Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

PROGETTAZIONE, IMPLEMENTAZIONE E VALUTAZIONE DI SISTEMI PER LA FAULT TOLERANCE IN SIMULAZIONI DISTRIBUITE

Tesi di Laurea in Reti di Calcolatori

Relatore:
Gabriele D'Angelo

Presentata da:
Lorenzo Armaroli

II Sessione
Anno Accademico 2010-2011

Indice

Introduzione

La simulazione è un campo dell'informatica che trova applicazioni di notevole importanza nella ricerca, nella progettazione e nella produzione industriale.
Poter testare il comportamento di modelli e prototipi sperimentali prima di averli realizzati fisicamente porta ad un ingente risparmio di risorse e di tempo, in particolare quando serve modificare i prototipi iniziali per risolvere problemi emersi durante le fasi di test.

Le simulazioni però, a causa dell'elaborazione di sistemi complessi composti da un altissimo numero di elementi che interagiscono fra loro, sono anche tra le applicazioni più onerose in termini di tempo di esecuzione. Esistono infatti simulazioni che possono richiedere ore, giorni o addirittura settimane di tempo di calcolo.

Un approccio per rendere più veloce l'esecuzione delle simulazioni, è quello di implementare simulatori basati su architetture parallele e distribuite, per poter suddividere il carico di lavoro su più unità di elaborazione.
La simulazione parallela e distribuita però, pur riducendo il tempo richiesto per l'esecuzione delle simulazioni, introduce problematiche tipiche di concorrenza e sistemi distribuiti, come la sincronizzazione delle unità di elaborazione, il costo dovuto alla comunicazione tra le unità di elaborazione, e la necessità di rispettare il rapporto causa-effetto in un ambiente in cui l'ordine di esecuzione delle istruzioni non è determinabile a priori.

Tra le tecniche sviluppate per risolvere le problematiche introdotte dalla simulazione parallela e distribuita, odiernamente una delle meno affrontate è la tolleranza ai guasti,

cioè lo sviluppo di tecniche che permettano il corretto svolgimento di una simulazione, anche nell'eventualità in cui alcuni componenti dovessero guastarsi durante l'esecuzione. L'utilità di un simulatore tollerante ai guasti appare evidente immaginando simulazioni che richiedono lunghi tempi di esecuzione, in cui il verificarsi di guasti non pregiudica la correttezza dei dati ottenuti, e quindi non è necessario ricominciare le simulazioni dall'inizio, con un evidente e notevole risparmio di tempo.
Al tempo stesso, l'implementazione di tecniche per la tolleranza ai guasti nei sistemi distribuiti introduce altre problematiche, in quanto si rende necessario un certo livello di ridondanza, che porta ad un maggiore costo computazionale.

Gli obiettivi di questa tesi sono quindi in una prima fase la progettazione e l'implementazione di un sistema software per la simulazione parallela e distribuita che includa tecniche per la tolleranza ai guasti, e in una seconda fase la valutazione dell'effettiva utilità di questo sistema, in cui si osserveranno i benefici apportati dalle tecniche di tolleranza ai guasti implementate, in rapporto al conseguente aumento di costo computazionale.

Capitolo 1

La simulazione

La simulazione è un campo dell'informatica che consiste nell'**imitare per mezzo di software il comportamento di un sistema**. I sistemi che vengono imitati possono essere di varia natura, e i motivi per cui può essere interessante simularli sono molteplici. Tra le varie applicazioni infatti la simulazione può servire allo studio di sistemi fisici reali (simulazione analitica) e alla realizzazione di ambienti virtuali interattivi, i cui scopi vanno dall'intrattenimento (videogiochi) all'addestramento del personale a situazioni pericolose e/o difficilmente riproducibili nella realtà (simulatori di volo, ecc.) [3, 10].

Una delle applicazioni principali è fornire uno strumento per studiare sistemi complessi senza dover analizzare direttamente questi sistemi. Questo strumento può risultare utile sia perché analizzare il sistema simulato è più semplice, veloce ed economico rispetto all'analisi del sistema reale (ad esempio nello studio del traffico stradale), sia perché il sistema reale ancora non esiste, e se ne vuole studiare il comportamento prima dell'effettiva realizzazione (ad esempio la verifica del corretto funzionamento di algoritmi o dispositivi elettronici prima dell'effettiva realizzazione dei prototipi reali). In questo ultimo caso, di notevole rilevanza nel campo della produzione industriale, simulare un sistema ancora non implementato permette un enorme risparmio di risorse e di tempo, soprattutto nel momento in cui emergono eventuali problemi, che possono essere individuati e risolti prima della realizzazione fisica dei prototipi.

I software che si utilizzano per simulare un sistema sono chiamati simulatori, e possono essere costituiti da un unico processo (simulatori monolitici) o da più processi, che si possono trovare in concorrenza sullo stesso elaboratore o distribuiti su più elaboratori connessi in rete (simulatori paralleli e distribuiti). I simulatori devono inoltre rispettare due importanti vincoli:

- ogni simulazione deve essere perfettamente riproducibile, in modo che sia sempre possibile ripetere e analizzare la sequenza causale di eventi che porta a condizioni interessanti del sistema;

- deve essere rispettato il rapporto di causalità degli eventi che avvengono durante lo svolgimento della simulazione (questo vincolo, semplice da rispettare nel caso dei simulatori monolitici, diventa un problema di difficile gestione nel caso dei simulatori paralleli e distribuiti).

1.1 Simulazione monolitica

I simulatori monolitici sono costituiti da un unico processo che elabora il comportamento di tutte le entità coinvolte nel sistema simulato.

Grazie a questa caratteristica, i simulatori monolitici possono rispettare i vincoli di riproducibilità e di causalità in modo molto semplice.

La riproducibilità di una simulazione è garantita dal fatto che il simulatore è costituito da un unico processo, e quindi se a questo vengono dati gli stessi input, la sequenza di esecuzione risulterà essere sempre la stessa. L'unico aspetto su cui è necessario fare attenzione è la generazione di numeri casuali, che deve essere realizzata mediante un generatore di numeri pseudocasuali[1], a cui vengono dati in input semi conosciuti dall'utente e persistenti, cioè indipendenti da condizioni momentanee irripetibili.

Il rapporto di causalità è semplice da rispettare nel caso monolitico, perché tutti gli eventi che avvengono nella simulazione possono essere collocati temporalmente senza

[1]Un generatore di numeri pseudocasuali è un programma che genera, a partire da un seme iniziale dato in input, una sequenza di numeri apparentemente casuale, ma che risulta essere sempre la stessa per quel particolare seme.

nessuna ambiguità, grazie all'utilizzo di un orologio globale, accessibile da tutti gli elementi del sistema.

Nei simulatori monolitici inoltre è disponibile uno stato globale, che rende più semplice l'implementazione di entità che devono avere una visione globale dell'intero sistema (ad esempio un osservatore esterno che ha visibilità dell'intero sistema e che effettua la *proximity detection* [18], incaricato cioè di rilevare la vicinanza tra entità che si muovono all'interno di uno spazio simulato).

Potendo contare sulla possibilità di disporre di uno stato e di un orologio globali, una simulazione realizzata con simulatori monolitici[2] è più semplice da realizzare rispetto a una realizzata con simulatori paralleli e distribuiti. Per contro però, il carico di lavoro dell'intera simulazione grava su un'unica unità di elaborazione, o *PEU* (Physical Execution Unit). Questo fatto, unitamente all'alta capacità di calcolo richiesta da una simulazione e alla frequente necessità di ottenere risultati in tempi brevi, determina un limite superiore alla complessità delle simulazioni che possono essere eseguite tramite simulatori monolitici.

1.2 Simulazione parallela e distribuita

Al contrario della simulazione monolitica, per la simulazione parallela e distribuita (PADS, Parallel And Distributed Simulation [3, 5, 6]) si utilizzano simulatori composti da più processi in concorrenza, che possono essere eseguiti sullo stesso elaboratore (simulazione parallela) o su più elaboratori in rete (simulazione distribuita).

Ogni processo, chiamato LP (Logical Process) viene eseguito da una certa unità di elaborazione fisica (PEU[3], Physical Execution Unit) del sistema. Queste PEU si possono trovare sullo stesso elaboratore (nel caso di un'architettura multi-core o multi-CPU) e comunicare tra loro tramite memoria condivisa, oppure si possono trovare su elaboratori

[2] Un esempio di simulatore monolitico è *Peersim* [19].

[3] Una PEU sostanzialmente corrisponde a una CPU single-core, oppure a un singolo core di una CPU multi-core.

distinti (nel caso di un sistema distribuito su più elaboratori in rete) e comunicare tra loro attraverso la rete.

Risulta evidente che la memoria condivisa del caso parallelo sia un metodo di comunicazione molto più efficiente rispetto allo scambio di messaggi sulla rete. Per contro, la scalabilità di un sistema puramente parallelo è limitata dalle caratteristiche dell'architettura del singolo elaboratore multi-CPU e/o multi-core utilizzato, limite non presente nel caso di un sistema distribuito, che può essere composto da un numero arbitrario di elaboratori.

Le architetture reali per la simulazione parallela e distribuita di fatto sono un'unione di entrambe le architetture descritte: sono composte cioè da un numero arbitrario di elaboratori connessi in rete, dove ogni elaboratore può essere caratterizzato sia da un'architettura single-core, sia da un'architettura multi-core e/o multi-CPU (fig. 1.1).

Pur introducendo alcune problematiche tipiche di sistemi concorrenti e distribuiti, scegliere la simulazione parallela e distribuita piuttosto che la simulazione monolitica è spesso conveniente, o in alcuni casi addirittura obbligatorio, per i seguenti motivi.

- Rispetto all'esecuzione di tutta la simulazione su una singola unità di elaborazione, distribuire il carico di lavoro su più unità permette di ridurre il tempo totale richiesto dall'elaborazione della simulazione. Questo è molto importante quando un'applicazione della simulazione necessita di ottenere risultati entro tempi prestabiliti (sistemi real-time) o nel più breve tempo possibile (simulazione analitica). Contrariamente a quanto si può pensare però, distribuire una simulazione su N PEU non significa dividere per N il tempo totale richiesto dalla simulazione (caso ottimo), perché ogni simulatore distribuito introduce un overhead dovuto alla comunicazione e alla sincronizzazione tra i processi che lo compongono. Oltre a questo poi è sempre presente anche un overhead tipico dei programmi concorrenti, dovuto al fatto che, anche volendo, non si riesce mai a raggiungere un parallelismo perfetto, in cui il tempo di esecuzione parallela è esattamente il tempo di esecuzione sequenziale diviso per il numero di CPU che eseguono il programma. È infatti sempre presente in un programma concorrente una certa porzione di istruzioni che vengono comunque eseguite sequenzialmente (Legge di Amdahl [26]).

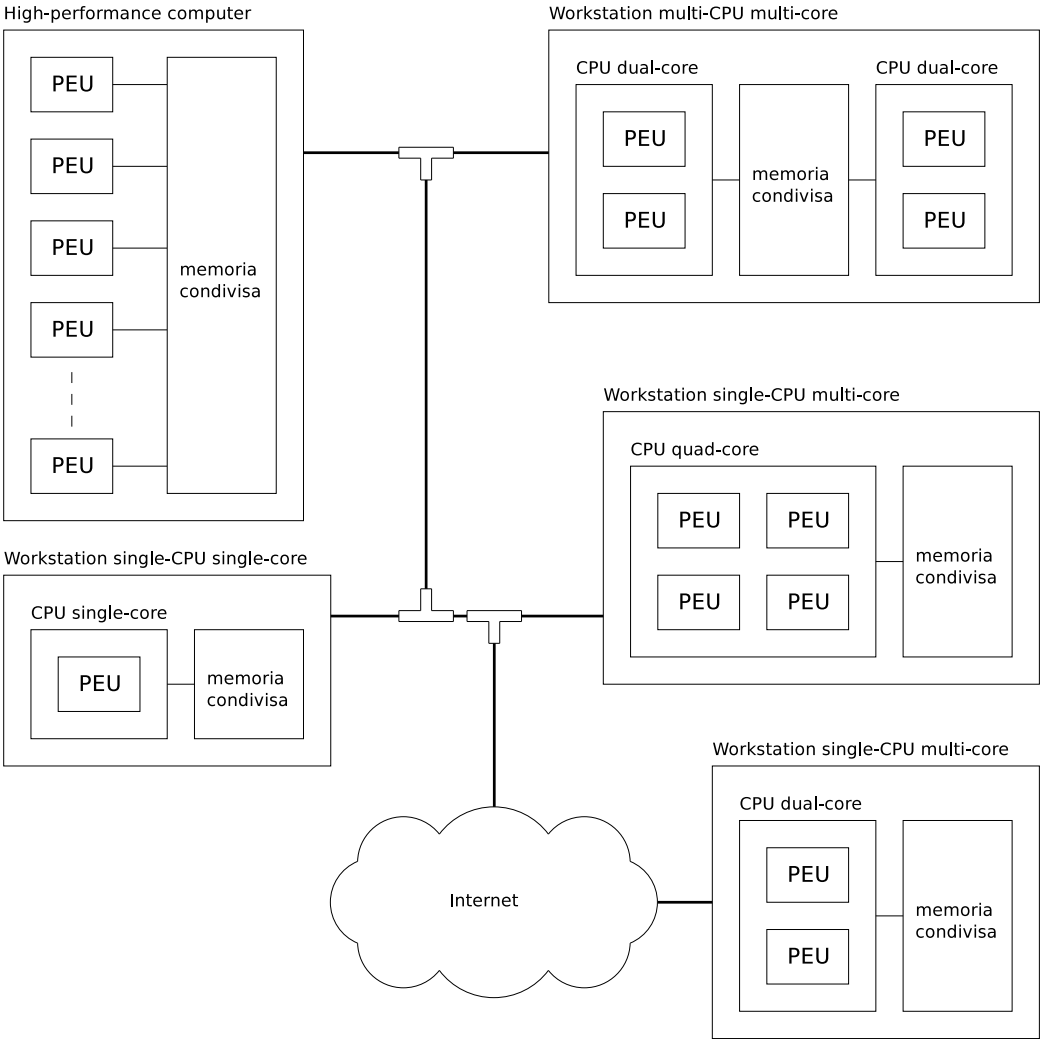

Figura 1.1: Un'architettura reale per la simulazione parallela e distribuita: si può vedere che il sistema distribuito può essere composto da elaboratori eterogenei, ognuno di questi dotato di un certo numero di CPU e di core, a cui corrispondono le PEU del sistema che eseguiranno gli LP della simulazione distribuita. Si può vedere inoltre che i canali di comunicazione tra le PEU sono anch'essi eterogenei: nella figura le varie PEU sono connesse tra loro tramite memoria condivisa, rete locale e Internet.

- Poter aumentare arbitrariamente il numero di unità di elaborazione permette di rendere il simulatore scalabile rispetto alle dimensioni del sistema simulato, rendendo possibili simulazioni di sistemi complessi composti da un altissimo numero di entità, che non sarebbero invece realizzabili con un simulatore monolitico a causa di limiti temporali.

- C'è un caso in cui la simulazione distribuita non è solo conveniente, ma addirittura una scelta obbligata, ed è il caso in cui gli elementi che compongono la simulazione si trovano in luoghi geograficamente distanti. In questo caso ovviamente non è possibile utilizzare una simulazione monolitica, perché è necessario far comunicare gli elementi distanti tra loro per mezzo della rete. Un esempio di simulazione distribuita appartenente a questo caso è quello dei videogiochi multiplayer, in cui più utenti possono interagire all'interno di un ambiente virtuale comune, ognuno giocando da casa propria, con il proprio personal computer connesso agli altri attraverso la rete [20]. Un altro esempio è quello di simulazioni in cui i componenti, appartenenti a diverse società o istituzioni, risiedono ognuno negli elaboratori della propria società, che non desidera che questi vengano ceduti a terzi per motivi di tutela della proprietà intellettuale. In questo caso, se si vuole utilizzare nella propria simulazione un determinato componente di una società terza, il proprio simulatore può comunicare con quello di tale società e ottenere i risultati senza che l'utente del proprio simulatore possa risalire all'implementazione del componente altrui.

Le problematiche introdotte, tipiche dei sistemi distribuiti, sono le seguenti.

- Nel caso distribuito, non poter utilizzare la memoria condivisa rende impossibile disporre di uno stato globale a cui le entità della simulazione possono fare riferimento. Questo significa che informazioni che dovrebbero poter essere esaminabili globalmente (ad esempio la posizione di tutte le entità in uno spazio simulato, allo scopo di implementare algoritmi di proximity detection) devono necessariamente essere propagate, aggiornate e mantenute sincronizzate tramite un alto numero di messaggi, che determina un notevole overhead di comunicazione sulla rete.

- Per evitare che si formino colli di bottiglia, il carico di lavoro dovrebbe essere mantenuto bilanciato sulle varie PEU che compongono il sistema distribuito. Può capitare infatti che alcuni elaboratori siano più lenti di altri, oppure ci siano entità che richiedono un maggior numero di calcoli di altre. In questi casi per rendere il sistema distribuito più efficiente sarebbe meglio caricare maggiormente di lavoro[4] gli elaboratori più potenti e scaricare quelli meno potenti.

- In un sistema parallelo e distribuito, l'impredicibilità dell'avvicendamento dei processi e l'assenza di uno stato globale rendono impossibile disporre di un orologio globale. Si rende quindi necessario l'utilizzo di complessi algoritmi di gestione del tempo, per poter eseguire la simulazione con la certezza di rispettare sempre il rapporto di causalità tra eventi (argomento trattato nella sezione successiva, *gestione del rapporto di causalità e del tempo virtuale*).

1.2.1 Gestione del rapporto di causalità e del tempo virtuale

Il rispetto del rapporto di causalità tra eventi è una caratteristica cruciale in un simulatore. È infatti necessario poter sempre stabilire tra due eventi coinvolti nella stessa catena causale[5] quale dei due è avvenuto prima e ha causato l'altro, per poterli gestire in tale ordine.

Per rispettare il rapporto di causalità è necessario introdurre nella simulazione il concetto di tempo virtuale (o simulato) [4]. Il tempo virtuale è un valore numerico che viene incrementato all'avvenire di ogni evento[6], e rende possibile determinare con sicurezza se un evento è avvenuto prima di un altro, controllando quale evento è associato a un valore

[4]Caricare un certo elaboratore di lavoro può significare sia far gestire a quell'elaboratore un numero maggiore di entità, sia far gestire a quell'elaboratore le entità che richiedono maggiori capacità di calcolo.

[5]Se due eventi appartengono a due catene causali scorrelate non è necessario stabilire quale è avvenuto prima, perché ai fini della sequenza di eventi non cambia nulla se viene gestito prima un certo evento rispetto ad un altro [4].

[6]La dimensione dell'incremento può essere regolare, oppure determinata da agenti esterni, come ad esempio l'avanzamento del tempo reale nel caso di una simulazione interattiva.

temporale (o *timestamp*) inferiore rispetto all'altro.

Il tempo virtuale non necessariamente corrisponde al tempo reale (WCT, Wall Clock Time). Nel caso delle simulazioni studiate per realizzare ambienti virtuali interattivi (come i videogiochi [20]) il tempo virtuale viene percepito da un essere umano, e pertanto, affinché il risultato dia luogo a una percezione realistica, l'avanzamento del tempo virtuale deve essere ancorato allo scorrimento del tempo reale. Nel caso della simulazione analitica invece, ciò che interessano sono i dati raccolti durante e al termine della simulazione, che devono essere prodotti nel più breve tempo possibile, pertanto l'avanzamento del tempo virtuale non è vincolato allo scorrere del tempo reale.

Rispetto della causalità nella simulazione parallela e distribuita

Affinché una simulazione parallela e distribuita rispetti il requisito di riproducibilità[7] è necessario che venga rispettato da ogni LP il vincolo di causalità locale [3]. In altre parole, significa che ogni LP deve eseguire gli eventi che lo riguardano nell'ordine temporale dato dai timestamp associati a tali eventi.

A causa della natura distribuita del sistema però, per evitare che venga violato il vincolo di causalità locale è necessaria l'introduzione di algoritmi di sincronizzazione tra LP, che si dividono in due principali categorie: algoritmi pessimistici (o conservativi) e algoritmi ottimistici [3, 5, 6, 10].

- Gli algoritmi pessimistici [3, 9] servono a evitare a priori che il vincolo venga violato, a costo di bloccare per il tempo necessario LP che sarebbero in grado di eseguire gli eventi di cui già dispongono, eventi che potrebbero però essere associati a un timestamp successivo a quello di eventi non ancora arrivati a tali LP. Tra gli algoritmi appartenenti a questa categoria, due importanti esempi sono l'algoritmo *Chandy-Misra-Bryant* (CMB) [9] e l'algoritmo *Time-stepped* [3]. In particolare *Time-stepped* è l'algoritmo di sincronizzazione utilizzato nell'architettura *GAIA-*

[7]Per rispettare il requisito di riproducibilità, una simulazione deve ripetere sempre la stessa sequenza di eventi e restituire sempre gli stessi risultati, ogni volta in cui viene eseguita a partire dallo stesso insieme di parametri (ad esempio i semi random, il numero di entità o determinati valori di costanti del modello della simulazione).

FaultTolerance di cui tratta questa tesi, e consiste nella suddivisione del tempo simulato in slot temporali (i *timestep*). I timestep costituiscono delle barriere temporali, per cui gli eventi appartenenti a un certo timestep vengono considerati concorrenti, mentre un evento con timestep superiore rispetto ad un altro viene considerato successivo. Il vincolo di causalità locale in questo modo viene sempre rispettato, a patto che gli LP avanzino di timestep in modo sincronizzato (in pratica questo significa che quando un LP ha terminato l'esecuzione di tutti gli eventi del proprio timestep, questo aspetta la fine del timestep di tutti gli altri LP prima di avanzare al timestep successivo).

- Gli algoritmi ottimistici [3, 11] (di cui il più importante esempio è *Time Warp* [11]) invece permettono la violazione locale del vincolo. Nel momento in cui questa violazione viene rilevata (ad esempio arriva ad un LP un messaggio contrassegnato da un timestamp precedente rispetto a quelli degli eventi già processati), si attivano meccanismi di rollback che riportano l'intero sistema ad uno stato coerente. Effettuato il rollback, la simulazione può poi ricominciare dallo stato coerente ripristinato, ora però potendo considerare anche gli eventi che avevano inizialmente causato la violazione del vincolo.

1.2.2 High Level Architecture (HLA), IEEE 1516

La simulazione distribuita ha sempre ricoperto un ruolo importante in ambito militare. Per un lungo periodo le simulazioni a scopo militare sono sempre state realizzate senza tener conto delle eventuali esigenze di dover essere scomposte in componenti e di dover essere messe in comunicazione le une con le altre. Queste esigenze, inizialmente trascurate, sono diventate però motivo di interesse al verificarsi di due eventi. Il primo è la fine della guerra fredda, che ha causato forti tagli alle spese militari dei paesi coinvolti, e questo ha aumentato l'interesse per il risparmio sullo sviluppo delle simulazioni, garantito dalla riusabilità del codice di simulazioni già realizzate. Il secondo è l'esplosione di Internet, che ha generato un forte interesse nella possibilità di mettere in comunicazione più simulatori, allo scopo di realizzare simulazioni composte da più entità distanti geograficamente.

L'unico modo per soddisfare queste esigenze, riusabilità del codice e interoperabilità tra più simulazioni, era quello di creare uno standard per le simulazioni distribuite, e così il dipartimento della difesa USA, collaborando con enti di ricerca e privati, ha dato vita allo standard High Level Architecture [7, 8] (HLA, IEEE Standard 1516).

HLA è uno standard che oltre a garantire riusabilità e interoperabilità delle simulazioni realizzate con questo metodo, introduce un'altra importante caratteristica: la separazione della logica del modello di simulazione dall'implementazione del sistema distribuito che esegue la simulazione. Questa caratteristica semplifica lo sviluppo e l'utilizzo delle simulazioni, perché permette agli sviluppatori e agli utenti di non essere a conoscenza dell'implementazione del sottostante sistema distribuito.

Lo standard HLA non riguarda minimamente l'implementazione dei simulatori distribuiti, che possono essere sviluppati nel modo ritenuto più opportuno, bensì definisce solo alcune specifiche concettuali e un insieme di regole che le implementazioni devono rispettare, tra cui le seguenti.

L'unità minima di simulazione secondo HLA è il *federato*, che può essere associato concettualmente al LP.

La simulazione distribuita nella sua interezza invece viene considerata una *federazione*, composta dai seguenti elementi:

- un insieme di federati, che costituiscono le entità della simulazione;
- una definizione delle possibili interazioni tra i federati, chiamata *Federation Object Model* (FOM);
- una *Runtime Infrastructure* (RTI) che rappresenta di fatto l'implementazione del sistema distribuito sottostante alla simulazione, e che rende possibile l'esecuzione dei federati e delle loro interazioni.

Il Federation Object Model a sua volta è parte di un documento più ampio, chiamato Object Model Template, e composto da:

- *Federation Object Model* (FOM), che è una definizione delle possibili interazioni tra federati all'interno della federazione;
- *Simulation Object Model* (SOM), che è una definizione degli oggetti usati all'interno del singolo federato.

1.3 Simulazione parallela e distribuita fault tolerant presente in letteratura

In letteratura non sono presenti molti articoli [12] correlati all'argomento discusso in questa tesi, la tolleranza ai guasti applicata alla simulazione parallela e distribuita. Questi articoli, pur essendo correlati a tale argomento, espongono approcci e metodologie differenti rispetto a quelle scelte nella realizzazione di GAIAFaultTolerance, il progetto software le cui progettazione, implementazione e valutazione costituiscono il principale argomento esposto in questa tesi.

Le tecniche di tolleranza ai guasti nei sistemi distribuiti si possono catalogare in due categorie principali: tecniche basate sulla ridondanza e tecniche basate sull'utilizzo di checkpoint [14].

- Le tecniche basate sulla ridondanza prevedono l'utilizzo di risorse extra per replicare elementi del sistema e per mantenere sincronizzate le copie di tali elementi, in modo che se un guasto colpisce un elemento del sistema, può subentrare al suo posto una sua replica.
- Le tecniche basate su checkpoint prevedono la memorizzazione su memoria stabile dei checkpoint durante l'esecuzione del sistema, in modo che il sistema distribuito, in seguito a un eventuale riavvio causato da un guasto, possa ricominciare la propria esecuzione partendo dall'ultimo checkpoint memorizzato.

Il progetto GAIAFaultTolerance consiste nella realizzazione di un'estensione al framework per la simulazione parallela e distribuita GAIA (cap. 2), e la metodologia implementata in questa estensione appartiene alla categoria delle tecniche basate sulla

ridondanza. Prevede infatti tecniche di tolleranza a guasti basate sulla replicazione delle entità presenti nelle simulazioni (cap. 3).

Negli articoli correlati si possono individuare invece le metodologie seguenti.

An architecture for fault-tolerant HLA-based simulation (Berchtold, Hezel, 2001)

Nell'articolo *An architecture for fault-tolerant HLA-based simulation* [13] (C. Berchtold, M. Hezel, 2001) si descrive un'architettura per realizzare simulazioni tolleranti ai guasti basata sullo standard High Level Architecture [7, 8] (HLA, IEEE Standard 1516) brevemente descritto in sez. 1.2.2.

Quest'architettura, chiamata *R-Fed* (Replica Federate), è basata sulla replicazione dei federati[8], e tra quelle esaminate è quella che più si avvicina concettualmente a GAIAFaultTolerance.

L'architettura consiste nella sostituzione del semplice federato con un sistema distribuito composto dai seguenti elementi, che possono (e devono, per garantire la tolleranza ai crash) trovarsi su host differenti:

- i componenti FT, che si occupano di dialogare direttamente con la Runtime Infrastructure[9], e che vengono visti da quest'ultima come dei normali federati;
- le repliche del federato.

I componenti FT a loro volta sono i seguenti:

- l'*FT Manager*, che si occupa di dialogare direttamente con le repliche dei federati e la RTI;
- la *Compare Unit*, che esamina le differenze tra le interazioni che le repliche dei federati scambiano con l'FT manager (e quindi indirettamente con la RTI);

[8] In HLA i Logical Process (LP) prendono il nome di federati (sez. 1.2.2).

[9] In HLA la Runtime Infrastructure (RTI) è l'infrastruttura che implementa la comunicazione tra i federati (sez. 1.2.2).

- il *Fault Detector*, che esamina i dati forniti dalla Compare Unit per rilevare eventuali guasti nelle Repliche;
- il *Property Manager*, che si occupa della gestione di tutti i parametri dell'architettura FT (numero di repliche, criteri per identificare i guasti, ecc.).

L'FT Manager agisce come un filtro, è connesso infatti alla RTI e comunica con essa come se fosse un normale federato, nascondendole l'architettura basata su repliche. I dati che scambia con la RTI, visti da essa come provenienti da un unico federato, sono in realtà una mediazione dei dati provenienti dalle varie repliche del federato, mediazione decisa dal Fault Detector.

Il Fault detector infatti è in grado di rilevare, oltre al semplice crash di alcune repliche, anche guasti bizantini, con controlli basati sul raggiungimento di una maggioranza (argomento trattato più approfonditamente in sez. 3.1.2). Questo meccanismo è lo stesso implementato in GAIAFaultTolerance, anche se applicato in modo differente.

Citando l'articolo, la tolleranza ai guasti bizantini descritta è possibile solo nel caso in cui questi colpiscano le repliche dei federati. Nel caso di guasti che colpiscono uno o più dei componenti FT, possono essere tollerati solo guasti di tipo crash, con meccanismi di checkpoint e ripristino del componente caduto.

Infine, l'architettura descritta nell'articolo (R-Fed) ha molti punti in comune con GAIAFaultTolerance (GAIAFT), ma anche alcune sostanziali differenze descritte in seguito.

- La granularità delle entità simulate è differente: in R-Fed la minima unità di simulazione è il federato (corrispondente a un LP), mentre in GAIAFT la minima unità di simulazione è l'entità simulata, e gli LP sono di fatto dei "contenitori" di entità simulate. Questo determina una differenza anche nella replicazione: in R-Fed infatti le repliche presenti sono copie di interi federati, mentre in GAIAFT le repliche presenti sono copie delle singole entità simulate, non esiste infatti in GAIAFT nessuna copia di un intero LP.
- Il punto precedente determina un'altra differenza. In R-Fed lo scopo della tolleranza ai guasti è impedire che un determinato federato si guasti, sostituendolo con

una replica quando questo succede. In GAIAFT invece un LP è libero di guastarsi senza che venga effettuata nessuna operazione per ripristinarlo. Questo perché, per i meccanismi di ridondanza implementati in GAIAFT, la simulazione continua a funzionare anche se cadono alcuni LP.

Lo scopo di GAIAFT infatti è quello di "spalmare" le copie della porzione di simulazione interna ad ogni LP sull'intero sistema distribuito, suddividendole tra i vari LP. In questo modo, se un LP si guasta (crashando o cominciando a comunicare dati errati), la porzione di simulazione che veniva gestita da esso continua a sopravvivere all'interno degli altri LP.

- Anche se apparentemente il filtro fornito dall'FT Manager di R-Fed ricorda il sistema scelto per GAIAFT, ciò che cambia è il collocamento di tale filtro. In R-Fed, l'FT Manager si frappone tra la RTI e le repliche dei federati, inviando i messaggi diretti a un federato a tutte le relative repliche, e filtrando i messaggi inviati dalle repliche, per dare alla RTI solo un messaggio per ogni insieme di copie.

  In GAIAFT invece il filtro si frappone tra il modello della simulazione e i layer sottostanti. È il modello della simulazione che viene trasformato da GAIAFT in un modello di simulazione equivalente, che prevede delle copie per ogni entità presente (sez. 3.1). Al contrario di quanto succede in R-Fed, la ridondanza, nascosta solo ai layer superiori, è perfettamente visibile ai layer sottostanti (compresi LP e RTI), e le copie delle entità vengono trattate come normali entità (dal punto di vista dei layer sottostanti, le entità e le copie sono addirittura indistinguibili).

- Infine un'altra differenza derivante dal punto precedente è relativa alla connessione tra componenti FT e repliche in R-Fed: citando l'articolo, questa connessione non può essere effettuata per mezzo della RTI (ai cui occhi tutto questo sistema non esiste, considerandolo un semplice federato), ma richiede degli appositi canali di comunicazione (ad esempio realizzati tramite connessioni TCP/IP). Anche GAIAFT sfrutta connessioni TCP/IP dedicate, ma questo avviene solo in fase di inizializzazione del sistema, dopodiché vengono utilizzate dagli LP solo le connessioni offerte dalla RTI.

Fault-Tolerant Distributed Simulation (Damani, Garg, 1998)

Nell'articolo *Fault-Tolerant Distributed Simulation* [14] (Om. P. Damani, Vijay K. Garg, 1998) viene descritto un metodo per introdurre tecniche di tolleranza ai guasti in un simulatore distribuito basato sull'approccio ottimistico [3].

Disponendo già di checkpoint e meccanismi di rollback dovuti all'approccio ottimistico del simulatore, le tecniche di tolleranza ai guasti descritte sono basate sull'uso di tali checkpoint, memorizzati su memoria stabile. In questo modo, in caso di crash (anche dell'intero sistema) è possibile disporre di uno stato coerente da cui ricominciare la simulazione, dato dall'insieme dei checkpoint memorizzati su memoria stabile.

Si tratta di un approccio molto differente da quello scelto per lo sviluppo di GAIAFaultTolerance. Quest'ultimo infatti, a differenza di quello descritto nell'articolo, è basato su un simulatore distribuito con approccio pessimistico (più precisamente approccio time-stepped). Un'altra importante differenza risiede nella categoria di tecniche per la tolleranza ai guasti implementate: GAIAFaultTolerance utilizza tecniche basate sulla ridondanza delle entità simulate, mentre l'approccio descritto nell'articolo prevede tecniche basate sull'uso di checkpoint. L'uso dei checkpoint limita però la tolleranza ai soli crash, mentre utilizzando tecniche di ridondanza è possibile estendere la tolleranza anche ad altri tipi di guasto. Un altro vantaggio delle tecniche di ridondanza è che in caso di guasti questa permette alla simulazione di continuare senza nessuna interruzione, mentre nel caso dell'uso dei checkpoint il sistema si blocca quando avvengono crash, e deve essere comunque riavviato per poter riprendere il lavoro dall'ultimo checkpoint.

Per contro, la soluzione proposta dall'articolo citato può resistere anche a un eventuale crash dell'intero sistema, mentre nel caso delle tecniche basate sulla ridondanza, il numero di guasti che il sistema può tollerare è limitato da un determinato valore. Questo valore viene scelto dall'utente ma è comunque sempre strettamente inferiore rispetto al numero di elementi che compongono il sistema, oltre ad aumentare l'overhead del sistema proporzionalmente al numero di guasti tollerati.

Replicated objects in timewarp simulations (Agrawal, Agre, 1992)

Nell'articolo *Replicated objects in timewarp simulations* [15] (D. Agrawal, J. R. Agre, 1992) si descrive un approccio molto più simile a quello scelto per GAIAFaultTolerance, basato sull'utilizzo di repliche di elementi del sistema, che subentrano a tali elementi in caso di crash.

Ci sono comunque alcune differenze tra l'approccio descritto nell'articolo e GAIAFaultTolerance. La differenza principale è data dalla differente granularità degli elementi che vengono replicati. Nell'architettura descritta nell'articolo gli elementi che vengono replicati sono gli interi LP che compongono un simulatore distribuito basato su time warp. In GAIAFaultTolerance invece gli elementi che vengono replicati non sono gli LP nella loro interezza, ma le singole entità simulate. Questo è reso possibile da una caratteristica dell'ambiente GAIA, per la quale gli LP non rappresentano nel modello della simulazione singole entità simulate, bensì contenitori di entità (sez. 2.2). Questa caratteristica rende quindi l'overhead dovuto alla replicazione delle entità più finemente regolabile, e rende inoltre il sistema maggiormente adattabile e dinamico grazie al fatto che le repliche delle entità simulate sono libere di migrare da un LP ad un altro durante lo svolgimento della simulazione (sez. 2.2).

L'altra importante differenza è data dal fatto che il simulatore distribuito descritto nell'articolo è basato su algoritmi di sincronizzazione ottimistici (più precisamente time warp) mentre GAIAFaultTolerance utilizza l'approccio pessimistico time-stepped. Questo fatto rende le due architetture molto differenti, a causa delle forti differenze tra i due approcci. L'architettura trattata nell'articolo infatti deve gestire le problematiche introdotte dall'uso di checkpoint e rollback [3, 6].

Infine, i tipi di guasto trattati nell'articolo si limitano ai semplici crash, ma, citando l'articolo, la tolleranza può essere estesa anche ad altri tipi di guasto (vengono citati ad esempio guasti intermittenti e guasti bizantini).

A framework for fault-tolerance in HLA-based distributed simulations (Eklof, Moradi, Ayani, 2005)

L'articolo *A framework for fault-tolerance in HLA-based distributed simulations* [16] (M. Eklof, F. Moradi, R. Ayani, 2005) descrive un framework basato sullo standard High

Level Architecture [7, 8] (HLA, IEEE Standard 1516) brevemente descritto in sez. 1.2.2.

Questo framework, chiamato Distributed Resource Management System (DRMS) implementa tecniche di tolleranza ai guasti basate sulla memorizzazione su memoria stabile e il ripristino di checkpoint in caso di guasti che colpiscono i federati.

L'algoritmo scelto per la gestione del tempo nel framework descritto è time warp (approccio ottimistico), e questa scelta si ricollega all'utilizzo dei checkpoint all'interno delle tecniche di tolleranza ai guasti utilizzate.

In cado di caduta dell'host, il ripristino dei federati persi viene effettuato per mezzo di una migrazione di tali federati dall'host caduto ad un altro funzionante, migrazione effettuata tramite la creazione del federato nell'host destinazione e il ripristino dell'ultimo checkpoint relativo al federato. Questo sistema permette al framework di adattarsi dinamicamente alle variazioni del sistema distribuito in seguito alla caduta di alcuni nodi.

I guasti ai federati che vengono tollerati dal framework si limitano a quelli di tipo crash, e sono i seguenti:

- crash dell'elaboratore host del federato;
- crash del singolo federato;
- interruzione permanente del collegamento di rete tra il federato e la Runtime Infrastructure (questo caso, anche se non è propriamente un crash del federato si comporta come tale, causando un'interruzione permanente del servizio).

Implement Fault Tolerant in distributed DEVS simulation (Bin Chen, Xiao-gang Qiu, 2009)

L'articolo *Implement Fault Tolerant in distributed DEVS simulation* [17] (Bin Chen, Xiao-gang Qiu, 2009) descrive un'architettura per la simulazione distribuita che, al contrario degli esempi visti precedentemente, sfrutta un sistema distribuito composto da processi eterogenei. I processi che compongono il sistema distribuito infatti svolgono ruoli differenti, e in prima analisi il sistema può essere scomposto in due sezioni: Cluster

e Controller.

La sezione Cluster è equivalente a un simulatore distribuito tradizionale, essendo composta da tutti i processi che implementano il modello della simulazione. Questi processi (Model Server) sono tra loro tutti equivalenti tranne il Root Server, che è analogo a un normale Model Server, ma in più si occupa di alcune funzioni di coordinamento degli altri processi (ad esempio ferma la simulazione quando viene raggiunta la condizione di terminazione).

La sezione Controller è la parte che si occupa di gestire la tolleranza ai guasti, ed è composta da due processi: Master Server e Logger Server. Il Master Server è il processo che si occupa di rilevare la presenza di guasti tramite dei ping periodici ai Model Server, e di, in caso di presenza di guasti, eseguire le procedure di ripristino dei Model Server caduti (che consistono in riavvio del processo e ripristino dello stato locale). Il Logger Server invece è il processo che si occupa di dialogare con i Model Server durante lo svolgimento della simulazione e di memorizzare gli stati dei Model Server in un apposito archivio di cui dispone (States Pool).

Come già accennato, il sistema di rilevamento dei guasti effettuato dal Master Server consiste nell'esecuzione di ping periodici ai Model Server, e nel rilevamento di un guasto nel caso in cui un Model Server non risponda entro un certo timeout. Questo sistema limita quindi la tolleranza ai soli guasti di tipo crash (citando l'articolo, i guasti tollerati sono crash hardware delle macchine del cluster, crash del sistema operativo delle macchine del cluster e crash dei processi appartenenti al cluster). Inoltre, sempre citando l'articolo, un altro vincolo del sistema è che Master Server e Logger Server non possono subire crash, devono essere considerati sempre affidabili.

Capitolo 2

L'ambiente GAIA/ARTÌS

Il progetto GAIAFaultTolerance trattato da questa tesi (cap. 3) si appoggia al framework per la simulazione parallela e distribuita *GAIA/ARTÌS* [1, 2].

GAIA/ARTÌS è un software composto da due layer, descritti nelle sezioni successive: ARTÌS è una Runtime Infrastructure (sez. 1.2.2), mentre GAIA è un layer che si appoggia ad ARTÌS, e introduce meccanismi di adattabilità a run-time del sistema distribuito alle condizioni dell'infrastruttura sottostante.

2.1 ARTÌS

Advanced RTI System [1, 2] (ARTÌS) è un middleware per la simulazione parallela e distribuita parzialmente ispirato ad una Runtime Infrastructure prevista da HLA [7, 8] (IEEE Standard 1516, descritto in sez. 1.2.2).

ARTÌS è un'architettura distribuita composta da vari Logical Process (LP) che è stata progettata per supportare un'alta scalabilità dei modelli di simulazione e per poter essere eseguita su un'infrastruttura hardware composta da un alto numero di PEU (Physical Execution Unit, sez. 1.2).

Un'importante caratteristica di ARTÌS consiste nell'uniformare e gestire adattivamente le possibili forme di comunicazione tra LP, allo scopo di ridurre le latenze dovute ai messaggi scambiati tra gli LP, ottenendo così simulazioni più veloci e performanti.

Più precisamente, ARTÌS fornisce una API comprensiva di funzioni generiche per lo scambio di messaggi (send, receive, broadcast, ecc.), che vengono usate indistintamente dagli LP senza che questi debbano porsi il problema di come è strutturata l'architettura hardware sottostante. Ciò che viene eseguito da queste funzioni invece cambia a seconda dell'infrastruttura sottostante: se per mezzo della API di ARTÌS avviene una comunicazione tra due LP che si trovano sullo stesso elaboratore, questa viene effettuata per mezzo di memoria condivisa; se invece la comunicazione avviene tra LP che si trovano su elaboratori differenti connessi in rete, questa viene effettuata per mezzo di protocolli di rete (TCP/IP).

In questo modo, chi implementa un simulatore basato su ARTÌS non deve minimamente preoccuparsi di implementare differenti metodi di comunicazione in base alle caratteristiche dell'infrastruttura hardware sottostante, sapendo al tempo stesso che per ogni comunicazione verrà scelto sempre il metodo che garantisce le prestazioni migliori, con i risultati di rendere le simulazioni più veloci e di renderne l'implementazione totalmente slegata dall'infrastruttura hardware su cui esse verranno eseguite.

ARTÌS inoltre fornisce servizi per la gestione del tempo virtuale (sez. 1.2.1), e supporta i principali algoritmi di sincronizzazione, sia con approccio pessimistico (Chandy-Misra-Bryant [9] e Time-stepped [3]) sia con approccio ottimistico (Time Warp [11]).

2.1.1 Simulation Manager

L'architettura distribuita di ARTÌS prevede un certo numero di processi (LP) di pari ruolo, senza quindi la presenza di elementi centralizzati. Esiste però nel sistema un'eccezione: un elemento centralizzato chiamato *SImulation MAnager* (SIMA).

Il Simulation Manager è un processo che serve a inizializzare e coordinare gli LP. Più precisamente ricopre vari ruoli, tra cui principalmente i seguenti:

- in fase di inizializzazione della simulazione, il SIMA analizza l'infrastruttura hardware su cui vengono eseguiti gli LP e li istruisce sui metodi di comunicazione che dovranno utilizzare (ad esempio gli LP che si trovano sullo stesso host dovranno comunicare tra loro tramite memoria condivisa);

- durante l'esecuzione della simulazione, il SIMA fornisce allo sviluppatore della simulazione la possibilità di disporre della funzione `SIMA_Barrier`, che costituisce una barriera di sincronizzazione per gli LP, utilizzabile dallo sviluppatore nel modo che egli ritiene più opportuno.

2.2 GAIA

Generic Adaptive Interaction Architecture [1, 2] (GAIA) è un framework che si appoggia al middleware per la simulazione parallela e distribuita ARTÌS.

Una delle caratteristiche di GAIA è l'introduzione di un paradigma per la creazione di modelli di simulazione basato sulle entità simulate: la minima unità di simulazione non è più quindi il LP, bensì l'entità simulata. In una simulazione GAIA infatti un singolo LP ricopre il ruolo di un "contenitore" di entità, potendo gestire un numero arbitrario di entità simulate tra loro indipendenti, al contrario di quanto accadrebbe in un simulatore distribuito tradizionale, in cui ogni entità dovrebbe essere associata a un proprio LP.

Il paradigma basato sugli LP visti come contenitori di entità ha reso possibile l'introduzione di una delle principali funzionalità offerte da GAIA: la migrazione delle entità. Questa funzionalità serve a rendere le simulazioni distribuite adattabili a run-time alle condizioni della rete, necessità generatasi dal seguente motivo.

Come spiegato nella sezione precedente, in una simulazione basata sul middleware ARTÌS gli LP possono comunicare tra loro tramite canali di comunicazione caratterizzati da prestazioni molto differenti (memoria condivisa e rete). Appare evidente quindi che il sistema risulterebbe ottimizzato se i canali di comunicazione più efficienti venissero utilizzati dalle entità che comunicano tra loro più spesso, e quelli meno efficienti venissero utilizzati da entità che comunicano tra loro raramente.

Questa ottimizzazione si può ottenere allocando nello stesso LP o in LP che si trovano vicini (cioè che possono comunicare con basse latenze) le entità che comunicano tra loro più spesso, e allocando in LP distanti (cioè che comunicano con alte latenze) le entità che comunicano tra loro più raramente.

L'allocazione delle entità purtroppo però non può essere decisa a priori, perché il tasso di comunicazione tra le entità può variare dinamicamente durante l'esecuzione della simulazione. È quindi necessario poter cambiare dinamicamente l'allocazione delle entità, e questa funzionalità viene fornita in GAIA dalla migrazione delle entità, cioè dalla possibilità di far migrare le entità da un LP all'altro, per ottimizzare l'utilizzo dei canali di comunicazione nei modi sopra citati.

L'analisi delle condizioni della rete, l'analisi dei tassi di comunicazione tra le varie entità e l'attuazione delle migrazioni vengono effettuate da GAIA durante l'esecuzione della simulazione: quando GAIA decide che un'entità deve migrare verso un certo LP informa per mezzo di messaggi il layer superiore, che dovrà provvedere a serializzare lo stato dell'entità e a spedirlo al LP di destinazione tramite l'apposita funzione `GAIA_Migrate`; il LP destinazione che riceve lo stato potrà quindi ricreare localmente l'entità migrata e cominciare a gestirla.

Come già detto le migrazioni di entità da cui è caratterizzata GAIA vengono decise solo sulla base di analisi delle condizioni della rete e dei tassi di comunicazione tra le entità simulate. Allo scopo di introdurre meccanismi di bilanciamento del carico di lavoro tra i vari LP, è stata sviluppata *GAIA+* [2], un'estensione di GAIA che, oltre alle analisi sopra citate, esegue anche analisi del bilanciamento del carico di lavoro, e decide di effettuare migrazioni volte a ottimizzarlo.

Capitolo 3

Progettazione di GAIAFaultTolerance

L'obiettivo di questa tesi è la progettazione e l'implementazione di un'estensione all'ambiente per la simulazione distribuita GAIA-ARTÌS (cap. 2).

Questa estensione, chiamata GAIAFaultTolerance (o GAIAFT), serve ad aggiungere al sistema meccanismi di tolleranza ai guasti, basati sulla ridondanza delle entità presenti nelle simulazioni.

I guasti tollerati da questo sistema sono i crash e i guasti bizantini. Un singolo crash può essere costituito da un crash hardware di un elaboratore, un crash software del sistema operativo, un crash software di un LP[1], o l'interruzione della connessione tra un elaboratore e il resto del sistema distribuito. Un singolo guasto bizantino invece può essere costituito da un qualunque comportamento anomalo di alcune o tutte le entità gestite da un singolo LP. In questo comportamento anomalo sono compresi assenza di comunicazione con altri elementi del sistema, comunicazione di dati errati con altri elementi del sistema, ecc. (maggiori approfondimenti in sez. 3.1.1 e in sez. 3.1.2).

La struttura di GAIAFT è quella di un ulteriore layer che si appoggia sopra a GAIA, fornendo funzionalità al livello superiore, il modello della simulazione.

[1]Un LP è uno dei processi che compongono un simulatore distribuito, vedere cap. 2.

La particolarità del layer GAIAFT è che, a meno di alcune eccezioni (trattate in seguito, in sez. 5.1), non fornisce funzionalità aggiuntive rispetto a GAIA: le funzionalità offerte sono le stesse di GAIA, e quella che cambia è solo l'implementazione sottostante. In questo modo un modello di simulazione già realizzato per GAIA, può essere riutilizzato su GAIAFT senza dover essere reimplementato (tranne che per poche modifiche relative alle eccezioni sopra citate).

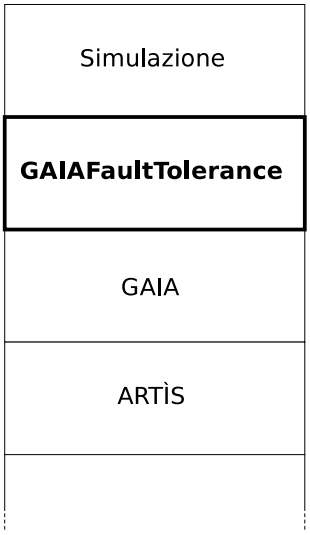

Figura 3.1: Layer che compongono un simulatore GAIAFT.

3.1 Tolleranza ai guasti tramite entità replicate

Il meccanismo su cui è basata la tolleranza ai guasti è la replicazione delle entità del modello della simulazione. L'idea di base è che una simulazione composta da N entità che si appoggia sul layer GAIAFT, viene trattata come una simulazione di $N*M$ entità (dove M è il numero di copie di ogni entità) in cui le M copie di ogni entità seguono l'evoluzione della simulazione indipendentemente l'una dall'altra, ma comportandosi tutte nello stesso modo.
In questo modo, se dovessero verificarsi la perdita o la corruzione di un'entità, esisterebbe una sua copia identica che potrebbe rimpiazzarla senza nessuna perdita di informazioni.

Questo comportamento è nascosto agli occhi dell'utente, che appoggiandosi al layer GAIAFT vedrebbe solo le N entità del proprio modello di simulazione, quando in realtà l'evoluzione di ognuna di quelle entità è simulata tramite l'evoluzione delle rispettive M copie.

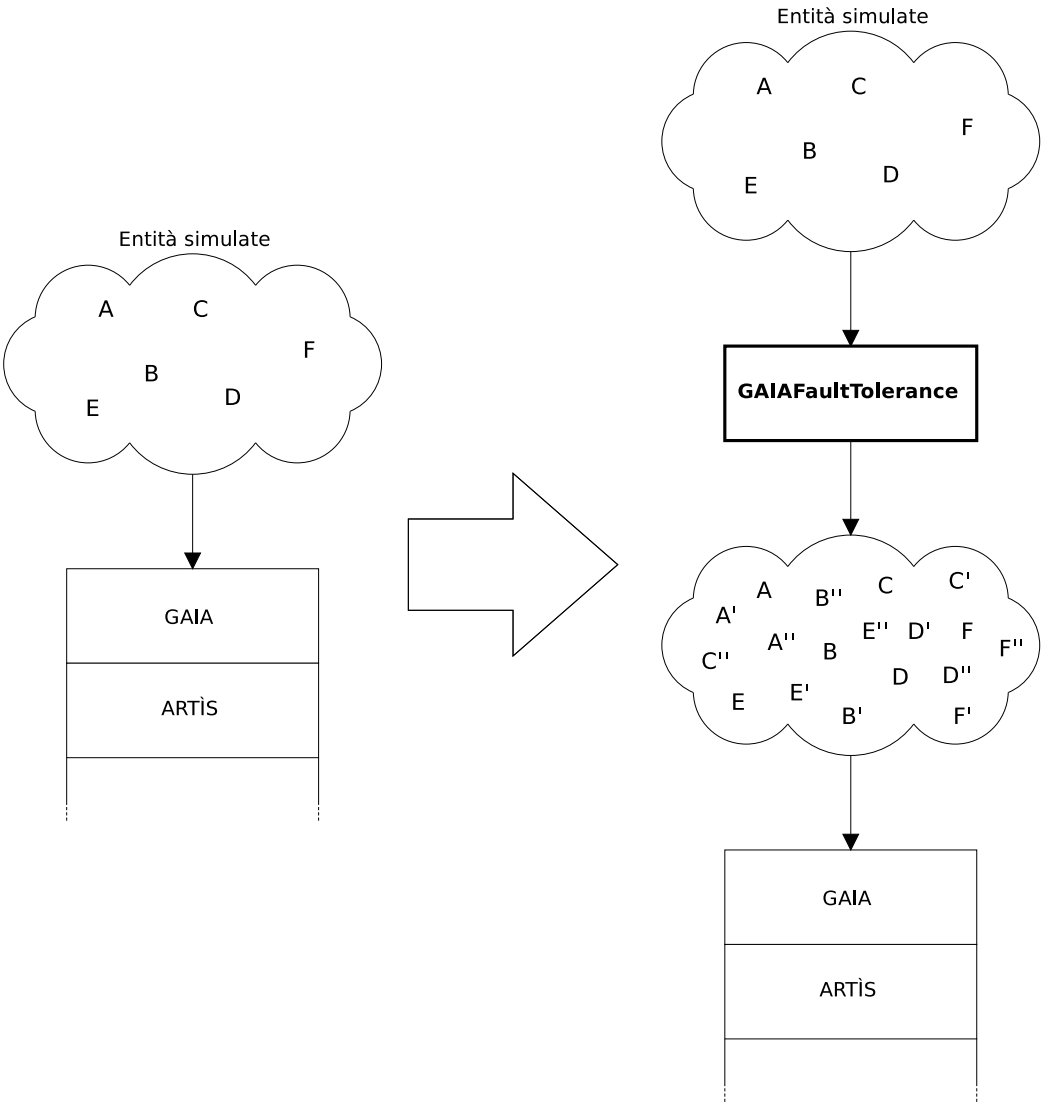

Figura 3.2: Replicazione delle entità attuata dal layer GAIAFT verso i layer inferiori.

3.1.1 Tolleranza ai crash

I guasti di tipo crash consistono nella cessazione di tutte le attività di una o più unità di elaborazione di un sistema distribuito (un esempio è lo spegnimento improvviso di uno dei computer che stanno elaborando un sistema distribuito).

Nel caso di una simulazione distribuita GAIA, il crash di un LP comporta la perdita dello stato di tutte le entità ospitate su quel LP. Per fare in modo che in questa simulazione rimanga viva almeno una copia per ogni entità del modello, è sufficiente avere in totale due copie per ogni entità, rispettando però un importante vincolo: le due copie di un'entità devono trovarsi su due LP diversi. In questo modo, qualunque sia il LP che subisce il crash, per ogni entità presente su quel LP, esiste una copia dell'entità su un altro LP, ottenendo così la tolleranza a un crash (fig. 3.3).

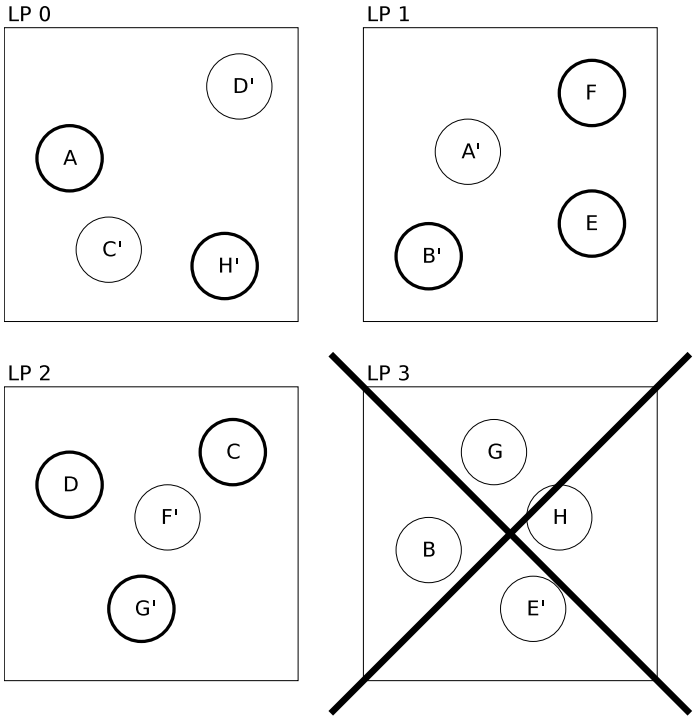

Figura 3.3: Esempio di tolleranza a un crash. Nel sistema sono presenti due copie di ogni entità. Si può osservare che in seguito a un crash rimane sempre almeno una copia viva per ogni entità (evidenziata in grassetto).

Estendendo questo metodo al caso di più crash all'interno del sistema distribuito, si può vedere in fig. 3.4 che se avvengono N crash, sono necessarie $N+1$ copie per ogni entità, distribuite su $N+1$ LP distinti, per poter avere la certezza di conservare almeno una copia viva di ogni entità del modello. Infatti, se fossero presenti solo N copie di ogni entità, nel caso pessimo le N copie di una certa entità potrebbero trovarsi negli N LP che hanno subito i crash, mentre se le copie sono almeno $N+1$ e tutte su LP diversi, allora in seguito a N crash rimane obbligatoriamente almeno una copia viva in uno degli LP che non hanno subito i crash (una dimostrazione matematica di questa caratteristica è trattata nel cap. 4).

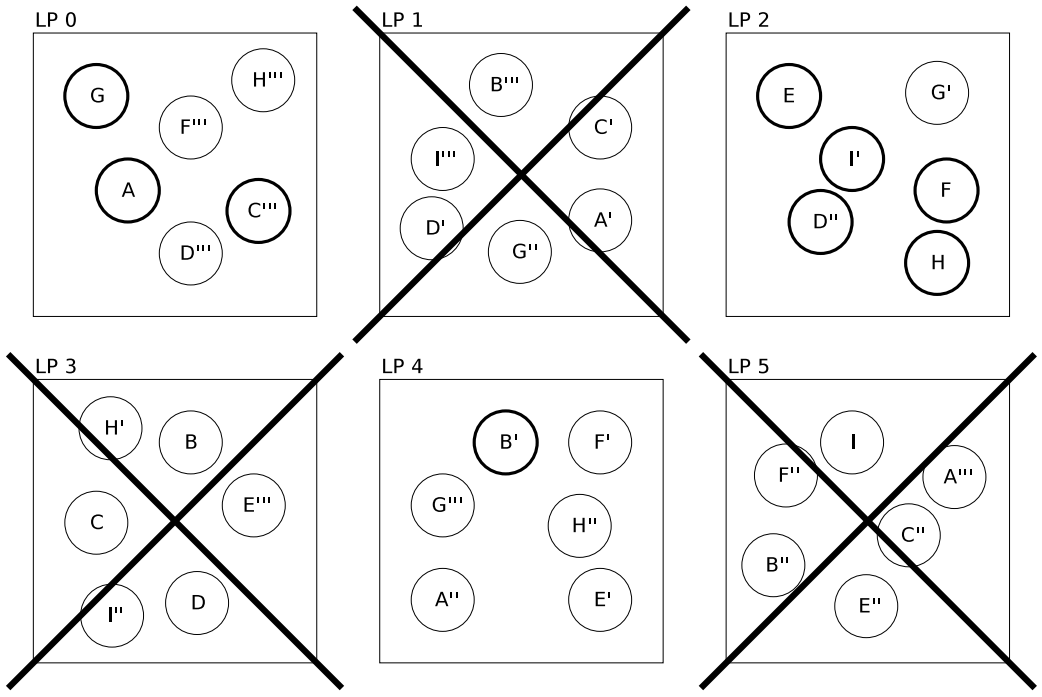

Figura 3.4: Esempio di tolleranza a tre crash. Nel sistema sono presenti quattro copie di ogni entità. Si può osservare che in seguito a tre crash rimane sempre almeno una copia viva per ogni entità (evidenziata in grassetto).

Il vincolo per cui le copie di una stessa entità devono obbligatoriamente trovarsi su LP distinti determina intrinsecamente un altro vincolo: il numero minimo di LP deve

essere sempre maggiore o uguale al numero di copie necessarie (nel caso di tolleranza a N crash quindi la simulazione deve essere distribuita su almeno $N + 1$ LP).

Da questo fatto inoltre, è interessante notare che, nel caso in cui il numero di LP (ad esempio $N + 1$) sia esattamente uguale al numero di copie previste dalla tolleranza scelta, ogni LP contiene una copia di tutte le entità presenti nella simulazione. In questo caso quindi, il risultato sarebbe analogo a quello ottenuto eseguendo $N + 1$ simulazioni indipendenti non tolleranti ai crash (ognuna gestita da un singolo LP), di cui potrebbero crasharne fino a N, in modo da garantire la sopravvivenza di almeno una simulazione completa.

Nel caso di questo scenario quindi si otterrebbe la stessa tolleranza ai crash, ma senza introdurre l'overhead di comunicazione causato dalla necessità di mantenere sincronizzate le copie delle entità. Da questo si può osservare che gli scenari in cui può essere conveniente usare la tolleranza ai guasti fornita da GAIAFT sono quelli che prevedono un numero di LP strettamente maggiore rispetto al numero di copie di entità necessarie.

Infine, anche se si tratta di un caso al di fuori delle specifiche, è interessante esaminare il caso in cui i crash sono N, e il numero di copie è minore o uguale a N (cioè insufficiente per il funzionamento del sistema descritto precedentemente). In questo caso, anche se non si può più avere la certezza matematica del corretto svolgimento della simulazione, esiste comunque una probabilità non nulla che la simulazione venga eseguita correttamente. Questa probabilità, dipendente dai numeri di LP, crash, entità e copie, viene trattata nel capitolo 4, *Analisi probabilistica della tolleranza ai crash.*

3.1.2 Tolleranza ai guasti bizantini

I guasti bizantini sono i guasti più generici, e per questo anche i più problematici, visto che i problemi causati da questi guasti sono molto difficili da identificare.

Un guasto bizantino infatti è un qualsiasi tipo di malfunzionamento avvenuto durante una computazione, che può portare a un qualunque risultato errato.

Più precisamente un guasto bizantino potrebbe dare luogo a uno di questi scenari:

- la cessazione del funzionamento di un componente del sistema distribuito, che quindi smette di comunicare (ad esempio il crash di un LP);

- l'invio di dati errati da parte di un componente del sistema distribuito (ad esempio l'invio di messaggi corrotti da parte di tutte le entità di un LP o di un sottoinsieme di entità di un LP);
- errori nell'elaborazione dei dati da parte di un LP, che portano a risultati errati e incoerenti.

Nel primo caso (i crash) sono sufficienti $N + 1$ copie per ogni entità, per permettere al sistema di sopravvivere senza perdita di dati a N crash (sez. 3.1.1).

Negli altri casi invece la situazione è più complessa, perché possono verificarsi due possibilità: nel migliore dei casi i dati e i messaggi errati possono essere riconosciuti a causa di incoerenze con il contesto in cui si trovano (ad esempio un numero negativo come risultato di un quadrato), ma nel caso peggiore invece i dati e i messaggi errati potrebbero essere completamente verosimili, rendendo impossibile stabilire se si tratta di messaggi corretti o errati.

Per risolvere quest'ultimo caso è quindi necessario usare il metodo della maggioranza, che comporta un'ulteriore ridondanza delle entità.

Più precisamente, se la tolleranza desiderata è a N guasti bizantini, per ogni entità devono esistere almeno $2N + 1$ copie. A ogni interazione tra entità si confrontano i messaggi ricevuti da tutte le copie, e potendosi essere verificati per definizione al massimo N guasti bizantini, i restanti $N + 1$ messaggi devono essere corretti e uguali tra loro.

In questo modo il sistema riesce a resistere anche al caso pessimo, cioè quello in cui avvengono N guasti bizantini in altrettanti componenti del sistema distribuito, e questi N componenti si comportano tutti nello stesso modo, inviando tutti lo stesso messaggio errato ma verosimile. In questo caso infatti, un componente che riceve in tutto $2N + 1$ messaggi, ne avrebbe $N + 1$ di un tipo, e N di un altro tipo. Non avendo nessun altro modo per distinguere i messaggi corretti da quelli errati, il componente capirebbe comunque che quelli corretti sono gli $N + 1$, perché da specifiche non possono avvenire più di N guasti bizantini (fig. 3.5).

Analogamente al caso della tolleranza ai crash, anche nel caso di tolleranza ai guasti bizantini è presente il vincolo per cui le copie di una stessa entità devono obbligatoriamente trovarsi su LP distinti.

Il primo motivo della presenza di questo vincolo è che i crash sono un sottoinsieme dei guasti bizantini, pertanto in un sistema di tolleranza a guasti bizantini deve essere considerata l'eventualità di crash agli LP, che interromperebbero l'attività di tutte le entità gestite da tali LP, riconducendo tutto al caso della tolleranza ai crash (sez. 3.1.1).

Il secondo motivo della presenza di questo vincolo è che viene contata come un singolo guasto bizantino sia la corruzione di una sola entità, sia la corruzione di tutte le entità gestite da un singolo LP. Pertanto, se due copie di una stessa entità si trovassero sullo stesso LP corrotto, potrebbero produrre identici risultati corrotti, rendendo vano il sistema a maggioranza descritto precedentemente. Questo perché, dal punto di vista del controllore della maggioranza, un singolo guasto bizantino che colpisse tutte le entità di un LP, conterebbe come il verificarsi di più guasti bizantini.

Analogamente al caso della tolleranza ai crash, il vincolo per cui le copie di una stessa entità devono obbligatoriamente trovarsi su LP distinti determina il vincolo per cui il numero minimo di LP deve essere sempre maggiore o uguale al numero di copie necessarie (nel caso di tolleranza a N guasti bizantini quindi la simulazione deve essere distribuita su almeno $2N + 1$ LP).

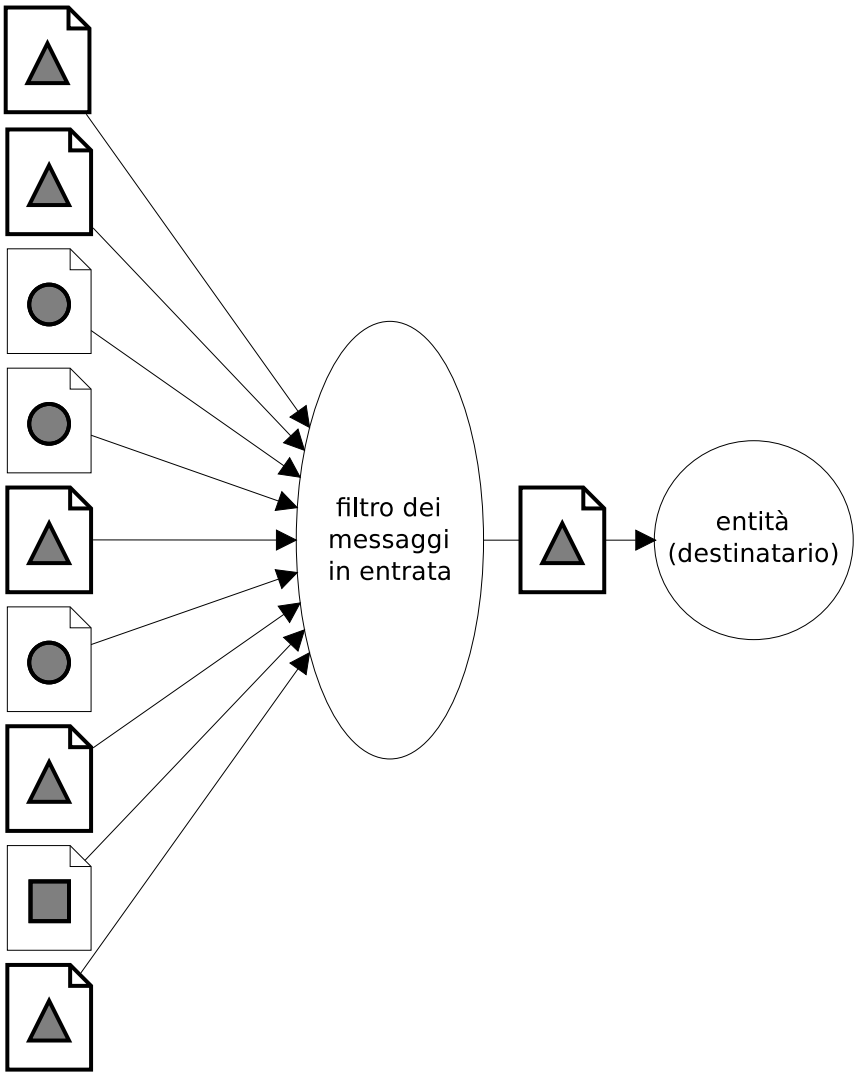

Figura 3.5: Ricezione dei messaggi basata sulla maggioranza assoluta. Nella figura il numero massimo di guasti bizantini tollerati è 4, e quindi ogni entità riceve un totale di 9 copie dello stesso messaggio (9 copie perché il numero minimo necessario di copie è $(2 * 4) + 1$). Se i messaggi differiscono tra loro, viene scelto quello di cui è presente la maggioranza assoluta delle copie, cioè la metà più uno del totale (almeno 5 copie). Il messaggio di cui esistono almeno 5 copie uguali è obbligatoriamente quello corretto, perché da specifica il massimo numero di guasti bizantini tollerati è 4, e quindi non si considera il caso in cui possano esserci 5 o più messaggi errati uguali tra loro.

3.2 Funzionalità introdotte da GAIAFaultTolerance

Come detto precedentemente, GAIAFaultTolerance introduce nuove funzionalità per realizzare il modello di tolleranza ai guasti descritto nelle sezioni precedenti. La maggior parte di queste funzionalità rimane comunque nascosta agli occhi dell'utente, che vede una simulazione basata su GAIAFT del tutto analoga ad un'altra basata su GAIA.
Alcune delle decisioni progettuali effettuate, infatti sono state prese proprio per garantire questa compatibilità tra l'utilizzo di GAIA e quello di GAIAFT.

3.2.1 Registrazione delle entità

La registrazione delle entità è notevolmente più complessa rispetto alla controparte di GAIA. In GAIA la registrazione può avvenire localmente all'interno di ogni LP, e non richiede comunicazione tra gli LP.
In GAIAFT invece, come spiegato in sez. 3.1, la registrazione di un'entità deve comportare automaticamente la registrazione di tutte le sue copie. Le copie di un'entità però devono trovarsi necessariamente tutte in LP diversi, e questo rende necessarie delle registrazioni sincronizzate tra più LP.

Una caratteristica di GAIA che è stata sfruttata consiste nella necessità di effettuare tutte le registrazioni prima dell'inizio della simulazione, in fase di inizializzazione del simulatore. L'impossibilità di avere richieste di registrazione durante lo svolgimento della simulazione ha reso possibile quindi relegare tutte le operazioni di sincronizzazione delle registrazioni in fase iniziale.

L'approccio scelto per realizzare questa sincronizzazione delle registrazioni è quello di utilizzare, durante la sola fase di inizializzazione, un server esterno che esegue le seguenti operazioni:

- stabilisce una connessione con ogni LP;
- raccoglie tutte le richieste di registrazione delle entità[2] dei vari LP;

[2]Le richieste di registrazione che vengono raccolte sono quelle delle entitàFT, cioè le entità tolleranti ai guasti che il layer GAIAFT rende visibili ai layer superiori. In questa fase non viene effettuata nessuna reale registrazione di entità nel layer sottostante GAIA.

- avendo a disposizione l'elenco completo delle entità registrate nel modello di simulazione, genera, secondo il modello di ridondanza scelto, delle liste di entità GAIA che gli LP devono registrare;

- invia a tutti gli LP le specifiche liste di entità GAIA che devono registrare, e questi eseguono le effettive registrazioni nel layer GAIA;

- raccoglie tutti i dati delle entità GAIA registrate da ogni LP, li unisce in un unico elenco, e infine invia questo elenco a tutti gli LP, in modo che durante la simulazione questi ultimi possano rintracciare le copie GAIA delle varie entità GAIAFT.

3.2.2 Generazione di numeri pseudocasuali

In ogni simulatore le entità necessitano di un generatore di numeri pseudocasuali, per prendere decisioni sullo svolgimento delle proprie attività durante la simulazione.

In GAIAFT però per ogni entità vista dall'utente ci sono N copie nel layer sottostante che devono comportarsi tutte nello stesso modo, prendendo sempre le stesse decisioni ed eseguendo sempre le stesse operazioni.

Questo genera un problema, perché se le copie di un'entità generassero in LP differenti dei numeri pseudocasuali per prendere una decisione, questi numeri sarebbero probabilmente diversi, essendo basati sui differenti semi random globali degli LP, e quindi darebbero luogo a differenti comportamenti e decisioni.

Per risolvere questo problema l'approccio scelto è stato quello di fornire a ogni entità di GAIAFT un seme random personale, che viene condiviso da tutte le copie e solo da queste[3]. In questo modo, visto che le copie di un'entità eseguono sempre tutte le stesse operazioni, e che nessun altro al di fuori delle copie di quell'entità può accedere al seme random personale, il risultato è che tutte le copie generano la stessa sequenza di numeri pseudocasuali, comportandosi quindi sempre in modo concorde tra loro e prendendo sempre le stesse decisioni.

[3]I semi random personali di ogni entità vengono distribuiti a tutti gli LP in fase di inizializzazione, durante la sincronizzazione delle registrazioni descritta in sez. 3.2.1.

3.2.3 Spedizione dei messaggi

Per realizzare il modello di tolleranza ai guasti descritto precedentemente, una copia dell'entità destinatario di un messaggio deve ricevere una copia del messaggio da ogni copia dell'entità mittente.

Al tempo stesso però, perché tutte le copie dell'entità destinatario ricevano il messaggio, lo stesso messaggio deve essere inviato da ogni copia del mittente a tutte le copie del destinatario.

Questo significa che ognuna delle copie del mittente deve inviare il messaggio a tutte le copie del destinatario, e di conseguenza il numero di copie di un singolo messaggio che transitano è il quadrato del numero di copie previste per ogni entità (fig. 3.6).

Più precisamente, il fattore di moltiplicazione del numero di messaggi che transitano in una simulazione GAIAFT rispetto al numero di messaggi che transitano in una simulazione GAIA è dato dalle seguenti espressioni, rispettivamente per i casi di tolleranza ai crash e tolleranza ai guasti bizantini.

Nel caso di tolleranza a N crash (che richiede quindi $N + 1$ copie per ogni entità) e di una simulazione GAIA che richiede in totale NUM_MSGS messaggi in transito, il numero di messaggi in transito per la stessa simulazione basata su GAIAFT è dato dalla seguente espressione:

$$NUM\_MSGS * (N + 1)^2$$

Nel caso di tolleranza a N guasti bizantini (che richiede quindi $2N + 1$ copie per ogni entità) e di una simulazione GAIA che richiede in totale NUM_MSGS messaggi in transito, il numero di messaggi in transito per la stessa simulazione basata su GAIAFT è dato dalla seguente espressione:

$$NUM\_MSGS * (2N + 1)^2$$

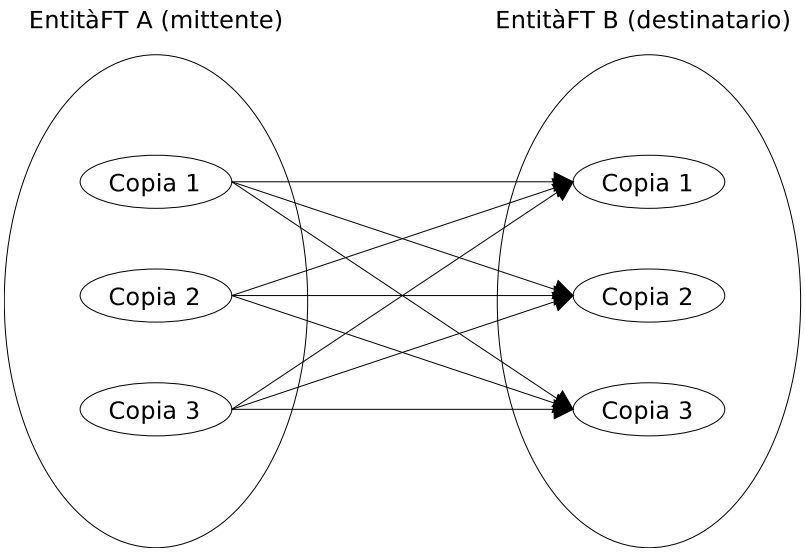

Figura 3.6: Copie di un singolo messaggio che vengono spedite da tutte le copie di una certa entitàFT mittente a tutte le copie di una certa entitàFT destinatario. Si può notare che il numero di messaggi in transito nell'intero sistema è il quadrato del numero di copie di ogni entità.

3.2.4 Ricezione dei messaggi

In seguito all'introduzione della ridondanza sui messaggi descritta in sez. 3.1, è necessario filtrare i messaggi in arrivo per eliminare le copie ridondanti.

Più precisamente, per ogni messaggio che transita tra due entità in più copie, è necessario selezionare solo una delle copie che vengono ricevute dal destinatario.

A seconda del tipo di guasti che si vogliono tollerare, i metodi per filtrare i messaggi in arrivo sono differenti.

Tolleranza ai crash

Nel caso della tolleranza ai crash (descritta in sez. 3.1.1), ciò che si vuole ottenere è che almeno una copia del messaggio arrivi a destinazione. Il filtro per i messaggi ricevuti in questo caso è molto semplice: consiste nel restituire al livello superiore la prima copia del messaggio ricevuta, e scartare tutte le successive. Viene scelta la prima copia e non una

delle successive per motivi prestazionali: in questo modo, il tempo totale di ricezione del messaggio corrisponde con il tempo di ricezione della copia più veloce.

Tolleranza ai guasti bizantini

Nel caso della tolleranza ai guasti bizantini (descritta in sez. 3.1.2), ciò che si vuole ottenere è che arrivi almeno una maggioranza assoluta[4] di messaggi uguali tra loro.

Il filtro per i messaggi ricevuti in questo caso è più complesso. Consiste infatti nell'aspettare una maggioranza assoluta di messaggi uguali tra loro e di restituire al livello superiore la prima copia del messaggio che soddisfa questa proprietà, scartando tutte le copie precedenti e successive.

In questa procedura di filtro si possono individuare tre fasi. La prima fase è quella in cui non è ancora stata raggiunta la maggioranza assoluta di un certo messaggio, e quindi i messaggi che arrivano vengono memorizzati e catalogati, allo scopo di stabilire quante copie di un certo messaggio sono arrivate. In questa fase potrebbero arrivare messaggi corretti (fig. 3.7) oppure messaggi corrotti dai guasti bizantini (fig. 3.8). In entrambi i casi i messaggi vengono confrontati con quelli già ricevuti e vengono conteggiati.

La seconda fase (fig. 3.9) è quella in cui arriva l'$N + 1$esima copia di un certo messaggio. Questa copia è la prima che determina il raggiungimento della maggioranza assoluta, e quindi da specifica è sicuramente corretta, perché al massimo possono esistere N copie di un ipotetico messaggio errato. Dopo l'analisi che confronta la copia con le altre e la conteggia, la copia viene restituita al livello superiore come messaggio sicuramente valido.

La terza fase (fig. 3.10) è quella in cui il messaggio sicuramente valido è già stato restituito al livello superiore, e quindi i messaggi arrivati successivamente non sono più utili, anche se corretti, e vengono pertanto scartati.

[4]La maggioranza assoluta consiste nella metà più uno del totale.

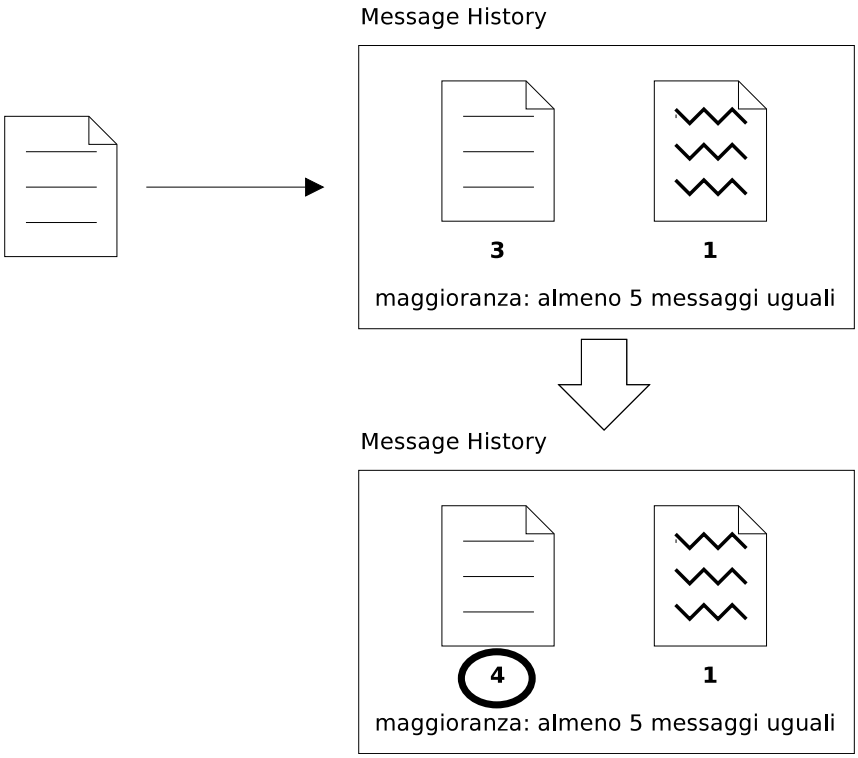

Figura 3.7: Ricezione di un messaggio corretto che non determina il raggiungimento della maggioranza assoluta (in questo caso 5 messaggi). Il messaggio viene confrontato con quelli già ricevuti e conteggiato.

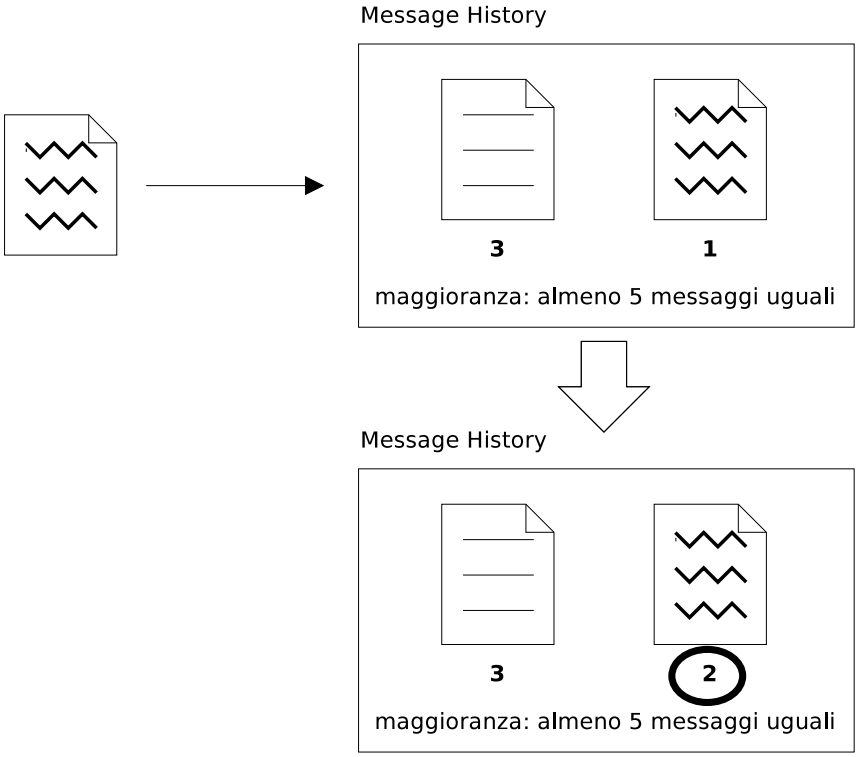

Figura 3.8: Ricezione di un messaggio corrotto a causa di un guasto bizantino. Il messaggio viene confrontato con quelli già ricevuti e conteggiato.

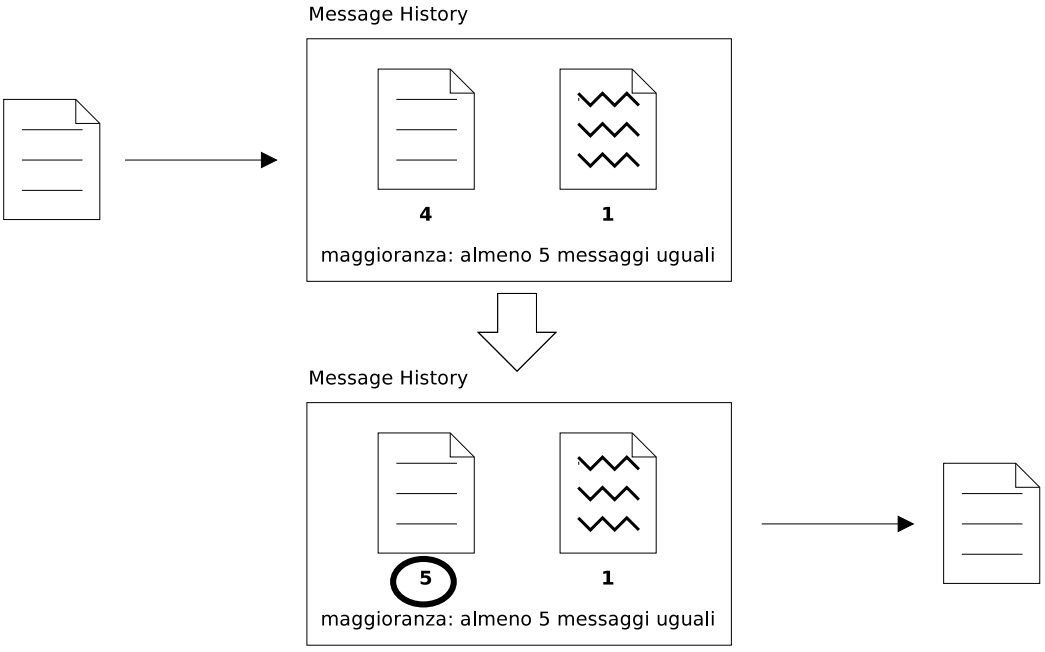

Figura 3.9: Ricezione del messaggio corretto che determina il raggiungimento della maggioranza assoluta (in questo caso 5 messaggi). Il messaggio viene confrontato con quelli già ricevuti, conteggiato e restituito al livello superiore come primo messaggio sicuramente corretto.

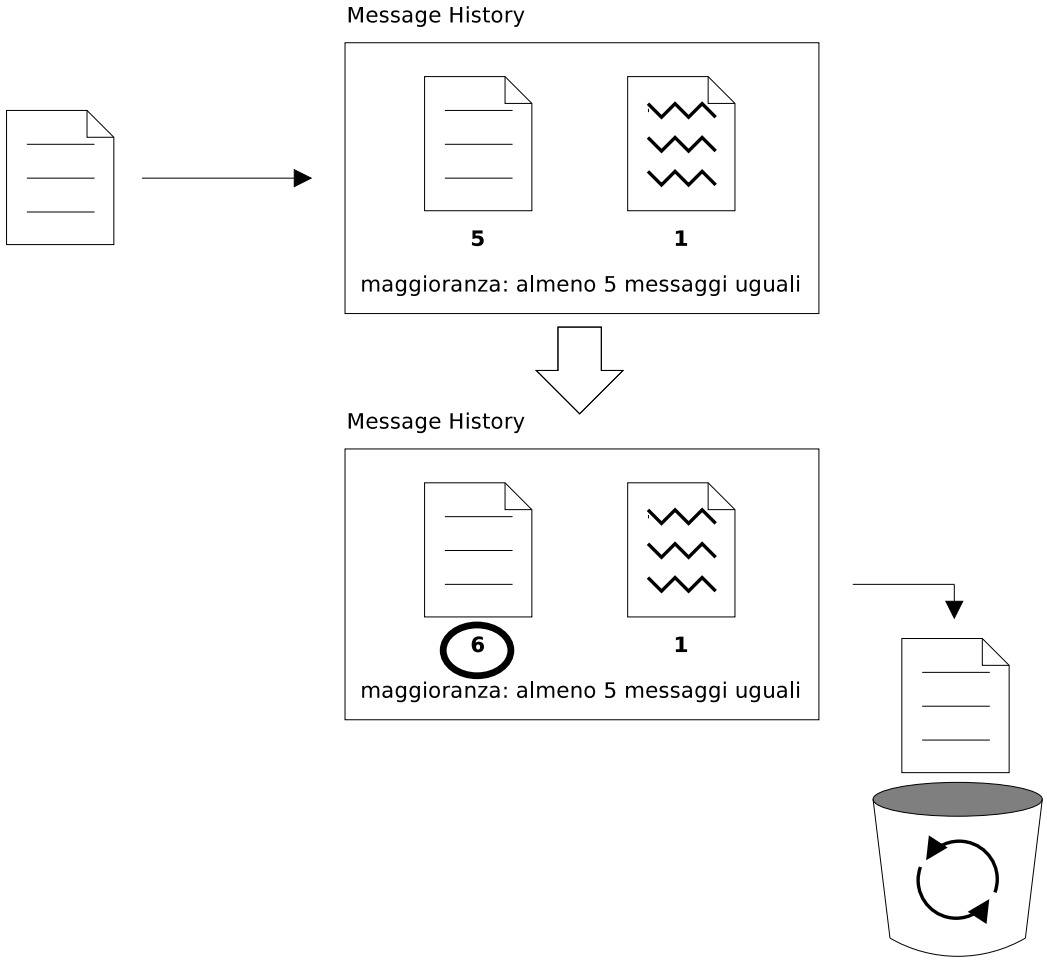

Figura 3.10: Ricezione di un messaggio arrivato dopo la ricezione dell'$N + 1$esimo messaggio corretto. Il messaggio non serve più al raggiungimento della maggioranza assoluta e quindi viene scartato.

3.2.5 Migrazioni di entità

Una delle principali caratteristiche dell'ambiente di simulazione GAIA è quella di permettere alle singole entità di migrare da un LP all'altro, ottimizzando il traffico di rete e rendendo la simulazione adattabile alle condizioni della rete (sez. 2.2). Queste migrazioni infatti vengono eseguite allo scopo di raccogliere nello stesso LP o in LP vicini tra loro[5] quelle entità che comunicano tra loro più spesso, in modo da veicolare il maggior numero possibile di messaggi tramite memoria condivisa e reti a bassa latenza, ottenendo di conseguenza un abbassamento della latenza media dei messaggi e una simulazione più performante (sez. 2.1).

Questa caratteristica però si scontra con un aspetto essenziale di GAIAFT: il vincolo per cui due o più copie di una stessa entità non possono mai trovarsi sullo stesso LP (sez. 3.1). Infatti, anche se in fase di registrazione le copie delle entità vengono distribuite sugli LP in modo che questo vincolo sia rispettato, se le migrazioni sono attivate non si può escludere che in seguito due o più copie di una stessa entità migrino nello stesso LP violando il vincolo sopra citato, essenziale per il corretto funzionamento di GAIAFT.

Questo problema è stato risolto con una piccola modifica nel framework GAIA. Prima della modifica, quando GAIA decideva che una certa entità dovesse migrare, questa decisione era irrevocabile. Dopo questa modifica invece GAIA per mezzo della funzione `GAIA_MigrableHandler`[6] chiede a GAIAFT se l'entità può migrare o no in un determinato LP, e GAIAFT controlla se su quel LP è presente una copia di quell'entità[7], rispondendo affermativamente o negativamente di conseguenza.

[5] LP vicini in termini di bassa latenza. Due LP possono essere considerati vicini sia perché si trovano sulla stessa PEU (Physical Execution Unit), e quindi possono comunicare tra di loro molto velocemente tramite memoria condivisa, sia perché si trovano in PEU vicine geograficamente o connesse tramite reti a bassa latenza.

[6] `GAIA_MigrableHandler` è una funzione definita all'interno di GAIAFT, che viene però richiamata da GAIA quando quest'ultima ha bisogno di sapere se una certa entità può migrare o no.

[7] Il controllo viene fatto da parte di GAIAFT per mezzo dell'apposita funzione `GAIAFT_CanIMigrateEntityOnLp`.

Evitare le collisioni

L'approccio scelto per risolvere il problema sopra descritto ha introdotto un'altra problematica: ci sono casi in cui GAIA decide che due copie di una stessa entità devono migrare durante lo stesso timestep. Questo può generare collisioni tra copie della stessa entità sullo stesso LP per questi due motivi:

- il controllo effettuato da GAIAFT per decidere se una migrazione può essere effettuata o no viene eseguito in base alle entità presenti su un certo LP, al momento della chiamata della funzione da parte di GAIA;

- da quando viene eseguito il controllo, un'entità impiega due timestep per completare la propria migrazione, e solo dopo quei due timestep viene considerata appartenente al LP di destinazione da GAIAFT.

A causa di questi due motivi, se due copie della stessa entità migrano su un certo LP durante lo stesso timestep (o con un solo timestep di differenza), GAIAFT concede ad entrambe il permesso di migrare, perché al momento del controllo non vede nessuna copia dell'entità su tale LP. A questo punto quindi la procedura di migrazione viene portata a buon fine per entrambe le copie, con il risultato di farle coesistere all'interno dello stesso LP, violando quindi il vincolo fondamentale di GAIAFT.

Per evitare questo problema è stato introdotto un sistema di permessi a slot temporali. Questo sistema prevede che il tempo simulato venga suddiviso in slot temporali ampi due timestep l'uno. Da questa ampiezza degli slot deriva il fatto che se due copie di una stessa entità vogliono migrare all'interno di due slot differenti, si può essere sicuri che non avverrà nessuna collisione.

A questo punto, per evitare che venga concesso il permesso a migrare a due copie della stessa entità all'interno dello stesso slot, è stato usato il seguente metodo:

- per ogni singola entità, ogni sua copia ha un numero di copia che la distingue dalle altre (prima copia, seconda copia, terza copia, ecc.);

- ad ogni slot temporale viene associato un numero di copia;

- viene concesso a una copia il permesso a migrare solo nel caso in cui il numero di copia sia lo stesso associato allo slot corrente (in altre parole, nello slot n possono migrare solo le prime copie, nello slot $n+1$ possono migrare solo le seconde copie, nello slot $n+2$ possono migrare solo le terze copie, ecc.);
- viene concesso il permesso a migrare solo nel primo dei due timestep che costituiscono lo slot, in modo che una migrazione abbia il tempo per completarsi senza poter generare una collisione con eventuali migrazioni che avverrano nello slot successivo.

Capitolo 4

Analisi probabilistica della tolleranza ai crash

Nella sezione 3.1.1 del capitolo precedente viene descritto il sistema per la tolleranza ai crash ideato per lo sviluppo di GAIAFaultTolerance (GAIAFT). Questo sistema prevede la presenza nelle simulazioni di copie di ogni entità, e se il numero di copie è N allora il sistema può resistere con sicurezza a $N - 1$ crash.

Questo avviene grazie al vincolo richiesto da GAIAFT (d'ora in poi *vincolo GAIAFT*), per cui le copie di una stessa entità non devono mai trovarsi nello stesso LP.

In questo capitolo verranno analizzati i benefici apportati dall'introduzione di questo vincolo. Verrà inoltre analizzato probabilisticamente un caso al di fuori delle specifiche di GAIAFT: il caso in cui avvenga un numero di crash maggiore del massimo numero di crash tollerato dal sistema.

Lo scenario analizzato è l'esecuzione completa di una simulazione tollerante ai crash, ed è caratterizzato dalle seguenti variabili:

- L è il numero totale degli LP;
- X è il numero di LP crashati durante la simulazione ($X \in [0, L]$);
- N è il numero di entità;
- C è il numero di copie per ogni entità ($C \in [1, L]$);

- $C-1$ è il numero di crash tollerati con sicurezza dal simulatore, che per il sistema di tolleranza descritto in sez. 3.1.1 corrisponde al numero di copie C meno 1.

Il primo passo di questa analisi è definire quando una simulazione è da considerarsi eseguita correttamente. Nel sistema di tolleranza ai crash descritto, l'esecuzione corretta di una simulazione prevede che almeno una copia di ogni entità sopravviva a un certo numero di crash avvenuto durante la simulazione. Si può dire quindi che la simulazione viene eseguita correttamente se si verifica l'evento E:

$$E = \text{(esiste almeno una copia viva di ogni entità al termine della simulazione)}$$

Ciò che si vuole trovare è $P(E)$, la probabilità in cui esiste almeno una copia viva di ogni entità al termine della simulazione, che è di fatto la probabilità di esecuzione corretta di una simulazione.

Per sopravvivere fino al termine della simulazione, una copia di un'entità non deve essersi mai trovata su uno degli LP caduti. Per ognuna delle N entità si può definire quindi l'evento F:

$$F_j = \text{(almeno una copia della } j\text{-esima entità non si trovava su un LP caduto)}$$

Si può notare che se si verifica F per tutte le N entità, allora si è verificato E. L'evento E è infatti la congiunzione di tutti gli F_j:

$$E = \bigwedge_{j=1}^{N} F_j$$

A questo punto si può definire l'evento G, che è la negazione di F:

$$G_j = \tilde{F}_j = \text{(tutte le } C \text{ copie della } j\text{-esima entità si trovavano su LP caduti)}$$

L'evento G è però a sua volta la congiunzione di tutti gli eventi H_i, definiti nel modo seguente:

$$H_i = \text{(la } i\text{-esima copia di una certa entità si trovava su un LP caduto)}$$

$$G = \bigwedge_{i=1}^{C} H_i$$

Per l'analisi probabilistica serve calcolare $P(H)$, la probabilità con cui una singola copia di un'entità si trovava su uno degli LP caduti. $P(H)$ può essere calcolata come il rapporto tra il numero di casi favorevoli (X, il numero di LP caduti) e il numero di casi possibili (L, il numero totale degli LP) [31], considerando una distribuzione uniforme delle entità su tutti gli LP:

$$P(H) = \frac{\text{numero casi favorevoli}}{\text{numero casi possibili}} = \frac{X}{L}$$

Avendo trovato che $P(H) = \frac{X}{L}$ è ora possibile calcolare $P(G)$, la probabilità dell'evento congiunzione di tutti gli H_i. Per questo calcolo però si devono analizzare i due casi seguenti.

- Più copie di una stessa entità non si possono mai trovare sullo stesso LP (il vincolo GAIAFT). Questo caso rappresenta il funzionamento reale del sistema di tolleranza ai crash implementato in GAIAFT, e la sua analisi viene trattata subito sotto.

- Più copie di una stessa entità si possono trovare sullo stesso LP (assenza del vincolo GAIAFT). Questo caso rappresenta un ipotetico funzionamento di un sistema di tolleranza ai crash in cui non fosse stato introdotto il vincolo GAIAFT. La sua analisi e un confronto con quella del reale funzionamento di GAIAFT verranno trattate in sez. 4.1.

Se gli eventi H_i sono stocasticamente indipendenti[1], $P(G)$ potrebbe essere calcolata come il prodotto di tutte le $P(H_i)$. Gli eventi H_i però nel caso di presenza del vincolo GAIAFT non sono stocasticamente indipendenti tra loro[2], perché il verificarsi di uno di loro è influenzato dal verificarsi degli altri.
Questo perché grazie al vincolo GAIAFT la presenza di una copia di una certa entità su un certo LP esclude automaticamente la presenza su quel LP di altre copie della stessa entità.

[1]Due eventi sono stocasticamente indipendenti quando il verificarsi di uno non influisce sulla probabilità di verificarsi dell'altro [31].

[2]Al contrario, nel caso di assenza del vincolo GAIAFT trattato in sez. 4.1, gli eventi H_i sono stocasticamente indipendenti, pertanto è possibile calcolare la probabilità dell'evento congiunzione G tramite il prodotto delle probabilità $P(H_i)$.

Nel caso di eventi stocasticamente dipendenti, la probabilità della loro congiunzione ($P(G)$) deve essere calcolata mediante la formula della probabilità condizionata [31]. Questa formula è valida sia per eventi stocasticamente indipendenti, sia per eventi stocasticamente dipendenti, e stabilisce che la probabilità di una congiunzione di due eventi A e B, è uguale al prodotto tra la probabilità di un evento ($P(B)$), e la probabilità dell'altro evento, condizionata al sapere che il primo evento si è verificato ($P(A|B)$):

$$P(A \wedge B) = P(A|B)P(B)$$

In particolare quindi, se si considerano due eventi H_1 e H_2, $P(H_2|H_1)$ è la probabilità del verificarsi di H_2 sapendo che si è verificato H_1. Ma se si è verificato H_1 (con probabilità $P(H_1) = \frac{X}{L}$), allora significa che la copia dell'entità dell'evento H_2 non può trovarsi sul LP in cui si trovava la copia dell'evento H_1. Se si è verificato H_1, questo LP viene infatti escluso dai possibili LP in cui poteva trovarsi la copia dell'evento H_2, perché vi si trovava già la copia dell'evento H_1. La probabilità con cui la copia dell'evento H_2 si trova su un LP caduto di conseguenza va calcolata ignorando il LP caduto di H_1, e considerando quindi come casi favorevoli solo $X - 1$ LP caduti e come casi possibili solo $L - 1$ LP totali.

Seguendo questo ragionamento e usando la formula della probabilità condizionata, è possibile calcolare $P(G)$ nel modo seguente:

$$\begin{aligned}
&P(H_1) = \frac{X}{L} \\
&P(H_1 \wedge H_2) = P(H_2 \mid H_1)\ P(H_1) = \left(\frac{X}{L}\right)\left(\frac{X-1}{L-1}\right) \\
&P(H_1 \wedge H_2 \wedge H_3) = P(H_3 \mid H_1 \wedge H_2)\ P(H_1 \wedge H_2) = \left(\frac{X}{L}\right)\left(\frac{X-1}{L-1}\right)\left(\frac{X-2}{L-2}\right) \\
&\vdots \\
&P\left(\bigwedge_{i=1}^{C} H_i\right) = P\left(H_C \middle| \bigwedge_{i=1}^{C-1} H_i\right) P\left(\bigwedge_{i=1}^{C-1} H_i\right) = \prod_{i=0}^{C-1} \frac{X-i}{L-i} = P(G)
\end{aligned}$$

Ora che è stata calcolata $P(G)$, si può trovare la probabilità di F, l'evento complementare di G:

$$P(F) = P(\tilde{G}) = 1 - P(G) = 1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}$$

Si può inoltre calcolare la probabilità di esecuzione corretta della simulazione $P(E)$. L'evento E infatti è la congiunzione di tutti gli eventi F_j. Questi eventi sono equiprobabili, avendo tutti probabilità $P(F) = 1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}$. Inoltre, riferendosi alle entità della simulazione, gli F_j sono stocasticamente indipendenti, e pertanto è possibile trovare la probabilità della congiunzione calcolando il prodotto delle loro probabilità:

$$P(E) = P\left(\bigwedge_{j=1}^{N} F_j\right) = \prod_{j=1}^{N} P(F_j) = P(F)^N = \left(1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}\right)^N$$

Ora che è stata calcolata la probabilità di esecuzione corretta della simulazione $P(E)$, si possono fare su di essa le seguenti osservazioni.

- Come è intuibile, se il numero di crash è 0, allora $P(E) = 1$:

  $$P(E) = \left(1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}\right)^N = (1-0)^N = 1 \qquad \text{per } X = 0$$

  In questo caso infatti lo 0-esimo membro della produttoria è $\frac{0}{L} = 0$, e quindi annulla tutta la produttoria.

- Estendendo il caso del punto precedente, se il numero di crash avvenuti (X) è minore o uguale al numero di crash tollerati dal sistema ($C-1$), allora $P(E) = 1$. Questo succede perché l'indice della produttoria i assume tutti i valori interi tra 0 e $C-1$, ma se $X \leq C-1$, tra i valori di i compare anche X. Questo significa che nella produttoria è presente il membro $\frac{X-X}{L-X} = 0$, che annulla tutta la produttoria[3]. Di conseguenza $P(E) = (1-0)^N = 1$. È interessante notare che questo fatto dimostra matematicamente la correttezza del sistema di tolleranza a $C-1$ crash implementato in GAIAFT.

[3]In questo caso, essendo presente il vincolo $X \leq C-1 < L$, X è sempre strettamente minore di L. Non può succedere quindi che esista un membro della produttoria nella forma $\frac{L-L}{L-L} = \frac{0}{0}$.

- Un'altra osservazione su un fatto facilmente intuibile è che se tutti gli LP hanno subito un crash ($X = L$), allora la simulazione non può essere completata correttamente ($P(E) = 0$):

$$P(E) = \left(1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}\right)^N = \left(1 - \prod_{i=0}^{C-1} 1\right)^N = (1-1)^N = 0 \qquad \text{per } X = L$$

Le osservazioni appena trattate riguardano due casi limite (nessun LP guasto, tutti gli LP guasti), e il caso all'interno delle specifiche di GAIAFT in cui il numero di crash avvenuti è minore o uguale al numero massimo di crash tollerato.

Nonostante sia al di fuori delle specifiche di GAIAFT, è interessante analizzare anche il caso in cui il numero di crash avvenuti sia strettamente maggiore del numero di crash tollerati da GAIAFT. In questo caso ($X \in [C, L-1]$), la probabilità $P(E)$ di esecuzione corretta della simulazione non è più 1, ma rimane comunque maggiore di 0 (diventa 0 solo nel caso estremo $X = L$ trattato precedentemente).

In questo caso si ha $X \geq C$, e quindi $X - C$ è sempre maggiore o uguale a 0. Grazie a questa proprietà è possibile riscrivere $P(G)$ nel modo seguente:

$$P(G) = \prod_{i=0}^{C-1} \frac{X-i}{L-i} =$$

$$= \left(\frac{X}{L}\right)\left(\frac{X-1}{L-1}\right)\left(\frac{X-2}{L-2}\right)\left(\frac{X-3}{L-3}\right)\cdots\left(\frac{X-(C-1)}{L-(C-1)}\right) =$$

$$= \frac{X(X-1)(X-2)(X-3)\cdots(X-(C-1))}{L\,(L-1)\,(L-2)\,(L-3)\,\cdots\,(L-(C-1))} =$$

$$= \frac{\left(\frac{X!}{(X-C)!}\right)}{\left(\frac{L!}{(L-C)!}\right)} = \left(\frac{X!}{(X-C)!}\right)\left(\frac{(L-C)!}{L!}\right) =$$

$$= \frac{X!\,(L-C)!}{L!\,(X-C)!}$$

Di conseguenza $P(E)$ essendo uguale a $(1-P(G))^N$ può essere espressa dalla seguente funzione, definita per ogni X maggiore o uguale a C:

$$P(E) = \left(1 - \frac{X!\,(L-C)!}{L!\,(X-C)!}\right)^N$$

Ricapitolando, la probabilità di esecuzione corretta di una simulazione $P(E)$ è espressa dalla seguente funzione, il cui grafico è osservabile in fig. 4.1, e può assumere i seguenti valori in base a X, il numero di crash avvenuto nel sistema:

$$P(E) = \left(1 - \prod_{i=0}^{C-1} \frac{X-i}{L-i}\right)^N = \begin{cases} 1 & \text{se } X < C \\ \left(1 - \frac{X!\,(L-C)!}{L!\,(X-C)!}\right)^N & \text{se } X \in [C, L-1] \\ 0 & \text{se } X = L \end{cases}$$

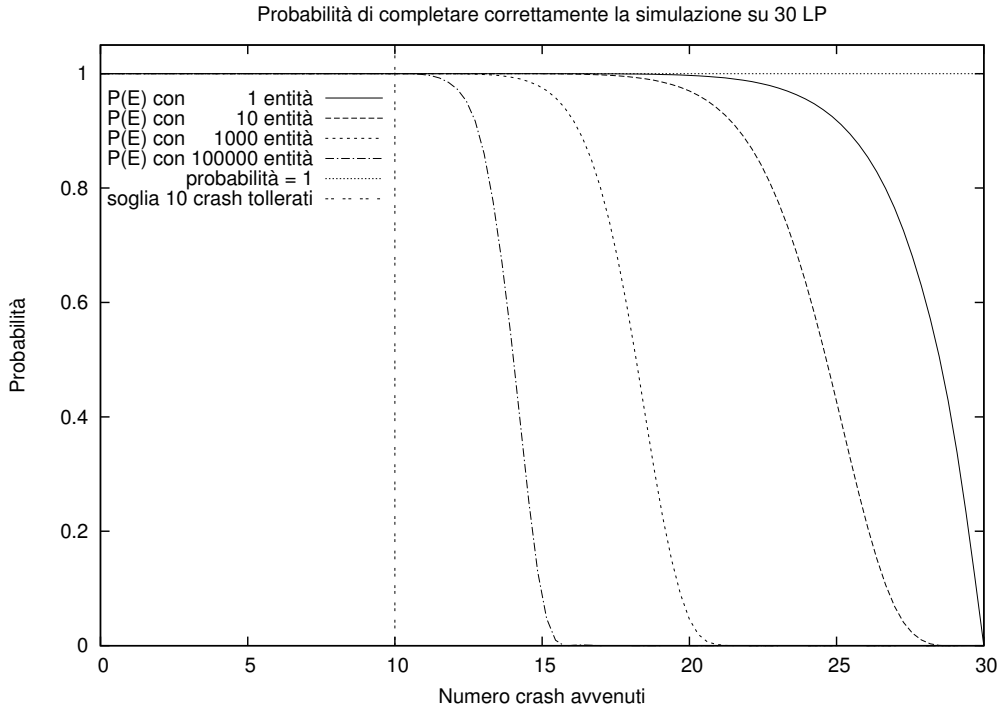

Figura 4.1: Grafici della probabilità $P(E)$ di esecuzione corretta di una simulazione, al variare del numero di crash avvenuti durante la simulazione. Le simulazioni sono distribuite su 30 LP e sono tolleranti a 10 crash (prevedono quindi 11 copie per ogni entità). I quattro grafici si riferiscono a simulazioni da 1, 10, 1000 e 100000 entità.

Su $P(E)$ si possono fare due ulteriori osservazioni.

- Se il numero di crash X è uguale al numero di copie C (cioè è avvenuto un solo crash oltre la soglia dei $C-1$ crash tollerati), $P(E)$ si riconduce alla seguente forma, espressa in funzione del coefficiente binomiale $\binom{L}{C}$:

$$P(E) = \left(1 - \frac{X!\,(L-C)!}{L!\,(X-C)!}\right)^N = \left(1 - \frac{C!\,(L-C)!}{L!}\right)^N = \left(1 - \binom{L}{C}^{-1}\right)^N$$

  per $X = C$

- Per $X \in [C, L-1]$, la probabilità $P(F) = P(\tilde{G}) = 1 - \frac{X!\,(L-C)!}{L!\,(X-C)!}$ è sempre positiva e strettamente minore di 1. Questo significa che in questo intervallo, $P(E) = P(F)^N$

decresce all'aumentare del numero di entità N:

$$X \in [C, L-1] \quad \Longrightarrow \quad 0 < P(F) < 1 \quad \Longrightarrow \quad \lim_{N \to +\infty} P(F)^N = 0$$

In fig. 4.2 è possibile osservare questo andamento decrescente all'aumentare del numero di entità, fino a un massimo di 100000 entità simulate.

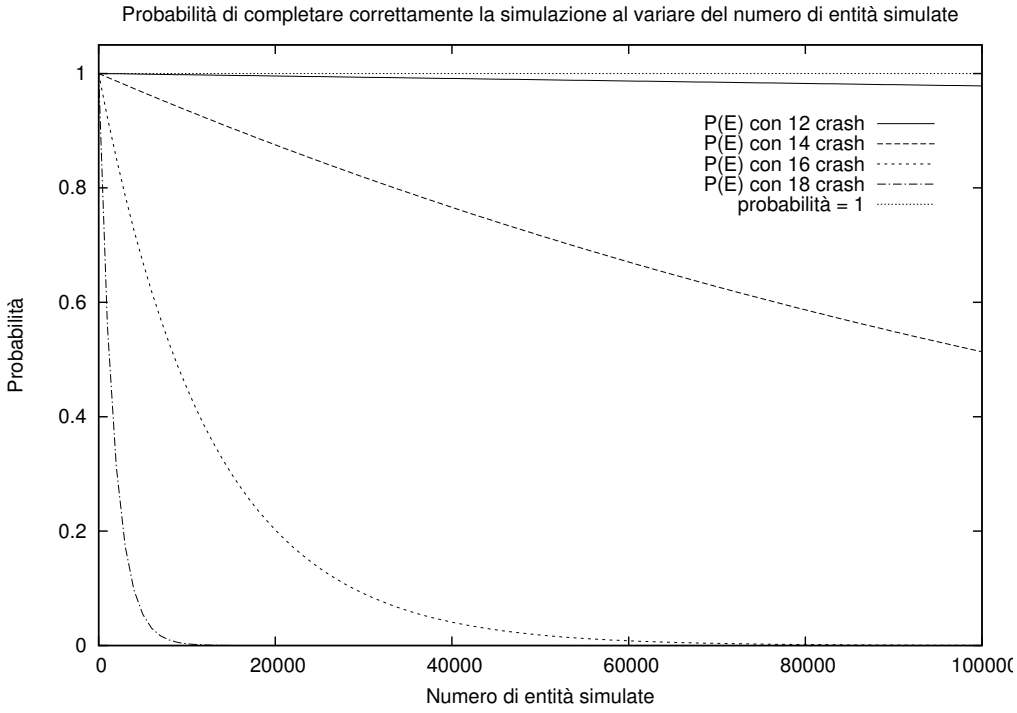

Figura 4.2: Grafici della probabilità di esecuzione corretta di una simulazione $P(E)$, al variare del numero di entità simulate presenti. Le simulazioni sono distribuite su 30 LP e sono tolleranti a 10 crash (prevedono quindi 11 copie per ogni entità). I quattro grafici si riferiscono a simulazioni in cui sono avvenuti 12, 14, 16 e 18 crash.

4.1 Analisi dell'assenza di vincolo GAIAFT

Fino ad ora è sempre stato analizzato il caso corrispondente al reale funzionamento di GAIAFT, in cui più copie di una stessa entità non si possono mai trovare sullo stesso LP a causa della presenza del vincolo GAIAFT.

In questa analisi verrà invece trattato il caso ipotetico in cui il vincolo GAIAFT sia assente, e pertanto sia possibile per più copie di una stessa entità trovarsi sullo stesso LP.

Per l'analisi bisogna riconsiderare le variabili e gli eventi analizzati nella sezione precedente. Le variabili che verranno utilizzate sono le seguenti:

- L = numero totale degli LP;
- X = numero di LP crashati durante la simulazione ($X \in [0, L]$);
- N = numero di entità;
- C = numero di copie per ogni entità [4].

Gli eventi che verranno analizzati sono i seguenti.

- E = (esiste almeno una copia viva di ogni entità al termine della simulazione)
  $E = \bigwedge_{j=1}^{N} F_j$
- F_j = (almeno una copia della j-esima entità non si trovava su un LP caduto)
  $F_j = \tilde{G}_j$
- G_j = (tutte le C copie della j-esima entità si trovavano su LP caduti)
  $G_j = \bigwedge_{i=1}^{C} H_i$
- H_i = (la i-esima copia di una certa entità si trovava su un LP caduto)

Nella sezione precedente è stata calcolata la probabilità $P(H) = \frac{X}{L}$. In seguito però non è stato possibile calcolare la probabilità congiunta di tutti gli H_i con il prodotto delle loro probabilità, a causa della loro non indipendenza stocastica.

In questo caso invece gli H_i sono eventi stocasticamente indipendenti, perché la collocazione di una copia di un'entità su un certo LP, in assenza di vincolo GAIAFT non influenza minimamente la collocazione delle altre copie di quell'entità.

[4] Al contrario del caso analizzato nella sezione precedente, il numero di copie C ora non è limitato dal numero di LP, non essendo presente il vincolo GAIAFT.

Questo rende possibile trovare la probabilità congiunta $P(G)$ calcolando il prodotto di tutte le $P(H_i)$:

$$P(G) = P\left(\bigwedge_{i=1}^{C} H_i\right) = \prod_{i=1}^{C} P(H_i) = \prod_{i=1}^{C} \frac{X}{L} = \left(\frac{X}{L}\right)^C$$

Di conseguenza $P(E)$, essendo uguale a $(1 - P(G))^N$, può essere espressa dalla seguente funzione:

$$P(E) = \left(1 - \left(\frac{X}{L}\right)^C\right)^N$$

Un'osservazione che si può fare su questa funzione è che la certezza assoluta di esecuzione corretta della simulazione ($P(E) = 1$) può essere raggiunta solo introducendo un numero infinito di copie. Con un numero finito di copie la probabilità è sempre strettamente minore di 1, per quanto sia alta in caso di pochi crash. L'unica eccezione è infatti costituita dal caso banale $X = 0$, il caso in cui non avviene nessun crash.

In fig. 4.3 si può osservare il confronto tra la funzione di $P(E)$ appena trovata, che si riferisce al caso di assenza del vincolo GAIAFT, e la funzione di $P(E)$ trovata nella sezione precedente, che si riferisce al caso di presenza del vincolo GAIAFT. Si può osservare che, mantenendo costanti tutti gli altri parametri, l'introduzione del vincolo GAIAFT garantisce un innalzamento della probabilità di esecuzione corretta della simulazione, oltre a garantire con certezza l'esecuzione corretta nel caso in cui avvenga un numero di crash inferiore a un prestabilito numero massimo di crash tollerati.

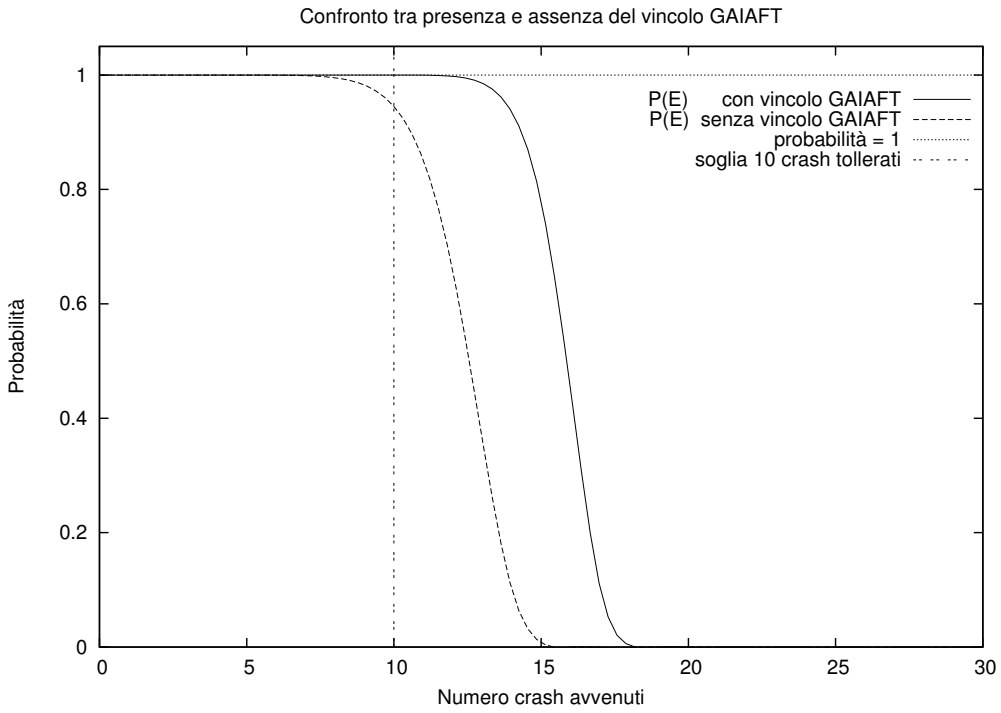

Figura 4.3: Grafici della probabilità $P(E)$ di esecuzione corretta di una simulazione, al variare del numero di crash avvenuti durante una simulazione di 10000 entità, distribuita su 30 LP. I due grafici si riferiscono alla probabilità $P(E)$, nei casi di assenza e di presenza del vincolo GAIAFT. Nel caso di presenza del vincolo GAIAFT, la tolleranza è a 10 crash (sono previste 11 copie di ogni entità in entrambi i casi, ma in assenza del vincolo GAIAFT la tolleranza a 10 crash non può essere garantita).

Capitolo 5

Implementazione di GAIAFaultTolerance

L'implementazione del layer GAIAFaultTolerance (o GAIAFT) è un pacchetto software che si appoggia al layer sottostante GAIA (cap. 2). Più precisamente, GAIAFT mette a disposizione dei layer superiori un'API per realizzare simulazioni tolleranti ai guasti, sfruttando a sua volta l'API di GAIA.

Come già accennato nel capitolo sulla progettazione di GAIAFaultTolerance (cap. 3, sez. 3.1.1, sez. 3.1.2), i guasti tollerati da questo sistema sono i crash (crash hardware, crash software, e interruzione del collegamento di rete) e i guasti bizantini (qualunque comportamento anomalo, tra cui in particolare l'invio di dati errati da parte di entità della simulazione).

Similmente all'API di GAIA, le cui funzioni sono identificate dal nome preceduto dal prefisso *GAIA_*, tutte le funzioni dell'API di GAIAFT sono precedute dal prefisso *GAIAFT_*.

5.1 Compatibilità con GAIA

Il layer GAIAFaultTolerance è stato implementato in modo da minimizzare le differenze tra l'implementazione di una simulazione basata su GAIA e l'implementazione di una simulazione basata su GAIAFaultTolerance.

Per minimizzare le differenze, l'API di GAIAFaultTolerance rispecchia l'API di GAIA, con alcune eccezioni che verranno trattate in seguito. In questo modo l'utente può implementare simulazioni su GAIA o GAIAFT indifferentemente, senza dover acquisire ulteriori conoscenze sull'API di GAIAFT, ad eccezione di alcune funzioni che sono state modificate per esigenze di implementazione o introdotte per aggiungere funzionalità necessarie.

Le funzioni di GAIAFT che presentano differenze nell'header e nell'utilizzo rispetto a quelle dell'API di GAIA sono le seguenti.

`GAIAFT_Initialize:` oltre ai parametri di *GAIA_Initialize*, questa funzione richiede anche i parametri relativi al sistema di tolleranza ai guasti, cioè un'ulteriore porta in ascolto sul SIMA[1], il numero massimo di guasti che deve essere tollerato dal sistema, e infine il tipo di tolleranza ai guasti desiderata (nessuna tolleranza, tolleranza a crash o tolleranza a guasti bizantini; vedere sez. 5.6.3 e sez. 5.6.4).

`GAIAFT_Register:` rispetto alla versione originale di GAIA, questa funzione richiede come ulteriore parametro il seme random da associare all'entità che si vuole registrare. Questo è reso necessario dal fatto che ogni entità in GAIAFT deve avere un suo seme random, che deve essere uguale per tutte le copie dell'entità presenti nel sistema. Questa caratteristica è necessaria per ottenere lo stesso comportamento da tutte copie di un'entità (maggiori approfondimenti in sez. 3.2.2 e sez. 5.4).

Le funzioni non presenti in GAIA che sono state introdotte in GAIAFT sono le seguenti.

`GAIAFT_RegisterStop:` è una funzione che è stata introdotta per segnalare a GAIAFT quando viene terminata la fase di registrazione delle entità, e deve essere chiamata

[1]SImulation MAnager: è uno dei componenti di GAIA, un server di supporto che gestisce le fasi di inizializzazione della simulazione (sez. 2.1.1).

dopo l'ultima chiamata di *GAIAFT_Register*. La sua introduzione è dovuta al fatto che a differenza di GAIA, al termine delle registrazioni delle entità GAIAFT deve effettuare tutte le registrazioni delle copie negli altri LP[2], e per come è strutturato il sistema GAIAFT non ha mezzi per distinguere l'ultima *GAIAFT_Register* dalle precedenti. Con l'introduzione di questa funzione invece GAIAFT è in grado di far partire al momento opportuno le procedure di finalizzazione delle registrazioni, che vengono eseguite proprio durante l'esecuzione della funzione.

`GAIAFT_StartingSend:` per come è stato ideato il sistema di tolleranza a N guasti bizantini, ogni entità aspetta almeno $N+1$ copie di ogni messaggio prima di considerarlo valido. Il primo messaggio che fa partire la simulazione però solitamente proviene da un singolo LP da cui parte la simulazione, quindi arriverebbe all'entità di destinazione una sola copia del messaggio che verrebbe scartata. Per questo è stata introdotta la funzione *GAIAFT_StartingSend*, da usare solo per l'invio del primo messaggio, che semplicemente invia più copie dello stesso messaggio perché queste vengano ricevute correttamente dal destinatario.

`GAIAFT_CanIMigrateEntityOnLp:` questa funzione restituisce un booleano, che dice se è possibile o no far migrare un'entità in un determinato LP, per evitare collisioni tra copie di un'entità che migrano sullo stesso LP (maggiori approfondimenti in sez. 3.2.5).

`GAIAFT_ThereIsEntityCopyOnLp:` questa funzione restituisce un booleano, che dice se una copia di una certa entità è presente in un determinato LP. Questa funzione non ha uno scopo preciso, ma è a disposizione dell'utente nel caso in cui questo abbia bisogno di tale informazione nell'implementazione della propria simulazione.

`GAIAFT_GetEntityList:` restituisce la lista delle entità presenti nell'intera simulazione. Questa funzione è stata introdotta per fornire un metodo specifico per dare all'utente la lista delle entità. Precedentemente, a questo scopo veniva usata la tabella hash globale delle entità, ma questa è inservibile in una simulazione GAIAFT, non facendo distinzione tra le entità e le rispettive copie.

[2]Per la descrizione dettagliata di questa procedura vedere la sez. 5.3.

`GAIAFT_GetIdLocalCopyOfFtEntity:` ad ogni entità della simulazione (provvista di un ID personale) corrispondono varie copie.
La funzione *GAIAFT_GetIdLocalCopyOfFtEntity*, chiamata in un certo LP, restituisce l'ID della copia dell'entità data in input che si trova su quel LP. L'introduzione di questa funzione è stata necessaria per poter fornire al simulatore l'ID dell'entità realmente presente nel LP corrente, durante la fase di validazione di un messaggio ricevuto.

`GAIAFT_GetMyRandSeed:` questa funzione ritorna il puntatore al seme random specifico di una certa entità. L'utilizzo di questa funzione è necessario in quanto ogni entità dispone di un seme random personale, uguale per tutte le copie (sez. 5.4). La funzione è studiata per integrarsi con il generatore di numeri pseudocasuali incluso nel pacchetto *GAIA-ARTÌS*, infatti il valore ritornato dalla funzione può essere direttamente utilizzato come parametro per le funzioni di generazione di numeri pseudocasuali.

`GAIAFT_SetGlobalRandSeedPtr:` nel caso in cui la tolleranza ai guasti venga disabilitata, e di conseguenza non ci siano copie di entità nel sistema e non siano necessari semi random differenti per ogni entità, la funzione vista nel punto precedente *GAIAFT_GetMyRandSeed* restituisce un unico seme random globale per tutte le entità. Questo seme random deve essere inizializzato in fase iniziale usando la funzione *GAIAFT_SetGlobalRandSeedPtr*.

5.2 FTregister

Il Fault Tolerance Register, abbreviato come FTregister, è una complessa struttura dati globale presente in ogni LP, e contiene tutte le informazioni necessarie al funzionamento del layer GAIAFT. Per il corretto funzionamento di GAIAFT, è necessario che tutti gli FTregister contenuti nei vari LP siano uguali tra loro e sincronizzati in caso di modifiche.

L'istanza dell'FTregister che si trova su un determinato LP è composta da alcuni dati statici sul LP e da alcune tabelle hash[3], descritti nell'elenco seguente.

[3]Tabelle hash di tipo *Hashtable* (sez. 5.8.3).

`int id_lp`: è l'ID del LP che contiene l'istanza dell'FTregister.

`int entities_num`: è il numero totale di entità presenti nella simulazione. Per numero di entità si intende il numero di entità previste dall'utente, quindi senza contare le copie introdotte da GAIAFT.

`int copies_number`: è il numero di copie presenti per ogni entità.

`hashtable_t entities_table`: è una tabella hash che contiene i dati di ogni entità. Più precisamente per ogni entità la tabella contiene l'ID di tutte le sue copie, gli LP in cui si trovano le copie, e il numero di copia, un numero che identifica le singole copie di ogni entità[4].

`hashtable_t sub_entities_table`: è una tabella hash che permette, partendo dall'ID di una copia, di trovare velocemente l'ID dell'entità corrispondente e il numero di copia (queste informazioni sono già contenute nella tabella `entities_table`, ma la ricerca dell'informazione desiderata richiederebbe nel caso pessimo tempo $O(n)$, con n la dimensione totale della tabella hash. Si è scelto quindi di aggiungere una tabella che permetta di restituire l'informazione desiderata in tempo $O(1)$, velocizzando notevolmente l'operazione).

`hashtable_t ftentities_priv_info`: è una tabella hash che contiene, per ogni entità, il contatore dei messaggi inviati[5] e il seme random personale di tale entità.

Nonostante la complessa struttura sottostante, l'FTregister è stato implementato in modo da essere visto dall'utente come una scatola chiusa. Il modulo che contiene la struttura dati infatti fornisce la propria API per inserire ed estrarre dati dall'FTregister, e da questa API non traspare nulla dell'implementazione sottostante, in modo che non serva conoscere e considerare la struttura sottostante per poter utilizzare il modulo. L'FTregister quindi dall'esterno deve essere utilizzato come un generico contenitore di dati accessibile solo tramite le funzioni di inserimento ed estrazione fornite dalla sua API.

[4]Ogni entità non dispone di un generico insieme di copie, bensì di una lista gerarchica in cui ogni copia può essere distinta dalle altre per mezzo di un numero di copia (prima copia, seconda copia, terza copia, ecc.). Questa distinzione viene utilizzata all'interno dell'algoritmo che evita le collisioni di entità durante le migrazioni (sez. 3.2.5).

[5]Per la descrizione di cos'è e a cosa serve il contatore dei messaggi inviati, vedere la sez. 5.6.2.

5.3 Register handler

La fase di inizializzazione di una simulazione GAIA prevede da parte di ogni LP la registrazione delle entità che parteciperanno alla simulazione, per mezzo della funzione `GAIA_Register`.

Come spiegato nella sezione precedente (sez. 5.2), ogni LP dispone di una struttura chiamata FTregister che contiene tutti i dati utili per il funzionamento di GAIAFT, e tutti gli FTregister degli LP devono essere uguali tra loro.

Ogni FTregister deve contenere gli ID delle entitàFT e gli ID delle relative copie, ma questi ID vengono assegnati utilizzando la funzione `GAIA_Register`.

Visto che le copie di ogni entità devono trovarsi sempre su LP differenti, per inizializzare gli FTregister è necessario effettuare una serie di operazioni, al termine della quale tutti gli LP hanno registrato localmente tutte le copie delle entitàFT a loro assegnate e dispongono all'interno dei loro FTregister delle informazioni su tutte le entità presenti nel sistema.

Per effettuare le operazioni di inizializzazione si è resa necessaria l'introduzione di un gestore centralizzato che si occupa di raccogliere, elaborare e scambiare i dati tra gli LP durante la fase iniziale. Per non introdurre un nuovo processo, questo gestore è stato implementato come un thread indipendente all'interno di un processo centralizzato già presente in GAIA, il *Simulation Manager* (SIMA, sez. 2.1.1). Essendo ospitato dal processo SIMA, il gestore introdotto da GAIAFT è stato chiamato *SIMA GAIAFT Register Handler* (SIMA GFTRH).

Le operazioni svolte dal SIMA GFTRH consistono nella raccolta delle registrazioni delle entità da parte degli LP, nella preparazione delle liste di copie che andranno registrate dai vari LP, nell'invio delle liste suddette, nella ricezione degli ID delle copie registrate, e infine nell'invio agli LP dell'elenco globale di tutte le copie di entità registrate. Queste operazioni sono articolate nelle seguenti fasi.

Durante la prima fase (fig. 5.1) il SIMA GFTRH si mette in attesa di una singola connessione su un'apposita porta da parte del LP 0. Il LP 0 si connette e invia alcune informazioni preliminari che servono al SIMA GFTRH per effettuare le operazioni successive.

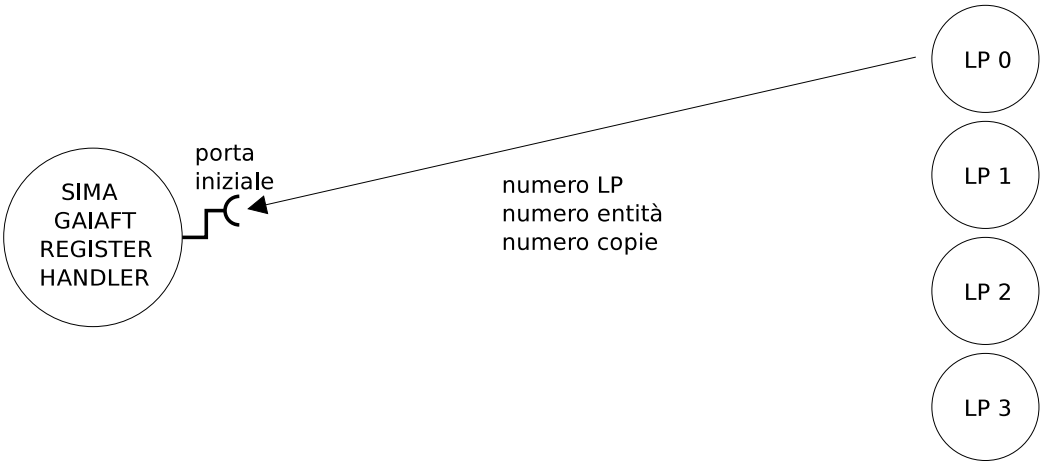

Figura 5.1: Prima fase dell'inizializzazione degli FTregister: LP 0 invia al SIMA GFTRH il numero di LP, il numero di entità e il numero di copie di ogni entità.

A questo punto il SIMA GFTRH apre un'altra porta dedicata alla ricezione degli ID, e tutti gli LP gli inviano il proprio ID, come raffigurato in fig. 5.2.

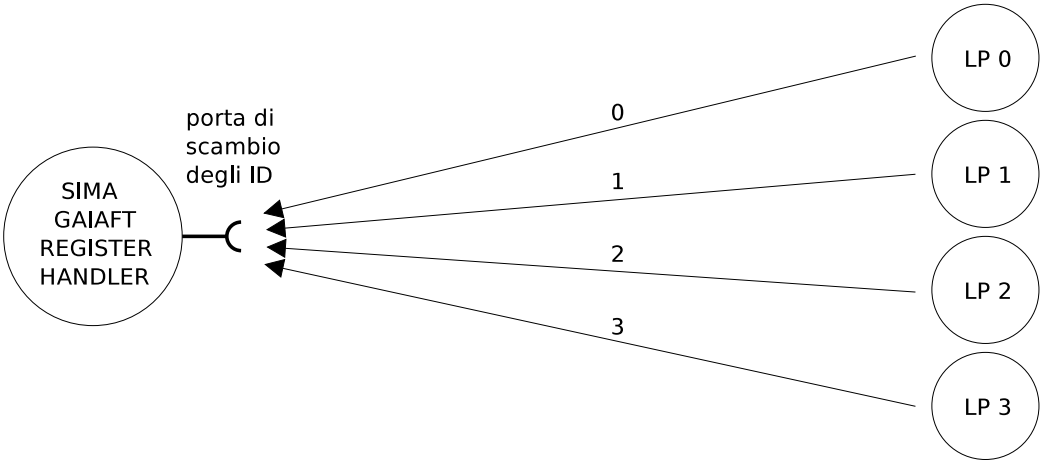

Figura 5.2: Seconda fase dell'inizializzazione degli FTregister: tutti gli LP inviano il proprio ID al SIMA GFTRH, che a sua volta attiva una porta privata per ogni LP.

Nella terza fase emerge un'importante differenza tra GAIA e GAIAFT.

- In GAIA la registrazione delle entità viene effettuata all'interno della funzione `GAIA_Register`.

- In GAIAFT la funzione che sostituisce concettualmente la `GAIA_Register` è la `GAIAFT_Register`, ma all'interno di questa non viene effettuata nessuna registrazione. Le richieste di registrazione effettuate con la `GAIAFT_Register` vengono solo memorizzate dal SIMA GFTRH, e vengono poi effettuate realmente in seguito soltanto all'interno della funzione `GAIAFT_RegisterStop`. Gli scopi di quest'ultima funzione sono quelli di segnalare al SIMA GFTRH che la fase di registrazione delle entità è terminata, di ricevere dal SIMA GFTRH la lista di entità da registrare, di effettuare realmente le registrazioni con la `GAIA_Register`, di inviare al SIMA GFTRH gli ID delle entità restituiti dalla `GAIA_Register`, e infine di ricevere la lista completa di tutte le entità e inserirla nell'FTregister.

In questa fase (fig. 5.3) gli LP eseguono delle registrazioni di entitàFT usando la `GAIAFT_Register`, che invia la richiesta al SIMA GFTRH e si mette in attesa dell'ID da restituire.
Ricevuta la richiesta, il SIMA GFTRH la memorizza localmente, genera un ID per l'entitàFT registrata e invia l'ID al LP, che viene poi ritornato dalla `GAIAFT_Register`. Questa operazione viene ripetuta per ogni entità registrata dagli LP.
Questa fase termina quando tutti gli LP eseguono la funzione `GAIAFT_RegisterStop`, che nel codice della simulazione deve essere chiamata un'unica volta subito dopo all'ultima delle `GAIAFT_Register`. Tutte le fasi successive sono comprese all'interno dell'esecuzione della funzione `GAIAFT_RegisterStop`.

La quarta fase è suddivisa in due sottofasi. La prima coinvolge solo il SIMA GFTRH, che dopo aver raccolto tutte le richieste di registrazione prepara per ogni LP la lista di entità che deve registrare (fig. 5.4). In queste liste, oltre alle entità che sono state registrate direttamente dagli LP, vengono inserite le copie delle entità registrate da altri LP, nella quantità richiesta dalla tolleranza ai guasti desiderata.
Se ad esempio (seguendo l'esempio della fig. 5.4) sono previste tre copie per ogni entità (quindi un'entitàFT A prevede la presenza nel sistema delle copie A, A' ,A"), le entità registrate da LP 0 vengono replicate su LP 1 e su LP 2, come evidenziato in fig. 5.4.

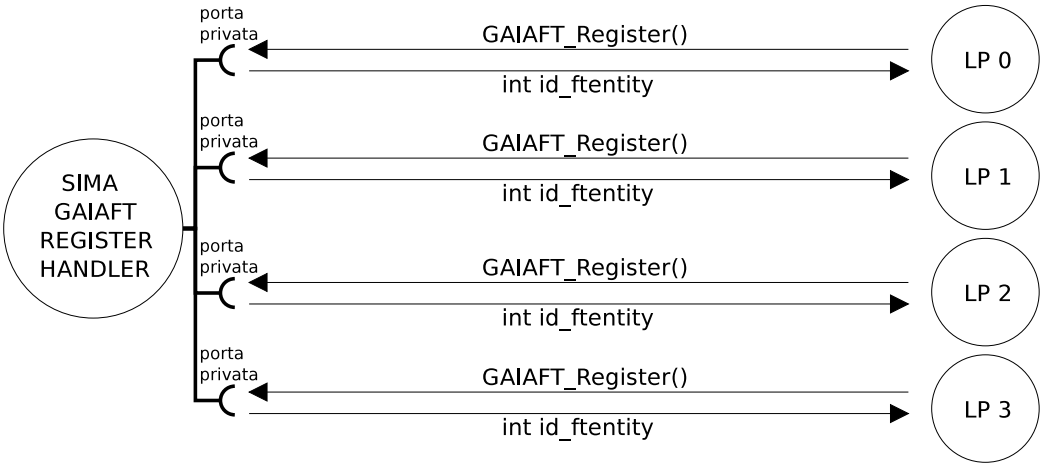

Figura 5.3: Terza fase dell'inizializzazione degli FTregister: quando un LP esegue una `GAIAFT_Register`, la richiesta viene inviata al SIMA GFTRH, che la memorizza localmente e rispedisce al LP l'ID dell'entitàFT registrata.

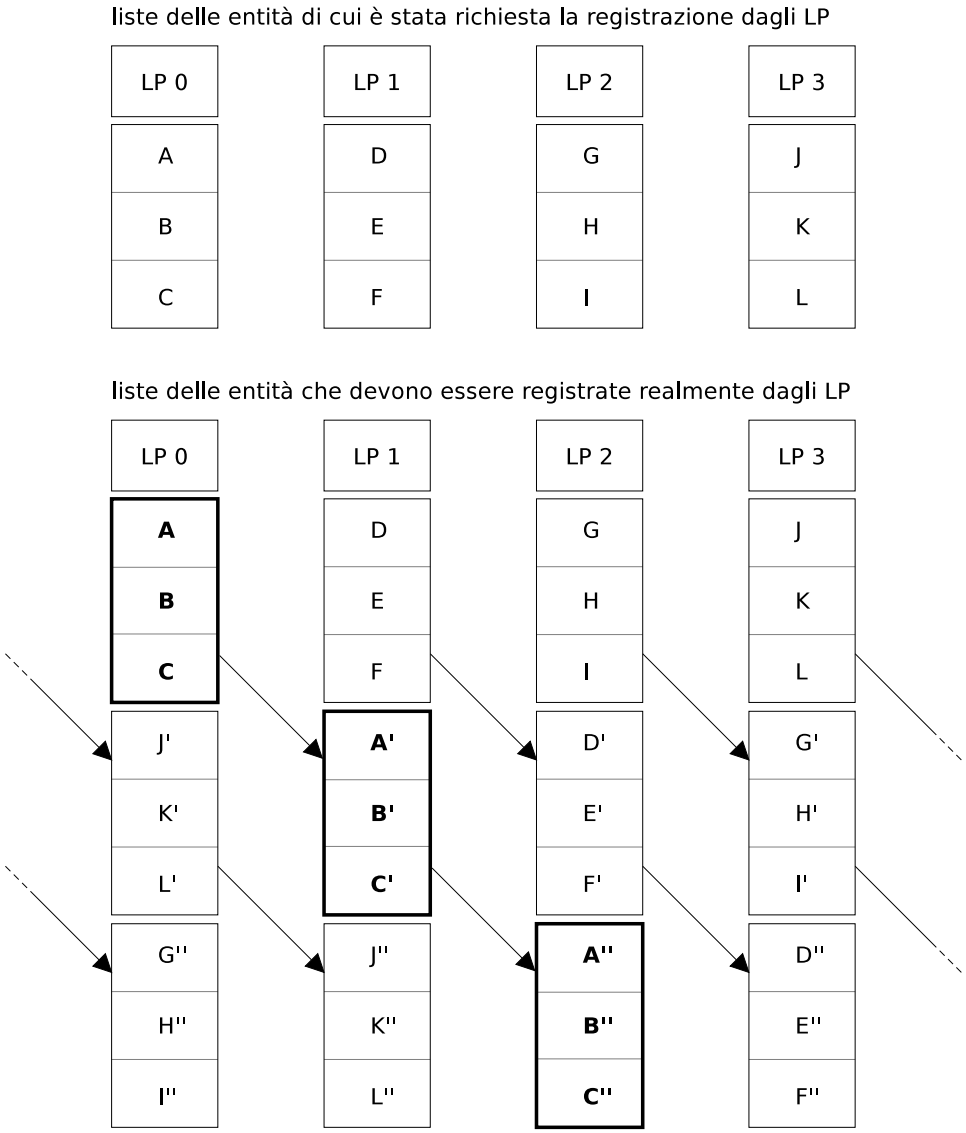

Figura 5.4: Liste di entità preparate dal SIMA GFTRH che devono essere registrate dai vari LP.

La seconda sottofase (fig. 5.5) consiste nella sequenza di operazioni seguente.
Inizialmente il SIMA GFTRH invia agli LP le liste di copie di entità che ognuno di essi deve registrare. Quando un LP riceve la propria lista, registra (tramite la funzione di GAIA `GAIA_Register`) tutte le entità della lista memorizzando gli ID restituiti dalla `GAIA_Register`.
Quando tutte le registrazioni sono state effettuate, gli LP rispediscono al SIMA GFTRH la lista delle entità, comprensiva di tutti gli ID delle copie assegnati dalla `GAIA_Register`.

Figura 5.5: Quarta fase dell'inizializzazione degli FTregister: il SIMA GFTRH invia agli LP le liste di copie di entità che ogni LP deve registrare; gli LP le registrano usando la funzione di GAIA `GAIA_Register`, e inviano al SIMA GFTRH gli ID delle copie registrate restituiti dalla `GAIA_Register`.

Nella quinta e ultima fase (fig. 5.6) infine il SIMA GFTRH unisce in un unico elenco i dati sulle entità ricevuti dagli LP al termine della fase precedente, e rispedisce a tutti gli LP questo elenco globale di tutte le copie di entità presenti nell'intero sistema. Quando gli LP ricevono questo elenco provvedono a inserire nei propri FTregister le informazioni che esso contiene.

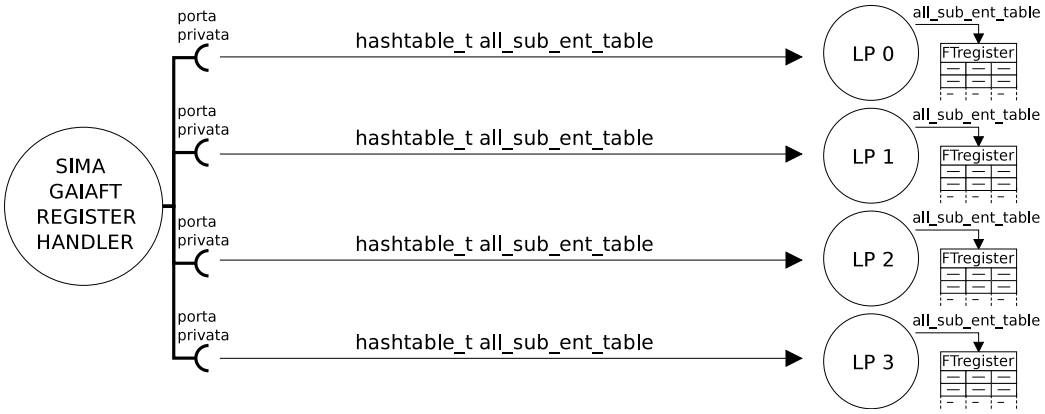

Figura 5.6: Quinta fase dell'inizializzazione degli FTregister: unendo i dati ricevuti dagli LP, il SIMA GFTRH genera un elenco complessivo di tutte le copie di entità presenti nella simulazione, che invia a tutti gli LP; gli LP ricevono questo elenco e inseriscono tutti i dati nel proprio FTregister.

5.4 Semi random personali

Come detto in sez. 3.2.2, in un simulatore le entità necessitano spesso di disporre di un generatore di numeri pseudocasuali. Questo però rappresenta un problema in GAIAFT, perché le copie di una certa entità devono comportarsi tutte nello stesso modo eseguendo sempre le stesse operazioni. Non è più possibile quindi per le entità generare numeri pseudocasuali sulla base di un seme random globale condiviso da tutte le entità di un LP, bensì ogni entità deve disporre di un proprio seme random personale, uguale per tutte le copie.

Questi semi random personali vengono assegnati alle entità dall'utente in fase di registrazione, inserendoli come parametri della funzione `GAIAFT_Register`. In seguito alla fase di inizializzazione descritta nella sezione precedente (sez. 5.3) questi semi vengono memorizzati nell'FTregister, in modo che ogni copia di una determinata entità possa disporre del seme personale dell'entità, che è uguale per essa e per tutte le altre copie di quell'entità.

In questo modo i numeri pseudocasuali generati dalle copie di un'entità sono uguali tra loro, e le copie si comportano quindi in modo concorde, prendendo sempre decisioni

identiche.

Per l'accesso ai semi random personali di ogni entità, GAIAFT fornisce la funzione `GAIAFT_GetMyRandSeed`, che richiede in input l'ID dell'entità e ritorna il puntatore al relativo seme random.
Il valore ritornato è il puntatore al seme perché è proprio questo valore che viene richiesto come parametro dalle funzioni di generazione di numeri pseudocasuali incluse in GAIA. La funzione infatti è stata realizzata così proprio per poter essere utilizzata come parametro per queste funzioni di generazione di numeri pseudocasuali.

Nel caso in cui la tolleranza ai guasti venga disabilitata, e di conseguenza non ci siano copie di entità nel sistema e non siano necessari semi random differenti per ogni entità, la funzione *GAIAFT_GetMyRandSeed* restituisce un unico seme random globale per tutte le entità. Questo seme random deve essere inizializzato in fase iniziale usando l'apposita funzione *GAIAFT_SetGlobalRandSeedPtr*.

5.5 GAIAFT_Send

La funzione `GAIAFT_Send` sostituisce la funzione `GAIA_Send`, e serve per l'invio di un messaggio da un'entità ad un'altra. La `GAIA_Send` originale semplicemente invia un singolo messaggio da un'entità GAIA ad un'altra, mentre la `GAIAFT_Send` invia più messaggi (ognuno di questi inviato usando la `GAIA_Send`) per realizzare il sistema di tolleranza ai guasti.

Più precisamente un'entità di GAIAFT (d'ora in poi chiamata entitàFT per chiarezza) è in realtà rappresentata da un insieme di entità di GAIA (copie dell'entitàFT) che pur trovandosi su LP diversi, dispongono tutte dello stesso stato, eseguono tutte le stesse operazioni, e inviano/ricevono tutte gli stessi messaggi. Per fare in modo che tutte le copie di una entitàFT ricevano i messaggi ad essa destinati, ogni messaggio deve essere inviato da un certo mittente a tutte le copie dell'entitàFT destinatario. Oltre a questo, visto che tutte le copie eseguono le stesse operazioni, quando una di queste invia un messaggio a una certa entitàFT significa che tutte le copie inviano il messaggio, e quindi ogni copia dell'entitàFT riceve tutti i messaggi inviati dalle copie dell'entitàFT mittente, come illustrato in fig. 5.7.

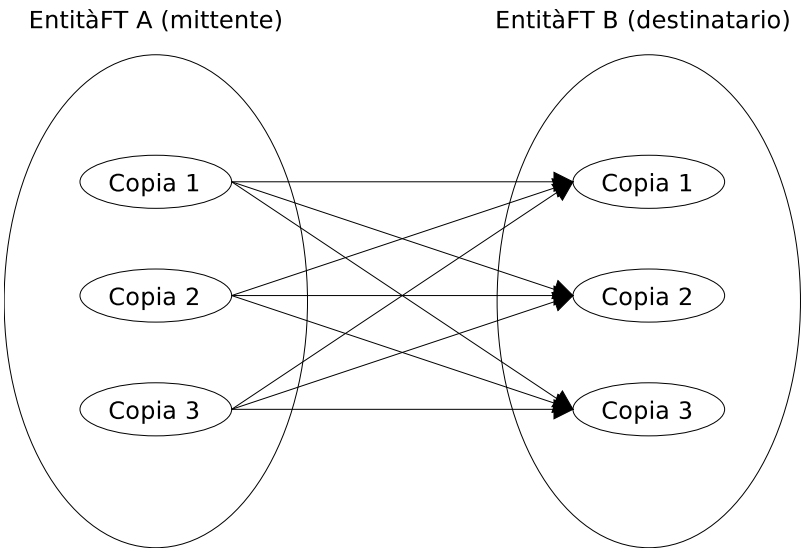

Figura 5.7: Rappresentazione di tutti i messaggi che transitano da un'entitàFT ad un'altra.

Per ottenere il comportamento descritto sopra, la funzione `GAIAFT_Send` è implementata nel modo seguente:

- quando un'entitàFT A[6] vuole inviare un messaggio a un'entitàFT B, la funzione `GAIAFT_Send` richiede all'FTregister (sez. 5.2) la lista delle copie dell'entitàFT B e i relativi identificatori assegnati da GAIA;
- in testa al pacchetto da spedire viene incluso l'*header GFT* (header GAIAFaultTolerance), contenente alcuni dati utili al destinatario per elaborare i messaggi ricevuti, tra cui il contatore del messaggio (descritto in sez. 5.6.2) che viene incrementato di conseguenza, e un codice per il controllo dell'integrità dell'header calcolato tramite l'algoritmo MD5 [28];
- la funzione `GAIA_Send` richiede come parametro l'identificatore GAIA dell'entità mittente, pertanto questa informazione viene richiesta all'FTregister a partire dall'identificatore dell'entitàFT;

[6]Gli identificatori delle entitàFT non sono correlati agli identificatori delle entità del layer sottostante GAIA. Le corrispondenze tra entitàFT ed entità di GAIA sono memorizzate tutte all'interno dell'FTregister.

- infine, ora che la funzione dispone di tutte le informazioni sulle entità del layer sottostante GAIA, e ha aggiunto al corpo del messaggio l'header GFT, per ogni copia di B presente nella lista viene inviata una copia del messaggio.

5.6 GAIAFT_Receive

La funzione `GAIAFT_Receive` sostituisce concettualmente la funzione `GAIA_Receive`, cioè la funzione che serve a ricevere i messaggi diretti alle varie entità della simulazione.

`GAIA_Receive` è una funzione bloccante che viene invocata ciclicamente da ogni LP ad ogni timestep, e ogni volta che un LP riceve un messaggio diretto ad una delle entità che risiedono su di esso, la funzione si sveglia e restituisce il messaggio. L'entità destinatario viene poi identificata dal LP, che provvede quindi a recapitare il messaggio. Il LP si rimette infine in attesa di un nuovo messaggio con un'altra `GAIA_Receive`.
Il comportamento descritto è raffigurato nel diagramma in fig. 5.8.

Al contrario di quanto accade in una simulazione GAIA, con la ridondanza sui messaggi introdotta da GAIAFT (sez. 5.5) per ogni messaggio che viene spedito, al destinatario arriva un certo numero di copie. In caso di assenza di errori questo numero è esattamente il numero di copie dell'entità mittente presenti nella simulazione, in caso di presenza di errori questo numero può essere anche minore.

Visto che in GAIAFT un LP si aspetta di ricevere più copie di ogni messaggio, ma la funzione deve restituire al livello superiore solo un messaggio per ogni insieme di copie, all'algoritmo descritto in fig. 5.8 GAIAFT aggiunge un filtro che permette alla funzione di analizzare i messaggi ricevuti per restituirne solo una copia delle tante che riceve. Più precisamente, la funzione `GAIAFT_Receive` utilizza due differenti algoritmi di ricezione a seconda che la tolleranza desiderata sia ai crash o ai guasti bizantini, algoritmi descritti rispettivamente in sez. 5.6.3 e in sez. 5.6.4.

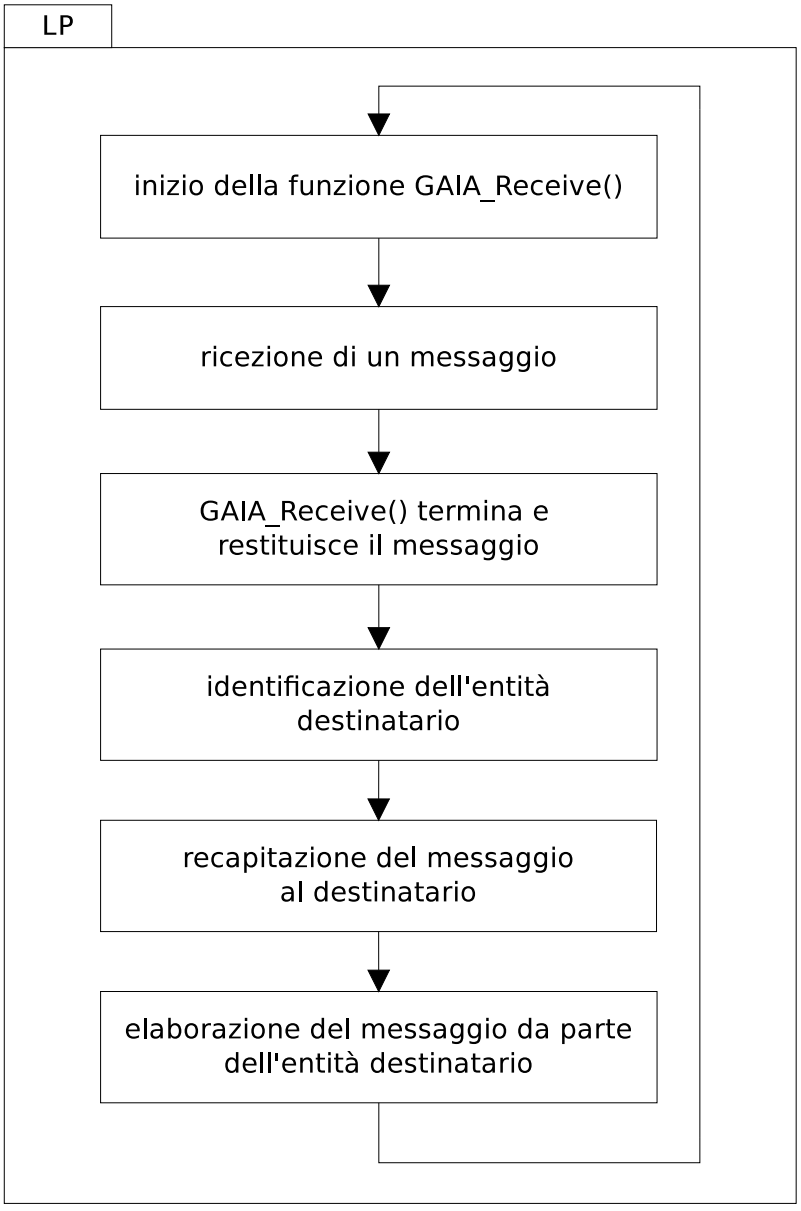

Figura 5.8: Diagramma di flusso di ricezione ed elaborazione dei messaggi da parte di un LP in una simulazione GAIA.

5.6.1 Filtro preliminare MD5

In aggiunta agli algoritmi principali sopra citati, sia per la tolleranza ai crash sia per la tolleranza ai guasti bizantini è presente un filtro preliminare, basato sul controllo dell'integrità degli header dei messaggi ricevuti. Il controllo viene effettuato tramite l'algoritmo MD5 [28], confrontando il codice di controllo ottenuto alla ricezione del messaggio con il codice di controllo incluso nell'header al momento dell'invio (sez. 5.5), e scartando i messaggi il cui confronto ha rilevato codici differenti.

Nel caso della tolleranza ai crash questo filtro preliminare permette al sistema di estendere la tolleranza del sistema anche a messaggi i cui header si sono corrotti durante il trasferimento via rete.

Nel caso della tolleranza ai guasti bizantini questo filtro preliminare serve ad alleggerire l'algoritmo di ricezione tollerante ai guasti bizantini, filtrando i messaggi i cui header si sono corrotti durante il trasferimento via rete, e diminuendo quindi il numero di messaggi ricevuti da analizzare.

5.6.2 Contatore dei messaggi inviati

Essendo presenti nella simulazione N copie di ogni entità che eseguono tutte le stesse operazioni, quando un'entità mittente invia un messaggio, in realtà a livello implementativo sono N copie dell'entità che inviano N copie del messaggio a ognuna delle copie dell'entità destinatario (fig. 5.7). Di conseguenza, ognuna delle copie del destinatario riceverà una serie di messaggi uguali che verranno filtrati opportunamente per ricavarne un unico messaggio fault tolerant.

Per come è strutturato il simulatore GAIA però, è ammesso per un'entità inviare più messaggi all'interno di un singolo timestep, e questi messaggi possono anche essere uguali tra loro. Nel caso di invio volontario di più messaggi uguali quindi GAIAFT deve essere in grado di distinguere le copie di uno stesso messaggio da una serie di più messaggi uguali.
Per effettuare questa distinzione viene inserito dal mittente un contatore nell'*header GFT*, che viene incrementato a ogni invio di un messaggio. Visto che tutte le copie di un'entità eseguono le stesse operazioni, i loro contatori risultano sempre uguali per lo

stesso messaggio, e quindi tutte le copie di uno stesso messaggio contengono un contatore uguale. Al contrario, una sequenza intenzionale di messaggi uguali contiene dei valori differenti del contatore, e quindi il sistema si accorge che non si tratta di copie dello stesso messaggio e si comporta di conseguenza.

5.6.3 Tolleranza ai crash

La ricezione dei messaggi di GAIAFT in caso di tolleranza ai crash è basata sulla restituzione della prima copia di un messaggio arrivata, e sullo scarto delle copie successive. Questo perché, se per specifica vengono tollerati al massimo N crash, la tolleranza ai crash prevede almeno $N + 1$ copie di ogni entità, e quindi ne sopravvive almeno una, permettendo al destinatario di ricevere almeno una copia del messaggio.

Per migliorare le prestazioni velocizzando l'analisi dei messaggi in arrivo, l'algoritmo di ricezione dei messaggi (fig. 5.9) è basato sull'inserimento dei relativi *header GFT* all'interno di una tabella hash di tipo *Hashtable* (sez. 5.8.3), utilizzando come chiave dell'elemento inserito proprio l'intera struttura `header_gft_t`. In questo modo, è possibile capire in tempo $O(1)$ se un messaggio con lo stesso header (cioè inviato da una copia della stessa entità mittente, destinato a una copia della stessa entità destinatario, e con lo stesso contatore) è già stato ricevuto o no.
Se non è già stato ricevuto allora il messaggio corrente è il primo ad essere stato ricevuto, e la funzione `GAIAFT_Receive` restituisce al livello superiore il corpo del messaggio.
Se invece è già stato ricevuto, significa che il messaggio corrente è una copia arrivata successivamente e pertanto va scartato. Il livello superiore però si aspetta che se la funzione `GAIAFT_Receive` termina è a causa della ricezione di un messaggio. Per risolvere questo problema, dopo aver scartato la copia di un messaggio la funzione non termina, bensì richiama sè stessa ricorsivamente per la ricezione di un nuovo messaggio, questo finché una delle chiamate ricorsive della funzione termina restituendo un messaggio valido, che viene quindi restituito a cascata fino alla chiamata iniziale della funzione.

Per ridurre la memoria necessaria per l'analisi dei messaggi ricevuti, la tabella hash che contiene gli *header GFT* dei messaggi viene svuotata all'inizio di ogni timestep, in modo che gli header contenuti nella struttura siano al più limitati al numero di messaggi arrivato all'interno di un singolo timestep. Questo è reso possibile da una specifica

di GAIA che prevede che i messaggi che devono arrivare a destinazione durante un determinato timestep, arrivano necessariamente all'interno di quel timestep, anche a costo di rallentare l'intera simulazione.

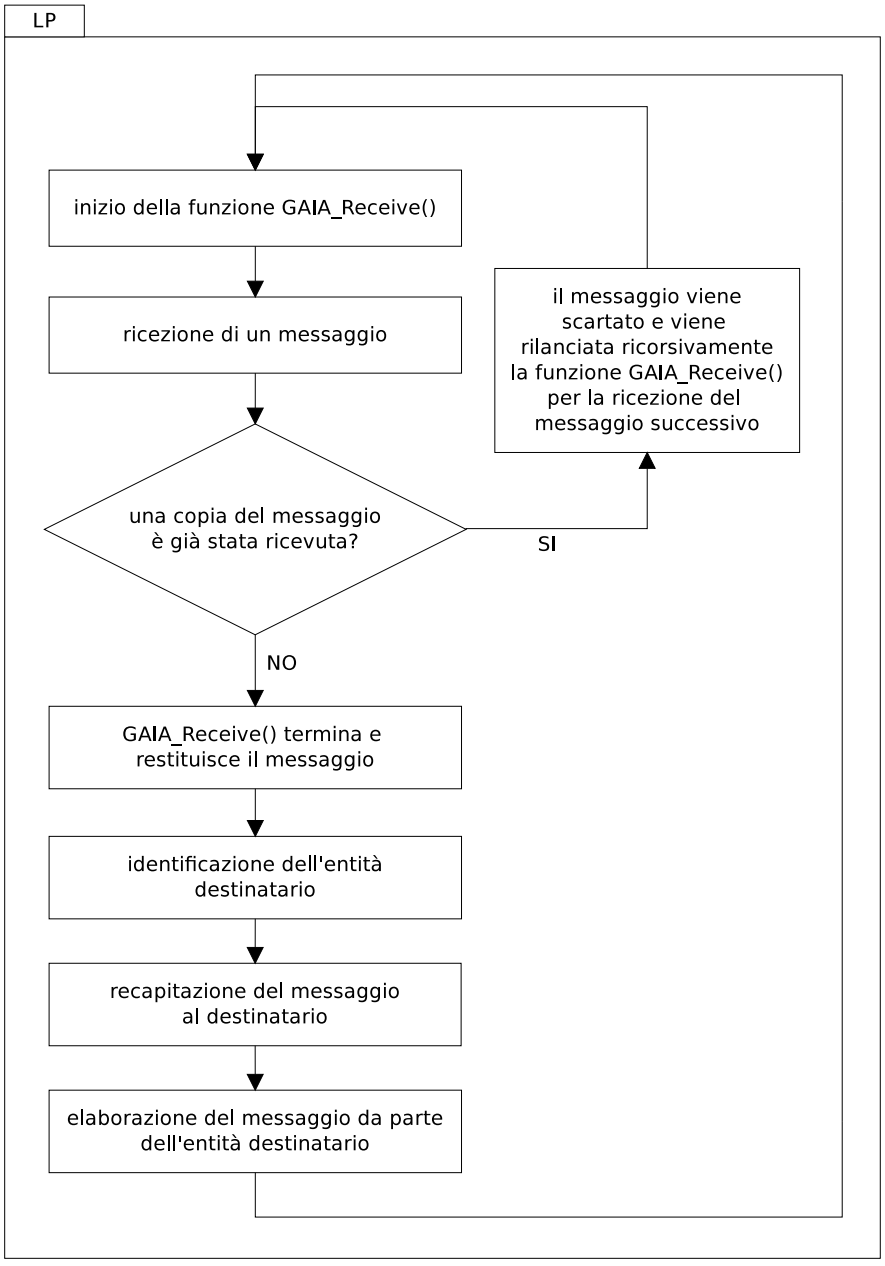

Figura 5.9: Diagramma di flusso di ricezione ed elaborazione dei messaggi da parte di un LP in una simulazione GAIAFT, con tolleranza ai crash.

5.6.4 Tolleranza ai guasti bizantini

La ricezione dei messaggi di GAIAFT in caso di tolleranza ai guasti bizantini è basata sull'attesa di una maggioranza di copie di un messaggio uguali tra loro sul totale dei messaggi attesi, sulla restituzione della prima copia del messaggio che soddisfa questa condizione di maggioranza, e infine sullo scarto delle copie successive. Questo perché, se per specifica vengono tollerati al massimo N guasti bizantini, la tolleranza ai guasti bizantini prevede almeno $2N + 1$ copie di ogni entità.
Da specifiche infatti, potenzialmente potrebbero esserci fino a N messaggi corrotti o assenti. Nel caso pessimo questi N messaggi corrotti potrebbero essere uguali tra loro, e per distinguerli dagli altri servirebbe di conseguenza un numero di messaggi corretti maggiore di N (quindi $N + 1$) su cui applicare un meccanismo di maggioranza, cioè un'analisi che considera valido un messaggio se ne vengono ricevute almeno $N + 1$ copie uguali tra loro.
Da questo deriva la necessità di avere almeno una ridondanza di $2N + 1$ copie di ogni messaggio, su cui cercare una maggioranza di almeno $N + 1$ copie uguali tra loro.

Rispetto all'analisi dei messaggi nel caso della tolleranza ai crash, in questo caso l'algoritmo è notevolmente più pesante, a causa di due motivi: il primo è la necessità di aspettare almeno l'$N + 1$esimo messaggio per poter restituire al livello superiore un messaggio fault tolerant valido; il secondo è la necessità di confrontare non solo gli *header GFT*, ma gli interi messaggi, per rilevare eventuali errori nel corpo dei messaggi.

Per migliorare le prestazioni velocizzando l'analisi dei messaggi in arrivo, l'algoritmo di ricezione dei messaggi (fig. 5.10) è basato sull'inserimento degli stessi messaggi all'interno di una tabella hash di tipo *Hashtable* (sez. 5.8.3), utilizzando come chiave dell'elemento inserito proprio l'intero messaggio. Allegato a ogni messaggio, viene inserito nella tabella anche un contatore di copie uguali ricevute fino a quel momento.
In questo modo, è possibile distinguere in tempo $O(1)$ i tre casi seguenti.

- Il messaggio corrente è la $N + 1$esima copia di un certo messaggio. In questo caso il messaggio corrente è da specifiche il primo messaggio sicuramente corretto (infatti le copie di un messaggio corrotto potrebbero essere al massimo N), e di conseguenza viene restituito al livello superiore dalla funzione `GAIAFT_Receive`.

- Sono state ricevute (compreso il messaggio corrente) meno di $N + 1$ copie del messaggio corrente. In questo caso il messaggio corrente non è ancora con sicurezza un messaggio corretto (potrebbe essere una delle copie di un messaggio corrotto), e quindi viene semplicemente incrementato il contatore che memorizza quante copie di quel messaggio sono arrivate fino a quel momento.

- Compreso il messaggio corrente, sono già state ricevute più di $N + 1$ copie del messaggio. In questo caso la prima copia sicuramente corretta del messaggio è già stata restituita dalla funzione `GAIAFT_Receive`, e di conseguenza il messaggio corrente viene scartato.

Nel secondo e nel terzo caso il messaggio ricevuto non viene ritornato dalla funzione `GAIAFT_Receive`, ma il livello superiore si aspetta che se la funzione termina è a causa della ricezione di un messaggio. Per risolvere questo problema, dopo aver memorizzato internamente o scartato la copia di un messaggio la funzione non termina, bensì richiama sè stessa ricorsivamente per la ricezione di un nuovo messaggio, questo finché una delle chiamate ricorsive della funzione termina restituendo un messaggio valido, che viene quindi restituito a cascata fino alla chiamata iniziale della funzione.

Per ridurre la memoria necessaria per l'analisi dei messaggi ricevuti, la tabella hash viene svuotata all'inizio di ogni timestep, in modo che la dimensione della struttura sia al più limitata al numero di messaggi arrivato all'interno di un singolo timestep. Questo è reso possibile da una specifica di GAIA che prevede che i messaggi che devono arrivare a destinazione durante un determinato timestep, arrivano necessariamente all'interno di quel timestep, anche a costo di rallentare l'intera simulazione.

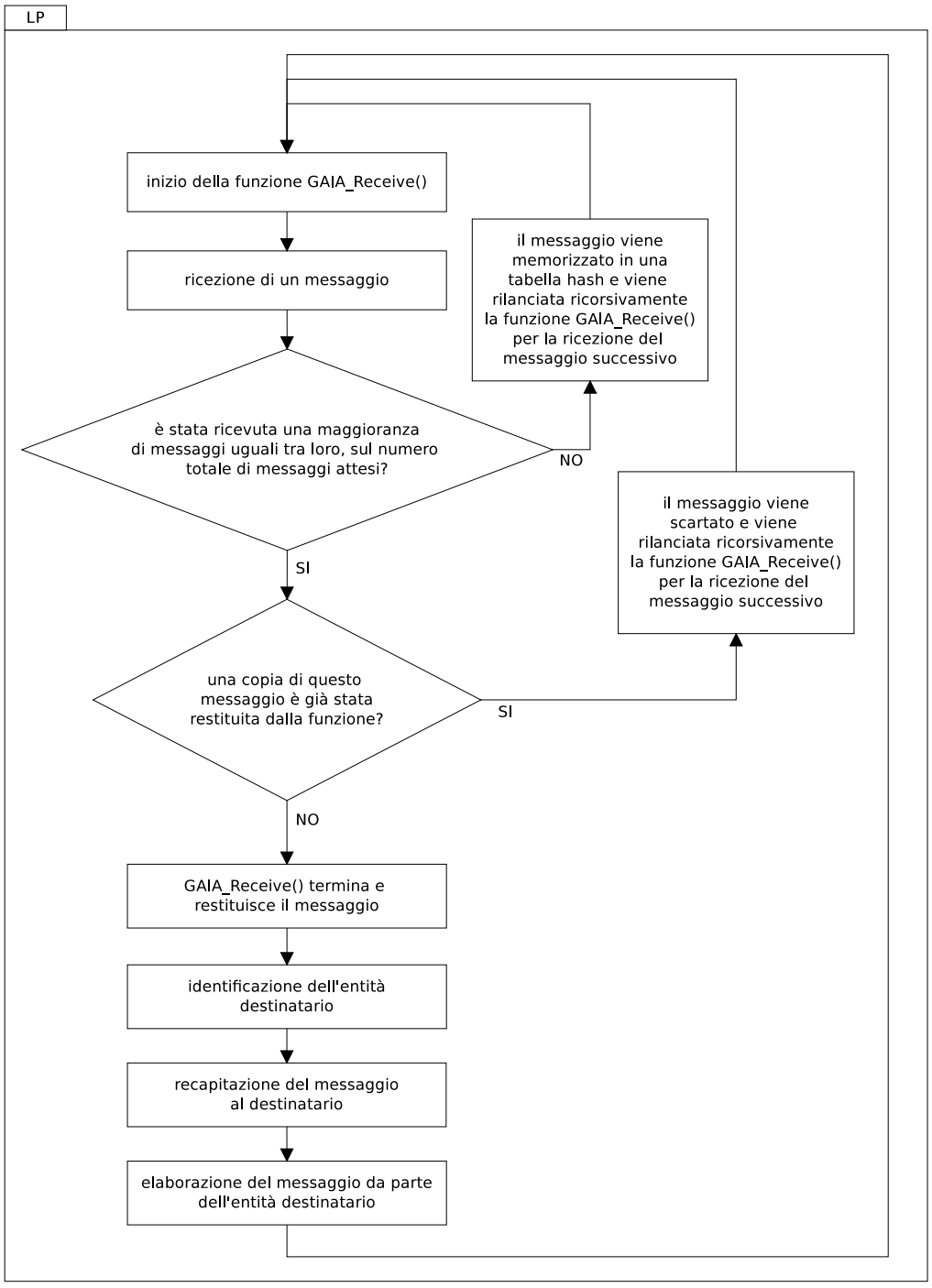

Figura 5.10: Diagramma di flusso di ricezione ed elaborazione dei messaggi da parte di un LP in una simulazione GAIAFT, con tolleranza ai guasti bizantini.

5.7 Gestione delle migrazioni

Come spiegato nelle sezioni 2.2 e 3.2.5, una delle principali caratteristiche del framework GAIA è quella di permettere alle entità simulate di migrare da un LP ad un altro, per ottimizzare il sistema distribuito adattandolo dinamicamente alle condizioni dell'infrastruttura sottostante.

A livello implementativo una migrazione di entità in GAIA avviene nel modo seguente:

- GAIA decide che una determinata entità deve migrare;
- GAIA invia al LP che contiene l'entità un messaggio `NOTIF_MIGR`, e a tutti gli altri un messaggio `NOTIF_MIGR_EXT` (in entrambi i tipi di messaggio viene specificato il LP di destinazione della migrazione);
- il LP che riceve il messaggio `NOTIF_MIGR` (LP di origine) memorizza l'entità interessata in una lista temporanea e aspetta la fine del timestep corrente;
- alla fine del timestep corrente, per ogni entità presente nella lista temporanea il LP di origine serializza gli stati locali di queste entità, e li invia a GAIA tramite l'apposita funzione `GAIA_Migrate`;
- all'inizio del timestep successivo, il LP di destinazione riceve da GAIA un messaggio `EXEC_MIGR` contenente lo stato dell'entità migrata;
- il LP di destinazione salva localmente lo stato dell'entità migrata, dopodiché questa si risveglia nel nuovo LP.

In GAIAFT questa procedura è stata leggermente modificata, a causa di due motivi. Il primo motivo è la necessità di mantenere sempre sincronizzate le informazioni relative agli LP di appartenenza delle copie delle entità, informazioni contenute negli FTregister di tutti gli LP. Il secondo motivo è la necessità di far mantenere alle entità migrate il proprio seme random personale e il proprio contatore dei messaggi inviati, informazioni memorizzate nell'FTregister del LP di origine della migrazione.

Per sincronizzare in tutti gli LP le informazioni relative agli LP di appartenenza delle copie delle entità, sono state implementate le seguenti modifiche alla procedura di migrazione di GAIA:

- quando un LP riceve un messaggio di tipo `NOTIF_MIGR` o di tipo `NOTIF_MIGR_EXT`, GAIAFT intercetta il messaggio e salva in una tabella temporanea l'ID dell'entità che vuole migrare e l'ID del LP di destinazione della migrazione (entrambe informazioni contenute nei due messaggi sopra citati);
- alla fine del timestep, all'interno della funzione `GAIAFT_TimeAdvance` viene aggiornato l'FTregister, inserendo il nuovo LP di appartenenza per tutte le entità specificate dalla tabella temporanea del punto precedente.

Per spedire insieme allo stato dell'entità migrata anche i relativi seme random e contatore invece, sono state implementate le seguenti modifiche alla procedura di migrazione di GAIA:

- la funzione `GAIAFT_Migrate` include nel pacchetto dello stato dell'entità una struttura contenente il seme random personale e il contatore dei messaggi inviati, dopodiché invia il nuovo pacchetto a GAIA tramite la funzione `GAIAFT_Migrate`;
- quando il LP di destinazione riceve il messaggio `EXEC_MIGR`, questo messaggio viene intercettato e viene estratta dal pacchetto la struttura contenente seme random e contatore;
- viene aggiornato l'FTregister locale inserendo il seme random e il contatore ricevuti;
- al messaggio `EXEC_MIGR` viene tolta la parte aggiuntiva inserita dalla `GAIAFT_Migrate`, e viene riottenuto quindi il messaggio originale, da restituire al livello superiore.

5.8 Strutture dati di supporto

Viste la grande mole e l'eterogeneità dei dati trattati da GAIAFT, per poter migliorare le prestazioni del sistema sono state sviluppate alcune strutture dati di supporto versatili, usate per svariati scopi all'interno del software.

5.8.1 Object

Alla base delle strutture dati sviluppate c'è la struttura *Object*, che serve a incapsulare un generico dato di dimensione e formato arbitrari, fornendo un'interfaccia omogenea per la sua manipolazione. In questo modo, le strutture che vedremo in seguito[7] potranno contenere e manipolare dati completamente eterogenei senza preoccuparsi del loro formato.

L'implementazione di un oggetto *Object* è una struttura `object_t` che contiene i campi seguenti.

`byte_obj* obj:` è il puntatore all'area di memoria che contiene i dati dell'oggetto.

`int size:` è la dimensione in byte dei dati dell'oggetto.

`boolean is_dyn_alloc:` è un valore booleano che serve a distinguere un oggetto la cui area di memoria è stata allocata appositamente sullo heap alla creazione dell'oggetto, rispetto a un oggetto la cui area di memoria è stata allocata precedentemente alla creazione dell'oggetto, sullo heap, sullo stack o staticamente (maggiori dettagli nella sottosezione *Creazione e copia di oggetti*).

`int description:` è una descrizione dell'oggetto, scelta dall'utente dell'oggetto a seconda delle sue esigenze[8].

[7] List (sez. 5.8.2) e Hashtable (sez. 5.8.3).

[8] La descrizione viene vista dall'utente come una stringa, ma viene memorizzata come un intero nella struttura. Il metodo usato per questa conversione è descritto dettagliatamente più avanti nella sottosezione *Description*

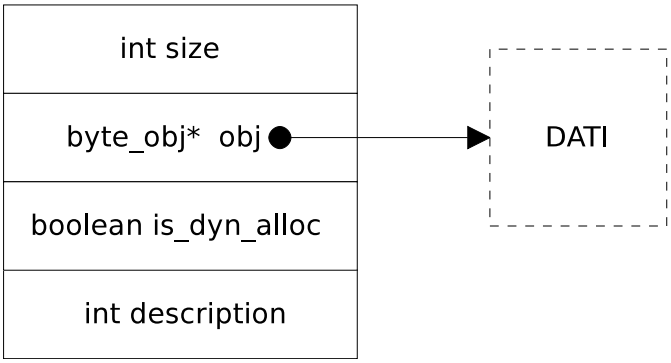

Figura 5.11: Rappresentazione di un oggetto *Object*.

Creazione e copia di oggetti

I possibili utilizzi degli *Object* sono molto vari, e per questo l'allocazione dello spazio di memoria necessario a contenere i dati può essere di diversa natura a seconda delle esigenze del programmatore. In particolare si distinguono[9] oggetti volatili, il cui utilizzo è limitato a un breve periodo, da oggetti persistenti, che vanno memorizzati per un lungo periodo.

Gli oggetti volatili vengono creati a partire da un puntatore ad un'area di memoria fornito in input dall'utente, solitamente un'area allocata sullo stack, e quindi valida solo durante lo svolgimento locale di una funzione, oppure più raramente un'area allocata sullo heap, ma di cui l'utente deve assumersi la responsabilità della deallocazione.

Gli oggetti persistenti invece allocano per i propri dati uno spazio apposito sullo heap, e quando non servono più devono essere poi deallocati con l'apposita funzione `object_free`.

Al fine di semplificare l'utilizzo degli *Object* nella programmazione del proprio software e rimuovere possibili ambiguità e fonti di errori di programmazione, la creazione e la copia di oggetti seguono questi accorgimenti:

- l'API del modulo *Object* non permette la creazione diretta di oggetti persistenti, possono essere creati solo oggetti volatili usando la funzione `object_new`;

[9]Distinzione fatta per mezzo del booleano `is_dyn_alloc`, descritto nella sezione precedente.

- la copia di un oggetto viene sempre realizzata con un oggetto persistente (perché i dati devono essere copiati in una nuova zona di memoria, allocata appositamente), per mezzo della funzione `object_alloc_copy`;

- l'utente può comunque creare un oggetto persistente, semplicemente creandone uno volatile con la `object_new` e dando l'oggetto temporaneo ottenuto in input alla `object_alloc_copy`, che restituisce l'oggetto persistente.

Description

Il campo description è una descrizione testuale a disposizione del programmatore che utilizza gli *Object* nel proprio software. Un classico utilizzo ad esempio è quello di riconoscere il tipo di un oggetto per potervi applicare le giuste funzioni[10].

La descrizione è testuale agli occhi del programmatore, ma viene tradotta automaticamente in un numero intero a livello implementativo. La traduzione da stringa a intero (in fase di memorizzazione) e da intero a stringa (in fase di lettura) vengono effettuate tramite un dizionario globale comune a tutti gli *Object*, realizzato per mezzo di tabelle hash *Hashtable* (sez. 5.8.3).

A causa di questo approccio, l'utilizzo di oggetti *Object* richiede l'utilizzo di tabelle *Hashtable*, ma le tabelle *Hashtable* a loro volta sono implementate come collezioni di oggetti *Object*. Per evitare i loop che sarebbero causati da questa architettura mutuamente ricorsiva, è prevista l'apposita descrizione `NO_DESCRIPTION` che viene tradotta automaticamente con un valore intero di riferimento, senza richiedere l'utilizzo delle tabelle hash del dizionario di conversione.

[10]Due esempi di utilizzo del campo description all'interno di questo software, sono la distinzione di chiavi costituite da tipi di dato diversi all'interno di tabelle *Hashtable* (sez. 5.8.3), e il riconoscimento del tipo di un oggetto ricevuto via rete utilizzando la funzione `receive_object` (sottosezione successiva, *Serializzazione e spedizione*).

Serializzazione e spedizione di Object

Ogni oggetto *Object* prevede la possibilità di essere serializzato, ad eccezione degli oggetti che contengono a loro volta dei puntatori ad altre zone di memoria.

La serializzazione di un oggetto consiste nell'inserire in un unico vettore di byte sequenziale tutti i dati contenuti nell'oggetto, compresi dimensione e campo description, per mezzo della funzione `object_serialize`. Questo vettore può essere poi deserializzato tramite la funzione inversa `object_deserialize`, che restituisce l'oggetto originale.

Un dettaglio da considerare nella procedura di serializzazione è che il campo description viene inserito sotto forma di stringa, perché potendo l'oggetto essere ricevuto e deserializzato da un altro processo o da un altro elaboratore, questi ultimi avrebbero un differente dizionario di conversione intero/stringa rispetto a quello del processo originale, in cui il valore numerico del campo description non verrebbe riconosciuto. Per risolvere questo problema quindi il campo description viene serializzato sotto forma di stringa, e tale stringa viene poi inserita nel dizionario al momento della deserializzazione. In questo modo, sia se la stringa esiste già nel dizionario, sia se è nuova, nell'oggetto deserializzato viene memorizzato il corretto valore numerico locale corrispondente alla stringa data.

Serializzazione e deserializzazione sono funzionalità utili principalmente in due casi: memorizzazione di un oggetto su un file e spedizione di un oggetto ad un altro processo attraverso la rete.

Per il secondo caso in particolare, sono state sviluppate le funzioni `send_object` e `receive_object` allo scopo di semplificare la spedizione di oggetti su socket TCP/IP. Le due funzioni infatti sono caratterizzate da un'interfaccia molto semplice: la funzione `send_object` chiede come parametri solo il socket descriptor e l'oggetto da spedire, mentre la funzione `receive_object` chiede solo il socket descriptor e restituisce l'oggetto ricevuto. Questo permette all'utente di non preoccuparsi di specificare le dimensioni, la tipologia o altri parametri dell'oggetto da spedire.

5.8.2 List

La prima struttura sviluppata basata sugli *Object* è una lista. Più precisamente si tratta di una lista bilinkata circolare con sentinella [27], che cioè prevede per ogni elemento un puntatore al nodo precedente e uno al nodo successivo, prevede un collegamento tra il primo e l'ultimo nodo della lista, e infine prevede un nodo sentinella che fornisce un punto di accesso alla lista e ne costituisce l'inizio/fine (fig. 5.12).

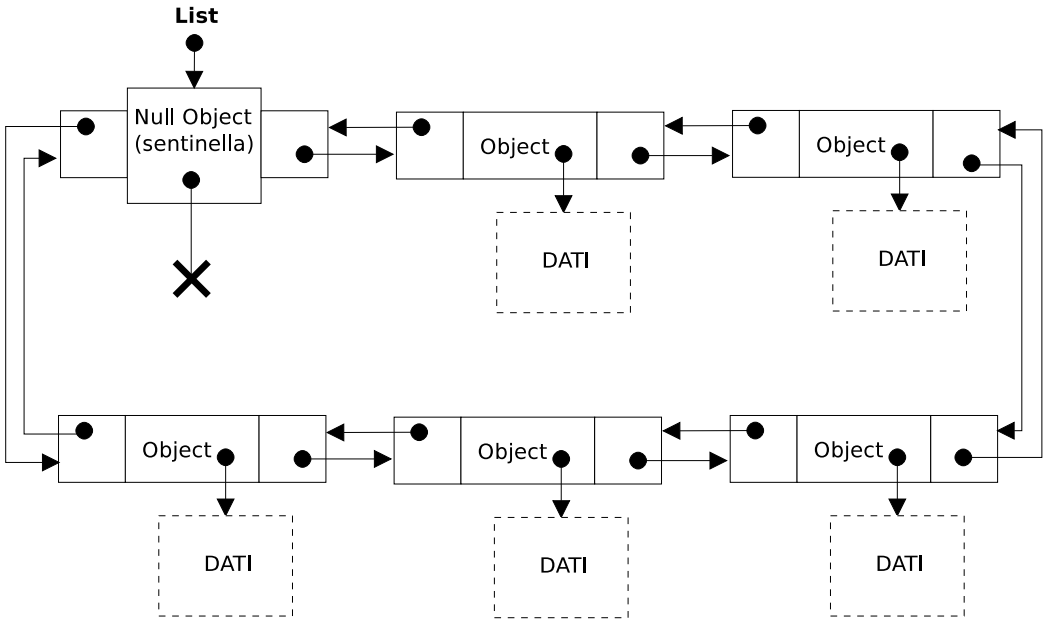

Figura 5.12: Rappresentazione di una lista di *Object* bilinkata circolare con sentinella.

Ogni elemento della lista contiene un *Object*[11], e questo rende possibile alla lista immagazzinare qualunque tipo di insieme di dati, anche costituito da formati di dato eterogenei.

[11] Sezione 5.8.1.

5.8.3 Hashtable

La struttura dati *Hashtable* è una tabella hash con liste di trabocco [27] di oggetti *Object*. Grazie alle ottime prestazioni che caratterizzano le tabelle hash[12] e alla grande versatilità dovuta all'utilizzo degli *Object*, la struttura *Hashtable* è stata usata in moltissimi campi e sotto moltissime forme all'interno del software GAIAFT, al punto che in alcuni casi ne costituisce addirittura il motore[13].

La caratteristica che rende così versatile questa struttura è il fatto che non solo l'elemento che viene inserito è un oggetto di tipo *Object*, ma anche la chiave di ricerca è un *Object*. Questo significa che sia la chiave sia l'elemento possono essere potenzialmente qualunque cosa rappresentabile in byte, e la tabella può contenere sia negli elementi sia nelle chiavi dati completamente eterogenei e di qualunque natura.

Questa caratteristica può però generare un problema di ambiguità: può capitare infatti che in una stessa tabella si vogliano usare come chiavi due oggetti apparentemente diversi, ma con uguale rappresentazione in byte. I due oggetti potrebbero essere ad esempio un numero intero e una stringa di quattro caratteri, entrambi descritti dagli stessi identici byte. Questi due oggetti infatti agli occhi dell'utente apparirebbero diversi, ma agli occhi del confronto tra oggetti effettuato byte per byte risulterebbero uguali.

Per risolvere questo problema, la funzione di creazione di un oggetto da usare come chiave richiede obbligatoriamente l'inserimento del campo description (sez. 5.8.1). In questo modo, oltre al confronto byte per byte si aggiunge anche il confronto delle description, così se un utente vuole usare nella stessa tabella chiavi eterogenee può farlo

[12] Le tabelle hash sono caratterizzate dal poter effettuare operazioni di inserimento, ricerca, eliminazione e modifica di elementi in tempo costante $O(1)$, se si trovano in condizioni ottimali di bilanciamento tra numero di elementi e numero di bucket. La struttura *Hashtable* prevede anche meccanismi che mantengono queste condizioni ottimali di bilanciamento. Per una descrizione dettagliata di questi meccanismi vedere la sottosezione successiva, *Autoridimensionamento e prestazioni*.

[13] Tra i vari utilizzi all'interno di GAIAFT, i semplici inserimento e ricerca di elementi in una *Hashtable* costituiscono integralmente l'algoritmo di analisi e ricerca dell'header e del corpo dei messaggi ricevuti dalle entità per individuare doppioni e messaggi errati (sez. 5.6).

senza pericolo di ambiguità, inserendo descrizioni diverse per tipi di dato diversi (tornando all'esempio sopra descritto, l'utente potrebbe, in fase di creazione delle chiavi, inserire descrizioni come "`intero`", "`stringa`", o ciò che ritiene più opportuno).

In fig. 5.13 si può vedere una rappresentazione dell'implementazione della tabella, costituita da una struttura globale e da un array di bucket (ogni bucket è una lista di tipo *List*, vedere sez. 5.8.2).

Il tipo `hashtable_t` è un puntatore alla struttura globale, e fornisce il punto di accesso alla tabella. Come si può vedere in fig. 5.13 la struttura globale contiene il puntatore all'array dei bucket, il contatore del numero di bucket, il contatore del numero di elementi, l'ultima autokey[14] inserita e un booleano che specifica se la tabella è di tipo *autoresizing*[15] o no.

[14]Se la tabella hash serve unicamente come collezione di dati generica senza necessità di ricerca, l'API del modulo *Hashtable* dispone di una funzione che inserisce un elemento scegliendo automaticamente una chiave univoca (l'`autokey`), senza che questa debba essere scelta o vista dall'utente. La tabella infatti offre sia funzioni per l'accesso ai dati basato sulle chiavi di ricerca, sia funzioni per l'accesso ai dati sequenziale, senza quindi dover disporre di nessuna chiave.

[15]Per approfondimenti sulle tabelle *autoresizing* vedere la sottosezione successiva, *Autoridimensionamento e prestazioni*.

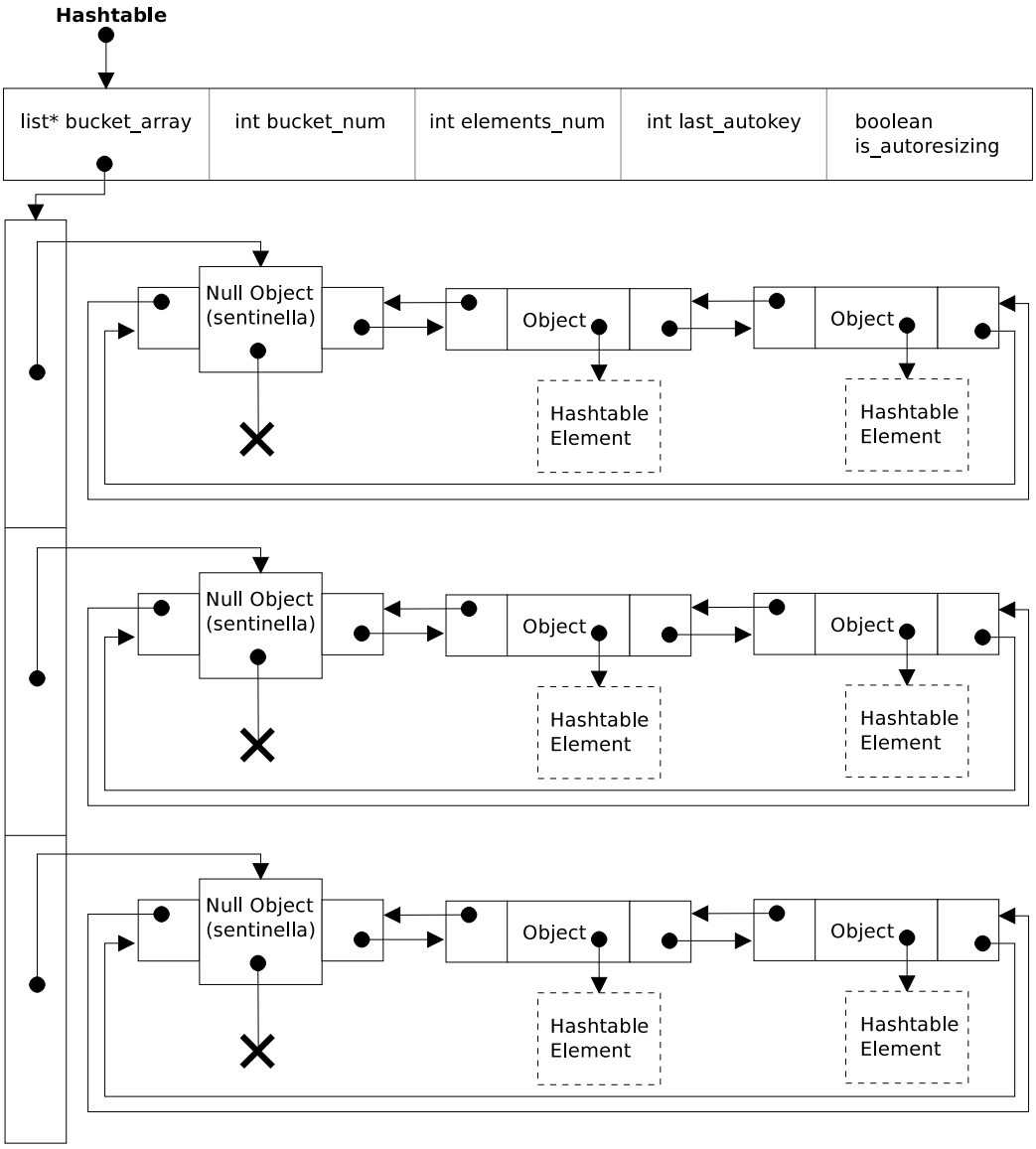

Figura 5.13: Implementazione della struttura *Hashtable* (i singoli elementi *Hashtable Element* sono descritti in fig. 5.14).

Ogni elemento delle liste di trabocco che costituiscono i bucket della tabella, è costituito a sua volta da un *Object* che incapsula una struttura chiamata *Hashtable Element*. Come si può vedere in fig. 5.14, questa struttura è composta da un puntatore al proprio elemento della lista di trabocco e da due oggetti di tipo *Object*: la chiave di ricerca e il

vero e proprio elemento da inserire nella tabella hash.

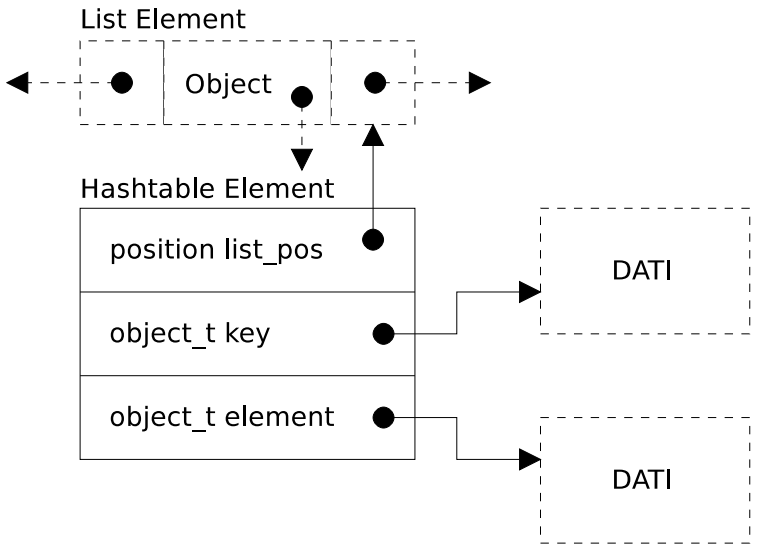

Figura 5.14: Implementazione di un elemento *Hashtable Element*.

Autoridimensionamento e prestazioni

Le tabelle hash degradano le proprie prestazioni all'aumentare del numero di elementi inseriti. Finché il numero di elementi non supera il numero di bucket le prestazioni sono ottime, ma quando il numero di elementi supera considerevolmente il numero di bucket, le operazioni di ricerca e inserimento degli elementi risultano rallentate, a causa dell'allungamento delle liste di trabocco.

Nell'implementazione della tabella hash *Hashtable* è stato sviluppato un metodo per mantenere inalterate le prestazioni, anche in seguito all'inserimento di una grande mole di elementi. La tabella hash, infatti, a ogni operazione di inserimento mantiene monitorata la lunghezza media delle liste di trabocco (ottenuta calcolando il rapporto tra numero di elementi e numero di bucket). In seguito, quando questo valore supera una certa soglia, la tabella hash si autoridimensiona, aumentando il numero di bucket e riabbassando così la lunghezza media delle liste di trabocco. Nel grafico in fig. 5.15 si può osservare che questa operazione viene ripetuta ad ogni superamento della soglia di carico accettabile della tabella, e determina un incremento proporzionale del numero di bucket.

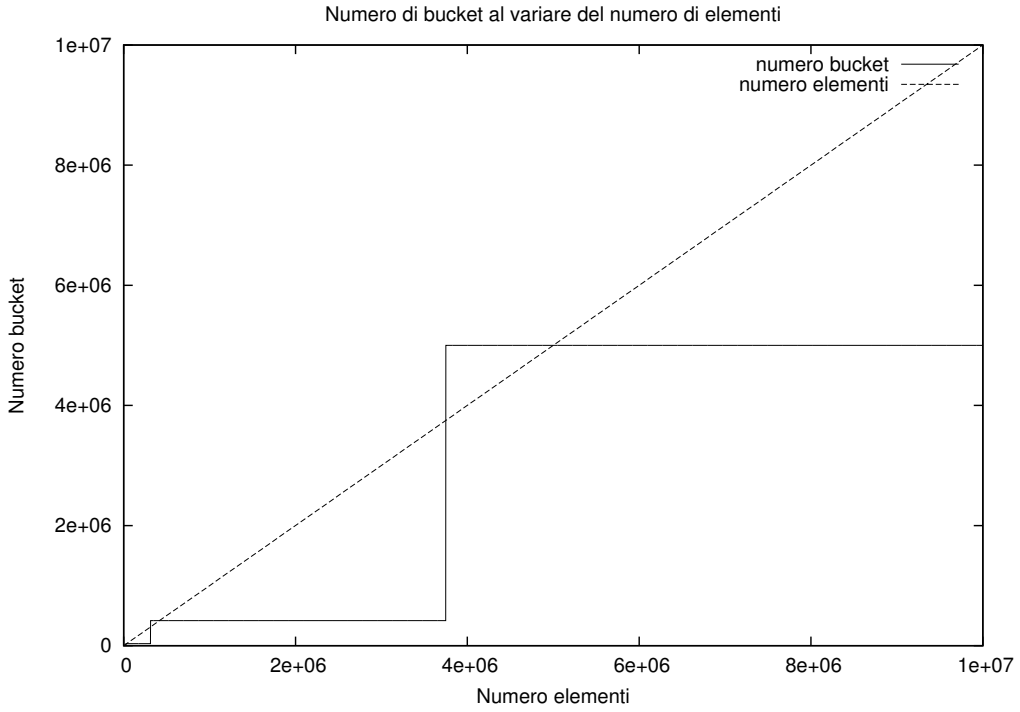

Figura 5.15: Ridimensionamento del numero di bucket della tabella hash all'aumentare del numero di elementi inseriti.

Nel grafico in fig. 5.16 si può osservare il risultato di un test prestazionale. Si può notare, in ognuna delle singole fasi discendenti, che all'aumentare del numero di elementi inseriti le prestazioni degradano, ma in seguito ai ridimensionamenti della tabella le prestazioni migliorano riassestandosi su valori soddisfacenti. In assenza del sistema di ridimensionamento automatico le prestazioni degraderebbero seguendo l'andamento discendente che si può osservare nel grafico, mentre usando questo sistema le prestazioni non degradano mai oltre una certa soglia prestabilita.

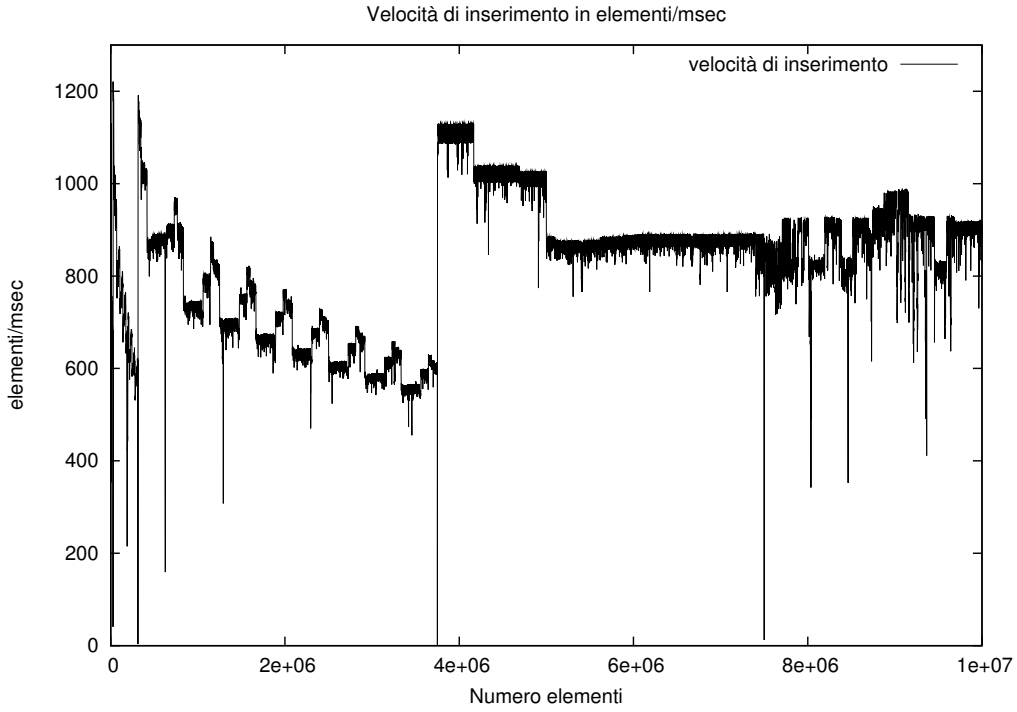

Figura 5.16: Velocità di inserimento degli elementi nella tabella hash.

Serializzazione e spedizione di Hashtable

La serializzazione/deserializzazione di intere tabelle hash è una funzionalità molto utile per poter memorizzare intere tabelle su file o spedirle ad un altro processo attraverso la rete.

In particolare, grazie alla versatilità delle tabelle hash di tipo *Hashtable* descritta precedentemente, trasferire attraverso la rete una *Hashtable* risulta essere un metodo molto semplice da implementare per trasferire in un unico blocco insiemi di dati eterogenei e complesse strutture dati, senza doversi preoccupare minimamente di cosa contiene tale tabella al momento della spedizione.

Analogamente al caso di serializzazione e deserializzazione degli *Object*, per le tabelle *Hashtable* sono previste le funzioni `hashtable_serialize` e `hashtable_deserialize`: la prima memorizza un'intera *Hashtable* in un unico vettore sequenziale di byte, mentre la

seconda prende in input il vettore di byte della tabella serializzata e restituisce la tabella deserializzata.

Come per la spedizione degli *Object*, per la spedizione di *Hashtable* su socket TCP/IP sono state sviluppate due funzioni caratterizzate da un'interfaccia molto semplice: la prima, `send_hashtable`, chiede come parametri solo il socket descriptor e la tabella hash da spedire, mentre la seconda, `receive_hashtable`, chiede solo il socket descriptor e restituisce la tabella hash ricevuta.

Le due funzioni inoltre implementano un controllo di integrità della tabella ricevuta basato sull'algoritmo MD5 [28]. Al momento della ricezione infatti, viene calcolato il valore MD5 della tabella serializzata ricevuta e confrontato con il valore MD5 della tabella serializzata originale. In questo modo, in caso di valori MD5 differenti possono essere rilevati eventuali errori nel trasferimento di una tabella.

Capitolo 6

Valutazione delle prestazioni

È logico supporre che l'introduzione del layer aggiuntivo GAIAFaultTolerance (GAIAFT) all'interno del framework GAIA/ARTÌS[1] abbia una certa influenza sulle prestazioni delle simulazioni basate su questa architettura. In questo capitolo viene quindi valutato l'impatto sulle prestazioni delle simulazioni, causato dall'utilizzo di GAIAFT.

6.1 Modelli di simulazione

Per testare il corretto funzionamento di GAIAFT ed effettuarne la valutazione delle prestazioni è stato necessario implementare ed eseguire delle simulazioni di prova. In questa sezione vengono descritti i modelli di simulazione che sono stati sviluppati per effettuare questi test.

6.1.1 Coda

Al fine di testare il corretto funzionamento della tolleranza a crash e guasti bizantini fornita da GAIAFT, è stato sviluppato inizialmente un modello di simulazione che prevedesse entità con ruoli diversificati, colli di bottiglia e una struttura non uniforme.

Questo modello serve a simulare entità che vogliono accedere a un servizio, il quale può essere erogato solo a un'entità per volta. Le entità quindi, se il servizio è libero

[1] Ambiente GAIA/ARTÌS: cap. 2.

vengono servite immediatamente, mentre se il servizio è occupato si mettono in attesa del proprio turno formando una coda. Una volta servite, le entità si mettono in attesa per un certo tempo, per poi riaccedere nuovamente al servizio rientrando in coda.

Più precisamente il comportamento del sistema (fig. 6.1) è il seguente. Le entità si trovano inizialmente in una zona di attesa. Ad un certo momento decidono di entrare in coda e chiedono il permesso al gestore degli accessi, che concede o nega loro tale permesso. Se il permesso viene negato, esse riprovano a chiederlo ripetutamente a intervalli di tempo, finché non viene loro concesso. Quando il permesso viene concesso, l'entità che si inserisce in coda comunica la propria presenza all'entità precedente, che a sua volta memorizza questa informazione. In questo modo, quando un'entità viene servita e lascia libero il servizio, comunica alla successiva in coda che il servizio è libero. Quando questo accade, l'entità successiva in coda riceve un messaggio di servizio libero dalla precedente, e occupa a sua volta il servizio.

Oltre al normale funzionamento della coda descritto sopra, vengono anche gestiti i casi in cui la coda è vuota. Se la coda è vuota quando un'entità entra in essa, l'entità viene immediatamente servita. Se invece al termine del servizio un'entità esce dalla coda e non c'è nessun'altra entità in attesa, l'entità che esce, invece di comunicare l'informazione di servizio libero alla successiva, comunica questa informazione al gestore degli accessi. Il gestore degli accessi a sua volta farà servire immediatamente la prima entità che si inserisce in coda.

Il gestore degli accessi è anch'esso un'entità, ma svolge un ruolo completamente differente rispetto alle altre entità simulate. La presenza del gestore degli accessi serve ad evitare problemi di concorrenza, quali collisioni di entità che voglio entrare contemporaneamente in coda, o entità che vogliono uscire contemporaneamente all'entrata in coda di altre entità. Senza il gestore degli accessi questi scenari potrebbero portare alla formazione di situazioni ambigue, essendo tutto lo svolgimento della simulazione basato sullo scambio di messaggi tra entità: potrebbe succedere ad esempio che due entità vogliano entrare in coda contemporaneamente e comunichino la propria intenzione all'ultima entità della coda. In questo caso, l'ultima entità della coda non saprebbe quale delle due scegliere come entità successiva, e di conseguenza non si verrebbe a formare una coda

lineare in cui le due entità sono concatenate una dopo l'altra.

Ciò che fa il gestore degli accessi in questa situazione è memorizzare l'ID dell'ultima entità in coda, e far entrare le entità una per volta per mezzo della concessione di un token. In questo modo si può essere sicuri che l'entità che entra in coda sia una sola, e possa tranquillamente comunicare la propria entrata alla precedente (l'ID dell'entità precedente è fornito dal gestore degli accessi). L'entità successiva potrà entrare solo dopo il rilascio del token da parte dell'entità corrente, e quando questo accadrà, l'entità corrente avrà già comunicato al gestore degli accessi di essere la nuova ultima entità della coda. Il gestore degli accessi quindi, potrà comunicare alla prossima entità che vuole accodarsi l'ID dell'ultima entità in coda, senza pericolo di ambiguità.

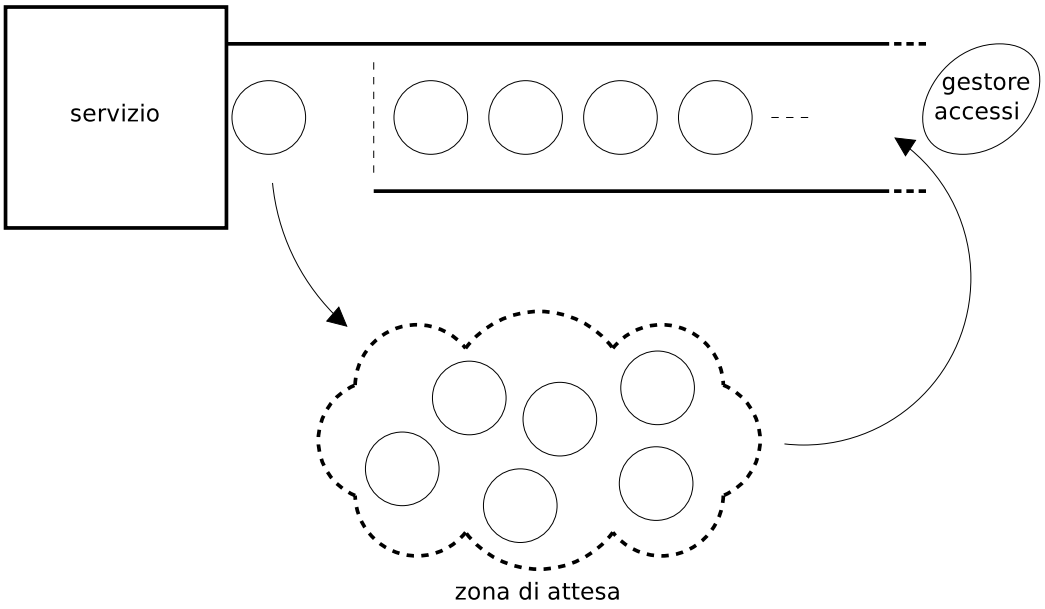

Figura 6.1: Comportamento delle entità (rappresentate dai cerchi) nel modello di simulazione di una coda. Il gestore degli accessi è anch'esso un'entità, anche se svolge un ruolo differente da quello delle altre.

6.1.2 Grafo casuale

Il modello di simulazione della coda visto nella sezione precedente è caratterizzato dalla presenza di entità con ruoli eterogenei e da una struttura delle interazioni tra le entità

non uniforme.

A causa di queste caratteristiche il modello della coda non è risultato adatto ad essere usato nei test per la valutazione delle prestazioni. Si è reso necessario quindi lo sviluppo di un altro modello di simulazione che rispondesse maggiormente ai requisiti di omogeneità e uniformità del sistema simulato.

Il modello scelto per questo scopo è stato quello di un grafo casuale (fig. 6.2). Questo modello prevede un grafo formato da un insieme di nodi omogenei, in cui ognuno di essi, corrispondente a un'entità simulata, ha un certo numero di vicini (cinque vicini nel modello implementato per i test, due vicini nell'esempio in fig. 6.2). Gli archi che collegano i vicini sono unidirezionali, è possibile quindi che la relazione di vicinanza tra due nodi non sia reciproca.

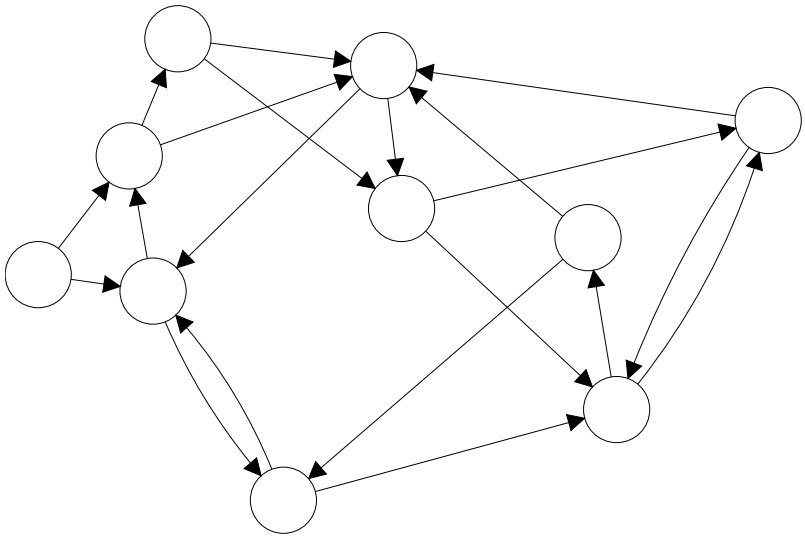

Figura 6.2: Modello di simulazione di un grafo casuale. Ogni nodo ha due archi uscenti, che rappresentano la vicinanza. Gli archi sono unidirezionali, quindi la relazione di vicinanza tra due nodi data da un arco non è necessariamente reciproca.

Durante la simulazione, l'attività di ogni nodo consiste nell'inviare, a intervalli di tempo generati casualmente, messaggi `PING` agli altri nodi, ricevere in risposta dei messaggi `PONG`, e calcolare una media delle latenze con cui vengono ricevuti i `PONG`. Le latenze

casuali vengono generate secondo una distribuzione lognormale, trattata più avanti nella sottosez. *Simulazione delle latenze di rete.*

Ogni nodo sceglie i nodi a cui spedire i `PING` con una certa probabilità tra i propri vicini, e con la probabilità complementare tra tutti i nodi del grafo. Ad esempio nel caso del modello implementato per i test, un nodo ha l'80% di probabilità di scegliere come destinatario un proprio vicino, e il 20% di probabilità di scegliere un nodo qualunque. Una volta effettuata questa scelta iniziale, il destinatario viene scelto uniformemente sia all'interno dell'insieme dei vicini, sia all'interno dell'insieme di tutti i nodi.

Inoltre, ogni nodo a intervalli di tempo casuali cambia la propria lista dei vicini, rigenerandone casualmente una nuova.

Più dettagliatamente, il modello di simulazione del grafo casuale è basato sulla gestione dei seguenti eventi.

Un nodo riceve un messaggio `START_SIM`. Questo messaggio viene inviato all'inizio a ogni nodo, per far partire la simulazione. Alla ricezione di questo messaggio, il nodo esegue le seguenti operazioni:

- genera cinque vicini casuali con distribuzione uniforme sull'insieme di tutti i nodi, e memorizza la lista nel proprio stato;
- genera un destinatario casuale per il primo `PING` (con probabilità 80% viene scelto uniformemente tra i vicini, con probabilità 20% viene scelto uniformemente tra tutti i nodi);
- genera un tempo di attesa casuale con distribuzione uniforme, dopo il quale invierà il primo `PING` al destinatario scelto;
- invia il messaggio `PING` al destinatario casuale, specificando come tempo di arrivo del messaggio il tempo casuale generato precedentemente;
- genera un altro tempo di attesa casuale con distribuzione uniforme (notevolmente più lungo del precedente), dopo il quale rigenererà la lista dei vicini;
- invia a sè stesso nel futuro il messaggio `CHANGE_NEIGHBORS`, specificando come tempo di arrivo il tempo casuale generato precedentemente;

- incrementa nel proprio stato il numero di `PING` inviati.

Un nodo riceve un messaggio `PING`. Alla ricezione di questo messaggio, il nodo esegue le seguenti operazioni:

- incrementa nel proprio stato il numero di `PING` ricevuti;
- genera una latenza casuale con distribuzione lognormale[2], dopo la quale risponderà al mittente con un messaggio `PONG`;
- risponde al mittente del `PING` inviando il messaggio di risposta `PONG`, specificando come tempo di arrivo del messaggio la latenza casuale generata precedentemente.

Un nodo riceve un messaggio `PONG`. Alla ricezione di questo messaggio, il nodo esegue le seguenti operazioni:

- calcola la media incrementale[3] delle latenze dei `PONG` ricevuti;
- genera un destinatario casuale per il prossimo `PING` (con probabilità 80% viene scelto uniformemente tra i vicini, con probabilità 20% viene scelto uniformemente tra tutti i nodi);
- genera un tempo di attesa casuale con distribuzione uniforme, dopo il quale invierà il prossimo `PING` al destinatario scelto;
- invia il messaggio `PING` al destinatario casuale, specificando come tempo di arrivo del messaggio il tempo casuale generato precedentemente;
- incrementa nel proprio stato il numero di `PING` inviati.

Un nodo riceve da sè stesso un messaggio `CHANGE_NEIGHBORS`. Alla ricezione di questo messaggio, il nodo esegue le seguenti operazioni:

[2]La distribuzione scelta per le latenze casuali è la distribuzione lognormale, essendo quella che meglio approssima il reale comportamento della rete (vedere sottosez. *Simulazione delle latenze di rete*).

[3]La media incrementale delle latenze viene aggiornata alla ricezione di ogni `PONG` con la seguente formula:

$$\text{nuova media} = \frac{(\text{media precedente} * (\text{ping inviati} - 1)) + \text{ultima latenza}}{\text{ping inviati}}$$

.

- rigenera cinque nuovi vicini casuali con distribuzione uniforme sull'insieme di tutti i nodi, e aggiorna il proprio stato sostituendo la vecchia lista dei vicini con quella appena generata;
- genera un tempo di attesa casuale con distribuzione uniforme, dopo il quale rigenererà nuovamente la lista dei vicini;
- invia a sè stesso nel futuro il messaggio `CHANGE_NEIGHBORS`, specificando come tempo di arrivo il tempo casuale generato precedentemente.

Simulazione delle latenze di rete

Il modello di simulazione del grafo casuale prevede nodi che spediscono messaggi `PING` ad altri nodi, e ottengono in risposta messaggi `PONG`. Ciò che si vorrebbe ottenere è che il tempo di risposta simuli il tempo di andata e ritorno (Round Trip Time, RTT) di uno scambio di messaggi spediti su una rete reale. Il Round Trip Time di uno scambio di messaggi tra due nodi consiste nella somma di tre valori: la latenza del messaggio in andata, il tempo di computazione necessario per elaborare e inviare il messaggio di risposta, e la latenza del messaggio di risposta.

Nel caso del grafo casuale simulato, si considera il tempo di elaborazione del messaggio di risposta trascurabile rispetto alle latenze di andata e ritorno, pertanto il Round Trip Time tra due nodi simulati consiste semplicemente nella somma della latenza del messaggio `PING` in andata e della latenza del messaggio `PONG` di risposta.

Le singole latenze di rete sono a loro volta scomponibili in due parti, *latenza fissa* e *latenza variabile*:

- la latenza fissa è dovuta ai limiti fisici del canale di comunicazione (principalmente alla distanza fisica tra gli estremi della comunicazione), e quindi rimane invariata per ogni messaggio scambiato, non essendo influenzata dalle condizioni momentanee della rete;
- la latenza variabile cambia da messaggio a messaggio, ed è dovuta alle condizioni momentanee della rete, che possono dipendere da caratteristiche dei protocolli di comunicazione, elaborazione e smistamento dei messaggi, congestione del traffico di rete, prestazioni dell'infrastruttura di rete, ecc.

Da analisi sul traffico di rete [37, 38, 39], si è notato che la durata delle latenze dei singoli messaggi in transito sulla rete segue una distribuzione **lognormale traslata** [21, 22, 23, 24, 25].

Questa distribuzione infatti rispecchia il comportamento della latenza descritto precedentemente. La traslazione verso destra è causata dall'influenza della latenza fissa al di sotto della quale non è possibile scendere. L'andamento della funzione lognormale invece descrive la distribuzione della latenza variabile: la maggior parte dei messaggi subisce una latenza di poco superiore a quella fissa, mentre solo alcuni messaggi subiscono latenze maggiori (fig. 6.3).

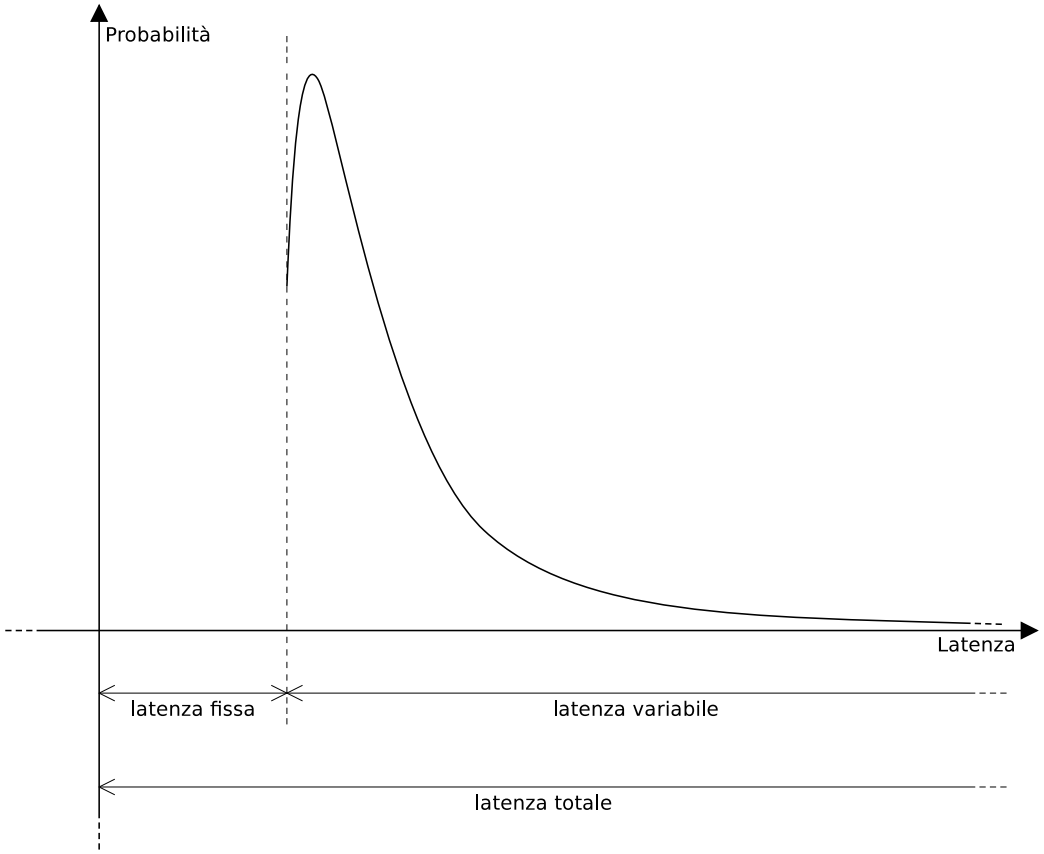

Figura 6.3: Distribuzione di probabilità lognormale traslata, che simula il comportamento delle latenze di rete.

Generazione pseudocasuale di una latenza

Disponendo del generatore uniforme di numeri pseudocasuali incluso in GAIA, per generare dei numeri pseudocasuali con distribuzione lognormale è stato implementato un algoritmo di tipo *acceptance-rejection* [29, 30].

Questo tipo di algoritmo permette di generare numeri pseudocasuali con una qualunque distribuzione di probabilità, basandosi sulla funzione di densità di tale distribuzione. Più precisamente, considerando ad esempio di voler generare un numero pseudocasuale compreso tra due estremi MIN e MAX, con una generica distribuzione che ha come densità la funzione $f(x)$, l'algoritmo è il seguente:

1. viene generato un numero x con distribuzione uniforme tra MIN e MAX;
2. si calcola il valore di $f(x)$, e si genera un numero y con distribuzione uniforme tra 0 e 1;
3. se $f(x) > y$, il valore x viene restituito come il numero pseudocasuale generato, altrimenti viene scartato, e si ripete l'operazione dal passo 1.

Il generatore lognormale di numeri pseudocasuali, utilizzato nel modello di simulazione per la generazione delle latenze di rete simulate, è stato implementato secondo questo metodo, utilizzando la funzione densità della distribuzione lognormale [34]:

$$f(x, \zeta, \sigma^2) = \frac{1}{x\sqrt{2\pi\sigma^2}}\, e^{-\frac{(\log(x)-\zeta)^2}{2\sigma^2}}$$

In figura 6.4 si può vedere il risultato di un test dell'algoritmo descritto, utilizzato per generare numeri pseudocasuali con distribuzione lognormale.

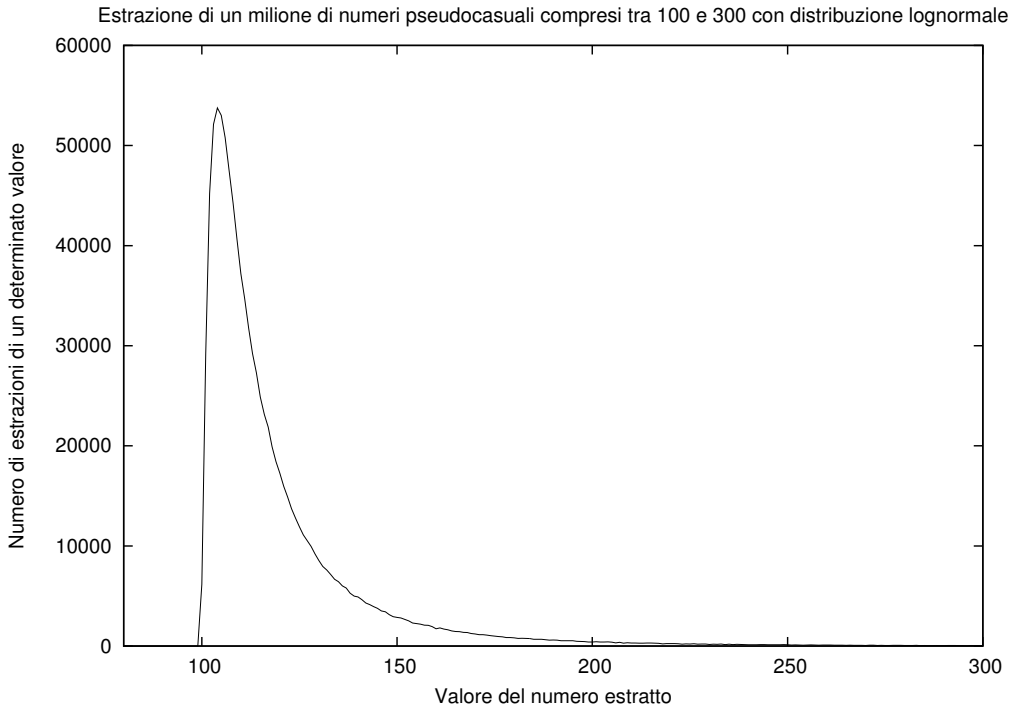

Figura 6.4: Risultati sperimentali ottenuti da un test del generatore pseudocasuale lognormale: è stata generata una sequenza di un milione di numeri pseudocasuali con distribuzione lognormale (parametri $\zeta = 2.5$ e $\sigma^2 = 1$), compresi tra gli estremi 100 e 300.

6.2 Ambiente di test

Il layer GAIAFaultTolerance implementa un sistema di ridondanza delle entità, che permette a GAIA di fare a meno del contributo di un numero prefissato di LP colpiti da crash o guasti bizantini, senza perdere nessun dato utile per la simulazione.

In altre parole, in una simulazione GAIAFT se un LP viene colpito da un guasto, GAIA può considerare guasto quel LP e continuare la simulazione senza di esso, con la garanzia di ottenere al termine della simulazione gli stessi risultati che otterrebbe in assenza di LP guasti.

Il layer GAIAFT però si limita all'implementazione dei sistemi per la ridondanza delle entità, per la gestione delle entità ridondanti, e per filtrare i messaggi provenienti da LP guasti (cap. 5).

GAIAFT quindi non gestisce la rimozione di un LP guasto dal sistema distribuito, ma si occupa solamente di mettere GAIA nella condizione di poter rimuovere tale LP, senza che ci siano perdite di entità nella simulazione.

Questo significa che l'implementazione degli algoritmi per rimuovere un LP guasto dal sistema deve essere parte integrante di GAIA. L'attuale implementazione di GAIA però non permette la rimozione "a caldo" di uno o più LP, non permettendo il corretto funzionamento di una simulazione GAIA/GAIAFT nel caso in cui avvengano dei guasti reali. Questo perché, prima di progettare modifiche al codice di GAIA, è stato considerato opportuno verificare l'effettiva utilità del layer GAIAFT, tramite la fase di valutazione delle prestazioni trattata in questo capitolo.

Il corretto funzionamento del layer GAIAFT è stato quindi testato eseguendo dei crash e dei guasti bizantini simulati, ad esempio bloccando l'invio di ogni messaggio da parte di un certo numero di LP per simularne i relativi crash, oppure inserendo dati errati (verosimili e non) nei messaggi inviati da un certo numero di LP per simularne i relativi guasti bizantini.

Per testare l'influenza dell'introduzione di GAIAFT sulle prestazioni di una simulazione GAIA, è stato eseguito un vasto insieme di test allo scopo di quantificare le variazioni prestazionali al variare di alcuni parametri di simulazione. Tra questi parametri (sez. 6.2.2), sono stati testati in particolare i seguenti: diverse quantità di entità

simulate e LP, sistema di tolleranza ai guasti attivato o disattivato, tolleranza a crash o guasti bizantini, e migrazioni di entità attivate o disattivate.

6.2.1 Infrastruttura utilizzata

L'infrastruttura hardware su cui sono stati eseguiti tutti i test è il cluster di macchine del laboratorio GNU/Linux del Dipartimento di Scienze dell'Informazione dell'Università di Bologna.

I test sono stati effettuati eseguendo simulazioni distribuite composte da un certo numero di LP, e ogni LP è stato eseguito su una singola macchina riservata ad esso (ad eccezione di tre test specificati in seguito, in cui sono stati eseguiti 2 e 4 LP per ogni host). Oltre alle macchine riservate agli LP, ne è stata usata una ulteriore per eseguire il processo SIMA[4].

Per ridurre al minimo eventuali interferenze nei risultati dovute all'uso delle macchine del laboratorio da parte di altri utenti, tutti i test sono stati effettuati unicamente di notte tramite sessioni Secure Shell (SSH), al di fuori degli orari di apertura del laboratorio.

Oltre a questo, per ogni test sono state effettuate 15 ripetizioni, dalle quali sono state estrapolate le medie dei risultati, comprensive di intervalli di confidenza (sez. 6.2.2).

A causa dell'alta quantità di tempo richiesto dai test e degli orari in cui essi sono stati eseguiti, è stato necessario automatizzare le procedure di test tramite l'uso di script.

Più precisamente, lo script sviluppato a tale scopo dispone di un file su cui è memorizzata la lista dei parametri delle simulazioni da eseguire. Per ogni singolo test su questa lista lo script esegue 15 ripetizioni, e per ogni ripetizione cambia il seme random iniziale[5].

Le singole ripetizioni vengono eseguite dallo script seguendo questa procedura:

[4]Il Simulation Manager (SIMA) è uno dei componenti di GAIA: un server di supporto che gestisce le fasi di inizializzazione della simulazione (sez. 2.1.1).

[5]Ogni test viene effettuato a partire da un certo seme random iniziale, tramite il quale vengono poi scelti, in un insieme di semi certificati, tutti i semi utilizzati in ogni simulazione (sez. 6.2.2).

- lo script invia a una macchina del laboratorio, tramite SSH File Transfer Protocol (SFTP), l'insieme di parametri della ripetizione corrente, tra cui i parametri di simulazione e il seme random iniziale specifico per tale ripetizione;
- lo script lancia la prima sessione SSH su una macchina del laboratorio, nella quale viene avviato il processo SIMA;
- per ogni LP della simulazione, lo script lancia una sessione SSH su una macchina dedicata a tale LP, avviando su tutte le macchine coinvolte i rispettivi processi LP, e dando così inizio alla simulazione;
- mentre la simulazione viene eseguita, lo script periodicamente ne sonda la terminazione tramite SFTP;
- quando lo script rileva la terminazione della simulazione, preleva via SFTP i risultati prodotti, li memorizza localmente e ricomincia una nuova ripetizione;
- se la ripetizione appena terminata è l'ultima del test corrente, lo script raggruppa i dati ottenuti da tutte le ripetizioni del test corrente, e passa al test successivo.

6.2.2 Parametri delle simulazioni e misurazioni effettuate

L'insieme dei test effettuati consiste nell'esecuzione di simulazioni caratterizzate da varie combinazioni di parametri, al fine di quantificare l'impatto prestazionale della variazione dei valori di questi parametri.

Gli insiemi di parametri che sono stati testati sono i seguenti:

- {4000, 8000, 12000, 16000, 20000} entità simulate, su simulazioni da 10000 timestep e distribuite su {3, 4, 5} LP;
- {2000, 4000, 8000, 16000} entità simulate, su simulazioni da 4000 timestep e distribuite su {4, 8} LP;
- {2000, 4000, 8000, 16000} entità simulate, su simulazioni da 4000 timestep, distribuite su 8 LP (2 LP per host) e su 16 LP (4 LP per host);

- {2000, 4000, 8000, 16000} entità simulate, su simulazioni da 4000 timestep, distribuite su 4 LP e con migrazione attivata;

- {2000, 6000} entità simulate, su simulazioni da 10000 timestep, distribuite su 5 LP, tolleranti a {0, 1, 2} guasti;

- {2000, 6000} entità simulate, su simulazioni da 2000 timestep, distribuite su 8 LP (2 LP per host), tolleranti a {0, 1, 2, 3} guasti.

Ognuno dei test elencati è stato eseguito con tutti i tipi di tolleranza ai guasti forniti da GAIAFT: tolleranza disattivata, tolleranza ai crash e tolleranza ai guasti bizantini.

Wall Clock Time

La misurazione effettuata durante i test per quantificare le prestazioni delle simulazioni è il Wall Clock Time (WCT), cioè il tempo reale necessario per portare a termine una simulazione.

Per ridurre l'influenza della fase transiente di inizializzazione della simulazione, il Wall Clock Time di un test viene misurato a partire dal primo timestep elaborato dal simulatore. Il tempo misurato va quindi dal primo all'ultimo timestep di simulazione, saltando la fase di inizializzazione precedente al primo timestep.

Per fornire dati più accurati, ogni test viene ripetuto 15 volte, determinando la presenza di 15 differenti misurazioni del Wall Clock Time per ogni combinazione di parametri testati. Su queste misurazioni vengono poi calcolati media campionaria e intervalli di confidenza.

Per ottenere la media campionaria e gli intervalli di confidenza, è stato sviluppato un programma ausiliario che calcola tali dati a partire dalle 15 misurazioni del Wall Clock Time ottenute in ogni test.

Il programma è tarato per calcolare media e intervalli di confidenza al 99,5% dei risultati di 15 ripetizioni di un test [32], utilizzando quindi il 14° quantile della distribuzione *t di Student* $\mathbf{t}_{14} = 2,97684$ [33, 35, 36].

Semi random certificati

Ognuna delle 15 ripetizioni di un test viene eseguita a partire da un differente seme random iniziale. Questo seme random però non costituisce il vero seme utilizzato dal generatore di numeri pseudocasuali di GAIA. Ogni entità simulata in GAIAFT infatti richiede un proprio seme personale (sez. 3.2.2), e tutti i semi utilizzati nella simulazione devono essere scelti da un file di semi certificati.

Il seme iniziale invece è solo un normale numero che viene preso in input dagli LP. Gli LP poi provvederanno a usare questo seme per generare degli indici casuali, con cui scegliere i veri semi certificati all'interno dell'apposito file.

6.3 Risultati dei test prestazionali

Sono stati effettuati vari test per confrontare le prestazioni delle simulazioni GAIAFT in un insieme di possibili scenari differenti. In particolare, per ogni test sono state confrontate le prestazioni dei casi con tolleranza disattivata (corrispondenti cioè alle prestazioni di GAIA senza l'utilizzo di GAIAFT), con tolleranza ai crash, e con tolleranza ai guasti bizantini. Tranne i test delle prestazioni al variare del numero di guasti tollerati (sez. 6.3.4), tutti i test effettuati con tolleranza attivata sono stati eseguiti con tolleranza a un guasto (un crash o un guasto bizantino a seconda dei casi).

L'analisi delle prestazioni negli scenari testati è stata effettuata confrontando i rispettivi WCT (Wall Clock Time) necessari per il completamento delle simulazioni.

6.3.1 WCT al variare del numero di entità

Sono state testate le prestazioni di una simulazione da 10000 timestep al variare del numero di entità. Questo test è stato ripetuto tre volte distribuendo la simulazione rispettivamente su 3, 4 e 5 LP.

In fig. 6.5 si può osservare il confronto tra tutti i test effettuati. Si può osservare che i casi 3 LP e 4 LP hanno prodotto risultati simili per tutte le quantità di entità testate, con prestazioni leggermente migliori nel caso dei 4 LP. Al contrario, nel caso 5 LP le

prestazioni delle simulazioni sono particolarmente degradate, considerando che a parte il numero di LP, gli altri parametri di simulazione sono gli stessi dei casi 3 LP e 4 LP.

Osservando il comportamento delle differenti modalità di tolleranza ai guasti, si può vedere che per tutti i tre casi (3, 4, 5 LP) esiste un rapporto più o meno costante tra i WCT relativi a tolleranza disattivata, tolleranza ai crash e tolleranza ai guasti bizantini (comportamento evidenziato per ogni caso nei grafici 6.6, 6.7 e 6.8). Questo rapporto è proporzionale al numero di messaggi ridondanti scambiati dalle copie delle entità nei casi di tolleranza a crash e di tolleranza a guasti bizantini (sez. 3.2.3).

Nei grafici 6.6, 6.7 e 6.8 inoltre si può osservare più dettagliatamente l'influenza del numero di entità (4000, 8000, 12000, 16000, 20000 entità) sulle prestazioni della simulazione.

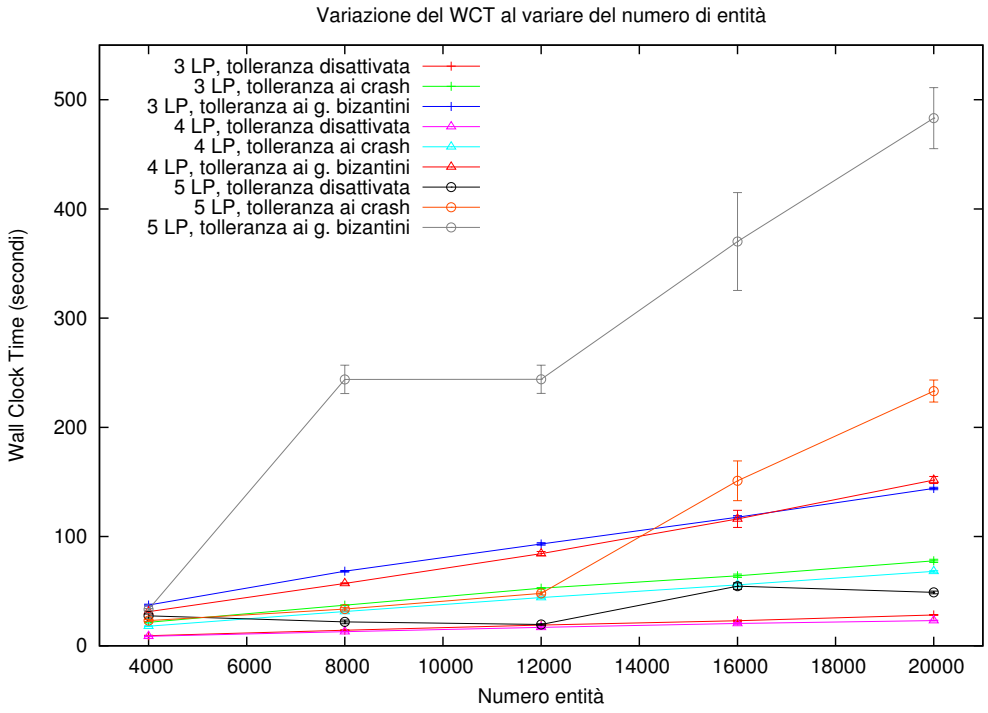

Figura 6.5: Confronto tra i risultati dei test di simulazioni distribuite su 3, 4 e 5 LP, al variare del numero di entità simulate e del metodo di tolleranza ai guasti scelto. Gli intervalli di confidenza sono calcolati al 99,5%.

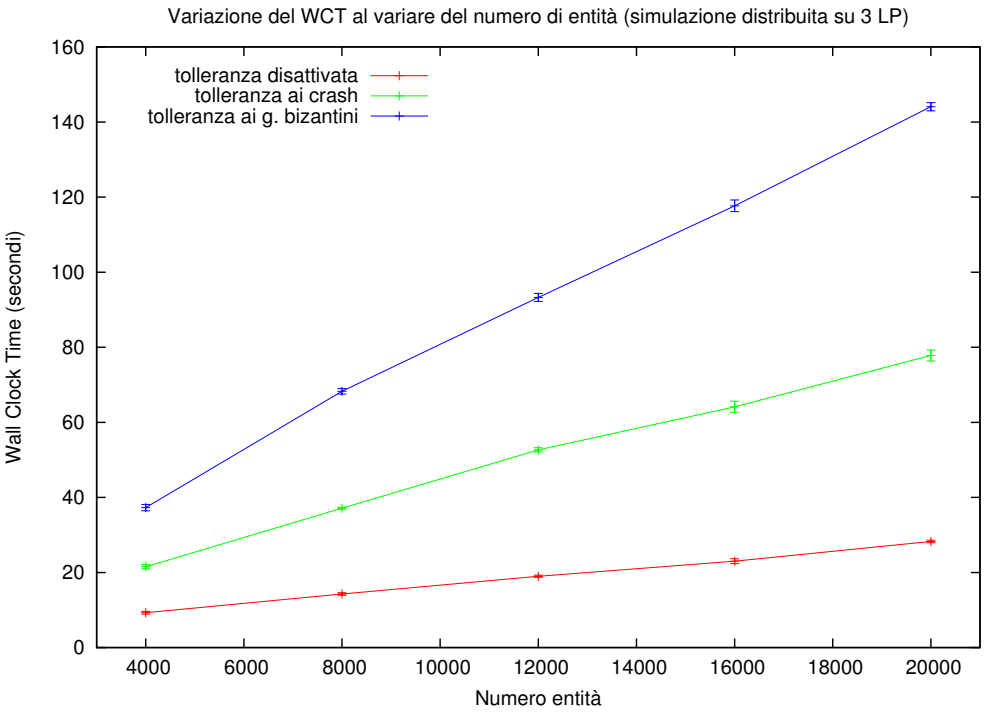

Figura 6.6: Confronto tra differenti metodi di tolleranza ai guasti e differenti quantità di entità simulate, in simulazioni distribuite su 3 LP. Si può osservare la somiglianza dei risultati con quelli misurati nel caso 4 LP. Si può inoltre osservare l'influenza del numero di entità simulate sulla durata delle simulazioni. Gli intervalli di confidenza sono calcolati al 99,5%.

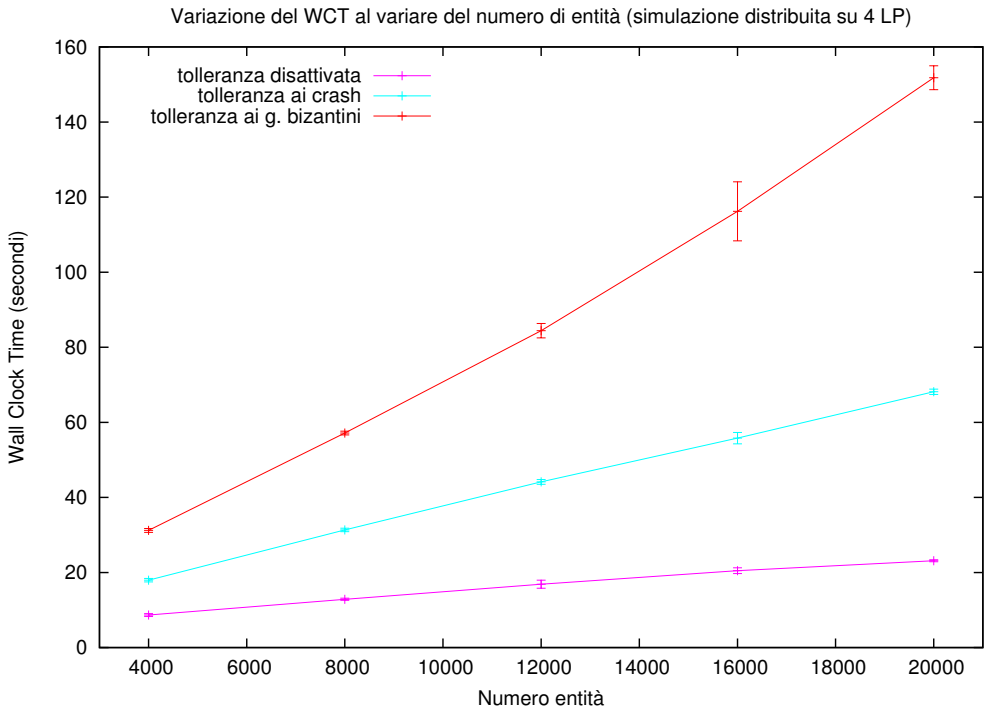

Figura 6.7: Confronto tra differenti metodi di tolleranza ai guasti e differenti quantità di entità simulate, in simulazioni distribuite su 4 LP. Si può osservare la somiglianza dei risultati con quelli misurati nel caso 3 LP. Si può inoltre osservare l'influenza del numero di entità simulate sulla durata delle simulazioni. Gli intervalli di confidenza sono calcolati al 99,5%.

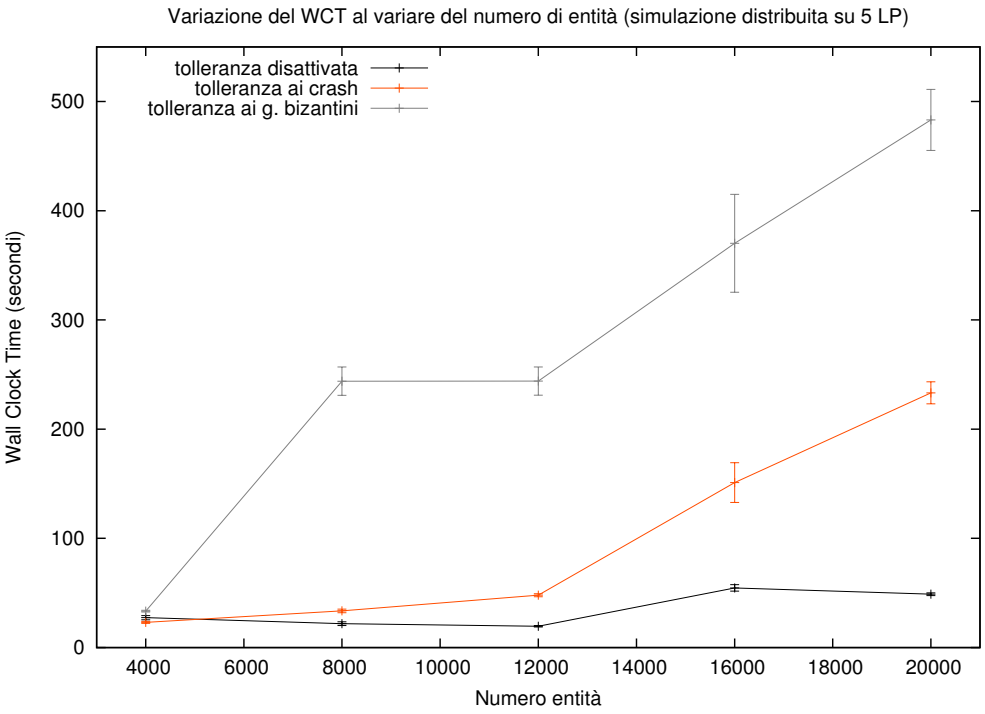

Figura 6.8: Confronto tra differenti metodi di tolleranza ai guasti e differenti quantità di entità simulate, in simulazioni distribuite su 5 LP. Si possono osservare le prestazioni degradate rispetto ai casi 3 e 4 LP. Nonostante questo, si può comunque osservare che il rapporto tra le prestazioni dei tre differenti metodi di tolleranza (disattivata, crash, guasti bizantini), è simile a quello osservabile nei casi 3 e 4 LP. Si può infine osservare l'influenza del numero di entità simulate sulla durata delle simulazioni. Gli intervalli di confidenza sono calcolati al 99,5%.

6.3.2 WCT al variare del numero di LP

In questa sezione vengono analizzati da un punto di vista differente gli stessi risultati dei test analizzati nella sezione precedente (sez. 6.3.1).

Nei due grafici seguenti (fig. 6.9 e 6.10) è possibile osservare le variazioni di WCT al variare del numero di LP.

Ognuno dei due grafici si riferisce a una certa quantità di entità simulate: nel primo grafico è osservabile il comportamento di una simulazione da 8000 entità, mentre nel secondo grafico è osservabile il comportamento di una simulazione da 16000 entità.

È possibile osservare un comportamento simile in entrambi i grafici: le prestazioni dei casi 3 e 4 LP sono simili tra loro e in media leggermente inferiori nel caso 4 LP, mentre nel caso 5 LP le prestazioni degradano e il WCT si alza notevolmente.

In particolare le prestazioni degradano per un numero di entità maggiore (caso delle 16000 entità) e con attivata la tolleranza ai guasti bizantini. Nel caso opposto infatti (8000 entità con tolleranza disattivata o tolleranza ai crash) le prestazioni rimangono stabili anche nel caso dei 5 LP.

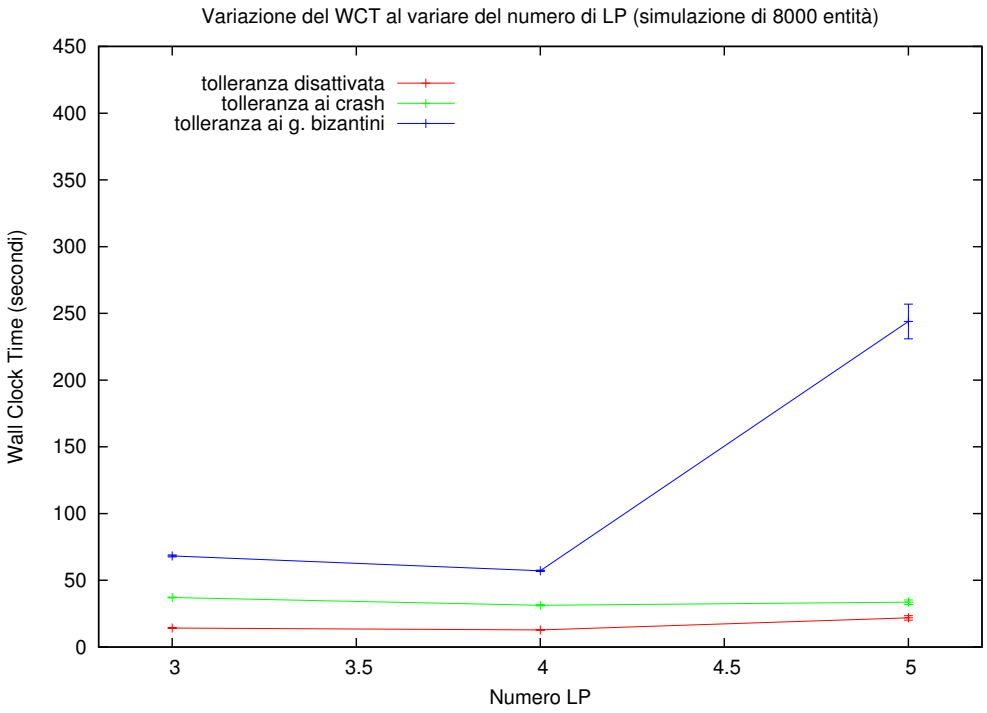

Figura 6.9: Confronto tra i WCT di simulazioni distribuite su differenti quantità di LP. Tutti i test visualizzati in questo grafico si riferiscono a simulazioni da 8000 entità. Si può osservare che nei casi di tolleranza disattivata e tolleranza ai crash le prestazioni rimangono stabili per 3, 4 e 5 LP, mentre nel caso dei guasti bizantini le prestazioni degradano quando la simulazione viene distribuita su 5 LP. Gli intervalli di confidenza sono calcolati al 99,5%.

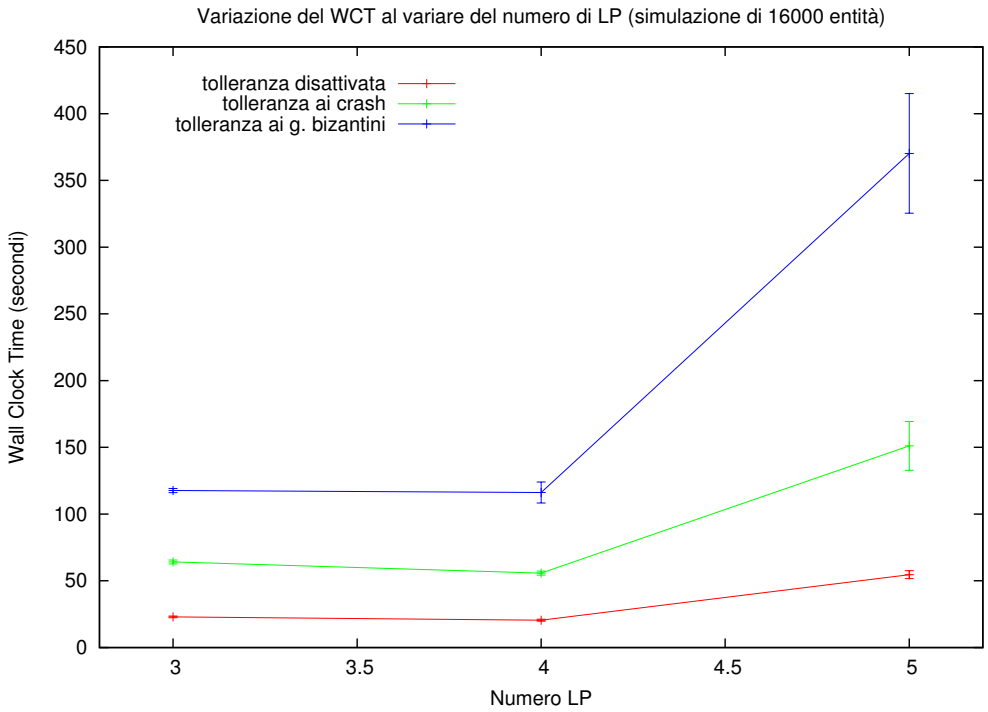

Figura 6.10: Confronto tra i WCT di simulazioni distribuite su differenti quantità di LP. Tutti i test visualizzati in questo grafico si riferiscono a simulazioni da 16000 entità. Si può osservare che le prestazioni rimangono stabili per 3 e 4 LP, mentre degradano nel caso dei 5 LP, per tutti i tre metodi di tolleranza testati. Gli intervalli di confidenza sono calcolati al 99,5%.

6.3.3 Variazioni del numero di LP eseguiti su ogni host

Nella sezione precedente è stato osservato che un numero di LP maggiore di 5 determina un notevole degrado delle prestazioni. I test trattati però si riferivano tutti al caso in cui ogni host esegue un singolo LP.

In questa sezione vengono confrontate le prestazioni di simulazioni distribuite su un numero maggiore di LP, in cui però può essere eseguito più di un LP su un singolo host. A causa di quest'ultimo parametro infatti le prestazioni potrebbero variare, perché un numero di LP maggiore per ogni host significa un carico di lavoro maggiore per host, ma anche una riduzione dell'overhead della comunicazione via rete. LP che si trovano sullo stesso host infatti possono comunicare tramite memoria condivisa.

Le simulazioni che vengono eseguite in questa serie di test prevedono tutte una durata di 4000 timestep e la presenza di 2000, 4000, 8000 e 16000 entità simulate, rispettivamente nei casi di tolleranza disattivata, tolleranza ai crash e tolleranza ai guasti bizantini.

Il numero di LP su cui vengono distribuite tali simulazioni e la quantità di LP eseguiti su ogni host sono i seguenti:

- 4 LP su 4 host (1 LP per host);
- 8 LP su 8 host (1 LP per host);
- 8 LP su 4 host (2 LP per host);
- 16 LP su 4 host (4 LP per host).

Nei grafici in fig. 6.11 si può vedere un confronto tra tutti i test effettuati. Si può osservare che le prestazioni del caso 4 LP sono notevolmente migliori di quelle degli altri casi esaminati. Queste seguono inoltre l'andamento crescente lineare osservato nei test della sezione precedente, causato dall'influenza sulle prestazioni del numero di entità simulate e del metodo di tolleranza ai guasti scelto.

Riguardo agli altri casi, si può vedere che l'influenza sulle prestazioni di numero di entità e metodo di tolleranza ai guasti è praticamente nulla. Le prestazioni in questi casi sono infatti determinate quasi unicamente dal numero di LP, che determina un degrado

delle prestazioni tale da minimizzare l'influenza delle altre differenze tra i casi testati.

Si possono comunque osservare prestazioni leggermente migliori nel caso 8 LP su 4 host, rispetto a quelle del caso 8 LP su 8 host. Questo leggero miglioramento è dovuto al fatto che due LP sullo stesso host possono comunicare in modo molto più efficiente, tramite memoria condivisa.

Nel caso 16 LP su 4 host invece, nonostante il numero di host sia limitato a 4, è osservabile un grave degrado delle prestazioni, notevolmente superiore a quello dei casi 8 LP su 4 host e 8 LP su 8 host. Da questa osservazione si può dedurre che il leggero miglioramento dovuto all'utilizzo della memoria condivisa, non è comunque sufficiente a contrastare il degrado delle prestazioni causato dall'alto numero di LP.

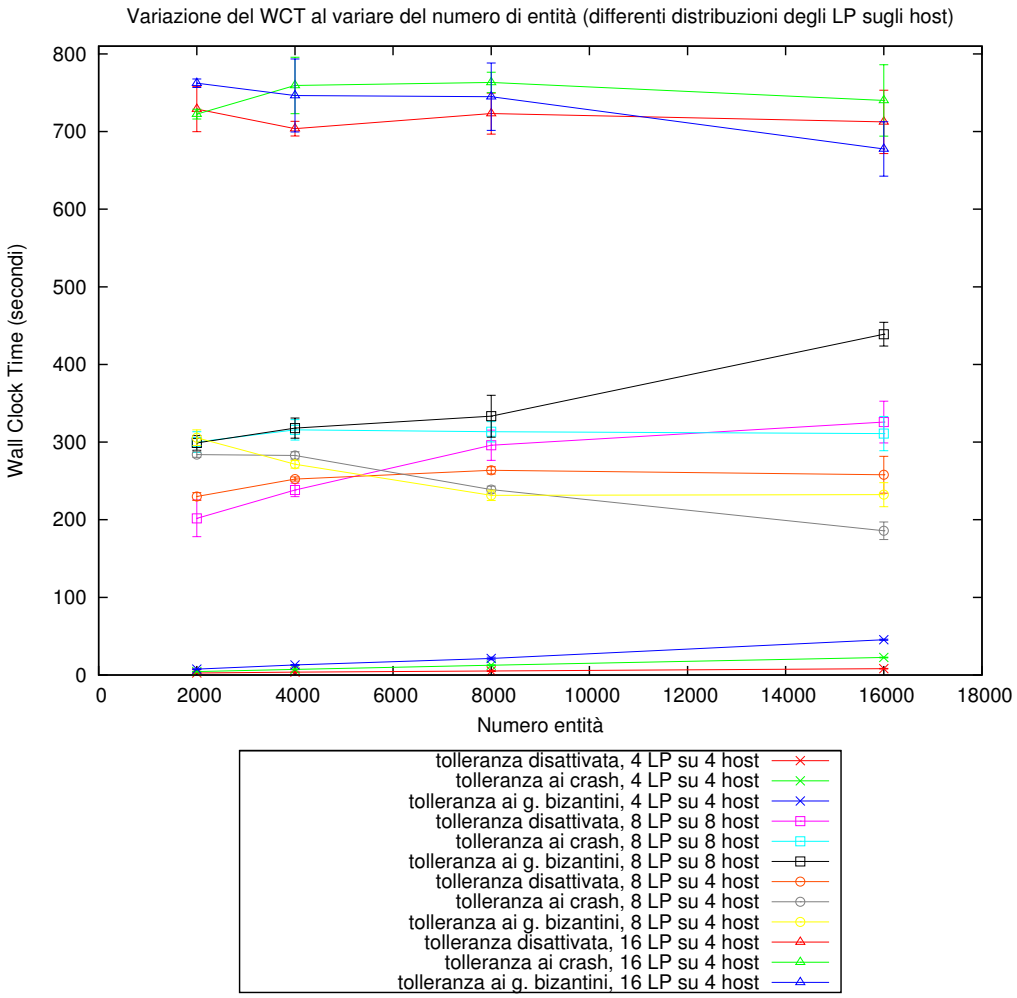

Figura 6.11: Confronto tra i WCT di simulazioni distribuite su differenti quantità di LP, e differenti quantità di LP per host. Si può osservare che le prestazioni del caso 4 LP su 4 host sono influenzate da numero di entità e metodo di tolleranza ai guasti, mentre le prestazioni degli altri casi sono influenzate principalmente dal numero di LP. Si possono notare prestazioni leggermente migliori nel caso 8 LP su 4 host rispetto a quelle del caso 8 LP su 8 host. Si può notare infine il notevole degrado delle prestazioni nel caso 16 LP su 4 host. Gli intervalli di confidenza sono calcolati al 99,5%.

6.3.4 WCT al variare del numero di guasti tollerati

In questa sezione viene analizzata l'influenza sulle prestazioni del numero di guasti tollerati da GAIAFT. Per questa analisi sono state effettuate due serie di test distinte: una distribuita su 5 LP (eseguiti su 5 host) e una distribuita su 8 LP (eseguiti su 4 host).

Come visto nelle sezioni precedenti, un numero di LP maggiore di 4 generalmente causa un degrado delle prestazioni che aumenta notevolmente all'aumentare degli LP. Questo significa che, come osservato precedentemente, più è alto il numero degli LP, minore è l'influenza sulle prestazioni degli altri parametri, come il numero di entità, la tolleranza ai guasti scelta e il numero di guasti tollerato.

La scelta di eseguire la prima serie di test su 5 LP è dovuta al fatto che 5 è il numero minimo di LP per poter tollerare fino a 2 guasti bizantini, ma è anche il numero massimo di LP che non determini un eccessivo degrado delle prestazioni. Un test su 5 LP rappresenta quindi un buon compromesso per osservare differenti valori del numero di guasti tollerati, senza che il numero di LP influisca eccessivamente.

La seconda serie di test su 8 LP invece è stata eseguita per poter testare anche il caso di tolleranza a 3 guasti. Per ridurre l'overhead dovuto alla comunicazione su rete, gli 8 LP sono stati eseguiti su 4 host. In altre parole, ogni host ha eseguito due LP invece di uno, similmente a quanto visto nei test della sezione precedente.

Il test effettuato su 5 LP (fig. 6.12) comprende simulazioni da 10000 timestep, caratterizzate da 2000 e 6000 entità simulate. I metodi di tolleranza testati sono tolleranza disattivata (nel grafico 6.12 corrispondente a 0 guasti tollerati), tolleranza a 1 e 2 crash, e tolleranza a 1 e 2 guasti bizantini. In questo test si può osservare che all'aumentare del numero di entità le prestazioni rimangono buone nel caso di tolleranza a un guasto, mentre nel caso di tolleranza a due guasti il tempo richiesto per terminare le simulazioni aumenta notevolmente. Questo accade in particolare con la tolleranza ai guasti bizantini, che prevede un numero maggiore di copie di entità, un conseguente numero maggiore di messaggi scambiati tra entità (sez. 3.2.3), e un tempo maggiore per gestire la ricezione di ogni singolo messaggio (sez. 3.2.4).

Il test effettuato su 8 LP (fig. 6.13) comprende simulazioni da 2000 timestep, carat-

terizzate da 2000 e 6000 entità simulate. I metodi di tolleranza testati sono tolleranza disattivata (nel grafico 6.13 corrispondente a 0 guasti tollerati), tolleranza a 1, 2 e 3 crash, e tolleranza a 1, 2 e 3 guasti bizantini. Nei grafici relativi a questo test si può osservare un andamento generale leggermente crescente all'aumentare del numero di entità. Come già visto nella sezione precedente però, in una simulazione distribuita su 8 LP le prestazioni degradano notevolmente, anche nel caso in cui vengano eseguiti 2 LP su ogni host. A causa di questo degrado delle prestazioni, si può osservare nei grafici che l'influenza sul WCT di numero di entità e metodo di tolleranza è minima, al punto da far ottenere risultati simili in casi molto differenti tra loro. Ad esempio, il caso 2000 entità e tolleranza a un crash consiste in una simulazione GAIA da 4000 entità, mentre il caso 6000 entità e tolleranza a tre guasti bizantini consiste in una simulazione GAIA da 14000 entità. Nonostante questa differenza, i risultati ottenuti in questi due test sono simili, proporzionalmente alle differenze esaminate negli altri test visti precedentemente.

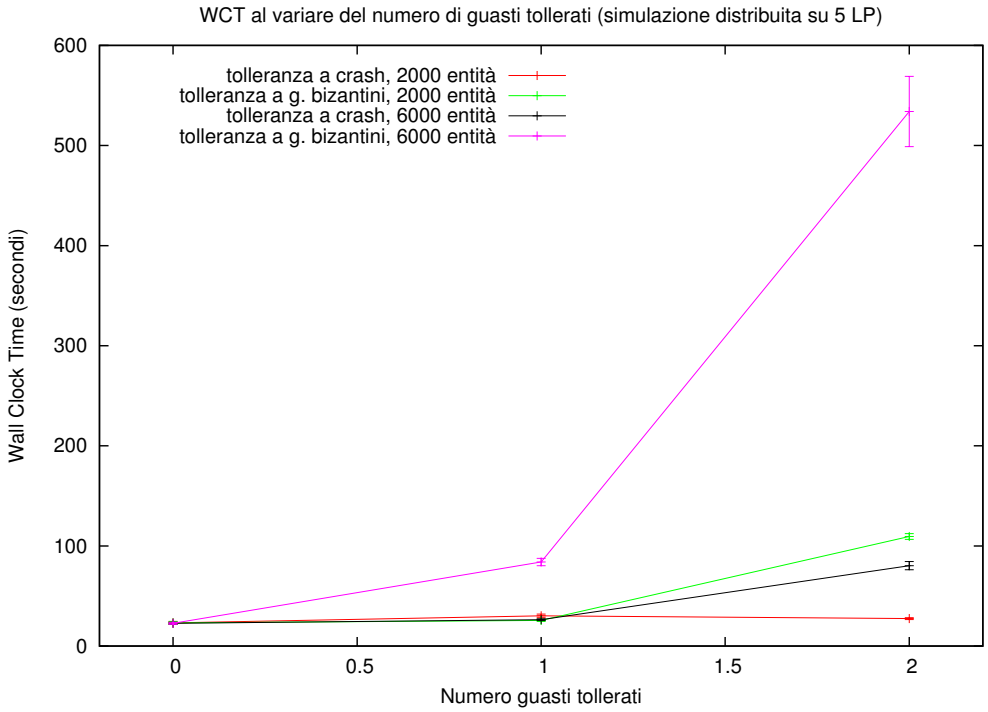

Figura 6.12: Confronto tra differenti quantità di guasti tollerati su una simulazione da 10000 timestep, distribuita su 5 LP. Le quantità di guasti tollerati in questa serie di test sono: 0 guasti (tolleranza disattivata), $\{1, 2\}$ crash, e $\{1, 2\}$ guasti bizantini. Gli intervalli di confidenza sono calcolati al 99,5%.

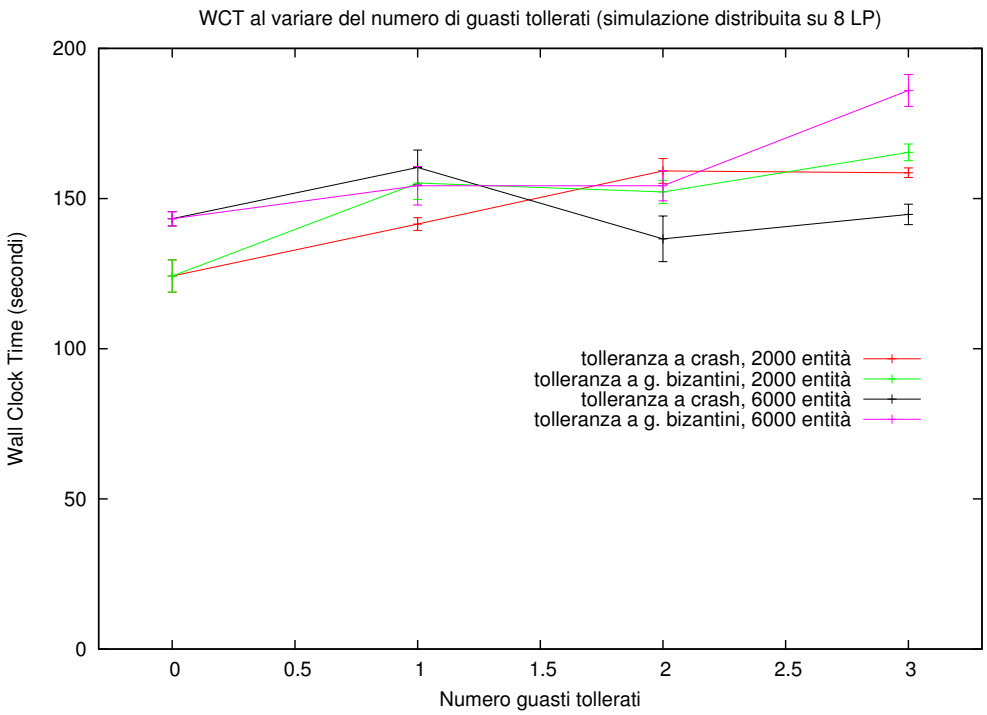

Figura 6.13: Confronto tra differenti quantità di guasti tollerati su una simulazione da 2000 timestep, distribuita su 8 LP. Le quantità di guasti tollerati in questa serie di test sono: 0 guasti (tolleranza disattivata), {1, 2, 3} crash, e {1, 2, 3} guasti bizantini. Gli intervalli di confidenza sono calcolati al 99,5%.

6.3.5 Migrazioni attivate e disattivate

Il layer GAIAFT supporta l'attivazione delle migrazioni di entità in GAIA. Le migrazioni in GAIA servono per modificare a tempo di esecuzione il collocamento delle entità sui vari LP, adattandolo alle condizioni dell'infrastruttura sottostante e alle relazioni tra le entità. In GAIAFT però, nonostante le migrazioni siano supportate (sez. 5.7), queste devono sottostare al vincolo per cui non possono essere presenti più copie di una stessa entità sullo stesso LP. In questa sezione verrà analizzata la differenza di prestazioni determinata dall'attivazione delle migrazioni di GAIA.

Nel grafico 6.14 si possono osservare le prestazioni di simulazioni da 4000 timestep distribuite su 4 LP, con 2000, 4000, 8000 e 16000 entità simulate. Tutti questi test sono stati effettuati sia con la migrazione disattivata, sia con la migrazione attivata.

Nel grafico si può osservare che mediamente l'attivazione delle migrazioni comporta l'introduzione di un overhead, che aumenta leggermente il WCT in ogni test effettuato. Si può vedere che questo overhead è molto ridotto nel caso di tolleranza disattivata, mentre aumenta nei casi di tolleranza ai crash e tolleranza ai guasti bizantini, pur rimanendo sempre basso, proporzionalmente al WCT del caso testato.

Vista la stabilità dell'infrastruttura di rete sottostante, l'adattabilità data dall'introduzione delle migrazioni non ha portato a vantaggi significativi nella riduzione del WCT, introducendo anzi un overhead. Questo overhead è comunque limitato, proporzionalmente al WCT misurato nei vari casi, non causando un particolare degrado delle prestazioni.

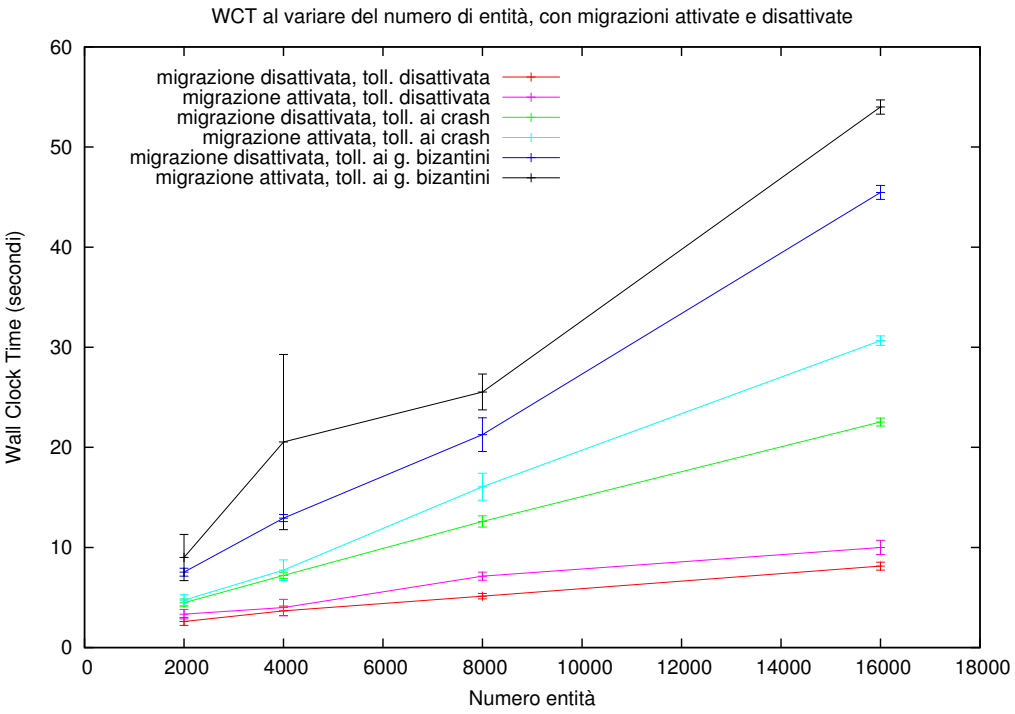

Figura 6.14: Confronto tra simulazioni con migrazioni disattivate e simulazioni con migrazioni attivati. Si può osservare che il comportamento delle simulazioni con migrazione è molto simile a quello delle simulazioni senza migrazione. Nonostante questo, nelle simulazioni con migrazione si può osservare un overhead dovuto alla gestione delle migrazioni da parte di GAIA e GAIAFT. Questo overhead rimane comunque basso in ogni test, proporzionalmente al WCT misurato. Gli intervalli di confidenza sono calcolati al 99,5%.

Conclusione e sviluppi futuri

L'obiettivo di questa tesi è stato la progettazione, l'implementazione e la valutazione di GAIAFaultTolerance, un layer software da includere nel framework per la simulazione parallela e distribuita GAIA/ARTÌS. Lo scopo di questo layer è aggiungere a GAIA/ARTÌS meccanismi di tolleranza ai guasti, basati sulla ridondanza dei dati delle simulazioni.

Più precisamente, le simulazioni distribuite basate sul framework GAIA/ARTÌS sono composte da un certo numero di Logical Process (LP), ognuno dei quali gestisce un insieme di entità simulate che interagiscono fra loro per mezzo di messaggi. Il layer GAIAFaultTolerance introduce meccanismi di ridondanza sulle entità, introducendo un certo numero di copie per ogni entità simulata. Questi meccanismi di ridondanza sulle entità servono a realizzare simulazioni tolleranti a un certo numero di crash o di guasti bizantini degli LP.

La tolleranza a N crash è stata ottenuta introducendo nelle simulazioni $N + 1$ copie per ogni entità, in modo che rimanga sempre almeno una copia viva per ogni entità. La tolleranza a N guasti bizantini invece è stata ottenuta introducendo nelle simulazioni $2N + 1$ copie per ogni entità. In questo modo, per N copie di una certa entità colpite da guasti bizantini, ne sono sempre presenti $N + 1$ sane, che formando una maggioranza possono essere prese come riferimento per stabilire il corretto comportamento di tale entità.

In GAIAFaultTolerance il tipo di guasti tollerati (crash o guasti bizantini) e il numero massimo di guasti tollerati sono parametri impostabili a piacimento all'inizio di una simulazione. L'unico vincolo su questi parametri è che il numero di copie per ogni entità richiesto dal metodo di tolleranza scelto non deve essere superiore al numero di LP. Questo perché, al fine di garantire il corretto funzionamento del sistema di tolleranza

ai guasti, è presente un vincolo in GAIAFaultTolerance per cui più copie di una stessa entità non devono mai trovarsi sullo stesso LP.

Il raggiungimento dell'obiettivo di questa tesi è stato ottenuto progettando e implementando il layer GAIAFaultTolerance sopra descritto, sviluppando un apposito ambiente di simulazione di prova in grado di simulare crash e guasti bizantini, ed eseguendo una serie di test per verificare il corretto funzionamento e l'impatto prestazionale di GAIAFaultTolerance. In fase di valutazione delle prestazioni, i risultati ottenuti si sono dimostrati coerenti con le aspettative iniziali, mostrando il corretto funzionamento di GAIAFaultTolerance e un impatto prestazionale in linea con quello ipotizzato in fase di progettazione.

Volendo proseguire lo sviluppo di GAIAFaultTolerance, la naturale prosecuzione consiste nell'implementazione di un insieme di modifiche al framework GAIA/ARTÌS, per permettere ad esso di sfruttare realmente le funzionalità introdotte da GAIAFaultTolerance. Tra queste modifiche, in particolare servirebbe introdurre un sistema per permettere a GAIA/ARTÌS di rimuovere LP guasti a tempo di esecuzione.

Questo perché il ruolo di GAIAFaultTolerance è quello di mettere GAIA/ARTÌS nella condizione di poter rimuovere dal sistema distribuito un LP guasto, senza che avvenga nessuna perdita nei dati della simulazione. L'attuale implementazione di GAIA/ARTÌS però non permette la rimozione a tempo di esecuzione di un LP guasto, anche se grazie alle funzionalità introdotte da GAIAFaultTolerance questa operazione potrebbe essere eseguita senza perdita di dati utili.

Inoltre, un'altra caratteristica di GAIA/ARTÌS che si potrebbe modificare è la gestione delle migrazioni. GAIA/ARTÌS infatti prevede la possibilità di far migrare le entità da un LP ad un altro per adattare la loro distribuzione all'infrastruttura sottostante, al fine di ottimizzare l'utilizzo da parte loro dei canali di comunicazione tra LP. GAIAFaultTolerance supporta l'attivazione delle migrazioni da parte di GAIA/ARTÌS, ma è emerso dai test che queste non hanno portato all'incremento prestazionale desiderato. Andrebbero quindi riviste le politiche con cui GAIA/ARTÌS decide di effettuate delle migrazioni, tenendo in considerazione il fatto che con GAIAFaultTolerance ogni entità simulata prevede la presenza di un certo numero di copie.

Bibliografia

[1] L. Donatiello, G. D'Angelo, L. Bononi, M. Bracuto. *Gruppo di ricerca sulla Simulazione Parallela e Distribuita*. Università di Bologna, Dipartimento di Scienze dell'Informazione. `http://pads.cs.unibo.it`.

[2] Gabriele D'Angelo, Michele Bracuto. *Distributed simulation of large-scale and detailed models*. Int. J. Simulation and Process Modelling, Vol. 5, No. 2, pp. 120–131.

[3] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

[4] L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM, 21:558–565, July 1978.

[5] Richard M. Fujimoto. *Parallel Discrete Event Simulation*. Proceedings of the 1989 Winter Simulation Conference: pp. 19–28, 1989.

[6] Richard M. Fujimoto. *Parallel and Distributed Simulation*. Proceedings of the 1999 Winter Simulation Conference: pp. 122–131, 1999.

[7] *IEEE Standard for Modeling and Simulation High Level Architecture (HLA) - Framework and Rules*. IEEE Std. 1516-2000, pp. i–22, 2000.

[8] Frederick Kuhl, Richard Weatherly, Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall PTR, 2000.

[9] J. Misra. *Distributed discrete event simulation*. ACM Computing Surveys, 18(1):39–65, 1986.

[10] Averill M. Law, W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 3rd edition, 1999.

[11] D. Jefferson. *Virtual Time*. ACM Transactions Program. Lang. Syst., 7(3):404–425, 1985.

[12] Tobias Kiesling. *Fault-Tolerant Distributed Simulation: A Position Paper*. Universität der Bundeswehr München, Institut für technische Informatik, July 29, 2003.

[13] C. Berchtold, M. Hezel. *An architecture for fault-tolerant HLA-based simulation*. In Proceedings of the 15th European Simulation Multiconference, 616–620. Prague, Czech Republic, 2001.

[14] Om. P. Damani, Vijay K. Garg. *Fault-Tolerant Distributed Simulation*. Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on , pp. 38–45, 26-29 May 1998.

[15] D. Agrawal, J. R. Agre. *Replicated objects in timewarp simulations*. In Proceedings of 1992 Winter Simulation Conference, pp. 657–664, 1992.

[16] M. Eklof, F. Moradi, R. Ayani. *A framework for fault-tolerance in HLA-based distributed simulations*. In Proceedings of 2005 Winter Simulation Conference, pp. 1182–1189, Dec. 2005.

[17] Bin Chen, Xiao-gang Qiu. *Implement Fault Tolerant in distributed DEVS simulation*. Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference on , vol.1, pp.135–138, 28-29 Nov. 2009.

[18] L. Bononi, M. Bracuto, G. D'Angelo, L. Donatiello. *Proximity detection in distributed simulation of wireless mobile systems*. Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems, pp. 44–51, 2006.

[19] Alberto Montresor and Márk Jelasity. *PeerSim: A scalable P2P simulator*. Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), pp. 99–100. Seattle, WA, September 2009.

[20] Grenville Armitage, Mark Claypool, Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, 2006.

[21] Johannes Faerber. *Network game traffic modelling*. Proceedings of the 1st ACM workshop on Network and System Support for games, April 2002.

[22] Jonathan Bentini. *Architettura distribuita scalabile per applicazioni interattive e multiutente su internet*. Master's thesis, Università di Bologna, Giugno 2003.

[23] John C. McEachen. *Traffic characteristics of an internet-based multiplayer online game*. In Fourth International Conference on Information, Communication & Signal Processing, 2003.

[24] Ludovico Gardenghi, Sandro Pifferi, Gabriele D'Angelo. *Design and evaluation of a migration-based architecture for massively populated Internet Games*. Technical Report UBLCS-04-2, March 2004.

[25] Larry L. Peterson, Bruce S. Davie. *Computer Network - A Systems Approach*. Morgan Kaufmann, second edition, 2000.

[26] G. M. Amdahl. *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS Conference Proceedings, 1967.

[27] Alan Bertossi. *Algoritmi e Strutture di Dati*. UTET, 2004.

[28] Ronald Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, 1992. `http://tools.ietf.org/html/rfc1321`.

[29] James E. Gentle. *Random Number Generation and Monte Carlo Methods, second edition*. Springer, 2003.

[30] PlanetMath.org. *Acceptance-Rejection Method*. http://planetmath.org/encyclopedia/AcceptanceRejectionMethod.html.

[31] Massimo Campanino, Francesca Biagini. *Elementi di Probabilità e Statistica*. Springer, 2006.

[32] Weisstein, Eric W. *Confidence Interval*. From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/ConfidenceInterval.html.

[33] Weisstein, Eric W. *Student's t-Distribution*. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Studentst-Distribution.html.

[34] Weisstein, Eric W. *Log Normal Distribution*. From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/LogNormalDistribution.html.

[35] Thomas Hill, Pawel Lewicki. *Statistics: Methods and Applications*. StatSoft, 2005.

[36] StatSoft. *Distribution Tables*. http://www.statsoft.com/textbook/sttable.html.

[37] GARR - The Italian Academic and Research Network. http://www.garr.it/garr-b-home-engl.shtml.

[38] Gerald Combs et al. *Ethereal: A network protocol analyzer*. http://www.ethereal.com.

[39] Van Jacobson, Craig Leres, Steven McCanne, et al. *Tcpdump*. http://www.tcpdump.org.