanto il disaccoppiamento temporale delle interazioni tra gli agenti non prevede una gestione centralizzata degli eventi. Le primitive Linda sono generalmente poco flessibili in termini di controllo delle interazioni e possono essere utilizzate solamente al fine di implementare semplici protocolli di sincronizzazione.

Il processo che inserisce una tupla nel medium di coordinazione tramite `out()` deve terminare tale operazione prima che il processo richiedente la medesima tupla esegua la `in()`, altrimenti il processo che ha effettuato la richiesta resta in attesa. Variando l'ordine cronologico di esecuzione delle primitive `in()` e `out()` varia il comportamento dei coordinabili, poiché chi esegue la `in()` deve attendere l'esecuzione della `out()`. La `in()` non possiede un meccanismo per impostare il periodo di attesa di una determinata tupla da parte di un coordinabile. Questo si traduce in un'attesa indefinita, potenzialmente infinita da parte di un processo, qualora la tupla non fosse presente nel medium di coordinazione. L'operazione `rd()` ha un comportamento analogo alla `in()`, con l'unica differenza sostanziale che, una volta prelevata la tupla compatibile con il template, non viene modificato il contenuto dello spazio delle tuple. La `out(T)` è istantanea o al più influenzata dalle latenze del medium di coordinazione. Le estensioni al set di istruzioni native (primitive predicative e bulk) sono tutte istantanee: `inp()` e `rdp()` prelevano/leggono l'eventuale tupla compatibile, restituendo un messaggio di errore in caso non fossero presenti tuple compatibili nel medium di coordinazione; `in_all(TT)` e `rd_all(TT)` prelevano/leggono le eventuali tuple compatibili, restituendo una collezione vuota in caso non fossero presenti tuple compatibili nel medium di coordinazione. Dalle considerazioni appena fatte si deduce che la responsabilità di una eventuale coordinazione temporale degli agenti in Linda, per quel che riguarda le operazioni, non è stata implementata.

JavaSpace al contrario, pur appartenendo alla famiglia Linda, prevede la possibilità di poter assegnare una scadenza alle operazioni di coordinazione limitandole entro un certo intervallo temporale. L'implementazione viene effettuata definendo un *timeout* assegnato all'operazione mediante le primitive bloccanti e predicative:

- `public Entry take(Entry tmpl, Transaction txn, long timeout)`

  corrisponde alla `in()` di Linda. Il parametro timeout serve a limitare il tempo massimo di attesa da parte di un processo per una tupla compatibile al proprio template (espresso in millisecondi). L'operazione può restituire la entry rimossa dal JavaSpace (qualora fosse disponibile prima del timeout) oppure `NULL`;

- `public Entry read(Entry tmpl, Transaction txn, long timeout)`

  corrisponde alla `rd()` di Linda. È uguale a `take()` con la differenza che preleva una copia della tupla lasciando inalterato il medium di coordinazione (*no destructive reading*).

- `public Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)`

  corrisponde alla `inp()` di Linda. Il parametro timeout in questo caso serve a ritardare nel tempo la risposta dell'operazione al processo (che altrimenti sarebbe istantanea). L'operazione può restituire la entry rimossa dal JavaSpace (qualora fosse disponibile prima del timeout) oppure `NULL`;

- `public Entry readIfExists(Entry tmpl, Transaction txn, long timeout)`

  corrisponde alla `rdp()` di Linda. È uguale a `takeIfExists()` con la differenza che preleva una copia della tupla lasciando inalterato il medium di coordinazione (*no destructive reading*).

3.2.2 Il tempo legato alle operazioni nei modelli Linda-like

I modelli di coordinazione Linda-like, pur prevedento una gestione centralizzata degli eventi, generalmente relegano questa funzionalità al medium di coordinazione, non preoccupandosi di circoscrivere, in intervalli temporali definiti, le richieste fatte dai processi tramite operazioni primitive.

LGL prevede una gestione temporale delle operazioni in quanto i controller hanno a disposizione una variabile `clock` che rappresenta il tempo locale (al controller). È possibile, ad esempio, che un controller impartisca una direttiva ad un agente, direttiva da eseguire dopo aver atteso un tempo *t*. A quel punto il controller memorizza la direttiva, attende il tempo *t* (utilizzando il suo `clock` locale) e poi la esegue sul coordinabile [11].

In TOTA si è scelto arbitrariamente di associare le computazioni contenute nel parametro $\mathcal{P}$ della tupla (vedi sez. 3.1.2) al linguaggio di coordinazione nel modello di Ciancarini, ovvero alle operazioni in Linda. Dalla documentazione a disposizione [10] si evince che non è possibile inserire un riferimento temporale all'interno di questo tipo di elaborazioni, pur essendo possibile scatenarle in determinati istanti e limitarne l'effetto in intervalli temporali definiti (tramite il parametro $\mathcal{M}$, vedi sez. 3.3.2).

Mars si distingue dagli altri modelli in quanto è stato creato sulle specifiche di JavaSpace e ne condivide pertanto gli stessi meccanismi di timeout (vedi sez. 3.2.1). Inoltre quest'ultimo estende le primitive JavaSpace implementando due funzioni analoghe alle primitive bulk di Linda (vedi sez. 2.3) dotate di timeout [2]:

- `Entry[] takeAll(Entry tmpl, Transaction txn, long timeout)` estrae tutte le tuple corrispondenti al tuple template.
- `Entry[] readAll(Entry tmpl, Transaction txn, long timeout)` legge tutte le tuple corrispondenti al tuple template.

3.3 Modello del tempo legato al medium di coordinazione

3.3.1 Il tempo legato al medium di coordinazione in Linda

Nei modelli di coordinazione di classe Linda non è previsto impostare comportamenti in reazione ad eventi temporali, in quanto non implementano un layer control-driven. Le architetture di classe data-driven vengono dette deliberative e prevedono che il medium di coordinazione interagisca con i coordinabili mediante un comportamento proattivo, ovvero mediante decisioni basate sulla logica mediate da pattern matching e manipolazioni simboliche. Le architetture ibride implementano sia il layer data-driven che quello control-driven

e pertanto incorporano caratteristiche appartenenti sia alle architetture deliberative che a quelle reattive, ovvero:

- *il sottosistema deliberativo* contiene una rappresentazione simbolica della realtà in cui è immerso, sviluppa stategie e prende decisioni
- *il sottosistema reattivo* reagisce rapidamente agli eventi che avvengono, senza dover fare ragionamenti complessi

Un modello di coordinazione temporale può essere implementato, all'interno del medium di coordinazione, solamente nelle architetture che possiedono un sottosistema reattivo, peculiarità dei modelli di coordinazione basati su centri di tuple.

3.3.2 Il tempo legato al medium di coordinazione nei modelli Linda-like

Nei modelli di coordinazione di classe Linda-like non sempre è previsto impostare comportamenti in reazione ad eventi temporali in quanto tale implementazione non viene sempre fatta a livello del layer control-driven.

LGL implementa il proprio layer control-driven all'interno dei controller[3]. I controller hanno un'idea di tempo attraverso la variabile locale `clock` di cui abbiamo scritto in precedenza (vedi 3.2.2). Rimane comunque un'implementazione parziale dato che `clock` è per l'appunto una variabile locale, non condivisa a livello di sistema [11].

In TOTA si è scelto arbitrariamente di associare le operazioni contenute nel parametro $\mathcal{M}$ della tupla (vedi sez. 3.1.2) al medium di coordinazione nel modello di Ciancarini, ovvero al ruolo della programmazione dei centri di tuple nei modelli Linda-like. Dalla documentazione a disposizione [10] si evince che, tramite questo tipo di computazioni, è possibile applicare comportamenti relativi alla permanenza e alle regole di diffusione nel tempo delle tuple sul medium di coordinazione. In particolare sono citate le impostazioni di un lease per le tuple e di un trigger periodico per scatenare eventi.

[3] I controller sono stati collocati, al fine di poter unificare in un'unica sezione la trattazione dei modelli temporali nei centri di tuple, nel medium di coordinazione.

Mars prevede un layer control-driven implementato tramite reaction programmabili a livello temporale; esso implementa il supporto sia per reazioni istantanee che per reazioni temporizzate nel medium di coordinazione [2]. Quando viene scatenata una reazione vengono invocati nell'ordine il metodo `InitAndRun()` della reazione e viene istanziato un nuovo thread (che può essere istantaneo o lanciato dopo un determinato intervallo temporale) che esegue il metodo `run()` della reazione stessa. È possibile definire in Mars anche l'istante in cui la reazione viene invocata, mediante tre tipi di reazioni:

- `pre()`: l'esecuzione di queste reazioni avviene in sostituzione delle operazioni che si sarebbero dovute svolgere. Vengono invocate prima di accedere al medium di coordinazione;
- `infra()`: l'esecuzione di queste reazioni avviene subito dopo l'accesso allo spazio di tuple. Permettono di ridefinire il valore restituito dall'operazione dopo la sua esecuzione;
- `post()`: l'esecuzione di queste reazioni avviene dopo che l'operazione sullo spazio di tuple è conclusa. Non modificano il valore restituito dalle operazioni sul medium di coordinazione.

Capitolo 4

Il tempo in TuCSoN e ReSpecT

4.1 Il modello di coordinazione TuCSoN

TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork)[1] appartiene alla famiglia dei modelli coordinazione Linda-like (vedi sez. 2.5). Coerentemente a quanto fatto precedentemente, si è scelto di introdurre il modello tramite le sue componenti, secondo la classificazione del meta-modello del Professor Ciancarini (vedi Intro. Cap. 2), per poi descriverne il funzionamento:

- ***gli agenti*** TuCSoN***:*** corrispondono alle entità di coordinazione la cui mutua interazione è governata dal modello. Ogni agente viene identificato mediante un qualsiasi nome `aname` (termine ammissibile a livello di logica di prim'ordine Prolog senza variabili) [18] e da un `UUID` (*Universally Unique Identifier*)[2] assegnatogli univocamente dal middleware in modo da contraddistinguere ogni agente del sistema;

- ***i centri di tuple*** ReSpecT***:*** corrispondono al medium di coordinazione e, contrariamente agli agenti, la loro mobilità è vincolata al dispositivo che li ospita. Essi sono caratterizzati dalle seguenti proprietà:

[1] http://apice.unibo.it/xwiki/bin/view/TuCSoN/

[2] `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`

- *inspectability:* i centri di tuple possono essere ispezionabili dagli agenti per mezzo di alcune primitive, come ad esempio la `rd()` e la `rd_s`(), per conoscere sia lo stato informativo che di specifica del centro di tuple;
- *malleability:* è possibile modificare dinamicamente nel tempo le specifiche di comportamento (tramite le primitive `in_s()` e `out_s()`) e il contenuto informativo (tramite le primitive `in()` e `out()`) dei centri di tuple;
- *linkability:* è possibile invocare tramite reazioni delle primitive su altri centri di tuple, anche non appartenenti allo stesso nodo.

Un centro di tuple ReSpecT viene identificato in maniera univoca da `tname@netid:port`, dove `tname` è il nome del centro di tuple (termine ammissibile a livello di logica di prim'ordine Prolog senza variabili) [18], mentre la coppia `netid:port` indica il nodo di appartenenza.

I centri di tuple vengono raggruppati in *nodi* TuCSoN che insieme agli agenti costituiscono l'astrazione topologica del sistema. un nodo viene identificato in maniera univoca da `netid:port`, dove `netid` indica l'indirizzo IP o l'entry DNS del dispositivo che ospita il nodo, mentre port è il numero di porta su cui il nodo è in ascolto. Ogni device può ospitare al suo interno diversi nodi;

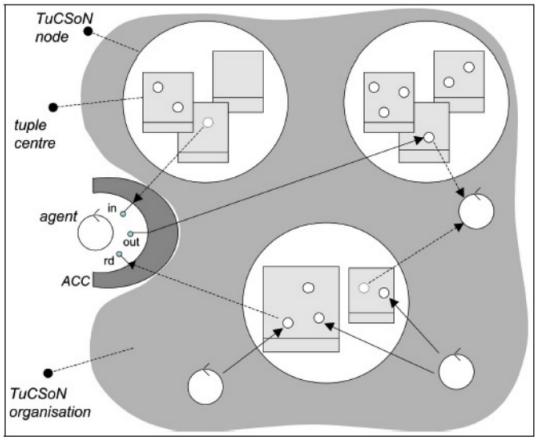

Figura 4.1: architettura di un nodo TuCSoN [14]

- ***tuple Linda:*** definiscono il linguaggio di comunicazione;

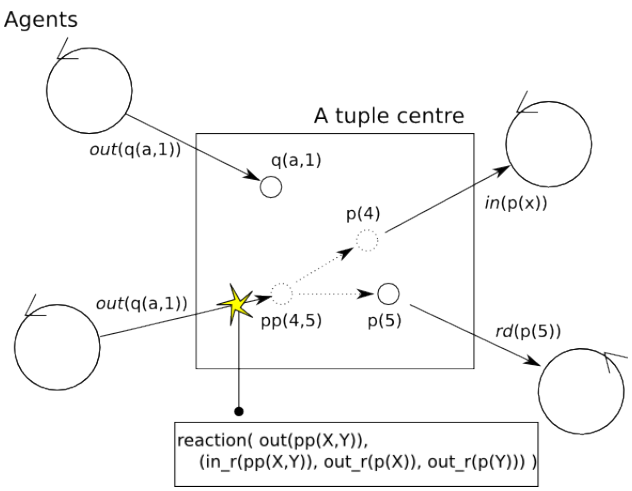

Figura 4.2: coordinazione tra agenti mediante un centro di tuple ReSpecT

- ***primitive Linda e reaction programmabili***: definiscono il linguaggio di coordinazione, le prime per quel che riguarda il layer data-driven, le seconde per quel che riguarda il layer control-driven (figura 4.2). Le primitive Linda vengono utilizzate per interfacciarsi al centro di tuple ReSpecT implementando il comportamento proattivo degli agenti che si concretizza nello scambio di tuple attraverso il medium di coordinazione. Le reaction, programmate tramite ReSpecT (**Re**action **Spec**ification **T**uples) [1] mediante la sintassi di tuProlog [18], implementando il comportamento reattivo del centro di tuple che si concretizza nelle reazioni ad eventi scatenanti. Esse sono espresse nella forma

  ```
  reaction(E,G,R)
  ```

  dove con `E` si vuole indicare un descrittore di un evento, `G` la guardia ovvero l'insieme di condizioni che devono essere soddisfatte affinchè la reazione possa essere eseguita ed `R` la lista di operazioni da eseguire non appena accade un'evento compatibile ad `E`. Non appena si verifica un evento `Ev` compatibile con `E` alle condizioni di `G`, vengono eseguiti tutti i *reaction goals* contenuti in `R` [1].

Il modello TuCSoN e può essere considerato a tutti gli effetti un'estensione di Linda. L'idea di fondo che sta alla base dei centri di tuple ReSpecT (considerati più genericamente artefatti di coordinamento) è quella di avere una prima classe di astrazioni che incapsula le leggi di coordinamento al fine di supportare e gestire le attività dei sistemi multiagente. Gli agenti si interfacciano al centro di Tuple ReSpecT (analogo allo spazio delle tuple) mediante primitive Linda non preoccupandosi di definire l'interazione entro coordinate spaziali o temporali. Il centro di tuple incapsula gli eventi comunicativi e tramite reazioni influenza la semantica delle operazioni effettuate dagli agenti. Tutto ciò si traduce nel fatto che i coordinabili vengono totalmente sollevati dall'onere di gestire il coordinamento, che viene relegato al centro di tuple. In questo modo si realizza il disaccoppiamento tra gli agenti e le relative logiche di coordinazione, che vengono quindi spostate dai coordinabili al medium di coordinazione. Un centro di tuple pertanto non è più da considerarsi come un mero spazio di indirizzamento condiviso, bensì come un artefatto di coordinazione che possiede un proprio stato e reagisce ad eventi di comunicazione, che avvengono al proprio interno a tempo di esecuzione, condizionando il comportamento dei coordinabili. Il comportamento dei centri di tuple, a sua volta, può essere modificato dinamicamente a seconda delle esigenze specifiche dell'applicazione. In TuCSoN è possibile variare a tempo di esecuzione le specifiche di comportamento dei centri di tuple.

Prima di analizzare le componenti del modello TuCSoN al fine di trovare riscontri riguardo agli eventuali modelli temporali implementati è doveroso fare una precisazione. Indipendentemente dal fatto che vi siano o meno modelli temporali già implementati all'interno delle componenti TuCSoN, questo non significa che non sia possibile realizzarli. ReSpecT è un linguaggio *Turing completo* [6] quindi vi è comunque la possibilità di implementare tutti i riferimenti temporali che si ritengano necessari. Le reazioni temporizzate introducono però un overhead non sempre trascurabile. Inoltre la sintassi necessaria per realizzare ad esempio operazioni composte può diventare facilmente contorta e poco leggibile[3]. Per questo è opportuno fornire ai programmatori degli strumenti di sviluppo *first class*.

[3] Servirebbe un pò di *zucchero sintattico*.

4.2 Il modello temporale nelle tuple ReSpecT

4.2.1 Analisi delle tuple ReSpecT

Per quel che riguarda le tuple in TuCSoN non sono stati implementati riferimenti temporali.

La presenza delle tuple ReSpecT *nel medium di coordinazione non è temporalmente definita*.

4.2.2 Possibili implementazioni di un modello temporale per le tuple ReSpecT

Si può pensare di programmare le tuple ReSpecT, inserendovi un *tempo di lease* come in JavaSpace (vedi sez. 3.1.1). Questo significa modificare la struttura della tupla prevedendo un campo che contenga un riferimento temporale. Esso dovrà essere facoltativo, al fine di mantenere la retrocompatibilità, con un valore di default qualora non venga definito da un agente inconsapevole. Gli agenti consapevoli della possibilità di associare un lease alle tuple devono peraltro avere la possibilità di farlo, per cui occorrerà prevedere un parametro specifico (facoltativo) nella primitiva `out()`, oppure prevedere semplicemente un campo specifico (anche questo facoltativo, eventualmente presente ma vuoto) all'interno delle tuple, senza modificare la `out()`. In quest'ultimo caso, il medium di coordinazione deve leggere all'interno delle tuple per conoscere l'eventuale lease time.

Se il lease time è scritto all'interno delle tuple allora può essere all'occorrenza reso disponibile contestualmente al contenuto informativo per gli agenti che le prelevano. Se invece il lease time viene passato insieme alle tuple esclusivamente come informazione per il medium di coordinazione che le dovrà gestire allora questa informazione non sarà direttamente disponibile per gli agenti. Da notare che è sempre possibile, implementando un metodo per modificare il lease time, cancellare tale parametro da una tupla prima che questa venga prelevata, al fine di mantenere eventualmente privata tale informazione.

D'altra parte se non si prevede un campo specifico per il lease time all'interno delle tuple, questo potrebbe in linea di principio essere comunque inserito all'interno del contenuto informativo da parte del medium di coordinazione alla stregua di un campo ordinario.

Si può pensare di non voler modificare le primitive di inserimento native Linda; in questo caso la scelta cadrebbe a prima vista nell'inserire il lease time all'interno delle tuple. Sarebbe comunque possibile aggiungere una primitiva `out(tuple, lease)` senza eliminare la primitiva nativa `out(tuple)`.

È possibile spingersi oltre: si può inserire anche un riferimento temporale all'interno della tupla, quantomeno un riferimento all'istante in cui la tupla è stata inserita. Si possono ovviamente implementare metodi per leggere e eventualmente modificare questo timestamp. Inoltre si potrebbe implementare un meccanismo di notifica che avvisi l'agente dell'eliminazione di una sua tupla scaduta da parte del medium di coordinazione.

4.3 Il modello temporale nelle operazioni ReSpecT

4.3.1 Analisi delle operazioni ReSpecT

4.3.1.1 Primitive base

Le primitive di base `out()`, `rd()`, `in()`, `rdp()`, `inp()`, sono le stesse di Linda (vedi sez. 2.3), con l'aggiunta di:

- `no(TT)`: ricerca una tupla compatibile con il template `TT` nello spazio di tuple; nel caso non ve ne siano, l'operazione ha successo e restituisce al chiamante il template `TT`, altrimenti l'esecuzione viene sospesa, per poi essere ripresa e terminata con successo nel momento in cui si verifichi la condizione di assenza di tuple compatibili.
- `nop(TT)`: analoga alla primitiva no, ma con una semantica predicativa invece che sospensiva.
- `get()`: legge tutte le tuple presenti nello spazio di tuple senza rimuoverle; se sono presenti più tuple le restituisce come lista al chiamante altrimenti ritorna una lista vuota.
- `set(LT)`: inserisce la lista di tuple `LT` nello spazio di tuple.

Dall'analisi di questo primo set di primitive si evince che non è stato implementato alcun tipo di modello temporale.

4.3.1.2 Primitive bulk

L'insieme delle primitive bulk è composto da `rd_all()`, `in_all()`, `out_all()`, `no_all()`. Le prime due sono identiche alle loro omologhe in Linda (vedi sez. 2.3); le seconde presentano una semantica simile a quella delle primitive di base ma lavorano con liste di tuple.

Dall'analisi delle primitive bulk si evince che non è stato implementato alcun tipo di modello temporale.

4.3.1.3 Primitive spawn

Le *primitive spawn* sono state pensate per consentire attività di coordinazione complesse. Esse possono attivare un agente TuCSoN (di tipo Java o tuProlog) o eseguire primitive nello stesso nodo in cui vengono invocate. La semantica spawn *non è sospensiva.* Dalla documentazione disponibile non è possibile definire la sintassi esatta di queste operazioni [14], pur conoscendone le caratteristiche generali, ovvero che necessitano di due parametri per definire rispettivamente le attività di coordinamento da svolgere e il centro di tuple target della computazione. Un terzo parametro è previsto per la sincronizzazione dell'attività di spawn.

Qualora venisse utilizzato un timeout si potrebbe affermare, come unica eccezione, che questo tipo di primitive prevedono un modello temporale che ne circoscrive la durata in un intervallo definito.

4.3.1.4 Primitive uniform

Le primitive uniform sono `uin()`, `uinp()`, `urd()`, `urdp()`, `uno()`, `unop()`. Si differenziano dalle rispettive versioni ReSpecT non uniform per il fatto che la tupla compatibile viene ricercata sempre mediante logiche casuali ma con distribuzione di probabilità *uniforme* sul medium di coordinazione. Mediante l'impiego di queste primitive è possibile modellare il comportamento stocastico di un sistema coordinato.

Dall'analisi delle primitive uniform si evince che non è stato implementato alcun tipo di modello temporale.

4.3.1.5 Primitive di meta-coordinazione

TuCSoN definisce nove primitive di meta-coordinazione, utilizzate dagli agenti per modificare il comportamento del centro di tuple tramite reazioni ReSpecT. Tali primitive sono:

- `out_s(E ,G ,R)`: scrive una tupla di specifica nel centro di tuple restituendola all'agente dopo l'avvenuto completamento dell'operazione.
- `in_s(ET, GT, RT)`: cerca una tupla di specifica nel centro di tuple compatibile al proprio template. Se è presente viene restituita all'agente e rimossa. Altrimenti attende fino a quando non viene inserita una tupla compatibile nel tuple centre (semantica sospensiva).
- `rd_s(ET, GT, RT)`: analoga ad `in_s()` con la differenza che preleva dal tuple centre una copia della tupla di specifica (no destructive reading).
- `rdp_s(ET, GT, RT)`, `inp_s(ET, GT, RT)`: analoghe rispettivamente a `rd_s(ET, GT, RT)` e `in_s(ET, GT, RT)`, con la differenza che non sono bloccanti (semantica predicativa).
- `no_s(ET, GT, RT)`: cerca una tupla di specifica nel centro di tuple compatibile al proprio template. Se non è presente l'operazione ha successo e viene restituito all'agente il tuple template. Altrimenti attende fino a quando non sono presenti tuple compatibili nel tuple centre (semantica sospensiva).
- `nop_s(ET, GT, RT)`: analoga alla `no_s()` ma con semantica predicativa anzichè sospensiva.
- `get_s()`: legge tutte le tuple di specifica nel centro di tuple e le restituisce sotto forma di lista. Se non sono presenti tuple di specifica restituisce una lista vuota.
- `set_s(E1, G1, R1 ... E`$_n$`, G`$_n$`, R`$_n$`)`: reimposta il comportamento dello spazio di tuple inserendo le proprie tuple di specifica. Restituisce all'agente la lista delle tuple inserite dopo l'avvenuto completamento dell'operazione.

Dall'analisi delle primitive di meta-coordinazione si evince che non è stato implementato alcun tipo di modello temporale.

4.3.2 Possibili implementazioni di un modello temporale per le operazioni ReSpecT

Si può pensare di programmare le operazioni ReSpecT, inserendo un *timeout* nelle primitive bloccanti come in JavaSpace (vedi sez. 3.2.1). Questo significa modificare la struttura delle operazioni bloccanti prevedendo un parametro che contenga un riferimento temporale. Esso dovrà essere facoltativo, al fine di mantenere la retrocompatibilità, con un valore di default qualora non venga definito (da parte di un agente inconsapevole di questa possibilità).

Il timeout deve poi essere gestito dal medium di coordinazione, che si deve preoccupare di eliminare in qualche modo la richiesta a cui è associato se non viene soddisfatta in tempo.

Si può pensare di non voler modificare le primitive di prelevamento native Linda; in questo caso si potrebbe pensare di far gestire al centro di tuple le richieste (imponendo un timeout) fatte dagli agenti inconsapevoli dell'esistenza del parametro. Sarebbe comunque possibile aggiungere una primitiva `in(tuple template, timeout)` senza eliminare la primitiva nativa `in(tuple template)`.

Occorrerà quindi aggiungere una copia di ciascuna delle primitive bloccanti aggiungendo il parametro `timeout` all'elenco dei parametri formali, nelle classi in cui vengono definite tali primitive (nelle classi ACC del package `alice.tucson.api`).

Per quanto riguarda un'eventuale notifica all'agente, da parte del centro di tuple, relativa al timeout dell'operazione, occorre osservare che l'agente, la cui operazione è scaduta, non può non essere consapevole di:

1. non essere più in stato di wait;
2. non aver ricevuto la tupla richiesta.

È anche vero, peraltro, che un agente non particolarmente raffinato potrebbe non avere ben chiaro il motivo per cui la richiesta scaduta non sia andata a buon fine. In questo caso[4] può essere utile prevedere un sistema di notifica. Si potrebbe, ad esempio, eliminare la richiesta all'occorrenza del timeout imponendo l'accettazione di una tupla specifica. Occorrerebbe, per questo, modificare anche le classi di definizione degli agenti, rendendoli capaci di accettare tale tupla specifica pur quando questa non corrisponda al template richiesto.

[4] Anche per agevolare la programmazione di agenti particolarmente raffinati.

4.4 Il modello temporale nei centri di tuple ReSpecT

4.4.1 Analisi dei centri di tuple ReSpecT

Un centro di tuple ReSpecT può essere esteso in modo da reagire ad eventi temporali tramite un framework costruito per gli artefatti di coordinazione temporizzati. I centri di tuple ReSpecT possiedono una nozione di tempo costituita da tre dimensioni:

1. *locale:* individua la coordinata spaziale della misurazione ed è un riferimento valido solamente all'interno dello spazio di indirizzamento del centro di tuple.

2. *relativo:* individua la coordinata temporale della misurazione ed è un riferimento valido solamente nell'intervallo temporale che descrive il ciclo di vita del centro di tuple.

3. *discreto:* definisce il modo in cui viene misurato il tempo all'interno di un elaboratore, ovvero scandito da un numero finito di istanti.

Il tempo viene generato da un'orologio interno posseduto dal centro di tuple: non è possibile inizialmente stabilire relazioni tra i tempi locali di due centri di tuple. Il tempo locale di un centro di tuple temporizzato vale zero nel momento in cui viene creato dall'infrastruttura (run-time). Il tempo assoluto è calcolabile sommando l'istante in cui il centro di tuple temporizzato è stato istanziato al suo tempo locale. Così come per gli eventi di comunicazione, è possibile specificare reazioni scatenate da eventi temporali. Le reazioni temporizzate seguono la stessa semantica: una volta innescate, vengono messe nel set delle reazioni scatenate e vengono eseguite atomicamente in ordine casuale. Dal momento che in un determinato istante può verificarsi un solo evento temporale, ogni reazione temporale è eseguita una sola volta. Da ciò ne consegue che un centro di tuple temporizzato può essere programmato per reagire allo scorrere del tempo. Il linguaggio *Timed* ReSpecT è un'estensione che, tramite predicati, rende possibile specificare reazioni ad eventi temporali [15]. I predicati temporali sono i seguenti:

- `current time(?Time)` Questo predicato ha successo (in qualche modo restituisce "vero") se `Time` (tipicamente una variabile) corrisponde al tempo corrente del centro di tuple. Esempio:
  ```
  reaction(in(p(X)),
  ```

`(current time(Time),out r(request log(Time,p(X))))`
ogni volta che un agente esegue in(p(X)), il centro di tuple inserisce una nuova tupla (tramite l'istruzione `out`) al tempo `Time`. Questa (l'inserimento della tupla *r()* nel centro di tuple) è un'operazione che dipende dal tempo, ovvero viene eseguita in un determinato istante.

- `event time(?Time)` Questo predicato ha successo se `Time` coincide con il tempo del centro di tuple nell'istante in cui viene scatenata la reazione causata da un evento comunicativo.

- `before(@Time), after(@Time), between(@MinTime,@MaxTime)` Questi predicati hanno successo se il tempo del centro di tuple locale è rispettivamente minore di `Time`, maggiore di `Time` o compreso tra il tempo `MinTime` e `MaxTime`.

In Timed ReSpecT, le reazioni ad eventi temporali sono analoghe alle normali *reactions*, ovvero:

```
reaction(time(Time), body)
```

che specificano la lista di operazioni da eseguire (il `body` della *reaction()*) non appena il centro di tuple raggiunge il valore temporale `Time` (l' `head` della *reaction()*). Tutte le *reaction* la cui `head` è compatibile con l'evento temporale possono essere scatenate e il loro `body` inserito nel set delle reazioni scatenate. L'update del centro di tuple avviene atomicamente e qualora anche solo un'operazione scatenate dalle *reaction* non vada a buon fine, viene riportato il centro di tuple alle condizioni originali. Tramite Timed ReSpecT è possibile inoltre implementare modelli temporali sia nelle tuple che nelle primitive:

- ***Timed Requests:*** modellano temporalmente le primitive limitando, ad esempio, l'effetto di quelle bloccanti. Un'agente inserisce una `in()` temporizzata strutturata come segue:

  ```
  in(timed(@Time, ?Template, -Res))
  ```

  Se viene inserita, entro lo scadere del tempo, una tupla compatibile a `Template` nel centro di tuple, viene rimossa e restituita all'agente insieme a `Template` con

`-Res` settato su `yes`; in caso contrario la richiesta viene sbloccata con un tuple template compatibile e `-Res` settato su `no`.

- ***Tuples in Leasing:*** modellano temporalmente le tuple limitandone la locazione nel centro di tuple entro intervalli temporali limitati. Le tuple possono essere inserite nel tuple set specificando un tempo di lease, come ad esempio, il tempo massimo di residenza all'interno del centro di tuple. Un'agente inserisce una tupla tramite una `out()` temporizzata strutturata come segue:

  ```
  out(leased(@Time, @Tuple))
  ```

  Contestualmente viene inserita una reazione che viene scatenata, qualora si verificasse che la tupla è ancora presente nel centro di tuple, allo scadere del tempo. Se la tupla è già stata rimossa l'operazione fallisce, altrimenti diviene compito di quest'ultima eliminarla.

È possibile, tramite gli strumenti messi a disposizione da Timed ReSpecT*, implementare reazioni, tuple e operazioni temporizzate secondo un'approccio general-purpose. Questo significa che, il modello temporale implementato nei centri di tuple* ReSpecT*, permette di esprimere politiche di coordinazione time-related su componenti differenti.*

4.4.2 Possibili estensioni di Timed ReSpecT

Nonostante Timed ReSpecT permetta di programmare i centri di tuple in modo che le reazioni possano dipendere dallo scorrere del tempo, non è stato ancora implementato un riferimento al tempo assoluto. In altri termini, non è attualmente possibile programmare una reazione in modo da scatenare un evento in una data e ora specifici.

Si può naturalmente implementare un riferimento temporale assoluto a livello di sistema.

Ciascuna macchina possiede già un riferimento temporale assoluto gestito dal sistema operativo, eventualmente sincronizzato con il resto del mondo, ad esempio tramite NTP (Network Time Protocol)[5].

[5] *http://it.wikipedia.org/wiki/Network_Time_Protocol*

Attualmente il medium di coordinazione mette a disposizione, per ciascun centro di tuple, una variabile $Time$ gestita da un contatore inizializzato a zero nel momento in cui viene istanziato. È possibile, in linea di principio, mettere a disposizione un riferimento temporale assoluto inizializzando opportunamente tale contatore e sincronizzandone gli incrementi. Un altro modo è quello di memorizzare in una variabile il tempo assoluto nell'istante corrispondente a $Time = 0$. In ogni caso è necessario ottenere un riferimento temporale assoluto all'atto della inizializzazione del sistema, eventualmente da controllare periodicamente; per questo si può procedere eleggendo un master del tempo tra i sistemi che ospitano i centri di tuple e utilizzarlo come riferimento per tutti gli altri elementi del sistema. Questo meccanismo di elezione/negoziazione del ruolo di master in un sistema distribuito è un pattern di coordinazione consolidato che si può ovviamente replicare in questa situazione.

Capitolo 5

Esperimenti in TuCSoN

In questo capitolo era prevista, oltre all'implementazione di un modello temporale per le tuple, anche una realizzazione relativa alle operazioni e un'estensione del modello Timed ReSpecT. In realtà i problemi sono stati comunque analizzati concettualmente (vedi sezz. 4.3.2 e 4.4.2), cercando di indicare una possibile direzione da seguire nella costruzione di operazioni temporizzate e riferimenti assoluti in TuCSoN. Si è deciso di implementare qui solamente un modello temporale per le tuple. È doveroso far notare che quest'ultimo è quello più aperto in termini di possibili soluzioni (vedi sez. 4.2.2) e che pertanto offre gli spunti più interessanti. Dal punto di vista concettuale, fra le varie scelte possibili, è sembrato più opportuno inserire all'interno della tupla un attributo che ne descrive una sua caratteristica, che in questo caso consiste nel tempo di permanenza nel medium di coordinazione. Questo è coerente tra l'altro con la filosofia di sviluppo dei creatori di TuCSoN che fa capo al gruppo di ricerca APICe[1], che prevede di cercare di mantenere l'interfaccia tra gli agenti e il medium di coordinazione simile a quella utilizzata dal modello Linda, ove possibile.

5.1 Implementazione di un modello temporale per le tuple ReSpecT

Per inserire il lease nelle tuple occorre inserire nella classe che definisce le tuple un parametro `lease`, intero in ms che esprime l'eventuale tempo di leasing, e un parametro

[1] http://apice.unibo.it/xwiki/bin/view/Main/

`timestamp` che contiene l'istante di creazione della tupla insieme ai relativi metodi get() e set(). Questo andrà gestito dal centro di tuple, definito dalla classe `RespectVMContext`, che dovrà eliminare la tupla all'istante $timestamp + lease$.

Per poter implementare un modello temporale per le tuple ReSpecT è stato quindi necessario modificare due classi del sistema e introdurne una nuova:

1. modifiche nella classe `LogicTuple` (package `alice.logictuple`):

   - è stato introdotto un parametro che memorizza il valore del lease time della tupla. Per fare questo è stato aggiunto un parametro all'interno della classe `LogicTuple` appartenente al package `alice.logictuple`:

     ```
     private int lease; // lease time in ms
     ```

   - è stato introdotto un parametro che memorizza il valore del timestamp della tupla, funzionale al calcolo del lease time (importando il package `java.util.Calendar` che contiene la definizione del tipo Calendar):

     ```
     private Calendar timestamp; //tuple's timestamp
     ```

   - per poter manipolare i campi `lease` e `timestamp` sono stati aggiunti 4 metodi:

     (a) `getLease()`: restituisce il valore del parametro `lease`;

     (b) `setLease(int)`: permette di impostare il valore del parametro `lease`;

     (c) `getTimespamp()`: restituisce il valore del parametro `timestamp`;

     (d) `setTimestamp(Calendar)`: permette di impostare il valore del parametro `timestamp`;

     È stato utilizzato il tipo `Calendar` perché contiene direttamente un metodo per aggiungere un int in millisecondi.

   - sono stati modificati tutti i costruttori delle tuple (ne verranno mostrati solo due) in modo da prevedere la creazione di un timestamp al momento dell'invocazione:

```
public LogicTuple(final String name) {
  this.info = new Value(name);
  this.timestamp = Calendar.getInstance();
  this.timestamp.setTimeInMillis(System.
     currentTimeMillis());
}


public LogicTuple(final String name,
  final TupleArgument[] list) {
    this.info = new Value(name, list);
    this.timestamp = Calendar.getInstance();
    this.timestamp.setTimeInMillis(System.
       currentTimeMillis());
}
```

2. modifiche nella classe `RespectVMContext` (package `alice.respect.core`):

   - all'interno dell'operazione `addTuple(Tuple)` sono state aggiunte le seguenti istruzioni racchiuse all'interno di un blocco `if()` che subordina la loro esecuzione all'esistenza di un tempo di lease (diverso da zero[2]):

     – `Calendar deleteTime = Calendar.getInstance()`: istanzia la variabile contenente l'istante in cui la tupla verrà eliminata dal tuple centre.

     – `deleteTime.setTimeInMillis(((LogicTuple) t).getTimestamp().getTimeInMillis())`: viene inserito all'interno della variabile deleteTime il valore del timestamp della tupla t.

     – `deleteTime.add(Calendar.MILLISECOND,((LogicTuple) t).getLease())`: viene calcolato l'istante di cancellazione della tupla

[2]Un tempo di lease pari a zero non avrebbe senso, perché significherebbe che occorre eliminare la tupla nell'istante stesso della creazione. Il parametro `lease` non inizializzato assume il valore di default assegnato da Java per il tipo int, ovvero zero.

sommando al valore del timestamp contenuto nella variabile deleteTime il suo tempo di permanenza all'interno del centro di tuple (lease time).

- `Timer timer = new Timer(true)`: istanzia un oggetto timer che gestisce la schedulazione di task temporizzati.
- `RemoveTupleTimerTask timerTask = new RemoveTupleTimerTask(this, t)`: istanzia un timerTask creato appositamente per gestire l'eliminazione delle tuple dal medium di coordinazione (vedi oltre, punto 3).
- `timer.schedule(timerTask, deleteTime.getTime())`: al momento dello scadere del lease time l'oggetto timer schedula il timerTask affinchè elimini la tupla dal centro di tuple.

L'istruzione `addTuple(Tuple)` risulterà strutturata, dopo le modifiche, così come segue:

```
public void addTuple(final Tuple t) {
  this.tSet.add((LogicTuple) t);
  if (this.isPersistent) { this.
     writePersistencyUpdate((LogicTuple) t, ModType.
     ADD_TUPLE);
  }
  if (((LogicTuple) t).getLease() != 0){
    /** Instantiate deleteTime for the tuple t */
    Calendar deleteTime = Calendar.getInstance();
    /** Set deleteTime to the value of tuple t
     * timestamp */
    deleteTime.setTimeInMillis(((LogicTuple) t).
       getTimestamp().getTimeInMillis());
    /** Calculate deleteTime for the tuple t */
    deleteTime.add(Calendar.MILLISECOND,((LogicTuple
       ) t).getLease());
    /** Istantiate the Timer and the TimerTask */
    Timer timer = new Timer(true);
```

```
            RemoveTupleTimerTask timerTask = new
               RemoveTupleTimerTask(this, t);
            /** the timer schedule a
             *  thread RemoveTupleTimerTask that delete
             *  tuple t from tuple space at the time
             *  deleteTime */
            timer.schedule(timerTask, deleteTime.getTime());
          }
        }
```

3. creazione della classe `RemoveTupleTimerTask` (package `alice.respect.core`):

   La classe `RemoveTupleTimerTask` contiene i seguenti campi e metodi:

   - `private final RespectVMContext vm`: l'ID del centro di tuple su cui si andrà ad agire per eliminare la tupla.
   - `private final Tuple t`: la tupla da eliminare dal centro di tuple.
   - `public RemoveTupleTimerTask(final RespectVMContext rvm, final Tuple t)`: è il costruttore del timer task che si preoccupa di eliminare la tupla t dal centro di tuple rvm.
   - `public void run()`: metodo che sarà lanciato dallo scheduler al momento giusto e che contiene il codice per eseguire il task (l'eliminazione della tupla).

   Il codice della nuova classe istanziata è il seguente:

```
package alice.respect.core;

import java.util.TimerTask;

import alice.tuplecentre.api.Tuple;
import alice.tuplecentre.api.TupleTemplate;
```

```
public class RemoveTupleTimerTask extends TimerTask {
  private final RespectVMContext vm;
  private final Tuple t;

  public RemoveTupleTimerTask(final RespectVMContext
     rvm, final Tuple t) {
    super();
    this.vm = rvm;
    this.t = t;
  }

  @Override
  public void run() {
    vm.removeMatchingTuple((TupleTemplate)t);
  }
}
```

Conclusioni

È stato affrontato il problema del tempo nei sistemi distribuiti. È stato quindi usato il modello del Prof. Ciancarini per classificarne le componenti, poi sono state analizzate le possibilità di temporizzazione, o comunque di inserimento di un riferimento temporale, all'interno di ciascuna componente. È stato fatto riferimento, in particolare, al modello Linda e ai derivati Linda-like focalizzando in seguito l'attenzione sul modello TuCSoN e sul linguaggio ReSpecT.

Le componenti sono sostanzialmente tre: il linguaggio di comunicazione (le tuple), il linguaggio di coordinazione (le primitive) e il medium di coordinazione (lo spazio o il centro di tuple). Per quel che riguarda il tempo nel medium di coordinazione l'implementazione è già per la maggior parte in stato avanzato, mentre per le primitive è stata abbozzata l'implementazione di un timeout sulle operazioni bloccanti. Nelle tuple non era stato implementato alcun modello temporale. Si è voluto, con questo lavoro, dare un contributo allo sviluppo del middleware TuCSoN.

Da notare che in TuCSoN, tramite l'estensione Timed ReSpecT, che implementa il modello temporale nel medium di coordinazione, è possibile implementare sia operazioni che tuple temporizzate utilizzando reazioni temporizzate general purpose. Le reazioni temporizzate introducono però un overhead non sempre trascurabile. Inoltre la sintassi necessaria per realizzare ad esempio operazioni composte può diventare facilmente contorta e poco leggibile. Al fine di agevolare la programmazione di sistemi distribuiti che utilizzano TuCSoN, vale la pena di fornire ai programmatori degli strumenti di sviluppo *first class*.

Ringraziamenti

Ringrazio il Prof. Andrea Omicini e Il Dott. Ing. Stefano Mariani per il prezioso contributo fornito e la disponibilità dimostrata.

Ringrazio Michele, Laura, Cinzia, Fabio e Matteo per tutto quello che hanno fatto per me in questi anni.

Ringrazio la mia famiglia per l'affetto e il supporto.

Ringrazio Silvia, sempre al mio fianco, il centro della mia vita, il mio cuore.

Bibliografia

[1] ReSpecT at a Glance. http://alice.unibo.it/xwiki/bin/view/respect/.

[2] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, July/August 2000.

[3] Valeria Cardellini. Sincronizzazione nei sistemi distribuiti. Università degli studi di Roma Tor Vergata, 2009.

[4] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[5] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Coordination technologies for Internet agents. *Nordic Journal of Computing*, 6(3):215–240, Fall 1999. Selected Papers of the Tenth Nordic Workshop on Programming Theory (NWPT'98), October 14-16, 1998.

[6] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177, Atlanta, GA, USA, 27 February–1 March 1998. ACM. Special Track on Coordination Models, Languages and Applications.

[7] Galileo Galilei. *Dialogo sopra i due massimi sistemi del mondo*. 1632.

[8] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[10] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Pervasive Computing and Communications*, pages 263–273, 2004. 2nd IEEE Annual Conference (PerCom 2004), Orlando, FL, USA, 14–17 March 2004. Proceedings.

[11] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. 2000.

[12] Andrea Omicini. Tuple-based coordination of distributed systems. ALMA MATER STUDIORUM università di Bologna a Cesena, 2008.

[13] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[14] Andrea Omicini and Stefano Mariani. The tucson coordination model & technology: A guide. In *[TuCSoN v.1.10.3.0206, Guide v.1.0.2, January 9, 2013]*.

[15] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Time-aware coordination in ReSpecT. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer Berlin Heidelberg, April 2005. 7th International Conference (COORDINATION 2005), Namur, Belgium, 20–23 April 2005. Proceedings.

[16] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. 1998.

[17] Andrew S. Tanenbaum and Maarten Van Steen. *Sistemi Distribuiti*. Ed. Pearson, 2007.

[18] tuProlog User Guide. *http://tuprolog.sourceforge.net/doc/2p-guide.pdf*, [tuprolog v.2.1] edition.

[19] JavaSpace user guide. *http://river.apache.org/doc/specs/html/js-spec.html*.