ALMA MATER STUDIORUM-UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
Corso di Laurea in Ingegneria elettronica, informatica e telecomunicazioni

ELABORATO FINALE DI LAUREA
IN SISTEMI DISTRIBUITI

Negoziazione dinamica e rilascio di ACC in TuCSoN

Candidato:
Roberto Togni
Matricola 446555

Relatore:
Prof. Andrea Omicini

Correlatore:
Ing. Stefano Mariani

Anno Accademico 2011–2012
Sessione II

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Andrea Omicini e l'Ing. Stefano Mariani per il prezioso contribuito dato al mio lavoro, i consigli, la disponibilità e la simpatia dimostrata. Un sentito ringraziamento va anche agli amici che mi hanno sempre supportato durante questo percorso durato tre anni e ora giunto al termine. Infine, ringrazio i miei genitori per aver sempre creduto nelle mie capacità e aver finanziato i miei studi; senza il loro aiuto non sarei arrivato a questo traguardo importante della mia vita, perciò GRAZIE!

Indice

Introduzione

Questa tesi si colloca nell'ambito del progetto TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork), tecnologia sviluppata presso il laboratorio di ricerca APICe[1]. TuCSoN è un middleware concepito per fornire i propri servizi ad un sistema di agenti distribuiti sfruttando i *centri di tuple* come medium di coordinazione. I centri di tuple sono particolari spazi di tuple che integrano la possibilità di specificarne dinamicamente il comportamento, e che gli agenti possono utilizzare per svolgere le loro attività sociali e coordinarsi reciprocamente. In particolare la tesi si concentrerà sul concetto di ACC (*Agent Coordination Context*), ovvero il contesto di coordinazione di un agente, introdotto all'interno del middleware TuCSoN come astrazione di base per modellare lo spazio dell'interazione e comunicazione dell'agente, nonché la percezione che esso ha dell'ambiente in cui è situato. Fra gli intenti principali per cui è stato introdotto il concetto di ACC, vi è anche la capacità di vincolare e governare l'interazione tra l'agente e il sistema in cui opera, sulla base di opportune regole che definiscono la sua posizione e i suoi permessi all'interno del sistema. In questo senso, l'astrazione di contesto di coordinazione può risultare utile per modellare, accanto alla coordinazione, aspetti relativi alla sicurezza nei sistemi distribuiti, incapsulando opportune politiche di autorizzazione e accesso controllato. La configurazione di un ACC deve avvenire proprio a seguito di una fase di negoziazione che sancisca il contratto tra l'agente e il sistema, sulla base dei permessi che quest'ultimo detiene.

Dopo un capitolo di introduzione alla teoria dei modelli di coordinazione *tuple-based* e TuCSoN, si analizzerà l'architettura del middleware nella sua versione attuale, con un occhio di riguardo alla struttura di ACC, per poi passare alla definizione e all'implementazione dei meccanismi che realizzano

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

la *negoziazione dinamica* di un contesto di coordinazione tra l'agente e il sistema TuCSoN .

Capitolo 1

Dai modelli di coordinazione tuple-based a TuCSoN

In questo capitolo si vuole fornire al lettore un background sulla teoria dei modelli di coordinazione *tuple-based* nei sistemi distribuiti, focalizzandosi prima sull'analisi di Linda, per poi passare alla descrizione di TuCSoN e dell'astrazione di Agent Coordination Context.

I sistemi software odierni sono tipicamente costruiti attraverso la composizione di diverse attività computazionali indipendenti e concorrenti, le quali interagiscono reciprocamente all'interno di un ambiente distribuito ed eterogeneo, possibilmente per il raggiungimento di un comune obiettivo. In questo scenario, lo scopo della *coordinazione* è quello di regolamentare lo spazio dell'interazione tra le attività indipendenti (quali processi, thread o agenti), disaccoppiate tra loro sia spazialmente che temporalmente, al fine di operare in modo organico e risultare in un comportamento complessivo coerente del sistema. In altre parole, la coordinazione è la "colla" che connette attività separate in un insieme [4]. Secondo il meta-modello proposto da Ciancarini [2], possiamo definire un sistema coordinato come una collezione di *entità coordinabili* che vivono ed interagiscono all'interno di uno spazio di coordinazione, che comprende lo spazio della loro interazione e i *medium di coordinazione*; questi ultimi sono le astrazioni che si fanno carico di promuovere e governare l'interazione tra i coordinabili, in accordo a determinate *leggi di coordinazione* (Figura 1.1). Tali leggi sono formalizzate in termini di *linguaggio di comunicazione*, che specifica la sintassi utilizzata per descrivere l'informazione scambiata, e *linguaggio di coordinazione*, che specifica invece la sintassi e la semantica delle operazioni di coordinazione

utilizzate dai coordinabili per interagire con il medium di coordinazione.

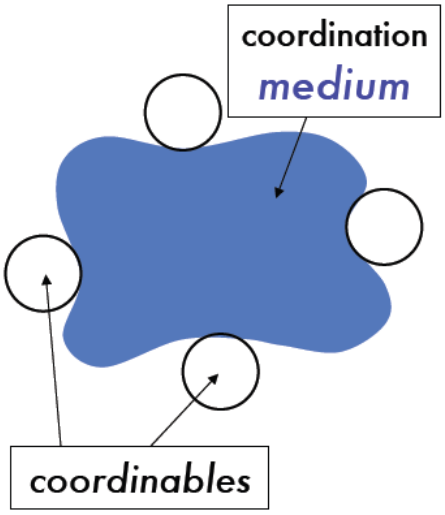

Figura 1.1: Raffigurazione di un meta-modello di coordinazione, in cui l'interazione tra i vari coordinabili è gestita dal medium di coordinazione.

In letteratura si individuano due classi principali di modelli di coordinazione [6]:

- ***Information-driven***: si concentrano sulle informazioni che vengono scambiate all'atto della comunicazione. Il medium di coordinazione funge da spazio di dati condiviso e ciò promuove il disaccoppiamento spazio-temporale tra le entità. Fanno parte di questa categoria i modelli tuple-based (Linda e derivati).
- ***Control-driven***: si concentrano sull'atto di comunicazione vero e proprio, ovvero gli eventi di coordinazione.

1.1 Modelli di coordinazione tuple-based

Sulla base delle definizioni appena fornite, si introduce ora la classe di modelli di coordinazione basata su tuple e le entità che li caratterizzano. Lo spazio di tuple (*tuple space*) mappa l'astrazione di medium di coordinazione e, in maniera analoga ad una *blackboard* associativa, svolge la funzione di spazio dati condiviso (Figura 1.2), utilizzato dalle entità coordinabili per comunicare, sincronizzarsi e cooperare, leggendo, scrivendo e consumando

strutture informative denominate *tuple* [7]. Le tuple, ovvero collezioni ordinate di dati eterogenei, sono le unità elementari alla base della coordinazione e costituiscono quindi il linguaggio di comunicazione. La manipolazione dello spazio di tuple da parte dei coordinabili avviene invece tramite l'invocazione di opportune *primitive*, il cui insieme forma il linguaggio di coordinazione.

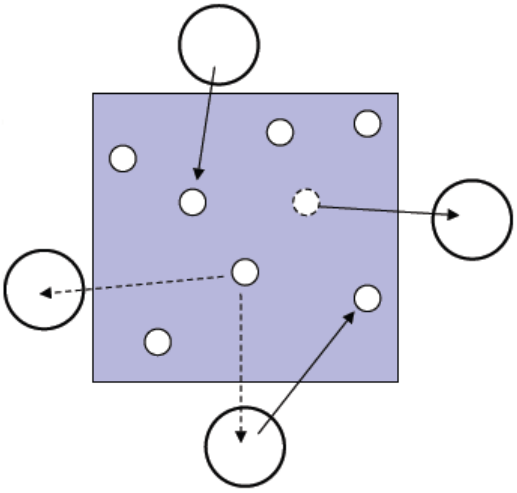

Figura 1.2: Visione dello spazio di tuple come *repository* condiviso.

1.1.1 Il modello di coordinazione Linda

Si analizzano ora le leggi di coordinazione nel caso specifico del modello Linda [3]; sviluppato nell'ambito della programmazione concorrente, è il più noto nella letteratura dei sistemi tuple-based ed è alla base dello stesso TuCSoN. Il linguaggio di comunicazione di Linda definisce la sintassi dell'informazione scambiata dai processi (i coordinabili) in termini di tuple, aggiungendo la possibilità di specificare *patterns* generici di tuple per mezzo di variabili contenute al loro interno (i *templates*, o *anti-tuples*), ed un *meccanismo di matching* responsabile di verificare la corrispondenza sintattica tra tuple e template. Il suo linguaggio di coordinazione definisce invece la sintassi e la semantica delle primitive di coordinazione. Nella sua formalizzazione originale, il modello Linda prevede le seguenti primitive:

- `out(T)`: inserisce la tupla `T` nello spazio di tuple.
- `in(TT)`: rimuove una tupla che fa match con il template `TT` dallo spazio di tuple, e la restituisce al chiamante (lettura distruttiva). Nel

caso in cui sia presente più di una tupla corrispondente al template fornito, ne viene scelta una in modo *non deterministico*. Se invece non vi è alcun match tra le tuple dello spazio ed il template, l'operazione di coordinazione viene sospesa e ripresa non appena almeno una tupla corretta diventa disponibile (semantica sospensiva).

- `rd(TT)`: recupera una tupla che fa match con il template `TT` dallo spazio di tuple, e la restituisce al chiamante senza cancellarla (lettura non distruttiva). Come per la primitiva `in(TT)`, la scelta della tupla risultato è non deterministica e la semantica sospensiva.

Per garantire una maggiore espressività del modello e rispondere alle esigenze di gran parte dei problemi presenti nei sistemi distibuiti, il set di primitive originali è stato in seguito esteso con l'aggiunta di primitive *predicative* (`inp`, `rdp`) e primitive *bulk* (`in_all`, `rd_all`). Le prime sono caratterizzate da una semantica di tipo *success/failure*, che porta al fallimento immediato dell'operazione nell'eventualità in cui non venga trovata alcuna tupla che faccia match con il template; le seconde, invece, consentono di gestire insiemi di tuple e hanno la particolarità di non essere bloccanti e di avere sempre successo:

- `inp(TT)`: analoga a `in(TT)`, ma con semantica predicativa, invece che sospensiva.
- `rdp(TT)`: analoga a `rd(TT)`, ma con semantica predicativa, invece che sospensiva.
- `in_all(TT)`: legge in maniera distruttiva tutte le tuple che fanno match con il template `TT` dallo spazio di tuple, e le restituisce al chiamante. Nel caso in cui non sia presente nessuna tupla corrispondente al template, viene restituita una collezione vuota.
- `rd_all(TT)`: legge in maniera non distruttiva tutte le tuple che fanno match con il template `TT` dallo spazio di tuple e le restituisce al chiamante. Nel caso in cui non sia presente nessuna tupla corrispondente al template, viene restituita una collezione vuota.

È importante notare come lo spazio dell'interazione possa facilmente trasformarsi in un collo di bottiglia nel caso in cui molti processi interagiscano con un solo spazio di tuple, portando così a problemi di congestione dovuti

alla sovrabbondanza di richieste. Per questo motivo, successive estensioni di Linda si sono concentrate sulla possibilità di suddividere lo spazio dell'interazione in più tuple spaces, a loro volta possibilmente distribuiti nella rete su host differenti. Un esempio *application-specific* della sintassi di questa operazione è `ts @ node ? primitive`, con la quale si richiede l'esecuzione di `primitive` sullo spazio di tuple `ts` situato nell'host `node`.

In base a quanto detto finora, possiamo quindi elencare le proprietà fondamentali che fanno capo al modello Linda [6]:

- ***Comunicazione generativa***: in base a questo principio, le tuple generate dai coordinabili hanno un'esistenza indipendente dalle entità che le hanno generate; ciascuna tupla è ugualmente accessibile da tutti i coordinabili, ma non è collegata a nessuno di essi. Ciò comporta un totale disaccoppiamento nell'interazione tra processi, in termini spaziali, temporali e referenziali.

- ***Accesso associativo***: le tuple all'interno dello spazio sono accedute dai coordinabili attraverso il loro contenuto e struttura sintattica (tramite l'uso del template e del meccanismo di matching), e non in base ad un eventuale nome o indirizzo.

- ***Semantica sospensiva***: l'esecuzione di alcune primitive (`in(TT)`, `rd(TT)`) può essere sospesa, fintanto che almeno una tupla corrispondente al template non diventa disponibile. Questa proprietà rende possibile la sincronizzazione tra le entità coordinabili del sistema, sulla base delle informazioni prelevate/depositate dal/nel medium di coordinazione.

1.2 Dagli spazi di tuple ai centri di tuple

Il modello di Linda appena descritto, per quanto si dimostri sintetico e versatile, soffre tuttavia di una mancanza di espressività del suo medium di coordinazione, lo spazio di tuple. L'intera coordinazione si fonda infatti su un insieme di leggi fissate una volta per tutte all'interno del tuple space, e che possono soddisfare le esigenze di certi problemi, ma non di altri. Nel caso si volessero aggiungere nuove funzionalità allo spazio di tuple, allo scopo di ottenere forme di interazione più articolate, un primo possibile approccio risolutivo potrebbe consistere nell'incorporare nuove primitive che permettano

di realizzare le politiche coordinazione prima non consentite. Tuttavia, questa soluzione non riesce a risolvere il problema in maniera *general-purpose* e non si adatta allo scenario di sistema aperto, poiché richiederebbe l'accoppiamento delle entità coordinabili, le quali dovrebbero essere a conoscenza delle nuove primitive introdotte. Una seconda possibile soluzione potrebbe consistere quindi nell'implementare specifici protocolli direttamente nei coordinabili. Anche questo approccio risulta però ingegneristicamente scorretto, in quanto, oltre ad aumentare la complessità dei coordinabili scarica su di loro la responsabilità della coordinazione, togliendola al medium di coordinazione.

1.2.1 L'astrazione di centro di tuple

Ciò che si vuole ottenere, è la possibilità di definire dinamicamente nello spazio di tuple delle nuove politiche di coordinazione che rispondano a specifiche esigenze applicative, programmando il comportamento del medium di coordinazione senza alterare la sintassi e la semantica delle leggi di Linda. Sostanzialmente si tratta di rendere il modello ibrido, inserendo un livello *control-driven* accanto a quello puramente *information-driven* di Linda.

Si introduce quindi l'astrazione di centro di tuple (*tuple centre*) [10], ovvero uno spazio di tuple dotato della possibilità di definirne il comportamento in risposta a determinati eventi di coordinazione. Tale specifica comportamentale del centro di tuple è espressa in termini di un preciso *reaction specification language*, che permette di associare a qualsiasi evento del centro di tuple un insieme, potenzialmente vuoto, di attività computazionali dette *reazioni*, eseguite localmente al centro di tuple. Ogni reazione può:

- accedere e modificare il centro di tuple, leggendo, consumando o producendo tuple;
- accedere alle informazioni relative all'evento scatenante, il quale è reso completamente osservabile;
- invocare primitive su altri centri di tuple.

Inoltre le reazioni posseggono una semantica di successo/fallimento *transazionale*, il che significa che se fallisce una sola istruzione, lo stesso accade per l'intera reazione, mentre lo stato del centro di tuple non viene in alcun modo modificato. In virtù di questa proprietà, tutto ciò è reso trasparente ai coordinabili, che continuano a vedere il tuple centre come un semplice spazio di

tuple, e percepiscono l'invocazione di una primitiva e le conseguenti reazioni innescate come una singola operazione atomica, senza nessuna possibilità di completamento parziale. Inoltre, diversamente dagli spazi di tuple standard, il cui comportamento si limita alle primitive Linda prefissate, le transazioni realizzate sui centri di tuple possono essere rese complesse a seconda delle necessità.

1.2.2 ReSpecT

ReSpecT (**Re**action **Spec**ification **T**uples)[1] definisce un ben preciso tipo di reaction specification language, il quale, all'interno del modello di tuple centre, resta invece un concetto parzialmente astratto che prescinde da una specifica implementazione. Tale linguaggio conferisce al centro di tuple una duplice nozione di spazio di tuple:

- da un lato si ha lo *spazio delle tuple ordinarie*, che rappresentano le unità informative fondamentali alla base della comunicazione fra i processi coordinabili;

- dall'altro si ha lo *spazio delle tuple di specifica*, che consentono di plasmare il comportamento e le regole del medium di coordinazione a seconda delle esigenze applicative.

ReSpecT presenta una sintassi che si rifà a Prolog, pertanto i suoi centri di tuple (Figura 1.3) adottano le *tuple logiche* sia per le tuple ordinarie, che per quelle di specifica. Le tuple ordinarie sono espresse nella forma predicati appartenenti alla logica del prim'ordine [5], mentre quelle di specifica sono particolari tuple logiche scritte nella sintassi `reaction(E,G,R)`. Con questa notazione si vuole indicare che se nel tuple centre si verifica un determinato evento `Ev`, che fa match con il descrittore di evento `E`, e la guardia (ovvero l'insieme delle condizioni che devono essere verificate allo scattare dell'evento) `G` risulta vera, allora verranno eseguiti tutti i *reaction goals* contenuti in `R` secondo la semantica transazionale vista in precedenza. Per una descrizione articolata della sintassi e della semantica di ReSpecT si rimanda a [9].

[1] `http://alice.unibo.it/xwiki/bin/view/ReSpecT/`

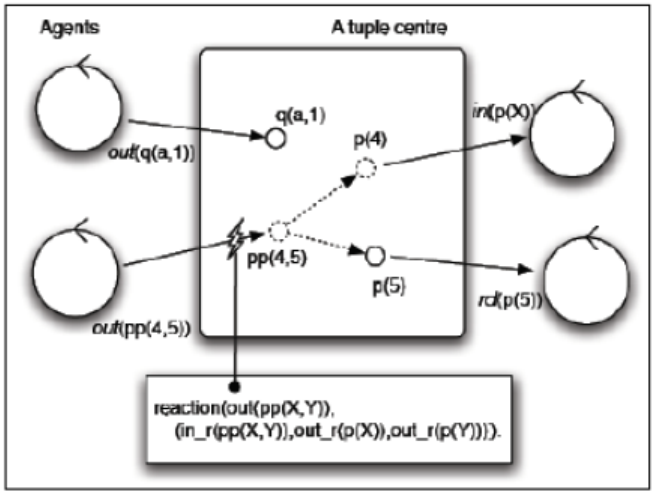

Figura 1.3: Coordinazione di agenti per mezzo di un centro di tuple ReSpecT.

Proprietà dei centri di tuple ReSpecT

I centri di tuple ReSpecT esibiscono tre proprietà fondamentali [6]:

- ***inspectability***: i centri di tuple sono ispezionabili dagli agenti/processi grazie ad alcune primitive, come ad esempio la `rd` e la `rd_s`, per conoscere lo stato del centro di tuple, sia nel contenuto informativo, che in quello di specifica. I centri di tuple sono ispezionabili anche dagli ingegneri/amministratori tramite appositi *tools* per monitorarne lo stato.

- ***malleability***: questa proprietà consente la modifica e l'adattamento del centro di tuple, sia nel suo contenuto puramente informativo (con le primitive `in` e `out`), che in quello di specifica (con le primitive `in_s` e `out_s`).

- ***linkability***: utilizzando l'identificatore completo di tuple centre (che comprende anche il suo indirizzo di rete), una reazione consente di invocare a sua volta delle primitive su altri tuple centres, eventualmente distribuiti.

1.3 Il modello di coordinazione TuCSoN

TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork)[2] è un modello (e anche una tecnologia) per la coordinazione sia di processi distribuiti, che di agenti autonomi e possibilmente mobili. Le astrazioni di coordinazione su cui si basa questo modello sono centri di tuple programmabili tramite il linguaggio ReSpecT, sparsi su nodi attraverso la rete, e utilizzati dagli agenti per interagire e coordinarsi con altri agenti remoti [13][12].

Richiamando l'ontologia proposta da Ciancarini, le entità che caratterizzano un sistema TuCSoN sono [11]:

- Gli ***agenti*** TuCSoN, ovvero le entità coordinabili. Essi sono sparsi nella rete e hanno la particolarità di essere intelligenti, proattivi e mobili, e dunque non associati permanentemente ad un particolare dispositivo.

- I ***centri di tuple*** ReSpecT, che rivestono il ruolo di medium di coordinazione e contrariamente agli agenti, la loro mobilità è dipendente dal dispositivo che li ospita.

- I ***nodi*** TuCSoN, i quali rappresentano l'astrazione topologica di base che ospita al suo interno i centri di tuple.

Ciascun nodo, centro di tuple e agente è identificato in maniera univoca all'interno di un sistema TuCSoN. L'identificatore di un nodo consiste nella coppia `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; dunque, un singolo device può ospitare al suo interno una molteplicità di nodi. Un centro di tuple risulta invece identificato univocamente da `tname @ netid : port`, dove `tname` è il nome ammissibile del centro di tuple, espresso come un termine della logica del prim'ordine senza variabili (*ground*), mentre la coppia `netid : port` denota, come appena visto, il nodo di appartenenza. Infine, ciascun agente è identificato da un nome comune `aname`, anch'esso un termine Prolog ground, e da un UUID (*Universally Unique Identifier*)[3] assegnatogli dal middleware in modo da distinguerlo da qualunque altro agente del sistema.

[2] `http://apice.unibo.it/xwiki/bin/view/TuCSoN/`
[3] `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`

1.3.1 Linguaggio di coordinazione

Il linguaggio di coordinazione di TuCSoN definisce un'insieme di primitive di coordinazione per consentire all'agente di interagire con i centri di tuple. Il linguaggio di comunicazione utilizzato è costituito da tuple logiche, poiché, come già detto, i centri di tuple sono programmati tramite ReSpecT (che è appunto un linguaggio *logic-based*). Ciascuna operazione di coordinazione è costituita da una fase di *invocation*, che consiste nell'invio della richiesta di esecuzione da parte dell'agente verso il tuple centre designato, e una fase di *completion*, durante la quale l'agente riceve dal tuple centre il risultato dell'operazione. La sintassi astratta con cui si esprime l'invocazione di una operazione di coordinazione `op` su di un centro di tuple è

```
tcid ? op
```

dove `tcid` è l'identificatore del tuple centre, comprensivo delle informazioni sul nodo di appartenenza (possiamo anche scriverlo nella forma `tname @ netid : port`).

Di seguito si elencano le varie primitive di coordinazione presenti nella più recente formalizzazione del modello e middleware TuCSoN [11].

Primitive di base

L'insieme delle primitive di base è formato da `out`, `rd`, `in`, `rdp`, `inp`, analoghe alle loro controparti Linda già descritte, con l'aggiunta di:

- `no(TT)`: ricerca una tupla che faccia match con il template `TT` dallo spazio di tuple. Nel caso non vi sia alcuna corrispondenza tra le tuple dello spazio ed il template, l'operazione di coordinazione ha successo, ritornando al chiamante il template `TT`; altrimenti l'esecuzione viene sospesa, per poi essere ripresa e terminata con successo nel momento in cui il match non sussiste per nessuna tupla dello spazio .

- `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 e le restituisce come lista al chiamante. Nel caso non sia presente alcuna tupla, l'esecuzione ha comunque successo e ritorna una lista vuota.

- `set(LT)`: inserisce la lista di tuple `LT` nello spazio di tuple.

Primitive bulk

L'insieme delle primitive bulk è composto da `rd_all`, `in_all`, `out_all`, `no_all`. Le prime due hanno esattamente la stessa semantica delle loro controparti Linda già descritte; le seconde presentano una semantica simile alle versioni non-bulk, ma lavorano con liste, piuttosto che singole tuple.

Primitive uniform

Le primitive uniform sono `uin`, `uinp`, `urd`, `urdp`, `uno`, `unop`. Si differenziano dalle versioni non-uniform per il fatto che la ricerca della tupla corrispondente avviene con probabilità *uniforme* per tutte le tuple dello spazio. Tali primitive sono dunque utili per modellare il comportamento stocastico di un dato sistema coordinato.

Primitive di meta-coordinazione

TuCSoN definisce nove primitive di meta-coordinazione, utilizzate dagli agenti per interagire e sincronizzarsi nello spazio di specifica del tuple centre, leggendo, consumando e scrivendo reazioni ReSpecT. Tali primitive sono:

- `rd_s`, `in_s`, `out_s`
- `rdp_s`, `inp_s`
- `no_s`, `nop_s`
- `get_s`, `set_s`

la cui semantica è la stessa delle loro controparti di base che lavorano nello spazio di tuple ordinarie.

1.3.2 Architettura del modello

Un sistema TuCSoN è prima di tutto caratterizzato dall'insieme dei suoi nodi (possibilmente distribuiti). Ogni nodo offre i propri servizi all'interno di un host connesso alla rete ad una certa porta, dalla quale ascolta le richieste in arrivo. Il sistema prevede un *default port number*, il cui valore è 20504; pertanto con una invocazione di operazione di questo tipo

```
tname @ netid ? op
```

si dichiara implicitamente 20504 come numero di porta.

All'interno di ogni nodo è presente una collezione di centri di tuple, che costituisce lo spazio di coordinazione del nodo. I centri di tuple hanno un ruolo di primaria importanza all'interno di un sistema TuCSoN; questi *artefatti di coordinazione* rappresentano infatti le risorse con cui gli agenti interagiscono per estrarre/emettere le informazioni oppure sincronizzarsi, regolando così le loro attività sociali.

I centri di tuple di un nodo possono essere *attuali* o *potenziali*: al boot del nodo qualunque centro di tuple ammissibile è potenziale e si reifica in una astrazione *run-time* solo alla prima richiesta di operazione su di esso. Ogni nodo TuCSoN definisce un tuple centre di *default* il cui nome è `default`; pertanto con una invocazione di operazione di questo tipo

```
@ netid : portno ? op
```

si fa riferimento implicitamente al centro di tuple `default`.

Lo spazio di coordinazione TuCSoN

Lo spazio di coordinazione TuCSoN presenta una duplice nozione: quella di spazio *globale* e *locale*. Quello globale è definito come l'insieme di tutti i centri di tuple disponibili su ogni nodo della rete, ciascuno identificato dal suo nome completo, mentre quello locale è l'insieme di tutti i centri di tuple disponibili su tutti i nodi di un singolo hosting device, e a cui si può fare riferimento evitando di specificare l'indirizzo di rete. Pertanto un agente, con la seguente invocazione

```
tname : portno ? op
```

si riferirà implicitamente al suo spazio di coordinazione locale, ovvero al contesto dell'host in cui è in esecuzione.

1.3.3 L'astrazione di Agent Coordination Context

Strettamente correlata a TuCSoN è l'astrazione di *Agent Coordination Context* (ACC)[8], definibile come una sorta di "interfaccia" che separa l'agente dall'ambiente in cui questo è situato (in generale, il sistema multiagente o MAS). Il concetto di *situatedness* fa parte della natura stessa di un agente, definito come una entità *goal* o *task-oriented* che incorpora il controllo

al suo interno ed è strettamente accoppiata con l'ambiente in cui avviene l'interazione; infatti, le azioni di un agente hanno senso solo all'interno di un contesto ben definito, che rappresenta pertanto l'astrazione di primo livello atta a modellare gli effetti che l'ambiente (nella sua accezione sia spaziale, che temporale) produce sulla interazione e la comunicazione tra gli agenti. Più precisamente, un *contesto di coordinazione* nasce nell'ambito dei sistemi coordinati con lo scopo di fornire all'agente una visione dello spazio in cui esso vive, opera e comunica con altri suoi pari, e allo stesso tempo per disciplinare l'interazione dell'agente con il sistema stesso (per definizione, il problema della coordinazione riguarda anche il fatto di *vincolare* l'interazione).

Riprendendo la definizione iniziale di ACC, si noti che con il termine "interfaccia" non si vuole fare riferimento alla nozione propria della programmazione *object-oriented*, seppur essa rivesta per l'agente un ruolo simile a quello per i sistemi a oggetti. Mentre nel paradigma object-oriented, l'interfaccia fornisce una visione dall'esterno verso l'interno dell'oggetto, definendo l'insieme di metodi pubblici che si possono invocare su quest'ultimo, un Agent Coordination Context modella la percezione dell'ambiente esterno dal punto di vista dell'agente (ovvero dall'interno), definendo lo spazio delle sue interazioni ammissibili. Tuttavia, analogamente ad una interfaccia, un contesto di coordinazione separa concettualmente l'agente dall'ambiente, permettendo una progettazione e uno sviluppo indipendente di questi ultimi.

La metafora della sala comandi

La metafora più appropriata per comprendere concretamente il ruolo che un contesto di coordinazione riveste per l'agente all'interno di un sistema multiagente è quella della *sala comandi* [8](Figura 1.4). Secondo questa metafora, quando un agente entra in un nuovo ambiente gli viene assegnata una sala comandi che costituisce l'unico mezzo per percepire e interagire con l'ambiente in cui l'agente è situato. La sala offre all'agente dei dispositivi d'ingresso come luci e schermi (percezione) e dei dispositivi d'uscita come dei bottoni che attivano dei comandi (interazione). Quali e quante luci e bottoni siano disponibili per l'agente e per quanto tempo esso possa usarle varia a seconda dei casi, e dipende quindi dalla configurazione della sala e del sistema nel suo complesso. La configurazione di un contesto di coordinazione *ad hoc* per l'agente deve avvenire a seguito di una fase di *ne-*

goziazione, grazie alla quale si verifica quali siano le attività che l'agente è autorizzato a svolgere in accordo a predeterminate regole del sistema. Da queste considerazioni emerge quindi un'ulteriore *feature* dell'ACC: l'incapsulamento all'interno del sistema multiagente di politiche di sicurezza basate su meccanismi di autorizzazione.

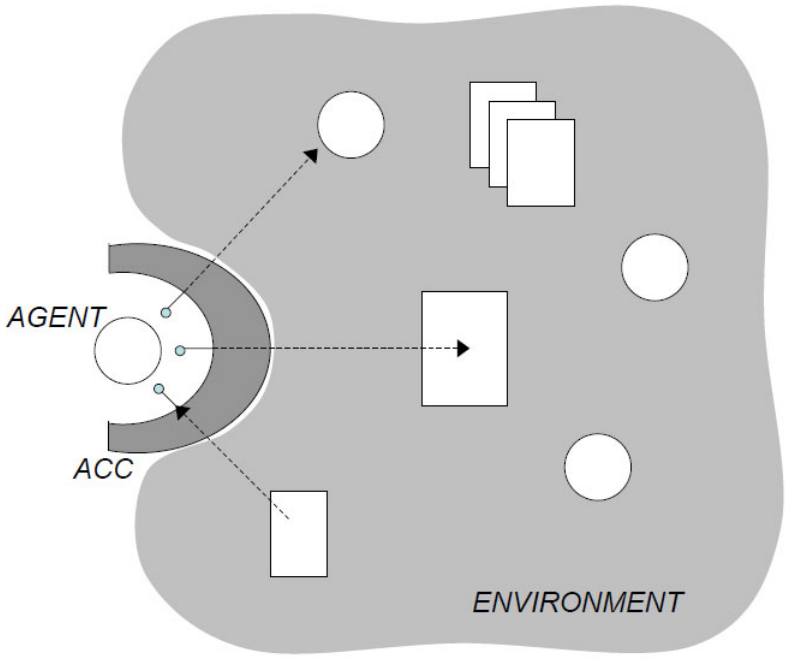

Figura 1.4: Visione di ACC come sala comandi per interagire con l'ambiente.

ACC nel modello TuCSoN

All'interno di una infrastruttura come TuCSoN, un *coordination context* serve a fornire all'agente una visione della collezione dei centri di tuple, il suo spazio di coordinazione. Tutte le varie operazioni non sono invocate dall'agente direttamente sui nodi, ma solamente tramite il proprio ACC, che funge in questo caso da mediatore dell'interazione, fornendo all'agente una *interfaccia* delle operazioni che può invocare (Figura 1.5). Allo stesso modo, il sistema di nodi conosce gli agenti con cui entra in contatto solo per mezzo dell'ACC con cui si rivolgono; quindi, non appena un agente rilascia il suo contesto, per il sistema quell'agente ha cessato di esistere. In quanto governatore dell'interazione, un ACC non è ovviamente vincolato a fornire al

proprio agente una visione completa dello spazio di interazione, ma può invece bloccare totalmente l'accesso verso alcuni centri di tuple o impedire solo certe operazioni per altri, a seconda della sua configurazione. Per esempio, un amministratore potrebbe voler creare un sistema di nodi introducendo un set di regole in base alle quali non può essere consentita l'alterazione dell'informazione su deteterminati centri di tuple, o semplicemente la lettura. Tutte queste regole possono essere iniettate nell'ACC attraverso una configurazione appropriata, a seguito di una negoziazione con il sistema. L'Agent Coordination Context rappresenta in questo senso il *contratto* tra l'agente e il sistema TuCSoN.

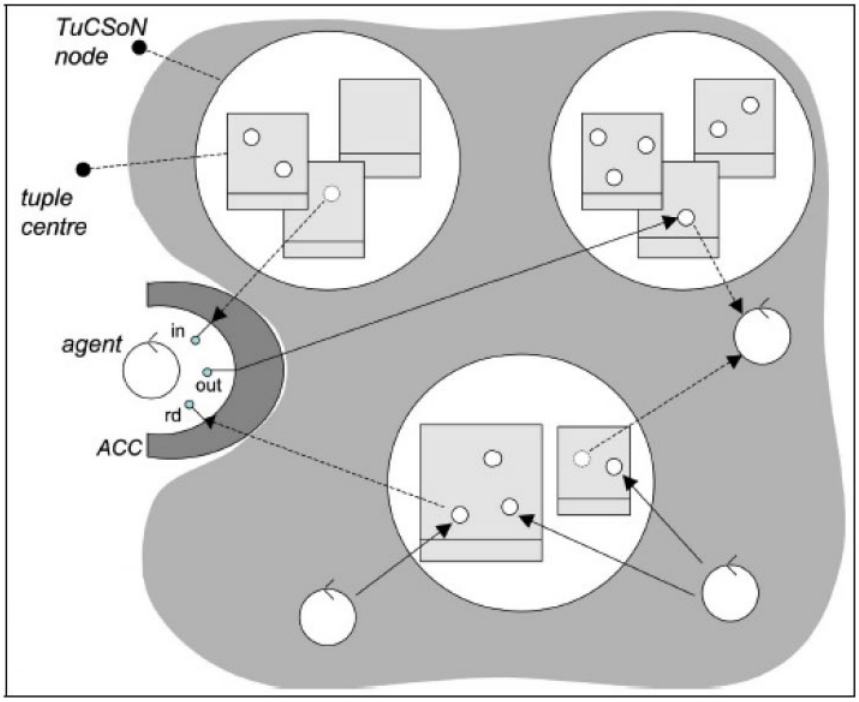

Figura 1.5: ACC nel modello TuCSoN.

Capitolo 2

Agent Coordination Context nella tecnologia TuCSoN

Dopo aver descritto l'architettura del modello, si vuole ora analizzare la tecnologia TuCSoN nella sua implementazione attuale[1] [11], concentrandosi in particolare sull'astrazione di ACC e le dinamiche che ne concernono la richiesta e l'utilizzo.

2.1 Cenni al middleware TuCSoN

L'intera infrastruttura è realizzata in Java e integra al suo interno le tecnologie ReSpecT e tuProlog[2], anch'esse Java-based. Il framework tuProlog risulta fondamentale per TuCSoN, poiché alla base dello stesso motore ReSpecT e del parsing di tuple logiche e identificatori.

Il middleware fornisce i servizi per specificare e governare l'interazione e la coordinazione, ma non mette a disposizione quelli per il supporto diretto degli agenti e del loro *life-cycle* (di cui si occupano invece piattaforme come JADE[3]), che devono quindi essere integrati all'interno di TuCSoN. A tal proposito è presente un set di API (*Application Programming Interface*), che il programmatore del sistema può utilizzare per permettere ai propri agenti Java di interfacciarsi ai servizi offerti da TuCSoN. In particolare, le API comprendono anche una libreria tuProlog (`Tucson2PLibrary`), la

[1] al momento della stesura di questo documento la versione più recente del middleware TuCSoN è la 1.10.2.0205

[2] `http://alice.unibo.it/xwiki/bin/view/Tuprolog/`

[3] `http://jade.tilab.com/`

quale implementa le varie operazioni di coordinazione in forma di predicati logici, in modo da consentire l'uso di TuCSoN anche agli agenti tuProlog [1].

Il middleware mette a disposizione del programmatore anche una serie di *tools* utili al debug, per ispezionare ed interagire con i centri di tuple. Ne fanno parte il CLI (*Command Line Interface*), ovvero una interfaccia shell che include un set di comandi per interagire direttamente con l'infrastruttura TuCSoN in modo molto simile a quello che farebbe un agente, e lo strumento Inspector, che consente agli utenti di monitorare lo stato dei centri di tuple (spazio ordinario e di specifica, operazioni sospese, reazioni scatenate).

2.1.1 Architettura dell'infrastruttura

A livello architetturale, un *Agent Coordination Context* risulta diviso in due componenti topologicamente separati: `ACCProxyAgentSide` e `ACCProxyNodeSide` (Figura 2.1).

ACCProxyAgentSide

L'entità `ACCProxyAgentSide` (package `alice.tucson.service`) rappresenta il contesto di coordinazione in dotazione all'agente, mediante il quale può attuare l'interazione con i nodi; in particolare, esso si occupa di tenere traccia di tutte le sessioni di dialogo aperte con i vari nodi del sistema, e per ognuna gestisce il ciclo di vita delle operazioni di coordinazione (fase di invocazione e completamento). Ciascun agente possiede quindi un *unico* riferimento ad un contesto di coordinazione locale, ovvero `ACCProxyAgentSide`, per interfacciarsi allo spazio di coordinazione TuCSoN.

ACCProxyNodeSide

Lato nodo si ha invece l'entità `ACCProxyNodeSide`, con il compito di rimanere in ascolto delle richieste in ingresso provenienti dalla sua controparte lato agente e gestire l'esecuzione delle varie operazioni di coordinazione. Una volta ricevuta la richiesta di invocazione di una operazione da parte dell'agente, il proxy lato nodo accede al livello dei centri di tuple (*TupleCentreContainer*) e la esegue localmente. Se al primo accesso il tuple centre non dovesse essere ancora presente, allora verrà istanziato dinamicamente. Quando un centro di tuple passa da potenziale ad attuale, si reifica in un

componente attivo a run-time con un proprio flusso di controllo; le funzionalità dei centri di tuple poggiano direttamente sulla macchina virtuale ReSpecT. `ACCProxyAgentSide` è in relazione 1-n con `ACCProxyNodeSide`, pertanto in ogni nodo in cui si riceve una richiesta di partecipazione da parte di un certo agente, sorgerà un apposito proxy messo in comunicazione con la controparte lato agente; dall'altra parte `ACCProxyNodeSide` è in relazione 1-1 con `ACCProxyAgentSide`, dunque tale proxy nasce per dialogare unicamente con quello dell'agente a cui è associato.

Livello di comunicazione

Al più basso livello infrastrutturale si trova il *layer di comunicazione*, il quale si occupa di inviare le *request* e le *response* delle operazioni fra i nodi dell'infrastruttura e gli agenti. Questo livello è responsabile della comunicazione tra i moduli distribuiti del sistema TuCSoN e può sfruttare, in linea di principio, diversi protocolli di comunicazione, seppur attualmente sia stato effettivamente implementato solo il supporto a TCP.

I componenti che si occupano della comunicazione via rete sono implementati nel package `alice.tucson.network`. La classe `TucsonProtocol` modella la sessione di comunicazione tra un agente e un nodo. Tale classe, essendo astratta, può essere estesa per supportare specifici protocolli di comunicazione: per esempio la classe `TucsonProtocolTCP` implementa il supporto a TCP. Le classi `TucsonMsgRequest` e `TucsonMsgReply` modellano rispettivamente la richiesta di invocazione di una certa operazione di coordinazione da parte di un agente, e il risultato dell'operazione inviato dal nodo all'agente. Entrambi sono oggetti serializzabili, in modo da poter essere scritti sullo stream della sessione di dialogo agente-nodo.

2.2 Richiesta di un contesto lato agente

Sia `ACCProxyAgentSide`, che `ACCProxyNodeSide` sono vere e proprie astrazioni run-time con un loro ciclo di vita. La creazione di un ACC avviene a seguito della richiesta, da parte dell'agente, di un contesto di coordinazione tramite la chiamata al metodo statico

```
EnhancedACC getContext(Object aid)
```

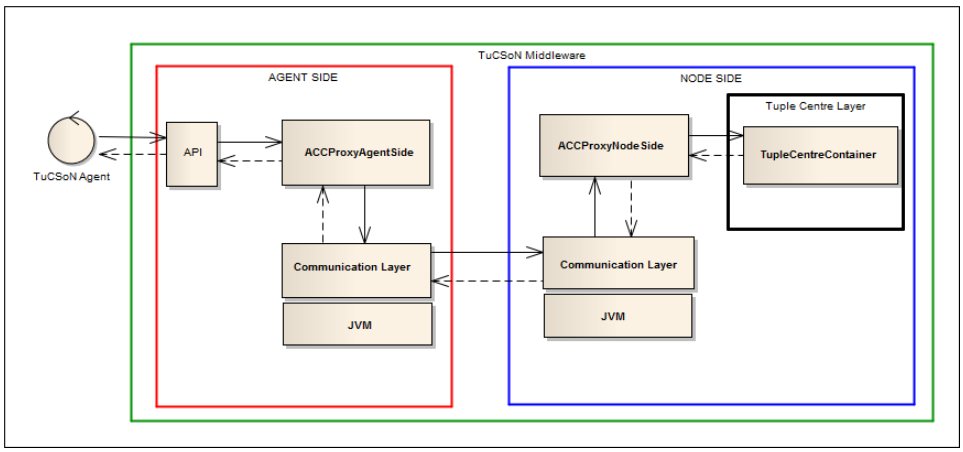

Figura 2.1: Architettura del middleware TuCSoN.

della classe `TucsonMetaACC` (package `alice.tucson.api`). Questo metodo comporta l'istanziazione di un oggetto `ACCProxyAgentSide` associato all'agente con identificativo `aid`, fornito all'atto della creazione. Tuttavia, questo è un dettaglio infrastrutturale che rimane del tutto nascosto all'agente, il quale non percepisce l'ACC come diviso in due componenti proxy, bensì come un'unica entità locale ad esso. Come si può notare, infatti, il riferimento restituito dal metodo `getContext()` non è di tipo `ACCProxyAgentSide`, ma `EnhancedACC`. Questo tipo è definito da una interfaccia facente parte di un set di interfacce ACC standard presenti nelle API, che definiscono solo i metodi relativi alle varie operazioni di coordinazione invocabili, nascondendo dunque la struttura interna e la natura distribuita dell'ACC nei proxy lato agente e nodo. In altre parole, l'implementazione del contesto lato agente è resa trasparente all'agente, il quale può interagire con esso invocando solamente i metodi definiti nell'interfaccia esposta da `ACCProxyNodeSide`.

2.2.1 Le interfacce ACC di TuCSoN

Di seguito viene presentata la gerarchia (Figura 2.2) dei vari Agent Coordination Context presenti nella versione attuale del middleware TuCSoN. Ciascuno di essi è modellato come una interfaccia Java che definisce un set di metodi, ognuno relativo a una certa primitiva di coordinazione. I metodi definiti in ognuno di queste interfaccie ritornano un riferimento di tipo `ITucsonOperation`, la cui interfaccia definisce gli appositi metodi per consentire all'agente di esplorare le informazioni relative all'invocazione (tuple argomento) e l'esito della operazione richiesta (tuple risultato).

- `RootACC` non consente alcuna operazione di coordinazione, ma solamente un'operazione `exit` per rilasciare il contesto attuale chiudendo tutte le sessioni avviate con i nodi e uscire dal sistema.

- `OrdinarySynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre abilitando solo le primitive di base. Dal punto di vista dell'agente il comportamento è *bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente sospende comunque la sua esecuzione in attesa del completamento dell'operazione.

- `OrdinaryAsynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre, abilitando solo le primitive di base. Dal punto di vista dell'agente il comportamento è *non bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente prosegue comunque la sua esecuzione, per essere poi notificato in maniera asincrona al completamento dell'operazione.

- `SpecificationSynchACC` consente di interagire con lo spazio di specifica del tuple centre, abilitando solo le primitive di meta-coordinazione. Dal punto di vista dell'agente il comportamento è *bloccante*: qualunque sia la semantica dell'operazione di meta-coordinazione invocata (sospensiva o predicativa), l'agente sospende comunque la sua esecuzione in attesa del completamento dell'operazione.

- `SpecificationAsynchACC` consente di interagire con lo spazio di specifica del tuple centre, abilitando solo le primitive di meta-coordinazione. Dal punto di vista dell'agente il comportamento è *non bloccante*: qualunque sia la semantica dell'operazione di meta-coordinazione invocata (sospensiva o predicativa), l'agente prosegue comunque la sua esecuzione, per essere poi notificato in maniera asincrona al completamento dell'operazione.

- `SynchACC` è l'unione di `OrdinarySynchACC` e `SpecificationSynchACC`.

- `AsynchACC` è l'unione di `OrdinaryAsynchACC` e `SpecificationAsynchACC`.

- `DefaultACC` è l'unione di `SynchACC` e `AsynchACC`.

- `BulkSynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre, abilitando solo le primitive bulk. Dal punto

di vista dell'agente il comportamento è *bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente sospende comunque la sua esecuzione in attesa del completamento dell'operazione.

- `BulkAsynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre, abilitando solo le primitive bulk. Dal punto di vista dell'agente il comportamento è *non bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente prosegue comunque la sua esecuzione, per essere poi notificato in maniera asincrona al completamento dell'operazione.

- `UniformSynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre, abilitando solo le primitive uniform. Dal punto di vista dell'agente il comportamento è *bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente sospende comunque la sua esecuzione in attesa del completamento dell'operazione.

- `UniformAsynchACC` consente di interagire con lo spazio delle tuple ordinarie del tuple centre, abilitando solo le primitive uniform. Dal punto di vista dell'agente il comportamento è *non bloccante*: qualunque sia la semantica dell'operazione di coordinazione invocata (sospensiva o predicativa), l'agente prosegue comunque la sua esecuzione, per essere poi notificato in maniera asincrona al completamento dell'operazione.

- `EnhancedSynchACC` è l'unione di `SynchACC`, `BulkSynchACC` e `UniformSynchACC`.

- `EnhancedAsynchACC` è l'unione di `AsynchACC`, `BulkAsynchACC` e `UniformAsynchACC`.

- `EnhancedACC` è l'unione di `EnhancedSynchACC` e `EnhancedAsynchACC`. Questa interfaccia è implementata dalla classe `ACCProxyAgentSide`.

Essando `EnhancedACC` il tipo del riferimento restituito da `getContext()`, l'agente ottiene quindi il contesto di coordinazione più completo, e potrà utilizzarlo per invocare qualunque primitiva TuCSoN. L'ACC risulta in questo senso configurabile solo a tempo di compilazione; tramite `getContext()` si ottiene l'ACC più comprensivo ed è possibile scegliere quale interfaccia

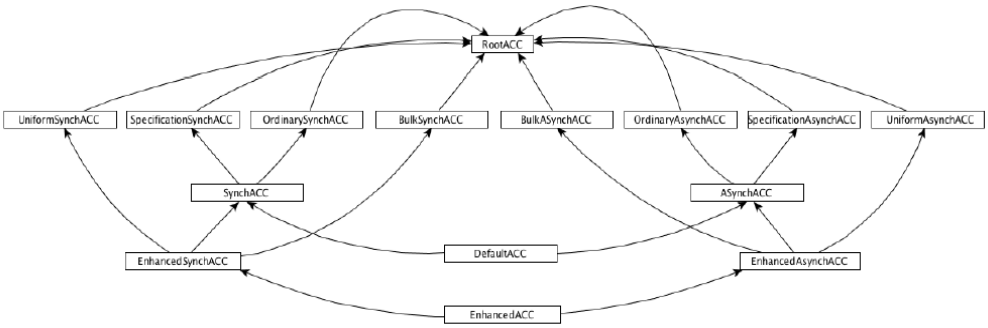

Figura 2.2: Organizzazione gerarchica delle interfacce ACC definite nel middleware TuCSoN. La freccia indica una relazione di estensione.

ACC far utilizzare all'agente semplicemente cambiando il tipo di riferimento del contesto. Nel caso dell'esempio riportato nel frammento di codice sottostante, associando il contesto a un riferimento di tipo `SynchACC` si vincola staticamente a invocare solamente i metodi dell'interfaccia `SynchACC`.

```
1 //TuCSoN Agent behaviour...
2
3 SynchACC acc = null;
4 acc = TucsonMetaACC.getContext(aid);
```

L'astrazione di ACC attualmente implementata in TuCSoN è dunque la più semplice possibile, e non prende in considerazione aspetti relativi all'autorizzazione e la sicurezza. Qualunque agente ottiene indiscriminatamente un contesto di coordinazione che gli consente di eseguire qualunque operazione TuCSoN su qualunque centro di tuple, senza accordarsi prima con una autorità di sistema.

2.3 Creazione della sessione di comunicazione con il nodo

Una volta istanziato un `ACCProxyAgentSide` e assegnato all'agente, la controparte ACC lato nodo non sorge finché non si richiede la prima operazione di coordinazione su di un centro di tuple di tale nodo. La classe

`TucsonNodeService` (package `alice.tucson.service`), che implementa l'astrazione di nodo TuCSoN e l'insieme di servizi che esso offre, provvede a creare in fase di boot un thread `WelcomeAgent` in ascolto sulla porta del nodo. Questo thread rimane in attesa di richieste di partecipazione da parte di un agente, e una volta ricevuta la delega a un oggetto `ACCProvider`, il quale si occuperà di istanziare l'apposito oggetto `ACCProxyNodeSide`, messo in comunicazione 1-1 con il proxy lato agente (Figura 2.3).

2.3.1 Utilizzo dell'ACC

Agent Side

Le dinamiche che concernono l'uso dell'ACC sono soprattutto inerenti alle azioni di *request*, *execution* e *completion* di una operazione TuCSoN. Un agente richiede di eseguire una operazione su un particolare tuple centre invocando l'apposito metodo dell'interfaccia esposta da `ACCProxyAgentSide`, per esempio una primitiva `out` sincrona sul tuple centre `default @ localhost : 20504` (Figura 2.4). Una volta richiesta l'operazione, essa viene delegata ad un metodo protetto della classe, `doOperation()` (Figura 2.5). Questo metodo provvede a prelevare la sessione di dialogo con il nodo (oggetto `TucsonProtocol`) mediante il metodo `getSession()`, oppure a crearla, se non è presente, inviando una richiesta di partecipazione verso il nodo; in questo modo si scatena anche la creazione del proxy lato nodo di cui si è parlato appena sopra. Una volta ottenuta la sessione si sfruttano i servizi del layer di comunicazione per inviare la richiesta (`TucsonMsgRequest`); tale richiesta include diverse informazioni tra cui il tipo di operazione di cui si richiede l'invocazione e la tupla (o le tuple argomento). Se la comunicazione fallisce (a causa della non disponibilità del nodo, di problemi fisici nella rete, eccetera), è generata una eccezione nel proxy lato agente. L'oggetto `TucsonOperation` viene creato e associato a un opportuno flusso di controllo che rimarrà in ascolto sullo stream per ricevere il messaggio di completion (`TucsonMsgReply`).

Node Side

Lato nodo, la controparte `ACCProxyNodeSide` possiede un proprio flusso di controllo che ha il compito di rimanere in ascolto sullo stream in attesa di richieste da parte dell'agente. Una volta ricevuta, il proxy provvede a

```
1  public void run(){
2      TucsonProtocol mainDialog = null;
3      try{
4        mainDialog = new TucsonProtocolTCP(new ServerSocket(↩
           port));
5      }catch(IOException e){}
6
7      TucsonProtocol dialog = null;
8      boolean exception = false;
9      try{
10       while(true){
11         log("Listening on port " + port + " for incoming ACC ↩
             requests...");
12         dialog = mainDialog.acceptNewDialog();
13         dialog.receiveFirstRequest();
14
15         if(dialog.isEnterRequest()){
16           dialog.receiveEnterRequest();
17           ACCDescription desc = dialog.getContextDescription↩
               (); //contains some informations about the ↩
               caller agent
18           log("Delegating ACCProvider received enter request↩
               ...");
19           contextManager.processContextRequest(desc, dialog);
20         }
21       }
22     }catch(Exception e){
23         exception = true;
24     }
25
26     if(exception){
27       try{
28         dialog.end();
29       }catch(Exception e){}
30     }
31 }
```

Figura 2.3: Metodo `run()` del thread `WelcomeAgent`.

```
1 //TuCSoN Agent behaviour...
2
3 TucsonTupleCentreId tcid;
4 LogicTuple tuple;
5 ITucsonOperation op;
6
7 try{
8     tcid = new TucsonTupleCentreId("default","localhost","↩
          20504");
9     tuple = LogicTuple.parse("persona(Roberto, Togni)");
10    op = acc.out(tcid,tuple,(Long)null);
11 }catch(Exception e){}
```

Figura 2.4: Sequenza di istruzioni che un agente TuCSoN deve eseguire per eseguire una primitva `out` sincrona sul centro di tuple `default @ localhost : 20504`.

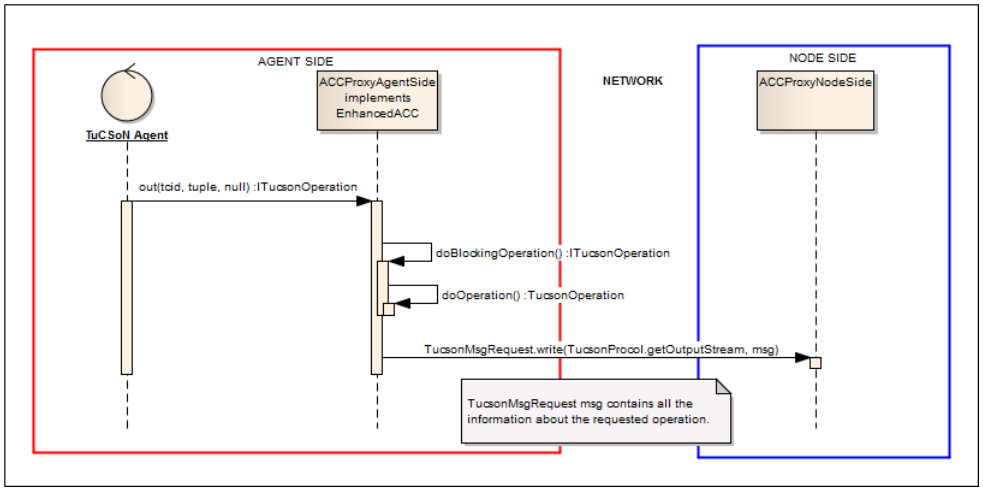

Figura 2.5: Diagramma di sequenza che mostra la sequenza di operazioni lato agente scatenate all'atto dell'invocazione di una operazione `out` sul contesto.

interpretarla, e in seguito a gestirla utilizzando i servizi della classe statica `TupleCentreContainer`. Essa fornisce al proxy l'interfaccia d'accesso per l'interazione con lo spazio di centri di tuple locali al nodo e ha il compito di localizzare il tuple centre target, provvedendo anche a crearlo e a configurarlo nel caso in cui questo non sia presente perchè mai acceduto.

Quando un'operazione è conclusa, il tuple centre ne notifica, in modo asincrono, il completamento al proxy del nodo, che utilizzerà poi i servizi offerti dal livello di comunicazione per inviare a sua volta una notifica, chiaramente anch'essa asincrona, ad `ACCProxyAgentSide`. A questo punto l'oggetto `TucsonOperation` può essere aggiornato con le informazioni sull'esito dell'operazione (successo o fallimento, e tuple risultato) e restituito all'agente.

Rilascio del contesto

Una volta che l'agente ha terminato i suoi compiti, può rilasciare il proprio contesto richiamando su di esso l'operazione `exit()`. Il metodo provvede a chiudere ogni sessione di dialogo aperta con i vari nodi, distruggendone così i rispettivi proxy lato nodo. Rilasciando il contesto, l'agente abbandona a tutti gli effetti il sistema TuCSoN e non è più riconosciuto dai vari nodi con i quali era precedentemente in contatto.

Capitolo 3

Negoziazione dinamica di ACC

Questo capitolo si concentrerà sulla progettazione delle astrazioni software che implementano la negoziazione di un *Agent Coordination Context* tra l'agente e il sistema di nodi TuCSoN. Nel precedente capitolo si evidenziati i limiti dell'implementazione di ACC presente nella versione attuale del middleware, che consente solo una assegnazione *statica* di un contesto per l'agente, in uno dei possibili tipi predefiniti già descritti. Il passo in avanti che si vuole ora poter fare, consiste nell'introdurre all'interno della infrastruttura Java un meccanismo di negoziazione con i nodi del sistema che sia *dinamico* e attivo a run-time, e che incapsuli politiche di autorizzazione per gli agenti partecipanti al sistema.

3.1 Requisiti ed ipotesi preliminari

Un agente che vuole poter iniziare la propria sessione di lavoro con un qualunque nodo TuCSoN, deve prima ottenere dall'infrastruttura un Agent Coordination Context, per mezzo del quale sarà poi in grado di interagire con lo spazio di coordinazione invocando le operazioni che gli sono concesse. Tuttavia, una volta ottenuto il contesto, l'agente non sarà comunque abilitato a partecipare attivamente al contesto di coordinazione di un certo nodo finché non negozia con esso. Per accedere ai servizi di coordinazione offerti dai centri di tuple di quel nodo, l'agente dovrà negoziare la propria partecipazione a tale nodo, e a seguito di essa configurare in maniera appropriata il suo ACC. Un agente tenuto a negoziare la sua partecipazione verso un

nodo deve poter richiedere esplicitamente le sue *capabilities*, ovvero le azioni che desidera poter svolgere; queste azioni si esprimono in termini di primitive coordinazione, trascurando la semantica sincrona/asincrona dal punto di vista dell'esecuzione dell'agente. Poiché le potenzialità dell'agente sono negoziate come intersezione tra quelle che esso richiede, e quelle che può effettivamente ottenere, deve esistere una entità *Authority* di amministrazione che tenga traccia delle primitive di coordinazione che l'agente è autorizzato ad invocare.

Una volta ultimata la fase di negoziazione tra l'agente e il nodo, ogni successiva operazione di coordinazione dovrà essere ammessa o bloccata *localmente* dal contesto prima che venga inviata la richiesta di invocazione verso il nodo remoto. Da queste considerazioni risulta evidente che l'ACC in dotazione all'agente debba essere dotato di un componente `ACCFilter`, che svolga la funzione di filtraggio delle operazioni autorizzate da quelle che non lo sono. Tale componente riflette una delle proprietà già menzionate che caratterizzano l'ACC, ovvero la capacità di *governare e vincolare* le interazioni tra l'agente e il sistema. Per essere precisi, un ACC sarà dotato di una molteplicità di filtri, ciascuno associato a un particolare nodo, in quanto, secondo il modello infrastrutturale trattato in questa tesi, ogni nodo TuCSoN è *indipendente*, e definisce un proprio spazio di coordinazione con autorizzazioni possibilmente differenti da quelle di altri nodi. Nella Figura 3.1 è rappresentata una prima modellazione dell'entità `ACCFilter` come componente di `ACCProxyAgentSide`; la classe dispone di una operazione `doCheck()`, che serve a controllare se una certa primitiva è permessa o meno dal filtro.

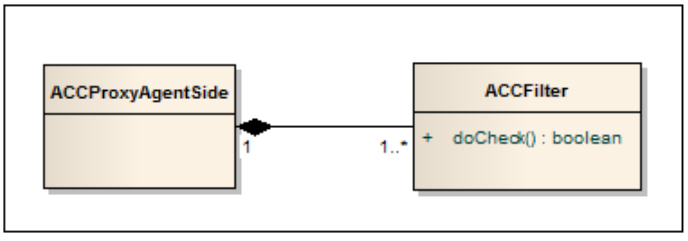

Figura 3.1: Un primo modello dell'entità `ACCFilter`.

L'astrazione di ACC, in virtù delle sue proprietà, può risultare utile al fine di modellare la presenza di un agente all'interno di un *contesto organizzativo*, per esempio definendo lo spazio delle interazioni ammissibili in rispetto

delle regole di tale organizzazione. Nel caso di TuCSoN, l'introduzione di un contesto di coordinazione per l'agente rende possibile una estensione dell'infrastruttura verso l'approccio di RBAC (*Role-Based Access Control*)[15][14], per gestire accanto alla coordinazione, anche problemi relativi alla sicurezza all'interno di sistemi strutturati come organizzazioni complesse. Nell'ottica di un modello di sistema a controllo d'accesso basato su ruoli come RBAC, quando l'agente negozia il proprio ACC specifica anche i ruoli che vuole attivare; se la richiesta è compatibile con le regole correnti dell'organizzazione, l'ACC viene configurato con le opportune caratteristiche derivanti dai ruoli richiesti e assegnato all'agente. Da quel momento in avanti esso potrà usare l'ACC per interagire con l'organizzazione, sfruttando le azioni e le percezioni che gli sono permesse dall'ACC stesso. Uno speciale centro di tuple denominato `$ORG` funge da *repository* per le regole (dinamiche) dell'organizzazione.

Nell'ambito di questa trattazione si trascureranno i complessi vincoli derivanti da RBAC, adottando un modello di sistema TuCSoN che abbraccia una visione di sistema come organizzazione di centri di tuple molto più semplificata (ogni nodo definisce una propria organizzazione, con proprie regole). Nonostante ciò, si desidera mantenere il concetto di *ruolo*, anch'esso ulteriormente semplificato, come astrazione di base per la modellazione dei permessi concessi all'agente. Per ruolo, in questo caso, si intende un insieme o categoria di primitive di coordinazione che l'agente ha il diritto di eseguire, e ciascun ruolo è distinguibile dagli altri per mezzo di un nome che funge da identificatore. I ruoli risultano utili a modellare la posizione di un agente all'interno di un contesto e le *capabilities* a lui assegnate con un livello di astrazione più elevato, piuttosto che esprimerle in termini di semplici liste di primitive; è dunque possibile raggruppare primitive di un certo tipo (per esempio tutte quelle che effettuano una lettura sul centro di tuple, oppure tutte quelle che agiscono sullo spazio di specifica) all'interno di uno specifico ruolo, e raggruppare semanticamente agenti che rivestono le stesse posizioni.

3.2 Progetto della Authority

Il primo problema che ci si pone, è quello di definire un'entità che si occupi di mantenere le regole e i permessi relativi agli agenti partecipanti al sistema TuCSoN. Questa autorità potrebbe far parte del nodo a cui l'agente richiede il proprio *coordination context*, ovvero nel momento in cui esso invoca il me-

todo statico `getContext()` della classe `TucsonMetaACC`, che a sua volta scatena l'istanziazione di un oggetto della classe `ACCProxyAgentSide`, il contesto di coordinazione lato agente (Figura 3.2). Si noti che nella implementazione attuale di TuCSoN non è presente alcun protocollo di negoziazione all'atto della richiesta di ACC: infatti il middleware assegna indiscriminatamente un contesto (o meglio, un riferimento al contesto) di tipo `EnhancedACC` all'agente, che gli permette di invocare senza restrizioni qualunque primitiva di coordinazione su qualunque centro di tuple di ogni nodo del sistema. Una volta ottenuto il contesto, è comunque possibile vincolare l'agente ad invocare solo certe primitive, esplicitandone l'assegnazione a uno dei tipi di ACC predefiniti (descritti nel Capitolo 2). Ad esempio con

```
OrdinarySynchACC acc = getContext();
```

il linguaggio Java stabilisce che tramite il riferimento `acc` è possibile invocare solamente i metodi definiti nell'interfaccia `OrdinarySynchACC`, ovvero tutti quelli relativi alle primitive di coordinazione di base e con semantica sincrona per l'agente. Questo vincolo è tuttavia definito *staticamente*, a tempo di compilazione, e non a seguito di un accordo con una autorità di sistema e sulla base di regole specifiche.

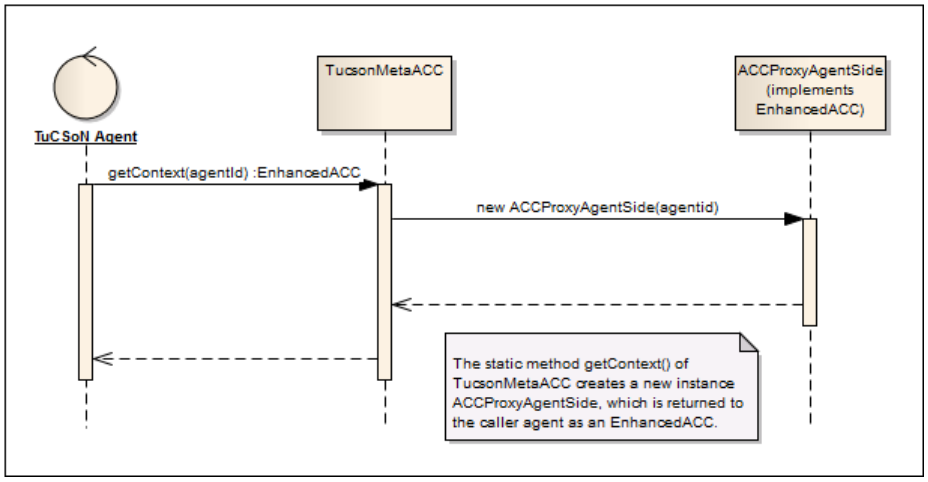

Figura 3.2: Diagramma di interazione che visualizza la sequenza delle principali chiamate a metodo durante la creazione di un contesto lato agente (`ACCProxyAgentSide`).

3.2.1 Il tuple centre $ORG come autorità di sistema

La classe `TucsonNodeService`, che implementa l'astrazione di nodo e dei servizi che offre, provvede anche a creare, all'avvio, uno speciale centro di tuple di configurazione di nome `$ORG`, introdotto appositamente nella prospettiva di una estensione di TuCSoN verso RBAC [11]. Questo tuple centre ha lo specifico compito di conservare informazioni, in forma di tuple logiche, relative al nodo e agli agenti che vi partecipano, e di coordinare alcune attività interne all'infrastruttura, come l'apertura e la chiusura della sessione tra un agente e il nodo, e la gestione dei centri di tuple; è pertanto un tuple centre *riservato*, di amministrazione e non aperto alle normali attività di coordinazione degli agenti. Adottando l'idea di `$ORG` come autorità di sistema, si possono sfruttare le potenzialità offerte dall'astrazione di centro di tuple, da un lato per inserire nello spazio ordinario le tuple contenenti tutte le informazioni sui ruoli presenti e su quelli che un certo agente può ricoprire, dall'altro per incapsulare nel medium di coordinazione un opportuno protocollo di negoziazione basato sulla richiesta di attivazione di ruoli da parte degli agenti, programmandone il comportamento tramite il linguaggio ReSpecT.

Le informazioni sui ruoli

Per mantenere le informazioni relative ai ruoli negoziabili con il nodo si introduce una tupla strutturata nel seguente modo:

```
role(RoleId, Primitives)
```

dove `RoleId` indica il nome del ruolo (espresso come termine logico ground), mentre `Primitives` è una lista Prolog contenente i nomi delle primitive (atomi) che il ruolo ammette. A titolo di esempio, consideriamo che l'amministratore del nodo voglia introdurre due ruoli: *reader*, che ammette tutte le primitive di lettura (distruttiva e non) dallo spazio ordinario, e *writer*, che ammette invece tutte le primitive di scrittura sullo spazio ordinario. In questo caso il tuple centre `$ORG` dovrà essere aggiornato con l'inserimento di queste due tuple logiche:

- `role(reader, [rd, rdp, in, inp, rd_all, in_all, uin, uinp, urd, urdp, get])`
- `role(writer, [out, set, out_all])`

Secondo questo principio, le informazioni sui ruoli possono ammettere alcune inconsistenze: infatti questa formalizzazione non esula l'amministratore dal poter definire in nodi diversi ruoli con uno stesso nome, ma con permessi differenti.

Le informazioni sulle relazioni agente-ruolo

Per esplicitare nel centro di tuple `$ORG` le autorizzazioni relative ai ruoli che gli agenti possono attivare, si introduce invece la tupla

```
role_assignment(Aid, RoleId)
```

dove `Aid` è l'identificatore dell'agente (il semplice nome ammissibile, senza UUID) e `RoleId` l'identificatore del ruolo; entrambi sono termini ground. Per ogni ruolo che un agente ha il diritto di assumere nel contesto di un certo nodo, dovrà essere presente una tupla di questo tipo. Per esempio, se l'amministratore vuole stabilire che l'agente di nome `roberto` può assumere il ruolo `reader`, mentre l'agente `stefano` il ruolo `writer`, dovrà aggiornare lo spazio informativo di `$ORG` con queste due tuple:

- `role_assignment(roberto, reader)`
- `role_assignment(stefano, writer)`

Tali permessi possono essere caricati in `$ORG`, insieme alle tuple `role/2`, direttamente al boot del nodo. A tale scopo, è stata modificata la classe `TucsonNodeService`, aggiungendo un ulteriore parametro di avvio

```
-roleConfig <file>
```

dove in `<file>` si specifica un file di testo contenente una lista Prolog di tutte le tuple `role/2` e `role_assignment/2` da scrivere in `$ORG`. Considerando le varie tuple introdotte nei due esempi precedenti, un eventuale file di configurazione per la classe `TucsonNodeService` dovrà essere così formato

```
1 [  role_assignment(roberto, reader),
2    role_assignment(stefano, writer),
3    role(reader, [rd, rdp, in, inp, rd_all, in_all, uin, uinp↩
         , urd, urdp, get]),
4    role(writer, [out, set, out_all])  ]
```

La classe RoleId

Per modellare l'identificatore di ruolo come un termine logico ground all'interno del middleware Java, introduciamo la classe `RoleId` nel package `alice.tucson.api`. Come si vede in Figura 3.3, questa libreria si appoggia direttamente alla libreria di tuProlog (package `alice.tuprolog`). La Figura 3.4 mostra invece l'implementazione dei due costruttori della classe.

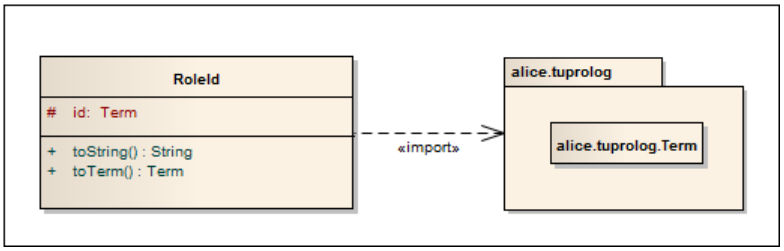

Figura 3.3: Modello della classe `RoleId`.

3.3 Negoziazione e creazione di un filtro per l'ACC

Il centro di tuple `$ORG`, autorità del nodo, possiede ora tutte le informazioni necessarie per negoziare con un agente intenzionato a partecipare al contesto di un nodo TuCSoN, richiedendo l'insieme di ruoli che vorrebbe attivare. Una volta ottenuto un contesto di coordinazione (`EnhancedACC`), tutte le successive invocazioni di operazioni verso un certo nodo saranno rifiutate localmente finché non si negozia con esso e non si configura il rispettivo filtro. A tal proposito, sono stati aggiunti all'interfaccia `RootACC` i metodi `negotiate()` e `inspectNode()` (Figura 3.5), poi ereditati anche da tutte le altre interfacce, compreso `EnhancedACC`. Queste due operazioni, che potremmo definire di "meta-interazione", sono effettivamente implementate nel contesto di coordinazione lato agente (`ACCProxyAgentSide`) e sono direttamente richiamabili dall'agente al fine di eseguire speciali attività di configurazione che *precedono* la vera interazione con lo spazio di coordinazione. Tramite queste due operazioni è infatti possibile contattare, eseguendo un protocollo di azioni prestabilito, il centro di tuple `$ORG` di un nodo prescelto ancora prima che sia stata effettivamente instaurata una negoziazione e inizializzato un `ACCFilter`.

```
1 protected Term id;
2
3 public RoleId(String sid) throws InvalidRoleIdException {
4         try{
5             id=Term.createTerm(sid);
6         } catch (InvalidTermException e){
7             e.printStackTrace();
8             throw new InvalidRoleIdException();
9         }
10        if (!id.isGround()){
11            throw new InvalidRoleIdException();
12        }
13 }
14
15 public RoleId(Term tid) throws InvalidRoleIdException {
16        try{
17            id=tid.getTerm();
18        } catch (Exception ex){
19            throw new InvalidRoleIdException();
20        }
21        if (!id.isGround()){
22            throw new InvalidRoleIdException();
23        }
24 }
```

Figura 3.4: Implementazione dei due costruttori della classe `RoleId`.

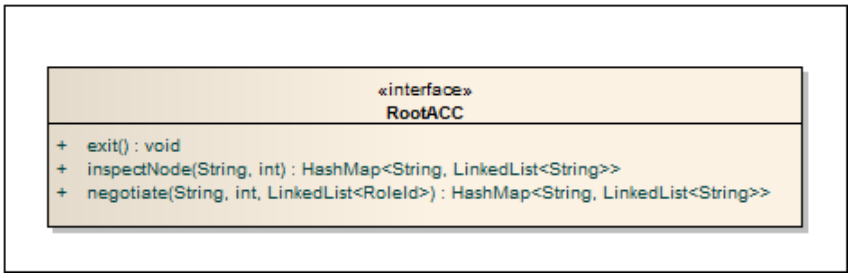

Figura 3.5: Interfaccia `RootACC` di `alice.tucson.api` con l'aggiunta dei due nuovi metodi `inspectNode()` e `negotiate()`.

3.3.1 Il metodo inspectNode()

Questo metodo viene richiamato dall'agente sul contesto specificando le informazioni (indirizzo e porta) sul nodo che si vuole contattare, per andare poi ad estrarre dal suo tuple centre `$ORG` le informazioni sui ruoli che l'agente può assumere (con le relative primitive ammesse). Questa operazione può risultare molto utile, poiché un'agente (eventualmente mobile) prima di iniziare a interagire con un nodo potrebbe voler conoscere le sue potenzialità, in modo da scegliere in un secondo momento se migrare in tale nodo o contattarne un altro.

Sostanzialmente questo metodo provvederà a contattare il tuple centre `$ORG` del nodo desiderato sfruttando le operazioni di coordinazione già presenti. Tramite la primitiva `rd_all` si richiede la lettura di tutte le tuple che fanno match col template `role_assignment(aid, RoleId)`. Una volta ottenute queste tuple, è possibile estrarre da ognuna l'informazione sul roleId e richiamare la `rdp` per effettuare una lettura di una tupla che faccia match con il template `role(roleId, _)`. In questo modo, oltre ai ruoli che può assumere l'agente, si riesce anche ad ottenere la lista delle primitive che tale ruolo ammette. L'insieme di tutte le informazioni prelevate viene inserito in una struttura `HashMap<String, LinkedList<String>>`, dove per ogni roleId (chiave di tipo `String`) si conserva la lista delle sue primitive (`LinkedList<String>`). Una apposita eccezione `InspectNodeException` è stata introdotta, e viene lanciata dal metodo nel caso in cui si verificasse un qualunque errore di sorta durante l'esecuzione.

La Figura 3.6 schematizza l'interazione agente-nodo durante l'esecuzione di `inspectNode()`, mentre la Figura 3.7 mostra uno stralcio di implementazione del metodo.

3.3.2 Il metodo negotiate()

Questo metodo riceve in input, oltre alle informazioni relative al nodo destinatario, una lista di oggetti `RoleId`. Anche questo, al pari di `inspectNode()` si appoggia alle primitive già implementate per contattare il centro di tuple `$ORG`. Per ogni ruolo della lista si richiede la lettura di una tupla che faccia match con il template

```
role_request(aid, role(roleId, _))
```

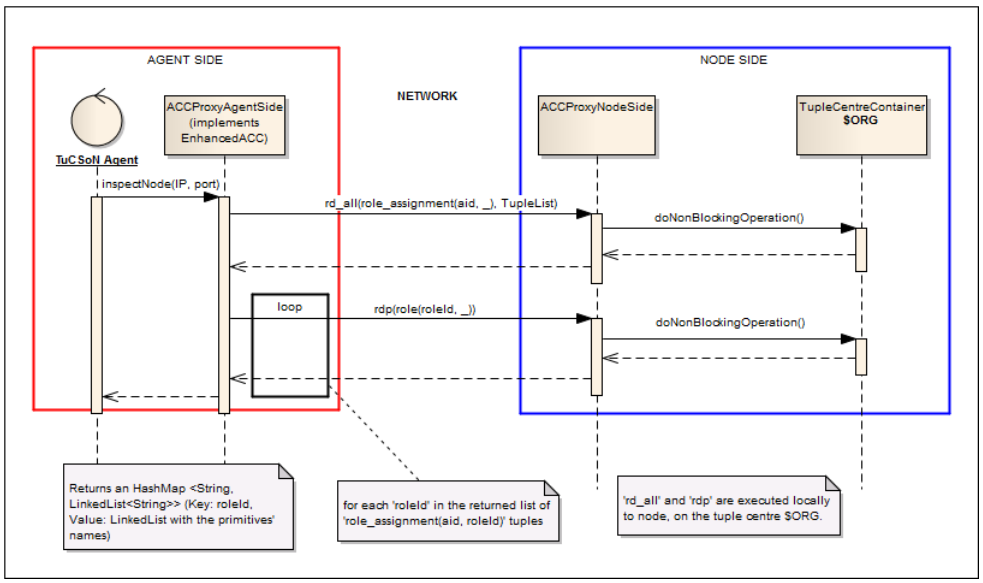

Figura 3.6: Diagramma di interazione che mostra la sequenza delle chiamate principali all'esecuzione del metodo `inspectNode()`.

dove, come si può osservare, l'unica variabile è il secondo argomento del termine composto `role/2`, e pertanto ci si aspetta di leggerne il valore nella tupla risultato unificata con il template. A questo punto, è possibile programmare il centro di tuple `$ORG` in modo da effettuare in un'unica transazione tutti i controlli necessari all'atto dell'invocazione della primitiva di lettura di questa tupla dallo spazio e ritornarla al mittente se i requisiti sono soddisfatti. Per farlo, arricchiamo lo spazio di specifica al boot di `$ORG` con la seguente reazione ReSpecT:

```
reaction(inp(role_request(Aid, role(RoleId, Primitives))),
        (request, from_agent), (
        rd(role_assignment(Aid, RoleId)),
        rd(role(RoleId, Primitives)),
        out(role_request(Aid, role(RoleId, Primitives)))
)).
```

In questa reazione l'evento scatenante è la primitiva `inp` che ha come argomento il template `role_request(Aid, role(RoleId, Primitives))`, mentre le guardie `request` e `from_agent` specificano che la reazione deve essere eseguita durante la fase di invocazione della primitiva e che l'evento

```
1 //Inputs: String netid, int port
2 isMetaInteraction = true;
3 HashMap<String, LinkedList<String>> rolesInfo = new HashMap<↩
      String, LinkedList<String>>();
4
5 try {
6     TucsonTupleCentreId tcid = new TucsonTupleCentreId("'$ORG↩
          '",netid,Integer.toString(port));
7     LogicTuple template = LogicTuple.parse("role_assignment("↩
          +aid.getAgentName()+",_),_");
8     //Reading all tuples matching "role_assignment(aid,_)"
9     ITucsonOperation op = rd_all(tcid,template,(Long)null);
10    if(op.isResultSuccess()){
11         List<LogicTuple> resTupleList = op.↩
               getLogicTupleListResult();
12         //for each result tuple...
13         for(LogicTuple t : resTupleList){
14            String role = t.getArg(1).toString();
15            template = LogicTuple.parse("role("+role+",_)");
16            //Reading the tuple matching "role(role,_)"
17            op = rdp(tcid,template,(Long)null);
18            TupleArgument arg = op.getLogicTupleResult().↩
                  getArg(1);
19            if(!arg.isList()){
20                throw new InspectNodeException();
21            }
22            LinkedList<String> primitives = new LinkedList<↩
                  String>();
23            Iterator<?> it = arg.listIterator();
24            while (it.hasNext()){
25                primitives.add(it.next().toString());
26            }
27            rolesInfo.put(role, primitives);
28        }
29    }
30    isMetaInteraction = false;
31    return rolesInfo;
32
33 } catch (Exception e) {
34    isMetaInteraction = false;
35    throw new InspectNodeException();
36 }
```

Figura 3.7: Stralcio di implementazione del metodo `inspectNode()`.

scatenante deve provenire da un agente. Nel *reaction body* si effettua prima una `rd` di una tupla corrispondente al template `role_assignment(Aid, RoleId)`, e in seguito una `rd` di una tupla corrispondente al template `role(RoleId, Primitives)`. Se le due letture non falliscono, il centro di tuple ha verificato tutti i permessi necessari ed emette con `out` la tupla richiesta dal mittente; se la reazione fallisce, anche la primitiva `inp` avrà esito negativo, e il ruolo richiesto sarà semplicemente scartato.

In altri termini, se la primitiva `inp` termina con successo significa che il ruolo richiesto è stato ottenuto, e nella tupla risultato sarà presente, fra gli argomenti, anche la lista delle primitive ammesse dal ruolo. La lista Prolog è poi convertita in un oggetto `LinkedList` e inserita in una struttura `HashMap`, associata alla chiave roleId. Questa struttura è in seguito passata al costruttore del filtro (classe `ACCFilter`) e poi ritornata all'agente chiamante, il quale potrà così esplorare tutte le informazioni sui ruoli attivati. Una apposita eccezione `NegotiationException` è stata introdotta, e viene lanciata dal metodo nel caso in cui si verificasse un qualunque errore di sorta durante l'esecuzione. Se la negoziazione ha successo, l'ACC lato agente sarà stato configurato con un filtro *ad hoc* per quel nodo, e provvederà a controllare la legittimità di tutte le successive operazioni di coordinazione TuCSoN verso quel nodo. Se successivamente alla prima negoziazione effettuata, l'agente desidera richiedere al nodo l'attivazione di altri ruoli, è sufficiente invocare nuovamente il metodo `negotiate()`, che comporterà dunque l'avvio di una nuova negoziazione e l'instanziazione di un nuovo filtro che va a sovrascrivere il precedente.

La Figura 3.8 schematizza l'interazione agente-nodo durante l'esecuzione di `negotiate()`, mentre la Figura 3.9 mostra uno stralcio di implementazione del metodo.

3.4 La classe ACCFilter

La classe Java `ACCFilter` (package `alice.tucson.service`) modella l'entità filtro introdotta nei requisiti. L'oggetto mantiene fra i suoi campi una `HashMap<String, LinkedList<String>>`, che riceve in ingresso all'atto della costruzione, contenente tutte le informazioni su ruoli e primitive consentite. All'atto dell'esecuzione di una certa operazione di coordinazione, a cui corrisponde l'invocazione di un metodo pubblico di `ACCProxyAgentSide`, deve essere richiamato un metodo di controllo sul-

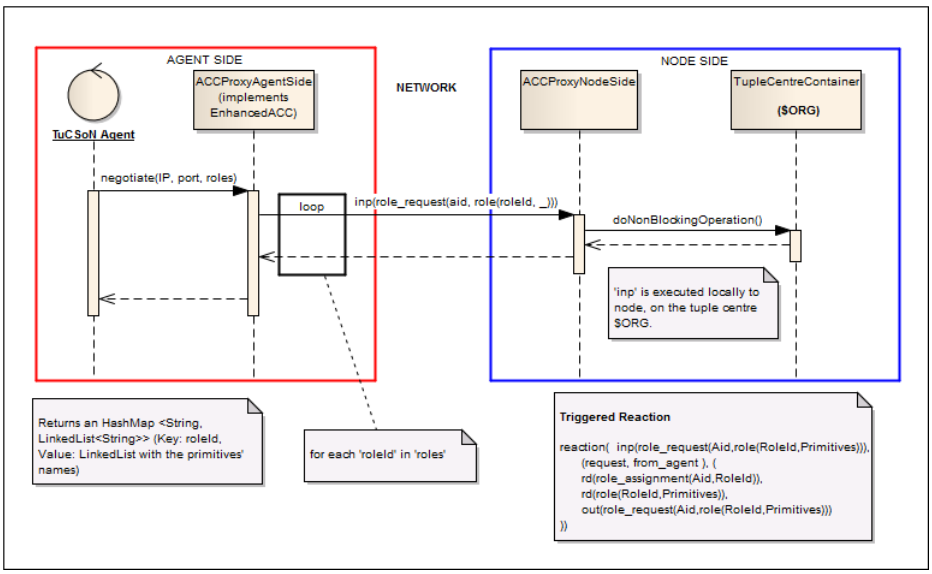

Figura 3.8: Diagramma di interazione che mostra la sequenza delle chiamate principali all'esecuzione del metodo `negotiate()`.

l'oggetto `ACCFilter` (Figura 3.10) per verificare che la primitiva richiesta possa effettivamente essere eseguita; tale controllo deve essere ovviamente effettuato prima che la richiesta di invocazione arrivi alla controparte dell'ACC lato nodo (`ACCProxyNodeSide`).

3.4.1 Dinamica del filtro

Il filtro è il componente di `ACCProxyAgentSide` che si occupa di vincolare l'interazione con lo spazio di coordinazione TuCSoN. Come già detto, viene istanziato uno specifico filtro per ogni nodo con cui si negozia la partecipazione; l'insieme dei filtri è dunque conservato in un campo di `ACCProxyAgentSide`

```
HashMap<String, ACCFilter> filters
```

dove la chiave è una stringa del tipo `IP : port` che identifica il nodo, mentre il valore è l'oggetto filtro associato a tale nodo.

Per come è realizzato il contesto, un agente ha la possibilità di eseguire qualunque operazione di coordinazione TuCSoN invocando il rispettivo metodo pubblico esposto dall'interfaccia `EnhancedACC`. Tuttavia, prima che venga inviata la richiesta verso il nodo, l'operazione deve essere passata al vaglio del filtro, il quale valuterà, in base alla sua configurazione,

```
1 //Inputs: String netid, int port, LinkedList<RoleId> roles
2 isMetaInteraction = true;
3 HashMap<String,LinkedList<String>> rolesInfo = new HashMap<↩
      String,LinkedList<String>>();
4
5 try {
6     TucsonTupleCentreId tcid = new TucsonTupleCentreId("'$ORG↩
          '",netid,Integer.toString(port));
7     for(RoleId roleId : roles){
8       String role = roleId.toString();
9       LogicTuple tuple = LogicTuple.parse("role_request("+aid↩
            .getAgentName()+",role("+role+",_))");
10      ITucsonOperation op = inp(tcid,tuple,(Long)null);
11      if(op.isResultSuccess()){
12        TupleArgument arg = op.getLogicTupleResult().getArg↩
              (1).getArg(1);
13        if(!arg.isList()){
14           throw new NegotiationException();
15        }
16        Iterator<?> it = arg.listIterator();
17        LinkedList<String> primitives = new LinkedList<String↩
              >();
18        while (it.hasNext()){
19           primitives.add(it.next().toString());
20        }
21        rolesInfo.put(role, primitives);
22      }else{
23        log(aid.getAgentName()+" cannot obtain "+"'"+role+"'.↩
              ");
24      }
25    }
26    filters.put(netid+":"+port, new ACCFilter(rolesInfo));
27    isMetaInteraction = false;
28    return rolesInfo;
29
30 }catch (Exception e){
31    isMetaInteraction = false;
32    throw new NegotiationException();
33 }
```

Figura 3.9: Stralcio di implementazione del metodo `negotiate()`.

```
1 //HashMap containing for each roleID (String) the LinkedList ↩
      with all its primitives (String)
2 protected HashMap<String, LinkedList<String>> rolesInfo;
3
4 public boolean doCheck(String primitive){
5
6         Set<String> roleSet = this.rolesInfo.keySet()
7         for(String role: roleSet)
8             LinkedList<String> primitives = rolesInfo.get(↩
                  role);
9             for(String str : primitives){
10                if(str.equals(primitive)){
11                    return true;
12                }
13            }
14        }
15        return false;
16 }
```

Figura 3.10: Implementazione del metodo `doCheck()` della classe `ACCFilter`.

se l'agente dispone dei permessi necessari per eseguirla sul nodo. Il controllo avviene all'interno di `doOperation()`, metodo protetto della classe `ACCProxyAgentSide` a cui è delegata la vera e propria esecuzione di una operazione di coordinazione. Tale metodo provvede a:

- prelevare la sessione di dialogo (oggetto `TucsonProtocol`) verso il nodo tramite il metodo `getSession()`;
- inizializzare l'oggetto `TucsonOperation`, che comprende tutte le informazioni relative all'operazione di coordinazione eseguita;
- impacchettare la richiesta di invocazione (oggetto `TucsonMsgRequest`) e inviarla nello stream.

Il controllo da parte del filtro avverrà ovviamente prima di tutte queste operazioni ed è eseguito dal metodo `doCheck()` della classe `ACCFilter`. Questo metodo accetta in ingresso una stringa contenente il nome della primitiva di coordinazione che si vuole eseguire; se l'output è `false`, allora verrà istanziato un oggetto `TucsonOperation` *vuoto*, settando solamente un opportuno campo booleano `allowed` a `false`, per indicare che tale

operazione è stata rifiutata dall'ACC, e l'operazione avrà termine. All'interfaccia `ITucsonOperation` è stato dunque aggiunto un metodo pubblico `isAllowed()` per controllare lo stato di questo campo, in modo da fornire al chiamante uno strumento per verificare l'effettiva accettazione o rifiuto di una operazione da lui eseguita.

Check pre-negoziazione

Nella Figura 3.11 è mostrata la sequenza delle chiamate scatenate all'atto di una invocazione della primitiva `get` sul centro di tuple `default @ localhost : 20504`. Se il filtro relativo al nodo `localhost : 20504` non è stato ancora inizializzato, significa che l'agente non ha ancora negoziato la sua partecipazione al contesto di quel nodo, e pertanto l'operazione sarà bloccata a prescindere. In tal caso, un nuovo oggetto `TucsonOperation` viene inizializzato settando il campo `allowed` a `false` e ritornato all'agente.

Si noti che le primitive richiamate all'interno dei metodi `negotiate()` o `inspectNode()` non vengono in questo caso bloccate. Si è detto infatti che `negotiate()` richiama al suo interno l'operazione `inp`; poiché la negoziazione non è ancora avvenuta, il filtro non è stato istanziato, tuttavia in questo caso speciale la primitiva deve comunque riuscire ad essere invocata sul nodo destinatario. Un apposito flag `isMetaOperation` indica, se `true`, che una certa primitiva TuCSoN è richiesta all'interno di una operazione di "meta-interazione", quale `negotiate()` e `inspectNode()`, e deve essere dunque invocata anche in caso di filtro non inizializzato.

Check post-negoziazione

La Figura 3.12 mostra invece un secondo caso di invocazione di `get` sul centro di tuple `default @ localhost : 20504`, ma con negoziazione già avvenuta e filtro inizializzato. In questo caso verrà invocato il metodo `doCheck()` sull'oggetto `ACCFilter` per verificare la legittimità dell'operazione. Se l'esito è `true`, il metodo `doOperation()` prosegue inviando al nodo la richiesta di esecuzione; in caso di `false`, invece, una nuovo oggetto `TucsonOperation` viene inizializzato settando il campo `allowed` a `false` e ritornato all'agente. Al termine dell'esecuzione di `doOperation()`

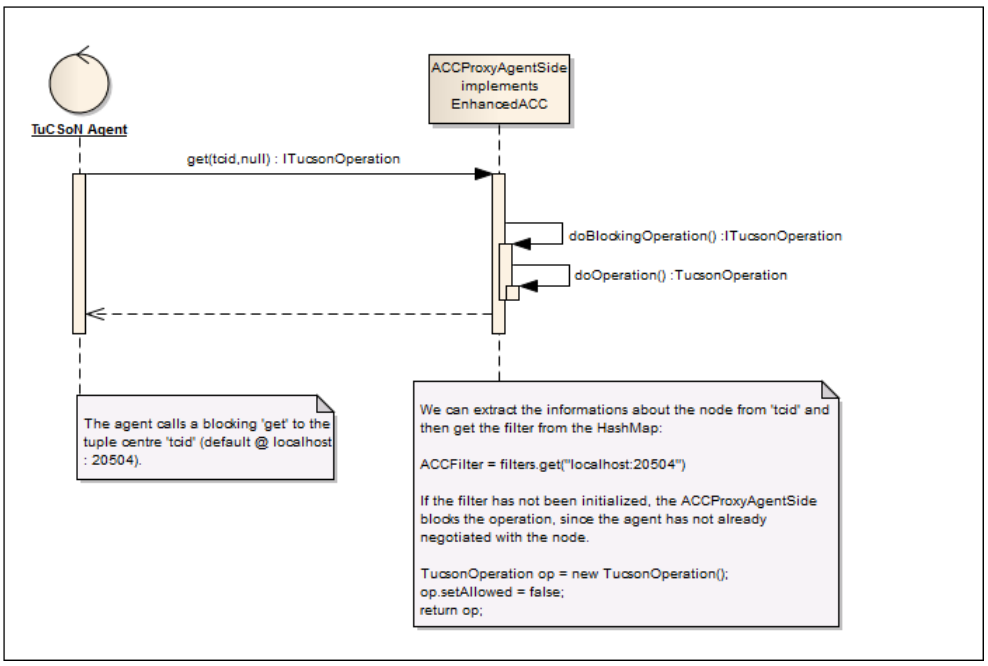

Figura 3.11: Diagramma di interazione che visualizza la chiamata sul contesto della primitiva `get`, senza aver prima negoziato col nodo.

prima che l'oggetto `TucsonOperation` sia ritornato all'agente, si provvede a settare `allowed` a `true`.

Anche in questo caso, se l'operazione TuCSoN è in realtà richiamata all'interno di `negotiate()` o `inspectNode()`, il controllo sul filtro viene bypassato e l'operazione eseguita.

3.5 Ulteriori sviluppi

Le operazioni `inspectNode()` e `negotiate()` qui introdotte sono state rese disponibili anche in forma di predicati logici nella libreria `Tucson2Prolog` (package `alice.tucson.api`). In questa classe Java sono definite le teorie e i predicati logici che consentono ai programmi tuProlog di interagire con i nodi del sistema TuCSoN e invocarvi le varie operazioni di coordinazione. Nella Figura 3.13 è riportata l'implementazione Java del predicato `inspectNode/3`. Gli argomenti in input sono tre termini logici (tipo `Term`): i primi due sono le informazioni su indirizzo e porta del nodo che si vuole contattare, mentre il terzo è la lista delle informazioni sui ruoli che

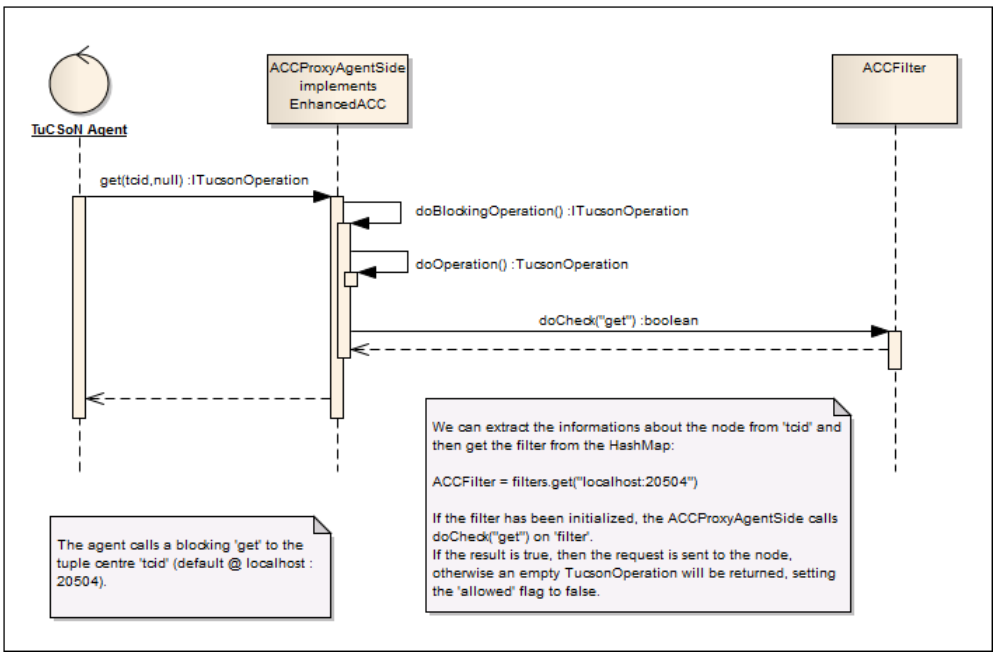

Figura 3.12: Diagramma di interazione che visualizza la chiamata sul contesto della operazione `get`, successivamente alla negoziazione col nodo.

l'agente può attivare (ritornato a seguito della valutazione). La lista risulta formata nel seguente modo:

```
[ruoloA([p1,p2,...]), ruoloB([p1,p2,...]),...]
```

Nella Figura 3.8 è invece riportata l'implementazione Java del predicato `negotiate/4`. Gli argomenti in input sono quattro termini logici: i primi due sono relativi alle informazioni sul nodo con cui si vuole negoziare, il terzo è la lista contenente i nomi dei ruoli (termini ground) che si desidera attivare, mentre il quarto è la lista contenente le informazioni sui ruoli attivati (ritornato a seguito della valutazione). Quest'ultima risulta formata in maniera analoga a quella presentata nel caso del predicato `inspectNode/3`.

Le due nuove operazioni non sono state aggiunte come comandi per il CLI, che continua quindi a funzionare bypassando la negoziazione e il controllo da parte del filtro, in quanto strumento di debug in supporto al programmatore.

Per maggiori dettagli relativi all'implementazione Java si rimanda al repository su Google Code[1]; le modifiche presentate in questa tesi sono reperibili all'interno del branch Git *togni*.

[1] `https://code.google.com/p/tucson/`

```
1 public boolean inspectNode_3(Term netid, Term port, Term ↩
      admissibleRoles){
2
3     if(context == null)
4       return false;
5
6     HashMap<String, LinkedList<String>> result;
7     try{
8       result = context.inspectNode(netid.toString(),Integer.↩
            parseInt(port.toString()));
9     }catch(InspectNodeException e){}
10
11    Struct rolesInfo = new Struct();
12    Set<String> roleSet = result.keySet();
13    //for each admissible role...
14    for(String role: roleSet){
15      LinkedList<String> primitives = result.get(role);
16      Term [] termArray = new Term[primitives.size()];
17      Iterator<String> i = primitives.iterator();
18      int j=0;
19      while(i.hasNext()){
20        termArray[j] = Term.createTerm(i.next());
21        j++;
22      }
23      Struct p = new Struct(termArray);
24      Struct roleInfo = new Struct(role,p);
25      rolesInfo.append(roleInfo);
26    }
27
28    unify(admissibleRoles,rolesInfo);
29    return true;
30 }
```

Figura 3.13: Implementazione del predicato logico `inspectNode/3`.

```
1 public boolean negotiate_4(Term netid, Term port, Term ↩
      desiredRoles, Term activatedRoles){
2
3     if(context == null || !desiredRoles.isList())
4             return false;
5
6     LinkedList<RoleId> roles = new LinkedList<RoleId>();
7     Iterator<?> it = ((Struct)desiredRoles).listIterator();
8     while (it.hasNext()){
9             try {
10                roles.add(new RoleId(it.next().toString()));
11            } catch (InvalidRoleIdException e) {}
12    }
13
14    HashMap<String, LinkedList<String>> result;
15    try{
16        result = context.negotiate(netid.toString(),Integer.↩
              parseInt(port.toString()),roles);
17    }catch(NegotiationException e){}
18
19    Struct rolesInfo = new Struct();
20    Set<String> roleSet = result.keySet();
21    //for each activated role...
22    for(String role: roleSet){
23      LinkedList<String> primitives = result.get(role);
24      Term [] termArray = new Term[primitives.size()];
25      Iterator<String> i = primitives.iterator();
26      int j=0;
27      while(i.hasNext()){
28        termArray[j] = Term.createTerm(i.next());
29        j++;
30      }
31      Struct p = new Struct(termArray);
32      Struct roleInfo = new Struct(role,p);
33      rolesInfo.append(roleInfo);
34    }
35
36    unify(activatedRoles,rolesInfo);
37    return true;
38 }
```

Figura 3.14: Implementazione del predicato logico `negotiate/4`.

Conclusioni e sviluppi futuri

In questa tesi è stata proposta e realizzata una prima implementazione di un meccanismo di negoziazione dinamica di un *coordination context* per l'agente, utilizzando un approccio *role-based*: un ACC è così configurato per vincolare l'accesso alle risorse (centri di tuple) in accordo con i ruoli attivati in quel momento. Utilizzare il già presente tuple centre `$ORG` come *repository* del nodo per mantenere le regole di autorizzazione, si è rivelata una soluzione molto flessibile e in accordo con la filosofia di TuCSoN, secondo la quale ogni servizio può essere fornito agli agenti per mezzo dei centri di tuple. Con le nuove modifiche apportate, l'amministratore è ora in grado di incapsulare all'interno di un proprio sistema di nodi TuCSoN l'insieme delle autorizzazioni e dei vincoli sulle azioni che gli agenti possono eseguire, a differenza della situazione precedente, in cui l'ACC era implementato in maniera limitata e ottenuto senza negoziazione.

Nel semplice caso trattato, ciascun nodo definisce una organizzazione a sé, e quindi un proprio centro di tuple `$ORG` con proprie autorizzazioni, in quanto la tecnologia TuCSoN, nella sua versione attuale, implementa debolmente il concetto di nodi strutturati in organizzazioni complesse. In tal caso, si rende necessario estendere l'infrastruttura in modo tale da consentire all'amministratore di modellare topologie più articolate, in cui i vari nodi sono in qualche modo relazionati tra di loro. Secondo questa visione, un'organizzazione potrebbe essere quindi costituita da più nodi, uno dei quali svolge il particolare compito di *gateway* (contenente i centri di tuple utilizzati per coordinare le attività di configurazione dell'organizzazione), mentre gli altri sono adibiti solamente alle normali attività di coordinazione: un agente interessato a partecipare all'organizzazione, potrebbe quindi negoziare con il tuple centre `$ORG` del nodo gateway e configurare il proprio ACC in modo da ottenere permessi validi anche per tutti gli altri nodi associati [15].

Sviluppi futuri potrebbero includere un'estensione dell'infrastruttura verso

l'approccio più avanzato di RBAC, includendo accanto alle relazioni agente-ruolo, anche relazioni di tipo *inter-ruolo* [14], allo scopo di vincolare maggiormente le potenzialità di un agente. Potrebbe essere opportuno, inoltre, sviluppare ulteriormente il concetto di *ruolo*, che in questa tesi assume l'accezione, semplice e molto limitata, di insieme di primitive che l'agente è autorizzato a invocare; nella prospettiva di organizzazioni complesse, sarebbe utile incapsulare all'interno del ruolo anche le autorizzazioni relative ai centri di tuple su cui si può interagire, e per ciascuno di essi il modo con cui si può interagire (primitive invocabili). Tale problema deriva dal fatto che i nodi stessi ospitano al loro interno centri di tuple speciali quali il già menzionato `$ORG`, e `$OBS`, che si occupa invece di conservare il *log* di tutti gli eventi avvenuti nel nodo, con cui gli agenti dovrebbero poter interagire in maniera limitata; infatti, se si desse loro la possibilità di effettuare scritture o letture distruttuive su tali centri di tuple, ciò risulterebbe in un comportamento incoerente e dannoso per l'organizzazione.

Fra le altre prospettive di sviluppo vi è anche quella di introdurre accanto all'autorizzazione, anche meccanismi di *autenticazione*, per fare in modo che un agente in fase di negoziazione possa dimostrare la propria identità. In ogni caso le idee sviluppate in questa tesi si possono ritenere senza dubbio pienamente *compliant* e come valido punto di partenza per future estensioni dell'infrastruttura TuCSoN orientate verso la modellazione di sistemi che riguardano aspetti organizzativi e di sicurezza più complessi.

Bibliografia

[1] tuProlog User Guide. `http://tuprolog.sourceforge.net/doc/2p-guide.pdf`. [tuProlog v.2.1].

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

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

[4] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[5] John W. Lloyd. *Foundations of Logic Programming*. Springer, 1st edition, 1984.

[6] Andrea Omicini. Coordination-based Systems. `http://campus.unibo.it/80694/`. Course of Distributed Systems, Academic Year 2011/2012.

[7] Andrea Omicini. On the semantics of tuple-based coordination models. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 175–182, New York, NY, USA, 28 February – 2 March 1999. ACM. Special Track on Coordination Models, Languages and Applications.

[8] Andrea Omicini. Towards a notion of agent coordination context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA, October 2002.

[9] Andrea Omicini. Formal ReSpecT in the A&A perspective. In Carlos Canal and Mirko Viroli, editors, *5th International Workshop on Foun-*

dations of Coordination Languages and Software Architectures (FOCLASA'06), pages 93–115, CONCUR 2006, Bonn, Germany, 31 August 2006. University of Málaga, Spain. Proceedings.

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

[11] Andrea Omicini and Stefano Mariani. The TuCSoN coordination model & technology: A guide. `http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide`. [TuCSoN v.1.10.2.0205, Guide v.1.0.1, October 4 2012].

[12] Andrea Omicini and Franco Zambonelli. Coordination of mobile agents for information systems: the TuCSoN model. In *AI*IA'98 Workshop on Knowledge Integration*, pages 94–98, Padova, Italy, 23–25 September 1998. Edizioni Progetto Padova.

[13] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[14] Alessandro Ricci and Andrea Omicini. Agent coordination contexts: Experiments in TuCSoN. In Flavio De Paoli, Sara Manzoni, and Agostino Poggi, editors, *AI*IA/TABOO Joint Workshop "Dagli oggetti agli agenti: dall'informazione alla conoscenza" (WOA 2002)*, Milano, Italy, 18–19 November 2002. Pitagora Editrice Bologna.

[15] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Agent coordination contexts in a MAS coordination infrastructure. *Applied Artificial Intelligence*, 20(2–4):179–202, February–April 2006. Special Issue: Best of "From Agent Theory to Agent Implementation (AT2AI) – 4".