Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

Coordinazione e tolleranza ai guasti: preservare lo spazio dell'interazione in TuCSoN

Tesi in
Sistemi Autonomi

Relatore:
Andrea Omicini

Presentata da:
Matteo Delvecchio

Correlatore:
Stefano Mariani

Sessione III
Anno Accademico 2015/2016

Indice

Introduzione

Al giorno d'oggi i sistemi software sono elementi costitutivi della quotidianità della società, ramificando il loro impiego in pressoché tutti i settori e contesti. L'evoluzione di tali sistemi rende necessaria la consapevolezza che non si tratta più di semplici strumenti, ma di vere e proprie infrastrutture, che necessitano di ingegnerizzazione al fine di soddisfare requisiti di funzionalità ed essere affidabili: così come consideriamo ponti, strade, edifici, città e via discorrendo come entità (tendenzialmente) affidabili e sicure, dovremmo allo stesso modo concepire sistemi software che forniscono all'utilizzatore lo stesso grado di sicurezza e naturalezza nell'uso: pertanto è necessario costruire sistemi a partire da un profondo grado di analisi e progettazione, nonché di verifica e collaudo.
In questo panorama un argomento fondamentale ai fini di garantire robustezza e affidabilità è la possibilità di gestire situazioni impreviste e anomale, in modo tale da mantenere costantemente il corretto funzionamento del software.
L'elaborato ha l'obiettivo di esplorare, a partire dallo studio di un'infrastruttura di coordinazione di sistemi autonomi, le tematiche inerenti alla robustezza e alla gestione di guasti, realizzando nello specifico alcune parti di estensione a tale infrastruttura. La valutazione dei risultati che saranno ottenuti permetterà di comprendere meglio quali strategie sono più efficaci in questo contesto specifico, ma anche di delineare meglio caratteristiche e proprietà necessarie ai fini di flessibilità, robustezza e capacità di recupero nell'ambito dei sistemi distribuiti ed autonomi.

Capitolo 1

Panoramica su TuCSoN

1.1 Definizione di modello di coordinazione

I sistemi distribuiti al giorno d'oggi rivestono un ruolo sempre più rilevante: basti pensare al numero di dispositivi dotati di capacità computazionali che utilizziamo quotidianamente, i quali sono spesso eterogenei e distribuiti dal punto di vista fisico, ma continuamente interconnessi. Naturalmente ciò che fa di un gruppo di apparati distribuiti un sistema è l'interazione tra le parti, per cui un insieme di elementi può essere considerato (nella giusta chiave di astrazione) come un'unica entità.
L'aspetto di interconnessione tra le parti del sistema è chiaramente particolarmente delicato. Tra gli aspetti più complessi a cui prestare attenzione abbiamo:

- **Distribuzione** La distribuzione può essere sia fisica (distribuzione spaziale) che logica: nel caso di quest'ultima le attività computazionali che costituiscono il sistema vivono in contesti di esecuzione diversi ed eterogenei (come ad esempio dispositivi e macchine virtuali differenti).

- **Concorrenza e Parallelismo** Lo scenario comprende una moltitudine di entità (flussi di controllo, processi, Thread, attori, *agenti*), attivi simultaneamente, e per i quali bisogna predisporre adeguati meccanismi di gestione della concorrenza e di bilanciamento.

- **Interazione tra le entità** L'interazione sociale deve garantire gli aspetti caratteristici di cooperazione, coordinazione (per il raggiungimento di obiettivi collettivi) e competizione, nonché rispettare e gestire le dipendenze tra le entità.

- **Interazione con l'ambiente** Riveste un ruolo chiave nei sistemi distribuiti la capacità di relazionarsi con risorse ambientali (altre entità computazionali presenti nell'ambiente circostante oppure entità e fenomeni fisici).

Un riferimento di grande importanza per la costruzione di sistemi distribuiti è il mondo biologico: la metafora naturale permette di apprendere strategie utili anche nel contesto informatico.
Osservando la maggior parte dei sistemi distribuiti presenti in natura, quali la società umana, le organizzazioni di molte specie di insetti e via discorrendo, emerge una caratteristica fondamentale, ossia la capacità di sviluppare comportamenti complessi attraverso la coordinazione.
Nella metafora biologica troviamo un esempio di Medium di coordinazione nell'ambiente, nel quale le entità riescono a comunicare informazioni mediante la percezione dello stato ambientale e la sua modificazione, e grazie ad esso sono in grado di sviluppare forme di auto organizzazione elaborate: troviamo esempi di questo fenomeno, detto **stigmergia** in numerosi *insetti sociali*, oltre che in numerose altre forme di vita. Il famoso algoritmo euristico di *Ant Colony Optimization*, si basa proprio sull'utilizzo dell'ambiente come medium di comunicazione.
Partendo da queste considerazioni è possibile individuare delle caratteristiche utili anche per la costruzione di sistemi distribuiti software.
Per sviluppare un sistema di comunicazione tra un numero non definito a priori di entità una strategia possibile è dunque quella di definire uno spazio di interazione dotato di certe caratteristiche: esso ha l'obiettivo di *abilitare, promuovere, disciplinare le interazioni ammissibili, desiderabili, necessarie tra le entità interagenti.*[1]
Così come l'ambiente è dotato di leggi fisiche che scatenano specifici comportamenti a seconda delle circostanze, sarebbe desiderabile poter ottenere meccanismi simili nel mondo sw, ossia dotare gli spazi di interazione di specifiche comportamentali, possibilmente non fissato a priori e mutabile, dunque programmabile.

1.2 TuCSoN: descrizione generale

TuCSon (Tuple Centre Spread over the Network)[2] nasce dall'idea di creare uno spazio dati condiviso e associativo il cui comportamento può essere cucito sulle necessità dell'applicazione specifica. Esso si basa sulla nozione di ***Tuple Centre***, ossia un *Tuple Space* dotato della capacità di essere programmato, garantendo dunque la possibilità di poter adattare il medium di coordinazione in base allo

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

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

specifico problema di coordinazione, evitando per giunta di utilizzare un comportamento definito a priori. Il linguaggio di specifica del comportamento dei centri è espresso attraverso un *reaction specification language*, il quale associa un qualsiasi evento del *Tuple Centre* ad un insieme di attività computazionali dette *reazioni*. L'implementazione usata di default in TuCSoN di tale linguaggio è ReSpecT (*Reaction Specification Tuples*)[3]: esso è un linguaggio *logic-based* che promuove il modello di coordinazione basato su tuple, consentendo di catturare eventi (di interazione) e associarli a reazioni (attività computazionali eseguibili localmente al *Tuple Centre*) in risposta a tali eventi.

1.2.1 Elementi costitutivi

È di seguito riportata una panoramica delle componenti principali che costituiscono l'infrastruttura TuCSoN:

- Le entità che necessitano del medium di coordinazione sono dette Agenti TuCSoN. Essi possono essere processi distribuiti così come agenti autonomi o agenti mobili. Tali agenti possono trovarsi ovunque sulla rete e possono interagire con i centri di tuple ospitati da qualsiasi nodo TuCSoN raggiungibile, inoltre essi possono muoversi indipendentemente dal dispositivo su cui sono eseguiti.

- I Nodi TuCSoN [1.1] rappresentano l'astrazione topologica che ospita i centri di tuple: essi definiscono il contesto di coordinazione, il quale regola l'accesso ai centri di tuple da parte degli agenti. Si parla pertanto di *Agent Coordination Context* (ACC)[4], un'interfaccia che consente di relazionarsi con i centri presenti e ne regola le interazioni possibili.

- Il medium di coordinazione di default è costituito dalle tuple di specifica ReSpecT presenti nei centri di tuple.

- I centri di tuple, come già accennato, sono spazi di tuple programmabili [1.2], ovvero il cui comportamento può essere impostato al fine di soddisfare le politiche di coordinazione volute, fornendo pertanto un medium di coordinazione *general purpose* personalizzabile. Tale comportamento è dinamico in quanto può essere adattato (riprogrammando il medium di coordinazione) oppure ispezionato a tempo di esecuzione.

[3] http://apice.unibo.it/xwiki/bin/view/ReSpecT/WebHome

[4] Omicini, A. and Denti, E. (2001). From tuple spaces to tuple centres. Science of Computer Programming, 41(3):277–294.

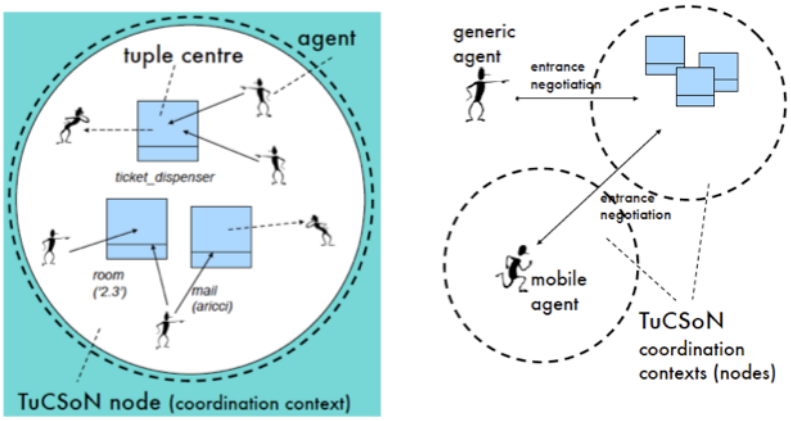

Figura 1.1: A sinistra: rappresentazione di un nodo TuCSoN. A destra: rappresentazione dell'interazione con il nodo

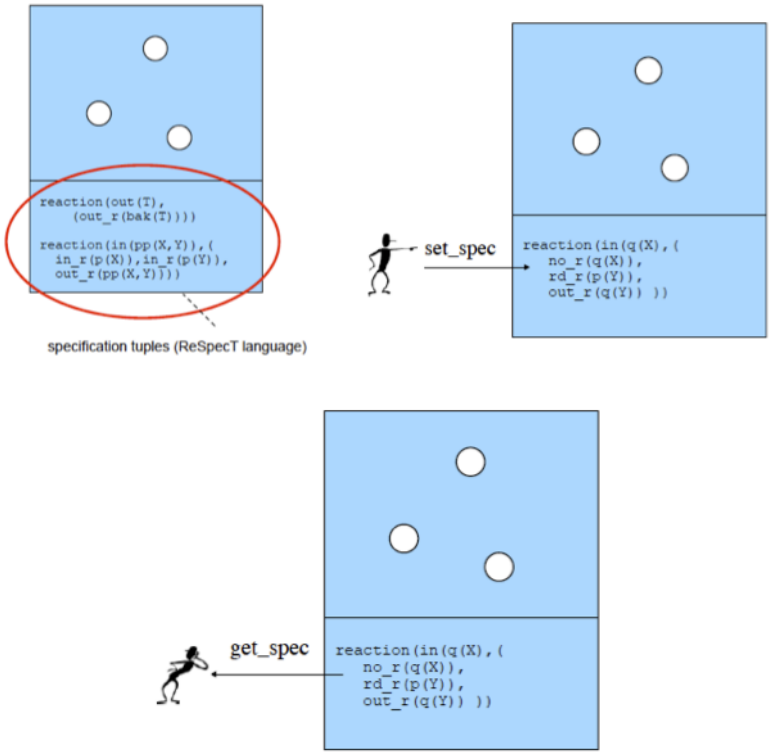

Figura 1.2: Rappresentazione di centri di Tuple, in particolare la parte di specifica

1.2.2 Topologia

Sostanzialmente gli agenti e i centri di tuple sono distribuiti sulla rete: naturalmente i *Tuple Centre* (da qui in poi chiamati anche "tc") esistono all'interno dei nodi TuCSoN. In linea di principio gli agenti possono muoversi indipendentemente dal dispositivo in cui sono eseguiti, mentre i centri di tuple sono permanentemente associati al device su cui sono eseguiti.

1.2.3 Interazione

Essendo gli agenti entità proattive e i centri di tuple entità reattive, tali agenti necessitano di operazioni di coordinazione al fine di poter interagire con i centri. Tali operazioni sono basate sul linguaggio di coordinazione di TuCSoN, e sono composte di due fasi [1.3]:

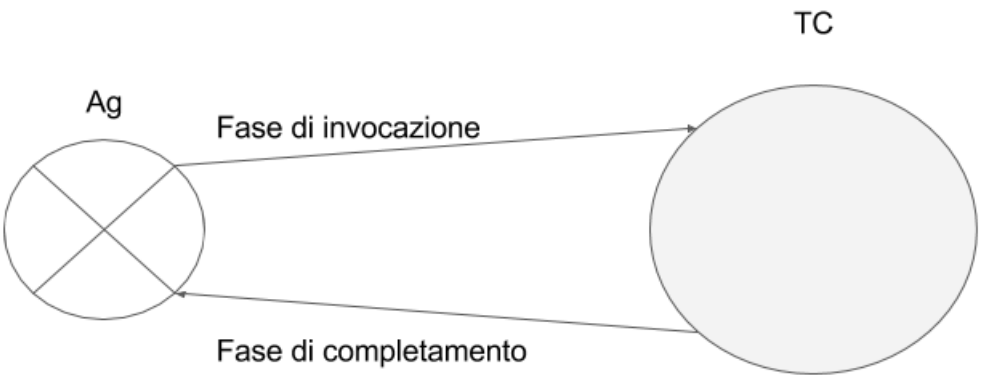

Figura 1.3: Fasi delle operazioni di coordinazione

- **Fase di invocazione:** essa consiste nella richiesta da parte dell'agente, la quale raggiunge il centro di tuple, arricchita da informazioni relative alla richiesta stessa
- **Fase di completamento:** essa è la risposta del tc, inoltrata all'agente chiamante, nella quale sono contenute anche informazioni sul risultato dell'esecuzione dell'operazione

Il linguaggio di coordinazione di TuCSoN mette a disposizione 9 primitive con le quali costruire operazioni di coordinazione:

- **out** permette di inserire una tupla nel tc bersaglio

- **rd**, **rdp** consentono di leggere una tupla corrispondente al template utilizzato sul tc designato

- **in**, **inp** permettono di estrarre una tupla corrispondente al template utilizzato sul tc designato

- **no**, **nop** permettono di verificare la presenza di tuple corrispondenti al template dato nel centro bersaglio

- **set** consente di sovrascrivere l'insieme di tuple presenti nel tc bersaglio

- **get** consente di leggere tutte le tuple presenti nel centro di tuple designato

I centri di tuple pertanto forniscono uno spazio condiviso per la comunicazione basata su tuple, oltre che la coordinazione *tuple-based* grazie al comportamento programmabile.

1.2.4 Coordinazione con ReSpecT

Le operazioni di meta-coordinazione di TuCSoN sono basate sul linguaggio di specifica ReSpecT e sulle primitive di meta-coordinazione: queste ultime riflettono le primitive di coordinazione TuCSoN viste nella sezione precedente, differenziandosi da queste per il fatto che vanno ad agire sullo spazio di specifica del *Tuple Centre*. Viene riportato l'elenco di tali primitive:

- **out_s** permette di inserite una tupla di specifica nello spazio di specifica del tc bersaglio

- **rd_s**, **rdp_s** consentono di leggere una tupla di specifica corrispondente al template di specifica utilizzato sul tc designato

- **in_s**, **inp_s** permettono di estrarre una tupla di specifica corrispondente al template di specifica utilizzato sul tc designato

- **no_s**, **nop_s** permettono di verificare la presenza di tuple di specifica corrispondenti al template di specifica dato nel centro bersaglio

- **set_s** consente di sovrascrivere l'insieme di tuple di specifica presenti nello spazio di specifica del tc bersaglio

- **get_s** consente di leggere tutte le tuple di specifica presenti nello spazio di specifica del centro di tuple designato

Come per le operazioni di coordinazione, nelle operazioni di meta coordinazione sono presenti le fasi di invocazione e di completamento.

Reazioni ReSpecT
Come già anticipato nelle sezioni precedenti, ReSpecT funge da linguaggio di specifica del comportamento in quanto consente di definire computazioni all'interno di un *Tuple Centre*: tali computazioni prendono il nome di **reazioni**.
Più precisamente una reazione è definita come una sequenza di predicati logici, funzioni e primitive ReSpecT, le quali vengono eseguite nel complesso atomicamente e in modo transazionale, per cui la loro semantica è globalmente di successo o di fallimento. ReSpecT consente di associare gli eventi che riguardano il tc alle reazioni:
"Dato un evento *Ev* e una tupla di specifica *reaction(E,G,R)*, tale tupla associa una reazione $R\theta$ ad *Ev* se e solo se θ = mgu(*E,Ev*) (mgu indica il *most general unifier*, così come definito in programmazione logica) ed il predicato di guardia G è verificato" [5]

Vista la loro natura transazionale, le reazioni che non hanno successo non hanno effetto sullo stato del *Tuple Centre*.
Sequenze di reazioni sono eseguite in modo sequenziale e non deterministico; inoltre esse sono eseguite atomicamente, ovvero vengono eseguite prima di servire altri eventi ReSpecT.

1.2.5 Ciclo di esecuzione della VM ReSpecT

La VM ReSpecT è costituita sostanzialmente da un FSA (*Finite State Automata* - Automa a Stati Finiti), il cui ciclo di esecuzione inizia nel momento in cui è eseguita l'invocazione di una primitiva da parte di un agente o dal tc stesso.

1. Viene generato un evento ϵ.
2. L'evento ϵ raggiunge il tc designato (ossia lo stesso di destinazione della primitiva invocata).
3. L'evento ϵ viene inserito in modo ordinato in una coda di input (*Input Queue* - *InQ*).

InQ è processata soltanto nel momento in cui la VM si trova nello stato di *idle*, il che significa che nessuna reazione è in esecuzione:

[5] cfr. Andrea Omicini. Formal ReSpecT in the AA perspective. Electronic Notes in Theoretical Computer Science, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

1. Il primo evento ϵ (ossia il più vecchio, in conformità con la politica FIFO) presente nella coda *InQ* è spostato nel *multiset Op* delle richieste in attesa.

2. Le reazioni associate all'invocazione di ϵ sono attivate attraverso la loro aggiunta al *multiset Re*, nel quale sono contenute le reazioni innescate.

3. Successivamente, le reazioni presenti in *Re* i cui predicati di guardia sono verificati vengono predisposte per essere eseguite, mentre le altre sono rimosse da *Re*.

4. Infine, sono eseguite le reazioni ancora presenti in *Re*, in maniera sequenziale e non deterministica.

Ognuna delle reazioni eseguite in *Re* può scatenare:

- Ulteriori reazioni, aggiunte in maniera ordinata ad *Re*

- Eventi di output, che rappresentano invocazioni di operazioni di concatenazione (*linking operation*). Tali operazioni consistono in primitive invocate su un altro centro di tuple; esse sono asincrone in modo da non influenzare la semantica transazionale delle reazioni.
  Grazie a questa proprietà i centri di tuple ReSpecT sono collegabili tra loro, e qualunque reazione ReSpecT può invocare qualsiasi primitiva di coordinazione su un qualsiasi centro di tuple presente in rete.

  Gli eventi di output sono depositati inizialmente nel *multiset Out* che rappresenta gli eventi in uscita, per poi essere inseriti nella coda *Output Queue* (*OutQ*) del tc se e solo se la reazione che ha generato l'evento di output viene eseguita con successo.

La fase conclusiva del ciclo di esecuzione della VM si ha quando *Re* è vuoto:

1. Ulteriori richieste in attesa presenti in *Op* sono, se possibile, eseguite.

2. il completamento delle operazioni e/o delle primitive di link ritorna ai rispettivi invocatori.

1.3 Aspetti avanzati

- ***Bulk Primitives:*** Al fine di trattare con maggiore efficienza gruppi di tuple sono state introdotte primitive *bulk* (ossia primitive "di massa"), le quali consentono di utilizzare una singola operazione di coordinazione per

restituire l'intero insieme di tuple che corrispondono al template dato.[6]
Tali metodi sono completati con successo anche nel caso in cui nessuna tupla corrisponda al template fornito: in questo caso veine restituita una lista di tuple vuota.
Le primitive di coordinazione di massa messe a disposizione dal linguaggio di coordinazione TuCSoN sono 4:

  - **out_all:** permette di inserire nel tc designato la lista di tuple data.
  - **rd_all:** consente di leggere tutte le tuple presenti nel tc che corrispondono al template dato.
  - **in_all:** consente di estrarre tutte le tuple presenti nel tc che corrispondono al template dato.
  - **no_all:** permette di verificare l'assenza di tuple che corrispondono al template fornito nel centro di tuple bersaglio.

- **Primitiva *spawn:*** Al fine di delegare attività computazionali di coordinazione al medium di coordinazione stesso, TuCSoN fornisce la primitiva di "*spawn*" con la quale viene attiva una computazione parallela gestita in modo asincrono rispetto al chiamante. L'esecuzione di *spawn* è locale al centro di tuple nel quale è invocata, così come i suoi risultati: pertanto non è possibile effettuare operazioni remote utilizzando questa primitiva.

- **Operazioni sincrone e asincrone:** Le operazioni di coordinazione elencate nelle sezioni precedenti possono essere invocate in due modalità: nel caso sincrono il flusso di controllo dell'agente chiamante è bloccato in attesa del completamento dell'operazione, mentre nel caso asincrono i flussi di controllo del chiamante e dell'operazione di coordinazione sono disaccoppiati, permettendo pertanto all'agente di mantenere la propria autonomia.
Dettagli sulle API per le operazioni asincrone disponibili e sul funzionamento dell'invocazione di operazioni asincrone in TuCSoN sono rintracciabili in "Asynchronous Operation Invocation in TuCSoN"[7]

- ***Role-Based Access Control:*** L'*Access Control* è un meccanismo di sicurezza di estrema importanza, tale da essere incorporato nella definizione stessa di sicurezza dell'*RFC 2828 Internet Security Glossary*, ed è definito dall'ente di standardizzazione *ITU-T* (*International Telecommunication Union – Telecommunication Standardization Bureau*) come

[6] cfr. Antony Ian Taylor Rowstron.Bulk Primitives in Linda Run-Time Systems. PhD thesis, The University of York, 1996.

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

"la prevenzione dell'uso non autorizzato di risorse, inlcusa la prevenzione dell'uso di risorse in modo non autorizzato"[8]

L'*Access Control* fornisce delle politiche che specificano quali tipi di accesso sono consentiti, sotto quali circostanze e da chi. Tra gli approcci utilizzati quello scelto dall'infrastruttura TuCSoN è quello basato sui ruoli (da qui il termine *Role-Based Access Control*). Tale approccio si basa su 3 elementi principali [1.4]:

- **Utenti:** Sono i soggetti, ossia le entità (processi, thread, agenti ecc.) capaci di accedere ad oggetti (in questo caso ai centri di tuple).
- **Ruoli:** Sono il livello intermedio tra i soggetti e gli oggetti, ossia ciò che disaccoppia l'utente dalla risorsa: non è infatti l'identità dell'utente a fornire le modalità di accesso, bensì il ruolo che l'utente assume. I ruoli definiscono un insieme di privilegi, modalità e vincoli di accesso che l'utente si trova a rispettare nel momento in cui li impersona.
- **Risorse:** Sono gli oggetti sui quali è controllato l'accesso.

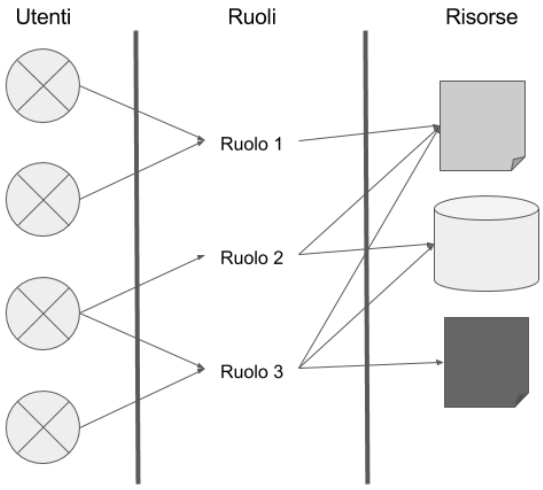

Figura 1.4: Esempio schematico di RBAC

[8] cfr. http://www.itu.int/en/ITU-T/Pages/default.aspx

Per aderire all'RBAC-TuCSoN, è necessario che gli agenti acquisiscano un *meta-ACC*: si ricorda che un ACC è l'interfaccia che abilita l'esecuzione di operazioni sul tc da parte degli agenti e ne vincola le interazioni possibili, pertanto è l'astrazione che permette all'infrastruttura di gestire, oltre alle questioni di coordinazione, le questioni di sicurezza. Il passo seguente è quello dell'attivazione di un ruolo da parte dell'agente al fine di acquisire un ACC: ciò si traduce nel relazionare l'agente, a seconda del ruolo, ad un *Agent Coordination Context* dotato preventivamente di un filtro che consente solo le operazioni consentite in relazione al ruolo dell'agente.

Un approfondimento in merito a tale tematica è accessibile alla documentazione ufficiale[9]

[9]http://apice.unibo.it/xwiki/bin/view/TuCSoN/Documents - "RBAC In TuCSoN"

Capitolo 2

Tolleranza agli errori: tipologie e meccanismi

Oggetto cardine di questo elaborato è la tolleranza agli errori di cui la persistenza rappresenta un elemento. É importante fornire una panoramica estesa su questo argomento al fine di comprendere l'importanza dei contributi illustrati nelle sezioni successive e di chiarire in maniera generale ma esaustiva le declinazioni che tale tematica può assumere.

2.1 Caratteristiche generali

In un sistema distribuito, caratterizzato dunque da molteplici entità ed attività, può verificarsi con probabilità maggiore che un componente smetta di funzionare come previsto: se è vero che in questo contesto risulta difficile evitare completamente errori e guasti, è pero necessario poter mantenere, nelle forme possibili, il sistema in funzione. La ***Fault Tolerance*** (tolleranza ai guasti) è la proprietà di un sistema di erogare le proprie funzionalità anche in presenza di guasti. Per garantire tale tolleranza bisogna prestare attenzione ad alcune proprietà fondamentali:

- **Disponibilità:** Meglio nota con il termine *Availability*, indica la probabilità di trovare il sistema accessibile, in quanto il suo funzionamento è corretto. Questa proprietà è importante anche per il dimensionamento di un sistema in termini di accessi al secondo.

- **Affidabilità (*Reliability*):** Essa è la misura del tempo in cui il sistema riesce ad essere in esecuzione con continuità, ossia il tempo medio prevedibile tra due guasti in successione. Maggiore è l'affidabilità, maggiore sarà anche la disponibilità del sistema.

- **Sicurezza:** Il termine più adatto a questa proprietà è *Safety*. Essa non è quantificabile con precisione, e vuole indicare la capacità del sistema di gestire i fallimenti, per cui un guasto non comporta risultati catastrofici al sistema.

- **Manutenibilità:** Meglio nota col termine *Maintainability*, essa è la grandezza probabilistica che esprime il tempo medio necessario per la riparazione dei guasti; pertanto descrive la facilità con cui i guasti sono corretti. Un sistema con un'alta manutenibilità è di conseguenza un sistema con un alto grado di disponibilità. Una tematica strettamente correlata a questa proprietà è la possibilità di riparare un componente a *"run-time"* (si parla in questo caso di *"recover from failure"*).

2.2 *Faults*

I guasti (chiamati appunto col termine *"fault"*) sono intesi come una deviazione di qualunque natura rispetto al comportamento previsto del sistema. Descrivendo una semplice catena di cause ed effetti, si dice che un sistema "fallisce" quando manifesta comportamenti non previsti e non desiderati; un errore è una parte dello stato del sistema che può aver causato un fallimento, mentre la causa di un errore è un guasto. I malfunzionamenti dipendono da una grande varietà di fattori, quali problemi di rete, bug nel software, danni all'hardware ed errori dell'utilizzatore. I guasti vengono classificati in tre categorie principali:

1. Guasti transienti: essi si manifestano una sola volta per poi svanire, pertanto non sono riproducibili, ma solo osservabili tramite meccanismi di log. Ne è un esempio il mancato invio di un messaggio sulla rete (time out), il quale viene inviato correttamente la volta seguente.

2. Guasti intermittenti: non bloccano l'esecuzione del sistema ma si manifestano periodicamente. Ne è un esempio una connessione di rete instabile.

3. Guasti permanenti: essi continuano ad esistere finché il componente guasto non viene riparato o rimpiazzato. Ne sono esempi i bug software e i danni all'hardware. Solitamente tali problemi causano *Partial Failure* (fallimenti di porzioni del sistema) e nei casi più gravi bloccano l'esecuzione del sistema.

Le categorie di guasti sopra elencate si differenziano ulteriormente in guasti *"fail-silent"* e guasti *"Byzantine"*: nei primi il componente guasto smette di funzionare e pertanto non produce più output (oppure produce output di notifica del guasto), nei secondi il componente guasto continua a funzionare producendo un output errato, risultando dunque più problematici da gestire.

2.2.1 Principali tecniche di gestione dei guasti

È possibile distinguere tre tipologie di controllo dei guasti:

- Prevenzione: l'obiettivo è quello di cercare di minimizzare le possibilità di guasto, per cui si parla di ***"Fault Avoidance"***, ossia si cerca di garantire, attraverso le fasi di design e di validazione, che il sistema sia accurato.

- Rimozione: procedura che consiste nel tentare di rimuovere guasti che si sono palesati, attraverso verifiche ed esami.

- Tolleranza/Previsione: questo approccio coincide con la definizione di **Fault Tolerance**; è prevista la possibilità di avere malfunzionamenti nel sistema, perciò l'obiettivo è quello di costruire sistemi che sono in grado di compensare i guasti e continuare a funzionare correttamente.

2.3 Ridondanza e persistenza

2.3.1 Ridondanza: caratteri generali

L'approccio generale con cui si costruiscono sistemi in grado di compensare i guasti è quello della ridondanza.
La ridondanza è attuabile a più livelli:

- **Ridondanza fisica:** essa riguarda i dispositivi, e consiste nel fornire al sistema componenti aggiuntivi in grado di far fronte alla perdita di eventuali dispositivi guasti. Ne sono un esempio i Server di backup e i dischi *RAID*(*Redundant Array of Independent Disks*).
  Nel caso fisico occorre distinguere il concetto di ridondanza dal concetto di replicazione: mentre quest'ultima infatti consiste nella creazione di più unità identiche, le quali operano in contemporanea (e quindi in maniera concorrente), la ridondanza prevede l'impiego di una sola unità funzionante, mentre le altre sono disponibili a colmare l'eventuale interruzione di quest'ultima.

- **Ridondanza di informazioni:** questa strategia tenta di fornire tolleranza agli errori mediante la copia o la codifica dei dati; un esempio tipico di codifica è il codice di Hamming il quale consente, previa introduzione di bit supplementari (ridondanti) nel messaggio originale, di rilevare e correggere un certo quantitativo di bit errati.

- **Ridondanza temporale:** consiste nell'esecuzione ripetuta della stessa operazione, utilizzando opportuni meccanismi (ad esempio ritrasmissione a timeout). Questa metodologia risulta efficace in particolare contro guasti tran-

sienti ed intermittenti, mentre non produce benefici in situazioni di guasti permanenti.

Una forma di ridondanza che merita una particolare attenzione ai fini dell'elaborato è la persistenza. Essa è una forma di ridondanza di informazioni che consiste nella replica dei dati.In particolare essa é intesa come la capacità dei dati di perdurare anche in seguito alla terminazione del componente che li ha generati. I dati replicati sono mantenuti con l'obiettivo di poter essere ripristinati a seguito di perdite di informazioni causate da guasti.

2.3.2 Ridondanza: benefici, costi e prestazioni

La ridondanza è dunque un metodo che migliora l'affidabilità, la disponibilità, la sicurezza e la manutenibilità del sistema, ma ha sicuramente un impatto sulle prestazioni del sistema: nel caso della ridondanza di informazioni ad esempio si ha un aumento di bit per ciascun dato o addirittura copie dello stesso dato, il che comporta l'impiego di memoria aggiuntiva e di tempo e risorse computazionali per codificare o replicare i dati. Anche la ridondanza temporale comporta uno spreco in termini di tempo e risorse a causa della ripetizione delle operazioni. La ridondanza fisica ha principalmente un cattivo impatto sui costi in quanto necessita di infrastrutture aggiuntive. È inferibile che più un sistema cerca di essere tollerante ai guasti, più esso è dispendioso, perciò la questione si sposta verso una comparazione di costi e benefici: pragmaticamente è necessario interrogarsi su quanto sia costoso fornire un servizio altamente robusto e su quali siano piuttosto i tempi di indisponibilità del sistema accettabili, oltre che chiedersi come tale robustezza impatti le performance, quanto siano frequenti effettivamente tali situazioni anomale e cosa comporti la mancata gestione del guasto.

Capitolo 3

Gestione della persistenza "idle-to-idle"

3.1 Persistenza come meccanismo di fault tolerance

Come descritto nel capitolo precedente, una forma di fault tolerance può essere erogata attraverso una corretta gestione della persistenza dei *Tuple Centre*: questo aspetto non solo permette di evitare la perdita delle informazioni presenti nei nodi, ma consente anche, con meccanismi opportuni, di poter riprendere eventuali operazioni interrotte a causa dell'indisponibilità del *Tuple Centre*: lato Agente, è possibile strutturare dei blocchi di verifica del successo delle operazioni tramite le API di *ITucsonOperation*, pertanto sono ipotizzabili scenari in cui un *Tuple Centre* non è più disponibile, e l'Agente rimane in attesa del suo ripristino, per poi poter richiedere l'operazione che è stata interrotta.

3.2 La persistenza in TuCSoN

Allo stato attuale TuCSoN supporta la persistenza sia dello spazio di tuple ordinario che di quello di specifica: essa è abilitata tramite l'inserimento della tupla speciale *cmd(enable persistency([tcid]))* in un apposito centro di tuple chiamato *$ORG*, il quale comporta a sua volta l'inserimento della tupla *is_persistent* nel centro di tuple che si desidera rendere persistente.
Una volta attivato tale meccanismo di persistenza, viene generato un file *XML* che riporta informazioni quali il nome completo del tc, la data e l'ora dell'abilitazione della persistenza ecc., oltre che il contenuto vero e proprio del *Tuple Centre*, organizzato in modo incrementale: appena la persistenza è abilitata viene creato

un nodo di *Snapshot*, una fotografia del contenuto del tc espresso in forma di tuple, tuple di specifica e predicati, mentre al presentarsi di aggiornamenti sul *Tuple Centre* vengono generati nodi di *Update* che descrivono l'operazione effettuata sul tc, il tipo di tupla oggetto del cambiamento e la tupla vera e propria. Ad esempio la rimozione di una tupla dal centro causerebbe la creazione di un nodo di update che descrive l'azione di cancellazione di tale tupla.
Il ripristino (ossia l'operazione di *Recovery*) è effettuato automaticamente sul *Tuple Centre* non appena esso torna disponibile.

3.2.1 Problematiche emerse

Durante la fase di analisi è stato spontaneo interrogarsi su quali altre informazioni fosse necessario mantenere per garantire il ripristino e anche la possibilità di riprendere il funzionamento desiderato. È stato osservato che, per come funziona il ciclo di esecuzione della Virtual Machine ReSpecT, le uniche informazioni rilevanti sono proprio quelle mantenute nell'attuale livello di persistenza. Tuttavia sono presenti alcuni aspetti problematici.
Un primo aspetto problematico è emerso durante alcune fasi di debug della VM e degli aggiornamenti di persistenza al fine di comprendere l'esatta sequenza di gestione delle strutture dati dedicate: gli aggiornamenti al file di persistenza non sono eseguiti solamente durante lo stato di *idle* della VM, ma anche durante gli altri stati. Questo comportamento non garantisce la consistenza e l'atomicità degli aggiornamenti in quanto si possono avere situazioni in cui l'operazione viene invocata quindi scritta su file di persistenza e nel frattempo il tc va in crash, ottenendo per cui la situazione in cui, al ripristino, viene ripristinata una falsa immagine del *Tuple Centre*.
Un altro argomento di discussione è la scelta della modalità di mantenimento delle informazioni effettuata. La scelta di salvare in maniera incrementale e differenziale l'evoluzione dello stato del tc risulta probabilmente molto rapida in fase di backup ma in fase di recovery non è in grado di offrire buone prestazioni, in quanto è necessario ricalcolare tutte le operazioni svolte per determinare lo stato.
A questo proposito è interessante notare il seguente caso di studio, il quale non entra nel merito dell'implementazione ma concettualmente permette di inquadrare meglio la problematica.
Si consideri di avere un *Tuple Centre* con cui interagiscono due agenti A e B. Il comportamento che si vuole avere è una sorta di "ping pong" testuale estremamente banale, in cui l'agente A interagisce con B inserendo nel tc la tupla *ping()* (previa consumazione della tupla *pong()* qualora sia presente), mentre l'Agente B, se trova presente nel *Tuple Centre* la tupla *ping()*, inserisce la tupla *pong()* [3.1].

Risulta evidente che la cardinalità massima del centro è di una tupla. Attivando la

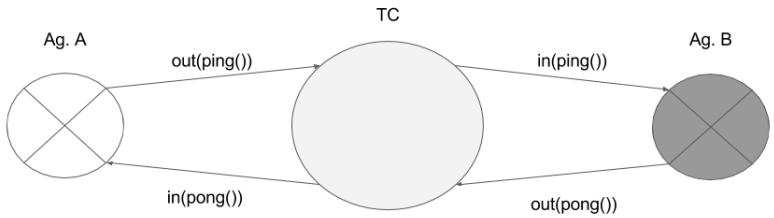

Figura 3.1: Rappresentazione dell'esempio proposto

persistenza sul tc ed eseguendo questa sequenza di scambi di informazioni tra i due agenti per un numero considerevole di cicli avremo, in caso di crash e successiva recovery, un notevole spreco di tempo per ricalcolare uno stato assai poco corposo. È chiaro che esistono numerosi casi pratici in cui si verificano situazioni analoghe, dove il numero di tuple presenti nei tc ha una bassa cardinalità e sono più frequenti aggiornamenti a tali tuple che aggiunte vere e proprie. Un esempio di tale categoria è rintracciabile nei sistemi che si basano sul pattern *"publish/subscribe"*: è possibile immaginare sistemi in cui un'entità (un Agente) aggiorna costantemente un numero limitato di informazioni (tuple) e un gruppo di Agenti in ascolto si limita a leggere tali informazioni[3.2].

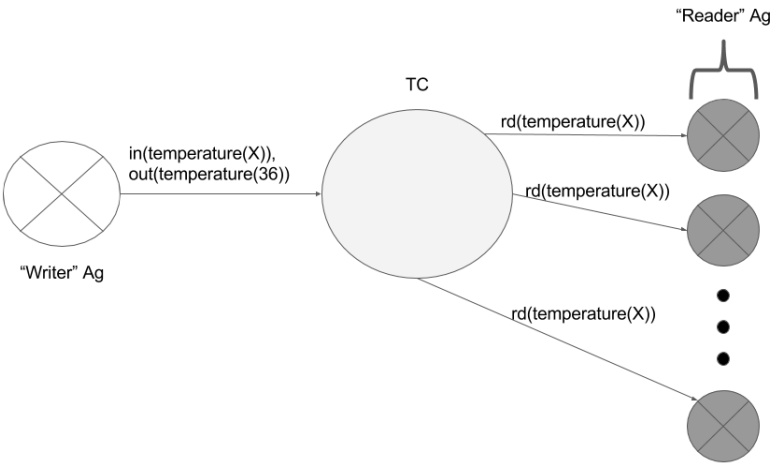

Figura 3.2: Esempio di architettura in cui la cardinalità del tc rimane limitata

Più in generale, sarebbe più semplice avere già un'immagine precisa fruibile per la fase di recovery: qualitativamente parlando il dominio di scenari in cui ciò si rivelerebbe più prestante è decisamente ampio.

Il problema degli scenari in cui è presente l'aggiornamento ripetuto delle tuple presenti riguarda anche la fase di scrittura del file di persistenza: se ad esempio una tupla logica viene rimpiazzata con una tupla avente lo stesso *template* ma nuovo contenuto informativo (o più generalmente viene rimpiazzata), ciò si traduce con due scritture di update nel file persistente, una che notifica la cancellazione della vecchia tupla e l'altra che notifica l'inserimento di quella nuova. Pertanto, in casi in cui è frequente l'aggiornamento delle informazioni contenute nel centro di tuple, si rivela essere poco prestante un sistema di tracciamento degli aggiornamenti così costituito.

3.3 Soluzione proposta: persistenza a Snapshot

L'idea che sta alla base di questo metodo di erogazione della persistenza alternativo è quella di rendere più veloce la ripresa dell'esecuzione dopo un arresto anomalo, oltre che correggere la mancanza di consistenza nella scrittura del file di persistenza. Concettualmente si vuole realizzare un meccanismo che, al termine di ogni ciclo di esecuzione della VM, generi una fotografia (*Snapshot*) del *Tuple Centre*.
Potenzialmente il beneficio dell'utilizzo di questa strategia può estendersi anche alla fase di scrittura del file di persistenza, pertanto saranno allegati anche alcuni test per confrontare entrambe le fasi e trarre conclusioni più fondate.

3.3.1 Compatibilità ed interoperabilità

Come detto nel paragrafo precedente, le informazioni da gestire nel caso *"idle-to-idle"* sono le stesse gestite attualmente dal meccanismo di persistenza; si può dunque pensare che il metodo proposto possa affiancare e combinarsi con il metodo già esistente. A questo proposito verranno proposti nelle sezioni successive alcuni test di verifica delle prestazioni, ma è opportuno precisare fin d'ora che, nell'ottica di poter unire le due metodologie in futuro, a livello di architettura si è scelto di mantenere la stessa divisione in moduli presente allo stato attuale, anche per non impattare altri aspetti del sistema in caso di grossi *refactory*.

3.3.2 Cenni sull'architettura e gli strumenti utilizzati

Molto sinteticamente, l'estensione si basa sulla modifica della classe *ReSpecTVmContext*: è stato inserito al suo interno il metodo privato *writeSnapshot()*, il quale è responsabile della scrittura dello stato attuale del tc su file. Tale metodo viene richiamato all'interno della funzione *enablePersistency*, la quale precedentemente

avviava la scrittura su file XML dello stato iniziale del tc (ossia il nodo di snapshot). Sono stati eliminati tutti i richiami alla funzione writePersistencyUpdate, presente sempre all'interno di *ReSpecTVmContext*, poiché non più necessari: sarebbe stato errato inoltre mantenere come riferimento per gli aggiornamenti i punti del codice in cui si richiamava tale metodo per i problemi di consistenza descritti in precedenza.
Per la parte di recovery invece è necessario mettere mano al metodo *checkPersistentTupleCentres* della classe *TucsonNodeService*, il quale è responsabile del controllo, in fase di avvio, di eventuali centri di tuple che necessitano il ripristino in quanto persistenti. La modifica del metodo è banale e resa necessaria dal fatto che è stata cambiata la costruzione del nome del file di persistenza e la sua estensione: è stato utilizzato il formato Json, perchè ritenuto più compatto e più semplice da mappare su oggetti java.
Per gestire in maniera più agevole le parti di serializzazzione e deserializzazione è stato fatto ricorso alla libreria **Jackson**[10], famosa per la sua praticità e prestanza. In particolare essa consente di mappare un il contenuto di un file Json in un "POJO" (*plain old Java object*) e viceversa in maniera estremamente semplice: per fare ciò è necessario costruire una classe "template" contenente i campi che si vogliono estrarre o introdurre nel file Json, e i rispettivi *"getter"* e *"setter"*, oltre che eventuali metodi di utility a seconda delle necessità. Per maggiore chiarezza qui di seguito è mostrata la classe template utilizzata per lo snapshot del centro di tuple:

```
package alice.tucson.persistency;

import java.util.Date;
import java.util.List;

import alice.logictuple.LogicTuple;
import alice.ReSpecT.core.TupleSet;
import alice.ReSpecT.core.tupleset.AbstractTupleSet;
import alice.ReSpecT.core.tupleset.ITupleSet;
import alice.ReSpecT.core.tupleset.TupleSetCoord;

public class PersistencyDataVM {

   public static class TupleCentreID{
      private String name;
      private int port;
      private String node;
```

[10]https://github.com/FasterXML/jackson
http://wiki.fasterxml.com/JacksonInFiveMinutes

```
   public String getName(){
      return this.name;
   }
   public int getPort(){
      return this.port;
   }
   public String getNode(){
      return this.node;
   }

   public void setName(String n){
      this.name=n;
   }
   public void setPort(int i){
      this.port=i;
   }
   public void setNode(String n){
      this.node=n;
   }
}

private List<String> tupleSet;
private List<String> specTupleSet;
private List<String> prologPredicateSet;
private Date date;
private TupleCentreID tcId;

public TupleCentreID getTupleCentreID(){
   return this.tcId;
}

public void setTupleCentreID(TupleCentreID id){
   this.tcId=id;
}

public Date getDate(){
   return this.date;
}

public void setDate(Date d){
   this.date = d;
}
```

```
public List<String> getTupleSet(){
   return this.tupleSet;
}

public void setTupleSet(List<String> tSet){
   this.tupleSet = tSet;
}

public void addTuple(String t){
   this.getTupleSet().add(t);
}

public void removeTuple(String t){
   this.getTupleSet().remove(t);

}

public List<String> getSpecTupleSet(){
   return this.specTupleSet;
}

public void setSpecTupleSet(List<String> stSet){
   this.specTupleSet = stSet;
}

public void addSpecTuple(String t){
   this.getSpecTupleSet().add(t);
}

public void removeSpecTuple(String t){
   this.getSpecTupleSet().remove(t);
}

public List<String> getPrologPredicateSet(){
   return this.prologPredicateSet;
}

public void setPrologPredicateSet(List<String> predSet){
   this.prologPredicateSet = predSet;
}

public void addPredicate(String t){
```

```
      this.getPrologPredicateSet().add(t);
   }

   public void removePredicate(String t){
      this.getPrologPredicateSet().remove(t);
   }

   public String toString(){
      return this.tupleSet.toString()
           +"\n"
           +this.specTupleSet.toString();
   }
}
```

Viene dunque riportato anche il funzionamento del *mapper* di Jackson che permette di scrivere su file Json (o leggere da esso):

```
package alice.tucson.persistency;

import java.io.File;
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerationException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;

public class PersistencyVMJSON {
   ObjectMapper mapper;

   public PersistencyVMJSON(){
      mapper = new ObjectMapper();
      mapper.enable(SerializationFeature.INDENT_OUTPUT);
   }

   public void writeSnapshot(String path, String
       filename,PersistencyDataVM persistentVM){
      try {

         mapper.writeValue(new File(path+filename+".json"),persistentVM);
      } catch (JsonGenerationException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
```

```
      } catch (JsonMappingException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      } catch (IOException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }

}
```

Per ulteriori delucidazioni, si rimanda alla documentazione del codice.

3.4 Test comparativi

I piani di test presenti in questa sezione sono fondamentalmente a scopo di benchmarking tra il nuovo metodo di erogazione della persistenza e la vecchia strategia, al fine di valutare i domini in cui ciascun metodo prevale in efficacia rispetto all'altro.
Tali testo sono contenuti nel package *alice.tucson.persistencyTest*.

3.4.1 Jetm

Per condurre i test nel modo più accurato possibile è stato fatto affidamento sulla libreria **Jetm**(*Java*^{TM} *Execution Time Measurement Library*)[11], la quale consente di valutare il tempo di esecuzione di particolari parti di codice. Essenzialmente tale libreria mette a disposizione costrutti per poter inserire dei marcatori finalizzati a delimitare la parte di codice che si desidera monitorare e poter stampare in output il tempo impiegato, oltre ad informazioni come il numero di volte in cui un frammento di codice è chiamato, il tempo minimo e il massimo impiegati per tale codice, tempo totale e tempo medio.

- **Test 1: Tempi di creazione dello snapshot date diverse cardinalità del centro di tuple**
  Il primo piano di test prevede l'inserimento di un numero fisso di tuple per poi abilitare la persistenza al fine di osservare il tempo impiegato a creare il file di backup in funzione di tale numero. Va ricordato che i due metodi di erogazione della persistenza sono differenti anche in termini di numero di metodi utilizzati: mentre il vecchio meccanismo prevedeva un primo blocco di istruzioni per creare il file persistente e il primo snapshot, per poi utilizzare il metodo di update per gli aggiornamenti successivi, il nuovo metodo utilizza lo stesso metodo sia per il primo snapshot che per quelli successivi. A livello pratico ciò si traduce in stampe di output Jetm con composizioni diverse, di cui è fornito un esempio:

```
|------------------------------------------------------------|---|---------|--------|--------|--------|
|                     Measurement Point                      | # | Average |   Min  |   Max  |  Total |
|------------------------------------------------------------|---|---------|--------|--------|--------|
| ReSpecTVMContext:enablePersistency(part of snapshot write) | 1 |  56,175 | 56,175 | 56,175 | 56,175 |
|------------------------------------------------------------|---|---------|--------|--------|--------|
| ReSpecTVMContext:writepersistencyUpdate)                   | 1 |   7,409 |  7,409 |  7,409 |  7,409 |
|------------------------------------------------------------|---|---------|--------|--------|--------|
```

[11] http://jetm.void.fm/

Nel primo caso, relativo al vecchio sistema di persistenza, si osservano due diversi punti di misura, il primo relativo allo snapshot vero e proprio, il secondo relativo ad un update. Pertanto il numero di chiamate non viene incrementato né tantomeno i valori di massimo, minimo, il tempo medio e il tempo totale .

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1	184,294	184,294	184,294	184,294

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	2	94,768	5,242	184,294	189,535

Nel secondo caso invece, si osserva una sequenza di due chiamate dello stesso metodo; dunque il numero di chiamate # risulta incrementato così come il tempo totale, e i valori medio, di minimo e di massimo sono aggiornati.

I risultati presenti ai punti successivi tengono conto di questa differenziazione e sono pertanto già normalizzati.
Note: Jetm mostra che ad un singolo snapshot corrispondono, in entrambi i casi, due chiamate a metodo per fatto che l'inserimento della tupla speciale *cmd(enable persistency([tcid]))* abilita la persistenza causando la scrittura dello snapshot iniziale, e comporta inoltre l'inserimento della tupla *is_persistent* nel tc, la quale viene a sua volta inserita nel file di persistenza a seconda delle modalità previste dalle due strategie (dunque come update nel primo caso e come nuovo snapshot nel secondo caso).

a)Cardinalità del tc = 10 tuple

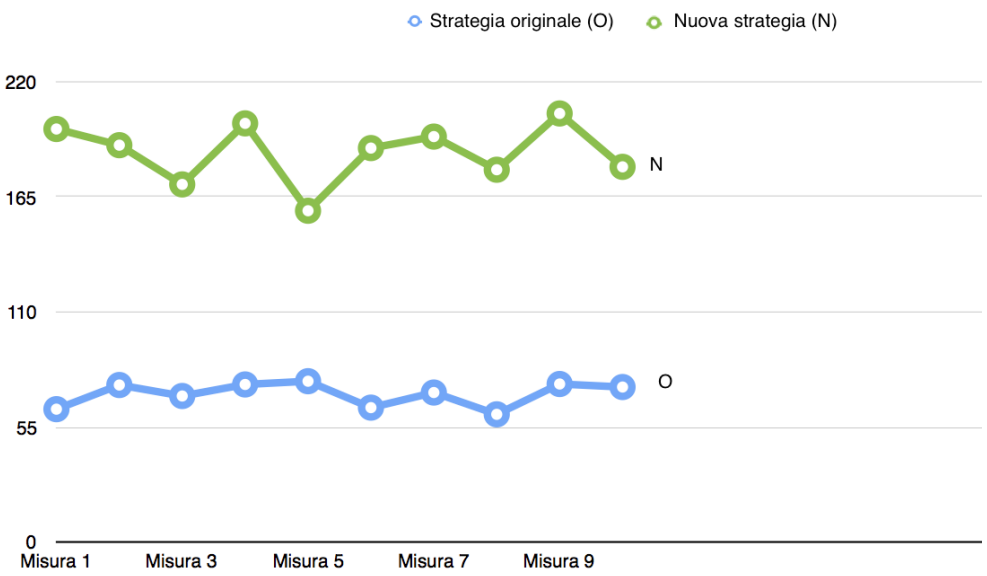

Figura 3.3: Tempi (in millisecondi) relativi a 10 misurazioni, con card(tc)=10

Note: un dato non visibile dal grafico riguarda il tempo medio di inserimento della singola tupla *is_persistent* dopo la creazione del primo snapshot. Esso è mediamente di 9,130 ms nel caso originale, e di 5,559 ms nell'estensione proposta.

b)Cardinalità del tc = 1000 tuple

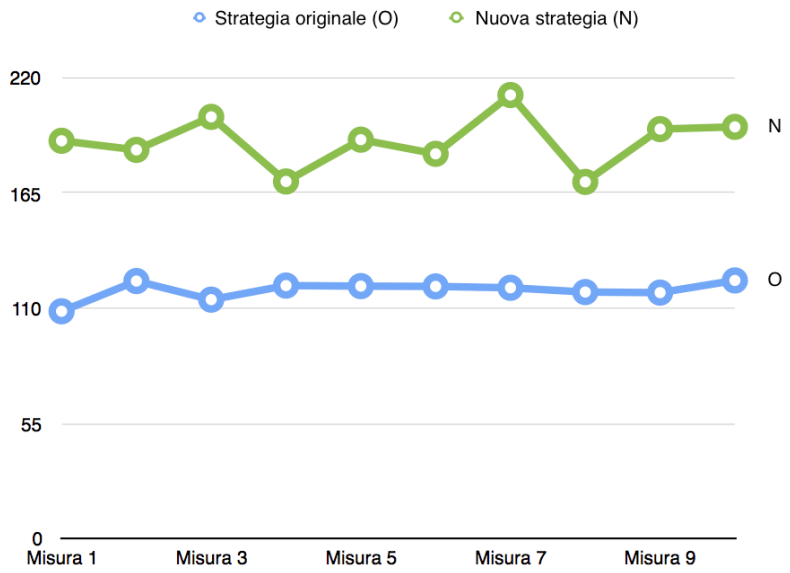

Figura 3.4: Tempi (in millisecondi) relativi a 10 misurazioni, con card(tc)=1000

Note: si osserva, rispetto al precedente gruppo di test, un andamento simile al precedente nella nuova strategia introdotta, la quale si mantiene costante nonostante il tc sia aumentato di due ordini di grandezza. Il vecchio meccanismo di persistenza ha subito invece un degrado più rilevante, ma comunque rimane più prestante. Il tempo di inserimento della singola tupla *is_persistent* ha impiegato in questa serie di misurazioni mediamente 37,684 ms nel metodo tradizionale e 7,243 ms nella nuova incarnazione.

c)Cardinalità del tc = 10000 tuple

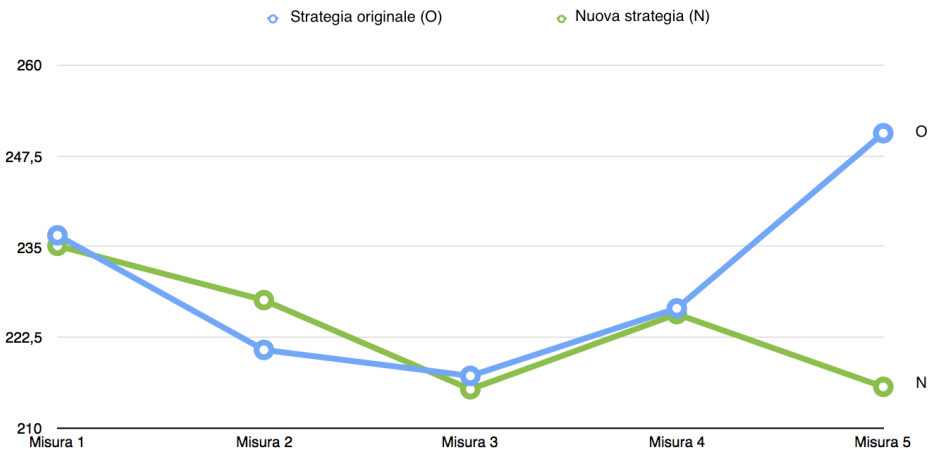

Figura 3.5: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=10000

Note: sono state diminuite le misurazioni per un semplice fatto di rapidità nel valutare gli esiti dei piani. Si osserva che la nuova strategia di persistenza ha prestazioni pressoché identiche alle cardinalità precedenti, non superando la soglia dei 235 ms. Il vecchio sistema di backup si rivela essere ora simile alla sua controparte se non peggiore in alcune battute. Ciò è dovuto anche al tempo di inserimento della tupla singola *is_persistent*, che richiede al sistema nativo mediamente ben 115,796 ms, contro i 19,621 del nuovo contributo.

Considerazioni sulla prima tipologia di test
Da tali risultati sembrerebbe essere gravoso l'inserimento della singola tupla *is_persistent* dopo la creazione dello snapshot per le performance della strategia nativa, tuttavia non è questo il solo fattore di degrado; anche eliminando tale misura, si osserva che il nuovo sistema rimane praticamente costante nei tempi al variare della cardinalità, con al più variazioni trascurabili. Il sistema originale di persistenza invece subisce un aumento nei tempi significativo (seppur non eccessivamente repentino) all'aumentare della cardinalità del tc.

- **Test 2: Tempi di scrittura di tuple in seguito all'abilitazione della persistenza**
  Tale piano di test prevede, dopo l'attivazione della persistenza, l'inserimento ad intervalli regolari di una tupla (senza però rimuovere ciò che si trova nel tc ad ogni passo); tale inserimento è ripetuto lo stesso numero di volte per entrambi i metodi. Per facilitare e rendere più opportuno il funzionamento di Jetm per questo piano di test verranno trattati allo stesso modo i momenti di

snapshot e di update appartenenti al metodo nativo anche a livello di codice: così facendo verrà fornito in risposta un unico punto di misura.

a)Numero di tuple inserite del tc = 100

Misurazioni relative al vecchio sistema di erogazione della persistenza:

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	101	2,578	1,348	54,407	260,336

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	101	2,499	1,345	58,557	252,430

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	101	2,612	1,339	58,937	263,836

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	101	2,686	1,349	65,989	271,277

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	101	2,652	1,388	62,123	267,887

Note: il numero di operazioni svolte per eseguire 100 scritture è di 101 operazioni per il fatto che viene inserita la tupla speciale *is_persistent*.

Misurazioni relative al nuovo sistema di erogazione della persistenza:

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	102	3,591	1,044	175,253	366,285

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	102	3,156	1,019	159,542	321,873

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	102	3,498	1,001	194,056	356,778

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	102	3,107	0,988	159,068	316,922

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	102	3,218	0,968	165,820	328,284

Note: il numero di operazioni svolte per eseguire 100 scritture è di 102 operazioni a causa della diversa strategia di scrittura; durante il primo ciclo *"idle-to-idle"* della VM viene inserita nel file persistente la tupla già presente nel tc (tale tupla viene inserita precedentemente all'abilitazione della persistenza per far fronte alla scelta implementativa dell'infrastruttura per cui non è possibile attivare la persistenza su un tc vuoto, presente peraltro in entrambi i meccanismi di erogazione della consistenza - si osservi il codice per dettagli) e in un successivo ciclo viene inserita la tupla speciale *is_persistent*.

In generale emerge dal confronto dei risultati una migliore performance della vecchia strategia, tuttavia i tempi minimi registrati sono inferiori nella nuova strategia. Osservando inoltre qualitativamente i tempi massimi di entrambi i meccanismi e i tempi totali si desume rapidamente come i primi siano determinanti nelle prestazioni di questi ultimi; come è noto dalla prima tipologia di test condotti, e verificabile eseguendo i test ed osservando le valutazioni Jetm iniziali, tali massimi corrispondono alle operazioni di scrittura dello snapshot di persistenza iniziale, pertanto è ipotizzabile che tale offset di tempo divenga sempre più trascurabile al crescere del numero di operazioni fatte sui *Tuple Centre*.

b)Numero di tuple inserite del tc = 1000
Misurazioni relative al vecchio sistema di erogazione della persistenza:

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	1001	2,101	1,242	59,490	2.103,224

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	1001	2,008	1,239	62,279	2.010,268

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	1001	2,011	1,303	62,311	2.012,703

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	1001	1,991	1,250	56,833	1.992,578

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writepersistencyUpdate)	1001	1,966	1,270	51,848	1.967,735

Misurazioni relative al nuovo sistema di erogazione della persistenza:

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1002	1,224	0,655	167,370	1.226,312

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1002	1,218	0,684	157,488	1.220,289

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1002	1,269	0,670	194,443	1.271,855

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1002	1,249	0,653	184,880	1.251,803

Measurement Point	#	Average	Min	Max	Total
ReSpecTVMContext:writeSnapshot	1002	1,211	0,680	171,128	1.212,993

Note: si osserva un tempo totale superiore nel vecchio meccanismo di persistenza, sia per quanto riguarda il tempo medio impiegato da ciascun aggiornamento, sia per il tempo di esecuzione totale: ciò indica che l'offset iniziale di scrittura del primo blocco di persistenza viene abbondantemente colmato al crescere del numero di operazioni (dato osservabile anche attraverso una comparazione dei valori di media, massimo e minino) ed in particolare rende preferibile l'utilizzo della nuova strategia di erogazione della persistenza.

Considerazioni sulla seconda tipologia di test
In generale si osserva che i tempi di scrittura degli aggiornamenti su file persistenti sono in entrambi i casi sufficientemente bassi (con picchi che si aggirano intorno ai 2 ms).
Un dato di elevata importanza riguarda chiaramente il tempo medio di esecuzione di ogni singola operazione di aggiornamento: esso rivela che, in controtendenza alle ipotesi iniziali, in effetti la riscrittura del file di persistenza ad ogni ciclo della VM è più snella della singola aggiunta di un aggiornamento sull'albero XML: ciò naturalmente può essere attribuito alla diversa tecnologia di riferimento con cui sono organizzati i backup ed è ipotizzabile che l'utilizzo della tecnologia Json al posto di XML nel vecchio sistema di persistenza possa contribuire al miglioramento delle performance.

- **Test 3: Tempi di ripristino del Centro di Tuple**
  Benché al fine di valutare le performance sia più interessante osservare come esse possano essere potenzialmente peggiorate dalle operazioni di backup, in quanto spesso il ripristino si presenta come situazione rara e anomala, è tuttavia rilevante osservare anche le prestazioni relative alla fase di ripristino, che all'inizio di questo contributo sono state oggetto di esame: al fine di garantire la proprietà caratteristica della *fault tolerance* dell' *availability*, ossia la probabilità di trovare il sistema accessibile, e di promuovere un buon grado di *maintainability* (ovvero la facilità con cui viene riparato un guasto del sistema), è preferibile eseguire la fase di recovery il più celermente possibile.
  A questa finalità è stato introdotto un piano di test che valuta i soli tempi di recovery, valutando come essi siano influenzati dalla mole di informazioni da ripristinare. Un ulteriore piano di test è introdotto invece per valutare come il meccanismo di scrittura incrementale tipico del primo meccanismo di persistenza e il meccanismo a snapshot di questo contributo impattino i tempi di ripristino nel caso di aggiornamenti di una stessa tupla.

  Prima variante: valutazione dei tempi di ripristino in funzione della cardinalità del *Tuple Centre*
  Per eseguire in generale i test di ripristino sono necessarie due classi presenti all'interno di *alice.tucson.persistencyTest*: una classe si occupa di inserire, nelle modalità richieste dal test, le tuple necessarie alla valutazione e di abilitare la persistenza, l'altra si occupa semplicemente di avviare il sistema per avviare il ripristino automatico.
  Nel caso di questo primo piano di test le classi utilizzate sono *PerfomanceTestSetUpForRecovery.java*, responsabile di preparare l'ambiente di test, e *PerformanceTestRecover.java*.

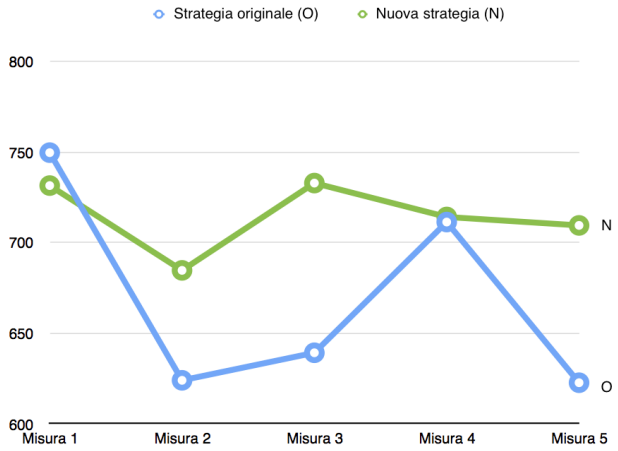

Figura 3.6: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=100

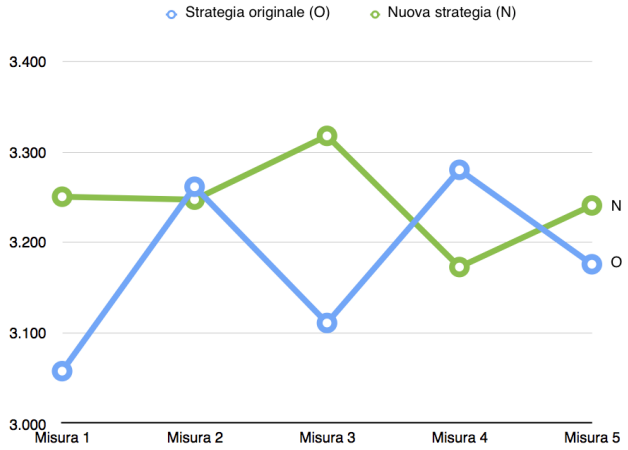

Figura 3.7: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=1000

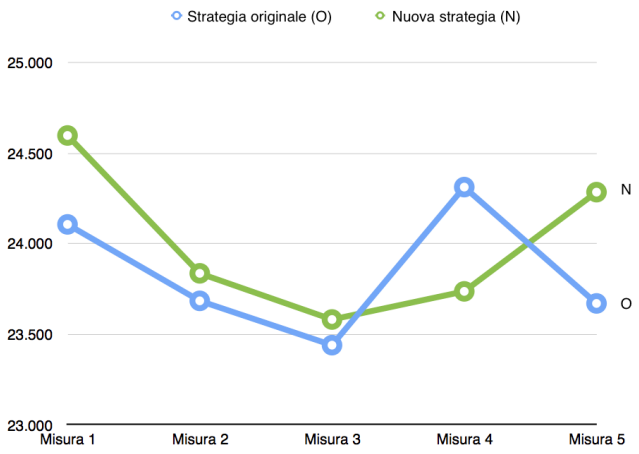

Figura 3.8: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=10000

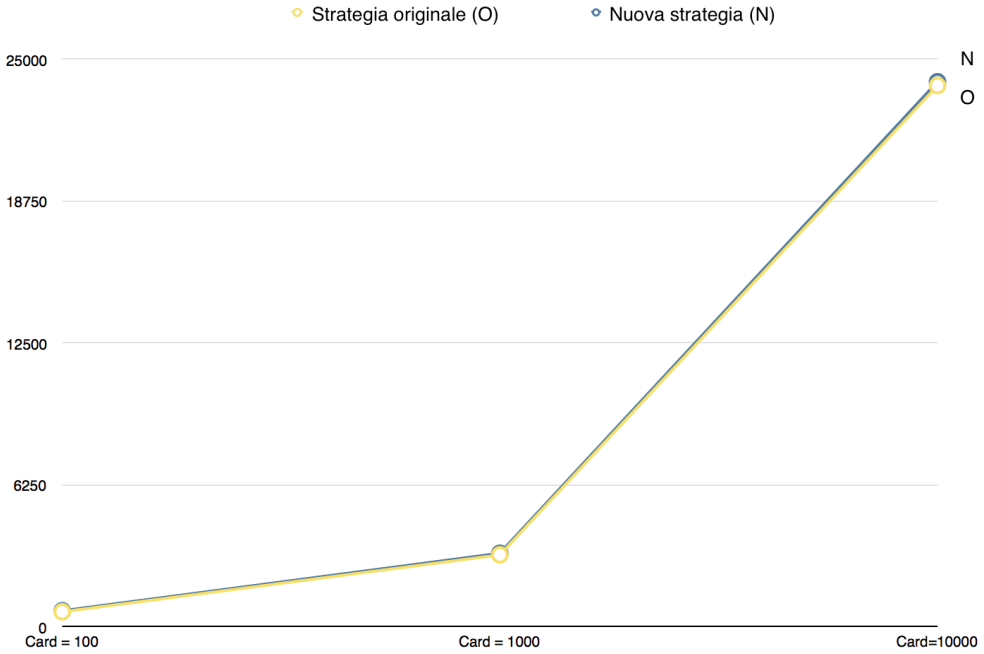

Figura 3.9: Media delle misurazioni eseguite in funzione della cardinalità

Considerazioni in merito alla prima variante
Per motivi di rapidità la serie di misurazioni effettuate è stata diminuita da 10 a 5 (scelta che sarà mantenuta anche nei piani di test successivi): risulta particolarmente ovvio che un tale numero misurazioni ha una validità statistica inferiore, tuttavia permette di estrapolare abbastanza efficacemente l'andamento generale delle prestazioni delle due strategie (nel caso si voglia determinare un profilo statistico più accurato sarebbe opportuno eseguire un gruppo di misurazioni). Detto ciò, si osserva qualitativamente un risultato simile per entrambe le strategie, con un degrado maggiore, seppur lieve, nella nuova strategia: va puntualizzato a difesa di quest'ultima, che essa non è stata ottimizzata allo stato dell'arte attuale, per cui l'algoritmo di ripristino, contenuto nel metodo *recoveryPersistent* di *ReSpecTVMContext* ha sicuramente margini di miglioramento. Va ribadito in ogni caso che la differenza di prestazioni è pressoché impercettibile alle dimensioni del tc prese in esame, e che, alle luce dell'alta variabilità riscontrata durante le misure, sarebbero necessari test più approfonditi determinare con certezza l'effettiva superiorità di un algoritmo sull'altro.

Seconda variante: valutazioni dei tempi di ripristino in funzione delle operazioni di aggiornamento eseguite sul *Tuple Centre*
Tale variante consiste in una serie di aggiornamenti al contenuto del Centro di tuple. I test sono organizzati in modo tale che la cardinalità del centro di tuple sia fissa mediante aggiornamenti, in forma di rimozioni e aggiunte, delle tuple presenti. Naturalmente questa tipologia di test ha una variabilità maggiore in quanto i parametri che ne condizionano l'esito sono due: il numero di tuple iniziali e il numero di aggiornamenti effettuati. Avendo tuttavia osservato che la cardinalità ha un impatto piuttosto simile, è obiettivo di questi test osservare soprattutto l'impatto del numero di sovrascritture.

a)Numero di tuple presenti nel tc = 100 ; numero di aggiornamenti per ciascuna tupla = 10

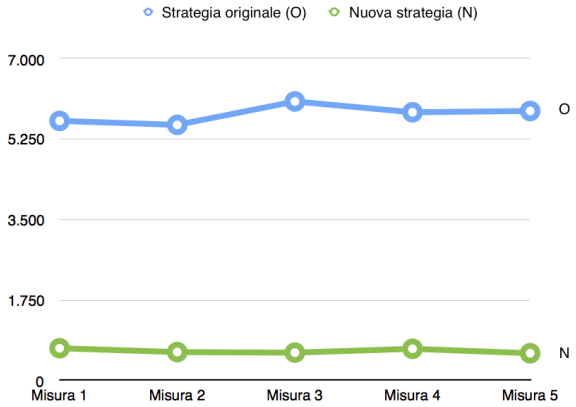

Figura 3.10: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=100 e 10 aggiornamenti per ciascuna tupla

Note: tempo medio per la strategia di persistenza originale = 5,8 secondi; tempo medio per la nuova strategia = 641,7 millisecondi.

b)Numero di tuple presenti nel tc = 100 ; numero di aggiornamenti per ciascuna tupla = 50

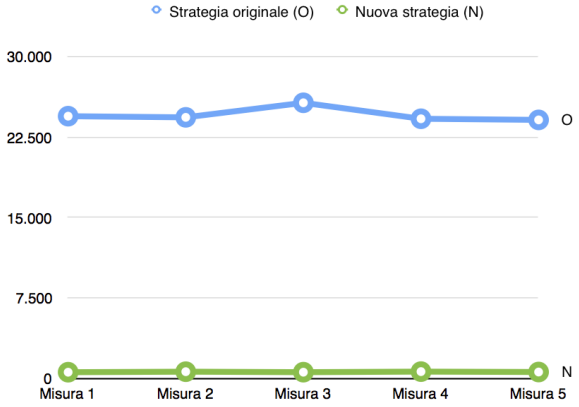

Figura 3.11: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=100 e 50 aggiornamenti per ciascuna tupla

Note: tempo medio per la strategia di persistenza originale = 24,6 secondi; tempo medio per la nuova strategia = 632,2 millisecondi.

Considerazioni in merito alla seconda variante

La situazione rappresentata in questo piano di test in generale vuole ripercorrere il caso limite in cui le tuple presenti sono, nella loro totalità, aggiornate continuamente, e il loro numero non subisce variazioni. È tuttavia sensato pensare che questo caso limite sia più vicino al caso tipico di quanto non lo sia il caso in cui le tuple vengono inserite nei tc in maniera immutabile: per sua natura TuCSoN non si riduce ad essere un semplice deposito di informazioni, ma alla luce delle caratteristiche viste, quali il comportamento programmabile dei tc ed il ruolo di coordinatore tra entità del sistema, appare evidente come la norma sia quella di avere centri di tuple estremamente dinamici ed in continuo mutamento, perlomeno nella maggior parte degli scenari di utilizzo possibili. Un caso particolare è quello dell'utilizzo di alcune *bulk cordination primitive* come *in_all* il cui effetto potenziale è quello di rimuovere tutte le tuple logiche del tc: in questo caso estremo la vecchia strategia, si troverà a ricalcolare lo stato del tc al momento dell'abilitazione della persistenza, e tutti gli aggiornamenti successivi comunque, con un notevole spreco di risorse.

Inoltre un altro aspetto estremamente rilevante, non catturato nei primi gruppi di test, riguarda le prestazioni in fase di scrittura: come accennato inizialmente in questa sezione, gli aggiornamenti continui influiscono in maniera negativa sulle prestazioni della vecchia strategia di persistenza in fase di scrittura, sia per il fatto che una operazione di aggiornamento si traduce in due scritture su file persistente, sia per la scelta implementativa e tecnologica fatta, per la quale ogni singola scrittura porta in sé una grande verbosità.

I test hanno mostrato come 50 operazioni di sovrascrittura su 100 tuple presenti nel tc (ordine di grandezza basso considerato il potenziale flusso di dati e tempo di vita dell'infrastruttura) impattino in maniera decisamente diversa i due sistemi, con un tempo medio di 87,5 secondi nel primo caso contro i 3,9 secondi della nuova strategia.

Entrando nel merito dei risultati del test, per quanto siano composti da poche misurazioni, essi tendono a confermare le aspettative riguardo al nuovo meccanismo realizzato: esso si rivela costante a prescindere del numero di aggiornamenti eseguiti. Focalizzando invece l'attenzione sul vecchio meccanismo di persistenza, osserviamo come esso sia decisamente meno performante; già con sole 10 riscritture osserviamo un tempo abbondantemente sopra i 5 secondi, circa 9 volte il tempo richiesto dalla nuova strategia, per poi arrivare ad un degrado pari a 39 volte circa il tempo richiesto da quest'ultima.

Il grafico sottostante mostra in maniera più apprezzabile tale differenza:

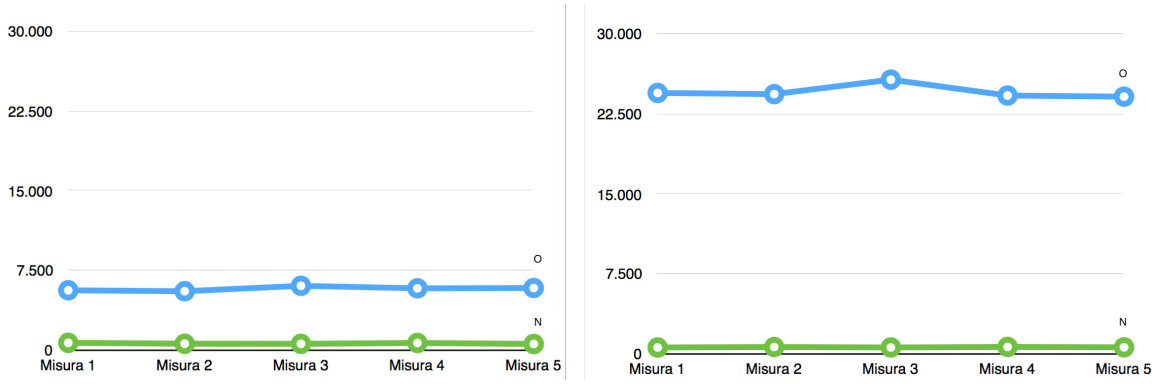

Figura 3.12: Confronto dei risultati: a sinistra è riportato il caso di un numero di aggiornamenti pari a 10, a destra il numero di aggiornamenti è pari a 50

3.4.2 Considerazioni complessive in merito ai test effettuati

I test eseguiti confermano che la fase di ripristino del contenuto dei tc è più rapida se si utilizza la nuova strategia di persistenza: il test di ripristino in funzione della sola cardinalità del centro di tuple mostra come i due metodi siano mediamente equivalenti, ma non permette di osservare appieno il funzionamento della vecchia strategia di erogazione della persistenza. Senza introdurre aggiornamenti (di qualunque natura, sia modifiche che aggiunte o rimozioni) non è possibile valutare la parte di update che compone la strategia di persistenza originale, pertanto è considerata solo la parte di snapshot, che chiaramente (a meno di limiti imposti dalle diverse tecnologie) è piuttosto simile. La parte rilevante riguarda piuttosto il caso in cui il tc subisca variazioni durante il suo ciclo di vita, caso che dovrebbe corrispondere in linea di principio al caso tipico: in questo caso in effetti si osserva un netto miglioramento della nuova strategia di persistenza, principalmente per il fatto che dispone direttamente del contenuto informativo da iniettare nel *Tuple Centre* che si sta ripristinando.
Per quanto riguarda le parti di test inerenti alla sola fase di backup, i risultati rivelano che la vecchia strategia è poco efficiente per grandi cardinalità di tuple presenti inizialmente nel tc e per cospicui aggiornamenti: le prestazioni della nuova strategia sono invece sostanzialmente indipendenti dalla cardinalità iniziale e soprattutto si rivelano essere più prestanti durante gli aggiornamenti del file persistente. L'unica pecca che possiede il nuovo sistema è un offset di tempo più corposo nella scrittura dello snapshot iniziale rispetto alla strategia classica, per cui in caso di piccole cardinalità iniziali e di una mole esigua di aggiornamenti non riesce a colmare tale svantaggio.
Vista la grande varietà di casistiche in cui il nuovo metodo di erogazione della persistenza si rivela più prestante si può affermare che, allo stato attuale, la nuova

strategia sia consigliata, non solo perché risponde efficacemente al problema dell'inconsistenza dei backup, ma anche per una migliore qualità del servizio, che si riflette anche in una migliore fault tolerance in termini di velocità di recupero da guasti, di disponibilità e di *safety*.
Entrando nel merito delle prestazioni, una nuova coordinata di esplorazione può interessare le specifiche scelte tecnologiche, osservando nel dettaglio come esse impattino lo stato dell'arte attuale. Una possibile strategia a questo proposito può essere quella di utilizzare la tecnologia Json anche nel vecchio sistema di erogazione della persistenza per osservare in che misura le scarse performance siano dovute all'uso dell'XML.
Naturalmente la vecchia strategia ha il pregio di conservare tutte le interazioni che hanno interessato il centro di tuple, per cui può rilevarsi utile in termini di monitoraggio e storicizzazione della vita di tale tc: ulteriori indagini future possono essere fatte in questo senso.

3.5 Persistenza su un flusso di controllo indipendente

Una prima, semplice ottimizzazione consiste nel creare un Thread separato per gestire la persistenza in fase di scrittura; tale modifica in realtà si rivela essere utile in entrambi i meccanismi presi in esame. I benefici di tale modifica consistono prevalentemente in un aumento di performance dell'infrastruttura TuCSoN, in quanto il flusso di controllo della VM ReSpecT non è più coinvolto nella scrittura su file di backup. Tale modifica trova anche una motivazione concettuale nel fatto che l'operazione di scrittura della persistenza è estranea alla logica applicativa della Virtual Machine ReSpecT, inoltre va considerato che nel caso tipico ci si aspetta che un tc sia disponibile in rete ed un evento di crash sia una anomalia non troppo frequente; è svantaggioso utilizzare il flusso di controllo principale dunque per un'operazione che (seppur in piccola parte) degrada le prestazioni del sistema e che con buona probabilità si rivela essere rilevante con una frequenza estremamente bassa.

L'implementazione di tale estensione consiste nella creazione di un "Worker Thread" all'interno della classe *RespectVMContext*, il quale è responsabile della scrittura su file persistente: osserviamo a questo proposito il metodo *writeSnapshotWithPerformanceTest*.

```
private void writeSnapshotWithPerformanceTest(){

    etmMonitor = EtmManager.getEtmMonitor();
    setup();

    this.persistencyWorkerThread = new Thread(){public void
        run(){System.out.println("*** worker Thread
        ***");writeSnapshot();}};
    this.persistencyWorkerThread.start();
    try {
      this.persistencyWorkerThread.join();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }

    monitor.aggregate();
    monitor.render(new SimpleTextRenderer());
    tearDown();

}
```

Va ricordato che questo metodo, prima dell'aggiunta del flusso di controllo separato, aveva la funzione di "wrapper" del metodo writeSnapshot, al fine di effettuare un monitoraggio mediante libreria Jetm adeguato. In questa incarnazione viene sfruttato per gestire il nuovo Thread, al quale è affidato il compito di scrivere lo snapshot.

Un aspetto di importanza cruciale riguarda la sincronizzazione dell'accesso alle strutture della VM soggette a backup: al fine di garantire la consistenza degli snapshot è opportuno coordinare le modifiche ai multiset delle tuple (logiche, di specifica e predicati *Prolog*) da parte del flusso di controllo principale rispetto alle operazioni di lettura di tali multiset dal Thread adibito alle scritture su file.
La soluzione adottata è quella di gestire le potenziali corse critiche trasformando i metodi di lettura e scrittura propri delle strutture dati interessate in metodi *synchronized*: tale dicitura indica che il metodo in questione può essere eseguito da un solo flusso di controllo per volta. Nel dettaglio:

- I set delle Tuple logiche e di quelle di specifica sono istanziati in *RespectVMContext* come oggetti di tipo *TupleSetCoord* e *TupleSetSpec* rispettivamente: tali tipologie estendono a loro volta la classe astratta *AbstractTupleSet*. È pertanto sufficiente definire all'interno della classe astratta quali metodi debbano essere *synchronized*.

- Il set dei predicati Prolog fa invece riferimento alla classe *TupleSet*: è dunque in questa classe che sono definiti i metodi *synchronized*.

Per dettagli riguardanti tale ottimizzazione si rimanda al codice sorgente delle classi menzionate.

3.5.1 Comparazione qualitativa delle prestazioni

I seguenti test hanno lo scopo di verificare se effettivamente le prestazioni derivanti dall'aggiunta del nuovo flusso di controllo siano migliori, o perlomeno non introducano un appesantimento dovuto alla gestione di sezioni critiche tale da annullare i benefici scaturiti dalla separazione dei flussi.
Naturalmente tali test riguardano la sola fase di scrittura su file persistente, poiché il ripristino non ha necessità di avere un suo flusso indipendente.

- **Test con cardinalità iniziale del tc pari a 10 tuple**

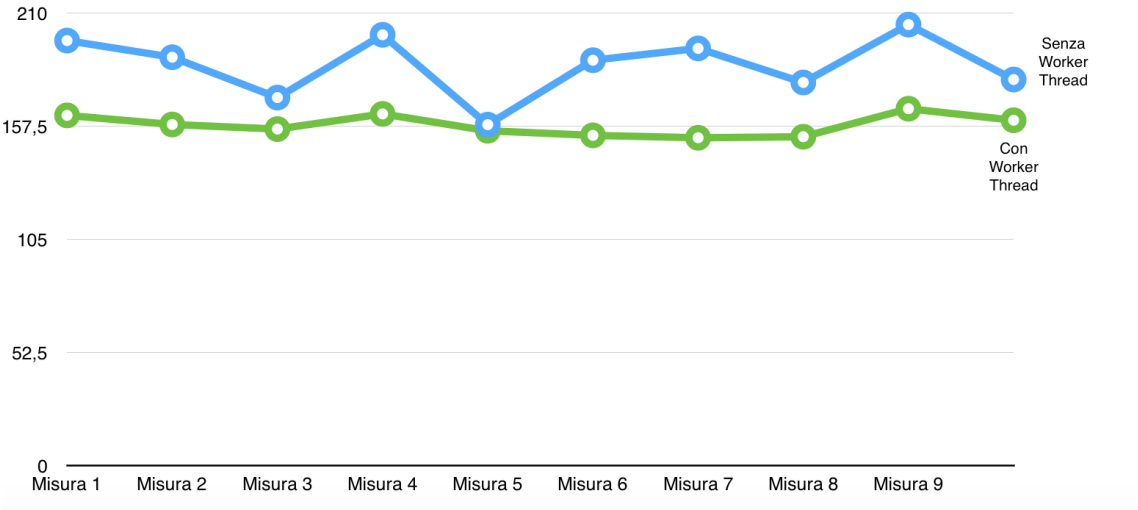

Figura 3.13: Tempi (in millisecondi) relativi a 10 misurazioni, con card(tc)=10

- **Test con cardinalità iniziale del tc pari a 1000 tuple**

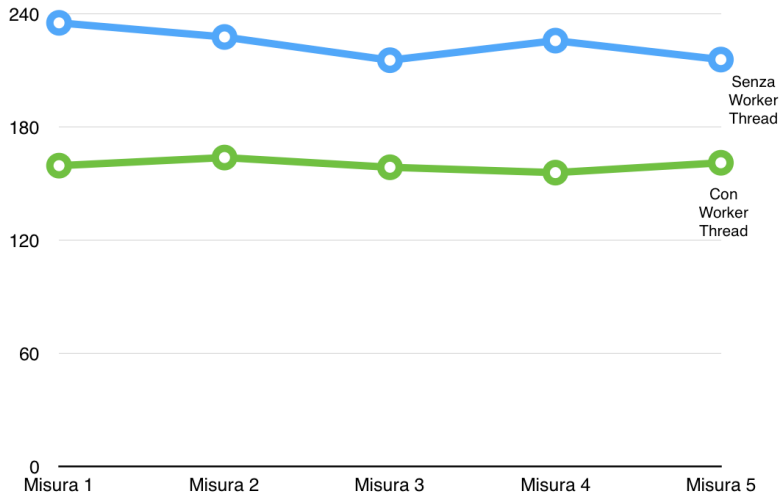

Figura 3.14: Tempi (in millisecondi) relativi a 5 misurazioni, con card(tc)=1000

I test in [3.13] e [3.14], mostrano qualitativamente un andamento che conferma la bontà dell'aggiunta del Worker Thread: in particolare con un numero di tuple più considerevole tale trend risulta ancora più evidente.

Un altro caso contemplato dai test riguarda l'impatto medio che ha l'aggiunta di una tupla ad ogni ciclo della VM: in questo modo può essere valutato il tempo medio richiesto per l'inserimento di un singolo elemento nel file persistente.

- **Test con aggiunta progressiva di 1000 tuple**

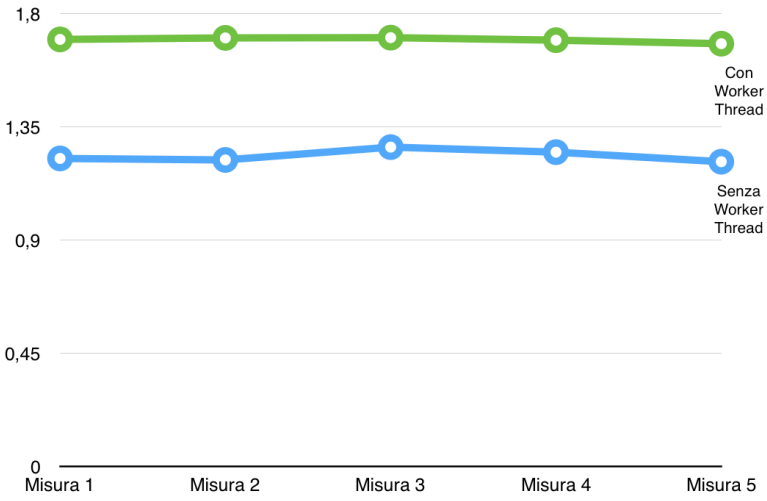

Figura 3.15: Tempi medi (in millisecondi) di scrittura di uno snapshot, relativi a 5 misurazioni

In figura [4.1] notiamo un (seppur relativamente basso) degrado di prestazioni: circa mezzo millisecondo di differenza. Tale degrado può essere attribuito al contesto specifico, in cui la sezione critica è sollecitata maggiormente. Al fine di valutare meglio l'influenza del numero di operazioni eseguite ad ogni ciclo, è stato modificato il corpo del test nel seguente modo:

```
for(int i=0; i< 999; i++){
   acc.out(ttcid, new LogicTuple("tuple", new Value(i+1)),
                    Long.MAX_VALUE);
  //Thread.sleep(200);
```

L'eliminazione della sospensione del Thread che controlla l'esecuzione del test permette alla VM di gestire potenzialmente più operazioni di *out*, pertanto svincoliamo il processamento di una sola operazione (e in questo caso di un solo inserimento

di tupla nel tc) a ciclo.
Di seguito sono riportati i tempi medi:

Measurement Point	#	Average
RespectVMContext:writeSnapshot	944	1,165

Measurement Point	#	Average
RespectVMContext:writeSnapshot	964	1,210

Measurement Point	#	Average
RespectVMContext:writeSnapshot	997	1,133

Measurement Point	#	Average
RespectVMContext:writeSnapshot	968	1,179

Measurement Point	#	Average
RespectVMContext:writeSnapshot	986	1,181

É interessante osservare il numero di scritture su file persistente eseguite; rispetto alle 1002 operazioni (numero che è già stato oggetto di riflessione nelle sezioni precedenti), abbiamo un numero minore e variabile: tale valore indica che le 1000 operazioni effettive effettuate sul tc non sono sempre state eseguite in cicli separati. Concentrando l'attenzione sui tempi medi di scrittura di un singolo snapshot osserviamo un netto miglioramento rispetto al piano di test precedente, il quale conferma che il degrado di prestazioni è legato sostanzialmente alla maggior frequenza di sezioni critiche: dovendo accedere alla sezione critica ad ogni scrittura di snapshot è preferibile poter gestire più operazioni in un unico ciclo

"idle-to-idle", tanto più che il collo di bottiglia causato dai metodi *synchronized* è più probabile ad alte frequenze di scrittura.
Una nota chiarificatrice riguarda la funzione di *sleep* rimossa nel piano di test: tale metodo non influisce sulle misure fatte dal monitor Jetm in quanto non rientra negli estremi di monitoraggio, dunque il miglioramento di prestazioni non dipende da questo aspetto. Ciò che invece ha consentito l'aumento di performance è proprio l'abolizione del vincolo del numero di operazioni gestite ad ogni ciclo, scaturito dalla rimozione di tale *sleep*.

In sintesi, l'aggiunta del flusso di controllo separato per gestire la scrittura su file persistente rappresenta un'importante estensione dal punto di vista architetturale in primis, poiché separa la business logic della creazione di backup da quella della VM, in secondo luogo perché produce, laddove presente, un degrado di prestazioni sostanzialmente trascurabile e relativo a particolari scenari, mentre in buona parte dei casi offre prestazioni migliori, e potenzialmente è in grado di apportare ulteriori miglioramenti futuri alle performance del sistema.

3.6 Sviluppi futuri

La strategia proposta e la sua realizzazione sono sperimentali e non hanno pertanto una completa copertura ed ottimizzazione. Una prima estensione, tesa a rendere più *"user friendly"* il meccanismo di persistenza all'utilizzatore, può riguardare l'abilitazione della persistenza su centro di tuple vuoto (o più precisamente su un tc che non ha avuto ancora interazioni col mondo esterno): l'infrastruttura TuCSoN è strutturata in modo tale che un tc inizi il suo ciclo di vita nel momento in cui viene invocata una operazione su di esso. Si può pensare di introdurre un meccanismo che, in caso di richiesta di abilitazione della persistenza previa attivazione del tc, conservi tale richiesta, per poi inoltrarla al tc non appena esso è avviato. È inoltre necessario costruire un meccanismo di gestione dei guasti ancora più accurato, in quanto rimane ancora possibile la situazione in cui il centro di tuple diventa offline mentre è in corso la scrittura dello snapshot. A questo proposito sono possibili strategie di sicurezza come il mantenimento di una coppia di snapshot, uno relativo allo stato corrente, l'altro relativo all'idle precedente: ad ogni idle lo snapshot più recente rimpiazza il vecchio file mentre quello che viene creato diventa il più recente.
Altre ottimizzazioni osservate riguardano perlopiù aspetti relativi alle prestazioni: una fra tutte è la gestione corretta di cicli in cui vengono fatte operazioni che non alterano lo stato del centro di tuple, attualmente non gestita.

Per quanto riguarda l'esplorazione di altre strategie di erogazione della persistenza

"idle-to-idle", sarebbe ipotizzabile e interessante poter osservare in future indagini l'introduzione di una strategia che utilizzi la tecnologia stessa di TuCSoN per erogare la persistenza: l'idea è quella di utilizzare centri di tuple di backup, possibilmente su Nodi differenti, al fine di fornire una sorta di ridondanza dei componenti. In questo senso si apre una grande varietà di interessanti ripercussioni, come la possibilità di attivare dinamicamente un centro di backup e di instradare le operazioni su di esso in maniera automatica, e molte altre proprietà, tuttavia è chiaro che possono insorgere alcune problematiche: in primis abbiamo una forma di ridondanza che non contempla una vera persistenza, in quanto i dati di backup non sono scritti su file persistente, e sono dunque legati alla vita del centro che li ospita, in secondo luogo una cattiva connessione, nel caso in cui il centro ridondante è su un diverso Nodo può rendere lento il backup e il ripristino, cosa che non accade se il file di backup è locale al centro di tuple. Analisi future volte ad indagare questa possibilità (ed eventuali varianti di essa) possono rivelarsi interessanti e promettenti.

Capitolo 4

Gestione della persistenza tra gli stati della VM

La seconda estensione proposta riguarda la gestione della persistenza ad una granularità più fine: anziché gestire la persistenza ad un ciclo completo della VM, si può pensare di mantenere le informazioni relative alle strutture dati accessorie alla VM per essere in grado di riprendere l'esecuzione di essa dall'ultimo stato completato con successo.

4.1 Considerazioni iniziali

La prima questione da gestire è l'identificazione dei momenti in cui è effettuabile la scrittura su file persistente.
Considerando l'automa a stati finiti che compone la VM, è auspicabile che la scrittura persistente avvenga al completamento delle operazioni proprie dello stato in cui essa si trova. Osservando il problema dal punto di vista della fase di ripristino è invece necessario disporre di quello che sarebbe divenuto lo stato successivo della VM: lo stato corrente, in quanto completato, non necessita di essere eseguito nuovamente. Da queste due riflessioni emerge che il punto ideale in cui effettuare la scrittura è l'arco sotteso dallo stato attuale (lo stato completato con successo) e dallo stato futuro: al momento del ripristino pertanto lo stato da cui riprendere l'esecuzione sarà lo stato futuro.
La seconda questione riguarda le strutture dati da mantenere su file persistente. Oltre al contenuto del tc sono necessarie tutte le strutture, citate nel paragrafo 1.2.5, responsabili del mantenimento degli eventi, delle operazioni e delle reazioni da gestire. Ai fini di realizzare una soluzione meglio ottimizzata inoltre è importante che ogni stato aggiorni le sole strutture dati con cui interagisce, eliminando

perciò l'overhead di scrittura di tutte le strutture su file ad ogni transizione di stato.

4.2 Soluzione proposta

La strategia proposta è quella di attivare la scrittura persistente al termine dell'esecuzione di ciascuno stato: nello specifico la scrittura deve essere eseguita una volta individuato lo stato successivo, ma prima di eseguire la transizione, in modo da realizzare effettivamente la scrittura nell'arco tra i due nodi.
È previsto l'utilizzo di un solo modello dei dati da scrivere su file: ogni VMState aggiornerà tale modello in relazione alle strutture dati di sua competenza.
Per la fase di ripristino si prevede il caricamento delle informazioni presenti su file persistente nelle strutture dati corrispondenti e si designa come stato da cui riprendere l'esecuzione lo stato indicato come stato futuro dal file di persistenza.

4.2.1 Implementazione della fase di backup

La fase di backup viene attivata in modo analogo alla strategia di gestione della persistenza originale di TuCSoN, ossia inserendo la tupla *cmd(enable persistency([tcid]))* nel tc *$ORG*.
Il metodo *enablePersistency* all'interno della classe *RespectVMContext*, responsabile in origine dell'abilitazione della persistenza classica in TuCSoN, ora è responsabile di notificare a tutti gli stati della VM che la persistenza è attiva, oltre che di creare il modello unico dei dati persistenti e inoltrarlo a ciascun VMState. Di seguito è riportato lo stralcio di codice all'interno del metodo che implementa le specifiche appena enunciate:

```
PersistencyDataVMState stateModel = new PersistencyDataVMState();
Map <String, AbstractTupleCentreVMState> states = this.getAllStates();
for(AbstractTupleCentreVMState state : states.values()){
    state.setPersistecyActive(isPersistent);
    state.setPersistentModel(stateModel);
}
```

Naturalmente per poter effettuare tali operazioni è stato necessario introdurre (a livello di classe astratta, *AbstractTupleCentreVMState*) i metodi *setPersistecyActive* e *setPersistentModel*, utilizzati rispettivamente per notificare l'attivazione della persistenza e per impostare il modello su cui scrivere il backup. Spostando l'attenzione verso i VMState, essi sono stati modificati al fine di poter scrivere su file persistente. La classe astratta *AbstractTupleCentreVMState* è stata arricchita dai seguenti campi:

```
private boolean isPeristencyActive;
private PersistencyDataVMState state;
```

Così facendo ogni classe concreta è in grado di gestire l'attivazione della persistenza e possedere il riferimento al modello dei dati da scrivere su file persistente. Ogni stato che estende *AbstractTupleCentreVMState* possiede inoltre i campi di tipo stringa aggiuntivi *fileName* e *filePath* e un costruttore creato ad-hoc per questa estensione. Prendiamo a modello la classe *SpeakingState*:

```
public SpeakingState(final AbstractTupleCentreVMContext tcvm, String
    path, String filename) {
        super(tcvm);
        this.fileName = filename;
        this.filePath = path;

    }
```

In tale maniera ogni VMState ha i riferimenti al file di persistenza su cui scrivere il modello, una volta aggiornato.
Prima di descrivere nel dettaglio il funzionamento di ogni stato della VM, è opportuno riassumere in maniera compatta il comportamento atteso [4.1]:

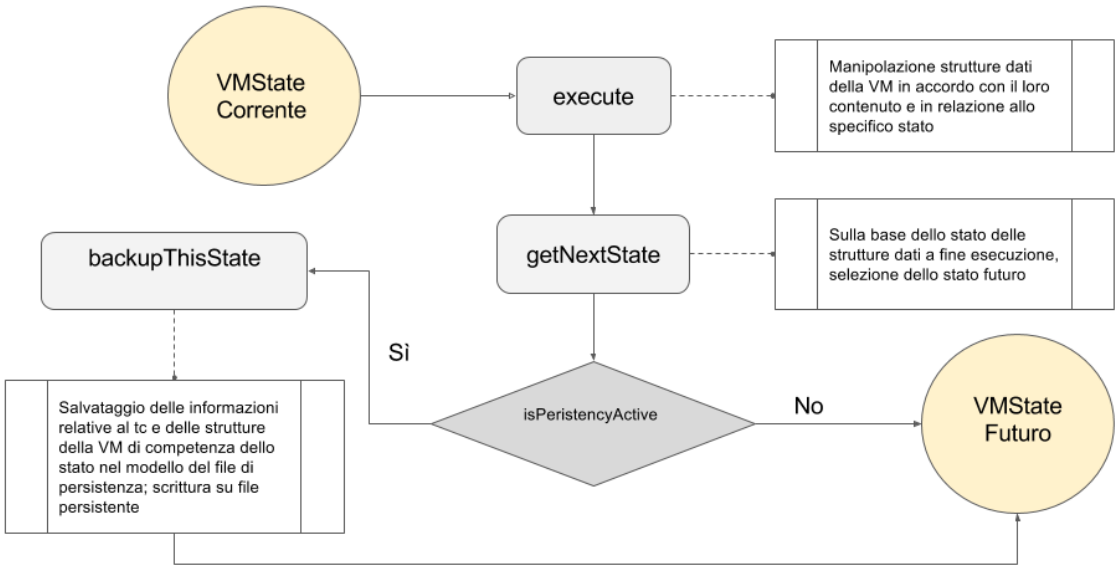

Figura 4.1: Rappresentazione dell'esecuzione di uno stato della VM

4.2.2 Implementazione della fase di ripristino

La fase di ripristino comporta un livello di complessità aggiuntivo rispetto al caso "idle-to-idle": in questo caso infatti non solo è necessario ricostruire le strutture dati accessorie alla VM e il contenuto del *Tuple Centre*, ma è necessario accedere all'automa a stati finiti che costituisce la VM per poter selezionare lo stato da cui riprendere il ciclo di esecuzione.
A tale proposito è stata determinante l'analisi dell'infrastruttura: la classe *AbstractTupleCentreVMContext* seleziona nel suo costruttore come stato iniziale il *ResetState*; tale stato ha un'unica transizione verso l'*IdleState* in quanto è responsabile dell'annullamento del ciclo della VM (inteso come pulizia delle strutture dati e riavvio).
In questo scenario dunque, non appena il sistema viene avviato (o riavviato a seguito di guasti), è possibile unicamente riavviare il ciclo della VM da uno stato di idle. Alla luce di questo, la strategia ipotizzata è quella di abilitare, in caso di persistenza attiva, ulteriori archi di transizione nell'automa a stati finiti della VM, al fine di collegare lo stato di reset con un qualsiasi altro stato della VM: in questo modo durante il ripristino è possibile richiamare il *ResetState*, e una volta ricaricate le strutture dati, percorrere la transizione verso lo stato enunciato dal file persistente. Una nota implementativa riguarda la natura della classe *ResetState*: il metodo *execute* che essa implementa non fa altro che "azzerare" lo stato della VM, pertanto è necessario operare il ripristino delle strutture dati tra l'esecuzione di tale metodo e del metodo *getNextState*, in modo tale che all'invocazione di quest'ultimo le strutture siano effettivamente state ripristinate, o in alternativa, di specificare che nelle situazioni di ripristino in atto, tale stato non dovrà azzerare le strutture dati, delegando il compito alla funzione di ripristino in modo da eseguirla prima del recupero delle strutture dati.

A livello di infrastruttura la fase di ripristino è attivata in maniera analoga alla strategia vista nel capitolo precedente, per cui all'avvio del sistema il *TucsonNodeService* verifica la presenza di file persistenti tramite il metodo *checkPersistentTupleCentres* (nome mantenuto dalle implementazioni precedenti, anche se lievemente forviante in quanto tale metodo non ripristina soltanto il contenuto del tc in questa incarnazione, ma anche lo stato e le strutture di pertinenza della VM): una volta individuato il file da ripristinare, è richiamato a livello di *RespectVMContext* il metodo *recoverPersistentVMState*, responsabile del vero e proprio ripristino delle informazioni oltre che del riavvio del ciclo di esecuzione a partire dallo stato designato.

Per poter meglio comprendere l'estensione all'automa a stati finiti, è qui riportato il codice della classe *ResetState*:

```
package alice.tuplecentre.core;

public class ResetState extends AbstractTupleCentreVMState {

    private AbstractTupleCentreVMState idleState;
    //for recover:
    private AbstractTupleCentreVMState reactingState;
    private AbstractTupleCentreVMState listeningState;
    private AbstractTupleCentreVMState speakingState;
    private AbstractTupleCentreVMState fetchEnvState;

    public String nextState = "";


    public ResetState(final AbstractTupleCentreVMContext tcvm) {
        super(tcvm);
    }

   @Override
    public void execute() {
        if (super.vm.isStepMode()) {
            this.log();
        }
        if(this.nextState != "ListeningState" &&
            this.nextState != "SpeakingState" &&
            this.nextState != "ReactingState" &&
            this.nextState !="FetchEnvState"
            ){
         this.vm.reset();
        }
        //so the vm.reset() is called only during normal lifecycle, not
            in recovery
    }

   @Override
    public AbstractTupleCentreVMState getNextState() {
      switch(this.nextState){
      case "ListeningState": this.clearNextState(); return
          this.listeningState;
```

```
    case "SpeakingState" : this.clearNextState(); return
        this.speakingState;
    case "ReactingState" : this.clearNextState(); return
        this.reactingState;
    case "FetchEnvState" : this.clearNextState(); return
        this.fetchEnvState;
    default: this.clearNextState(); return this.idleState;
    }
  }

  @Override
  public boolean isIdle() {
      return false;
  }

  @Override
  public void resolveLinks() {

      this.idleState = this.vm.getState("IdleState");
      //new arcs:
      this.fetchEnvState = this.vm.getState("FetchEnvState");
      this.reactingState = this.vm.getState("ReactingState");
      this.listeningState = this.vm.getState("ListeningState");
      this.speakingState = this.vm.getState("SpeakingState");
  }

 @Override
 public void backupThisState(String nextState) {
    // do nothing.
 }

 public void setNextState(String s){
    this.nextState = s;
 }

 public String getFutureState(){
    return this.nextState;
 }

 public void clearNextState(){
    this.nextState = "";
 }
```

```
}
```

Lo stralcio di codice riportato necessita di qualche delucidazione:

- Tra i campi sono stati inseriti gli altri stati della VM in modo da poter abilitare le transizioni verso di essi. È presente inoltre il campo *nextState*, il quale è predisposto a contenere la rappresentazione testuale dello stato futuro: in caso di ripristino in corso, tale campo permetterà la transizione verso lo stato designato in quanto ne conterrà la denominazione, altrimenti consentirà la classica transizione verso l'*IdleState*.

- Il metodo *execute* è ora preposto al reset della VM (ossia la pulizia delle strutture dati) tranne che nel caso in cui sia in corso il ripristino: in questo caso infatti esso non svolge nessuna computazione. Tale strategia è stata selezionata al fine di evitare che, in fase di ripristino, il *ResetState* svuotasse le strutture appena recuperate. In fase di ripristino l'istruzione *vm.reset()* verrà effettuata direttamente dal *VMContext*.

- Prima di procedere con l'analisi del metodo *getNextState* è opportuno soffermarsi sui 3 metodi finali della classe: *backupThisState* nello stato di reset non fa nulla in quanto tale stato non deve conservare nessuna informazione. Le funzioni *setNextState* e *getFutureState* permettono di assegnare e di ritornare il valore del campo *nextState*, mentre *clearNextState* funge da funzione di reset di tale campo.

- Il metodo *getNextState* opera una sorta di "pattern matching" sul testo contenuto in *nextState*: il caso di default è chiaramente la transizione verso lo stato di idle, mentre nei casi in cui *nextState* contiene la rappresentazione di uno stato (diverso da idle o reset), il che significa che è in corso il ripristino, si attiva la transizione verso tale stato. Gli stati di reset ed idle sono esenti dall'essere selezionati tramite specifica stringa poiché non hanno impatto sull'FSA: nel caso del *ResetState* non sarà mai presente un backup che faccia riferimento al suo nome (è evidente che tale backup avrebbe le strutture dati vuote, o che un backup che contiene lo stato di reset come stato futuro è inutile in quanto il reset, una volta invocato, cancellerà ogni traccia), mentre lo stato di idle, sia nel caso in cui la persistenza non sia attiva che sia lo stato futuro designato dal ripristino è comunque lo stato in cui il *ResetState* opererà la sua transizione. Prima di effettuare la transizione è necessario tuttavia cancellare il riferimento allo stato futuro, in modo che esso non rimanga accidentalmente impostato, tramite il metodo *clearNextState*.

A livello di pseudocodice, la funzione recoverPersistentVMState, richiamata dal *RespectVMContext* a seguito dell'individuazione di un file persistente opportuno, è composta dai seguenti passi:

```
public void recoverPersistentVMState(final String path, final String
    file, final TucsonTupleCentreId tcName){

    //reset delle strutture dati ad opera del VMContext
    this.reset();

    ObjectMapper mapper = new ObjectMapper();
    //recupero del riferimento al prossimo stato
    PersistencyDataVMState state = mapper.readValue(new File(path+file),
        PersistencyDataVMState.class);

    //setting del prossimo stato al ResetState
    ResetState resetState = (ResetState)this.getState("ResetState");
   resetState.setNextState(state.getNextState());

   //ripristino delle strutture dati:
   /*
   *TupleSet
   *SpecTupleSet
    *PrologPredicates
    *
   *INPUT QUEUE
   *ENV INPUT QUEUE
   *PENDING OPS
   *TRIGGERED REACTIONS
   *TIME TRIGGERED REACTIONS
    *TEMP OUTPUT QUEUE
   */

}
```

4.2.3 Misurazioni qualitative delle prestazioni della fase di backup

Al fine di valutare l'overhead che ogni operazione comporta viene misurato, tramite la libreria Jetm, il tempo di esecuzione della funzione *backupThisState*, presente in tutti gli stati dell'automa. Le misurazioni effettuate si basano principalmente sullo scenario in cui vengono inserite tuple in modo ripetuto in un Tuple Centre,

con possibilità di innescare reazioni o meno, e osservare i tempi minimi, massimi e medi impiegati da ciascuno stato.
Va però specificato che questo insieme di rilevazioni ha una valenza assai sommaria e non ha alcuna pretesa di fornire una precisa statistica: è naturale che, in base alle dinamiche della VM, il metodo *backupThisState* della stessa classe può avere valori assai differenti, influenzando peraltro fortemente la media dei tempi registrati.
Dunque l'obiettivo di questa verifica dei tempi è perlopiù osservare in più momenti il tempo impiegato, per individuare un'intervallo (caso peggiore, caso minore) di riferimento.

Il primo insieme di misurazioni ritrae un caso abbastanza semplice: dopo aver attivato la persistenza, viene letta da file la specifica ReSpecT e subito dopo vengono inserite 10 tuple nella forma *t(X)*, dove X è un numero progressivo: la specifica ReSpecT è costruita in modo da rispondere a ciascun inserimento con la rimozione di tale aggiunta e l'inserimento della tupla *tt(X)*. Ciò significa che, seppure in quantità diverse, il ciclo di esecuzione coinvolgerà gli stati di Idle, Listening, Reacting e Speaking.
Di seguito sono riportati i risultati delle misurazioni:

Measurement Point	#	Average	Min	Max	Total
IdleState:backupThisState	11	19,925	0,234	215,044	219,177
ListeningState:backupThisState	11	0,591	0,183	3,039	6,503
ReactingState:backupThisState	10	0,360	0,251	0,589	3,604
SpeakingState:backupThisState	22	0,372	0,216	0,629	8,180

Risalta in maniera evidente l'alto valore medio registrato nell'*IdleState*, e in modo ancora più netto il suo valore massimo: tuttavia il valore medio è fortemente influenzato dalla prima operazione di scrittura su file persistente: come è stato osservato nei test relativi alla prima estensione, la prima scrittura di persistenza risulta particolarmente onerosa, per poi essere assai più efficiente ad ogni step successivo. É sufficiente, al fine di verificare tale affermazione, dividere il valore massimo per il numero di operazioni svolte: otteniamo come risultato 19,546 millisecondi, il che significa la quasi totalità del tempo medio registrato dipende fortemente da questa scrittura iniziale. Le misurazioni successive comporteranno un numero di scritture superiori al fine di verificare con più precisione come varia l'incidenza della prima scrittura.

A livello globale osserviamo tempi piuttosto bassi ovunque, mediamente inferiori al millisecondo. Sono state eseguite numerose altre esecuzioni dello stesso scenario (non riportate per motivi di praticità, ma replicabili a partire dai sorgenti) che rimangono assai fedeli a quella riportata.

Il secondo insieme di di misurazioni estende il primo per numero di tuple inserite e reazioni innescate. Abbiamo a questo proposito il seguente modulo di specifica:

```
reaction(
reaction(
   out(t(X)),
   (completion,success),
   (in(t(X)),out(tt(X)))
).

reaction(
   out(r(X)),
   (completion,success),
   (in(r(X)),out(r1(X)))
).

reaction(
   out(r(X)),
   (completion,success),
   (out(r2(X)))
).
```

Alla prima reazione dunque sono state aggiunte altre due specifiche; la prima è innescata all'inserimento di una tupla del tipo *r(X)* e produce come effetto la sua rimozione e l'inserimento di una tupla *r1(X)*, la seconda, innescata in maniera analoga, produce come effetto l'inserimento della tupla *r2(X)*.
Per quanto riguarda il numero di tuple inserite, sono previste tre tipologie di esse (*t(X)*, *r(X)*, *f(X)*): sulla base della parte di specifica ReSpecT appena osservata, si inferisce che, a seconda della tupla presa in carico, è possibile avere da zero a due reazioni innescate contemporaneamente, il che si traduce tra l'altro con un pari numero di autoanelli nel *ReactingState*. Le tre tipologie di tuple mostrate sono inserite ciascuna 1000 volte all'interno del tc.

Measurement Point	#	Average	Min	Max	Total
IdleState:backupThisState	3001	11,818	0,184	193,848	35.464,415
ListeningState:backupThisState	3001	10,608	0,117	238,218	31.833,929
ReactingState:backupThisState	3000	11,913	0,196	161,526	35.739,768
SpeakingState:backupThisState	6002	12,394	0,181	529,121	74.390,941

È riportato il grafico delle misure medie[4.2]:

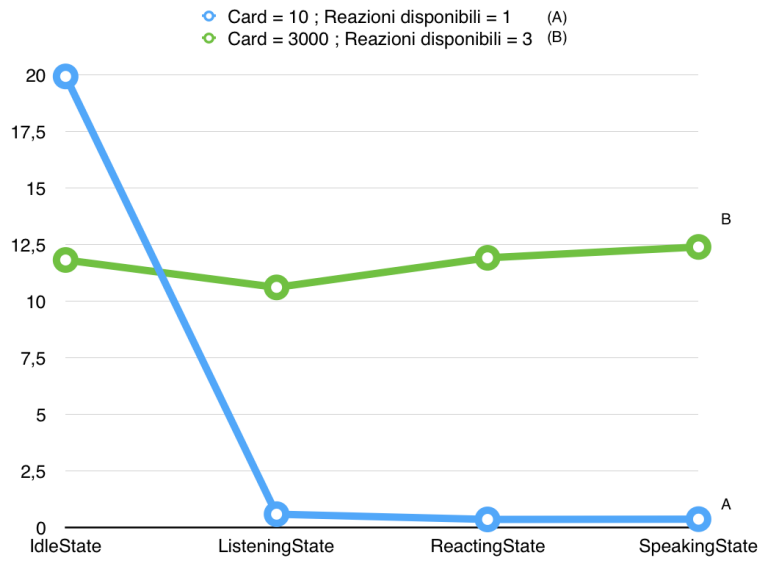

Figura 4.2: Confronto delle prestazioni medie degli stati in relazione alla cardinalità delle operazioni eseguite sul centro di tuple

Ulteriori misurazioni fatte su questo scenario hanno confermato le misurazioni riportate in figura[4.2] : i tempi medi registrati si sono aggirati sempre trai 10 e i 14 millisecondi. Osservando i risultati ottenuti nell'estensione "idle-to-idle" emerge chiaramente un certo degrado di prestazioni: ciò è però abbastanza prevedibile, in quanto le strutture dati da memorizzare sono in un numero maggiore e le scritture sono più frequenti. Detto questo va considerato che l'overhead apportato dall'estensione, in un caso abbastanza rilevante come quello mostrato, in cui sono inserite lato agente 3000 tuple, e a causa delle reazioni innescate un altro quantitativo corposo, rimane comunque contenuto entro i 14 millisecondi: in diversi scenari tale latenza è più che accettabile.

4.3 Considerazioni

La soluzione presentata offre un grado di fault tolerance maggiore, in quanto cattura un intervallo computazionale più ristretto.
L'importanza di tale estensione risiede non solo nella possibilità di recuperare il contenuto informativo e reinserirlo nel tc, ma anche di poter riprendere l'eventuale esecuzione in corso al momento del guasto: nel caso di persistenza "idle-to-idle" è possibile ripartire da una sorta stato "di sicurezza", perdendo tuttavia la computazione corrente, mentre in questo caso la computazione è in grado di essere rieseguita.
Come già accennato, un tale meccanismo di erogazione di persistenza ad una granularità maggiore (intesa in questo caso come numero di strutture dati coinvolte e frequenza di scritture), comporta necessariamente costi aggiuntivi, pertanto è sempre bene valutare il giusto livello richiesto dallo specifico contesto, considerare l'effettiva probabilità di avere guasti, e l'entità del danno che essi comportano.

Le due estensioni viste sono separate ma possono benissimo essere integrate: l'inserimento di due tuple distinte per i due livelli di persistenta e la gestione differenziata dei file persistenti ad opera del *TucsonNodeService* rendono possibile la progettazione di un sistema ibrido, e adattabile alle esigenze dettate dal contesto specifico.

Conclusioni

L'esplorazione delle forme di erogazione della persistenza relative all'infrastruttura TuCSoN è incentrata prevalentemente in questo elaborato sulla ridondanza di informazioni e, in forma minore, di ridondanza temporale.
È opportuno riassumere l'insieme dei contributi apportati all'infrastruttura TuCSoN:

- È stato sostituito il vecchio sistema di erogazione della persistenza con un sistema effettivamente "idle-to-idle" (in cui quindi le scritture su file sono consistenti), che gestisce gli aggiornamenti non più in modo incrementale, bensì come snapshot del contenuto del *Tuple Centre* e utilizza JSON come tecnologia di riferimento per il file persistente.

- Sono state comparate le due soluzioni (il meccanismo di persistenza originale e la nuova strategia) in misurazioni di performance nei vari scenari possibili al fine di delineare un quadro di performance complessivo.

- È stato introdotto un flusso di controllo dedicato alla scrittura persistente, e sono state valutate le implicazioni in termini di concorrenza e prestazioni.

- È stata introdotta un'estensione alla persistenza "idle-to-idle" atta a gestire i guasti durante il ciclo di esecuzione della VM: tale estensione prevede di aggiornare il contenuto del file persistente tra ogni *VMState*, mantenendo dunque, oltre al contenuto del centro di tuple, le strutture dati accessorie alla VM. Al fine di poter riprendere l'esecuzione nello stato in cui era stata interrotta è stata modificata la struttura dell'FSA e modificata l'architettura degli stati ad hoc.

- Sono state condotte misurazioni al fine di comprendere come una soluzione a grana più fine, come quella proposta, impattino le prestazioni, in modo da essere occasione di riflessione sulla tematica della comparazione di costi e benefici.

Naturalmente la sperimentazione effettuata è ancora acerba ma offre spunto per numerose altre esplorazioni. Un aspetto che in particolare necessita di attenzione

riguarda l'effettivo ripristino delle funzionalità, in particolare in merito alla seconda estensione: per una completa ripresa è necessaria una forma di consapevolezza lato agente del ripristino delle funzionalità del tc momentaneamente perdute, tramite opportune strutture di controllo implementabili dall'utilizzatore, o inseribile direttamente nelle specifiche funzionali dei *TucsonAgent*: in questo modo si può ottenere una forma di flessibilità pressoché completa e propria dei sistemi autonomi.

Una considerazione importante riguarda la complessità che scaturisce dall'inserimento di un layer di persistenza: tale livello aggiunto va a coinvolgere il sistema in maniera proporzionale rispetto al grado di tolleranza ai guasti che si vuole raggiungere e alle tipologie e proprietà che si tenta di privilegiare, pertanto è bene concepire fin da subito un sistema dotato di questo genere di meccanismi al fine di evitare costosi refactory ed upgrade.

Entrando nel merito dell'infrastruttura TuCSoN, in un'ottica prettamente incentrata sull'autonomia dei sistemi distribuiti, va sottolineato con forza che la *fault tolerance* è una proprietà di fondamentale importanza. Traendo ispirazione dall'autonomia biologica, il concetto di robustezza è talmente rilevante da essere non solo una parte ben definita dell'autonomia ma addirittura da essere un prerequisito di essa, a garanzia del mantenimento di identità, struttura ed organizzazione. Definendo l'autonomia come la *"caratteristica di mantenere forma e funzione nel tempo e di acquisire una flessibilità auto determinata"*[12], è chiaro che si sta ribadendo che un'efficace gestione dei fallimenti sia un aspetto chiave, e si sta sottolineando nuovamente come sia di primaria necessità fornire una sufficiente consapevolezza degli eventuali guasti in atto da parte degli agenti fruitori e dell'infrastruttura TuCSoN.

In conclusione, la sperimentazione svolta ha permesso di individuare, valutare e comparare differenti strategie di erogazione della persistenza, e di interrogarsi in senso generale sulle proprietà di *Fault Toleance* sulle modalità di erogarle in un contesto distribuito. In ulteriori sviluppi è interessante collaudare soluzioni ibride che permettano di ottenere il grado di persistenza necessario a seconda delle esigenze, evitando peraltro di utilizzare un livello eccessivo (e dunque costoso) nei contesti in cui esso non è necessario.

[12]cfr. Bernd Rosslenbroich. *On the Origin of Autonomy. A New Look at the Major Transitions in Evolutions.* History, Philosophy and Theory of the Life Sciences. Springer International Publishing, 2014.

Bibliografia

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

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

[3] http://apice.unibo.it/xwiki/bin/view/ReSpecT/WebHome

[4] Omicini, A. and Denti, E. (2001). From tuple spaces to tuple centres. Science of Computer Programming, 41(3):277–294.

[5] Andrea Omicini. Formal ReSpecT in the AA perspective. Electronic Notes in Theoretical Computer Science, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[6] Antony Ian Taylor Rowstron.Bulk Primitives in Linda Run-Time Systems. PhD thesis, The University of York, 1996.

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

[8] http://www.itu.int/en/ITU-T/Pages/default.aspx

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

[10] https://github.com/FasterXML/jackson
http://wiki.fasterxml.com/JacksonInFiveMinutes

[11] ttp://jetm.void.fm/

[12] Bernd Rosslenbroich. *On the Origin of Autonomy. A New Look at the Major Transitions in Evolutions.*History, Philosophy and Theory of the Life Sciences. Springer International Publishing, 2014.