Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

MAS: Esperienza di utilizzo del framework ARTIS/GAIA

Relatore:
Gabriele D'Angelo

Presentata da:
Lorenzo Bergami

I Sessione di Laurea
Anno Accademico 2024/2025

Introduzione

Le simulazioni al computer rappresentano oggi uno strumento essenziale per lo studio di sistemi complessi. Grazie a esse, è possibile osservare fenomeni dinamici, formulare previsioni, testare ipotesi o valutare scenari che, per limiti pratici, economici o etici, risulterebbero difficili da esplorare nel mondo reale.

In particolare, le simulazioni a eventi discreti (Discrete Event Simulation, DES) costituiscono una classe consolidata di approcci simulativi, nei quali il sistema evolutivo viene descritto come una successione ordinata di eventi nel tempo. Per simulazioni di larga scala, l'interesse della comunità scientifica si è progressivamente orientato verso l'impiego di tecniche di esecuzione parallela e distribuita, che consentono di ridurre i tempi di elaborazione e sfruttare in modo efficiente risorse computazionali eterogenee.

L'ambito delle *Parallel and Distributed Simulation* (PADS) si concentra proprio su questi aspetti, proponendo metodologie e strumenti per garantire che l'esecuzione distribuita mantenga coerenza logica e correttezza rispetto al comportamento di riferimento.

In questo contesto si inserisce il framework *ARTÌS/GAIA*, una piattaforma per la simulazione distribuita e adattiva sviluppata all'interno dell'Università di Bologna. Pur essendo stato impiegato in progetti di ricerca, esso è scarsamente documentato nella letteratura italiana, ed è quindi apparso interessante studiarlo e testarne le potenzialità in un contesto sperimentale.

Il lavoro presentato in questa tesi prende avvio da un'attività di tirocinio curricolare precedente, che ha offerto un primo contatto operativo con il

framework ARTÌS/GAIA e con la simulazione di sistemi multi-agente. La tesi ne rappresenta una naturale prosecuzione e approfondimento, con l'obiettivo di analizzare in modo sistematico le caratteristiche del framework, esplorarne le potenzialità attraverso modelli dinamici di crescente complessità, valutarne il comportamento tramite strumenti di validazione empirica, e infine verificarne la flessibilità mediante il porting di un modello preesistente originariamente sviluppato in Python.

Il lavoro si apre con un'introduzione teorica ai principali concetti legati alla simulazione discreta, alla parallelizzazione (PADS) e ai sistemi multi-agente. Segue un'analisi delle librerie ARTÌS/GAIA, quindi una descrizione delle attività di sviluppo, testing e validazione dei modelli, concludendo con il porting di un modello originariamente scritto in Python.

Indice

Introduzione **i**

1 Background Teorico **1**

1.1 Simulazione 1

1.2 Simulazione a Eventi Discreti (DES) 3

1.3 MAS, ABS e simulazione parallela e distribuita (PADS) 4

1.4 ARTÌS/GAIA 6

2 Sviluppare con ARTIS/GAIA **1**

2.1 Studio del framework 1

2.2 Dinamiche dei modelli 3

2.3 Modelli sviluppati 6

2.4 Criticità 11

3 Testing e Correzione dei modelli **1**

3.1 Risoluzione degli identificatori 1

3.2 Visualizzazione della simulazione 2

3.3 Validatore 4

3.4 Errori di validazione riscontrati 6

3.5 Controllo sui messaggi 7

3.6 Robustezza e gestione della memoria 9

3.7 Metriche di comunicazione: Local e Global Communication Ratio 11

4 Porting **1**

4.1 Descrizione del modello 2

4.2 Implementazione in ARTÌS/GAIA 3

4.3 Differenze tra implementazione originale e porting 5

Conclusioni **11**

Appendice **13**

Bibliografia **19**

Elenco delle figure

1.1 Struttura concettuale dei livelli software ARTÌS e GAIA. . . . 8

2.1 Architettura esecutiva ARTÌS: il SiMa gestisce l'inizializzazione e sincronizzazione, mentre i Logical Processes (LPs) gestiscono entità locali e globali, comunicando direttamente tra loro durante l'esecuzione. 3

2.2 Evoluzione genetica della popolazione nel piano fertilità–immunità con parametri fine-tuned. I punti rappresentano entità vive; il colore identifica l'LP di appartenenza. 10

3.1 Evoluzione della simulazione al timestep 100 e 156. Gli agenti sani sono rappresentati in blu, quelli infetti in rosso. È possibile osservare l'espansione dell'infezione e la nascita di nuove entità. 3

3.2 Evoluzione della simulazione al timestep 224: l'infezione si è diffusa maggiormente. La visualizzazione consente di identificare facilmente la distribuzione spaziale delle entità e lo stato di salute della popolazione simulata. 4

3.3 Comunicazione locale/remota nel caso base: riproduzione e morte disattivate. 13

3.4 Comunicazione nel caso completo: riproduzione e morte attive. 13

4.1 Confronto tra il modello originale Python (sopra) e la versione ARTÌS/GAIA (sotto). 7

Elenco delle tabelle

4.1 Esempio di parametri sperimentali utilizzati col modello `GIDRA`. 16
4.2 Messaggi personalizzati utilizzati nei modelli. 16

Capitolo 1

Background Teorico

In questa sezione affronteremo i prerequisiti teorici necessari per comprendere il progetto di tesi.

1.1 Simulazione

La simulazione è il processo attraverso il quale si costruisce un modello — fisico oppure virtuale — in grado di riprodurre, in modo più o meno accurato, il comportamento di un sistema reale. Lo scopo è quello di analizzare, comprendere o prevedere l'evoluzione temporale del sistema stesso in risposta a determinati stimoli o condizioni. Si tratta di un approccio ampiamente adottato in numerosi ambiti scientifici e industriali, in quanto consente di esplorare fenomeni complessi in ambienti controllati e ripetibili.

Nel contesto dell'Informatica, si parla più specificamente di simulazione al calcolatore (computer simulation) [4], ovvero l'utilizzo di un computer per rappresentare l'evoluzione dinamica di un sistema reale attraverso un suo modello matematico virtualizzato. Questo modello è formalizzato sotto forma di programma, in cui le variabili e le relazioni tra le componenti del sistema sono espresse mediante equazioni o regole logiche. Se le relazioni matematiche sono abbastanza semplici, può essere possibile ottenere informazioni esatte su quesiti di interesse; questa è chiamata soluzione analitica. Nella maggior

parte dei casi però, l'esecuzione del programma simula il comportamento del sistema originario fornendo un'approssimazione della sua evoluzione nel tempo.

Un buon modello simulativo deve essere in grado di rappresentare fedelmente le relazioni funzionali che regolano il sistema reale. In fase di simulazione, il computer risolve le equazioni del modello, producendo una sequenza di stati che descrivono l'evolversi del sistema virtuale. L'obiettivo non è sempre quello di ottenere una replica perfetta del fenomeno originale, ma spesso una rappresentazione sufficientemente realistica per consentire analisi predittive, valutazioni comparative o esperimenti in silico.

Uno dei principali vantaggi della simulazione è la possibilità di studiare sistemi complessi, costosi, pericolosi o inaccessibili. Ad esempio, è possibile simulare il comportamento di una centrale nucleare in condizioni critiche, l'evoluzione di un'epidemia in una popolazione, o l'interazione tra componenti meccanici in un motore. Nello specifico caso di questa tesi, la simulazione viene utilizzata per modellare scenari come la diffusione di un'infezione in un contesto sociale.

Inoltre, grazie alla natura virtuale del modello, la simulazione consente di esplorare il comportamento del sistema al variare di determinati parametri iniziali o strutturali, in modo sistematico e a basso costo computazionale. Questo approccio è particolarmente utile per svolgere analisi di sensibilità o per ottimizzare strategie di intervento.

Gli strumenti disponibili per realizzare simulazioni sono numerosi e variegati, spaziando da semplici fogli di calcolo a software specializzati, fino a sistemi altamente paralleli eseguiti su supercomputer. Le applicazioni spaziano dalla ricerca accademica all'industria, dalla biologia alla finanza, fino alla meteorologia e alla robotica. Nel seguito della tesi verranno approfondite alcune classi specifiche di simulazione, in particolare quelle a eventi discreti (DES), i sistemi multi-agente (MAS) e la loro realizzazione in ambienti paralleli e distribuiti (PADS), con un focus sul framework ARTIS/GAIA utilizzato per lo sviluppo e l'esecuzione dei modelli presentati.

1.2 Simulazione a Eventi Discreti (DES)

La *Simulazione a Eventi Discreti* (Discrete Event Simulation, DES) è una classe di simulazioni in cui l'evoluzione del sistema modellato avviene attraverso **eventi che si verificano in istanti di tempo distinti e separabili**. In altre parole, lo stato del sistema può cambiare solo in corrispondenza di eventi, i quali avvengono in momenti precisi lungo una linea temporale discreta.

A differenza di altri approcci simulativi che rappresentano il tempo come un flusso continuo, nelle DES il tempo è trattato come una semplice metrica per ordinare cronologicamente gli eventi. Il *tempo simulato* non scorre in modo continuo: avanza per **salti**, da un evento al successivo. Questo approccio è detto *event-driven*, poiché il motore della simulazione è costituito dagli eventi stessi. In assenza di eventi, la simulazione resta inattiva: il sistema non subisce modifiche e il tempo simulato può avanzare direttamente al prossimo evento significativo, riducendo la complessità computazionale e migliorando l'efficienza.

Grazie alla **discrezione temporale**, ogni evento può essere associato a un timestamp ben definito, permettendo l'ordinamento e la gestione efficiente degli eventi futuri. Questo rende naturale l'impiego di *strutture dati ordinate*, come una *event queue*, dove il prossimo evento può essere recuperato tramite una semplice operazione di `dequeue`, e nuovi eventi possono essere inseriti in ordine temporale.

Una simulazione DES, ad alto livello, si basa tipicamente su tre componenti fondamentali [1]:

- **Entità**: rappresentano gli elementi del sistema che interagiscono fra loro per realizzare un comportamento collettivo (es. macchine, persone, processi).

- **Stato del sistema**: è l'insieme delle variabili che descrivono la configurazione del sistema in un determinato istante del tempo simulato.

Nelle DES, queste variabili cambiano valore in corrispondenza degli eventi, in modo istantaneo.

- **Eventi**: sono le occorrenze che provocano potenziali cambiamenti nello stato del sistema. Ogni evento è legato a un istante di tempo e può determinare un aggiornamento delle variabili di stato. Alcuni eventi, tuttavia, possono non modificare lo stato, ma ad esempio servire solo a schedulare la fine della simulazione.

L'evoluzione temporale di una DES è governata da una variabile detta *simulation clock*, che tiene traccia del tempo simulato corrente. È importante sottolineare che il tempo simulato è del tutto indipendente dal tempo reale impiegato dal computer per eseguire la simulazione.

L'approccio più comune per l'avanzamento temporale è il *next-event time advance*, in cui il simulation clock salta direttamente al tempo del prossimo evento pianificato. Tuttavia, esistono anche approcci alternativi come il *fixed-increment time advance*, che procede a passi regolari indipendentemente dalla presenza di eventi. Quest'ultimo è più comune in contesti paralleli o distribuiti [13].

1.3 MAS, ABS e simulazione parallela e distribuita (PADS)

Le simulazioni *Agent-Based* (ABS) si basano su una rappresentazione in cui il sistema è composto da una popolazione di entità autonome chiamate *agenti*. Ogni agente mantiene un proprio stato interno che può evolvere nel tempo, tipicamente in risposta a eventi o interazioni con altri agenti. Il termine *Multi-Agent System* (MAS) indica una classe di sistemi in cui molteplici agenti cooperano, competono o interagiscono all'interno di un ambiente comune, dando luogo a dinamiche emergenti complesse.

Uno dei punti di forza dell'approccio MAS/ABS è la sua elevata **potenza espressiva**: consente di modellare con naturalezza fenomeni decentralizza-

ti, distribuiti o emergenti, come reti sociali, mercati economici, ecosistemi, epidemie o sistemi robotici. Le interazioni tra agenti sono spesso formalizzate come *messaggi* o scambi di informazione, che avvengono secondo regole predefinite ma potenzialmente stocastiche.

Tuttavia, la simulazione di sistemi multi-agente su larga scala può diventare molto onerosa dal punto di vista computazionale, a causa dell'elevato numero di agenti e della complessità delle interazioni. Per questo motivo, è prassi ormai consolidata sfruttare tecniche di **parallelizzazione** per aumentare l'efficienza e ridurre i tempi di simulazione. La parallelizzazione può avvenire secondo schemi differenti: tramite multithreading su CPU, parallelismo massivo su GPU o, nei casi più generali, tramite **parallelismo distribuito** su più nodi di calcolo.

Un aspetto critico di qualsiasi simulazione parallela è il **partizionamento del lavoro**: il sistema simulato deve essere suddiviso tra più unità di elaborazione (Logical Processes, LP). Questo compito può essere non banale nel caso dei MAS, specialmente quando le interazioni tra agenti sono *stocastiche* e variano nel tempo. In tali casi, un **partizionamento statico** (definito a priori) può risultare inefficiente, poiché non tiene conto della natura dinamica e irregolare del carico computazionale.

Inoltre, non tutti gli agenti richiedono la stessa quantità di risorse per essere simulati: alcuni possono essere più complessi di altri, o coinvolti in un numero maggiore di interazioni. Questo può generare **sbilanciamenti di carico** tra i vari LP. A ciò si aggiunge la necessità di gestire non solo i messaggi simulativi veri e propri (le interazioni tra agenti), ma anche messaggi di sistema per la sincronizzazione, l'avanzamento temporale e il coordinamento tra LP [8] [11].

Per affrontare queste problematiche, si utilizzano tecniche di **riallocazione dinamica** delle entità, anche nota come *migrazione* degli agenti. Il principio è semplice: se un agente X, originariamente assegnato al LP_0, comunica principalmente con agenti gestiti da LP_1, è vantaggioso spostare X su LP_1. In questo modo, la maggior parte delle interazioni di X potrà essere gestita

localmente, riducendo l'**overhead di comunicazione inter-processo**.

Questo meccanismo rientra in una più ampia strategia detta **clustering**, che mira a raggruppare agenti con forti correlazioni comunicative. Il clustering è uno strumento fondamentale nelle tecniche di *load balancing* delle simulazioni parallele e distribuite: analizzando i pattern di interazione tra agenti, è possibile ridurre al minimo le comunicazioni remote e massimizzare quelle locali, migliorando così le prestazioni globali della simulazione.

L'approccio MAS, quando combinato con le architetture PADS (Parallel And Distributed Simulation), rappresenta una soluzione efficace per simulare sistemi complessi [9], scalabili e realistici. La gestione automatica del carico computazionale e la dinamicità dell'allocazione rendono questi ambienti particolarmente adatti a contesti in cui l'equilibrio tra correttezza, prestazioni ed efficienza è cruciale.

1.4 ARTÌS/GAIA

Il framework *ARTÌS/GAIA* è stato sviluppato per abilitare la simulazione parallela e distribuita di sistemi complessi, dinamici e su larga scala. Si compone di due livelli software distinti ma strettamente integrati:

- **ARTÌS** (Advanced RTI System) funge da middleware di base, offrendo servizi fondamentali come la comunicazione tra processi logici (LPs) e la sincronizzazione temporale;

- **GAIA** (Generic Adaptive Interaction Architecture) si appoggia ad ARTÌS e introduce meccanismi adattivi per ottimizzare dinamicamente il partizionamento delle entità simulate.

Il principale obiettivo del framework è massimizzare l'efficienza delle simulazioni in ambienti dinamici e distribuiti, riducendo l'overhead comunicativo e migliorando il bilanciamento del carico.

Architettura di ARTÌS

ARTÌS è progettato secondo un'architettura modulare a più livelli. Lo strato più basso è quello di comunicazione, che seleziona automaticamente il canale più efficiente in base alla topologia del sistema (es. memoria condivisa, TCP/IP, MPI) e applica ottimizzazioni come il *message marshalling*, per ridurre il numero di messaggi scambiati.

Il nucleo centrale del middleware (RTI Core) è ispirato allo **standard IEEE 1516 HLA** (High-Level Architecture) [10], ampiamente utilizzato in ambito simulativo per favorire l'interoperabilità tra simulatori diversi. Sebbene lo standard HLA non sia direttamente utilizzato nei modelli presentati in questa tesi, ARTÌS ne implementa i servizi fondamentali, tra cui:

- *Time Management* e sincronizzazione causale (es. algoritmi conservative come Chandy-Misra-Bryant [3] o approcci ottimistici come Time Warp [11]);
- *Data Distribution Management*, *Ownership Management*, e altri servizi previsti da HLA per la gestione delle entità e delle interazioni tra simulatori.

Per facilitare l'integrazione nei progetti, ARTÌS espone una famiglia di API, le *Unibo API*, progettate per essere leggere e adatte allo sviluppo rapido in `C/C++` e `Java`.

Architettura di GAIA

GAIA si concentra sul miglioramento delle prestazioni in scenari altamente dinamici, in cui le entità interagiscono frequentemente e in modo non predicibile. Il meccanismo principale introdotto è la **migrazione adattiva** delle entità tra LPs, che mira a:

- **Raggruppare** le entità che comunicano spesso, in modo da sfruttare comunicazioni locali più rapide;

- **Bilanciare** il carico computazionale tra LPs, evitando congestioni e rallentamenti.

La decisione di migrazione è guidata da una valutazione costi-benefici: il sistema stima il *costo della migrazione* (MigC), includendo il tempo di serializzazione (`MigCPU`), la trasmissione (`MigComm`) e il calcolo dell'euristica (`Heu`). La migrazione avviene solo se si prevede un miglioramento netto delle prestazioni [5].

Le euristiche utilizzate da GAIA sono **locali e leggere**: ogni LP valuta la propria situazione in modo autonomo, senza la necessità di una visione globale del sistema. Questo approccio mantiene basso l'overhead decisionale e favorisce la scalabilità.

Il meccanismo di bilanciamento del carico può essere **simmetrico**, distribuendo equamente le entità, oppure **asimmetrico**, tenendo conto della potenza computazionale effettiva di ciascuna PEU (Physical Execution Unit). L'obiettivo finale è minimizzare il *Model Interaction Cost* (MIC) [7], mantenendo la simulazione coerente e reattiva.

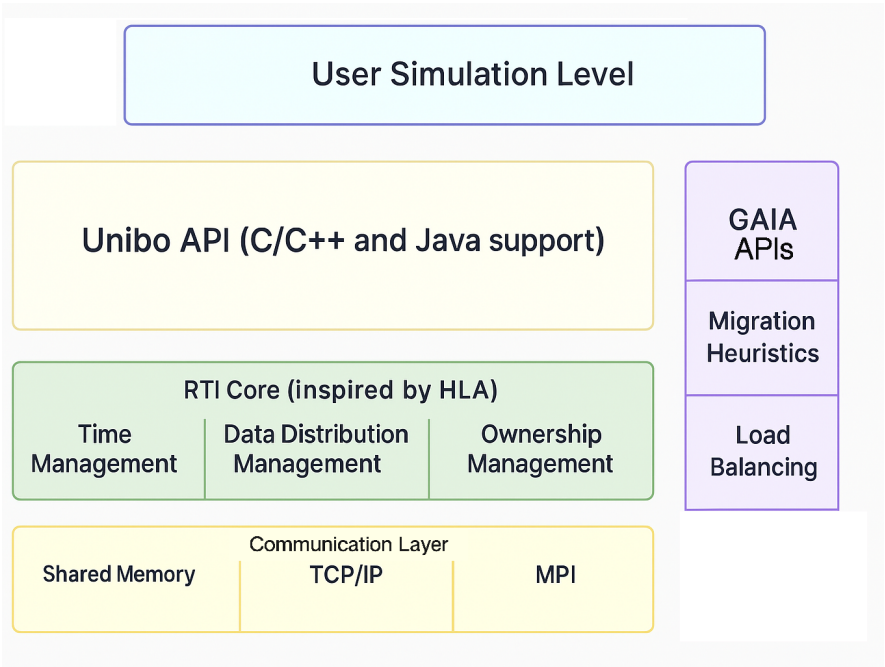


Figura 1.1: Struttura concettuale dei livelli software ARTÌS e GAIA.

Capitolo 2

Sviluppare con ARTIS/GAIA

2.1 Studio del framework

Il sistema prevede la presenza di molteplici *Logical Processes* (LPs), ciascuno dei quali funge da contenitore dinamico per un sottoinsieme delle entità simulate (SEs). Gli LPs sono assegnati a *Physical Execution Units* (PEUs), che possono essere CPU multicore, sistemi SMP, cluster o ambienti cloud. Ogni PEU può ospitare più LPs contemporaneamente. Durante l'esecuzione, i LPs si scambiano messaggi per coordinarsi e sincronizzarsi.

Ogni LP si occupa della gestione dello stato interno delle entità simulate e della comunicazione. Un ruolo centrale nella fase di configurazione e sincronizzazione è svolto dal componente **SiMa** (*Simulation Manager*), parte integrante dell'architettura ARTÌS. Il SiMa svolge un ruolo parzialmente centralizzato: si occupa di coordinare l'inizializzazione della simulazione, gestire le barriere di sincronizzazione e terminare ordinatamente l'esecuzione.

Alcune API fornite da ARTÌS e gestite tramite il SiMa includono: `SIMA_Initialize`, per la configurazione dei canali di comunicazione, l'identificazione dei LPs e la scelta delle porte di rete; `SIMA_Barrier`, per l'istituzione di barriere di sincronizzazione tra LPs; `SIMA_Finalize`, per la chiusura ordinata delle connessioni al termine della simulazione.

La topologia logica e i parametri di comunicazione vengono specificati

in un file di configurazione (`channels.txt`), che assegna un identificativo numerico a ciascun LP. La sequenza corretta di avvio prevede che il SiMa venga lanciato per primo, seguito dalle istanze dei LPs, che si connettono al coordinatore e ottengono le informazioni necessarie alla comunicazione.

Durante la simulazione, le interazioni tra entità sono gestite localmente se avvengono all'interno dello stesso LP o tra LPs residenti sulla stessa PEU (in ambiente con memoria condivisa). In questi casi, la comunicazione è rapida ed efficiente. Al contrario, le interazioni tra LPs su PEUs distinte richiedono comunicazioni di rete, che risultano più costose in termini di latenza e sincronizzazione.

Dopo aver familiarizzato con il concetto di Simulation Manager, ho analizzato alcune delle principali primitive offerte da GAIA, che verranno poi utilizzate nei modelli simulativi descritti nelle sezioni successive.

In particolare: `GAIA_Initialize(...)` è la funzione responsabile dell'inizializzazione del runtime per ciascun LP, mentre `GAIA_Register(...)` consente di registrare nuove entità simulate all'interno dell'ambiente esecutivo. La gestione dell'avanzamento temporale è affidata alla chiamata `GAIA_TimeAdvance()`, che implementa la logica di sincronizzazione tra LPs. Le funzioni `GAIA_Send(...)` e `GAIA_Migrate(...)` gestiscono rispettivamente l'invio di messaggi tra entità e la migrazione di agenti tra LP remoti, in risposta alle notifiche del runtime.

Queste primitive costituiscono l'interfaccia fondamentale per lo sviluppo di modelli distribuiti e saranno utilizzate nelle sezioni 2.2 e 2.3.

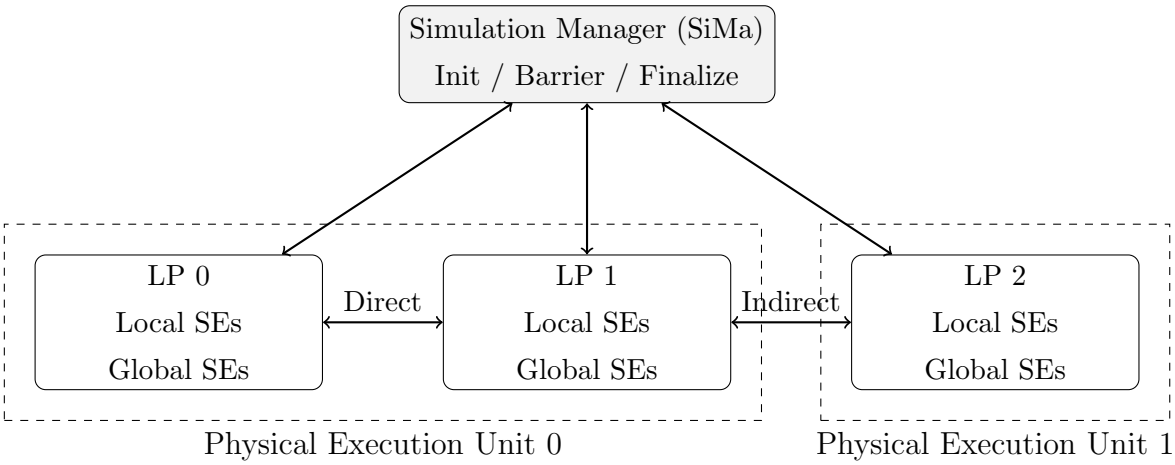


Figura 2.1: Architettura esecutiva ARTÌS: il SiMa gestisce l'inizializzazione e sincronizzazione, mentre i Logical Processes (LPs) gestiscono entità locali e globali, comunicando direttamente tra loro durante l'esecuzione.

2.2 Dinamiche dei modelli

Questa sezione descrive a livello operativo il flusso tipico di esecuzione di un modello simulativo implementato nel framework ARTÌS/GAIA. Il ciclo di vita di un modello si articola in tre fasi principali: *inizializzazione*, *esecuzione* e *terminazione*.

Inizializzazione

La fase iniziale prevede l'invocazione delle primitive `GAIA_Initialize()` e `GAIA_SetFstID()` per l'inizializzazione del contesto di esecuzione e dell'intervallo di identificatori assegnato a ciascun *Logical Process* (LP). Ogni LP, essendo un processo distinto e non un thread, esegue in autonomia una copia del codice. Non esiste condivisione di heap o stack tra LP: la comunicazione avviene esclusivamente tramite messaggi.

Durante questa fase, ogni LP istanzia le proprie strutture dati locali. Nei modelli sviluppati si è fatto ampio uso di tabelle hash, separate per le entità

locali e per quelle globalmente note (ma non residenti). Dopo l'inizializzazione, ciascun LP registra le entità di cui è responsabile tramite la primitiva `GAIA_Register()`, che rende l'entità visibile al framework e abilitata alla ricezione di messaggi e alla migrazione.

Esecuzione

Ciascun *Logical Process* (LP) entra in un ciclo `while` in cui, ad ogni iterazione, elabora un singolo messaggio ricevuto dal framework. Questo meccanismo è reso possibile dalla primitiva bloccante `GAIA_Receive()` che attende l'arrivo di un messaggio.

I messaggi ricevuti possono essere sia di sistema (predefiniti dal framework), sia personalizzati, definiti dall'utente in funzione della logica del modello. Tra i messaggi di sistema più rilevanti vi sono:

- `REGISTER`: comunica che un'entità registrata da un altro LP è ora visibile nel contesto globale;
- `EXEC_MIGR`: notifica la migrazione di un'entità da un altro LP, che deve essere inserita anche nelle strutture locali (e quindi gestita operativamente);
- `EOS` (End Of Step): segnala il termine del timestep corrente e abilita l'esecuzione delle operazioni di chiusura dello slot temporale.

Il codice eseguito all'arrivo di ciascun messaggio è contenuto in `event-handlers` che agiscono sulle strutture dati locali o globali. È fondamentale ricordare che gli LP non condividono memoria: ogni processo mantiene il proprio spazio di indirizzamento, contenente strutture locali (per le entità residenti) e strutture globali (per le entità non residenti ma note). Questa distinzione riflette un modello operativo distribuito in cui ogni LP ha visibilità solo parziale del sistema simulato, siccome mantenere in memoria tutte le informazioni del sistema sarebbe troppo oneroso e soprattutto andrebbe contro alle motivazione della simulazione distribuita. Per questo

motivo, una delle criticità progettuali nella fase di esecuzione riguarda la coerenza tra copie delle entità replicate tra LP. Valutiamo ad esempio un meccanismo infettivo, dove entità infette possono contagiare entità sane; la probabilità di contagio riuscito dipende dallo stato interno dell'entità target, che però è noto solo all'LP che la contiene. Se un'entità A residente su LP_0 interagisce con un'entità B residente su LP_1, è necessario che LP_0 invii un messaggio a LP_1 notificando un *tentativo di interazione* (es. infezione). Sarà responsabilità di LP_1, conoscendo il vero stato interno di B, determinare se l'interazione ha esito positivo. In caso affermativo, un nuovo messaggio viene inviato a tutti gli LP per comunicare la variazione di stato globale (es. avvenuta infezione). Questo scambio avviene tipicamente su più timestep:

- a $t = t_X$, il messaggio di tentativo è inviato;
- a $t = t_X + 1$, il LP target elabora il tentativo e, se necessario, propaga un evento confermato;
- a $t = t_X + 2$, tutti gli LP aggiornano le rispettive strutture.

Alla ricezione del messaggio `EOS`, vengono tipicamente eseguite le funzioni che determinano il comportamento attivo delle entità nel timestep corrente. Ad esempio, si stabilisce quali entità infette tenteranno di contagiare altre, quali entità si riprodurranno, e così via. In questo modo si realizza una separazione concettuale tra la fase di *reazione* agli eventi ricevuti nel timestep precedente e la fase di *generazione* di nuovi eventi che saranno elaborati nel passo successivo. Va però osservato che questa distinzione non è sempre netta: ad esempio, la ricezione di un messaggio di tentativo di infezione può già innescare la creazione di eventi futuri, che verranno notificati successivamente agli altri LP.

Completate le operazioni computazionali del timestep, viene infine effettuato il logging delle statistiche parziali, utile sia per l'analisi che per la validazione dei risultati.

Infine, ogni LP invoca `GAIA_TimeAdvance()`, funzione bloccante che sospende l'esecuzione fino a che il framework non autorizza l'avanzamento al

timestep successivo. Questo garantisce una forma di sincronizzazione globale tra LP, mantenendo il vincolo di coerenza temporale.

L'esecuzione prosegue fino al raggiungimento di un valore predefinito del `simulation_clock`, momento in cui il framework invia un messaggio `EOS` finale che innesca la fase di terminazione.

Terminazione

La fase di terminazione viene innescata automaticamente al raggiungimento del `simulation_clock` massimo (`END_CLOCK`). In quel momento, ciascun LP:

- stampa statistiche finali,
- libera le risorse dinamiche (es. hash tables, code di messaggi, entità),
- invoca la primitiva `GAIA_Finalize()` per concludere correttamente la simulazione.

2.3 Modelli sviluppati

L'obiettivo inziale dell'attività è stato fornire un'introduzione pratica alla modellazione e simulazione di *Multi-Agent Systems* (MAS), attraverso lo sviluppo incrementale di un modello simulativo originale. L'intero lavoro è stato realizzato in linguaggio `C`, integrando il codice all'interno del framework *GAIA*, che ha offerto il supporto necessario per l'organizzazione distribuita della simulazione, la migrazione automatica delle entità tra *Logical Processes* (LPs) e il bilanciamento del carico computazionale.

Il modello finale riproduce dinamiche ispirate a processi biologici come infezione, riproduzione, alimentazione e mortalità. L'obiettivo era esplorare l'effetto della variazione e trasmissione di tratti genetici sul comportamento della popolazione, richiamando in modo semplificato alcune dinamiche di adattamento evolutivo [12].

Modello base e approccio incrementale

Il lavoro è iniziato con l'analisi del modello esemplificativo `MIGRATION-WIRELESS`, presente nella distribuzione ufficiale di ARTÌS/GAIA. Tale modello simula un insieme di *Simulated Mobile Hosts* (SMHs) che si muovono in uno spazio toroidale bidimensionale secondo il modello di mobilità *Random Way Point* (RWP) [2], interagendo tramite messaggi wireless basati sulla prossimità spaziale.

Per semplificare l'approccio iniziale, è stata rimossa la componente wireless, ottenendo un primo modello in cui le entità mobili si muovono liberamente nello spazio e inviano messaggi di tipo `Hello` agli agenti presenti entro una distanza pari a `PROX_RADIUS`. Questi messaggi agiscono come semplici *ping*, senza alcuna semantica applicativa, e servono unicamente a generare interazioni tra entità prossimali. Il prototipo risultante è stato utilizzato per testare il comportamento del sistema di *self-clustering* di GAIA, verificando la capacità del bilanciatore di carico di raggruppare entità interagenti negli stessi LP, come atteso in una configurazione ottimizzata.

Modello `Infection` e `IDR`

In una seconda fase, il modello è stato esteso introducendo una dinamica di contagio: le entità possono nascere già infette oppure contrarre l'infezione quando si trovano entro una certa distanza `infection_radius`) da un agente infetto, con una probabilità definita (`infection_chance`). Questa fase ha richiesto la gestione dello stato interno degli agenti e la definizione di un meccanismo di propagazione, basato sul controllo delle distanze spaziali e sull'invio di messaggi tra LP.

Successivamente, è stata integrata la dinamica di *morte* e *riproduzione*, dando origine al modello `IDR` (*Infection–Death–Reproduction*). Le entità sane possono generare prole nella posizione in cui si trovavano, mentre quelle infette hanno una probabilità di morire. Inizialmente, le entità morte venivano semplicemente disattivate: non ricevevano più messaggi né producevano

eventi. Tuttavia, questa scelta ha portato a un progressivo degrado delle prestazioni, dovuto all'accumulo di agenti "zombie" ancora presenti nelle strutture dati. Per migliorare l'efficienza e la scalabilità, si è passati alla rimozione esplicita delle entità dalla simulazione.

Entrambe le dinamiche sono state implementate attraverso strutture dati dedicate. In particolare, alla ricezione di un messaggio di `morte`, o alla determinazione che un'entità debba *riprodursi*, l'entità coinvolta viene inserita in una lista apposita. Per ogni LP sono mantenute due liste di morte (una per entità locali e una per entità globali), e una lista di riproduzione. Queste strutture memorizzano i riferimenti (puntatori) ai nodi contenuti nelle `hash tables` delle entità.

Alla fine di ciascun timestep, due funzioni distinte si occupano di elaborare tali liste:

- nel caso della **riproduzione**, per ogni entità presente nella lista si invoca `GAIA_Register()`, generando una nuova entità. Poiché GAIA non consente di specificare direttamente i dati dell'entità creata, la logica del modello intercetta i messaggi di tipo `REGISTER` e, se esiste almeno un'entità in riproduzione, assume che si tratti del "newborn" corrispondente. Viene così effettuata l'inizializzazione esplicita dell'entità figlia, copiando attributi e posizione da quella madre.

- nel caso della **morte**, le funzioni percorrono le due liste e rimuovono le entità sia dalle strutture locali (se presenti) sia da quelle globali. Inoltre, per evitare inconsistenze, ogni entità eliminata viene anche rimossa da eventuali liste accessorie (come quelle di migrazione o riproduzione) in cui potrebbe comparire.

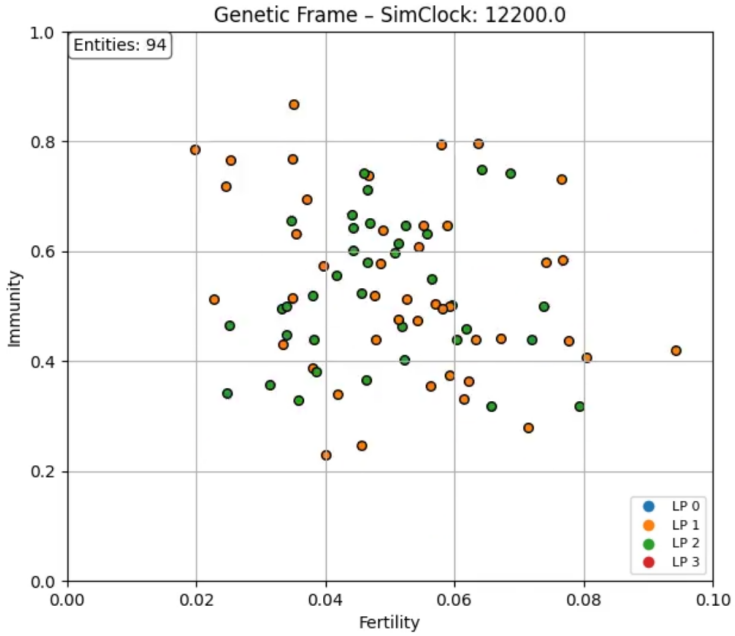
Modello `Genetic-IDR` e vincoli ambientali

Successivamente è stato introdotto un meccanismo genetico semplice: ogni entità possiede due attributi, *immunità* e *fertilità*, trasmessi alla prole con leggere variazioni casuali. Il comportamento risultante richiama, in

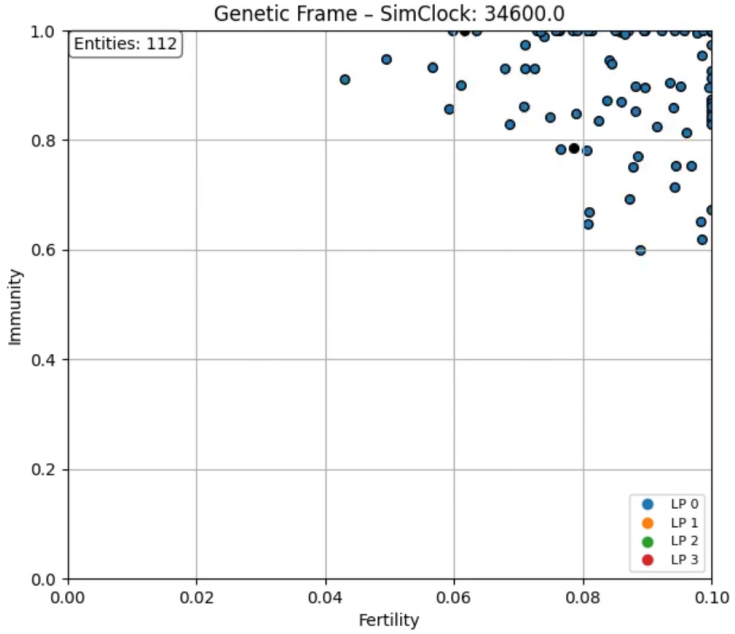
forma semplificata, alcune dinamiche evolutive: gli agenti con maggiore resistenza all'infezione tendevano a sopravvivere più a lungo, riprodursi più frequentemente e quindi a trasmettere i propri tratti genetici alle generazioni successive.

Per analizzare l'evoluzione del sistema è stato sviluppato lo script `genetic_visualizer.py`, che elabora i dati prodotti dalla simulazione di immunità e fertilità di ogni entità e genera immagini che rappresentano graficamente la distribuzione delle entità nel piano *fertilità–immunità* ad un determinato timestep. Ogni punto corrisponde a un'entità viva; ll grafico mostra quindi la composizione genetica della popolazione in un dato istante. Durante lo sviluppo del modello `Genetic-IDR`, è stata anche aggiunta la funzione `SpawnInfected` che reinserisce un'entità infetta nella simulazione quando non ne rimangono presenti. Questa logica garantisce che la popolazione sana continui a essere esposta all'infezione, evitando una crescita illimitata del numero di agenti sani. Si tratta di un ulteriore meccanismo introdotto per stabilizzare il sistema e favorire la pressione selettiva verso l'evoluzione genetica all'interno della popolazione simulata.

Per evitare fenomeni di sovrappopolazione, è stato infine introdotto un meccanismo di alimentazione, dando origine al modello `Genetic-IDRA` (*Infection–Death–Reproduction–Alimentation*). A ogni *timestep*, una quantità limitata di cibo viene distribuita tra le entità attive. Gli agenti che non riescono ad alimentarsi accumulano un livello crescente di *fame* e muoiono se viene superata una soglia critica. Questa dinamica ambientale ha migliorato la stabilità del sistema, mantenendo sotto controllo la dimensione della popolazione nel lungo termine.



(a) Distribuzione al primo frame analizzato: gli agenti presentano valori di immunità e fertilità distribuiti casualmente.



(b) Distribuzione all'ultimo frame analizzato: la popolazione mostra un'evidente tendenza verso valori di immunità e fertilità più elevati.

Figura 2.2: Evoluzione genetica della popolazione nel piano fertilità–immunità con parametri fine-tuned. I punti rappresentano entità vive; il colore identifica l'LP di appartenenza.

Attributi dinamici e sincronizzazione efficiente

Oltre a immunità, fertilità e fame, sono stati introdotti altri parametri dinamici per ogni agente:

- **Età** (`age`): incrementa a ogni timestep e determina la maturità riproduttiva e la morte per vecchiaia;
- **Livello di infezione** (`infection level`): varia nel tempo in modo stocastico, aumentando o diminuendo in base a valutazioni casuali. Se il valore scende a 0, l'agente guarisce; se invece supera una soglia prefissata, l'agente muore.

Per limitare l'overhead comunicativo, tali attributi vengono aggiornati localmente e condivisi solo in fase di migrazione, secondo la logica di condividere solo i dati strettamente necessari tra LP. Questo approccio ha permesso di mantenere la coerenza della simulazione riducendo le comunicazioni.

2.4 Criticità

I modelli realizzati hanno dimostrato un comportamento coerente con le attese ma si sono rivelati fragili, specialmente in esecuzioni prolungate. In particolare, si sono verificati errori a runtime, come *segmentation fault*, verosimilmente legati a problematiche nella gestione della memoria e alla rimozione incompleta delle entità simulate.

Il framework GAIA, infatti, non prevede meccanismi espliciti per la cancellazione o la creazione dinamica di entità durante l'esecuzione. La funzione `GAIA_Initialize(...)` richiede la dichiarazione anticipata del numero massimo di entità gestibili dal runtime, vincolando così la simulazione a una capacità fissa. In scenari dinamici, come quelli trattati nei modelli evolutivi, questa limitazione può introdurre conflitti nella gestione degli identificatori (ID) univoci delle entità tra LPs distinti, o portare a inefficienze nel bilanciamento del carico.

Un'ulteriore criticità riguarda la possibilità che alcune entità disattivate non vengano completamente rimosse dalle strutture dati locali dei LPs, causando accumulo di dati inutilizzati e rallentamento progressivo. Sebbene siano stati sviluppati strumenti di visualizzazione esterni in Python per osservare il comportamento della simulazione, non è ancora stato condotto testing.

Nel prosieguo del lavoro, sarà quindi importante approfondire:

- la gestione robusta della memoria e degli ID durante dinamiche di nascita/morte;
- la validazione della correttezza della simulazione;
- la progettazione di test scalabili e ripetibili;

Questi sviluppi, uniti a una maggiore automazione del monitoraggio e dell'analisi dei risultati, potranno contribuire a rendere i modelli più affidabili, scalabili e adatti a scenari simulativi complessi e realistici.

Capitolo 3

Testing e Correzione dei modelli

3.1 Risoluzione degli identificatori

Nel framework *GAIA*, l'assegnazione degli identificatori univoci alle entità simulate (SE) è progettata per garantire coerenza e correttezza in un ambiente di simulazione distribuito. Ogni entità registrata riceve un descrittore globale, valido per l'intera durata della simulazione, e utilizzabile per identificare la SE durante la comunicazione o la migrazione.

Durante la fase di inizializzazione (bootstrap), a ciascun *Logical Process* (LP) viene assegnato un intervallo di identificatori validi. La funzione `GAIA_SetFstID(...)` consente di specificare il primo ID disponibile che l'LP locale utilizzerà per registrare nuove entità. Al momento della chiamata a `GAIA_Register(...)`, il runtime assegna un ID all'entità che ne consente la visibilità globale: solo le SE registrate in questo modo possono scambiare messaggi o essere soggette a migrazione.

Poiché GAIA adotta una strategia di *migrazione adattiva*, l'ID di una SE non è legato in modo permanente a un LP specifico. L'associazione tra una SE e il suo LP può cambiare dinamicamente durante l'esecuzione, e per questo motivo l'identificatore deve restare valido indipendentemente dalla

sua collocazione attuale.

L'introduzione di dinamiche di riproduzione all'interno del modello ha complicato la gestione degli identificatori. In particolare, il numero totale di entità generate durante una simulazione non è noto a priori, poiché dipende dai parametri di esecuzione e dall'evoluzione stocastica del sistema. Il framework GAIA, tuttavia, richiede che il numero massimo di entità sia dichiarato al momento dell'inizializzazione attraverso la primitiva `GAIA_Initialize(...)`.

Per risolvere questo vincolo, è stato adottato un approccio che prevede l'allocazione preventiva di un intervallo distinto di identificatori per ciascun LP, calcolato come:

```
GAIA_SetFstID(LPID * MAX_ENT_PER_LP)
```

dove:

- `LPID` è l'identificativo numerico dell'LP;
- `MAX_ENT_PER_LP` è il numero massimo stimato di entità per LP.

Ogni LP inizializza il framework invocando `GAIA_Initialize(...)` con una stima conservativa del numero massimo di entità totali. A ogni nuova registrazione, il runtime verifica che l'ID generato rientri nell'intervallo assegnato: in caso contrario, la simulazione viene interrotta, segnalando una violazione delle condizioni di integrità del sistema.

3.2 Visualizzazione della simulazione

In primo luogo, è stato sviluppato uno script Python di visualizzazione `position_visualizer.py` che, a partire da file di log contenenti posizione e stato delle entità, genera un video animato della simulazione. Questo strumento ha permesso un controllo empirico, utile per osservare il comportamento complessivo del sistema: il movimento delle entità, la trasmissione dell'infezione, la nascita di nuovi agenti e la loro eventuale scomparsa.

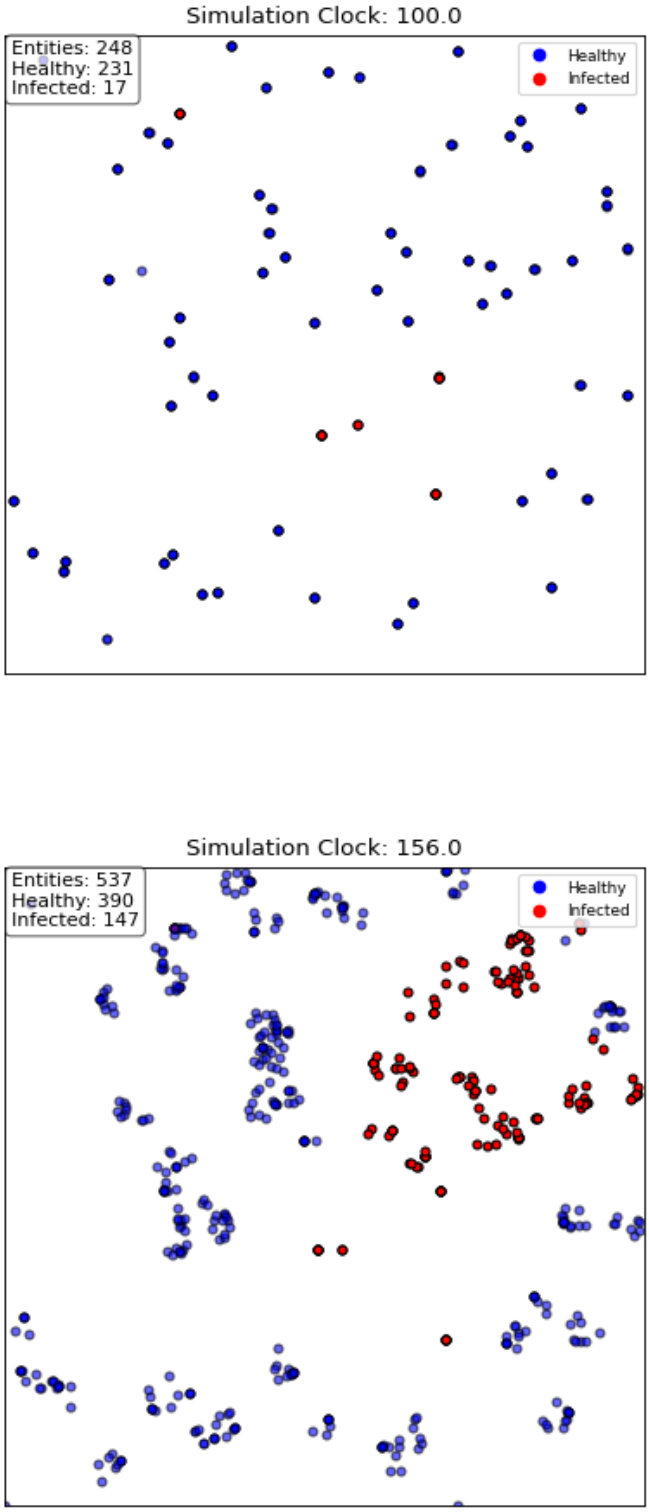


Figura 3.1: Evoluzione della simulazione al timestep 100 e 156. Gli agenti sani sono rappresentati in blu, quelli infetti in rosso. È possibile osservare l'espansione dell'infezione e la nascita di nuove entità.

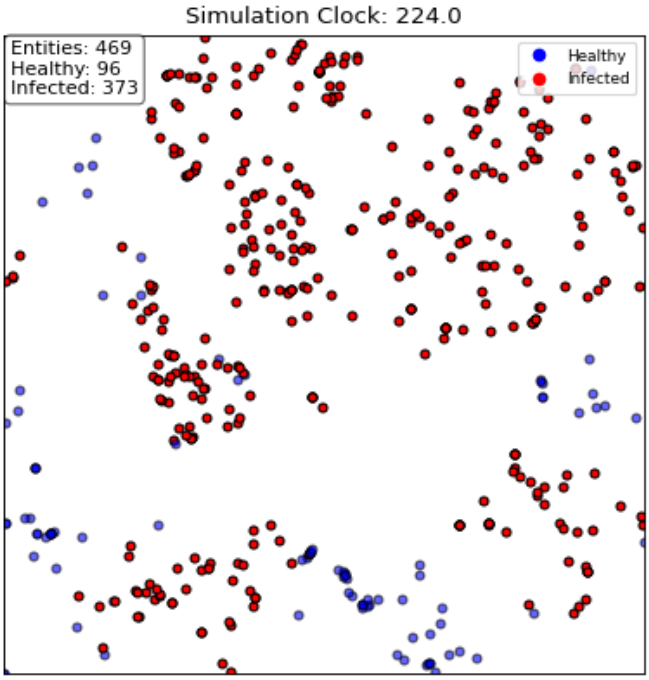


Figura 3.2: Evoluzione della simulazione al timestep 224: l'infezione si è diffusa maggiormente. La visualizzazione consente di identificare facilmente la distribuzione spaziale delle entità e lo stato di salute della popolazione simulata.

3.3 Validatore

Per accertare la correttezza semantica delle dinamiche non direttamente supportate dal framework — in particolare, la *riproduzione* e la *morte* delle entità — è stato necessario adottare un metodo di validazione automatizzata.

A tal fine è stato sviluppato uno script Python dedicato (`validator.py`), il cui obiettivo è analizzare i file di log generati da un LP specifico (denominato `LP_STAT`), che, a ogni timestep, stampa lo stato globale della simulazione: l'elenco delle entità presenti, i messaggi di riproduzione, i decessi, e l'orologio

simulato. Il validatore scorre cronologicamente questi file e applica una serie di controlli per rilevare incongruenze.

I controlli principali effettuati dallo script sono:

- **Duplicazione illegale**: nessuna entità può essere riprodotta più di una volta senza prima morire. Ogni nuova entità deve essere preceduta da un messaggio di tipo `R` (riproduzione).
- **Morte coerente**: ogni messaggio di morte (`D`) deve riferirsi a un'entità effettivamente attiva. Non possono verificarsi decessi multipli per la stessa entità, né decessi di entità mai comparse.
- **Presenza coerente**: un'entità considerata viva non può scomparire improvvisamente dal registro dello stato. Viceversa, entità già morte non devono più comparire nel sistema (nessuna "resurrezione").

L'output dello script è un file di log riepilogativo, che elenca:

- la durata della simulazione in passi temporali;
- il numero iniziale e totale di entità nate e morte;
- il numero e il dettaglio delle eventuali violazioni logiche rilevate.

Per automatizzare il processo di testing e validazione, è stato realizzato uno script di esecuzione (`run`) che consente di compilare ed eseguire la simulazione in quattro configurazioni distinte, ottenute combinando la presenza o l'assenza delle dinamiche di *riproduzione* e *morte*. Lo script gestisce in sequenza la ricompilazione del codice con le flag appropriate, l'esecuzione della simulazione e la validazione automatica tramite il validatore precedentemente descritto.

Questo approccio consente di testare il comportamento del sistema in ogni scenario possibile (`REPR=ON/OFF`, `DTH=ON/OFF`) in modo sistematico e ripetibile, evitando errori manuali e garantendo coerenza tra configurazione, esecuzione e analisi.

3.4 Errori di validazione riscontrati

Grazie al sistema di validazione descritto nella sezione precedente, sono state identificate criticità ricorrenti. Con la dinamica di *morte* abilitata, lo script rilevava sporadicamente la presenza di messaggi di decesso riferiti ad entità già dichiarate morte nel timestep precedente, violando i vincoli logici stabiliti dal validatore.

Per isolare il problema, sono stati realizzati due modelli semplificati e deterministici, focalizzati esclusivamente sulle dinamiche di riproduzione e morte. In questi modelli, `Testing_Death` e `Testing_Reproduction`, le entità non interagiscono tra loro: vengono generati messaggi a tavolino verso un insieme arbitrario di target, e successivamente si verifica che il numero di messaggi ricevuti corrisponda esattamente al numero inviato. I test hanno confermato il comportamento corretto del framework in questi contesti: anche con carichi elevati (fino a 100.000 messaggi), non si è osservata alcuna duplicazione o anomalia. Questo ha rafforzato l'ipotesi che il problema fosse legato a interazioni specifiche del modello più complesso, e non a malfunzionamenti del sistema sottostante.

Il debug del modello originale `Testing G_IDRA 1` ha permesso di individuare l'origine dell'errore: le funzioni che determinano le interazioni tra agenti venivano eseguite in un momento temporale disallineato rispetto alla rimozione delle entità morte. In particolare, durante un dato timestep, un'entità poteva ancora essere visibile alle funzioni di scansione come potenziale target, anche se era destinata a essere rimossa nello stesso passo temporale. Questo portava all'invio di messaggi verso entità che sarebbero state eliminate poco dopo, generando così errori di validazione.

Per risolvere il problema, è stato introdotto un attributo ausiliario `DEAD`, assegnato alle entità al momento della ricezione di un messaggio di morte. Questo flag consente di marcare logicamente l'entità come "non più valida" già all'inizio del timestep successivo, senza rimuoverla fisicamente fino al termine del passo. Le funzioni di scansione sono state quindi modificate per ignorare tutte le entità marcate come `DEAD`, evitando che possano essere

considerate target di interazioni.

3.5 Controllo sui messaggi

Sono stati introdotti controlli sui messaggi ricevuti dagli LP destinati a entità locali; in particolare, a ogni messaggio ricevuto si verifica che l'entità target appartenga effettivamente all'LP corrente. In caso contrario, la simulazione viene interrotta immediatamente.

Questo controllo ha evidenziato numerosi casi di messaggi recapitati all'LP errato. Tali eventi risultano particolarmente critici, poiché un messaggio ignorato può compromettere la correttezza semantica della simulazione: i parametri probabilistici impostati (come tasso di infezione, riproduzione, ecc.) non vengono rispettati, portando a risultati quantitativamente alterati (ad esempio, un numero di infezioni inferiore alle attese).

Come primo passo per analizzare la causa del problema, è stato sviluppato uno script di validazione automatica, `global_validator.py`, il cui obiettivo è verificare la coerenza globale della simulazione. Questo script analizza, per ciascun timestep e per ogni LP, le informazioni sulle entità morte, riprodotte e presenti, verificando che tali dati siano perfettamente allineati su tutti gli LP. L'ipotesi iniziale era che uno o più LP non aggiornassero correttamente il campo di appartenenza (`lp`) di alcune entità, portando così all'invio di messaggi verso l'LP sbagliato. Tuttavia, `global_validator.py` non ha rilevato alcuna incongruenza nei dati raccolti, suggerendo che il problema fosse di natura più sottile.

L'analisi è quindi proseguita tramite ispezione diretta dei log, con stampa delle entità coinvolte in errori di consegna. Dopo una fase di debugging, è stato individuato un caso particolare legato alla migrazione delle entità.

Questo tipo di errore si verifica quando un'entità è selezionata come target di un messaggio (`hello message`) proprio mentre si trova nella fase di migrazione. Il processo di migrazione prevede che il framework notifichi a tutti gli LP che, ad esempio, l'entità 0 — inizialmente appartenente a LP

1 — deve migrare. Di conseguenza, tutti gli LP aggiornano il campo `lp` dell'entità, indicando il nuovo LP di destinazione. L'LP sorgente, in aggiunta, inserisce la migrazione in sospeso in una lista interna e, al termine del timestep, invia un messaggio di trasferimento all'LP di arrivo.

Se nel frattempo un'altra entità sceglie l'entità 0 come target, il messaggio verrà indirizzato al nuovo LP, in quanto tutti i nodi hanno già aggiornato il campo `lp`. A questo punto può verificarsi una condizione di race: se il messaggio `hello` giunge all'LP di destinazione prima del messaggio di migrazione in ingresso, l'entità 0 non sarà ancora presente nella tabella locale, e il controllo fallirà. Se invece l'ordine dei messaggi è invertito, non si verifica alcun errore.

Questo problema si è manifestato con i messaggi di tipo `hello`, che rappresentano la tipologia di messaggio più numerosa e sono quindi statisticamente più soggetti a innescare condizioni di race. Per gestire tale situazione, è stato introdotto un controllo secondario: se un messaggio ha come destinatario un'entità non presente nella tabella locale, viene consultata la tabella globale per verificarne l'appartenenza corrente.

Sebbene l'evento sia raro, è teoricamente possibile che condizioni analoghe si verifichino anche con messaggi più complessi, al momento non trattati esplicitamente poiché mai rilevati durante la fase di test. A questo proposito, sono stati effettuati test mirati per tentare di riprodurre la condizione descritta, senza mai osservarne l'effettiva comparsa.

Qualora l'errore dovesse manifestarsi, una possibile soluzione consisterebbe nell'introdurre una struttura dati dedicata, come un heap (coda di priorità), in cui inserire i messaggi ricevuti dotandoli di un campo di priorità utilizzabile per stabilire l'ordine di estrazione. Tale heap verrebbe riempito con i messaggi ricevuti durante ogni timestep e svuotato al termine del timestep stesso, eseguendo per ciascun messaggio estratto l'event handler corrispondente in ordine coerente con la priorità assegnata. In questo modo sarebbe possibile mantenere ordinata e consistente l'elaborazione dei messaggi evitando condizioni di race.

Va inoltre segnalato un ulteriore scenario in cui può verificarsi la ricezione di un messaggio indirizzato a un'entità non più esistente: la migrazione. In particolare, può accadere che il framework notifichi la migrazione di un'entità che, nel frattempo, è già stata rimossa perché deceduta. Questo fenomeno si osserva principalmente in due casi:

- quando un LP risulta sovraccarico, il framework può selezionare entità da migrare anche tra quelle inattive o già eliminate, poiché la rimozione definitiva non è ancora esplicitamente supportata;
- quando un'entità ha interazioni frequenti con entità residenti su altri LP, può essere scelta per la migrazione al fine di ridurre l'overhead comunicativo, anche se la sua eliminazione è imminente.

Attualmente, il framework non prevede una gestione nativa della rimozione di entità, rendendo difficile evitare in modo sistematico queste condizioni. Resta da valutare in che misura tali situazioni impattino negativamente sull'efficacia del bilanciamento del carico e sulla stabilità del sistema in simulazioni di lunga durata.

3.6 Robustezza e gestione della memoria

Un altro limite riscontrato nei modelli sviluppati riguardava l'affidabilità: in alcune configurazioni parametriche, la simulazione poteva terminare con errori di tipo `segmentation fault`, indicando problemi di gestione della memoria.

Per individuare e risolvere questi problemi, è stata effettuata un'analisi approfondita con le opzioni di compilazione `-fsanitize=address` e `-g`, e strumenti di esecuzione come `valgrind`. Le indagini hanno evidenziato che alcuni errori potevano derivare da operazioni di `free()` effettuate più volte sulla stessa entità oppure l'accesso a strutture dati di entità già deallocate.

In particolare, la stessa entità poteva essere liberata sia nel contesto locale (LP corrente) sia globale, portando a un doppio `free()` e a possibili

accessi a memoria non più valida. Inoltre le liste di gestione delle entità potevano essere consultate anche a seguito di alcune operazioni di `free`. Per evitare questi rischi, è stato modificato il modello dati: le liste di gestione delle entità (migrazione, riproduzione, morte) mantengono ora solo le chiavi (ID) delle entità, e non più i puntatori diretti. In questo modo, si riduce drasticamente la possibilità di riferimenti incrociati invalidi, semplificando anche la procedura di deallocazione della memoria.

La funzione di rimozione delle entità è stata a sua volta rifattorizzata: è ora strutturata in due fasi distinte, una per la rimozione dell'entità dalle strutture dati, e una per il rilascio effettivo della memoria. Questa separazione ha permesso di migliorare il controllo sul ciclo di vita delle entità, evitando conflitti e comportamenti indefiniti.

A seguito di queste modifiche, e della risoluzione del problema discusso nella sezione precedente (duplicazione dei messaggi di morte), la simulazione ha dimostrato una maggiore stabilità e non sono più stati rilevati accessi a memoria liberata.

Tuttavia, l'uso del *memory sanitizer* (`fsanitize`) continua a evidenziare la presenza di **memory leaks**, nonostante il fatto che, al termine della simulazione, tutte le principali strutture dinamiche vengano esplicitamente deallocate: le liste di migrazione, riproduzione e morte, sia locali che globali, così come le due tabelle hash principali. Il *memory sanitizer* evidenza come una grossa parte dei momory leaks derivi da una funzione di libreria del framework, chiamata `TS_BroadcastV`. Questi leak potrebbero essere dovuti alle dinamiche introdotte nei modelli non previste dal framework, quali morte e riproduzione delle entità. Per quanto rigaurda i restanti memory leaks, la causa potrebbe risiedere in riferimenti mantenuti internamente dal framework o in strutture secondarie allocate indirettamente. Un'analisi più approfondita con `valgrind` in modalità dettagliata potrà aiutare a chiarire la natura e l'origine di queste perdite di memoria residue.

3.7 Metriche di comunicazione: Local e Global Communication Ratio

Nel contesto del framework *GAIA/ARTÌS*, il concetto di **Local Communication Ratio** (LCR) [6] rappresenta una metrica chiave per valutare l'efficacia del clustering dinamico delle entità simulate. Si tratta di una misura quantitativa dell'efficienza comunicativa interna a ciascun LP.

L'*LCR* è definito come il rapporto tra il numero di interazioni **locali** — ovvero messaggi scambiati tra entità appartenenti allo stesso LP (tipicamente sulla stessa unità di esecuzione fisica, PEU) — e il numero totale di interazioni originate da quell'LP, includendo sia quelle locali che quelle remote. Formalmente:

$$\text{LCR} = \frac{\text{Interazioni locali}}{\text{Interazioni locali} + \text{Interazioni remote}}$$

Un LCR elevato indica che il framework è riuscito a raggruppare efficacemente le entità che interagiscono frequentemente all'interno dello stesso LP, riducendo l'overhead di comunicazione remota e sfruttando la memoria condivisa. Viceversa, un LCR basso può suggerire una dispersione delle interazioni, con impatti negativi sulle prestazioni.

In modo analogo, è possibile definire il **Global Communication Ratio** (GCR) come la misura complementare:

$$\text{GCR} = 1 - \text{LCR}$$

Esso rappresenta la quota di comunicazioni remote, ovvero i messaggi che attraversano i confini tra LPs e richiedono trasmissione su rete, risultando più costosi in termini di latenza e overhead.

Un aspetto emerso nel lavoro di tesi è l'effetto che dinamiche altamente variabili, come *morte* e *riproduzione*, esercitano sul clustering adattivo di GAIA. Tali meccanismi modificano continuamente la distribuzione spaziale e semantica delle entità simulate, mettendo alla prova le euristiche di migrazione e riallocazione.

Per analizzare questo comportamento, sono stati registrati dati statistici a runtime utilizzando la primitiva:

```
GAIA_GetStatistics(&loc, &rem, &migr);
```

dove: `loc` è il numero di interazioni locali nel timestep corrente; `rem` è il numero di interazioni remote; `migr` è il numero di migrazioni effettuate.

Questi valori sono stati accumulati per tutta la durata della simulazione e stampati a file per generare log utili all'analisi comparativa.

Per analizzare visivamente l'evoluzione di questi dati nel tempo, è stato sviluppato lo script Python `communication_visualizer.py`, che genera un grafico per ciascuna delle quattro configurazioni simulate (`REPR=ON/OFF`, `DTH=ON/OFF`). Lo script legge due sorgenti principali:

- Il file `sim_comm_checker.out`, che riporta per ciascun *simclock* il valore di *Local Communication Ratio* e di *Remote Communication Ratio*;
- I file `sim_state_#.out` prodotti a ogni timestep, che contengono le entità vive suddivise per LP.

A partire da questi dati, lo script produce un grafico a doppio asse Y:

- Sul primo asse (a sinistra) vengono riportati i valori di LCR nel tempo, rappresentati da punti verdi; i valori di GCR non vengono riportati siccome sono il "complementare" di LCR e si è ritenuto che compromettessero la leggibilità del grafico.
- Sul secondo asse (a destra) vengono tracciate le curve relative al numero di entità vive per ciascun LP e alla popolazione complessiva.

Il grafico risultante consente di osservare la relazione tra la variazione nella dimensione della popolazione e il comportamento comunicativo del framework. In particolare, fluttuazioni nel valore del *Local Communication Ratio* (LCR) possono spesso essere ricondotte a variazioni nella distribuzione delle entità tra LP, causate da eventi come riproduzione o morte massiva. Di seguito si riportano due configurazioni significative:

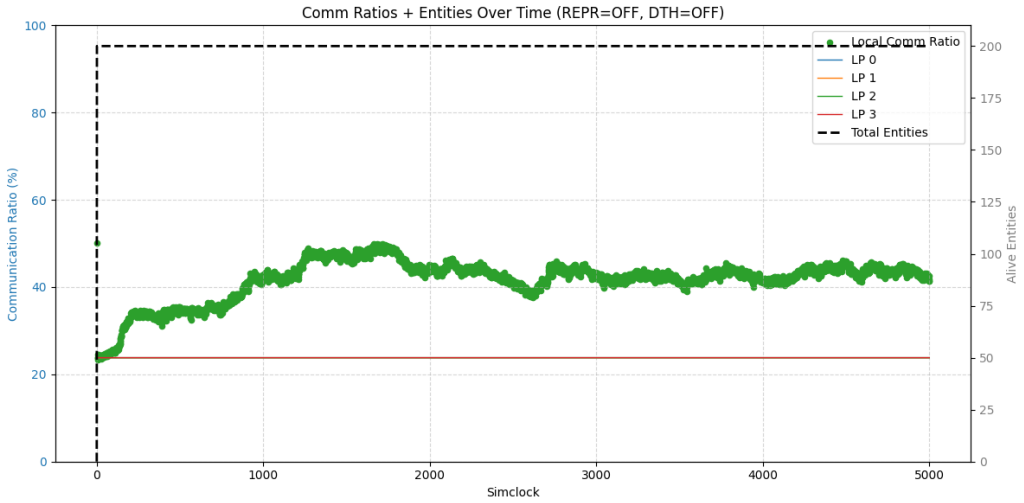


Figura 3.3: Comunicazione locale/remota nel caso base: riproduzione e morte disattivate.

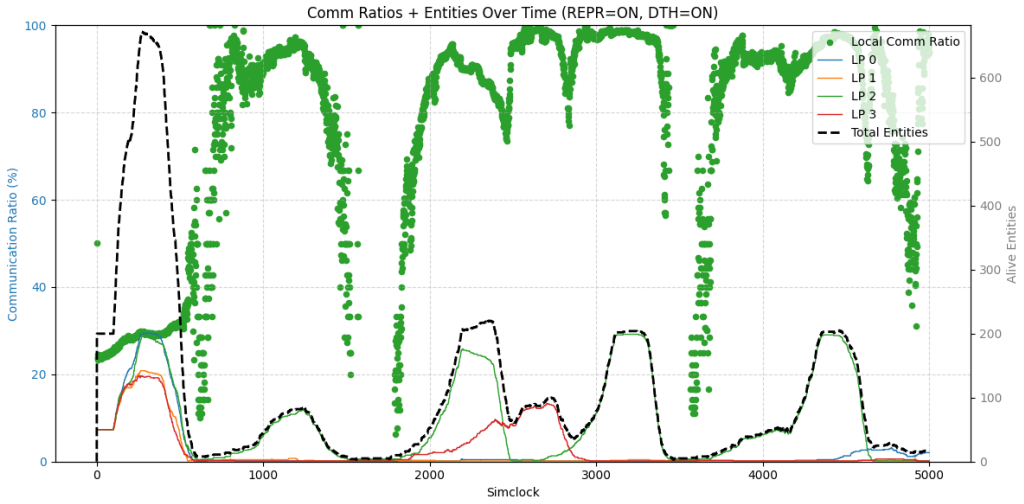


Figura 3.4: Comunicazione nel caso completo: riproduzione e morte attive.

L'analisi dei risultati suggerisce le seguenti osservazioni:

- Nel caso della Figura 3.1, in assenza di dinamiche evolutive (riproduzione e morte disattivate), i valori di LCR si mantengono stabili nel tempo, con un iniziale incremento e successive variazioni minime.

- Al contrario, nella Figura 3.2, dove sono attivi meccanismi di riproduzione e morte, il valore di LCR presenta oscillazioni significative. Tuttavia, si osservano anche picchi di comunicazione locale molto elevati. Questo risultato, in parte inaspettato, suggerisce che il framework riesca a mantenere l'efficacia del clustering anche in presenza di dinamiche altamente instabili. In particolare, si nota come i picchi di LCR corrispondano spesso a timestep in cui un singolo LP ha prodotto molte entità, rendendo probabili interazioni locali tra genitori e prole. In tali casi, il clustering efficace è più un effetto diretto della struttura della simulazione che delle euristiche del framework.

Un'ulteriore analisi interessante avrebbe potuto riguardare il comportamento del sistema con la migrazione disattivata, utilizzando la primitiva `GAIA_SetMigration(MIGR_OFF)`. Tuttavia, per motivi non ancora chiariti, in questa configurazione il framework produce valori `NaN` per le metriche di comunicazione durante tutta la simulazione, impedendo un confronto diretto.

Capitolo 4

Porting

Nel contesto dell'ingegneria del software, il termine *porting* indica il processo di adattamento di un programma, o di una parte di esso, per essere eseguito in un ambiente diverso da quello per cui era stato originariamente progettato. Questo può riguardare il sistema operativo, l'architettura hardware, o — come nel caso di questa tesi — il framework simulativo sottostante.

Il porting di un modello simulativo rappresenta un'operazione particolarmente utile per valutare la flessibilità di un framework: consente di confrontare direttamente lo stesso scenario implementato in ambienti diversi, analizzando come varia la gestione del parallelismo, la struttura dati, la semantica degli eventi e le prestazioni globali.

In questa tesi è stato selezionato un modello open source, disponibile pubblicamente all'indirizzo:

`https://github.com/paulvangentcom/python_corona_simulation`

Il progetto, sviluppato in linguaggio Python, simula la diffusione di un'infezione in una popolazione di agenti mobili. Il comportamento degli agenti include il movimento in uno spazio bidimensionale e la possibilità di contagio a distanza ravvicinata. Il modello originale è stato progettato per eseguire in modo completamente sequenziale.

Il lavoro svolto ha riguardato il porting di tale simulazione all'interno del framework distribuito *GAIA/ARTÌS*, con l'obiettivo di:

- confrontare la struttura concettuale del modello originale con quella necessaria in un contesto distribuito;
- verificare se il cambiamento di paradigma, da sequenziale a distribuito, può generare differenze significative nella logica simulativa.

Nel seguito del capitolo verrà descritta la struttura del modello originale, le modifiche effettuate per integrarlo nel framework, e un confronto qualitativo tra le due versioni.

4.1 Descrizione del modello

Come base per il porting è stato scelto il modello `simple_simulation.py`, un'applicazione Python che simula la diffusione di un'infezione virale all'interno di una popolazione mobile. Il modello è stato selezionato in quanto presenta dinamiche analoghe a quelle dei modelli precedentemente sviluppati (infezione, mobilità, mortalità), garantendo un contesto adeguato per valutare in modo realistico la portabilità delle soluzioni implementate nel framework ARTÌS/GAIA.

Ogni individuo della popolazione è rappresentato da un agente autonomo descritto tramite un array di dieci attributi numerici che includono posizione, direzione e velocità di movimento, stato di salute (sano, infetto, immune, deceduto), età e parametri relativi alla progressione dell'infezione.

La simulazione avanza a passi discreti (*frame*) seguendo lo schema seguente:

1. Verifica dei confini dello spazio simulato: se un'entità li supera, la posizione viene corretta tramite riflessione della direzione (`out_of_bounds()`).
2. Aggiornamento casuale di direzione e velocità (`update_randoms()`).
3. Aggiornamento della posizione in base a direzione e velocità (`update_positions()`).

4. Possibile contagio di entità sane vicine a individui infetti (`infect()`) con probabilità `infection_chance`. Se più della metà della popolazione risulta contagiata, tale probabilità viene moltiplicata per il numero di entità infette vicine. Sebbene questa logica non sia strettamente realistica, è stato interessante mantenerla per trasportarne la dinamica in GAIA.

5. Gli agenti infetti restano tali per una durata stocastica, al termine della quale guariscono o muoiono in base a probabilità che tengono conto anche dell'età (`recover_or_die()`).

6. Un modulo opzionale di visualizzazione (`matplotlib`) genera animazioni che mostrano la distribuzione spaziale degli agenti e l'andamento degli infetti nel tempo.

Il modello utilizza strutture dati essenziali (una matrice NumPy per la popolazione) e una logica sequenziale centralizzata. Sebbene efficace per simulazioni di piccola scala, questa impostazione non si adatta a esecuzioni distribuite. Il porting si è quindi concentrato sull'adattamento della logica di aggiornamento e comunicazione per sfruttare le potenzialità di ARTÌS/GAIA.

4.2 Implementazione in ARTÌS/GAIA

Il porting ha portato alla realizzazione del modello `PCSSimple_Porting`, in cui ogni individuo della simulazione è rappresentato come una *Simulated Mobile Host* (SMH). Le SMH si muovono nello spazio simulato, interagiscono tramite messaggi e possono infettarsi, guarire o morire, replicando le dinamiche del modello originale in un contesto distribuito e parallelo.

Posizione, stato di salute, età e parametri di guarigione sono mantenuti in `hash_table` globali e locali. Ogni Logical Process (LP) gestisce localmente le proprie entità, con le seguenti funzioni principali:

- `ScanMotion()`: gestisce il movimento delle entità locali, generando un *move_event* per ciascuna. Controlla le uscite dai confini (riflessione direzionale), aggiornando direzione, velocità e posizione, comunicando in broadcast le modifiche tramite `MoveMsg`, replicando le funzioni `out_of_bounds()`, `update_randoms()` e `update_positions()` del modello Python.

- `EstablishInfection()`: controlla per ciascuna entità sana locale quanti messaggi di infezione sono stati ricevuti (`infection_attempts`, aggiornati da `infection_event_handler()`). Se più della metà della popolazione è infetta, la probabilità di infezione viene aumentata moltiplicando `infection_chance` per il numero di tentativi ricevuti.

- `ScanInfection()`: itera sulle entità infette locali e identifica le entità sane circostanti (tramite una finestra di prossimità) a cui inviare messaggi di potenziale contagio (`InfectionMsg`). Tutte le entità sane rilevate riceveranno un `infection_message`, mentre l'effettiva valutazione del contagio viene demandata alla funzione `EstablishInfection()`; in questo modo si mantiene così il disaccoppiamento tra notifica di potenziale contagio e decisione di infezione vera e propria, replicando la logica della funzione `infect()` del modello originale.

- `UpdateInfectionState()`: aggiorna lo stato delle entità infette, determinando la transizione verso guarigione o decesso in base alle probabilità definite nel modello Python (`recover_or_die()`). Genera `DeathMsg` in caso di decesso e `EntityUpdMsg` in caso di guarigione o immunizzazione, notificando agli altri LP gli aggiornamenti di entità non locali.

- `ScanToBeRemoved()` e `ScanMigrating()`: gestiscono rispettivamente la rimozione delle entità dalle strutture dati e la loro migrazione verso altri LP.

Tutti i messaggi generati durante un timestep vengono ricevuti al **timestep successivo** e gestiti tramite *event handler* dedicati, che aggiornano coerentemente le strutture dati locali e globali garantendo consistenza semantica e corretto avanzamento della simulazione.

Gli stati delle entità (infezione, guarigione, decesso) vengono periodicamente salvati su file, consentendo analisi post-simulazione e debugging. Ogni LP produce un proprio log, mentre uno di essi (tipicamente l'LP 0) funge da *stat reporter* globale.

4.3 Differenze tra implementazione originale e porting

Il porting ha evidenziato differenze significative tra il modello sequenziale in Python e la sua versione distribuita in ARTÌS/GAIA, in termini di approccio concettuale, prestazioni, compatibilità e scalabilità.

Approccio concettuale

Il modello Python utilizza un'unica struttura dati globale (matrice `NumPy`) e gestisce lo stato degli agenti in modo sequenziale tramite confronti spaziali diretti.

Nel modello GAIA, ogni agente è gestito come un'entità autonoma da uno dei vari *Logical Processes*, interagendo tramite scambio asincrono di messaggi. La progressione dello stato è regolata da un paradigma *event-driven*, che ha richiesto una completa ristrutturazione della logica del modello.

Scalabilità

Il modello Python, pur semplice, è efficace su piccola scala grazie alle strutture `NumPy`, ma è vincolato all'uso di un singolo core.

Il modello GAIA consente l'esecuzione distribuita, con agenti suddivisi tra LP diversi e comunicazioni minimizzate tramite clustering adattivo.

Questa architettura introduce un certo overhead, ma consente la scalabilità, soprattutto in ambienti paralleli o distribuiti.

Compatibilità e adattamento

Il porting ha richiesto:

- La suddivisione dello stato tra tabelle locali e globali.
- La gestione degli agenti per sottoinsiemi locali gestiti dai singoli LP.
- L'uso di messaggi asincroni per aggiornamenti e interazioni.

Ciò ha comportato il disaccoppiamento tra logica del modello e rappresentazione delle entità, introducendo pattern asincroni di comunicazione.

Risultati

Per confrontare visivamente l'esito del porting, è stato replicato nella versione ARTÌS/GAIA un visualizzatore grafico analogo a quello utilizzato nel modello Python originale.

Sono state eseguite simulazioni con parametri identici (`population size`, `infection_chance`, `mortality`, `infection_radius`, ecc.), raccogliendo i dati a ogni timestep e generando visualizzazioni che mostrano l'evoluzione temporale e spaziale dell'infezione nelle due versioni del modello.

Durante queste prove, le distribuzioni dell'infezione ottenute nelle due versioni risultano molto simili, con soltanto lievi differenze residue. Entrambe le versioni riproducono coerentemente le dinamiche di contagio, guarigione e mortalità attese in base ai parametri di simulazione, mostrando comportamenti qualitativamente e quantitativamente allineati.

Gli scostamenti osservabili potrebbero essere attribuite a:

- Eventuali minime differenze dovute ai generatori di numeri casuali, sebbene la loro influenza sia verosimilmente trascurabile date le dimensioni e il numero di eventi simulati.

- Il cambio di paradigma tra il modello originale e la versione ARTÌS/GAIA, in cui la logica di aggiornamento dello stato è disaccoppiata e guidata da eventi asincroni. Questa architettura può influire lievemente sulla tempistica delle interazioni tra agenti, motivo per cui, in contesti particolarmente sensibili, potrebbe essere necessario un fine-tuning dei parametri per ottenere sovrapposizioni complete.

A scopo documentale, vengono riportate le immagini seguenti che mostrano un confronto visivo tra le due versioni con parametri identici:

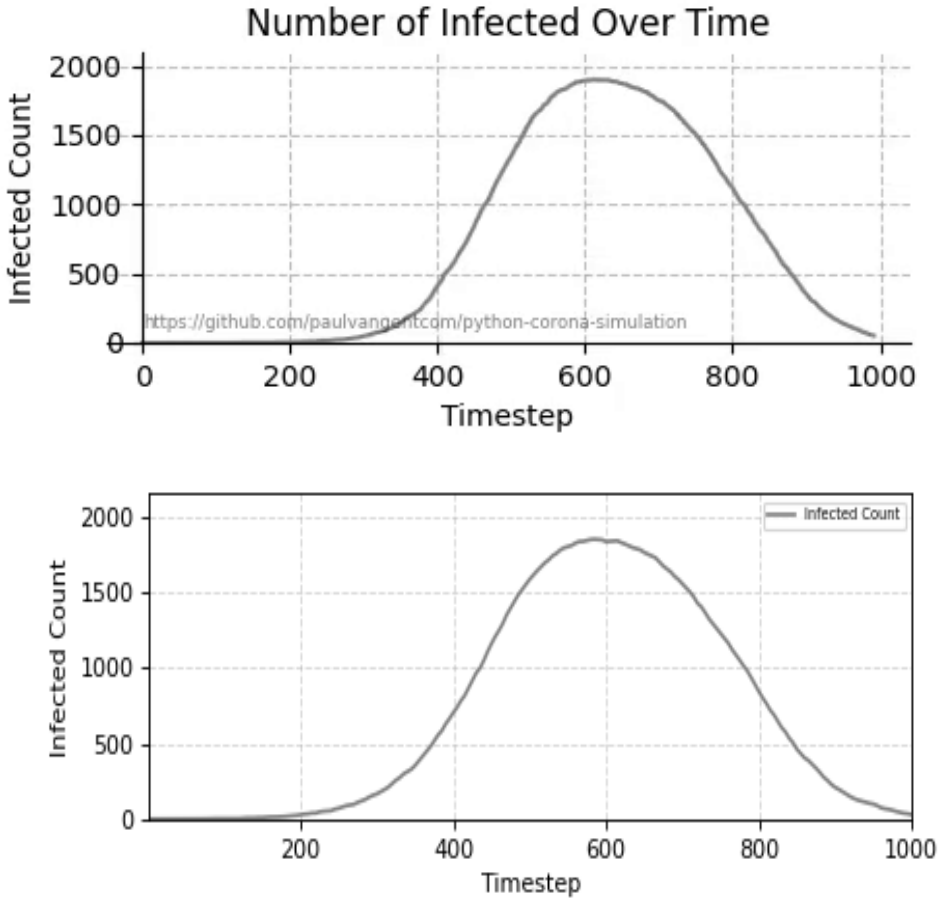


Figura 4.1: Confronto tra il modello originale Python (sopra) e la versione ARTÌS/GAIA (sotto).

A complemento delle visualizzazioni, si riportano anche i log finali dei conteggi delle entità nelle due simulazioni, che confermano la coerenza del comportamento complessivo pur in presenza di minime differenze residue:

Modello originale Python:

```
Healthy: 0
Sick: 47
Immune: 1904
Dead: 49
```

Versione ARTÌS/GAIA:

```
Final Healthy: 1
Final Infected: 29
Final Immune: 1931
Total removed: 39
```

In sintesi, il confronto ha permesso di verificare che l'approccio seguito per il porting mantiene fedelmente le caratteristiche del modello originale, abilitando l'esecuzione in un contesto distribuito e parallelo, con minime differenze che potranno essere oggetto di eventuali analisi di dettaglio in sviluppi futuri.

Considerazioni sulle prestazioni. Al momento non è stata eseguita una valutazione sistematica delle prestazioni tra le due versioni, sebbene in futuro potrà essere realizzata. Tale analisi risulta non banale, in quanto i paradigmi delle due implementazioni sono estremamente diversi: il tempo di esecuzione di una simulazione dipende fortemente dal decorso stesso della simulazione (ad esempio, se tutte le entità muoiono rapidamente, la simulazione si conclude più in fretta). Inoltre, sono state osservate alcune differenze residue tra le due versioni che possono influenzare i tempi, rendendo difficile stabilire una metrica univoca e valida per un confronto diretto delle performance. Eventuali valutazioni future richiederanno quindi un'attenzione specifica sia

alla definizione di casi di test coerenti, sia alla selezione delle metriche più rappresentative per l'obiettivo dell'analisi.

Conclusioni

Lo sviluppo e l'analisi di modelli simulativi mediante ARTÌS/GAIA ha offerto non solo l'opportunità di approfondire i principi teorici delle simulazioni a eventi discreti e dei sistemi multi-agente, ma anche di confrontarsi con le sfide concrete che emergono quando tali principi vengono tradotti in sistemi funzionanti.

Il processo di testing e l'integrazione di strumenti di validazione hanno evidenziato quanto le problematiche relative alla coerenza globale e alla gestione della memoria non siano semplici aspetti tecnici, ma fattori che condizionano l'affidabilità stessa dei risultati simulativi. Le metriche di comunicazione introdotte, come il Local e Global Communication Ratio, si sono rivelate strumenti utili non solo per l'ottimizzazione delle performance, ma anche per comprendere le dinamiche di interazione tra Logical Processes in scenari distribuiti.

Il porting di un modello preesistente verso ARTÌS/GAIA ha permesso di testarne le capacità di adattamento, evidenziandone punti di forza come la scalabilità e la gestione dell'esecuzione distribuita. Allo stesso tempo, è emersa la necessità di un'attenzione particolare nella gestione di eventi complessi, come riproduzione e morte delle entità, che richiedono un certo impegno implementativo. Sebbene il framework operi a un livello che consente un buon controllo sui dettagli del modello, le primitive disponibili possono talvolta richiedere soluzioni specifiche per gestire in modo efficace le dinamiche più articolate dei modelli sviluppati.

Un aspetto che è emerso chiaramente è che il lavoro con ARTÌS/GAIA

non si esaurisce con la realizzazione di un modello funzionante: la progettazione di strumenti di visualizzazione, la realizzazione di validatori automatici e il monitoraggio di metriche interne al sistema simulativo sono componenti fondamentali per rendere una simulazione un vero strumento di esplorazione scientifica, evitando che il modello diventi una "scatola nera" di cui si osservano solo gli output finali.

Infine, questa esperienza ha mostrato come la simulazione distribuita e parallela, pur richiedendo una cura particolare per la gestione dei dettagli implementativi, costituisca un approccio potente e necessario per affrontare modelli complessi in scenari che vanno dall'epidemiologia all'analisi di reti e infrastrutture. Un utilizzo consapevole e metodico degli strumenti disponibili può quindi fare la differenza tra una simulazione puramente descrittiva e uno strumento di analisi scientifica robusto e affidabile.

Appendice

Questa appendice raccoglie materiali tecnici, esempi, tabelle e indicazioni utili per la comprensione e la riproduzione delle attività descritte nella tesi.

Link al repository

Il repository utilizzato per questo lavoro è disponibile pubblicamente all'indirizzo:

`https://github.com/LAWRENCO/ExperienceWith-ARTISGAIA.git`

Modelli simulativi disponibili

All'interno del repository, nella cartella `MODELS`, sono presenti diversi modelli simulativi. I modelli sviluppati e testati progressivamente in questa tesi sono:

- `PCSSimple_Porting`
- `GIDRA`
- `GIDRA_LCR_GCR`

Questi modelli includono tutte le funzionalità sperimentate durante il lavoro di tesi e sono più completi e stabili rispetto a modelli precedenti presenti nel repository.

Comandi e ambiente di esecuzione

All'interno di ciascun progetto è disponibile uno script `./run` per l'esecuzione locale delle simulazioni. Lo script accetta tre parametri:

```
./run #TOT_LP #LP #SMH
```

dove:

- #TOT_LP: numero totale di Logical Processes
- #LP: numero di Logical Processes da eseguire localmente
- #SMH: numero totale di entità da simulare (ogni LP simulerà #SMH / #TOT_LP entità)

Lo script si occupa della compilazione del progetto, rimuovendo i file generati da compilazioni precedenti, ed esegue automaticamente un validatore Python al termine della simulazione. Il validatore, disponibile nella cartella `VALIDATOR`, permette di verificare la correttezza dell'esecuzione. I validatori principali sono `validator.py` e `global_validator.py`, descritti nei capitoli precedenti.

L'esecuzione genera file di output nella cartella `OUTPUT`, alcuni utili per il logging, altri per la visualizzazione dei risultati. In particolare, nel modello `GIDRA_LCR_GCR` lo script esegue automaticamente simulazioni in quattro configurazioni (`REPR ON/OFF`, `DEATH ON/OFF`).

Esempio di output: `GIDRA/OUTPUT/sim_summary.out`

```
Initial Infected: 289
X: 100
Y: 100
N.Entities: 2000
N.Logical Processes: 4
```

```
N.Hellos: 4775336
N.Migrations: 158
Final Infected: 125
Total removed: 2056
Total registrations: 2236
```

Visualizzatori

Nei folder dei modelli sono disponibili script di visualizzazione, tra cui:

- `position_visualizer.py`
- `genetic_visualizer.py`
- `communication_visualizer.py`

Questi possono essere eseguiti manualmente o tramite lo script `./visualize`, che accetta la sintassi:

```
./visualize [gen|pos|comm] [v|i]
```

dove il primo parametro seleziona il tipo di visualizzazione e il secondo consente di scegliere tra la generazione di un video (`v`) o delle sole immagini iniziali e finali (`i`). Alcuni modelli eseguono automaticamente il visualizzatore alla fine della simulazione, come avviene in `GIDRA LCR_GCR`.

Parametri di simulazione

Di seguito un esempio dei parametri utilizzati per le simulazioni sul modello `GIDRA`:

Parametro	**Valore**
Dimensione popolazione	240
Timestep massimi	5000
Food capacity	250
Standard Immunity	0.5
Standard Fertility	0.05
Reproduction Age	100
Death Age	450
Max Infection Level	100
Starving Hunger	15
Probabilità di morte	0.05
Abilitazione riproduzione	ON
Abilitazione morte	ON
Abilitazione migrazione	ON

Tabella 4.1: Esempio di parametri sperimentali utilizzati col modello `GIDRA`.

Messaggi definiti nel modello

Messaggio	**Descrizione**
`HelloMsg`	Messaggio di ping verso un'entità
`MoveMsg`	Notifica di cambio posizione di un'entità
`InfectionMsg`	Messaggio di contagio verso un'entità sana
`EntityUpdMsg`	Aggiornamento di stato (guarigione, contagio remoto)
`DeathMsg`	Segnalazione della morte di un'entità

Tabella 4.2: Messaggi personalizzati utilizzati nei modelli.

Glossario dei termini

ABS (Agent-Based Simulation) Approccio alla simulazione in cui il sistema è modellato come un insieme di entità autonome (agenti) che interagiscono tra loro e con l'ambiente secondo regole definite.

ARTÌS (Advanced RTI System) Framework per la simulazione parallela e distribuita a eventi discreti, che fornisce servizi per la gestione della sincronizzazione, comunicazione e gestione del tempo tra Logical Processes.

DES (Discrete Event Simulation) Tecnica di simulazione in cui il sistema viene modellato come una sequenza di eventi discreti nel tempo, ciascuno dei quali provoca un cambiamento nello stato del sistema.

Entity (Entità) Elemento base del modello simulativo, che rappresenta un individuo del sistema dotato di stato e comportamento.

Event (Evento) Unità di cambiamento nello stato del sistema simulato, che avviene in un istante specifico e modifica lo stato di una o più entità.

GAIA (Generic Adaptive Interaction Architecture) Componente software che, insieme ad ARTÌS, fornisce servizi avanzati per il bilanciamento del carico e la gestione adattativa delle partizioni durante la simulazione.

LP (Logical Process) Unità di elaborazione logica in una simulazione parallela o distribuita, responsabile della gestione di un sottoinsieme di entità e dell'elaborazione degli eventi ad esse associati.

LCR (Local Communication Ratio) Rapporto tra i messaggi scambiati localmente all'interno di un LP e il totale dei messaggi inviati, utile per valutare l'efficienza della partizione.

MAS (Multi-Agent System) Sistema in cui più agenti interagiscono tra loro con comportamento autonomo e potenzialmente emergente.

PADS (Parallel and Distributed Simulation) Tecniche per l'esecuzione di simulazioni in modo parallelo e distribuito, suddividendo il carico computazionale tra più processi o macchine.

Simclock Tempo simulato corrente in un dato istante dell'esecuzione della simulazione.

State (Stato) Descrizione delle caratteristiche attuali di un'entità, variabile in seguito all'elaborazione di eventi.

Bibliografia

[1] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2010.

[2] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *Wireless communications and mobile computing*, 2(5):483–502, 2002.

[3] K.M. Chandy, J. Misra, and R.E. Bryant. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[4] Byoung Kyu Choi and DongHun Kang. *Modeling and Simulation of Discrete Event Systems*. John Wiley & Sons, Incorporated, 2013. Accessed via ProQuest Ebook Central.

[5] Gabriele D'Angelo. Adaptive partitioning strategies for the parallel and distributed simulation of dynamic agent-based systems. In *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, pages 33–40, 2014.

[6] Gabriele D'Angelo and Stefano Ferretti. Parallel and distributed simulation from many cores to the public cloud: Experiments with the artÌs/gaia framework. *Concurrency and Computation: Practice and Experience*, 30(15), 2018.

[7] Gabriele D'Angelo and Moreno Marzolla. High-level simulation of large-scale iot scenarios with adaptive partitioning and load balancing. *Simulation Modelling Practice and Theory*, 108:102241, 2021.

[8] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[9] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley, 2000.

[10] IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla)–framework and rules, 2010. IEEE Std 1516-2010.

[11] David R. Jefferson. Virtual time. In *Proceedings of the 7th annual symposium on Principles of distributed computing*, pages 33–45, 1985.

[12] Ulrich Kutschera and Karl J. Niklas. The modern theory of biological evolution: an expanded synthesis. *Naturwissenschaften*, 102(7-8):1–20, 2015.

[13] David M. Nicol. Time warp vs. chandy/misra: A comparative experimental study. *ACM Transactions on Modeling and Computer Simulation*, 1(2):153–193, 1989.