La fault-tolerance nei sistemi multi-agente

Emanuele Panzavolta

Approfondimento per il corso di:
Sistemi Multi-Agente L-S

A.A. 2008/2009

Indice

Capitolo 1

Introduzione

I sistemi multi-agente rappresentano un paradigma che si sta lentamente affermando come modello per la progettazione di sistemi complessi e pervasivi, caratterizzati da un livello davvero elevato di distribuzione, dimensione e dinamicità. Tra le sfide che devono essere affrontate per affermare definitivamente il paradigma ad agenti, vi è quella di far sì che i MAS operino in modo corretto ed efficiente anche in ambienti complessi, dinamici e soggetti ad errori e guasti. Questa proprietà in letteratura è comunemente denominata *fault-tolerance*.

In linea di principio, la capacità dei MAS di riorganizzarsi dinamicamente, tramite opportuni protocolli di coordinazione, a fronte di cambiamenti del problema da risolvere, o degli agenti presenti, è senz'altro promettente; tuttavia una fetta considerevole del lavoro concreto sui sistemi multi-agente ha trattato sistemi chiusi, costituiti da agenti fidati, situati su infrastrutture affidabili. Quando invece si considerano contesti aperti e potenzialmente inaffidabili, emerge una serie di problemi da affrontare [5]:

- *Infrastrutture inaffidabili*: i sistemi distribuiti in rete devono affrontare problemi di comunicazione (perdita di connettività, messaggi persi o ritardati, ecc.) o di crash fisici dei nodi;
- *Agenti non fidati*: in sistemi aperti non è garantito che gli agenti si comportino in modo corretto, anzi potrebbero essere stati sviluppati deliberatamente per agire in modo fraudolento;
- *Disfunzioni emergenti*: numerosi meccanismi di coordinazione multi-agente espongono il rischio di produrre disfunzioni emergenti in modo dinamico, come ad esempio il comportamento caotico.

D'ora in poi indicheremo con il termine *fault* qualsiasi avvenimento o situazione che non fa parte del funzionamento normale di un sistema. Se, in occorrenza di un fault, il sistema non è in grado di reagire adeguatamente, e smette quindi di comportarsi nel modo corretto, allora si ha un fallimento di sistema; se viceversa il sistema riesce a fronteggiare la particolare situazione, allora esso è detto fault-tolerant.

Gradi di fault-tolerance In [2] sono presentati tre gradi di fault-tolerance:

- *Full fault-tolerance*: il sistema continua ad operare senza significative perdite di funzionalità o prestazioni, anche in presenza di fault;
- *Graceful degradation*: il sistema prosegue la sua attività, pur con qualche perdita di funzionalità o prestazioni;
- *Fail-safe*: le funzioni vitali sono preservate, mentre altre potrebbero non essere più operative.

Criteri di progettazione [2] indica tre criteri da rispettare nello sviluppo di sistemi fault-tolerant:

1. *Modularità*: essa consente di separare i vari task, conferendo un certo grado di isolamento ai componenti del sistema. Questo fa sì che, in caso di fault, la loro propagazione all'interno del sistema sia impedita o rallentata.
2. *Ridondanza*: la presenza di più copie di una stessa entità, possibilmente su nodi diversi della rete, conferisce maggiore robustezza a fronte di fallimenti fisici, ad esempio crash a livello hardware o software, oppure problemi di comunicazione via rete.
3. *Determinismo*: il linguaggio di programmazione e lo stile di implementazione dovrebbero essere scelti in modo che dal codice si possa dedurre il comportamento del sistema, senza lasciare spazio all'aleatorietà.

Agenti e fault-tolerance L'autonomia è una delle caratteristiche fondanti degli agenti: questa fa sì che essi possano prendere delle decisioni per conto proprio, acquisendo così una certa indipendenza rispetto al resto del sistema. Un agente può proseguire la propria attività anche se nel sistema non vi è nessun altro agente attivo; questo aspetto rende i MAS piuttosto robusti. Tuttavia, conseguenza dell'autonomia è l'opacità: il comportamento di un agente non può essere previsto a priori dagli agenti con cui esso sta interagendo. D'altra parte, è naturale che le entità partecipanti ad una mutua interazione abbiano delle assunzioni rispetto al comportamento dei partner; l'autonomia fa sì che gli agenti possano disattendere queste aspettative, e per l'interlocutore ciò sarà visto come un fault [6]. Queste considerazioni si possono riassumere nel seguente modo:

- i MAS sono intrinsecamente fault-tolerant, perchè l'autonomia degli agenti fa sì che i fault siano isolati, e non si propaghino automaticamente nel sistema;
- i MAS sono intrinsecamente **non** fault-tolerant, a causa del non-determinismo; infatti è difficile garantire comportamenti specifici, soprattutto nelle situazioni di fault.

Questa contraddizione appare difficilmente risolvibile, e nel resto del lavoro si mostrano i principali risultati prodotti in letteratura sull'argomento.

Capitolo 2

I fault nei MAS

2.1 Fault ed eccezioni

Il termine *fault* è spesso associato ad *eccezione*, il quale risulta più familiare in quanto ampiamente usato nei moderni linguaggi di programmazione ad alto livello. Il significato di eccezione è stato definito già negli anni '70, nel seguente modo:

> Tra le condizioni rilevate durante il tentativo di eseguire una qualche operazione, le condizioni di eccezione sono quelle portate all'attenzione di colui che ha invocato l'operazione. Quest'ultimo dunque può (oppure deve) reagire alla condizione. [17]

Se una condizione non è soddisfatta, l'operazione non è eseguita e ciò viene notificato al chiamante, che deve predisporre opportune procedure di exception-handling. Si può perciò affermare che un'eccezione è completamente determinata quando certe condizioni non sono soddisfatte nel contesto dell'invocazione di un'operazione. Però, richiamando il principio di autonomia, un agente è libero di valutare se il risultato dell'invocazione di un'operazione è normale o anomalo, oppure di stabilire quale sia il tipo di eccezione verificatasi; ciò in virtù dell'autonomia e del lasco accoppiamento tra agenti e risorse [1].

2.2 Origine delle eccezioni

Le principali fonti di errori sono quattro [19]:

1. Bug nell'implementazione del software;
2. Errori di progettazione del software;
3. Crash delle macchine;
4. Errori di comunicazione.

(3) e (4) possono, anzi dovrebbero essere tenuti in conto in fase di progettazione dei sistemi. (1) e (2) invece sono sempre elementi imprevisti, di cui in generale non è possibile garantire l'assenza. Da ciò si deduce quanto sia difficile, quantomeno in casi non banali, costruire sistemi esenti da fault; ecco perchè la fault-tolerance è un aspetto importante, che merita la giusta attenzione.

2.3 Una classificazione dei fault: dai sistemi distribuiti ai MAS

Riportiamo qui brevemente i concetti di [6], in cui si propone una classificazione dei fault nei MAS, evidenziandone le differenze rispetto ai sistemi distribuiti tradizionali.

2.3.1 Classificazione convenzionale

In [22] i fault per i sistemi distribuiti sono classificati in base a sette attributi:

- Fase di creazione o occorrenza;
- Boundary del sistema;
- Dimensione;
- Causa fenomenologica;
- Obiettivo;
- Capacità;
- Persistenza.

Ognuno di questi ha un insieme di valori esclusivi, come mostra la tabella:

Attributi	**Valori**
Fase di creazione o occorrenza	Operazionale
	Sviluppo
Boundary del sistema	Interno
	Esterno
Dimensione	Hardware
	Software
Causa fenomenologica	Naturale
	Artificiale
Obiettivo	Malizioso
	Non malizioso
Capacità	Accidentale
	Deliberata
	Incompetenza
Persistenza	Permanente Transitorio

L'insieme delle possibili combinazioni dei sette attributi dà 192 elementi; tra questi ne vengono ritenuti significativi 25, chiamati DSFaults (Distributed Systems Faults). Essi possono essere raggruppati in tre macro-gruppi non esclusivi:

- *Fault di sviluppo*: generati in fase di sviluppo;
- *Fault fisici*: dovuti all'hardware;
- *Fault di interazione*: fault esterni.

2.3.2 Classificazione nei MAS

I sistemi multi-agente rientrano nella classe dei sistemi distribuiti, perciò ereditano da essi i cosiddetti DSFaults (vedi 2.3.1). Infatti i MAS:

- siccome sono costituiti da programmi software, sono soggetti ai fault di sviluppo;
- siccome sono programmi distribuiti, sono soggetti ai fault fisici;
- essendo composti da programmi interagenti tra loro, sono soggetti ai fault di interazione.

Tuttavia i sistemi multi-agente non sono riconducibili ai sistemi distribuiti tradizionali; l'elemento peculiare che li contraddistingue è l'autonomia degli agenti, la quale si concretizza nell'impossibilità, da parte di un agente, di prevedere il comportamento di un altro agente. Ciò è potenzialmente causa di fault, che non rientrano nelle categorie viste in precedenza:

- questo tipo di fault non può essere considerato di sviluppo, perchè l'autonomia è una caratteristica naturale degli agenti;
- non si tratta nemmeno di fault operazionali, perchè non sono legati all'effettivo compimento di un servizio, piuttosto al comportamento autonomo degli agenti.

In [6] si sceglie di definire un nuovo valore di `Fase di creazione o occorrenza`, ovvero *Autonomia*. Questo è il valore appropriato per tutti i fault che si verificano per via del comportamento autonomo degli agenti, ad esempio:

- non rispondere ad una richiesta o rispondere in modo negativo, anche se ciò non è contemplato dal protocollo di interazione;
- generare un fault con l'obiettivo di ostacolare un altro agente;
- non fare quanto era stato promesso.

L'introduzione del valore Autonomia produce 96 nuove combinazioni ammissibili, delle quali solo 12 sono ritenute rilevanti. I 12 nuovi tipi vengono chiamati AAFaults (Autonomous Agents Faults).

Questa proposta di classificazione mostra, in modo oggettivo e quantitativo, un concetto facilmente intuibile almeno a grandi linee: le procedure per l'exception-handling applicate con successo nei sistemi distribuiti tradizionali, non sono sufficienti per ottenere una adeguata fault-tolerance nei sistemi multi-agente.

Capitolo 3

I due approcci principali

Una volta stabilito l'obiettivo (costruire MAS che siano fault-tolerant), appare evidente l'importanza di non farlo in maniera episodica: non deve essere compito del progettista, ogni volta, reinventare la ruota nella definizione delle eccezioni e delle relative procedure di gestione. Ciò è infatti al tempo stesso inefficiente (dal punto di vista del tempo di sviluppo) e pericolosamente incline ad errori di progettazione. Di conseguenza, occorre innanzitutto un modello di sistema, e di agente, fault-tolerant; a valle di questo importante passo, è auspicabile la realizzazione di un framework che faccia da ausilio al progettista nell'attuazione di tale modello.

A tal proposito, in letteratura ci sono due scuole di pensiero antitetiche: la visione *agent-centric* e quella *system-centric*. La prima afferma la necessità di affidare agli agenti stessi la responsabilità di rilevazione e gestione dei fault, mentre la seconda ritiene opportuno che l'attività di exception-handling sia effettuata a livello di sistema, ma non dai *problem-solving agents*. Le ragioni di entrambe le tesi sono presentate nel corso di questo capitolo.

3.1 Agent-centric

In questa sezione sono riprese le idee esposte in [1].

3.1.1 Una nuova definizione di eccezione

Definizione Riprendendo quanto visto in 2.1, il contesto dei sistemi multi-agente richiede di riformulare il concetto di eccezione. La definizione proposta è:

> Un eccezione è la valutazione, da parte di un agente, di un evento percepito come inatteso [1].

È nella sorgente delle eccezioni che risiede la differenza essenziale rispetto alle definizioni tradizionali. La sorgente non è l'evento, bensì l'agente stesso. Il carattere inatteso di un evento richiede la presenza di riferimenti interni all'agente, ad esempio la conoscenza di un protocollo, che serve a rappresentare le aspettative dell'agente. Gli agenti puramente reattivi non hanno eccezioni, per via dell'assenza di riferimenti interni [18].

Proprietà Da questa definizione si deducono alcune proprietà:

1. *Le eccezioni hanno significato all'interno di un agente.* Infatti il significato di un evento è stabilito all'atto della sua generazione, ma il suo carattere ordinario o eccezionale dipende dal punto di vista di ciascun agente ricevente: uno stesso messaggio, inviato a due agenti, potrebbe essere interpretato come eccezione da un agente ma non dall'altro. Non c'è nulla di sbagliato in tutto ciò, visto che questo è diretta emanazione del lasco accoppiamento tra agenti.

2. *Le eccezioni non rappresentano solamente errori* [17]. L'uso delle eccezioni nei linguaggi di programmazione è solitamente associato a situazioni problematiche, ma questa visione non è connaturata alla loro definizione.

3. *Le eccezioni negli agenti richiedono una conoscenza interna.* Il carattere di normalità o eccezionalità non fa parte di un evento in quanto tale, ma è stabilito da chi lo percepisce. Perciò, per differenziare i casi comuni da quelli inattesi è indispensabile una base di conoscenza interna a cui fare riferimento.

4. *I meccanismi di gestione delle eccezioni hanno natura asincrona.* Vista l'autonomia e il disaccoppiamento degli agenti, è impossibile garantire che situazioni di eccezione siano gestite in modo sincrono; infatti i tradizionali metodi di propagazione delle eccezioni non hanno un equivalente diretto nei MAS.

5. *Le eccezioni per gli agenti possono esistere anche senza che vi siano eccezioni a livello di codice degli agenti.* Infatti situazioni eccezionali si possono avere anche se tutti gli agenti stanno operando normalmente, e addirittura se il linguaggio di programmazione adottato non supporta i costrutti per le eccezioni.

3.1.2 Architettura di agente

Architettura tradizionale L'attività degli agenti è solitamente articolata nel ciclo *percezione - deliberazione - azione*; la fase di deliberazione può essere elementare per i *reflex agents*, o molto articolata per i *learning agents*. La figura 3.1 mostra l'architettura di questo tipo di agenti.

L'ambiente costituisce il contesto in cui l'agente vive, e con cui interagisce, al fine di acquisire informazioni (tramite i sensori) o di modificarne lo stato (eseguendo azioni tramite gli attuatori). I sensori inoltrano i dati percepiti ai meccanismi interni dell'agente. Questi rappresentano il cuore dell'agente, e a seconda dei casi possono essere realizzati da pianificatori, motori inferenziali o altro; per funzionare, hanno comunque bisogno di interagire con una rappresentazione interna. Questa, ad esempio nell'architettura BDI, è costituita da una serie di basi di conoscenza.

Architettura con exception-handling Le considerazioni fatte in precedenza spingono verso la definizione di una nuova architettura di agente, che comprenda al suo interno capacità di gestione delle eccezioni. In figura 3.2 sono mostrati gli elementi necessari per un'architettura di agente che soddisfi questo requisito. Gli elementi in bianco rappresentano i componenti indispensabili,

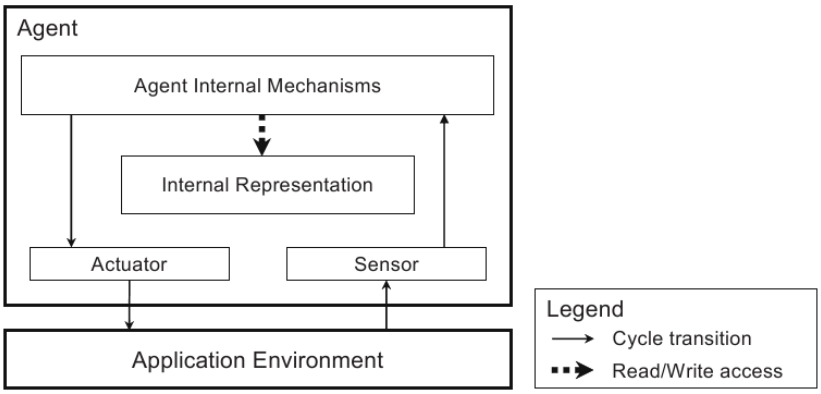

Figura 3.1: Architettura tradizionale di agente (da [6])

mentre quelli in grigio sono opzionali, e conferiscono all'agente la capacità di exception-handling.

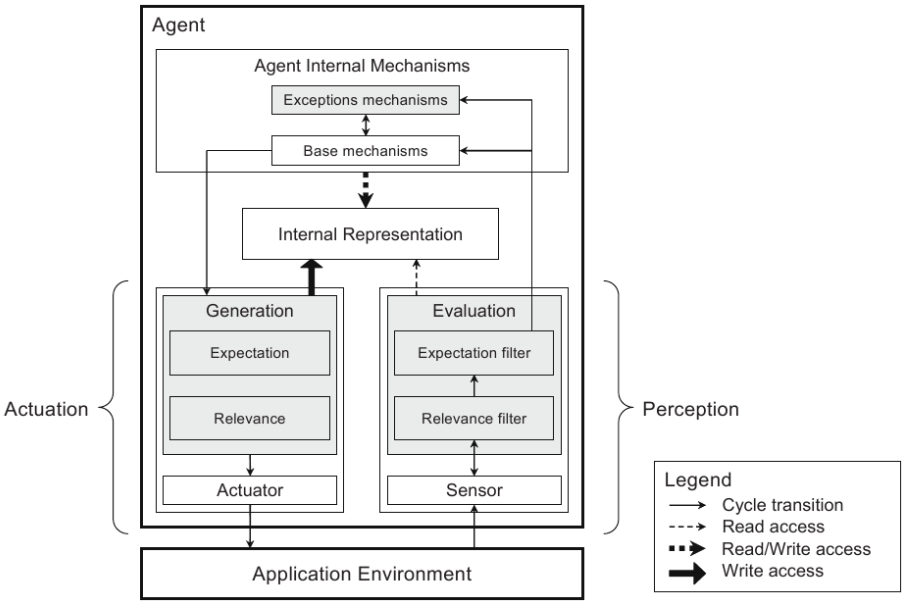

Figura 3.2: Architettura di agente con exception-handling (da [6])

L'elemento di spicco rispetto all'architettura tradizionale risiede nei livelli di percezione ed attuazione: questi non operano più semplicemente interagendo con l'ambiente, bensì hanno al loro interno un'elaborazione riguardante le aspettative e la rilevanza degli eventi. Su questi due elementi si costruiscono criteri volti a classificare gli eventi percepiti. La rilevanza serve a predisporre un filtro che separi gli eventi di interesse da quelli da ignorare (ad es. perchè riguardano potenziali eccezioni a cui l'agente non è interessato). Le aspettative riguardano invece gli eventi che l'agente considera normali, tra quelli che potrebbero verificarsi nel prossimo futuro; esse derivano in maniera piuttosto

diretta dai piani elaborati dai meccanismi interni dell'agente.

Dopo che il componente di percezione li ha valutati, gli eventi sono inoltrati ai meccanismi interni. Questo componente è modulare, e divide la vera e propria business-logic dell'agente dai meccanismi di gestione delle eccezioni. Complessivamente, i meccanismi interni dell'agente producono come output un'azione, che viene passata al componente di attuazione. Questo aggiorna i criteri di rilevanza e aspettativa per le future interazioni, ed esegue effettivamente l'azione indicata.

3.2 System-centric

Secondo [2], pensare di raggiungere una totale fault-tolerance in sistemi di dimensione e complessità considerevoli, basati su mezzi di comunicazione inaffidabili, non è realistico. La cosa diviene ancora più difficile considerando i sistemi multi-agente, visto che l'autonomia è un ulteriore fonte di problemi e fault (vedi 2.3.2). Infatti questo paradigma è essenzialmente non-deterministico, essendo assenti meccanismi di sincronizzazione o di controllo centralizzato. La dicotomia tra fault-tolerance e non-determinismo è risolta, negli approcci system-centric, diminuendo la libertà degli agenti al fine di salvaguardare l'integrità di sistema.

3.2.1 Sentinelle

Il contenuto di questa sezione è tratto in larga parte da [2]. Viene introdotto il concetto di sentinella: una sentinella è un agente, il cui scopo è proteggere certe funzioni, e/o evitare che la società di agenti si porti in certi stati indesiderati. La sentinella non prende parte all'effettivo problem-solving, ma può intervenire, se necessario, sul funzionamento del sistema, ad esempio:

- proporre piani alternativi agli agenti;
- escludere o isolare agenti non operanti in modo corretto
- modificare parametri degli agenti
- etc.

Questo modello è stato costruito facendo riferimento ad una particolare architettura di agente, chiamata DA-SoC [20]. In particolare da essa si deriva lo schema di interazione previsto.

Architettura DA-SoC Non è di nostro interesse introdurre in modo dettagliato l'architettura DA-SoC; ne vediamo solamente la parte necessaria per comprendere il modo in cui operano le sentinelle: lo schema di interazione.

Il meccanismo elementare di comunicazione in DA-SoC è l'invio di messaggi broadcast. Ciascun agente è dotato di una serie di piani di interazione, i quali rappresentano istanze ammissibili di conversazione con altri agenti; la parte dichiarativa di un piano di interazione contiene un nome di agente (indirizzo del ricevente) e un pattern di tipo (quali tipi di messaggi sono accettati). Quando un agente invia un messaggio, tutti gli altri lo ricevono, e tentano il matching con uno dei propri piani di interazione; se il matching ha esito positivo, il piano di interazione è messo in esecuzione. Ciò potrebbe consistere nell'esecuzione

di un piano dell'agente, nel cambiamento del suo modello interno del mondo, nell'invio di messaggi. Questo schema di interazione è chiamato *indirizzamento semantico* [21]. Esso consente agli agenti di interagire con altri senza bisogno di sapere dove questi si trovino effettivamente. Inoltre, un agente riceve effettivamente, tra tutti i messaggi in circolazione, solo quelli che semanticamente si conformano al suo goal e al suo attuale modello del mondo. Chiaramente entrambi possono variare nel corso dell'esecuzione di un agente.

Tornando alle sentinelle, abbiamo che ciascuna di esse, essendo un agente, interagisce con gli altri agenti tramite l'indirizzamento semantico; grazie ad esso, può costruirsi modelli dello stato degli agenti, in due modi:

- monitorando la comunicazione tra agenti (modalità passiva);
- chiedendo direttamente agli agenti (modalità attiva).

Checkpoint Data una certa funzione di sistema per la quale coopera un insieme di agenti, una sentinella può essere preposta a proteggere l'operatività di quella funzione. Allora la sentinella mantiene dei modelli degli agenti, di cui alcuni elementi sono ricavati osservandone le interazioni, altri (espressi come *beliefs*) sono direttamente copiati dal modello del mondo degli agenti. Questi ultimi sono chiamati checkpoint; essi consentono di giudicare lo stato di un agente, non solo in base al suo comportamento osservabile, ma anche secondo il suo stato interno. Ciò consente di rilevare precocemente fault interni ad un agente, o inconsistenze tra agenti (vedi figura 3.3).

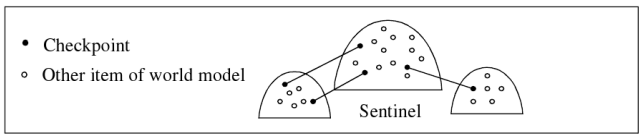

Figura 3.3: Checkpoint e modelli del mondo (da [2])

3.2.2 Servizi di exception-handling

In generale, ciascun sistema software si colloca in un dominio applicativo, in cui viene eseguita una certa business-logic. Tale logica di problem-solving è legata al dominio applicativo, ma è solitamente indipendente dallo specifico contesto di esecuzione. Quest'ultimo determina invece quali siano le possibili situazioni critiche, e le opportune procedure di risoluzione. Con le sentinelle, presentate in 3.2.1, si persegue la netta separazione tra logica di business e logica per la gestione delle eccezioni. Esse sono perciò un tassello fondamentale della visione system-centric, ma la loro definizione deve essere ulteriormente sviluppata per poter costituire una solida organizzazione per la fault-tolerance. Infatti ciascuna sentinella è addetta ad una funzione molto limitata mentre, come già discusso in precedenza, le procedure di fault-handling sono tanto più efficaci quanto più ampia è la visione sul sistema. In questa sezione si introduce il lavoro di [13], in cui si propone un modello di servizio di exception-handling.

Si propone infatti di creare un servizio di exception-handling che possa essere inserito nei sistemi multi-agente, senza bisogno di rivoluzionare la loro precedente architettura o struttura; l'unico requisito aggiuntivo consiste in certi tipi di interazione speciali che dovrebbero essere supportati dagli agenti. Questo servizio può essere visto come un medico. Esso è informato preventivamente sui tipi di problemi che si possono verificare nel sistema in questione, e come si manifestano; il servizio osserva continuamente la situazione, e quando rileva dei sintomi, allora diagnostica il possibile malfunzionamento, e valuta le opportune contromisure da prendere; tali azioni rimediative possono poi essere effettivamente compiute dal servizio, o delegate ai problem-solving agents. I principi ispiratori di questo modello sono quattro:

1. Si ottiene una netta divisione dei compiti: i problem-solving agents effettuano la loro ordinaria business-logic, mentre agenti speciali si incaricano di rilevare e risolvere le situazioni eccezionali. In questo modo i problem-solving agents sono totalmente riusabili, anche in contesti differenti e caratterizzati da diversi tipi di eccezione;

2. il servizio di exception-handling è fondato su una base di conoscenza che include meccanismi generali per la rilevazione, la diagnosi e la risoluzione di situazioni problematiche; questi si prestano ad essere riutilizzati in un'ampia gamma di domini;

3. come costo per rendere un MAS idoneo ad ospitare questo servizio, si richiede un'estensione delle capacità degli agenti: essi devono comprendere un linguaggio standard, ed essere dotati di un certo livello di auto-consapevolezza e auto-modificabilità;

4. se gli agenti supportano tale linguaggio, allora il servizio di exception-handling può essere inserito nel sistema tramite un semplice insieme di agenti.

Il concetto (2) deriva da un'ampia letteratura, specialmente sui sistemi esperti [23, 24]. Infatti vi sono molti principi generali di gestione delle eccezioni, ampiamente riusabili in più domini applicativi, per esempio:

- se il piano di un agente ha fallito, fai *backtracking* ad un piano differente, avente lo stesso goal;

- se una nuova risorsa ad alte prestazioni, e applicabile ad un task *time-critical*, diventa disponibile, effettua una riallocazione del task a questa nuova risorsa;

- se un processo articolato in serie opera troppo lentamente per riuscire a rispettare una certa scadenza, e i singoli subtask dipendono unicamente da quelli precedenti lungo la catena, allora riorganizza il lavoro in pipeline per aumentare la concorrenza;

- etc.

Il servizio interagisce con gli agenti con due scopi, per ciascuno dei quali è utilizzato un linguaggio separato:

- acquisire informazioni e rilevare le eccezioni: si utilizza il *query-language*;

- descrivere le azioni da compiere per risolvere le eccezioni: si impiega l'*action-language*.

Rilevazione delle eccezioni Va da sè che non è possibile rilevare le eccezioni senza disporre di un modello del comportamento normale di un sistema. Per questo ogni agente del sistema deve registrarsi presso il servizio EH (Exception-Handling), fornendo un modello, non necessariamente dettagliato, del suo comportamento ordinario. Il servizio sfrutta la propria base di conoscenza per estrarre i tipi di fallimento noti per quel modello di comportamento, e genera le opportune sentinelle.

Diagnosi e risoluzione Il servizio EH contiene una tassonomia, rigorosa oppure empirica, dei tipi di problemi e dei sintomi ad essi associati. Tale classificazione viene sfruttata per ipotizzare la causa da cui derivano certi sintomi rilevati dalle sentinelle. Una volta stabilita la (supposta) causa, si sfrutta un'ulteriore base di conoscenza (costituita da strategie di risoluzione dei problemi) per decidere quali contromisure prendere.

Query e action language Come spiegato in 3.2.1, le sentinelle non si limitano a monitorare il comportamento osservabile degli agenti, ma richiedono anche informazioni dirette sul loro stato interno. Ciò può avvenire solo se sentinelle ed agenti concordano su un linguaggio condiviso, che chiamiamo query-language, tramite cui specificare quali informazioni sono richieste dalle sentinelle. Inoltre, tra le azioni che il servizio EH stabilisce per reagire ad un'eccezione, alcune possono essere rivolte direttamente agli agenti: ad esempio, modificare il proprio piano corrente, oppure il proprio modello del mondo. Anche in questo caso occorre un linguaggio standard, chiamato action-language.

Il supporto ai due linguaggi è tutto ciò che è richiesto agli agenti in un MAS dotato di servizio EH. In [13] si nota che l'action-language è piuttosto semplice, mentre un query-language completo è indicativamente abbastanza vasto. Ebbene, la piena comprensione dei due linguaggi non è obbligatoria da parte degli agenti; è solamente una proprietà auspicabile in quanto consente un exception-handling più efficace. Anche [5] nota che ciascun agente può avere un proprio livello di comprensione dei linguaggi; se non li supporta affatto, allora quell'agente non contribuirà alla gestione delle eccezioni, mentre se li comprende pienamente, allora darà il massimo contributo nella risoluzione delle situazioni problematiche.

3.3 Considerazioni

Visti nelle sezioni precedenti gli approcci agent-centric e system-centric per la fault-tolerance nei sistemi multi-agente, è possibile fare un confronto e una valutazione critica. Tra gli elementi che fanno propendere a favore del primo, vi sono:

- una sovrastruttura basata su sentinelle costituisce un sistema di controllo, che però non garantisce totalmente l'assenza di situazioni non opportunamente gestite. In particolare, le sentinelle sono anch'esse agenti, e quindi

esposte ai sopra elencati rischi di crash, errori di comunicazione ecc. Le sentinelle rappresentano dunque un *single point of failure*;

- come visto in 3.2, le sentinelle devono accedere ai piani ed al modello del mondo mantenuti internamente dagli agenti: in altre parole, il concetto di agente perde, almeno in parte, le caratteristiche di completa autonomia ed opacità;

- in sistemi di complessità rilevante, predisporre sentinelle per qualsiasi tipo di fault che si possa verificare, è o impossibile, o proibitivo a causa dell'overhead introdotto (in termini sia di computazione, sia di numero di messaggi scambiati).

D'altra parte, diverse ragioni fanno propendere per l'approccio system-centric:

- avere un servizio di exception-handling separato dalla business-logic conferisce modularità all'organizzazione del sistema: i problem-solving agents sono più semplici da realizzare, e riutilizzabili in contesti differenti (in cui cambiano i tipi di eccezioni possibili); inoltre è probabile che anche la qualità della gestione delle eccezioni aumenti;

- un livello distinto di gestione dei problemi può essere sviluppato avvalendosi dell'ampia letteratura sull'exception-handling in generale (non riferito necessariamente ai sistemi multi-agente);

- diventa più fattibile la rilevazione di fault a livello di sistema (es. inconsistenze), che difficilmente potrebbero essere notate dai singoli agenti;

- l'approccio agent-centric mostra i suoi limiti specialmente nel caso di sistemi aperti: non è chiaro come agenti provenienti dall'esterno possano contenere una logica di exception-handling adeguata al contesto in cui giungono.

Tirando le somme, appare difficile eleggere uno dei due approcci come preferibile. Se si considera esclusivamente l'aspetto della fault-tolerance, allora quello system-centric spicca per la modularità e l'organizzazione nella divisione dei compiti. Però non va dimenticato che stiamo analizzando la fault-tolerance *nei sistemi multi-agente*; certi requisiti di quella visione, come la parziale perdita di autonomia ed opacità, stridono non poco con il paradigma ad agenti.

Capitolo 4

Fault-tolerance tramite replicazione

Nei capitoli precedenti abbiamo visto che la fault-tolerance nei sistemi (semplicemente distribuiti, o multi-agente) è una proprietà ad ampio raggio, che potenzialmente abbraccia tutti i tipi di fault possibili. Consultando la letteratura, si nota che il problema è stato spesso affrontato non nella sua generalità, ma considerando solamente un sottoinsieme della gamma di malfunzionamenti: in particolare, ci si è soffermati molto sul fallimento fisico degli agenti (crash dovuti a fallimenti hardware o software; errori di comunicazione). Si sono quindi studiati i meccanismi più opportuni per una fault-tolerance specifica nei confronti del fallimento fisico, tralasciando gli altri tipi di fault; questa problematica è stata trattata in letteratura già a proposito dei sistemi distribuiti (non necessariamente multi-agente), e la strategia comunemente ritenuta più idonea è la replicazione.

4.1 Sfide introdotte dalla replicazione

Questa sezione si rifà ai concetti presentati in [15]. La replicazione consiste nel creare una o più copie di uno o più agenti nell'ambito di un sistema multi-agente. Le repliche di un agente hanno le stesse capacità di quello originario, e nel loro complesso formano un gruppo di replicazione. Se un gruppo di replicazione è visibile all'interno di un MAS, l'interazione con questo, da parte di un altro agente, può avvenire in uno dei seguenti modi:

- l'agente invia le richieste ad una replica alla volta, e prosegue finchè non ottiene una risposta appropriata;
- l'agente invia richieste a tutte le repliche, e delle risposte ricevute:
  - ne sceglie una;
  - ne effettua una sintesi;
- l'agente sceglie una delle repliche, in base ad un certo criterio (velocità, affidabilità, etc.) ed interagisce solo con essa.

Inoltre, si possono distinguere due tipi di replicazione: eterogenea ed omogenea. Nella replicazione eterogenea, tutte le repliche hanno le stesse funzionalità, però potrebbero essere state implementate diversamente; al contrario, la replicazione omogenea prevede che tutte le copie siano istanza dello stesso codice applicativo.

Definito il concetto di replicazione, possiamo vedere quali aspetti problematici vanno affrontati riguardo a questa strategia.

4.1.1 Scelta del tipo di replicazione

Le due alternative consistono nella replicazione omogenea ed eterogenea. La replicazione omogenea mantiene bassi i costi di sviluppo (la business-logic va implementata una volta sola), però non è particolarmente robusta. Supponiamo di avere agenti deterministici, replicati in modo omogeneo: se ad un certo punto l'esecuzione di un agente incontra un bug software, o giunge in uno stato imprevisto, allora esso fallirà, insieme a tutte le sue repliche. Se invece la replicazione è eterogenea, allora la probabilità di avere il medesimo bug nelle diverse repliche decresce fortemente: ciò equivale ad un miglioramento della fault-tolerance.

4.1.2 Comunicazione tra agenti e sintesi dei risultati

La replicazione dovrebbe essere effettuata, se possibile, in modo trasparente, ovvero non dovrebbe essere visibile agli altri agenti il fatto che di un agente esistono N repliche. Questo aspetto acquisisce ulteriore importanza in sistemi aperti, in cui non c'è motivo affinchè gli agenti debbano essere consapevoli delle politiche di replicazione altrui. A questo punto, è lecito domandarsi come faccia un agente ad interagire con un gruppo di replicazione: nel caso in cui scelga una singola replica, su quali criteri si basi la scelta; se invece interagisce con tutte le repliche, come vengano gestite le risposte. Si presenta inoltre anche il problema duale della conversazione avviata da un gruppo di replicazione, e diretta ad un altro agente: o l'interazione è effettuata da una singola replica, scelta secondo certi criteri, oppure tutte le repliche avviano la conversazione (per cui l'agente destinatario dovrebbe essere pronto a ricevere messaggi multipli). Inoltre, l'operazione di sintesi (elaborazione di un singolo risultato a partire da N risposte) è facilmente definibile ad esempio in ambito matematico, ma non ha un significato netto in ambito generale.

4.1.3 Consistenza di lettura / scrittura

Gli agenti sono per definizione immersi in un ambiente, con il quale interagiscono continuamente. Tale interazione si manifesta in due tipi di attività, la percezione (che possiamo assimilare ad un'operazione di lettura) e l'azione (analoga alla scrittura). Se un dato ambientale ha un valore fisso, allora qualunque replica ne otterrà sempre lo stesso valore; se invece si tratta di grandezze variabili, allora varie repliche, impegnate nello stesso task, potrebbero effettuarne la lettura in momenti successivi, ed ottenere valori differenti (rispettando comunque la proprietà di consistenza di lettura [2]). Di conseguenza, ciascuna replica completerà il task producendo risultati diversi.

La consistenza di scrittura è fortemente legata a quella di lettura. Siccome spesso l'interazione con l'ambiente segue pattern di tipo lettura-scrittura, l'ese-

cuzione parallela di più operazioni di questo tipo può portare a inconsistenze e conflitti, come è noto dalla teoria dei sistemi informativi.

4.1.4 Sincronizzazione dello stato

Se tutte le repliche di un agente hanno uguale stato interno, allora non si pongono problemi di consistenza di lettura / scrittura: infatti per ogni task le repliche produrranno sempre i medesimi risultati. Per garantire la consistenza di stato tra le repliche, occorrono opportuni meccanismi di sincronizzazione, i quali possono operare solo se gli agenti sono in grado di leggere il proprio stato interno, e di modificarlo al fine di allinearsi con le altre repliche. Particolare attenzione va posta nei confronti di agenti in esecuzione: infatti l'aggiornamento di stato non deve interferire le azioni attualmente in corso di esecuzione, per non causarne l'annullamento o la sospensione.

Un altro ostacolo da superare riguarda la rappresentazione dello stato. Repliche omogenee non pongono problemi, grazie all'uguaglianza strutturale tra le copie; nel caso eterogeneo, invece, lo stato deve essere espresso in un formato comprensibile da ciascuna replica.

4.2 Gestione dinamica delle repliche

La ridondanza nei sistemi distribuiti è una tecnica comunemente adottata per dare maggiore robustezza a fronte di guasti e malfunzionamenti. La replicazione delle entità ha ovviamente un costo, quantificabile sia in termini di carico computazionale, sia di numero di messaggi scambiati. Occorre perciò valutare attentamente le esigenze di affidabilità anche a livello di singole di entità, attraverso un *fine-tuning* che ottimizzi l'allocazione delle risorse disponibili. Nei MAS ciò consiste nello stabilire, per ogni agente, il tipo e il numero di repliche assegnate. Tuttavia nell'ambito dei MAS effettuare un'impostazione statica dei parametri è un compito assai arduo per il progettista: infatti i sistemi multiagente si caratterizzano per la loro flessibilità e dinamismo, e in virtù di ciò emerge l'utilità di architetture in cui le politiche di replicazione possano essere aggiornate anche a tempo di esecuzione.

Il primo passo per supportare variazioni dinamiche delle politiche, consiste nell'incapsulare in qualche modo i gruppi di replicazione. A tal scopo si introduce il concetto di *proxy*.

4.2.1 Proxy

Un proxy fa da mediatore tra le repliche di un agente e il resto del MAS. La sua utilità sta nel:

1. far sì che il gruppo di replicazione sia visto dagli altri agenti come una singola entità;

2. controllare l'esecuzione e lo stato del gruppo di replicazione.

Utilizzando i proxy, la replicazione diviene trasparente, ovvero gli agenti del sistema appaiono come entità singole, indipendentemente dal numero di repliche di cui dispongono.

4.2.2 Gestione dei gruppi di replicazione

Il punto (2) è fondamentale per la gestione delle risorse computazionali. Possiamo individuare tre politiche di gestione delle repliche [15]:

1. *cold-standby*: il proxy elegge una delle repliche come attiva, e la lascia in esecuzione; le altre repliche vengono temporaneamente disattivate;

2. *hot-standby*: il proxy elegge una delle repliche come attiva, e la lascia in esecuzione; le altre repliche vengono disattivate, ma sono periodicamente aggiornate con lo stato della replica attiva;

3. *active*: tutte le repliche sono attive allo stesso tempo.

Le alternative sono disposte in ordine crescente riguardo al carico computazionale sul sistema, e in ordine decrescente riguardo al tempo di ripristino a fronte di un guasto. Nel cold-standby, la fase di funzionamento normale è assai efficiente ma, in caso di fallimento della replica attiva, prima di attivarne una di quelle sospese occorre aggiornarne lo stato. La modalità active invece mantiene l'agente sempre disponibile, anche a fronte di guasti di una replica, però obbliga il proxy ad effettuare una serie di compiti aggiuntivi, volti a garantire la consistenza di lettura / scrittura e la sintesi dei risultati.

Questi elementi pongono le basi per una gestione dinamica delle politiche di replicazione a livello di sistema multi-agente. Per modificare le caratteristiche di un gruppo di replicazione è sufficiente interagire con il relativo proxy, il quale si occupa di effettuare tutte le azioni necessarie ad applicare la modifica. Supponiamo di avere un MAS dedicato al supporto ed organizzazione per le unità di crisi: ad esempio, all'occorrenza di un particolare tipo di emergenza, si rivelino fondamentali gli agenti di un certo tipo, e si voglia conferire loro una alta affidabilità; rientrati dalla situazione eccezionale, si potrebbe voler riportare l'allocazione delle risorse ai valori standard. La fattibilità di scenari come questi aumenta in architetture di replicazione basate sui principi presentati in questa sezione.

Un esempio concreto di framework, pensato appositamente per la fault-tolerance tramite replicazione nei MAS, è costituito da DARX (Dynamic Agent Replication eXtension) [4, 10, 11].

4.3 Un modello predittivo

Il contributo delle proposte viste in 4.2 è dato dal fatto che le strategie di replicazione non devono più essere cablate nel sistema a tempo di progettazione, infatti possono essere aggiornate durante l'esecuzione. Tuttavia il cambiamento delle politiche è un atto deliberato, che viene compiuto esplicitamente da qualche attore: potrebbe essere un operatore umano, oppure un agente del sistema. In [16] si fa un passo in avanti: i cambiamenti delle politiche di replicazione non sono atti espliciti di qualche attore, perchè è il framework di replicazione ad aggiornarle autonomamente, in base alle esigenze correnti del sistema.

Uno schema siffatto necessita di un modello con cui rappresentare l'importanza relativa di un agente all'interno di una società; da questa si potrebbero ricavare i requisiti di affidabilità associati, e dunque il corretto grado di replicazione. A tal fine, viene proposto il concetto di criticità di un agente.

4.3.1 Criticità di un agente

Prendiamo in considerazione agenti intelligenti: ad ogni istante, ciascun agente è dotato di un piano, ovvero di una sequenza di azioni, solitamente a breve termine, che l'agente ritiene opportuno eseguire per il raggiungimento dei propri obiettivi. Ad un certo istante di tempo, la criticità di un agente è valutata in base alla criticità delle prossime azioni incluse nel suo piano: un agente che esegue operazioni importanti deve essere ritenuto più critico di altri. Introduciamo allora i concetti di criticità assoluta e relativa.

Criticità assoluta Ciascun tipo di azione ha una propria criticità assoluta, del tutto indipendente dagli attuali piani degli agenti. Il suo valore è fissato dal progettista, e dipende da alcuni fattori:

- quanti agenti sono in grado di eseguire quell'azione: se un'azione può essere eseguita da un gruppo ristretto di agenti, allora in caso di fallimento il riassegnamento del lavoro potrebbe essere difficoltoso; perciò la sua criticità assoluta è elevata;
- risorse necessarie per la sua esecuzione: se un'azione richiede elevate risorse oppure è molto lunga, allora è maggiormente critica;
- informazioni semantiche: il progettista, conoscendo il dominio applicativo, può decidere di conferire maggiore criticità a certe azioni piuttosto che ad altre.

Criticità relativa La criticità relativa di un'azione misura l'impatto che un suo eventuale fallimento avrebbe sul sistema complessivo. Essa dipende ovviamente dalla criticità assoluta dell'azione, ma anche da quanti agenti dipendono dalla sua esecuzione, e attendono i relativi risultati.

4.3.2 Funzionamento

Dunque la criticità di un agente è legata alla criticità relativa delle azioni incluse nel proprio piano. È noto che in ambienti dinamici occorre trovare un compromesso tra pianificazione e reattività, ed è dunque evidente che piani a lungo termine hanno significatività limitata per l'alta probabilità che nel corso del tempo il mondo si sposti dallo stato in cui era al momento dell'elaborazione del piano. Per questo, nel calcolo della criticità vengono considerati solo piani a breve termine.

Dopo aver stimato la criticità degli agenti, per concretizzare il meccanismo occorre risolvere i seguenti punti:

1. quali agenti devono essere replicati, e quante repliche vanno istanziate;
2. dove effettuare il *deployment* delle repliche.

Per quanto riguarda il punto (1), in [25] si propone di ricalcolare, con una cadenza δt, il numero di repliche che ogni agente deve avere: questo numero è direttamente proporzionale alla criticità dell'agente e al numero di repliche disponibili, ed inversamente proporzionale alla somma delle criticità di tutti gli

agenti del sistema. In [16] si estende questo schema per prendere in considerazione l'aspetto (2): infatti le repliche non sono considerate tutte uguali, in quanto si valuta l'affidabilità dei nodi in cui sono collocate. Perciò supponendo di assegnare ad ogni nodo un grado di affidabilità, allora il numero di repliche assegnate ad un agente potrebbe aumentare se queste vengono poste su nodi poco affidabili, o viceversa diminuire se i nodi scelti sono molto affidabili.

Capitolo 5

Conclusioni

In questo lavoro si è tentato di offrire una panoramica di quanto la letteratura ha prodotto recentemente sulla tematica della fault-tolerance nei sistemi multi-agente. Un aspetto che si può notare, osservando la bibliografia, è la data degli articoli: la maggior parte di essi appartiene agli anni a cavallo del 2000, e ben pochi sono quelli recenti, prodotti negli ultimi 2-3 anni. Questa perdita di interesse, quantomeno apparente, a nostro avviso si può spiegare attraverso un'analisi critica della produzione scientifica su questo argomento. La fault-tolerance è un settore di ricerca di gran lunga preesistente rispetto ai sistemi multi-agente, e nei lavori lavori presi in esame si è visto un certo sforzo di inquadrare quei concetti, come ad esempio quello di eccezione, nel nuovo paradigma.

In merito ad architetture intrinsecamente fault-tolerant, si sono formate due scuole di pensiero piuttosto nette e contrapposte: agent-centric e system-centric. Tuttavia pare che entrambe siano rimaste ad uno stadio primordiale, in cui i concetti sono stati definiti ma non sviluppati ulteriormente, rimanendo piuttosto astratti e generici. Infatti quei due approcci, che hanno validità generale e affrontano la fault-tolerance *tout-court*, sono ormai scomparsi dalla letteratura degli ultimi anni. Recentemente ci si è focalizzati molto su un tipo specifico di fault, il crash fisico degli agenti, e si è proposto come rimedio la replicazione degli agenti. Questo campo ha prodotto alcuni risultati, come l'introduzione di modelli per la gestione dinamica delle politiche di replicazione, e anche lo sviluppo di un framework per il supporto alla trasparenza e dinamicità della replicazione (DARX). Gli autori di questi ultimi lavori talvolta criticano le visioni agent-centric e system-centric (specialmente la seconda) per la complessità e l'infattibilità di tali approcci. Tuttavia, a ragion veduta, si osserva che la replicazione non offre alcuna alternativa a quei due modelli: semplicemente ignora una larga parte di fault che si possono avere nei sistemi multi-agente, concentrandosi solo su un particolare tipo di problemi.

In definitiva, il settore della fault-tolerance nei sistemi multi-agente appare attualmente poco vitale, e privo di spunti che possano portare veramente ad innovazioni rilevanti.

Elenco delle figure

Bibliografia

[1] E. Platon, N. Sabouret, and S. Honiden. A definition of exceptions in agent-oriented computing. In Proc. of ESAW, pages 161–174, 2006.

[2] Staffan Haegg, A Sentinel Approach to Fault Handling in Multi-Agent Systems, Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence: Multi-Agent Systems: Methodologies and Applications, p.181-195, August 27, 1996.

[3] S. Kumar, P. R. Cohen, and H. J. Levesque, The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams, in International Conference on Multi-Agent Systems (ICMAS-2000), Boston MA USA, 2000.

[4] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault tolerance for distributed multi-agent systems. In Proceedings of MAAMAW2001, 2001.

[5] Mark Klein , Juan-Antonio Rodriguez-Aguilar , Chrysanthos Dellarocas, Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The Case of Agent Death, Autonomous Agents and Multi-Agent Systems, v.7 n.1-2, p.179-189, July-September 2003.

[6] Katia Potiron, Patrick Taillibert, Amal El Fallah-Seghrouchni, A Step Towards Fault Tolerance for Multi-Agent Systems, LADS, Lecture Notes in Computer Science, Vol. 5118, pp. 156-172, Springer, 2007.

[7] Zahia Guessoum , Nora Faci , Jean-Pierre Briot, Adaptive replication of large-scale multi-agent systems: towards a fault-tolerant multi-agent platform, ACM SIGSOFT Software Engineering Notes, v.30 n.4, July 2005.

[8] Mallya, A.U., Singh, M.P.: Modeling exceptions via commitment protocols. In: Autonomous Agents and Multi-Agent Systems, pp. 122-129. ACM Press, New York (2005).

[9] Nico Roos , Annette ten Teije , Cees Witteveen, A protocol for multi-agent diagnosis with spatially distributed knowledge, Proceedings of the second international joint conference on Autonomous agents and multiagent systems, July 14-18, 2003, Melbourne, Australia.

[10] Z. Guessoum , J. P. Briot , S. Charpentier , O. Marin , P. Sens, A fault-tolerant multi-agent framework, Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, July 15-19, 2002, Bologna, Italy.

[11] Olivier Marin , Marin Bertier , Pierre Sens, DARX - A Framework For The Fault-Tolerant Support Of Agent Software, Proceedings of the 14th International Symposium on Software Reliability Engineering, p.406, November 17-21, 2003.

[12] Z. Guessoum, J.-P. Briot, P. Sens, and O. Marin. Toward fault-tolerant multi-agent systems. In MAAMAW'2001, Annecy, France, May 2001.

[13] Mark Klein , Chrysanthos Dellarocas, Exception handling in agent systems, Proceedings of the third annual conference on Autonomous Agents, p.62-68, April 1999, Seattle, Washington, United States.

[14] Sanjeev Kumar , Philip R. Cohen, Towards a fault-tolerant multi-agent system architecture, Proceedings of the fourth international conference on Autonomous agents, p.459-466, June 03-07, 2000, Barcelona, Spain.

[15] A. Fedoruk and R. Deters, Improving Fault-Tolerance by Replicating Agents, In Proceedings of 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, July 2002.

[16] Alessandro de Luna Almeida , Samir Aknine , Jean-Pierre Briot , Jacques Malenfant, A Predictive Method for Providing Fault Tolerance in Multi-agent Systems, Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology, p.226-232, December 18-22, 2006.

[17] Goodenough, J.B.: Exception Handling: Issues and a Proposed Notation. Commun. ACM 18(12), 683-696 (1975).

[18] Brooks, R.: Intelligence without representation. Artificial Intelligence 47(1-3), 139-159 (1991).

[19] Burns A., and Wellings A., Real-Time Systems and Their Programming Languages, Addison-Wesley, 1990, ISBN 0-201-17529-0.

[20] Haegg S., and Ygge F., Agent-Oriented Programming in Power Distribution Automation - An Architecture, a Language, and their Applicability, Ph.L. Thesis, LUNFD6/(NFCS-3094)/1-180/(1995), Lund University, Sweden, May 1995.

[21] Haegg S., Adaptation in a Multi-Agent System through Semantic Addressing, in Proceedings of the Workshop on Decentralized Intelligent and Multi-Agent Systems (DIMAS'95), Krakow, Poland, November 1995.

[22] Arlat, J., Crouzet, Y., Deswarte, Y., Fabre, J.C., Laprie, J.C., Powell, D.: Tolerance aux fautes. In: Akoka, I.W.J. (ed.) Encyclopedie de l'Informatique et des Systemes d'Information, Vuibert, pp. 241-270 (2006).

[23] Gruber, T.R, A Method For Acquiring Strategic Knowledge. Knowledge Acquisition, 1989. l(3): p. 255-277.

[24] Bamett, J.A., How Much Is Control Knowledge Worth? A Primitive Example. Artificial Intelligence, 1984. 22(1): p. 77-89.

[25] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, P. Sens, ?Dynamic and adaptive replication for large-scale reliable multi-agent systems?, In Software Engineering for Large-Scale Multi-Agent Systems (SELMAS), LNCS 2603, April 2003, pp. 182-198.