

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

---

CAMPUS DI CESENA  
SCUOLA DI INGEGNERIA E ARCHITETTURA  
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# INGEGNERIZZAZIONE DI RBAC-MAS IN TUCSON

Tesi di Laurea in Sistemi Distribuiti

Relatore:  
ANDREA OMICINI

Presentata da:  
EMANUELE BUCCELLI

Sessione III  
Anno Accademico 2013-2014



# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Background</b>	<b>3</b>
1.1 Role Based Access Control . . . . .	3
1.2 Agent Coordination Context e RBAC . . . . .	6
1.3 TuCSoN e ReSpecT . . . . .	8
1.3.1 TuCSoN . . . . .	9
1.3.2 ReSpecT . . . . .	11
<b>2 Analisi del progetto</b>	<b>15</b>
2.1 Presentazione lavoro di Galassi . . . . .	15
2.1.1 Modello RBAC proposto . . . . .	16
2.1.2 Topologia . . . . .	18
2.1.3 TrustBAC . . . . .	19
2.2 Struttura RBAC di base in TuCSoN . . . . .	21
2.3 Definizione obiettivi progettuali . . . . .	26
2.3.1 Nuova struttura RBAC . . . . .	27
2.3.2 Macrostruttura del sistema . . . . .	29
2.3.3 Comportamento dei sottosistemi . . . . .	32

---

<b>3</b>	<b>Progetto</b>	<b>37</b>
3.1	RBAC . . . . .	37
3.1.1	Struttura RBAC . . . . .	37
3.1.2	Struttura RBAC in \$ORG . . . . .	40
3.1.3	Proprietà RBAC . . . . .	41
3.1.4	Identificazione agenti ed ACC . . . . .	43
3.1.5	Classi agente . . . . .	44
3.1.6	Comportamento RBAC . . . . .	46
3.2	Interazioni con il sistema . . . . .	51
3.2.1	Amministrazione di un nodo . . . . .	53
3.2.2	Negoziazione/Attivazione di un ruolo . . . . .	58
3.2.3	Svolgimento di un ruolo . . . . .	66
3.2.4	Creazione ACC per lo svolgimento del ruolo . . . . .	67
3.3	Sicurezza . . . . .	70
3.4	Test plan . . . . .	72
<b>4</b>	<b>Caso di studio</b>	<b>75</b>
4.0.1	Nodo . . . . .	76
4.0.2	Agente amministratore . . . . .	77
4.0.3	Agente autorizzato . . . . .	83
4.0.4	Agente non autorizzato . . . . .	85
<b>5</b>	<b>Conclusioni</b>	<b>89</b>
	<b>Bibliografia</b>	<b>91</b>

# Elenco delle figure

1.1	Modello RBAC NIST . . . . .	6
2.1	Scenari iniziali . . . . .	27
2.2	Struttura RBAC di base . . . . .	30
3.1	Diagramma delle classi RBAC . . . . .	38
3.2	Ciclo di vita di Agente User . . . . .	62
4.1	Nodo installato . . . . .	77
4.2	Primitiva out con successo . . . . .	85
4.3	Primitiva rd con successo . . . . .	87



# Introduzione

La tesi presentata ha come obiettivo quello di fornire al linguaggio di coordinazione **TuCSoN** (**T**uple **C**entres **S**pread **o**ver the **N**etwork) una prima implementazione funzionante di un'infrastruttura **RBAC**, per il controllo degli accessi utilizzando una policy orientata ai **ruoli**. Il carattere della trattazione risulta essere più orientato verso il pratico, tralasciando ulteriori studi poiché la letteratura ci offre già una solida base di partenza. Il lavoro prende il via dall'analisi del lavoro sviluppato dall' Ing. Galassi [4], che ha analizzato le possibilità offerte da implementazione di RBAC nei sistemi **MAS** (Multi-Agent Systems), ed in particolare definendone una possibile struttura all'interno di TuCSoN. Sebbene l'eccellente lavoro svolto, il tutto è rimasto solamente teoria e non è stata accettata come un'*evoluzione* di TuCSoN; ritenendo che questo sia principalmente dovuto alla grande mole della trattazione sviluppata, la sua complessità intrinseca e la sua difficoltà ad essere implementata *in toto* senza generare confusione, si è deciso di muoversi verso un'implementazione di minore complessità che potesse essere assimilata, accettata ed integrata senza problemi all'interno di TuCSoN. Gli obiettivi della tesi vertono quindi all'estrapolazione di quello che è il *core* dell'infrastruttura organizzativa, con la creazione di una struttura che rispecchi questa organizzazione nei sistemi complessi, ed infine con il rilascio un'implementazione che sia funzionale ed utilizzabile in modo chiaro e intuitivo.





# Capitolo 1

## Background

In questo capito si vuole fornire al lettore un'introduzione basilare ai concetti trattati nella tesi, in particolare alle caratteristiche dell'infrastruttura **Role Based Access Control**, la cui implementazione risulta essere l'obiettivo della tesi, e dell' **Agent Coordination Context**, concetto fondamentale per l'elaborazione della tesi. Infine verrà dato spazio all'infrastruttura **TuC-SoN**, caso di studio scelto per l'implementazione del controllo degli accessi basato su ruoli.

### 1.1 Role Based Access Control

Il controllo degli accessi può essere considerato l'elemento centrale della sicurezza informatica, utilizzato per prevenire utenti non autorizzati dall'accedere alle risorse, prevenire che utenti legittimi vi accedano in modo non autorizzato e fare in modo che utenti legittimi abbiano un accesso autorizzato.

Questo controllo implementa una policy di sicurezza, la quale specifica chi o cosa possa avere accesso ad ogni risorsa del sistema e il tipo di accesso

che è permesso. Un meccanismo di controllo degli accessi media tra l'entità e le risorse del sistema, attraverso due passaggi:

- **Autenticazione:** il sistema deve autenticare l'utente, tipicamente determinando se esso ha accesso o no al sistema;
- **Autorizzazione:** una volta autenticato, è necessario verificare se il tipo di accesso dall'utente è permesso

Il comportamento del controllo degli accessi è influenzato fortemente dal tipo di *policy* utilizzata. Vengono generalmente definite tre tipologie di policy:

- **Discretionary access control (DAC):** controllo degli accessi basato sull'identità del richiedente e sulle regole di accesso. Il termine *discretionary* (discrezionale) rispecchia la possibilità da parte di un'entità, con diritti di accesso, di abilitare l'accesso ad un'altra entità;
- **Mandatory access control (MAC):** controllo che si basa sulla presenza a priori di un insieme di attributi di sicurezza, assegnati sia ai soggetti richiedenti che agli oggetti. Alla richiesta di un accesso, attraverso l'uso di regole di autorizzazione vengono esaminati questi attributi e deciso deve essere concesso o meno;
- **Role-based access control (RBAC):** controllo degli accessi che si basa sulla presenza di ruoli svolgibili dagli utenti nel sistema e da regole che indicano cosa è permesso agli utenti nei ruoli da loro svolti.

All'interno di un'infrastruttura RBAC un ruolo è definito come una *funzione lavorativa* interna all'organizzazione. Invece di essere collegati con le risorse del sistema, in questa tipologia di policy i diritti di accesso sono assegnati ai ruoli, e gli utenti sono associati ad essi, sia dinamicamente che

staticamente. Sia la relazione tra utenti e ruoli che tra ruoli e risorse è molti a molti. Solitamente l'insieme degli utenti è soggetto a cambiamenti, più o meno frequenti a seconda dell'organizzazione, mentre l'insieme di ruoli si mostra molto più statico. Infine si può notare come le risorse e i diritti di accesso associati ad un ruolo siano molto meno soggetti a modifiche. RBAC incorpora molto bene al proprio interno il principio del *minor privilegio*, dato che ad ogni ruolo è assegnato solamente il minor numero di privilegi possibile perché svolga il proprio compito.

Tramite RBAC è possibile quindi formare delle politiche di sicurezza per un'organizzazione, utilizzando delle astrazioni come ruoli, permessi, ecc.. I componenti di un sistema di questo tipo sono:

- **Ruolo:** un compito interno all'organizzazione, dotato di privilegi;
- **Utente:** entità che tenta di accedere alle risorse;
- **Permesso:** un particolare modo di accesso a uno o più oggetti;
- **Operazione:** un'azione su di un oggetto;
- **Sessione:** mappatura tra un utente e un insieme di ruoli attivati per esso;
- **Oggetto:** risorsa che può essere acceduta;

Il modello di riferimento di RBAC è quello *NIST*, mostrato in figura 1.1, che presenta altre particolarità:

- **SSD:** *Static Separation of Duties* (separazione statica dei compiti), cioè la definizione di un insieme di ruoli mutualmente esclusivi, tali che se un utente è assegnato ad un ruolo di un insieme allora non può essere assegnato a nessun ruolo di un altro insieme;

- **DSD:** *Dynamic Separation of Duty* (separazione dinamica dei compiti), limita la disponibilità dei permessi utilizzando dei vincoli sui ruoli, che possono essere attivati all'interno di una sessione utente;
- **Gerarchie dei ruoli:** si utilizza l'ereditarietà per fare in modo che un ruolo implicitamente includa anche i privilegi associati ad un altro ruolo.

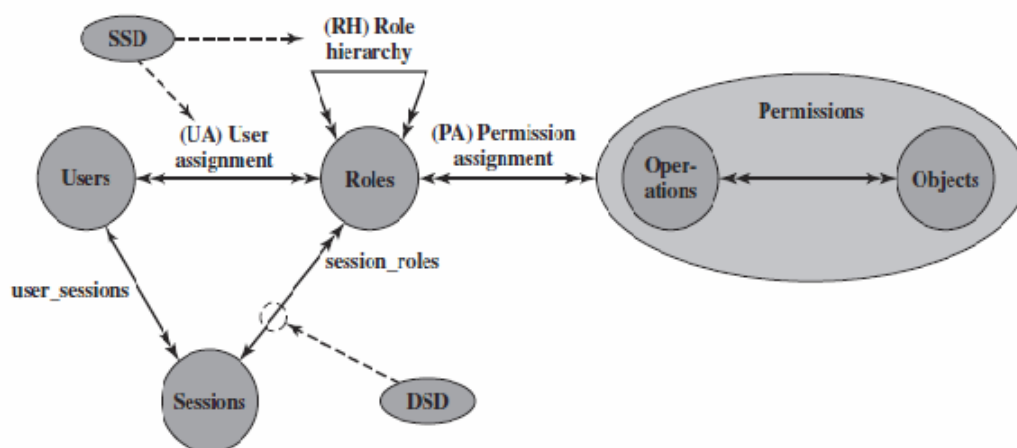


Figura 1.1: Modello RBAC NIST

Infine possiamo affermare come RBAC sia considerato tra gli approcci più promettenti per l'ingegnerizzazione della sicurezza nei sistemi complessi, ed è il motivo per cui la tesi verte sull'implementazione di esso in un sistema *MAS*.

## 1.2 Agent Coordination Context e RBAC

I *MAS*, Multi-Agent Systems (Sistemi multiagente), sono utilizzati molto frequentemente al fine di dare un giusto livello di astrazione per la modellizza-

zione di sistemi complessi. Molti approcci utilizzati per la loro organizzazione si basano sui ruoli, concentrandosi su tre punti:

- *Struttura*: i ruoli presenti;
- *Pattern*: le relazioni esistenti tra i ruoli;
- *Regole*: vincoli dei ruoli.

In questi sistemi l'*organizzazione* e la *coordinazione* sono fortemente collegate tra loro ed interdipendenti, e bisogna affrontarle in maniera unica e coerente. Con organizzazione si intende la struttura e le relazioni tra i ruoli, problemi di tipo statico, mentre con coordinazione si parla dei problemi dinamici, che definiscono quali siano i limiti di un agente all'interno del sistema. I due aspetti sono connessi nei MAS e molti problemi possono essere affrontati prendendoli in considerazione nella loro unione. In TuCSoN, che verrà trattato nella prossima sezione, per venire incontro a questa visione è stato introdotto il concetto di ACC (Agent Coordination Context), astrazione che definisce la struttura e le regole organizzative di una società ad agenti, rendendole accessibili, amministrabili e modellabili a tempo di esecuzione da parte di esseri umani e di agenti. L' **Agent Coordination Context** è introdotto come un modo per modellare i **MAS** dal punto di vista dell'agente rispetto all'ambiente in cui è inserito. Si tratta quindi di un *artefatto di mediazione* associato con un agente, e quello che rappresenta è un **contesto**, astrazione utilizzata per modellare ed ingegnerizzare l'ambiente ed i suoi effetti sulle interazioni e comunicazioni tra entità attive. Quindi, un ACC:

- Descrive l'ambiente al cui interno un agente può interagire;
- Abilita e gestisce le interazioni tra l'agente e l'ambiente.

La natura di un ACC può essere inoltre meglio intesa tramite la metafora della *stanza di controllo*, che può essere ritrovata in [Omi02].

Due fasi distinte caratterizzano le dinamiche di un ACC: negoziazione ed utilizzo. Un ACC deve essere negoziato dagli agenti per poter cominciare a svolgere attività all'interno di un'organizzazione. L'agente lo richiede specificando quale ruolo deve essere attivato. Se le regole lo permettono, un ACC viene creato, configurato a seconda dello specifico ruolo attivato, ed infine rilasciato all'agente. Quest'ultimo può utilizzarlo per interagire con l'ambiente dell'organizzazione, in accordo con le limitazioni imposte all'attivazione.

In un'architettura di tipo RBAC, l'ACC si comporta come l'entità che allo stesso tempo abilita e vincola le azioni dell'agente, secondo i permessi che sono stati assegnati al ruolo attivato nella negoziazione. Se gli agenti diventano così gli utenti dei sistemi, gli ACC ne costituiscono le sessioni aperte.

La tesi non vuole dilungarsi per quanto riguarda la trattazione teorica, per un approfondimento si rimanda alla consultazione di: [ORV05] [ORV03] e [RVO06]

### 1.3 TuCSoN e ReSpecT

Al fine di definire un'implementazione del modello RBAC proposto, è stato scelto come caso di studio l'utilizzo del modello di coordinazione **TuCSoN** e del linguaggio di coordinazione **ReSpecT**. In questa sezione sono stati introdotti per fornirne una conoscenza di base, utile per meglio capire le successive fasi di sviluppo del progetto.

### 1.3.1 TuCSoN

**TuCSoN** (Tuple Centres Spread over the Network) è un'infrastruttura *MAS* (Multi-Agent System) general-purpose, che permette la coordinazione e la comunicazione tra agenti, basandosi sull'utilizzo di *centri di tuple*, spazi di tuple programmabili, nei quali gli agenti possono accedere, leggere e consumare tuple. Questi centri di tuple sono localizzati in nodi distribuiti nella rete ed interconnessi tra loro, formando così lo *spazio di coordinazione*. Sono tre le entità che modellano l'infrastruttura:

- **Agenti**: le entità TuCSoN sparse nella rete che devono essere coordinate. Sono intelligenti, proattive e mobili;
- **Centri di tuple ReSpecT**: costituiscono il mezzo tramite il quale avviene la coordinazione. Sono passive e ancorate al nodo;
- **Nodi**: entità TuCSoN che rappresentano l'astrazione topologica di base, contiene al suo interno i centri di tuple.

Un sistema TuCSoN è quindi formato da un insieme di centri di tuple ed agenti che collaborano in modo coordinato in un insieme di nodi distribuiti, dove è in esecuzione un servizio TuCSoN. Un nodo è costituito da un dispositivo connesso alla rete, su cui è presente il servizio, e da una porta di rete dove il dispositivo si pone in ascolto delle richieste. Più nodi possono essere presenti sullo stesso dispositivo, in ascolto su porte di rete differenti. Gli agenti si comportano come entità **proattive**, comunicando attraverso l'utilizzo di tuple, mentre centri di tuple invece assumono un comportamento **passivo**, fornendo uno spazio per gli agenti, e anche **reattivo**, dato che possono essere programmati per rispondere alle azioni svolte sulle tuple.

I centri di tuple forniscono due spazi di tuple distinti:

- Uno spazio per la comunicazione tramite tuple;
- Uno spazio per l'impostazione del comportamento del centro di tuple tramite tuple di specificazione, affrontato nella sezione seguente.

Le entità da coordinare necessitano di operazioni di coordinazione per interagire con il medium di comunicazione: queste operazioni sono costruite tramite il linguaggio di coordinazione TuCSoN e le sue primitive di coordinazione.

Il linguaggio fornisce nove diverse primitive di coordinazione (  $T$  indica una tupla,  $TT$  indica un template):

- $out(T)$ :  $T$  viene inserita nel centro di tuple e, se si ha successo,  $T$  viene restituita;
- $in(TT)$ : viene ricercata una tupla che *unifichi* con  $T$ ; se viene trovata, allora viene rimossa e restituita, altrimenti l'esecuzione viene fermata fino a che non è presente una tupla unificante;
- $inp(TT)$ : come la primitiva precedente, solo che se non viene trovata una tupla unificante l'esecuzione non viene sospesa, ma viene ritornata  $T$ ;
- $rd(TT)$ : viene ricercata una tupla che *unifichi* con  $T$ ; se viene trovata, allora viene restituita ma non rimossa, altrimenti l'esecuzione viene fermata fino a che non è presente una tupla unificante;
- $rdp(TT)$ : come la primitiva precedente, solo che se non viene trovata una tupla unificante l'esecuzione non viene sospesa, ma viene ritornata  $T$ ;
- $get(TT)$ : restituisce tutte le tuple come una lista;



- **set(T)**: sovrascrive il centro di tuple con una lista di tuple;
- **no(TT)**: viene ricercata una tupla che unifichi con TT; se non viene trovata, allora l'operazione ha successo e viene restituita TT, altrimenti l'operazione è sospesa fino a che esiste una tupla unificante;
- **nop(TT)**: come l'operazione precedente, con la differenza che in caso di fallimento l'operazione non viene sospesa e viene invece ritornata la tupla con cui TT unifica.

TuCSon fornisce inoltre nove primitive di **meta-coordinazione**, utilizzate per costruire delle operazioni di meta-coordinazione. Queste permettono agli agenti la scrittura, lettura ed il consumo di tuple di specificazione ReSpecT. Queste nove primitive combaciano con le primitive di coordinazione appena presentate, con l'aggiunta però del postfisso *\_s*.

### 1.3.2 ReSpecT

I centri di tuple sono programmabili attraverso il linguaggio **ReSpecT** (**R**eaction **S**pecification **T**uples), offrendo così la possibilità di definire delle computazioni all'interno del centro di tuple, chiamate **reazioni**, e di associarle agli eventi di comunicazione in entrata o in uscita. tramite *tuple di specificazione* che determinano le reazioni agli eventi.

ReSpecT presenta una duplice natura:

- *Dichiarativa*: trattandosi di un linguaggio di *specificazione*, rende possibile l'associazione dichiarativa tra eventi e reazioni, grazie alle tuple logiche dette *tuple di specificazione*;

- *Procedurale*: essendo inoltre un linguaggio *reattivo*, ogni reazione può essere definita in modo *procedurale* come una serie di reazioni logiche, ognuna delle quali può avere successo o fallire.

Le reazioni che formano il comportamento del centro di tuple presentano la forma: `reaction(E,G,R)`.

- **E**: *testa* della reazione, corrisponde all'**evento** che porta allo scatenarsi di una o più reazioni;
- **G**: *guardia* che contiene delle condizioni che devono essere soddisfatte per l'avvio delle reazioni;
- **R**: il vero e proprio *corpo* della reazione, contiene i predicati `ReSpecT` che definiscono il comportamento da seguire.

Sebbene le reazioni innescate possano essere un qualsiasi numero, ciò che viene percepito dalle entità esterne è un'unica *transizione di stato* del centro di tuple, che ha successo solo se tutte le reazioni innescate hanno successo. Queste sono eseguite in maniera sequenziale con una semantica transazionale, da cui deriva che un fallimento non ha effetti sul centro di tuple `ReSpecT`, e la reazione viene abortita in modo **atomico**: tutti i cambiamenti effettuati fino a quel momento, dal momento della gestione dell'evento, vengono annullati.

Ogni volta che viene invocata una primitiva, sia da parte di un agente che da un centro di tuple, si hanno tre diverse fasi:

- **Invocation**: viene generato un evento `ReSpecT` che raggiunge il centro di tuple dove è inserito in una *input queue* (`InQ`), in modo ordinato tramite una politica *FIFO*;

- **Triggering:** si passa alla successiva fase non appena il centro di tuple si trova in uno stato *idle*, cioè non vi è nessuna reazione in esecuzione in quel momento. Il primo evento presente nella coda viene spostato tra le richieste che devono essere servite. A questo punto, le reazioni alla fase di invocazione dell'evento sono invocate e poi eseguite in un ordine **non deterministico**, scatenando a loro volta eventi in output o ulteriori reazioni;
- **Completion:** una volta esaurite le reazioni si passa all'ultima fase. Questa a sua volta può portare allo scatenarsi di ulteriori reazioni associate alla fase di completion dell'invocazione originaria. Una volta esaurite tutte le reazioni, viene inviata la risposta all'agente sorgente.



# Capitolo 2

## Analisi del progetto

Il lavoro svolto pone le sue basi su uno studio preliminare effettuato sulla tesi del dottor Galassi, il quale si è posto come obiettivo lo studio dell'integrazione di servizi di sicurezza del modello *RBAC* nei *sistemi multiagente*, più nello specifico in TuCSoN. Risulta quindi necessaria un'esaminazione ed uno studio approfonditi su di esso, con lo scopo di ottenere un solido ed avanzato punto di partenza su cui potere impostare il lavoro.

### 2.1 Presentazione lavoro di Galassi

Dal lavoro esaminato traspare come, al fine di riuscire a modellare *RBAC* all'interno dei sistemi *MAS*, sia necessario considerare due diverse dimensioni:

**Autenticazione.** Si vuole definire **come** gli agenti vengano identificati quando cercano di entrare nell'organizzazione. Corrisponde alla prima fase dell'ingresso nel sistema ed avviene durante la negoziazione di un *ACC*. L'enfasi principale viene posta sull'utilizzo di determinati protocolli al fine di garantire una sicura verifica dell'identità. La scelta ultima è ricaduta sul-

l'offrire la possibilità all' amministratore del nodo di definire più metodi di autenticazione tra:

- Nessuna autenticazione
- **TLS** (*Transport Layer Security*), evoluzione del protocollo **SSL**
- **TLS con mutua esclusione**

**Autorizzazione.** Il fine è **cosa** gli agenti siano autorizzati a fare. All'interno dell'organizzazione sono presenti dei *ruoli* a cui sono associati delle *politiche* di controllo degli accessi, i quali indicano per un agente che detiene un ruolo quali siano i *permessi* a lui garantiti. Si crea così un legame agente-ruolo all'interno della *sessione*, concetto racchiuso all'interno dell'ACC.

E' in quest'ultima direzione che si sviluppa il nostro interesse in questo studio. La presenza di politiche di autenticazione risulta necessaria al fine di garantire l'adeguato livello di sicurezza nel sistema, ma è stato scelto di escluderle dalla trattazione poiché si tratta di un problema di cui si è ampiamente trattato e studiato nella letteratura, e risulta essere un concetto ortogonale al dominio applicativo. L'istanziamento di una politica di autorizzazione invece mostra una dipendenza molto forte rispetto ad esso, per cui deve essere studiata *ad hoc* per riflettere le necessità del caso particolare, nel nostro caso TuCSoN.

### 2.1.1 Modello RBAC proposto

Nella trattazione studiata, i modelli RBAC sono stati incorporati all'interno dei MAS data la loro capacità di inquadrare i problemi di sicurezza; le fondamenta della struttura del modello organizzativo sono composte dalle

astrazioni trattate nell'introduzione: ruoli, sessioni, permessi, politiche, ecc. Parte predominante ovviamente è anche costituita dalle regole organizzative, descrivibili da due relazioni fondamentali:

- Relazioni agente-ruolo: un agente può svolgere un certo ruolo?
- Relazioni inter-ruoli: specifica le relazioni di dipendenza esistenti tra diversi ruoli

Adottare il modello RBAC in TuCSon offre la possibilità di descrivere le **strutture** e le **regole** utilizzando delle *tuple* immesse in specifici centri di tuple dell'organizzazione. In questo modo è resa possibile un'ispezione ed una modifica dinamica delle strutture e delle regole, sia da parte di agenti che da parte di esseri umani.

Di particolare interesse sono le fasi che portano alla partecipazione di un agente in un'organizzazione:

- Ammissione del ruolo: è verificato se un agente è adibito allo svolgere uno specifico ruolo nell'organizzazione
- Attivazione del ruolo: partecipazione attiva dell'agente nell'organizzazione, svolgimento di un ruolo

Per la gestione di queste fasi entra in gioco l' **Agent Coordination Context**, un concetto utilizzato per modellare l'organizzazione dei MAS integrandola con i problemi di coordinazione. Gli ACC costituiscono degli *artefatti* di mediazione individuale, in grado di *abilitare* e *vincolare* le interazioni; si tratta quindi di un'astrazione utilizzata ai fini della modellazione di RBAC in TuCSon. Infatti un agente che voglia interagire con l'organizzazione deve come prima cosa **negoziare** un ACC, specificando quali ruoli

desidera compiere. Questa negoziazione può risultare in un successo, con la creazione dell'ACC e la sua abilitazione allo svolgimento dei ruoli richiesti, o in un fallimento, nel qual caso l'agente non potrà interagire in un alcun modo con le risorse.

In caso positivo l'agente può usare l'ACC come interfaccia per interagire con gli altri agenti o con l'ambiente dell'organizzazione. Uno stesso agente può possedere più ACC nello stesso momento, uno per ogni organizzazione, ed ognuno di questi modella lo spazio d'azione dell'agente, cioè le operazioni che può eseguire.

Nella pratica, il comportamento che è stato modellato per l'utilizzo di questi ACC consiste nell'acquisizione iniziale di un *contesto* vuoto, che offre la possibilità di poter essere rilasciato o di *negoziare* un ruolo. Quando almeno un ruolo è stato attivato, allora il contesto viene aggiornato e l'agente può cominciare a svolgere il ruolo richiesto utilizzando l'ACC.

Lo svolgimento di un ruolo è vincolato dalle politiche ad esso associate. L'ACC utilizzato filtra le azioni permesse verificandone l'ammissibilità utilizzando un motore Prolog operante sulla teoria delle politiche dei ruoli. Questa teoria è formata da un insieme di regole del tipo:

```
can_do(CurrentState; Action; NextState) : Conditions.
```

L'azione *Action*, una qualche operazione sul nodo, può essere svolta solamente se il ruolo si trova nello stato *CurrentState* e le condizioni *Conditions* sono rispettate. Se questo avviene allora l'azione viene svolta e il ruolo passa allo stato *NextState*

### 2.1.2 Topologia

Nel lavoro analizzato, ampio spazio è lasciato all'implementazione del modello topologico dei nodi TuCSoN.



I centri di tuple sono ospitati in nodi TuCSoN distribuiti nella rete. E' possibile distinguere due diverse tipologie di nodi:

- **Posti**: Dove possiamo trovare i centri di tuple utilizzati dall'organizzazione per i propri bisogni.
- **Gateway**: Utilizzati invece per i centri di tuple adibiti a fini amministrativi.

Si può pensare quindi ad una organizzazione come composta da un insieme di **domini**, strutturati da un nodo *gateway*, per l'amministrazione del dominio, e da nodi *posti*, per gli obiettivi applicativi. Ai gateway è data la responsabilità di gestire l'entrata ed uscita degli agenti, l'attivazione dei ruoli e la struttura topologica; tutto questo avviene nel centro di tuple **\$ORG**. All'interno di questo sono presenti tutte le tuple necessarie all'amministrazione dell'organizzazione.

### 2.1.3 TrustBAC

Il modello RBAC si rivela adatto al controllo degli accessi per quanto riguarda utenti la cui identità è conosciuta a priori. L'ambiente in cui TuCSoN si muove si rivela però essere distribuito e aperto, il che comporta una forte dinamicità degli utenti e al tempo stesso la forte mancanza di conoscenza delle loro identità a priori. Si rivela quindi necessario un qualche modello basato su credenziali, comunemente realizzato come un controllo di username e password, sebbene emergano problematiche anche in questo caso.

Infatti, l'uso di credenziali non permette di accertarsi che queste siano state ottenute in modo malevolo, di collegare un utente al suo comportamento ed azioni e non viene conservato uno storico di queste dal momento in cui

sono rilasciate. Non vi è quindi alcun modo di punire un comportamento negativo o di premiarne uno positivo.

Per ovviare a questa problematica, è stato introdotto il modello **Trust-BAC**, il quale aggiunge i *livelli di fiducia*; ad ogni utente ne viene assegnato uno, sulla base delle azioni da lui intraprese, delle sue caratteristiche ed ovviamente delle credenziali presentate. Ad ognuno di questi livelli è mappato un insieme di ruoli e privilegi. Per cui ad ogni cambiamento del livello di fiducia di un utente corrisponde un cambiamento nell'insieme di privilegi a cui è possibile l'accesso.

Il livello di fiducia si basa su tre componenti:

- **Conoscenza:** Quello che si conosce dell'utente (informazioni generali, credenziali, ecc...).
- **Comportamento:** Deriva dall'esaminazione degli eventi riguardanti l'utente, ponendo un peso maggiore sulle azioni più recenti.
- **Raccomandazioni dall'utente:** Hanno un peso differente a seconda di chi le fornisce.

Il livello di fiducia corrisponde ad un numero compreso tra i valori -1.0 e 1.0 che indica generalmente:

- $Fiducia < 0.0$ : Utente fidato.
- $Fiducia = 0.0$ : Utente neutro.
- $Fiducia > 0.0$ : Utente non fidato.

Anche il modello *TrustBAC* presenta però alcune problematiche:

**Principio del minor privilegio non rispettato** . Alla richiesta di accesso all'organizzazione, ad un agente viene affidato un insieme di privilegi che si basa sul suo livello di fiducia, e non sulle sue reali necessità. Se da un lato questo può essere positivo poiché permette ad un agente di lavorare senza sapere a priori il ruolo necessario, dall'altro non segue il principio di *least privilege*, poiché potrebbe dare ad un agente sconosciuto un insieme di privilegi maggiore di quelli necessari.

**Soluzione:** Rimozione dell'assegnazione automatica dei ruoli, questi vengono assegnati dopo una richiesta specifica o una richiesta per lo svolgimento di un'operazione, nel caso esista un ruolo che la permetta ed il livello di fiducia sia adeguato. Se esiste, viene attivato il ruolo con il minor privilegio.

**Aggiramento sistema di ricompensa TrustBAC** . Un agente non autorizzato a ricevere un insieme di privilegi potrebbe cercare di ottenere un livello di fiducia maggiore semplicemente compiendo una buona azione in maniera ripetuta.

**Soluzione proposta:** Modifica della funzione del calcolo del livello di fiducia, in modo tale che sia *slow-to-increase* (lenta ad aumentare) e *fast-to-decrease* (rapida a diminuire). Inoltre si fa dipendere il permesso per svolgere un ruolo direttamente dalle credenziali presentate, non solo dal livello di fiducia lui associato; in questo modo l'attivazione di un ruolo richiede che l'agente sia qualificato e fidato.

## 2.2 Struttura RBAC di base in TuCSoN

Necessaria è poi un'analisi del supporto già presente all'interno di TuCSoN all'integrazione di una struttura RBAC. Il modello attuale presenta un supporto ridotto, sebbene ancora non lo implementi.

Di fondamentale importanza è la creazione, all'avvio del nodo, di un centro di tuple **\$ORG** adibito a tuple e reazioni che governano l'aspetto amministrativo dell'organizzazione. Qui infatti verranno poste tutte quelle tuple e reazioni che descrivono le strutture e le regole che governano il controllo sugli accessi che vogliamo implementare.

L'analisi si è concentrata poi sullo studio del file *boot\_spec.rsp*, collocato nel package *alice.tucson.service.config*, al fine di valutare se potesse essere una buona base di partenza. All'avvio del nodo viene inserita la reazione:

---

```
reaction(out(boot), true,  
(  
    in(boot),  
    no(boot_info(_)),  
    current_time(Time),  
    out(boot_info([time(Time)])),  
    out(context_id(0)),  
    out(authorized_agent(_)),  
    out(role(default, 'default role')),  
    out(policy(default, 'default policy')),  
    out(role_policy(default, default)),  
    out(role_assignment(_, _))  
)).
```

---

La reazione viene scatenata all'avvio del nodo, con l'inserimento di una semplice tupla *boot*. Oltre ad impostare informazioni come l'orario di partenza, vengono poste le tuple base per la gestione di una struttura di tipo RBAC. Come è possibile vedere, sono creati un ruolo ed una policy di default, mentre `role_policy(default, default)` indica l'associazione tra questi due elementi. E' possibile rendersi conto di come la base sia quindi presente, ma

non incida minimamente sul corso dell'accesso di un qualsiasi agente; infatti tramite `authorized_agent(_)` indichiamo che qualsiasi nome agente è un identificativo autorizzato all'ingresso, mentre con `role_assignment(,_)` viene indicato che ogni agente può essere associato ad un qualsiasi ruolo. `context_id(0)` viene utilizzato per tenere traccia del valore da assegnare alla successiva richiesta di contesto che ha successo, dopo la quale questo valore viene incrementato.

Al fine di ottenere un contesto, ogni agente deve effettuare una richiesta che può o meno avere successo. Questa richiesta viene rifiutata se

- Tra gli agenti autorizzati non risulta esserci l'agente richiedente:

---

```
reaction( inp(context_request(AgentId,_)),request,
(
  no(authorized_agent(AgentId)),
  out(context_request(AgentId,
    failed(agent_not_authorized))))
).
```

---

- Esiste già una sessione aperta per l'agente:

---

```
reaction( inp(context_request(AgentId,_)),request,
(
  rd(open_session(_,AgentId)),
  no(context_request(AgentId,ok(_))),
  out(context_request(AgentId,
    failed(agent_already_present))))
).
```

---

Mentre ha successo se l'agente che richiede il contesto è autorizzato e non è presente già una sessione associata a lui:

---

```

reaction( inp(context_request(AgentId,_)),request,
(
  rd(authorised_agent(AgentId)),
  in(context_id(Id)),
  Id1 is Id + 1,
  out(context_id(Id1)),
  no(open_session(_,AgentId,_)),
  out(open_session(Id,AgentId,[])),
  out(context_request(AgentId,ok(Id))))
).

```

---

In questo caso viene creata una nuova sessione con un certo identificativo di contesto e una lista vuota di ruoli attivati (le richieste per l'attivazione di ruoli avverranno solo successivamente all'ottenimento di un contesto vuoto).

Così come viene ricevuta una richiesta di attivazione, allo stesso modo un contesto può essere rilasciato attraverso un'ulteriore richiesta inviata dall'agente. Nel caso non vi sia una sessione associata all'agente ed al contesto che si è intenzionati a chiudere si va incontro ad un fallimento:

---

```

reaction( inp(context_shutdown(CtxId,AgentId,_)),request,
(
  no(open_session(CtxId,AgentId,_)),
  out(context_shutdown(CtxId,AgentId,
    failed(no_valid_context))))
).

```

---

Mentre se la richiesta risulta valida allora si procede alla rimozione della

sessione aperta:

---

```

reaction( inp(context_shutdown(CtxId,AgentId,_)),request,
(
  rd(open_session(CtxId,AgentId,_)),
  out(context_shutdown(CtxId,AgentId,ok)))
).

reaction( inp(context_shutdown(CtxId,AgentId,ok)),response,
(
  in(open_session(CtxId,AgentId,_)))
).

```

---

Una volta che un agente ottiene un contesto, che risulta essere vuoto in partenza, è adibito alla richiesta di attivazione di un ruolo:

---

```

reaction(
  inp(role_activation_request(CtxId,RoleId,Result)),request,
(
  in(open_session(CtxId,AgentId,RoleList)),
  rd(role(RoleId,Descr)),
  rd(role_assignment(RoleId,AgentId)),
  out(open_session(CtxId,AgentId,[RoleId|RoleList])),
  rd(role_policy(RoleId,PolicyId)),
  rd(policy(PolicyId,Policy)),
  out( role_activation_request(CtxId,RoleId,ok(Policy))))
).

```

---

La reazione ReSpecT scatenata inizialmente controlla l'esistenza di una sessione associata ad un certo agente e identificativo di contesto. Verifica poi

l'esistenza del ruolo e se sia permessa un'associazione tra questo e l'agente richiedente. Se non ci sono stati problemi di alcun tipo, il ruolo viene aggiunto alla lista dei ruoli associata ad un certo contesto e al richiedente viene ritornata la *Policy* associata al ruolo.

Dopo avere analizzato e testato la struttura preesistente in TuCSon, è risultata una buona base di partenza per la costruzione del nostro sistema. Come è stato presentato, son già presenti reazioni volte a considerare la richiesta e il rilascio di un contesto, attività che si può supporre corrispondano al punto iniziale e finale dell'interazione di un agente con il nodo; inoltre è presa in considerazione anche la richiesta di attivazione di un ruolo, parte estremamente importante nel ciclo di vita di un entità che voglia interagire con un sistema basato su RBAC. La reazione iniziale viene scatenata all'avvio del nodo, precisamente all'inserimento di una tupla *boot* all'interno di *\$ORG*, e può essere utile per inserire tuple che caratterizzino alcune proprietà del sistema RBAC.

## 2.3 Definizione obiettivi progettuali

In seguito all'esaminazione del lavoro presentato dall' Ing. Galassi e dopo un'analisi della base strutturale RBAC già presente all'interno di TuCSon, è necessaria una definizione finale su quale direzione debba seguire la tesi, quali siano gli obiettivi da raggiungere e le funzionalità RBAC da progettare e successivamente implementare.

Quello che principalmente si cerca di attuare, ed è stato il punto di nascita di questa tesi, è la definizione di un sistema che prenda spunto dal lavoro di Galassi, estraendone gli elementi di maggiore importanza e tralasciando tutto quello che non sembri importante, ma risulti conciso, funzionale, chiaro



e il più semplice possibile nel suo utilizzo, al fine di una sua integrazione all'interno di TuCSoN una volta conclusa l'implementazione.

Come prima cosa, risulta importante tenere a mente l'aspetto fondamentale di questo progetto, cioè la sua **praticità**; perciò immaginandone un utilizzo reale è possibile prevedere due scenari possibili:

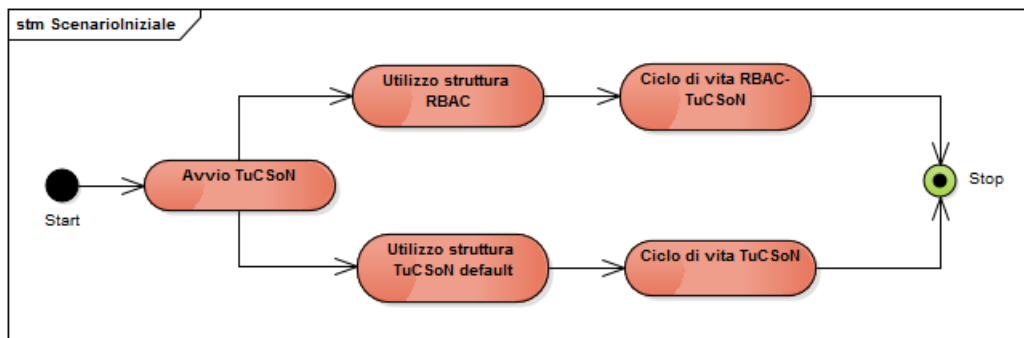


Figura 2.1: Scenari iniziali

Bisogna quindi assicurarsi che il sistema mantenga la possibilità di funzionare anche senza l'installazione di una struttura RBAC, cioè mantenendo tutte le sue caratteristiche precedenti. E' stato scelto quindi di rendere attivo il supporto RBAC solo in seguito ad un'esplicita richiesta da parte dell'organizzazione, in assenza della quale il comportamento rimane quello standard.

Gli obiettivi progettuali di questa tesi sono proposti tramite una trattazione suddivisa in *struttura* e *comportamento* delle varie componenti.

### 2.3.1 Nuova struttura RBAC

Di fondamentale importanza è la definizione del *modello RBAC*, dove saranno contenuti tutti i componenti principali utilizzati nell'organizzazione di un nodo. Si è scelto di procedere con una semplificazione del modello

proposto da Galassi presentato precedentemente, mantenendo solo le parti considerate essenziali per una corretta implementazione. Segue quindi un'esaminazione del nuovo modello, indicando ciò che è stato mantenuto e ciò che invece è stato scelto di scartare.

- **Agente:** entità che cerca di accedere alle risorse protette dal controllo degli accessi
- **Ruolo:** un compito svolgibile all'interno dell'organizzazione;
- **Permesso:** indica un'azione che è possibile effettuare;
- **Policy:** costituisce un insieme di uno o più permessi;
- **Agente autorizzato:** identifica un agente conosciuto dal sistema ed associato ad una determinata classe agente;
- **Agent Coordination Context:** al suo interno è presente il concetto di sessione di lavoro;
- **Classe Agente:** etichetta che specifica un particolare insieme di agenti, assegnabile ai ruoli che essi possono svolgere.

Ad ogni agente quindi possono essere assegnati uno o più ruoli all'interno dell'organizzazione, sempre se autorizzati a svolgerli. Un ruolo costituisce un compito svolgibile ed è assegnato ad una policy. Un ruolo può avere una singola policy, mentre una policy può essere assegnata a diversi ruoli. A sua volta una policy è collegata ad un insieme di permessi, ognuno dei quali specifica una singola operazione eseguibile. Una policy quindi può avere più permessi e allo stesso un permesso può essere associato a differenti policy. Un *ACC* incapsula al proprio interno la mappatura tra un agente ed un suo ruolo. Un agente può disporre di più *ACC*, uno per ogni ruolo attivato,

mentre questi sono assegnati ad una singola entità. Ad ogni ruolo è assegnata anche una classe agente, proprietà che viene associata anche ad ogni agente autorizzato dal sistema, che limita l'attivazione del ruolo solamente alle entità facenti parti della stessa classe agente.

Il controllo sull'esecuzione dei ruoli da parte degli agenti proposto da Galassi, comprendente cioè la definizione delle operazioni ammissibili in base allo stato corrente del ruolo, è stato abbandonato in favore di un più semplice controllo sulle operazioni permesse.

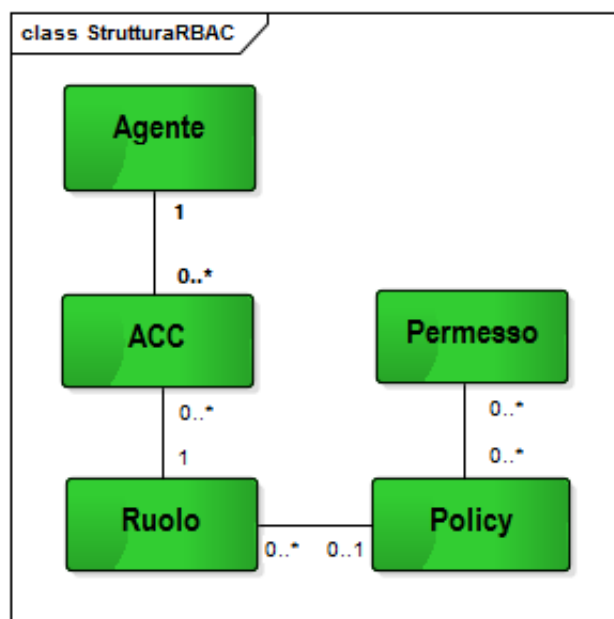
Siccome il modello *TrustBAC* proposto non è stato integrato in questo progetto, il concetto di *livello di fiducia* è stato tralasciato. Sono state tralasciate anche le *proprietà agente*, caratteristiche fornite da un agente al momento dell'entrata nell'organizzazione. E' stato scelto infatti di mantenere la complessità delle informazioni elaborate e memorizzate al minimo al fine di semplificare il lavoro. Al posto delle *politiche* proposte, cioè regole riguardanti l'esecuzione di pattern di azioni, è stato scelto di implementare le policy come un insieme di permessi sulle risorse.

### 2.3.2 Macrostruttura del sistema

E' stato possibile suddividere il sistema in tre differenti sottosistemi principali, su cui potersi concentrare separatamente al fine di semplificare l'analisi. Per ognuno di questi verranno poi identificati i punti dove sarà necessario concentrare l'attenzione.

#### **Agente user.**

Tramite *agente user* è stato identificato tutto ciò che concerne agenti di tipo esterno, cioè non appartenenti all'organizzazione ma che necessitano di



**Figura 2.2:** Struttura RBAC di base

interagire con essa, ed interni, agenti appartenenti all'organizzazione ma che non svolgono attività di amministrazione del nodo.

#### **Problematiche principali:**

- *Accesso:* Di principale importanza in questo caso è regolamentare l'accesso di queste entità alle risorse del sistema. La prima fase consisterà nel ricevimento di un mezzo atto alla negoziazione dei ruoli da poter attivare; ricevuto questo, si potrà procedere alla richiesta di attivazione di un ruolo, la cui riuscita dipenderà dalle politiche di gestione del nodo;
- *Azioni in linea con il ruolo:* a ruoli diversi corrisponderanno permessi diversi, sono quindi necessari dei controlli per garantire che le operazioni compiute siano in linea con i privilegi di cui si è in possesso.

**Agente admin.**

Parliamo di agenti che ottengono il permesso (per esempio tramite credenziali) per l'amministrazione della struttura organizzativa.

**Problematiche principali:**

- *Autenticazione*: L'ottenimento di privilegi amministrativi devono essere rilasciati solo conseguentemente alla verifica dell'identità del richiedente;
- *Manipolazione di struttura e regole*: Un amministratore deve essere in grado di poter cambiare dinamicamente la struttura e le regole che governano l'accesso alle risorse. Dovranno essere definiti quindi degli strumenti adatti a questo scopo, con particolare attenzione al mantenimento di un ambiente **consistente**.

**Nodo.**

Infine in questo sottosistema sono state analizzate tutte quelle problematiche pertinenti l'installazione del nodo nella nuova ottica RBAC.

**Problematiche principali:**

- *Struttura di base*: All'avvio del nodo viene creato il *centro di tuple* contenente le strutture e le regole di nostro interesse; dovremo quindi definire le proprietà RBAC di base del nodo;
- *Proprietà*: Supponendo che non tutte le organizzazioni effettuino un controllo degli accessi identico, deve essere data la possibilità di definire alcune delle proprietà.

### 2.3.3 Comportamento dei sottosistemi

Dopo aver presentato i componenti strutturali principali del dominio applicativo e aver introdotto una suddivisione in sottosistemi al fine di semplificare l'analisi, resta da definire il *comportamento* che la nostra implementazione dovrà seguire.

Risulta più semplice presentare questo comportamento suddiviso tra i tre sottosistemi presentati, per cui verranno analizzati singolarmente.

#### **Nodo.**

La creazione del nodo TuCSoN costituisce senza dubbio il punto iniziale. Esso comprende la creazione, oltre ovviamente all'avvio del nodo, di un centro di tuple denominato *\$ORG* che verrà utilizzato per contenere tutte le tuple necessarie al controllo degli accessi. Come detto precedentemente, lo scenario di default da considerare è l'esecuzione di TuCSoN senza RBAC, per cui è essenziale che le tuple immesse siano uguali o abbiano un comportamento equivalente a quelle già utilizzate nel modello originario.

Alcune proprietà RBAC possono essere definite prima del lancio, e vengono inserite ancor prima dell'inizializzazione della struttura RBAC, poiché non intaccano l'esecuzione di default di TuCSoN. Queste proprietà permettono il controllo su alcuni degli aspetti del funzionamento del nodo, tra cui:

- *Lista dei ruoli attivabili*: Permette di limitare o meno le richieste da parte degli agenti della lista dei ruoli attivabili, filtrati in base alla classe agente.  
Default: la richiesta viene permessa;
- *Autenticazione per admin*: Definisce se sia necessario provvedere delle credenziali per assumere il ruolo di admin del nodo;  
Default: credenziali richieste;

- *Credenziali admin*: Correlata con la precedente, definisce quali debbano essere nome utente e password dell'amministratore.  
Default: se non sono specificate le credenziali, allora non è permesso l'accesso ad un amministratore;
- *Permessi Inspectors*: Indica se siano o meno ammessi gli accessi da parte di Inspector.  
Default: l'introspezione degli Inspector è permessa;
- *Login richiesto*: Rende o meno obbligatorio il login da parte degli agenti per accedere al sistema.  
Default: login non richiesto;
- *Classe agente base*: Può essere indicata una classe agente base da assegnare agli agenti che non forniscono delle credenziali.  
Default: la classe agente base viene impostata tramite un valore presente nel nodo.

In base a come sono state impostate le proprietà verranno inserite delle tuple per riflettere nel nodo \$ORG queste scelte. Nel caso di immissione di credenziali amministrative si dovrà provvedere al criptaggio della password.

#### **Agente admin.**

Un agente il cui scopo è lo svolgimento di un ruolo di amministrazione deve per prima cosa ottenere uno strumento che lo renda in grado di effettuare operazioni sul centro di tuple \$ORG. Per fare questo, è necessario come prima cosa che la sua identità sia verificata in qualche modo, usualmente tramite l'invio e la verifica di credenziali (nel nostro caso nome utente e password). Una volta ricevuta l'autorizzazione, all'agente viene assegnato un ACC adatto allo svolgimento dei suoi compiti.

Compito dell'admin è ora la formazione della struttura RBAC desiderata, compiuta tramite la definizione di tutti i componenti base come ruoli, policy, ecc... In parallelo a questo è possibile andare a modificare dinamicamente le proprietà che sono state impostate all'avvio, le quali sono state trattate parlando del nodo.

Infine l'amministratore può inserire un insieme di agenti autorizzati all'ingresso nel sistema, definendone le credenziali e la classe di agente di appartenenza.

Una volta completata questa fase, si può procedere con l'installazione della struttura all'interno del centro di tuple, tramite l'utilizzo dell'ACC ottenuto in precedenza, attivando così il supporto per il controllo degli accessi basato su ruoli.

Questa installazione prenderà luogo tramite l'inserimento delle informazioni necessarie nel centro di tuple, sotto forma di tuple riguardanti l'aspetto strutturale:

- *Ruolo*: una tupla per ogni ruolo, accompagnata da una descrizione dello stesso e dalla classe agente necessaria per l'utilizzo;
- *Credenziali ruolo*: nel caso un ruolo necessiti di credenziali per essere ottenuto, queste sono contenute in un'ulteriore tupla;
- *Policy*: una per ogni policy presente, insieme alla lista dei permessi associati. Questa potrà anche essere vuota, come per esempio nel caso dell'introduzione di una policy di default che non permette operazioni;
- *Associazione ruolo-policy*: come è stato detto, ad ogni ruolo deve essere associata una policy. Questa tupla si fa carico di sottolineare questa associazione;



- *Agente autorizzato*: è presente una tupla per ogni agente che l'amministratore decide di autorizzare, contenente le credenziali e la classe agente di appartenenza.

Ovviamente il compito dell'agente amministratore non si interrompe qui, dato che anche durante l'esecuzione del nodo può intervenire dinamicamente per modificare le regole o la struttura del controllo degli accessi.

È inoltre possibile rimuovere l'infrastruttura RBAC dal nodo, il che comporta un ritorno alle politiche di default di TuCSon.

#### **Agente user.**

Infine trattiamo gli agenti definiti user, con cui identifichiamo gli agenti che non possiedono privilegi amministrativi, ma che richiedono un accesso alle risorse dell'organizzazione.

Come detto in precedenza, il processo di accesso nel sistema passa attraverso due fasi distinte:

- *Negoziazione*: Per prima cosa viene richiesto dall'agente un oggetto che sarà utilizzato per la negoziazione dei ruoli attivabili; questo strumento sarà sempre concesso e non permette all'entità di compiere operazioni sul nodo, tranne la richiesta di attivazione di un ruolo o il tentativo di effettuare login, nel caso si sia in possesso di credenziali;
- *Attivazione ed utilizzo dei ruoli*: Effettuata una richiesta per un ruolo con successo, viene fornito all'agente un ACC adibito solo allo svolgimento delle operazioni concesse dai privilegi del ruolo acquisito.

Dopo la fase di negoziazione è possibile effettuare una richiesta di login presentando le credenziali in possesso; se le credenziali sono verificate correttamente, si otterrà la classe agente corrispondente al proprio utente, che

permette l'attivazione dei ruoli a cui è associata quella determinata classe. Il login non è tuttavia necessario, ma senza di esso la classe agente utilizzata per le richieste risulterà essere quella base del nodo, rendendo quindi impossibile l'attivazione di ruoli associati ad una diversa classe agente. Tramite lo strumento di negoziazione è possibile richiedere l'attivazione di più ruoli; ognuna di queste attivazioni, ovviamente se conclusa con successo, porta alla creazione di un nuovo ACC utilizzabile dall'agente.

La negoziazione per un ACC è possibile tramite due differenti modalità:

- *Attivazione ruolo*: Si richiede l'attivazione di un ruolo particolare fornendone il nome. E' possibile richiedere la lista dei ruoli attivabili correlati dalle politiche ad essi associati;
- *Attivazione permessi*: In questo caso la richiesta contiene i permessi che si vuole utilizzare, uno o più di uno, e se esiste un ruolo che soddisfa la richiesta ed è utilizzabile allora viene attivato. Nel caso ne siano disponibili più di uno, allora viene scelto quello che offre il minor numero di privilegi.

Tenendo in mente l'obiettivo di un corretto funzionamento anche senza la presenza di una struttura RBAC, viene data ad un agente la possibilità di attivare un ruolo di default, tramite la quale semplicemente verrà fornito un ACC standard da utilizzare, ovviamente prima verificando che non sia stato installato un controllo degli accessi di tipo RBAC.

# Capitolo 3

## Progetto

### 3.1 RBAC

#### 3.1.1 Struttura RBAC

La struttura RBAC implementata prende spunto da quella proposta da Galassi, ma il tutto è stato volutamente semplificato in modo da ottenere come risultato un modello semplice e funzionale al nostro scopo. Le classi implementate sono quelle risultanti dall'analisi, presentate nel diagramma che segue insieme alle interfacce e contenute nel package *alice.tucson.rbac*

Il numero delle classi e la complessità generale sono stati drasticamente ridotti rispetto al progetto di partenza, tralasciando infatti concetti come *Society* e *SocietyClass*. Come è naturale, il livello di dettaglio imponibile alla struttura RBAC è nettamente meno approfondito, ma da questo risulta un'implementazione di molto più facile comprensione ed utilizzo.

La classe **TucsonRBAC** rappresenta *in toto* l'organizzazione, presentando come attributo una stringa contenente il nome dell'organizzazione. Ogni istanza della classe contiene al suo interno:

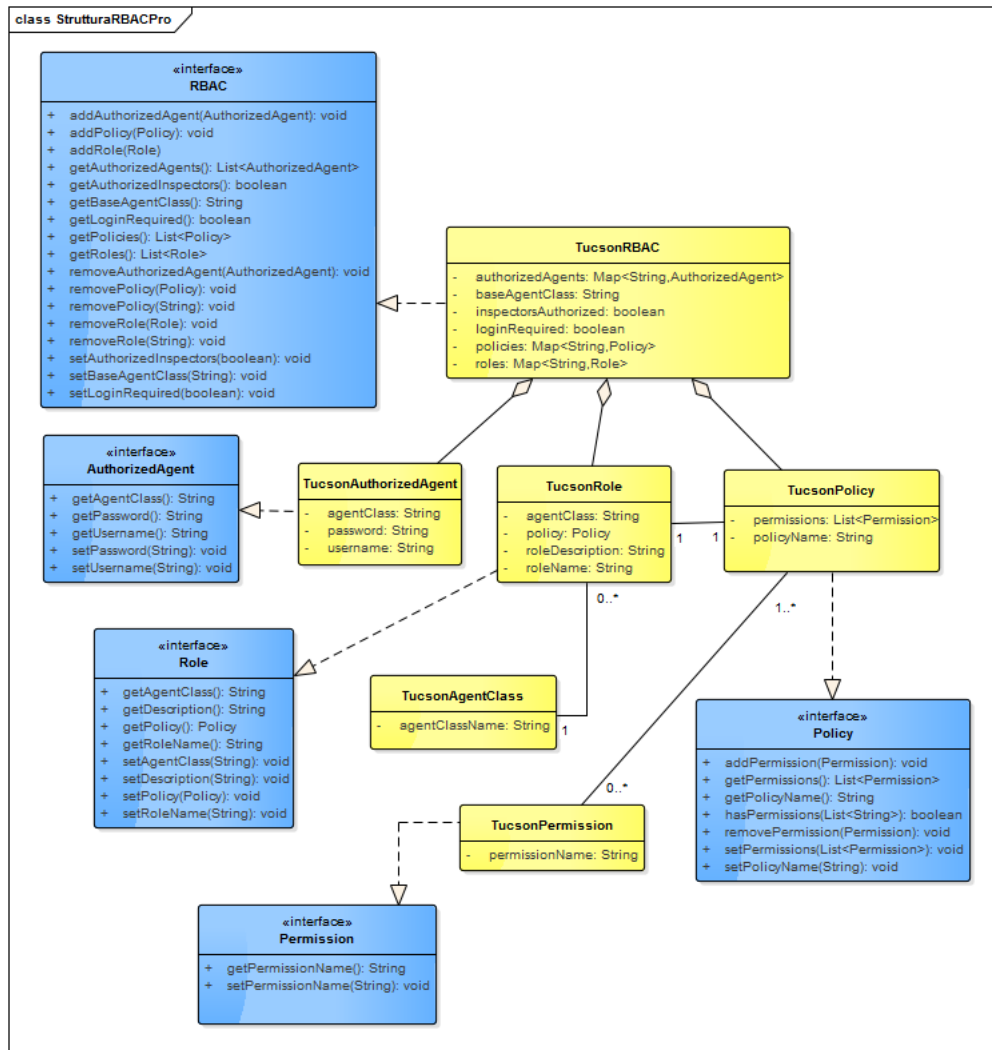


Figura 3.1: Diagramma delle classi RBAC

- Un insieme di *classi ruolo* istanze di **TucsonRole** in *roles*;
- Un insieme di *classi policy* istanze di **TucsonPolicy** in *policies*;
- Un insieme di *classi agente* istanze di **TucsonAuthorizedAgent** in *authorizedAgents*.

La classe **TucsonRole** rappresenta un ruolo all'interno del modello RBAC-MAS e contiene quattro attributi:

- **roleName:** stringa privata che contiene il *nome* associato al ruolo. E' un attributo obbligatorio per la classe;
- **roleDescription:** string privata che contiene la *descrizione* del ruolo. Non è obbligatoria e nel caso non sia inserita viene inserita una descrizione di default;
- **policy:** istanza della classe *TucsonPolicy*, che identifica la *policy* associata al ruolo. E' necessaria per definire i permessi associati al ruolo;
- **agentClass:** stringa privata che contiene la *classe agente* associata al ruolo, obbligatoria per vincolare l'attivazione del ruolo solo per gli agenti autorizzati. Nel caso non sia inserita, viene immessa la classe agente di default del nodo, cioè quella che assegna i privilegi inferiori.

La classe **TucsonPolicy** rappresenta una policy all'interno del sistema che può essere associata a più ruoli, contiene al suo interno due attributi:

- **policyName:** stringa privata che contiene il *nome* della policy. Attributo obbligatorio per la classe;

- **permissions:** lista privata con l'insieme dei *permessi* associati ad una policy. Questo insieme può essere vuoto, il che indica che la policy non dà nessun privilegio al ruolo a cui è assegnata.

La classe **TucsonPermission** rappresenta un permesso per una particolare azione eseguibile nel sistema. Presenta solo l'attributo privato **permissionName**, stringa che identifica l'*operazione* per cui è creato il permesso.

La classe **TucsonAuthorizedAgent** rappresenta un agente già conosciuto dal sistema, rappresentato da:

- **agentClass:** stringa privata che indica il nome della *classe agente* del sistema assegnata all'agente. È obbligatoria e nel caso non venga fornita viene assegnata all'agente la classe agente di default;
- **username:** stringa privata che contiene il *nome utente* assegnato all'agente;
- **password:** stringa privata che contiene la *password* assegnata all'agente.

### 3.1.2 Struttura RBAC in \$ORG

Il centro di tuple adibito alla gestione degli accessi alle risorse è \$ORG, creato in automatico all'avvio del nodo TuCSoN. E qui sono conservate sotto forma di tuple tutte le informazioni necessarie al corretto funzionamento. Per cui le proprietà e le strutture RBAC create sono memorizzate come tuple ed inserite sia alla creazione sia durante l'esecuzione del nodo.

Analizziamo le tuple che saranno inserite per ogni componente:

- **TucsonRole:** `role(Nome, Descrizione, ClasseAgente)`.

- **TucsonPolicy:** `policy(Nome, [Permission1, Permission2, ...])`.  
Dato che una policy può contenere più permessi, questi sono stati inseriti in una lista di stringhe. La policy inserisce nella tupla una stringa per ogni permesso a cui è associata;
- **TucsonRole-TucsonPolicy:** `role_policy(NomeRuolo, NomePolicy)`.  
E' presente una tupla per ogni associazione tra un ruolo ed una policy;
- **TucsonAuthorizedAgent:** `authorized_agent(Credenziali, ClasseAgente)`.  
Le credenziali sono formate dal nome utente unito alla password, dopo essere stata appositamente criptata.

### 3.1.3 Proprietà RBAC

Insieme alla struttura, per il corretto funzionamento del sistema e per conferire una maggiore personalizzazione del controllo degli accessi, sono state progettate alcune proprietà configurabili del sistema, come descritto nella fase di analisi. La configurazione di queste proprietà si rifletterà anche in questo caso nel centro di tuple \$ORG, dove saranno contenute delle tuple che le descrivono. In questo modo è possibile adeguare il comportamento del nostro sistema in base alle esigenze dell'organizzazione.

- **Lista dei ruoli:** un agente che tenta di accedere al sistema può non conoscere in anticipo i ruoli che sono stati messi a disposizione per l'accesso. Abilitando questa proprietà si rende possibile la richiesta per la lista dei ruoli a cui un agente può avere accesso. Quindi diversi agenti otterranno liste differenti in base alla propria classe agente.

Tupla: `list_all_roles(yes/no)`.

Comportamento di default: permessa;

- **Credenziali amministratore:** proprietà fondamentale per permettere l'identificazione ed autorizzazione di amministratori, e quindi permettere modifiche a struttura e proprietà in modo controllato.  
Tupla: `role_credentials(admin_role(Credenziali))`.  
Comportamento di default: nessun ruolo amministrativo;
- **Inspector abilitati:** tramite questa proprietà può essere permesso/negato l'accesso al centro di tuple \$ORG agli Inspector. Questa proprietà può essere molto utile quando si vuole garantire una maggiore riservatezza sul comportamento interno della struttura organizzativa.  
Tupla: `authorized_inspectors(yes/no)`.  
Comportamento di default: inspector autorizzati;
- **Login richiesto:** data la natura distribuita di TuCSon, risulta molto probabile l'accesso da parte di agenti di cui non siamo a conoscenza. Abilitando questa proprietà rendiamo l'accesso all'organizzazione possibile solo per gli agenti conosciuti che presentano credenziali valide.  
Tupla: `login_required(yes/no)`.  
Comportamento di default: login non richiesto;
- **Classe agente di default:** se ruoli o agenti sono inseriti senza una classe agente, o se un agente non autorizzato vuole ottenere accesso al sistema, allora viene loro assegnata una classe agente di default. In questo modo è possibile definire un livello base per gli agenti che non forniscono delle credenziali.  
Tupla: `default_agent_class(NomeClasseAgente)`.  
Comportamento di default: valore preimpostato assegnato alla classe agente di default.



### 3.1.4 Identificazione agenti ed ACC

Nella versione originale di TuCSoN un agente che interagisce con il nodo, tramite ovviamente l'utilizzo di un ACC rilasciato in precedenza, viene identificato unicamente dall'id fornito nel momento della richiesta di contesto. L'utilizzo di più ACC da parte di uno stesso agente porta ad errori nell'esecuzione: come verrà mostrato nella sezione 3.2.4, il comportamento interno di TuCSoN fa in modo che una richiesta di apertura di una sessione per un id già presente in una sessione porti ad un fallimento.

Una sessione all'interno del nodo è salvata nel formato:

```
open_session(ContextId, AgentAid, RoleList)
```

ContextId rappresenta un numero intero progressivo assegnato alle sessioni, AgentAid l'identificativo dell'agente detenente a sessione e RoleList la lista dei ruoli attivati. Come da analisi, ad ogni ruolo attivato con successo da un agente è creato un nuovo contesto. È necessario quindi nella nostra implementazione poter identificare separatamente le varie associazioni tra agenti ed ACC, in modo tale da poterle gestire separatamente.

Una possibile soluzione è quella di estendere l'impianto di identificazione delle sessioni in TuCSoN, inserendo la necessità di avere un secondo elemento da abbinare all'identificativo del'agente. Questo secondo elemento potrebbe essere un qualcosa di semplice come il nome del ruolo attivato o una qualche altra stringa (orario di attivazione, nome agente concatenato ad un valore incrementale, ecc...). È possibile che due agenti che stanno tentando un'attivazione abbiano lo stesso identificativo, mentre in base al metodo scelto per ottenere il secondo elemento si possono avere più o meno possibilità di collisione. Esiste per cui una possibilità, seppure molto piccola, che due agenti abilitati allo svolgimento del ruolo da loro richiesto incorrano in problemi.

Per risolvere questo problema è stato introdotto l'utilizzo di **UUID** (*Uni-*

*versally Unique Identifier*), un oggetto Java che rappresenta un identificatore universalmente univoco. Tramite l'utiizzo di questa classe è possibile generare dei valori da 128 bit in modo casuale. Questi valori sono utilizzati, in associazione con l'identificativo dell'agente, per identificare univocamente la sessione di un agente che sta svolgendo un determinato ruolo all'interno dell'organizzazione. La forma della sessione quindi deve cambiare di conseguenza:

```
open_session(ContextId, AgentAid, ACCUUID, Role)
```

ACCUUID rappresenta il nuovo valore UUID assegnato alla sessione, mentre Role ora rappresenta un singolo ruolo. Infatti ogni sessione ora è una mappatura tra un agente ed un ACC costruito *ad hoc* per lo svolgimento di un singolo ruolo, per cui decade l'utilità di una lista.

### 3.1.5 Classi agente

La sola presenza di ruoli ed agenti all'interno del sistema pone forti limitazioni alle possibilità di gestione degli accessi da parte di un amministratore. A livello base si potrebbe per esempio creare dei ruoli ad accesso libero per gli agenti non dotati di credenziali e specificare ruoli appositi per gli agenti autorizzati.

L'introduzione delle classi agenti aiuta proprio in questo ambito, sono usate in cooperazione con i ruoli al fine di rilassare i vincoli e permettere una maggiore trasparenza.

Una classe agente viene utilizzata per rappresentare tutti gli agenti appartenenti ad un certo insieme, solitamente agenti a cui l'organizzazione concede gli stessi privilegi nell'accedere alle proprie risorse. Allo stesso modo una classe agente è assegnata dall'amministratore ad ogni ruolo, indicando cioè che

un certo ruolo può essere svolto solamente dagli agenti appartenenti ad una determinata classe agente.

La trasparenza rispetto all'agente, quindi ad un utilizzatore esterno al sistema, è data dal fatto che sia il sistema a decidere e quindi rilasciare la classe agente associata ad una particolare entità. Con i ruoli invece è lo stesso agente che si deve prendere carico delle fasi di negoziazione ed attivazione.

Quindi all'interno di un'organizzazione può essere effettuato uno studio sulle entità che cercano di accedere al sistema, sui loro comportamenti e necessità, ed in base a questo effettuare una loro divisione in insieme e definire delle classi agente per ognuno di questi.

La struttura RBAC-MAS permette come detto l'accesso ad entità non in possesso di credenziali, non in grado quindi di identificarsi con il sistema, a meno che non venga loro esplicitamente proibita l'entrata per una scelta amministrativa. Per permettere loro l'accesso e lo svolgimento di un ruolo è introdotto il concetto di **classe agente base**. All'interno del nodo è presente una proprietà che indica quale sia la *classe agente base* del sistema, da qui è poi possibile definire un insieme di ruoli la cui classe agente sia quella. Un agente che ottiene lo strumento di negoziazione, ma non è conosciuto dal sistema, ha assegnata questa classe che gli permette l'attivazione di quei ruoli la cui classe sia quella base.

La struttura RBAC implementata deve essere il più *malleabile* possibile, permettendo ad un entità dotata di privilegi amministrativi la modifica dinamica di struttura e proprietà. Deve essere posta attenzione anche alla modifica della classe agente di base, per cui di conseguenza devono essere modificati i valori anche dei ruoli ed agenti base, e della classe dei ruoli e degli agenti, per cui un agente potrebbe non appartenere più alla classe necessaria allo svolgimento di un certo ruolo.

### 3.1.6 Comportamento RBAC

La gestione corretta dell'organizzazione richiede un grosso lavoro anche da parte del nodo, il quale si occupa di processare le richieste in arrivo e rispondere nel modo più adeguato. Il sistema deve essere sempre in uno stato consistente, per assicurare il corretto funzionamento dell'infrastruttura di controllo degli accessi. Per fare questo, sono state introdotte numerose **reazioni ReSpecT**, le quali si occupano di effettuare gli interventi adeguati sulla struttura RBAC in base alle tuple inserite nel centro di tuple \$ORG. Presentiamo in questa sottosezione i vari comportamenti del nodo in base al tipo e ai contenuti delle tuple ricevute. Il codice viene omesso a fini semplificativi, ma è visionabile nel file *boot\_spec.rsp* contenuto nel package *alice.tucson.service.config*. Si tratta di un insieme di reazioni ReSpecT che vengono inserite nel nodo al momento dell'avvio.

- **Attivazione RBAC:** alla ricezione della tupla `install_rbac`, inviata da un amministratore, il nodo imposta la propria proprietà `rbac_installed` a *yes*, indicando che è presente ora un controllo sugli accessi;
- **Disattivazione RBAC:** in modo opposto al precedente, alla ricezione della tupla `disinstall_rbac` il valore viene reimpostato a *no*. Inoltre vengono eliminate tutte le tuple riguardanti la struttura RBAC, quindi quelle riguardanti ruoli, policy e agenti autorizzati;
- **Richiesta di contesto:** la richiesta di contesto per un ACC di un ruolo arriva nella forma `context_request(NomeAgente, Risultato, ClasseAgente, UIDAgente)`. La risposta del nodo può variare a seconda dei casi:

- È già stata aperta una sessione per una determinata coppia NomeAgente-UUIDAgente, il *Risultato* viene posto ad *ok* e la reazione ha successo;
  - Non è presente una sessione aperta e l'agente ha inviato una classe agente differente da quella base, dimostrando così di aver effettuato con successo il login: viene aperta una nuova sessione e la classe agente di questa viene posta al valore presente nella tupla `context_request`;
  - Non è presente una sessione aperta, la classe agente dell'agente è quella base, quindi non ha effettuato il login, e il login è stato impostato come non necessario nel nodo: viene creata una nuova sessione con la classe agente base;
  - Non è presente una sessione aperta, la classe agente dell'agente è quella base (no login) e il login è stato impostato come necessario nel nodo: la richiesta di contesto viene rifiutata e il campo *Risultato* viene posto a `failed(login_required)`.
- **Rilascio del contesto:** una volta svolti i propri compiti, un agente rilascia l'ACC ottenuto (o gli ACC in base al numero di attivazioni di ruoli) tramite l'invio di una tupla `context_shutdown( IDContesto, NomeAgente, Risultato)`. Il valore UUID dell'agente non è necessario in questo caso, poiché vi è solamente una coppia di identificativo del contesto e di nome dell'agente che l'ha attivato. Se la sessione è presente allora la tupla viene rimossa, mentre se non è presente allora il valore di *Risultato* viene posto a `failed(no_valid_context)`;
  - **Aggiunta di un ruolo:** all'inserimento di una tupla per un nuovo ruolo, nel formato `role(NomeRuolo, DescrizioneRuolo, ClasseAgente)`,

viene controllato il valore dell'ultimo campo. Infatti, nel caso l'amministratore non abbia impostato una classe agente al momento della creazione, questo campo assume il valore *substitute*, il che indica che deve essere sostituito con la classe agente base del sistema. Questa reazione è stata introdotta per poter gestire al meglio il carattere dinamico della classe agente di base;

- **Rimozione di un ruolo:** tramite la tupla `remove_role(NomeRuolo)` è possibile rimuovere un ruolo dall'organizzazione. Viene effettuato un controllo sull'esistenza di questo ruolo e inoltre viene rimossa la tupla che indica la dipendenza con la sua policy;
- **Attivazione di un ruolo:** aperta una sessione nell'organizzazione, gli agenti possono provare a richiedere l'attivazione di un ruolo, tramite la tupla `role_activation_request( NomeAgente, UUIDAgente, NomeRuolo, Risultato)`. Il nodo reagisce in questo modo:
  - Effettua un controllo per l'esistenza del ruolo, ma si accorge che non è presente nel nodo. La richiesta viene rifiutata e *Risultato* viene posto a `failed(role_not_found)`;
  - Il ruolo esiste e anche una sessione per l'agente, inoltre le classi agente del ruolo e dell'agente combaciano. La richiesta viene accettata ed il ruolo viene attivato, aggiornando la sessione con il nuovo ruolo attivato;
  - Il ruolo esiste e vi è una sessione aperta per l'agente, ma la classe dell'agente non combacia con quella del ruolo. La classe agente del ruolo è però quella base, per cui la richiesta viene accettata ed è aggiornata la sessione con il nuovo ruolo attivato;

- Il ruolo esiste e anche una sessione per l'agente, ma le classe agenti non combaciano e quella del ruolo non è quella base. La richiesta viene rifiutata e *Risultato* viene posto a `failed( agent_not_authorized)`;
- **Lista delle policy attivabili:** la lista delle policy utilizzabile dall'agente viene richiesta tramite la tupla `policy_list_request( ClasseAgente, Lista)`. All'interno del nodo viene costruita una lista di policy, scartando quelle inadeguate per la classe agente del ruolo a cui sono assegnate e quelle non assegnate a nessun ruolo, ma mantenendo anche quelle assegnate a ruoli la cui classe agente è quella base;
- **Ruolo associato ad una policy:** tramite la tupla `policy_role_request( NomePolicy, Risultato)` si ottiene la lista dei ruoli a cui è associata la policy *NomePolicy*;
- **Policy assegnata ad un ruolo:** tramite la tupla `role_policy_request( Risultato, NomeRuolo)` si richiede quale sia la policy del ruolo *NomeRuolo*;
- **Rimozione di una policy:** utilizzando la tupla `remove_policy( NomePolicy)` si richiede la rimozione della policy *NomePolicy*. Verrà rimossa anche la relazione con un ruolo se presente, collegando il ruolo ad una policy di default presente nel nodo (un ruolo deve sempre avere una policy assegnata);
- **Impostazione della policy di un ruolo:** con la tupla `set_role_policy( NomeRuolo, NomePolicy, Risultato)` si può impostare la policy assegnata ad un ruolo, controllando che ruolo e policy esistano e modificando la tupla `role_policy(NomeRuolo, NomePolicy)`;

- **Aggiunta di un permesso:** l'inserimento della tupla `add_permission( Permesso, NomePolicy, Risultato)` porta all'aggiunta del permesso *Permesso* alla policy *NomePolicy*, dopo aver controllato che la policy esista e non contenga già il permesso;
- **Impostazione classe agente di un ruolo:** `set_role_class( NomeRuolo, ClasseAgente, Risultato)` scatena una reazione che porta alla modifica della classe agente di *NomeRuolo* in *ClasseAgente*;
- **Autorizzazione degli Inspector:** `authorize_inspectors( Valore)` porta alla modifica della tupla che indica se gli Inspector sono o meno autorizzati all'accesso. *Valore* può assumere i valori *yes* o *no*;
- **Impostazione login obbligatorio:** l'inserimento della tupla `set_login( Valore)` porta alla modifica della proprietà di login obbligatorio nel nodo. Anche in questo caso *Valore* può assumere i valori *yes* o *no*;
- **Impostazione classe agente di base:** la tupla `set_base_agent_class( NuovaClasseAgente)` porta alla modifica della classe agente base del nodo. Insieme a questo devono essere modificati i valori delle classi agenti base presente nei ruoli;
- **Richiesta di login:** tramite `login_request( Credenziali, Risultato)` si può effettuare una richiesta di login nell'organizzazione. Se le credenziali risultano verificate allora la richiesta ha successo e viene ritornato il valore della classe agente assegnata all'agente richiedente. Nel caso di un fallimento invece viene ritornato il valore della classe agente di base;
- **Lista dei ruoli attivabili:** immettendo nel centro di tuple la tupla `role_list_request( ClasseAgente, Risultato)` si richiede la lista



dei ruoli attivabili per una certa classe agente *ClasseAgente*. Viene ritornata, se permesso dal nodo, una lista contenenti i ruoli permessi insieme alla loro policy e alla lista dei permessi associati.

## 3.2 Interazioni con il sistema

Il primo passo per l'interazione con il sistema è, come illustrato precedentemente, la *negoziazione* di un ACC. Per fare in modo che un agente possa compiere questa azione, è necessario che sia in possesso di uno strumento pensato appositamente per questo. Questo viene rilasciato in seguito alle chiamate dei metodi statici della class **TucsonMetaACC**, contenuta nel package *alice.tucson.api*. Questa classe è stata modificata poiché precedentemente permetteva, tramite una semplice chiamata, di ricevere un **ACCPProxyAgentSide**, utilizzato dagli agenti per effettuare le operazioni sul nodo. Nella nuova ottica, ora per poter essere effettuate necessitano prima di una fase di negoziazione e di attivazione di un ruolo che le permettano, da qui la necessità di una modifica della classe. In questa fase il lavoro di Galassi prevedeva uno stesso tipo di richiesta di **contesto** sia da parte di agenti amministrativi che dai restanti; nel nostro caso invece è stato scelto di dividere le due funzionalità.

**Agente admin.** Per poter ottenere privilegi amministrativi si richiama il metodo **getAdminContext**, inserendo tra i parametri anche le credenziali:

---

```
public static MetaACC getAdminContext(final TucsonAgentId aid,
    String netid, int portno, String username, String password) {
    MetaACC acc = null;
    try {
```

```

        acc = new MetaACCProxyAgentSide(aid, netid, portno,
            username, password);
    } catch (TucsonInvalidAgentIdException e) {
        System.err.println("[Tucson-MetaACC]: " + e);
        e.printStackTrace();
        return null;
    }
    return acc;
}

```

---

Viene ritornata un'istanza della classe **MetaACCProxyAgentSide**, la quale permette all'agente admin lo svolgimento dei suoi compiti.

**Agente user.** Nel caso invece che non si vogliano attivare privilegi amministrativi, quello che si vuole ottenere è uno strumento per la negoziazione di un ruolo, tramite il metodo **getNegotiationContext**:

---

```

public static NegotiationACC getNegotiationContext(final String
    aid, String netid, int portno){
    NegotiationACC acc = null;
    try {
        acc = new NegotiationACCProxyAgentSide(new
            TucsonAgentId(aid), netid, portno);
    } catch (TucsonInvalidAgentIdException e) {
        System.err.println("[Tucson-NegotiationACC]: " + e);
        e.printStackTrace();
        return null;
    }
    return acc;
}

```

---

Se l'istanziamento della classe va a buon fine, **NegotiationACCProxyAgentSide** viene ritornato all'agente.

### 3.2.1 Amministrazione di un nodo

Parlando dell'amministratore, ottenendo un'istanza di **MetaACCProxyAgentSide** esso ottiene i privilegi per agire sulla struttura RBAC del nodo.

Questa classe è un'implementazione dell'interfaccia **MetaACC**, contenuta nel package *alice.tucson.api*, creata con lo scopo di fornire dei metodi per la modifica a tempo di esecuzione delle informazioni riguardanti RBAC, cioè le proprietà del nodo e la struttura, agendo sul centro di tuple \$ORG.

---

```
public interface MetaACC extends EnhancedACC {

    void add(RBAC rbac) ;
    void add(RBAC rbac, Long l, String node, int port);
    void removeRBAC();
    void removeRBAC(Long l, String node, int port);

    void add(Role role);
    void removeRole(String roleName);

    void add(Policy policy);
    void removePolicy(String policyName);

    void add(AuthorizedAgent agent);
    void remove(String agentName);
```

```
void setRolePolicy(String roleName, String policyName);
void addPermissionToPolicy(Permission permission, String
    policyName);
void setRoleAgentClass(String roleName, String agentClass);

void setBaseAgentClass(String newBaseAgentClass);
}
```

---

Passiamo ora ad esaminare la classe, esaminandone il comportamento e mostrando l'implementazione dei diversi metodi. E' presente una singola proprietà di interesse, **admin\_authorized**, un valore *booleano* che ci indica se l'autorizzazione sia andata a buon fine o meno. Come prima cosa, alla creazione della classe questo valore viene posto a false e si richiama il metodo `activateAdminRole()`, il cui scopo è tentare un'attivazione del ruolo di admin.

---

```
LogicTuple template = new LogicTuple("role_activation_request",
    new Value(this.aid.toString()),
    new Value(this.getUUID().toString()),
    new Value("admin_role"),
    new Value(getUsername()+":"+TucsonACCTool.encrypt(
        getPassword())),
    new Var("Result"));
ITucsonOperation op = inp(tid, template, (Long)null);
if(op.isResultSuccess()){
    LogicTuple res = op.getLogicTupleResult();
    if(res!=null &&
        res.getArg(4).getName().equalsIgnoreCase("ok")){
        admin_authorized = true;
    }
}
```

```
}  
}
```

---

Nel caso l'operazione abbia successo (cioè viene ritornato un *ok*), allora il valore viene posto a true.

Metodi:

- **add(RBAC rbac):** Tramite questo metodo si procede all'installazione di struttura e regole RBAC all'interno del nodo. La classe *RBAC* è stata trattata nella sezione 3.1.1. Come prima cosa si verifica che l'agente sia stato autorizzato controllando il valore booleano:

---

```
if(!admin_authorized)  
    throw new OperationNotAllowedException();
```

---

Nel caso il valore sia false allora l'agente non è stato autorizzato, per cui viene lanciata un'eccezione **OperationNotAllowedException**, contenuta nel package *alice.tucson.api.exceptions*. Altrimenti si continua e si procede con l'installazione dell'infrastruttura RBAC tramite

---

```
inp(tid, new LogicTuple("install_rbac"), (Long)null);
```

---

la quale avverte il nodo dell'attivazione di RBAC e lo porta, tramite un'opportuna reazione ReSpecT a modificare alcune delle sue proprietà. Si procede poi con l'inserimento della struttura RBAC, cioè tramite l'inserimento delle informazioni relative a ruoli, policy ed agenti autorizzati:

---

```
for(Role role : rbac.getRoles())  
    this.addRole(role, 1);
```

```
for(Policy policy : rbac.getPolicies())
    this.addPolicy(policy, 1);

for(AuthorizedAgent authAgent : rbac.getAuthorizedAgents())
    this.addAuthorizedAgent(authAgent, 1);
```

---

All'inserimento della tupla per un ruolo viene inserita anche una tupla per specificare la relazione ruolo-policy. Il contenuto di queste tuple è presentato nella sezione **3.1**.

Completato questo si passa alle tuple riguardanti le proprietà del nodo, le quali sono configurabili dinamicamente da un agente amministratore a tempo di esecuzione. Queste riguardano:

– **Classe agente di default.**

```
LogicTuple defaultClassTuple = new
    LogicTuple("set_default_agent_class", new
        Value(rbac.getDefaultAgentClass()));;
ITucsonOperation op = inp(tid, defaultClassTuple, 1);
```

---

– **Login obbligatorio.**

```
LogicTuple loginTuple = new LogicTuple("set_login", new
    Value((rbac.getLoginRequired())? "yes" : "no"));
ITucsonOperation op = inp(tid, loginTuple, 1);
```

---

– **Inspectors autorizzati.**

```
LogicTuple inspectorsTuple = new
    LogicTuple("authorize_inspectors", new
```

```
Value((rbac.getAuthorizedInspectors())? "yes" :  
      "no"));  
ITucsonOperation op = inp(tid, inspectorsTuple, 1);
```

---

All'inserimento di ognuna di queste, si scatena una reazione ReSpecT in \$ORG che va a modificare le tuple che contengono informazioni su queste proprietà.

- **removeRBAC():** Tramite questo metodo, un amministratore può velocemente rimuovere l'infrastruttura RBAC dal nodo. Questo comporterà la rimozione di tutte le tuple riguardanti la struttura e la reimpostazione delle proprietà ai loro valori di default.

```
if(!admin_authorized)  
    throw new OperationNotAllowedException();  
inp(tid, new LogicTuple("disinstall_rbac"), (Long)null);
```

---

- **add(Role role):** Viene aggiunto un ruolo. Deve essere fornita anche una policy valida perché il ruolo sia funzionante;
- **removeRole(String roleName):** Viene rimosso dal centro di tuple un ruolo, in base al suo nome. Conseguentemente viene rimossa anche la tupla che specifica la relazione ruolo-policy;
- **add(AuthorizedAgent agent):** Immesso un nuovo agente autorizzato nel sistema;
- **setRolePolicy(String roleName, String policyName):** La policy del ruolo *roleName* viene impostata a *policyName*, dopo aver controllato che il ruolo e la policy esistano;

- **addPermissionToPolicy(Permission permission, String policyName)**: Viene aggiunto il permesso *permission* alla lista dei permessi della policy *policyName*, dopo aver verificato che la policy esista e che non sia già presente;
- **setRoleAgentClass(String roleName, String agentClass)**: Viene impostata la classe agente del ruolo *roleName* a *agentClass*, dopo aver controllato che il ruolo esista;
- **setBaseAgentClass(String newBaseAgentClass)**: La classe agente di base viene impostata a *newBaseAgentClass*, vengono modificati anche i valori delle classe agente dei ruoli e delle sessioni aperte.

### 3.2.2 Negoziazione/Attivazione di un ruolo

Il lavoro presentato da Galassi mostra un approccio simile a quello che è stato utilizzato nel nostro caso, sebbene implementato in modo differente: la richiesta di attivazione di un ruolo è data dalla chiamata ad un metodo dell'ACC, *activateRole*, a cui viene passata un'operazione per la cui esecuzione si desidera ottenere i permessi. Come viene mostrato in questa sezione, la strada scelta è stata invece quella di offrire due possibilità per l'attivazione, tra le quali la seconda presentata si mostra molto simile all'implementazione di Galassi.

Il primo passo che un agente deve compiere per entrare nel sistema è quello di ottenere uno strumento per negoziare l'accesso. Questo avviene tramite la chiamata al metodo *getNegotiationContext* della classe **TucsonMetaACC**, il quale ritorna un'implementazione dell'interfaccia **NegotiationACC** contenuta nel package *alice.tucson.api* .



**NegotiationACCPProxyAgentSide**, l'implementazione di questa interfaccia utilizzata nel progetto, viene utilizzata allo scopo di richiedere l'attivazione di un ruolo. Il successo di questa richiesta, come ricordiamo, è dato dal fatto che la classe agente associata all'agente richiedente sia corrispondente alla classe agente a cui è assegnato il ruolo, cioè quindi che l'agente sia autorizzato allo svolgimento di un particolare compito all'interno dell'organizzazione. Per permettere questo, la classe agente viene incapsulata internamente alla classe, in modo tale da essere utilizzata nelle richieste effettuate. All'ottenimento di questo strumento l'agente ancora non ha avuto modo di potersi autenticare con il sistema, per cui viene considerato un'entità non ancora conosciuta dal sistema; è possibile rendere questo aspetto ponendo la classe agente a quella di base del nodo, tramite una semplice richiesta al centro di tuple \$ORG alla creazione della classe. In questo modo è possibile ottenere sempre il valore aggiornato della classe agente di base anche se questo subisce modifiche da parte di un amministratore. Se si è invece già conosciuti dal sistema è possibile effettuare il login, presentando le proprie credenziali (username e password). Il successo nell'autenticazione porterà all'impostazione della propria classe agente a quella associata all'agente e presente nell'organizzazione. La richiesta ed attivazione di un ruolo è stata resa possibili tramite due differenti modalità: l'agente può inviare una richiesta per l'attivazione di un ruolo specifico oppure per un insieme di operazioni per cui vuole ottenere i permessi.

È qui presentata (sono state omesse le eccezioni lanciate dai metodi a fini di leggibilità):

---

```
public interface NegotiationACC {  
  
    EnhancedACC activateRole(String roleName);
```

```
EnhancedACC activateRole(String roleName, Long l);

EnhancedACC activateRoleWithPermission(List<String>
    permissionsId);
EnhancedACC activateRoleWithPermission(List<String>
    permissionsId, Long l);

EnhancedACC activateDefaultRole();

void listActivableRoles();

boolean login(String username, String password);
}
```

---

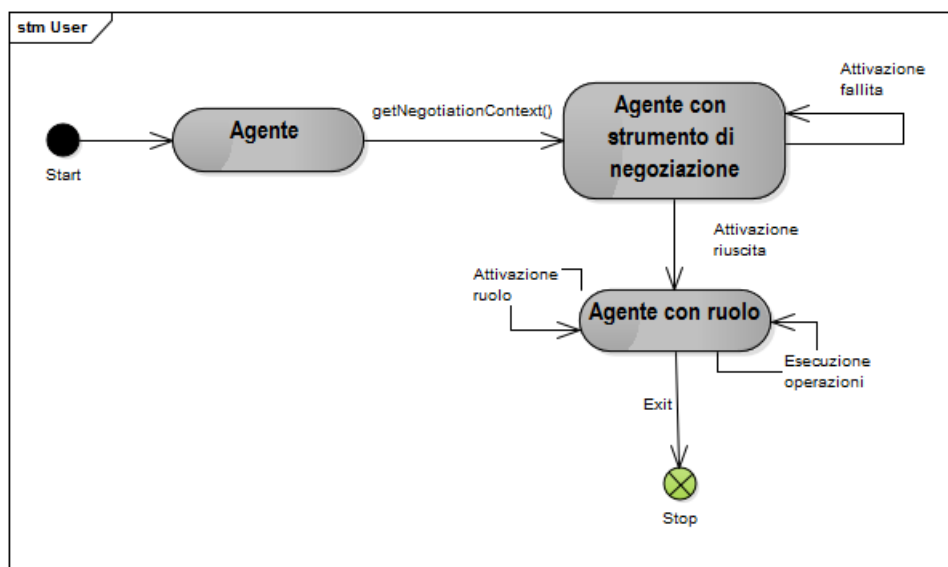
L'interfaccia propone differenti metodi:

- **activateRole(String roleName):** metodo utilizzato per richiedere l'attivazione di un particolare ruolo il cui nome è *roleName*;
- **activateRole(String roleName, Long l):** come il metodo precedente, con l'impostazione però del *timeout* dell'operazione TuCSon a *l*;
- **activateRoleWithPermission(List<String> permissionsId:** si richiede l'attivazione di un ruolo che contenga i permessi necessari contenuti in *permissionsId*;
- **activateRoleWithPermission(List<String> permissionsId, Long l):** come il metodo precedente, con timeout delle operazioni TuCSon impostato a *l*;

- **activateDefaultRole():** metodo utilizzabile nel caso non sia stata installata un'infrastruttura RBAC nel nodo, si richiede un ACC che permetta tutte le operazioni di default di TuCSon. Nel caso RBAC sia invece presente, la richiesta non viene soddisfatta;
- **listActivableRoles():** un agente può non conoscere a priori i ruoli stabiliti da un'organizzazione, o questi possono essere stati modificati nel frattempo, per cui è possibile richiedere una lista dei ruoli attivabili in base alla propria classe agente. Questa funzionalità può essere o meno permessa in base alle scelte nella gestione del nodo;
- **login(String username, String password):** tramite questo metodo è possibile identificarsi con l'organizzazione ed impostare così la propria classe agente a quella corrispondente alle nostre credenziali.

**Ciclo di vita** Il ciclo di vita di un agente senza ruoli amministrativi è riassunto nella figura 3.2. Inizialmente un agente non ha modo di agire sul nodo in alcun modo poiché non è possibile ora, diversamente da TuCSon senza RBAC, creare un ACC che gliene dia la possibilità. L'unica cosa che può essere fatta è la richiesta di uno *strumento di negoziazione*, che lo renda in grado di *negoziare* per l'attivazione di un ruolo. Una volta entrato in possesso di questo strumento, l'agente è in grado di comunicare con il nodo; può quindi ora inviare delle richieste per l'attivazione di un ruolo, tramite il quale poter interagire con le risorse dell'organizzazione in modo controllato. Queste richieste possono essere precedute da un tentativo di login tramite le credenziali in possesso dell'agente, che porta nel caso di successo alla definizione della classe agente dell'entità richiedente, fondamentale nel caso si stia tentando di attivare ruoli il cui accesso è limitato. Non appena viene attivato un ruolo, quello che avviene è la creazione di un ACC appositamen-

te *calibrato* per rispecchiare i permessi concessi; si tratta di un istanza di **RoleACCPProxyAgentSide**, che verrà trattato nella sezione 3.2.3.



**Figura 3.2:** Ciclo di vita di Agente User

L'agente entrato in possesso di questo nuovo ACC è ora in grado di effettuare operazioni in conformità con i permessi di cui è in possesso, ma può anche richiedere l'attivazione di un nuovo ruolo, il che comporta gli stessi passaggi precedenti e, nel caso di successo, la creazione di un secondo ACC.

Il numero di ruoli attivabili da un entità non è sottoposto a limitazioni. Una volta terminate le proprie attività, l'agente può rilasciare l'ACC, uscendo quindi dall'organizzazione e perdendo i ruoli attivati fino a quel momento.

### 3.2.2.1 Scelta della policy

La risposta ad una richiesta per l'attivazione di un ruolo preciso può portare solamente a due risultati possibili: la richiesta viene accettata e quindi l'agente è abilitato allo svolgimento del ruolo, oppure viene rifiutata e quindi l'agente non ottiene nessun privilegio. La situazione cambia però

quando la richiesta effettuata non è per un ruolo preciso, bensì per un insieme di permessi che si vuole svolgere sulle risorse dell'organizzazione. Vi possono essere infatti più ruoli, che il richiedente può attivare in base alla propria classe agente, che contengono l'insieme dei permessi necessari. È necessaria quindi una strategia per la scelta ottimale del ruolo tra quelli proposti.

Alla chiamata del metodo `activateRoleWithPermission` della classe *NegotiationACCPProxyAgentSide*, dopo aver controllato che RBAC sia stato installato, si richiede al nodo la lista delle policy attivabili, fornendo come parametro la propria classe agente. Questa funzionalità è stata inserita in **TucsonACCTool**, classe dotata di metodi statici contenuta in *alice.tucson.service.tools* e quindi utilizzabili da qualsiasi parte del nostro sistema.

---

```
public static List<Policy> getPolicyList(String agentClass,
    TupleCentreId tid, EnhancedACC acc){
    List<Policy> policies = new ArrayList<Policy>();
    try {
        LogicTuple policyListTuple = new
            LogicTuple("policy_list_request",
                new Value(agentClass),
                new Var("Result"));

        ITucsonOperation op = acc.inp(tid, policyListTuple,
            (Long)null);

        if(op.isResultSuccess()){
            LogicTuple res = op.getLogicTupleResult();

            TupleArgument[] policiesList =
```

```
        res.getArg(1).getArg(0).toArray();
    for(TupleArgument term : policiesList){
        TupleArgument[] permissionsTuples =
            term.getArg(1).toArray();

        Policy newPolicy = TucsonPolicy.createPolicy(
            term.getArg(0).toString(), permissionsTuples);
        policies.add(newPolicy);
    }
}
} catch (InvalidVarNameException |
        TucsonOperationNotPossibleException |
        UnreachableNodeException | OperationTimeOutException e) {
    e.printStackTrace();
}

return policies;
}
```

---

Alla ricezione della richiesta da parte del nodo corrisponderà lo scatenarsi di una reazione ReSpecT, la quale si occuperà di ritornare una lista contenente i ruoli attivabili, i permessi associati ad ogni ruolo e la classe agente necessaria all'attivazione.

In seguito al ricevimento del risultato della richiesta si procede con la trasformazione di questi dati in un formato maggiormente leggibile, associando questi valori ad elementi della struttura RBAC come ruoli o permessi, ed infine vengono ritornati alla struttura di negoziazione. Qui viene effettuata la scelta del ruolo da attivare richiamando il metodo interno *chooseRole*.

---

```
private Policy chooseRole(List<Policy> policies, List<String>
permissionsId){
    Policy bestPolicy = null;
    int bestPermissionsArity = 10000;

    for(Policy pol : policies)
        if(pol.hasPermissions(permissionsId))
            if(pol.getPermissions().size() <
                bestPermissionsArity){
                bestPermissionsArity =
                    pol.getPermissions().size();
                bestPolicy = pol;
            }

    return bestPolicy;
}
```

---

Le varie possibilità offerte vengono analizzate una dopo l'altra. Come prima cosa non vengono considerate le scelte dove non sono presenti tutti i ruoli richiesti. Ciò che resta è un sottoinsieme dei ruoli di partenza, quelli che sono in grado di soddisfare la richiesta di attivazione. Entra in gioco ora il **least privilege** (principio del minor privilegio), il quale comporta la concessione solo del minimo insieme di privilegi possibili in ogni istante, al fine del miglioramento della protezione del sistema. Infatti, è probabile che alcuni dei ruoli attivabili contengano al loro interno un numero di permessi superiori a quelli necessari all'agente. Seguendo il principio appena esposto, la scelta finale ricade sul ruolo che offre il minore numero di permessi all'agente.

Nel caso non sia presente un ruolo attivabile la cui policy soddisfi le richieste, allora viene lanciata un'eccezione del tipo *AgentNotAllowedException*.

L'implementazione adottata attualmente risulta quindi molto semplice. Un modo migliore potrebbe essere quello di costruire ad hoc un ruolo che contenga un sottoinsieme dei privilegi presenti, cioè che vengano scartati i permessi che non sono stati richiesti.

### 3.2.3 Svolgimento di un ruolo

Sebbene la parte di negoziazione non differisca in modo drastico da quella modellata da Galassi, lo svolgimento di un compito all'interno dell'organizzazione da parte di un agente segue politiche molto differenti. Non sono stati implementati gli stati, quindi non vi è un controllo su quale debbano essere le possibili azioni eseguibili in base allo stato in cui ci si trova.

Non appena viene attivato con successo un ruolo per un agente, quello che segue è la creazione di un ACC appositamente strutturato per permettere solamente le operazioni consentite dal compito che si è autorizzati a svolgere. Per lo scopo, viene istanziata un oggetto della classe **RoleACCProxyAgentSide**, contenuto nel package *alice.tucson.service*, che estende la classe **ACCProxyAgentSide** utilizzata di default per l'esecuzione delle operazioni TuCSOn. Ogni volta che viene eseguita un'operazione viene effettuato un controllo, interno alla classe, per verificare che l'agente abbia ottenuto i permessi per il suo svolgimento.

Incapsulato nella classe è l'attributo privato *role*, nel quale viene conservato il ruolo la cui politica deve essere rispettata. Questo valore viene impostato da parte di *NegotiationACCProxyAgentSide* al momento della creazione dell'oggetto e non può essere modificato, infatti l'attivazione di un altro ruolo comporta la creazione di un nuovo oggetto. Solamente a fini di semplificazione



ne dell'implementazione è stato aggiunto anche l'attributo *permissions*, una lista di stringhe contenenti i nomi delle operazioni consentite (sarebbe stato possibile ottenere questa lista ugualmente attraverso la policy associata al ruolo).

Per il controllo sulle operazioni consentite si utilizza il metodo privato `checkPermission(String permission)`:

---

```
private void checkPermission(String perm) throws
    TucsonOperationNotPossibleException{
    if(permissions.isEmpty() || !permissions.contains(perm))
        throw new TucsonOperationNotPossibleException();
}
```

---

Se l'operazione non è contenuta nella lista delle operazioni possibili, allora viene lanciata un'eccezione del tipo `TucsonOperationNotPossibleException`, altrimenti l'esecuzione continua nello stesso modo della superclasse estesa.

### 3.2.4 Creazione ACC per lo svolgimento del ruolo

Una volta trovato il ruolo migliore per soddisfare la richiesta di un agente, lo strumento di negoziazione (`NegotiationACCPProxyAgentSide`) provvede alla creazione di un ulteriore strumento (`RoleACCPProxyAgentSide`) che verrà utilizzato per effettuare le operazioni sul nodo. Questo consiste nell'estensione della classe *ACCPProxyAgentSide* e, sebbene alcune aggiunte utili per la nostra implementazione, il funzionamento delle operazioni segue lo stesso schema.

Nel nostro caso questo crea dei problemi, poiché alla prima operazione di un ACC viene inviata una richiesta di contesto al centro di tuple \$ORG per l'apertura di una sessione. Però nella fase di negoziazione precedente, durante

la richiesta di attivazione di un ruolo, era già stata inviata una richiesta di contesto per il nuovo ACC, per cui alla nuova richiesta troviamo già una sessione aperta. Secondo il comportamento base di TuCSoN questo porta al fallimento della nuova richiesta, poiché respinta dalla reazione ReSpecT:

---

```
reaction( inp(context_request(AgentId,_, AccUUID)),request,(
  rd(open_session(Id,AgentId,AccUUID,_)),
  no(context_request(AgentId,ok(_),AccUUID)),
  out(context_request(AgentId,failed(agent_already_present),AccUUID))))).
```

---

Nella risoluzione del problema sono state trovate ed implementate due differenti soluzioni, una lato agente ed una lato nodo.

**Intervento lato agente** La prima soluzione sviluppata è quella lato agente, cioè andando a modificare il comportamento dell'agente e lasciando inalterato il comportamento del nodo.

Il nostro problema deriva dalla creazione di un nuovo ACC per l'esecuzione delle operazioni dell'agente. È già presente però un ACC funzionante, interno a `NegotiationACCPProxyAgentSide` ed utilizzato per le precedenti richieste di attivazione. La soluzione più semplice quindi risulta il riutilizzo di questo anche da parte del nuovo `RoleACCPProxyAgentSide`.

Per effettuare le proprie richieste, l'ACC utilizza un *executor* interno, cioè un'istanza della classe **OperationHandler** contenuta in *alice.tucson.service*. Questa classe gestisce l'esecuzione delle operazioni sul nodo ed è responsabile di installare, memorizzare ed ottenere le connessioni verso tutti i centri di tuple contattati dall'agente TuCSoN. Per eseguire un'operazione, la classe cerca di ottenere una connessione verso il centro di tuple di destinazione, tramite il metodo `getSession`: se la connessione esiste allora viene ritornata, altrimenti

deve esserne creata una nuova (incapsulata nella classe `ControllerSession`) e poi memorizzata per uso futuro.

L'errore avviene proprio in questo momento: il nuovo ACC cerca di effettuare un'operazione ed il suo *executor* non trova una connessione verso la destinazione, quindi cerca di installarne una; come mostrato all'inizio questa richiesta viene rifiutata, poiché una sessione per l'agente è già stata aperta nel momento della negoziazione del ruolo.

La soluzione lato agente proposta è la più semplice. Al momento della creazione del nuovo `RoleACCProxyAgentSide` si sostituisce il nuovo `OperationHandler`, creato in modo automatico dal costruttore, con quello già presente nell'`internalACC` dello strumento di negoziazione. Questo avrà già installata la connessione verso il centro di tuple, per cui non vi dovrà essere una nuova richiesta di apertura di sessione alla prima operazione eseguita.

**Intervento lato nodo** La successiva soluzione implementata prevedere la modifica del comportamento lato nodo. Nella nuova ottica RBAC, risulta ora un evento normale la richiesta di contesto per un agente di cui è già aperta una sessione. Infatti accade ogni volta che viene negoziato ed attivato un ruolo. L'intervento lato nodo prevede come prima cosa la modifica di questo comportamento: se viene trovata una sessione già presente per un agente, allora si ha successo e viene ritornata come risposta l'*id* della sessione aperta.

La reazione è stata modificata come segue:

---

```
reaction( inp(context_request(AgentId,_, AccUUID)),request,(
  rd(open_session(Id,AgentId,AccUUID,_)),
  no(context_request(AgentId,ok(_),AccUUID)),
  out(context_request(AgentId,ok(Id),AccUUID))))).
```

---

A questo punto, ricevuto un ok come risposta, il nuovo `OperationHandler` può procedere con la creazione di una nuova `ControllerSession` che si baserà sulla sessione già aperta nel momento della negoziazione.

**Conclusione** Entrambe le soluzioni implementate consentono il funzionamento corretto. La prima però potrebbe mostrare complicazioni con la richiesta da parte dell'agente di più ruoli. Questo porterebbe infatti alla creazione di più ACC, i quali al loro interno utilizzerebbero tutti lo stesso `OperationHandler` originariamente di `NegotiationACCProxyAgentSide`. Come però è stato trattato nella sezione 3.1.4, si è scelto di differenziare all'interno del nodo ogni singolo ACC acquisito da un agente, tramite un utilizzo combinato del nome dell'agente e di un valore `UUID` univoco creato al momento dell'attivazione del ruolo. L'utilizzo di un `OperationHandler`, il quale detiene al proprio interno il valore di `UUID`, in un ottica condivisa creerebbe non pochi problemi, dovendo procedere all'implementazione di un comportamento per capire quale ACC di un agente stia effettuando un'operazione. A livello logico poi è risultato più corretto e lineare avere un singolo *executor* per ogni ACC e accettare che fosse già presente una sessione aperta (dal `NegotiationACCProxyAgentSide`) al momento della prima operazione di un nuovo `RoleACCProxyAgentSide`. Quindi la soluzione adottata nel progetto è la seconda, quella che ha richiesto un intervento lato nodo.

### 3.3 Sicurezza

Nel lavoro sviluppato da Galassi vengono analizzati ed utilizzati diversi *protocolli*, regole che devono essere seguite nella comunicazione tra diversi elementi connessi in una rete, al fine di rendere più sicuri gli scambi di messaggi tra le varie entità. Il lavoro svolto non è stato modificato, ma nel-

l'implementazione attuale di RBAC in TuCSoN non è stata implementata una sicurezza atta a preservare la confidenzialità dei messaggi.

Ai fini della sicurezza si è reso necessario però l'utilizzo di metodi di **criptaggio** nella gestione delle password. Infatti, con l'installazione di una infrastruttura RBAC nel nodo, sono inserite delle tuple contenenti le informazioni riguardanti gli agenti conosciuti ed autorizzati dal sistema. Siccome un agente è verificabile tramite le sue credenziali, nome utente e password, queste sono presenti nel nodo, insieme alla classe agente associata all'entità richiedente. Per mantenere la *riservatezza* la password non viene mai salvata nel centro di tuple come *plain text*, testo in chiaro, ma come risultato di un algoritmo di criptaggio. Per effettuare questa operazione è stato scelto l'algoritmo **SHA-256**, un algoritmo di hash che produce in output una stringa (*message digest*) di lunghezza fissa, partendo però da una di lunghezza variabile. Una funzione di questo tipo non è reversibile, non è quindi possibile risalire al messaggio originale da quello criptato, e non è possibile trovare due stringhe diverse che diano in output lo stesso risultato. In questo modo, anche entrando in possesso delle tuple presenti nel centro di tuple riguardanti gli agenti autorizzati, non è possibile risalire alla password in chiaro. Protezione da potenziali attacchi in cui l'attaccante può entrare in possesso dei messaggi scambiati, quindi anche le credenziali inviate per l'identificazione, è data dal criptaggio della password già da parte del mittente.

È scelta dell'amministratore del nodo consentire una maggiore o minore apertura del nodo, in base anche all'impostazione o meno di alcune proprietà:

- Inspector non consentiti: È possibile negare l'accesso al nodo agli Inspector, strumento di introspezione delle tuple di TuCSoN. Sebbene costituiscano un ottimo metodo per la visualizzazione delle informazioni salvate, l'accesso a tutte le tuple presenti nel nodo, anche quelle

amministrative, può non essere visto di buon occhio da parte di un'organizzazione. Tramite questa proprietà è quindi possibile garantire una maggiore riservatezza sulla struttura interna e sulla modalità di salvataggio delle informazioni nel nodo;

- Login obbligatorio: la natura distribuita ed eterogenea di un panorama distribuito rende spesso naturale la presenza di entità non conosciute a priori; proprio per questo sono state introdotte le classi agente, in modo tale da offrire spazio di azione anche agli agenti non autorizzati e noti a priori. Tuttavia può essere considerata la restrizione dell'accesso unicamente alle entità in possesso di credenziali valide.

Entrambe le proprietà considerate incidono a *livello di nodo*, non sul singolo centro di tuple dove sono inserite (in questo caso \$ORG). Infatti, l'accesso da parte di Inspector o agenti non autorizzati avviene attraverso una richiesta di contesto al centro di tuple amministrativo. Nel caso questa richiesta non venga soddisfatta, allora l'accesso è impedito per ogni centro di tuple del nodo interessato.

### 3.4 Test plan

Per la verifica del corretto funzionamento dell'implementazione è stato scelto di utilizzare gli esempi già presenti all'interno di TuCSoN nei package:

- `alice.tucson.examples.helloWorld`
- `alice.tucson.examples.diningPhilos`
- `alice.tucson.examples.distributedDiningPhilos`
- `alice.tucson.examples.timedDiningPhilos`

- `alice.tucson.examples.persistence`
- `alice.tucson.examples.situadness`
- `alice.tucson.examples.spawnedWorkers`

Come prima cosa i test sono stati effettuati in uno scenario privo dell'installazione di un'infrastruttura RBAC, per cui il funzionamento risulta molto simile a quello originario.

Purtroppo un comportamento di default non può avvenire correttamente senza modificare in minima parte il codice di questi esempi. Come era logico aspettarsi, la parte che necessita di modifiche è quella riguardante l'acquisizione di un contesto da parte di un agente, parte che ha subito pesanti modifiche nella nostra implementazione.

In TuCSoN solitamente un entità per ottenere un ACC effettua una chiamata del tipo:

---

```
final SynchACC acc = this.getContext();
```

---

Invece ora, sia che sia presente RBAC che non, bisogna passare attraverso una fase di negoziazione e una di attivazione di un ruolo. Senza un ruolo attivato non è possibile effettuare alcuna operazione primitiva sul ruolo.

La chiamata precedente diventa quindi ora:

---

```
NegotiationACC negACC =  
    TucsonMetaACC.getNegotiationContext(agentAid);  
SynchACC acc = negAcc.activateDefaultRole();
```

---

E con l'ACC ottenuto dall'attivazione del ruolo di default il flusso di esecuzione successivo rimane identico a prima. Se invece è presente un'installazione di RBAC attiva, allora a seconda dei bisogni della classe di esempio

bisognerà richiedere un ruolo per lo svolgimento delle operazioni necessarie. La riuscita o meno di chieste dipenderà dalla struttura organizzativa presente nel nodo.



# Capitolo 4

## Caso di studio

In questo capitolo si propone una simulazione del funzionamento del progetto creato, cercando di fare in modo di trattare tutti gli aspetti esposti finora. Verrà posta attenzione a ciò che succede all'interno dei vari componenti del sistema, utilizzando le informazioni stampate a console, le tuple inserite e le reazioni ReSpecT scatenate. Il caso di studio presentato è composto da:

- **Nodo TuCSoN:** Viene avviato un nodo TuCSoN con determinate proprietà impostate;
- **Agente amministratore:** Un agente ottiene i privilegi amministrativi e procede con un'istanziamento di un'istanza RBAC nel nodo. Durante l'esecuzione simula interventi dinamici sulla struttura e sulle proprietà RBAC, in modo da mostrare le reazioni del sistema;
- **Agente autorizzato `authorizedAgent`:** Un agente conosciuto ed autorizzato dal sistema effettua il login e negozia ed attiva ruoli a lui consentiti e ruoli non consentiti;
- **Agente non autorizzato `userAgent`:** Un agente invece non conosciuto dal sistema accede al sistema senza effettuare il login, cercando

poi di attivare ruoli a lui consentiti (ruoli con classe agente base) e ruoli non consentiti.

#### 4.0.1 Nodo

Viene qui presentato il codice utilizzato per l'installazione del nodo Tuc-SoN. I valori utilizzati per il nodo e per la porta sono quelli di default, cioè rispettivamente localhost e 20504.

---

```
public class TucsonNode {  
  
    public static void main(String[] args) {  
        TucsonNodeService tns = new TucsonNodeService();  
        tns.setAuthForAdmin(true);  
        tns.setAdminUsername("admin");  
        tns.setAdminPassword("abcd");  
        tns.setBaseAgentClass("base");  
        tns.setListAllRolesAllowed(true);  
        tns.setInspectorsAuthorized(true);  
  
        tns.install();  
    }  
}
```

---

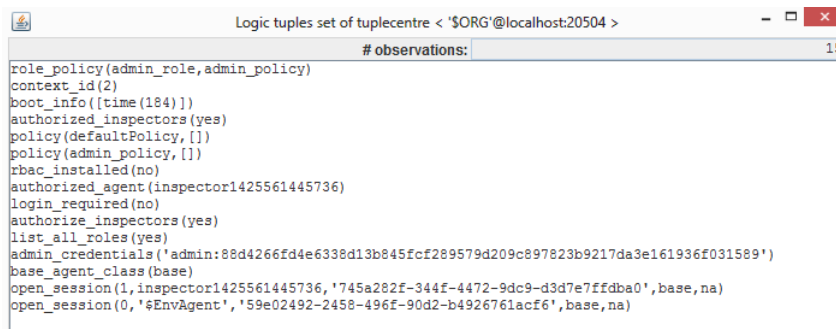
L'avvio del nodo avviene attraverso l'utilizzo di **TucsonNodeService** e l'impostazione di alcune proprietà:

- L'autorizzazione per ottenere i privilegi amministrativi è attivata;
- L'username dell'amministratore è admin;
- La password dell'amministratore è abcd;

- Il nome della classe agente di base è impostato a base;
- È abilitata la richiesta da parte degli agenti per la lista dei ruoli attivabili;
- È autorizzato l'accesso al nodo agli inspector.

Impostate queste, il nodo viene installato tramite l'utilizzo del metodo `install()`.

Tramite l'utilizzo di un Inspector, vediamo quali sono le tuple inserite nel centro di tuple \$ORG:



```

Logic tuples set of tuplecentre < '$ORG'@localhost:20504 >
# observations: 15
role_policy(admin_role,admin_policy)
context_id(2)
boot_info({time(184)})
authorized_inspectors(yes)
policy(defaultPolicy,[])
policy(admin_policy,[])
rbac_installed(no)
authorized_agent(inspector1425561445736)
login_required(no)
authorize_inspectors(yes)
list_all_roles(yes)
admin_credentials('admin:88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589')
base_agent_class(base)
open_session(1,inspector1425561445736,'745a282f-344f-4472-9dc9-d3d7e7ffdba0',base,na)
open_session(0,'sEnvAgent','59e02492-2458-496f-90d2-b4926761acf6',base,na)

```

**Figura 4.1:** Nodo installato

Bisogna ricordare che in questo momento, come possiamo capire dalla tupla `rbac_installed(no)`, RBAC ancora non è installato nel nodo. Per cui il funzionamento attuale è ancora quello di default di TuCSoN e le tuple aggiunte per le proprietà non influiscono in alcun modo sull'esecuzione.

## 4.0.2 Agente amministratore

In questa sezione viene simulato l'accesso al sistema da parte di un agente amministratore, il quale ottiene i privilegi amministrativi utilizzando le proprie credenziali. Fatto questo, procede con la formazione di una struttura

RBAC e la sua installazione all'interno del nodo. Infine sono mostrati alcuni suoi interventi dinamici effettuati su un'istanza di RBAC già funzionante.

Come prima cosa l'agente amministratore si occupa della creazione delle strutture dati che compongono l'istanza RBAC:

---

```
Permission prd = new TucsonPermission("rd");
Permission prdp = new TucsonPermission("rdp");
Permission pin = new TucsonPermission("in");
Permission pinp = new TucsonPermission("inp");
Permission pout = new TucsonPermission("out");

Policy policyrd = new TucsonPolicy("policyrd");
Policy policyrdrdp = new TucsonPolicy("policyrdrdp");
Policy policyin = new TucsonPolicy("policyin");
Policy policyout = new TucsonPolicy("policyout");

policyrd.addPermission(prd);
policyrdrdp.addPermission(prd);
policyrdrdp.addPermission(prdp);
policyin.addPermission(pin);
policyin.addPermission(pinp);
policyout.addPermission(pout);

Role roleRead = new TucsonRole("roleRead");
roleRead.setPolicy(policyrd);

Role roleReadP = new TucsonRole("roleReadP");
roleReadP.setPolicy(policyrdrdp);
```

```
Role roleReadIn = new TucsonRole("roleReadIn");
roleReadIn.setAgentClass("readClass");
roleReadIn.setPolicy(policyin);

Role roleWrite = new TucsonRole("roleWrite");
roleWrite.setAgentClass("writeClass");
roleWrite.setPolicy(policyout);

AuthorizedAgent agent1 = new TucsonAuthorizedAgent("readClass",
    "user", "abcdef");
AuthorizedAgent agent2 = new TucsonAuthorizedAgent("readClass",
    "user", "123456");
AuthorizedAgent agent3 = new TucsonAuthorizedAgent("writeClass",
    "peter", "olset935");
```

---

Una volta create queste strutture, si stabiliscono le loro relazioni: i permessi devono essere collegati alle policy e quest'ultime devono essere collegate ai ruoli. Infine si crea un oggetto della classe **TucsonRBAC**, che si comporta come un contenitore di tutta la struttura e delle proprietà RBAC che vogliono essere inserite nel nodo.

---

```
RBAC rbac = new TucsonRBAC("myOrg");
rbac.addRole(roleRead);
rbac.addRole(roleReadP);
rbac.addRole(roleReadIn);
rbac.addRole(roleWrite);
rbac.addPolicy(policyrd);
rbac.addPolicy(policyrdrdp);
rbac.addPolicy(policyin);
```

```
rbac.addPolicy(policyout);  
rbac.addAuthorizedAgent(agent1);  
rbac.addAuthorizedAgent(agent2);  
rbac.addAuthorizedAgent(agent3);
```

---

La stringa *myOrg* corrisponde al nome che vogliamo dare alla nostra organizzazione. Si tratta semplicemente di un elemento descrittivo poiché non influisce nell'esecuzione.

Si possono impostare alcune proprietà:

---

```
rbac.setAuthorizedInspectors(true);  
rbac.setBaseAgentClass("randomClassAgent");  
rbac.setLoginRequired(false);
```

---

A questo punto l'agente deve ottenere uno strumento per procedere all'installazione:

---

```
MetaACC adminACC =  
    TucsonMetaACC.getAdminContext(this.getTucsonAgentId(),  
    "localhost", 20504, "admin", "abcd");
```

---

La richiesta va a buon fine, dato che le credenziali fornite sono quelle impostate all'avvio del nodo, quindi viene ritornato un oggetto della classe **MetaACCProxyAgentSide**. L'amministratore procede successivamente come segue:

---

```
adminACC.add(rbac);  
adminACC.removePolicy("policyrd");  
adminACC.removeRole("roleRead");  
adminACC.setBaseAgentClass("newBaseClass");  
adminACC.exit();
```

---

---

`add(rbac)`: la struttura che è stata appena definita viene inviata al centro di tuple `$ORG`, vengono inserite le tuple riguardanti le strutture e cambiate se necessario quelle delle proprietà. Il centro di tuple ora contiene (per semplicità non sono presenti le tuple non di nostro interesse):

---

```
role_policy(roleWrite,policyout)
role_policy(roleReadIn,policyin)
role_policy(roleRead,policyrd)
role_policy(roleReadP,policyrdrdp)
role_policy(admin_role,admin_policy)
policy(policyrd, [rd])
policy(policyin, [in,inp])
policy(defaultPolicy, [])
policy(policyout, [out])
policy(admin_policy, [])
policy(policyrdrdp, [rd,rdp])
rbac_installed(yes)
authorized_agent(adminAgent)
authorized_agent(inspection1425568567493)
authorized_agent(inspection1425561445736)
authorized_agent(
    'user:bef57ec7f53a6d40beb640a780a639c83bc29ac8a9816f1fc6c5c6dcd93c4721',
    readClass)
authorized_agent(
    'peter:ede76df0afaad40e6363f2139513c33ac4e759ad28e225350628e9f81f9283c7',
    writeClass)
authorized_agent(
    'user:8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92',
    readClass)
```

```
role(roleWrite,ruolo_roleWrite,writeClass)
role(roleReadIn,ruolo_roleReadIn,readClass)
role(roleRead,ruolo_roleRead,randomClassAgent)
role(roleReadP,ruolo_roleReadP,randomClassAgent)

context_id(3)
authorized_inspectors(yes)
organisation_name(myOrg)
login_required(no)
authorize_inspectors(yes)
list_all_roles(yes)
admin_credentials(
    'admin:88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589')
base_agent_class(randomClassAgent)

open_session(2, adminAgent,
    '53f9be6b-5407-45d8-8514-2a54f52b7396', base, na)
open_session(1, inspector1425561445736,
    '745a282f-344f-4472-9dc9-d3d7e7ffdba0', base, na)
open_session(0, '\$EnvAgent',
    '59e02492-2458-496f-90d2-b4926761acf6', base, na)
```

---

Le tuple sono state suddivise in tre blocchi e indicano:

- Struttura RBAC;
- Proprietà RBAC. `context_id` contiene l'identificatore da assegnare alla prossima sessione che verrà aperta;
- Sessioni aperte degli agenti.



---

`removePolicy(policyrd)`: questo metodo porta alla rimozione della policy *policyrd*. Se la policy rimossa era precedentemente assegnata ad un ruolo, cioè esiste una tupla del tipo `role_policy(RoleId, policyrd)`, allora il nodo provvederà a rimuovere questa relazione e ad assegnare una policy di default al ruolo, dato che un ruolo deve sempre averne assegnata una. La tupla `policy(policyrd, [rd])` viene eliminata, mentre la tupla `role_policy(roleRead, policyrd)` viene modificata in `role_policy(roleRead, defaultPolicy)`;

`removeRole(roleRead)`: l'agente rimuove il ruolo *roleRead*, portando anche alla rimozione della tupla `role_policy(roleRead, defaultPolicy)`;

`setBaseAgentClass(newBaseClass)`: viene impostato un nuovo valore per la classe agente di base. Il nodo si deve occupare di aggiornare questo valore nelle tuple che descrivono i ruoli, in quelle degli agenti e in quelle delle sessioni aperte. Le tuple modificate sono, con i loro valori aggiornati:

- `role(roleReadP, ruolo_roleReadP, newBaseClass)`;
- `base_agent_class(newBaseClass)`;

Nel caso il nodo fosse stato in utilizzo da altri agenti con classe agente base, allora anche il valore della loro sessione sarebbe stato aggiornato;

`exit()`: terminati i propri compiti, l'agente amministratore richiama il metodo `exit()`, in modo tale da rilasciare l'ACC creato per lui. Questo porta alla rimozione lato nodo della sessione aperta.

### 4.0.3 Agente autorizzato

In questa sezione viene mostrato un agente dotato di credenziali, quindi autorizzato e conosciuto dall'organizzazione, che tenta di accedere al sistema per negoziare ed attivare un ruolo ed accedere alle risorse.

La prima fase dell'interazione riguarda l'acquisizione di uno strumento di negoziazione, un'istanza di `NegotiationACCProxyAgentSide`:

---

```
NegotiationACC negACC =
    TucsonMetaACC.getNegotiationContext("authAgent");
```

---

Ottenuto lo strumento, l'agente tenta di effettuare il login nel sistema:

---

```
negACC.login("peter", "olset935");
```

---

Dato che le credenziali fornite sono esatte, il sistema autorizza l'agente e cerca la classe agente che gli è stata assegnata, nel nostro caso `writeClass`, che viene impostata come valore del campo privato `agentClass` di `negACC`.

L'agente poi tenta di attivare un ruolo che gli permetta lo svolgimento di un'operazione, in questo caso `out`:

---

```
List<String> permissions = new ArrayList<String>();
permissions.add("out");
EnhancedACC acc = negAcc.activateRoleWithPermission(permissions);
```

---

Il nodo verifica che la classe agente del richiedente è adeguata per l'attivazione del ruolo, per cui accetta la richiesta e apre una nuova sessione per l'agente:

```
open_session(4, agente1, 'd5553d77-1742-48a1-9252-aad51a1882df',
writeClass, roleWrite) L'agente è ora abilitato allo svolgimento del ruolo
roleWrite e ad effettuare operazioni primitive out.
```

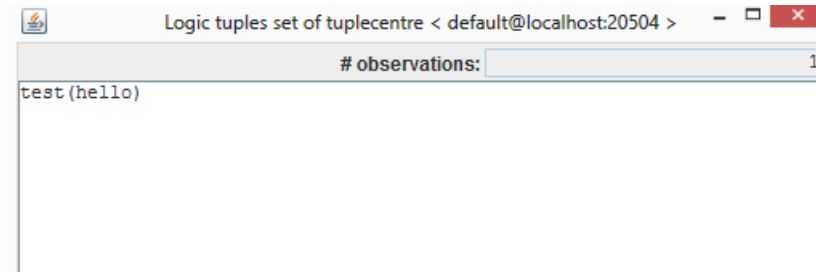
---

```
EnhancedACC acc = negAcc.activateRoleWithPermission(permissions);
acc.out(new TucsonTupleCentreId("default", "localhost", "20504"),
    new LogicTuple("test", new Value("hello")), (Long)null);
```

---

E tramite l'Inspector verificiamo come la richiesta sia andata a buon

fine:



**Figura 4.2:** Primitiva out con successo

L'agente autorizzato prova successivamente ad attivare un ruolo la cui classe agente non combacia con quella ottenuta nel momento del login, con il metodo *negACC.activateRole(roleReadIn)*. Come ci si può aspettare la richiesta non viene completata con successo, per cui viene lanciata un'eccezione del tipo *AgentNotAllowedException*.

#### 4.0.4 Agente non autorizzato

Infine si simula l'accesso al sistema di un agente non dotato di credenziali, per cui non autorizzato e conosciuto dall'organizzazione. Come è lecito aspettarsi, questa entità può richiedere solamente l'attivazione di quei ruoli associati alla classe agente base.

L'esecuzione segue le stesse fasi di negoziazione ed attivazione dell'agente autorizzato, sebbene ovviamente senza effettuare un tentativo di login.

---

```

NegotiationACC negAcc =
    TucsonMetaACC.getNegotiationContext("userAgent");
List<String> permissions = new ArrayList<String>();
permissions.add("rd");

EnhancedACC acc = negAcc.activateRoleWithPermission(permissions);

```

Come prima cosa, dopo la richiesta di uno strumento di negoziazione, viene ritornato un **NegotiationACCPProxyAgentSide** la cui proprietà `agentClass` è posta alla classe agente base del sistema, nel nostro caso - *randomClassAgent*, impostata dall'agente amministratore. Tramite questo strumento viene inviata una richiesta di attivazione di un ruolo che permetta l'operazione *rd*.

Il nodo si occupa dell'apertura di una nuova sessione per il nuovo agente, impostando la sua classe agente a quella base ed il valore del ruolo attivato a *na*, valore utilizzato dal nodo per indicare che nessun ruolo è ancora stato attivato per la sessione:

```
open_session(4,userAgent,'4ee6f2a5-9207-43b5-b21e-399e65f05995',
randomClassAgent, na)
```

Aperta la sessione, viene inviata una richiesta per l'apertura del ruolo *roleRead*, che contiene i permessi per le operazioni necessarie e la cui classe agente è accessibile dall'agente (essendo la classe agente di base). La richiesta va quindi a buon fine e la sessione viene aggiornata:

```
open_session(4,userAgent,'4ee6f2a5-9207-43b5-b21e-399e65f05995',
randomClassAgent, roleRead)
```

Infine viene ritornata un'istanza di **RoleACCPProxyAgentSide**, autorizzata solamente allo svolgimento dell'operazione *rd*.

Il nuovo ACC viene testato:

---

```
ITucsonOperation op = acc.rd( new TucsonTupleCentreId( "default",
    "localhost", "20504"), new LogicTuple( "prova", new Var("A")),
    (Long)null);
LogicTuple res = op.getLogicTupleResult();
say("Tupla ricevuta: " + res.toString());
```

E il risultato ottenuto ci dimostra come tutto sia andato a buon fine (la tupla è stata inserita precedentemente nel centro di tuple *default*):

```
....[RespectVMContext (default@localhost:20504)]: COMPLETION phase: [ op: prova(tuplaInserita),  
[userAgent]: Tupla ricevuta: prova(tuplaInserita)
```

**Figura 4.3:** Primitiva rd con successo

L'ACC ottenuto viene utilizzato nuovamente poi per effettuare un'operazione per la cui l'agente non ricopre un ruolo adeguato:

```
acc.out(new TucsonTupleCentreId( "default", "localhost", "20504"),  
new LogicTuple("provaOut(A)", (Long)null));
```

L'operazione non va a buon fine e viene lanciata l'eccezione *TucsonOperationNotPossibleException*.

Come ultima prova, l'agente tenta poi di attivare il ruolo *roleWrite*, la cui classe agente è *writeClass* e che quindi non può attivare, con `negAcc.activateRole(roleWrite)`. Come ci si può aspettare, viene lanciata un'eccezione di tipo *AgentNotAllowedException*.



# Capitolo 5

## Conclusioni

L'obiettivo principale di questa tesi è stata la creazione di una prima implementazione **pratica** e soprattutto **utilizzabile** di un meccanismo di controllo degli accessi all'interno dell'infrastruttura *TuCSoN*. La policy scelta per questo meccanismo si è basata su un approccio orientato ai ruoli, **RBAC**, per la sua affidabilità e la sua capacità di ingegnerizzare in modo naturale l'*organizzazione* all'interno di sistemi complessi. Il concentrarsi sul lato pratico è dato principalmente dal fatto che in letteratura esistono già innumerevoli studi in materia di organizzazione e coordinazione nei sistemi multiagente, ma ancora manca una tangibile implementazione all'interno del linguaggio TuCSoN. Il lavoro è quindi partito da una solida base teorica già formata, fornita nel nostro caso dal lavoro precedentemente svolto in materia da Galassi. Da questo si è cercato di estrapolare i concetti fondamentali di una struttura di controllo degli accessi, che potesse essere funzionale e semplice, non pretendendo di essere una soluzione definitiva e completa al problema. Il risultato ottenuto risulta soddisfare appieno gli obiettivi di partenza, con la creazione di un'infrastruttura funzionante e pronta ad essere integrata all'interno del linguaggio. Con la nuova implementazione è possibile

avere un reale controllo degli accessi, dove un agente amministratore è ora in grado di plasmare le autenticazioni ed autorizzazioni per l'accesso alle risorse come meglio crede, mentre un agente esterno ora ha un **accesso controllato** a queste risorse, portando quindi ad un notevole aumento nella sicurezza di questi sistemi. Ovviamente l'infrastruttura sviluppata potrà essere ampliata per permettere una maggiore complessità e la possibilità di avere una granularità maggiore nella gestione di ruoli, permessi ed agenti. In aggiunta la sicurezza nelle comunicazioni, sebbene non sia stata implementata poiché non di interesse nella presente trattazione, è stata ampiamente trattata nel lavoro analizzato in partenza ed è pronta per un'integrazione all'interno del linguaggio. Stesso discorso vale per le società di agenti e per la topologia, anche esse trattate in profondità in letteratura.



# Bibliografia

- [Omi02] 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.
- [ORV03] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Formal specification and enactment of security policies through Agent Coordination Contexts. *Electronic Notes in Theoretical Computer Science*, 85(3):17–36, August 2003. 1st International Workshop “Security Issues in Coordination Models, Languages and Systems” (SecCo 2003), Eindhoven, The Netherlands, 28–29 June 2003. Proceedings.
- [ORV05] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. RBAC for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science*, 128(5):65–85, 3 May 2005. 2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo’04), 30 August 2004. Proceedings.
- [RVO06] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Agent coordination contexts in a MAS coordination infrastructure. *Applied Artificial Intelligence: An International Journal*, 20(2–4):179–202,

February–April 2006. Special Issue: Best of “From Agent Theory  
to Agent Implementation (AT2AI) – 4”.

# Ringraziamenti

I miei ringraziamenti vanno al professore Omicini e all'Ing. Mariani per il supporto e l'aiuto concesso nello sviluppo di questa tesi, offrendo spunti per continuare nei punti morti trovati e tranquillità durante il lavoro. Ringraziamenti speciali sono per la mia famiglia, che mi ha seguito in tutto questo percorso, non esitando nemmeno una volta ad incoraggiarmi e spingermi verso la mia strada, anche a costo di molti sacrifici da parte loro. Ringrazio con il cuore Nicole, per l'immenso supporto che mi ha mostrato in questo periodo di ansie e di preoccupazioni, per esserci sempre e comunque nei miei momenti di bisogno e per credere costantemente in me. Ti ringrazio. *Last but not least*, un grazie finale a tutti i compagni di viaggio di questi anni universitari, sia i docenti che tutte quelle persone che ho potuto conoscere e che hanno condiviso con me questa esperienza.

Esperienza che ora giunge al termine, ma che non mancherà di essere ricordata con un po' di nostalgia.

Grazie.