ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

SIMULAZIONE DI ALGORITMI DI AUTO-ORGANIZZAZIONE BASATI SU GRADIENTE COMPUTAZIONALE IN ALCHEMIST

Elaborata nel corso di: Linguaggi e Modelli Computazionali L-M

Tesi di Laurea di:

ENRICO POLVERELLI

Relatore:
Prof. MIRKO VIROLI

Correlatori:
Ing. DANILO PIANINI
Ing. SARA MONTAGNA

ANNO ACCADEMICO 2011-2012
SESSIONE III

Alla mia famiglia, ai miei amici

e a tutti quelli che mi sono stati vicini

in questi tre anni di studi.

Indice

Introduzione **1**

1 Design pattern per sistemi auto-organizzanti **5**
1.1 Auto-organizzazione 5
1.2 Sistemi auto-organizzanti 5
1.3 Modelli computazionali per sistemi auto-organizzanti 6
1.4 Design pattern come metodologia. 8
1.5 Meccanismi, pattern ed uso di pattern 11
1.5.1 Meccanismi 11
1.5.1.1 Spreading pattern 12
1.5.1.2 Aggregation pattern 14
1.5.1.3 Evaporation pattern 16
1.5.2 Pattern 18
1.5.2.1 Gradient pattern 18
1.5.2.2 Digital pheromone pattern 20
1.5.3 Pattern di alto livello. 21
1.5.3.1 Ant foraging pattern. 22
1.5.3.2 Chemiotaxis pattern. 23
1.5.3.3 Morphogenesis pattern 24
1.5.3.4 Quorum sensing pattern. 25

2 Sistemi pervasivi **27**
2.1 Definizione e requisiti. 27
2.2 SAPERE: un framework per sistemi pervasivi 29
2.2.1 Architettura e linguaggio 29

3 Alchemist **33**
3.1 Motivazioni 33
3.2 Modello computazionale 35
3.3 Architettura. 37
3.4 Motore di simulazione 38
3.5 Domain Specific Language 40

4 Performance di Alchemist **45**
4.1 Gradiente YOUNGEST. 45

4.1.1 Descrizione del gradiente 45
4.1.2 Performance del gradiente. 47
4.2 Gradiente SAPERE 54
4.2.1 Descrizione del gradiente 54
4.2.2 Performance del gradiente. 54
4.3 Confronto fra il gradiente YOUNGEST e il gradiente SAPERE. 58

5 Pattern sviluppati 63
5.1 Segregation pattern 63
5.1.1 Modello 63
5.1.2 Scenari applicativi 65
5.1.3 Simulazione del pattern in Alchemist 65
5.1.3.1 Scenario 1 66
5.1.3.2 Scenario 2 67
5.1.3.3 Scenario 3 71
5.2 Alarm Sensor Network 73
5.2.1 Modello 73
5.2.2 Scenari applicativi 78
5.2.3 Simulazione del pattern in Alchemist 78
5.3 Gradiente con carico 81
5.3.1 Modello 81
5.3.2 Scenari applicativi 83
5.3.3 Simulazione del pattern in Alchemist 83
5.3 Traccia computazionale. 86
5.4.1 Modello 86
5.4.2 Scenari applicativi 89
5.4.3 Simulazione del pattern in Alchemist 90
5.4.3.1 Scenario 1 90
5.4.3.2 Scenario 2 93

6 Conclusioni e sviluppi futuri 95
2.1 Riassunto e contributi 95
2.2 Sviluppi futuri. 96
2.2.1 Architettura e linguaggio . . . .

Bibliografia 97

Appendice A: DSL per Segregation Pattern 99

A.1 DSL per Scenario 1 99
A.2 DSL per Scenario 2 99
A.3 DSL per Scenario 3 101

Appendice B: DSL per Alarm Sensor Network 103

Appendice C: DSL per Gradiente con Carico 107

Appendice D: DSL per Traccia computazionale 111

Introduzione

I sistemi computazionali moderni sono sempre più complessi, in particolare si sta assistendo all'esplosione dei sistemi pervasivi, cioè sistemi computazionali distribuiti su larga scala che comprendono al loro interno un elevato numero di componenti e dispositivi eterogenei fra loro. Dal punto di vista ingegneristico è quindi importante assicurarsi che questi sistemi abbiano diverse proprietà come scalabilità, reattività, tolleranza ai guasti unite a buone performance. Realizzare queste proprietà tramite un coordinamento centrale è praticamente impossibile su sistemi grandi come i sistemi pervasivi, sorge quindi la necessità di un nuovo approccio che le garantisca, questo approccio prende il nome di auto-organizzazione.

L'auto-organizzazione è la capacità di un sistema di evolvere strutture spaziali o temporali senza l'intervento di un coordinamento centrale ma solo grazie alle interazioni locali degli elementi del sistema stesso che porteranno all'emergenza del comportamento globale desiderato.

Nel campo dell'auto-organizzazione si fa spesso riferimento a sistemi naturali e se ne simulano i meccanismi, questi prendono il nome di design pattern. I design pattern trovano, in campo ingegneristico, uno spettro molto ampio di applicazione ma in questa tesi ci si è concentrati su una specifica classe di applicazione, quella dei sistemi pervasivi.

I sistemi pervasivi sono sistemi in cui è immerso l'ambiente in cui viviamo ed in cui sono presenti una grande varietà di componenti ed attori, tutti identificati con il nome termine *individui*. La grande dinamicità ed imprevedibilità dei sistemi pervasivi rende necessario un approccio auto-organizzante per garantire al sistema le proprietà sopra descritte.

L'obiettivo di questa tesi è quindi quello di sviluppare dei pattern di auto-organizzazione, basati su gradiente computazionale, che possano essere

utili per la progettazione dei sistemi pervasivi al fine di garantirne le proprietà desiderate.

Il gradiente computazionale è una struttura dati spaziale che ha origine in un certo nodo della rete detto “sorgente” e viene diffuso in tutti gli altri nodi della rete, grazie alla diffusione in tutti gli altri nodi della rete sarà presente l'indicazione della distanza minima fra il nodo e la sorgente. In letteratura esistono diversi tipi di gradiente, in questa tesi si è analizzato il gradiente YOUNGEST ed il gradiente SAPERE, dal punto di vista delle performance, per decidere quale fosse il gradiente più adatto per lo sviluppo degli algoritmi di questa tesi. Il gradiente SAPERE si è dimostrato il più adatto in quanto nettamente più performante dello YOUNGEST soprattutto nelle situazioni in cui diversi gradienti devono essere combinati insieme, requisito chiave per lo sviluppo dei pattern.

Un altro scopo della tesi è quello di dimostrare il ruolo fondamentale della simulazione nella progettazione dei sistemi pervasivi, per modellare e simulare gli algoritmi di questa tesi è stato usato Alchemist, un framework di simulazione di ispirazione bio-chimica. Alchemist è stato scelto per le sue performance elevate, per la semplicità di modellazione resa possibile dal suo Domain Specific Language e per la facilità con cui gli algoritmi modellati possono essere simulati.

In questa tesi sono stati sviluppati quattro algoritmi di auto-organizzazione che corrispondono ad altrettanti design pattern per la coordinazione spaziale: Segregation, Alarm Sensor Network, Gradiente con carico e Traccia computazionale.

Il pattern di Segregation viene applicato in presenza di diversi gradienti, divide la rete in porzioni in cui ogni nodo sarà a distanza minima dalla sorgente più vicina.

Alarm Sensor Network è un pattern avanzato per la generazione di un allarme su una rete di sensori. L'idea di base è quella di generare un gradiente per ogni valore errato rilevato dal sensore, se in un certo nodo è presente un numero sufficientemente elevato di gradienti allora viene generato un allarme,

sempre sotto forma di gradiente, che verrà diffuso sull'intera rete secondo diverse politiche.

Gradiente con carico è un pattern che ha come scopo quello di distribuire un certo numero di agenti mobili, in numero più o meno uguale, su tutte le sorgenti di gradiente presenti. L'idea che sta alla base di questo pattern è il peggioramento del valore della sorgente ogni volta che un agente la raggiunge.

Traccia computazionale ha come scopo quello di predire quale sarà il percorso seguito dagli agenti mobili nella risalita del gradiente, questo permette di generare una traccia che porta con sé informazioni utili per capire se nel futuro ci saranno zone della rete affollate ed eventualmente agire di conseguenza.

L'elaborato è composto da sei capitoli:

- Capitolo 1: viene definito con maggiore dettaglio il concetto di auto-organizzazione, vengono definiti i meccanismi che stanno alla base dei design pattern, vengono descritti i design pattern presenti in letteratura e come possono essere combinati per creare design pattern più complessi.
- Capitolo 2: in questo capitolo vengono descritti gli ecosistemi pervasivi in cui i design pattern precedentemente definiti andranno ad agire, con particolare attenzione al ruolo del gradiente computazionale.
- Capitolo 3: viene descritto il framework Alchemist e il linguaggio usato per modellare gli algoritmi, viene inoltre motivata la scelta di ricorrere ad Alchemist per lo sviluppo di questa tesi.
- Capitolo 4: viene presentata un'analisi delle performance del framework Alchemist, come dimostrazione delle motivazioni presenti nel precedente capitolo. L'analisi sarà concentrata in particolare sulle performance dei vari gradienti computazionali usati in Alchemist.
- Capitolo 5: vengono presentati gli algoritmi prodotti durante lo sviluppo di questa tesi, ogni algoritmo sarà descritto in modo formale e saranno

illustrati gli scenari in cui può essere utile usare tale algoritmo e verranno presentate le simulazioni dell'algoritmo prodotte in Alchemist.

- Capitolo 6: verrà ricapitolato quello che è stato fatto nella tesi, quello che la tesi ha prodotto ed i possibili sviluppi futuri dei design pattern in generale e di quelli sviluppati in particolare.

I capitoli 1, 2 e 3 sono tratti principalmente dalla letteratura, con riferimento ad [1], [2], [3], [4] e [5], mentre i contributi veri e propri alla tesi sono nei capitoli 4 e 5.

Capitolo 1

Design pattern per sistemi auto-organizzanti

Prima di poter iniziare è fondamentale definire alcuni concetti che stanno alla base di questa tesi e che sono fondamentali per la sua comprensione. In questo capitolo verrà quindi definito il concetto di auto-organizzazione e di sistema auto-organizzante e verrà discusso il ruolo che hanno i design pattern in tali sistemi

1.1 Auto-organizzazione

L'auto-organizzazione è quell'insieme di meccanismi che permette ad un sistema di esibire dinamiche complesse senza avere la completa conoscenza di tutte le informazioni, grazie alle interazioni locali dei suoi componenti.

In natura sono presenti molti sistemi biologici che esibiscono aspetti di auto-organizzazione, fra i più famosi citiamo la ricerca del cibo e la costruzione di nidi nelle colonie di formiche, che sono stati presi come modello per lo sviluppo di sistemi artificiali complessi.

1.2 Sistemi auto-organizzanti

Un moderno sistema computazionale è soggetto ad un continuo aumento di scala, per questo è fondamentale che sia auto-organizzante in modo che possa gestire cambiamenti e guasti in modo robusto. Di solito per realizzare

l'auto-organizzazione in un sistema si fa in modo che ogni entità del sistema sia autonoma e pro-attiva e si pone il focus sulle interazioni fra queste entità. Questo implica decentralizzazione, cioè la mancanza di un coordinatore centrale che gestisca le altre entità del sistema e località, cioè ogni entità ha conoscenza solo delle informazioni relative al suo vicinato ma non dell'informazione a livello globale.

Realizzare questi sistemi non è come realizzare i tradizionali sistemi software dato che sono presenti dinamiche non lineari ed i parametri interni di ogni tipo di entità possono influire pesantemente sul comportamento dell'entità quanto del sistema stesso. Per capire come progettare le singole entità al fine di ottenere il comportamento globale desiderato e l'emergenza del pattern specifico è necessario introdurre un modello computazionale di questi sistemi, così da poterli trattare in modo formale.

1.3 Modelli computazionali per sistemi auto-organizzanti

Nei sistemi biologici che presentano caratteristiche di auto-organizzazione si possono osservare due entità principali: gli organismi che prendono parte al processo biologico e l'ambiente in cui sono situati. L'ambiente fornisce risorse ed eventi esterni che cambiano lo stato del sistema. Gli organismi possono osservare l'ambiente, modificarlo e comunicare fra loro sia in modo diretto che indiretto, cioè depositando nell'ambiente informazioni che poi saranno percepite da altri. Come mostra la figura 1.1 il modello biologico di un sistema auto-organizzante è diviso in due layer:

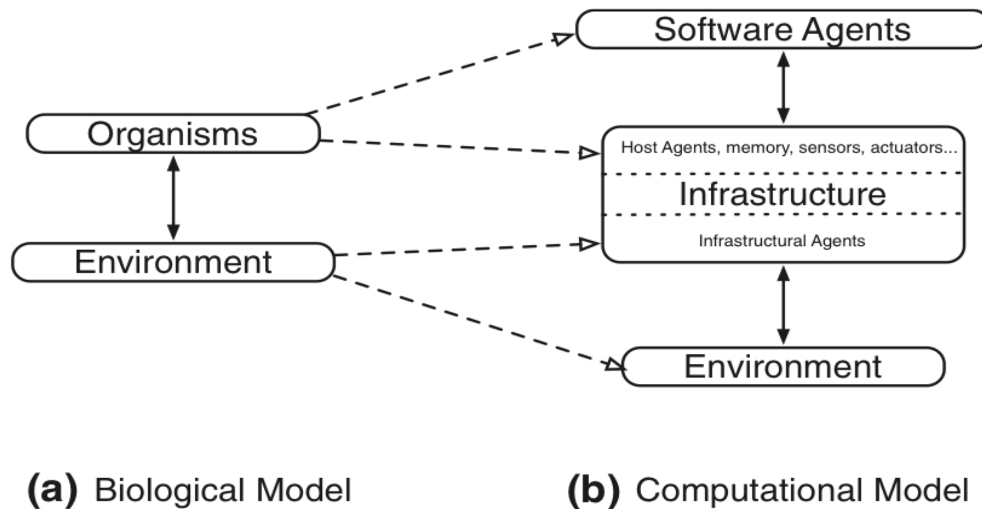


Figura 1.1: Entità del modello biologico e del modello computazionale. Immagine tratta da [1]

Per passare dal modello biologico a quello computazionale è necessario introdurre un ulteriore layer di infrastruttura, dato che un agente software deve essere inserito in un dispositivo dotato di capacità computazionale in modo da poter interagire con l'ambiente e comunicare con gli altri agenti. Il modello computazionale è quindi composto da tre entità:

- Agenti: entità software proattive ed autonome, che rappresentano gli organismi del modello biologico.
- Infrastruttura: contiene sensori e attuatori, cioè gli host dotati di potenza computazionale.
- Ambiente: il mondo reale in cui è situato il sistema, assolutamente analogo all'ambiente del modello biologico.

Gli eventi del modello computazionale vengono percepiti dagli agenti tramite la potenza computazionale fornita dall'infrastruttura che quindi permette agli agenti di simulare il comportamento degli organismi, oltre a fornire un luogo in cui l'informazione può essere memorizzata o letta. L'infrastruttura dovrà inoltre farsi carico di simulare la capacità dell'ambiente biologico di agire sulle entità presenti nel sistema, questo è possibile grazie agli agenti infrastrutturali. Gli

agenti infrastrutturali possono muoversi liberamente fra i vari host e sono responsabili della gestione delle informazioni depositate dagli agenti nei vari host e della diffusione dell'informazione verso altri host (come nel caso dello spreading), oltre a fornire funzionalità built-in all'interno di un middleware (come nel caso dell'evaporazione del feromone digitale).

Un sistema computazione auto-organizzante è quindi composto dagli agenti, dall'infrastruttura, dagli agenti infrastrutturali, dagli host e dall'ambiente che lo ospita, come mostrato in figura 1.2:

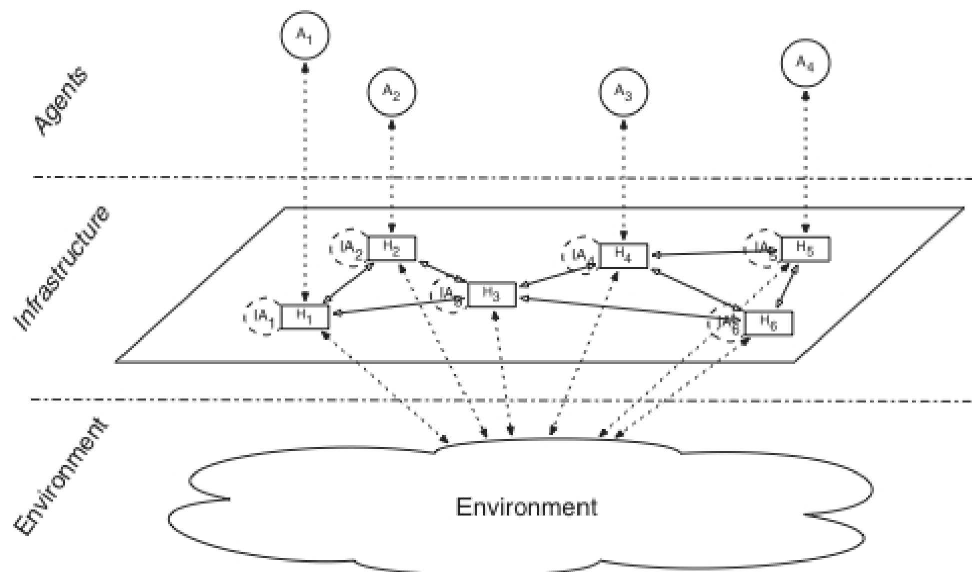


Figura 1.2: Modello computazionale completo. Immagine tratta da [1]

Il comportamento degli agenti è definito da un insieme di regole, chiamate regole di transizione, mentre gli host sono definiti dall'interfaccia che forniscono.

1.4 Design pattern come metodologia

Per costruire in sistema auto-organizzante si segue la metodologia classica dell'ingegneria del software: requisiti, analisi, progettazione, implementazione e collaudo, i design pattern auto-organizzativi svolgono un

ruolo fondamentale nella fase di progettazione che, come mostrato in figura 1.3, può essere a sua volta divisa in tre fasi:

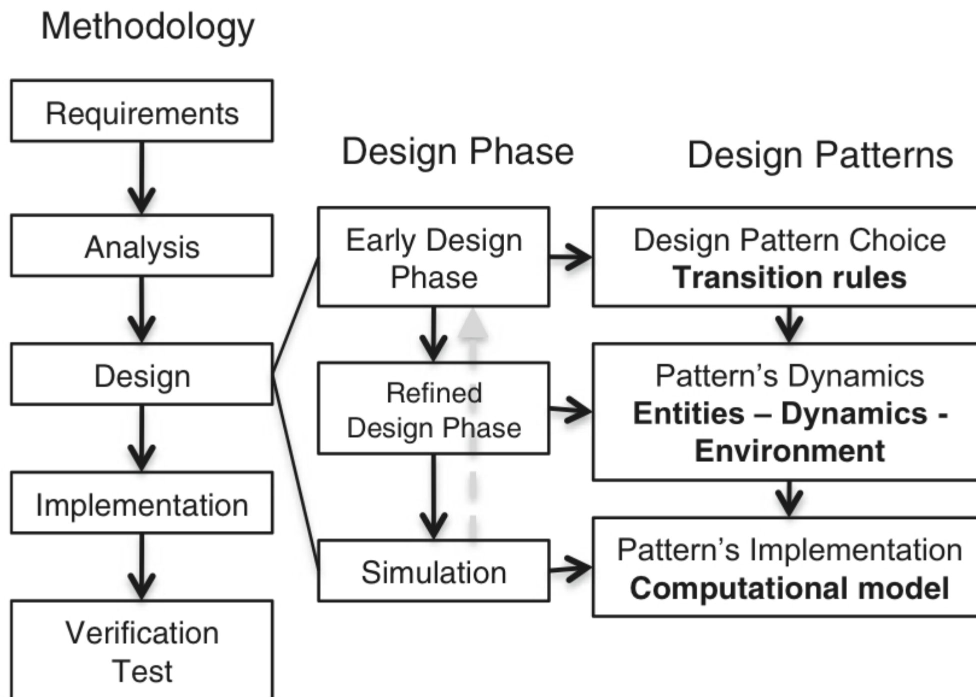


Figura 1.3: uso dei design pattern all'interno della fase di progettazione. Immagine tratta da [1]

Nella prima fase vengono scelti i design pattern appropriati per la costruzione del sistema in esame, in modo da definire i limiti di ogni problema e la corrispondente soluzione fornita dal pattern opportuno.

Nella seconda fase, detta di raffinamento, vengono identificate le entità in gioco e le interazioni necessarie fra le varie entità al fine di ottenere l'emergenza del comportamento desiderato.

Nella terza ed ultima fase, detta di simulazione, si specifica la descrizione dell'implementazione dei pattern. In questa fase è fondamentale effettuare il giusto tuning dei parametri per ogni pattern dato che, come già detto, possono influire in modo considerevole sul comportamento risultante.

Scegliere il pattern corretto per ogni problema è fondamentale quindi in letteratura esiste un catalogo di pattern, con comportamenti ben noti e descritti

formalmente, che possono facilitare questa scelta. Questo catalogo è diviso in tre layer, come mostrato in figura 1.4:

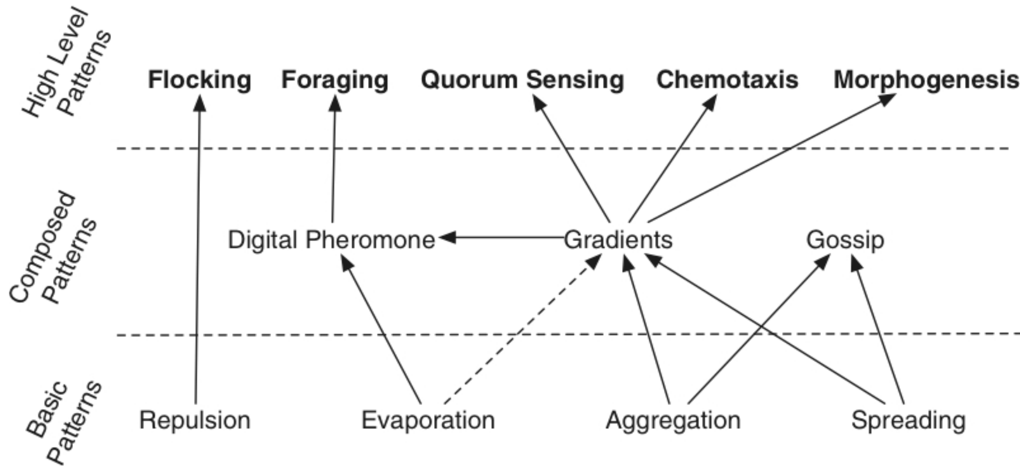


Figura 1.4: Catalogo dei pattern. Immagine tratta da [1]

- Meccanismi (o pattern atomici): i meccanismi di base possono essere usati singolarmente o possono essere combinati per creare pattern più complessi, è il livello più basso.
- Pattern: livello intermedio, sono pattern composti dai meccanismi del livello inferiore.
- Uso dei pattern: i pattern del livello intermedio possono essere usati per creare pattern di alto livello.

Le frecce in figura 1.4 indicano quali sono le relazioni fra i vari livelli, quelle continue indicano che un meccanismo/pattern è essenziale per la realizzazione del pattern di più alto livello mentre quelle tratteggiate indicano che la relazione è opzionale, ad esempio per realizzare il pattern Gradient è necessario combinare i pattern di Aggregation e Spreading mentre quello di Evaporation è opzionale.

Il comportamento di ogni pattern viene descritto tramite un'insieme di regole di transizione applicabili alle informazioni presenti nel sistema, queste regole sono rappresentate secondo la seguente notazione:

- Ogni informazione presente nel sistema è rappresentata come una tupla <L,C> dove L è la posizione in cui la tupla viene memorizzata mentre C indica il suo contenuto corrente
- Ogni regola di transizione è rappresentata come una reazione chimica sulle tuple presenti nel sistema, più formalmente:

$$name :: \langle L_1, C_1 \rangle, \ldots, \langle L_n, C_n \rangle \xrightarrow{r} \langle L'_1, C'_1 \rangle, \ldots, \langle L'_m, C'_m \rangle$$

- *name* è il nome della regola di transizione.
- Il lato sinistro della regola rappresenta i reagenti, cioè le tuple che dovranno essere presenti per far si che la reazione possa avvenire. L'esecuzione della regola comporterà la rimozione dei reagenti.
- Il lato destro rappresenta i prodotti, cioè le tuple che verranno inserite come conseguenza della reazione, le tuple inserite potranno essere nuove tuple, reagenti modificati o inalterati.
- Il rate *r* rappresenta a quale frequenza viene schedulata la reazione.

1.5 Meccanismi, pattern ed uso dei pattern

In questo sottocapitolo della tesi verranno descritti i meccanismi ed i pattern utili a capire le successive parti di questa tesi, oltre a fornire una panoramica dei pattern di alto livello che possono essere realizzati usando i pattern. Per semplicità non saranno descritti i meccanismi, i pattern ed i pattern di alto livello che non sono in relazione con il pattern Gradient.

1.5.1 Meccanismi

I meccanismi, detti anche pattern di base o pattern atomici, sono pattern molto semplici ma che trovano numerose applicazioni e sono fondamentali nella costruzione di pattern più complessi.

1.5.1.1 Spreading pattern

Lo spreading pattern è basato sulla comunicazione diretta fra agenti per l'invio progressivo di informazioni sul sistema in modo che ogni agente possa incrementare la sua conoscenza globale del sistema stesso. Lo spreading è conosciuto anche come diffusione, broadcast o flooding.

Lo spreading pattern è applicabile in tutti i sistemi in cui gli agenti compiono solo interazioni locali e non hanno conoscenza globale del sistema, per risolvere il problema ogni agente invia una copia delle sue informazioni a tutti i vicini e così l'informazione viene propagata da un nodo all'altro per tutto il sistema. L'informazione che viene diffusa aumenta la conoscenza globale di ogni agente mantenendo inalterati i vincoli di interazione locale.

Lo spreading è un meccanismo fondamentale usato in quasi tutti i pattern di più alto livello che verranno descritti successivamente.

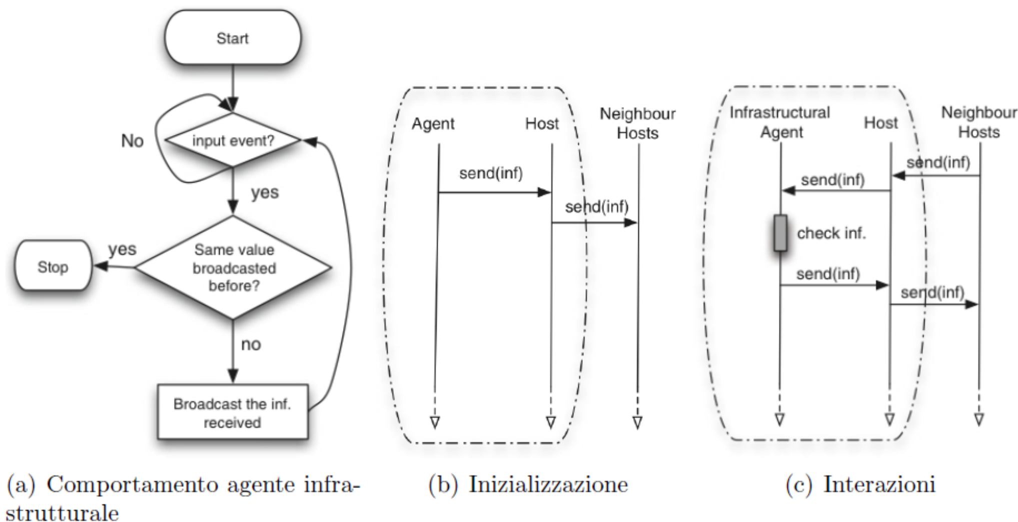
Nello spreading pattern è fondamentale trovare il giusto compromesso fra la velocità di diffusione ed il numero di messaggi: se si vuole un sistema molto reattivo ai cambiamenti si dovrà però tenere conto dell'alto numero di messaggi inviati che potrebbero intasare il sistema stesso. Esistono diversi modi per ridurre il numero di messaggi, ad esempio si può introdurre un numero massimo di passi che ogni messaggio può compiere oppure si può limitare il numero di vicini che riceveranno un messaggio da un certo nodo.

Le entità coinvolte nello spreading pattern sono host, agenti infrastrutturali ed agenti. L'agente fa partire lo spreading inviando l'informazione all'host in cui risiede, a questo punto è compito degli agenti infrastrutturali ri-diffondere l'informazione ai nodi vicini, tenendo conto di un eventuale numero massimo di passi o numero particolare di vicini da considerare, fino a raggiungere ogni nodo del sistema. Si dovranno inoltre evitare cicli infiniti e la duplicazione non necessaria delle informazioni. Più formalmente lo spreading pattern è descritto con la seguente regola:

$$spreading :: \langle L, C \rangle \xrightarrow{r_{spr}} \langle L_1, C_1 \rangle, ..., \langle L_n, C_n \rangle$$
$$where\ (L_1; ...; L_n) = \upsilon\,(L)\,, (C_1; ...; C_n) = \sigma\,(C, L)$$

dove υ(L) è una funzione che determina la sequenza di posizioni, nel vicinato di L, in cui l'informazione può essere diffusa. L'insieme di tali posizioni non può essere vuoto e non può essere composto solo da L. La funzione σ(C,L) elabora il contenuto della nuova informazione, che può cambiare durante il processo.

Per realizzare materialmente lo spreading viene usato un algoritmo di broadcast ottimizzato, in modo da evitare problemi di ridondanza, collisione o contesa, questa ottimizzazione può seguire un approccio probabilistico basato sulla posizione, sul conteggio o sulla distanza. Nella figura sottostante viene descritto il comportamento degli agenti strutturali così come è definito in letteratura:



(a) Comportamento agente infrastrutturale (b) Inizializzazione (c) Interazioni

Figura 1.5: (a) Comportamento degli agenti infrastrutturali, (b) Inizializzazione corrispondente, (c) interazione con gli host e gli host del vicinato. Immagine tratta da [1]

Lo spreading, oltre ad essere un meccanismo fondamentale per la composizione di pattern più complessi trova applicazione nella S*warm Motion Coordination* ed in numerosi problemi di ottimizzazione e coordinazione.

1.5.1.2 Aggregation pattern

L'aggregation pattern è un meccanismo di aggregazione delle informazioni, introdotto per sintetizzare le informazioni significative e ridurre il loro volume nel sistema al fine di evitare sovraccarichi di rete o di memoria.

L'aggregazione consiste quindi nell'applicazione locale di un operatore, detto operatore di fusione, che processa diverse informazioni e produce una macro informazione. Questo operatore può assumere diverse forme, le più note sono il filtraggio, il merging, l'aggregazione e la trasformazione.

Questo pattern ha tratto ispirazione dal processo biologico usato dalle formiche per raggiungere una fonte di cibo: l'aggregazione delle tracce di feromone rilasciate dalle formiche permette l'emergenza del percorso più breve dal formicaio alla fonte di cibo mentre i percorsi più lunghi vengono scartati.

L' aggregazione in natura viene effettuata dall'ambiente mentre nei sistemi computazionali è compito degli agenti o degli agenti infrastrutturali che aggregano l'informazione a cui accedono localmente. Normalmente un agente riceve informazioni dall'host in cui risiede, queste informazioni possono essere lette dall'host nell'ambiente, come nel caso di un sensore, o ricevute dagli host vicini, in ogni caso l'agente esegue l'aggregazione e termina, il processo non può essere ciclico. Più formalmente l'aggregation pattern può essere descritto tramite la seguente regola:

$$aggregation :: \langle L, C_1 \rangle, ..., \langle L, C_n \rangle \xrightarrow{r_{aggr}} \langle L, C'_1 \rangle, ..., \langle L, C'_m \rangle$$
$$where \, \{C'_1, ..., C'_m\} = \alpha\left(\{C_1, ..., C_n\}\right)$$

l'insieme di informazioni in input vengono aggregate secondo la funzione di aggregazione α per produrre in output un insieme di informazioni di cardinalità minore rispetto a quelle che avevo in input. La funzione α può essere implementata in vari modi, a seconda che voglia realizzare:

- Filtro: viene selezionato un sotto insieme delle informazioni ricevute.

- Trasformazione: viene cambiato il tipo di informazione ricevuta in input.
- Merging: viene unificata tutta l'informazione ricevuta in input per produrre un singolo pezzo di informazione.
- Aggregazione: viene applicato un operatore specifico alle informazioni ricevute in input.

Nella figura sottostante viene illustrato il comportamento di un agente aggregatore così come è descritto in letteratura:

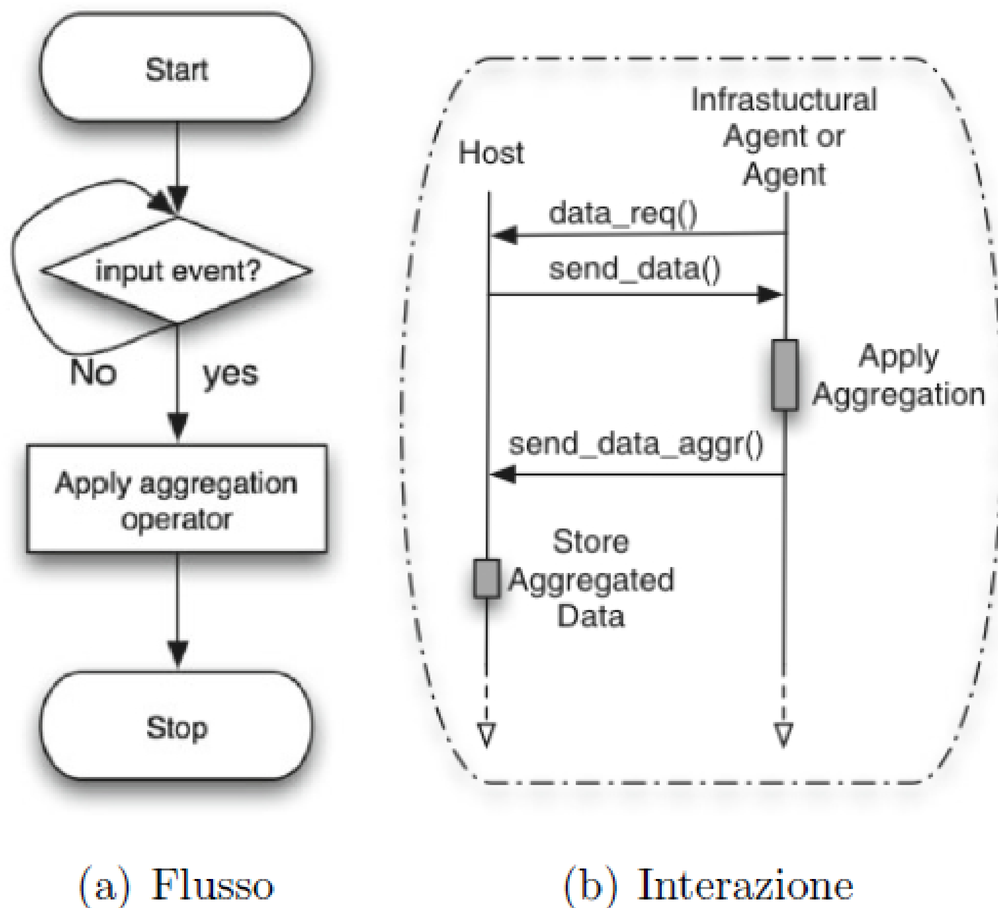


Figura 1.6: Comportamento di un agente aggregante. Immagine tratta da [1]

L'aggregazione, oltre ad essere un meccanismo fondamentale per la costruzione di pattern più complessi è alla base dell'algoritmo di *Ant Colony*

Optimization (ACO) e nello studio dell'aggregazione di basi di credenze individuali in una collettiva.

1.5.1.3 Evaporation pattern

L'evaporation pattern è un meccanismo che permette di gestire gli ambienti dinamici in cui l'informazione può degradare o diventare obsoleta.

Questo meccanismo riduce progressivamente la pertinenza di una certa informazione con il passare del tempo in modo che gli agenti presenti nel sistema abbiano sempre come conoscenza principale quella prodotta per ultima, cioè quella più aggiornata. La maggiore rilevanza delle informazioni più recenti rispetto a quelle più datate permette all'agente di decidere sempre sulla base di informazioni consistenti invece che sulla base di informazioni obsolete.

Anche questo meccanismo, così come l'aggregazione, è ispirato al feromone delle formiche: dopo un certo tempo il feromone depositato sul terreno evapora, facendo così scomparire un certo percorso dove non passano più formiche, questo permette l'emergenza del percorso più breve per arrivare alla fonte di cibo.

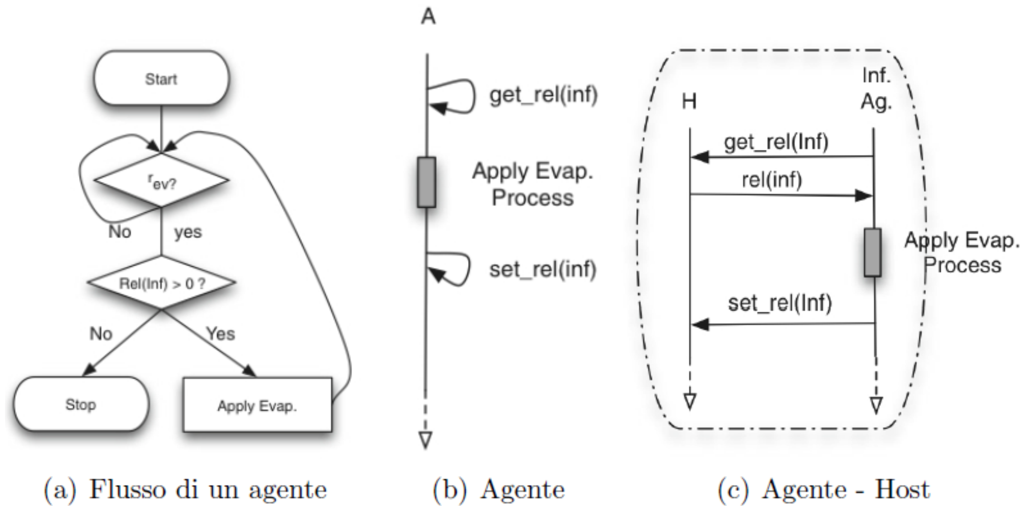
Nell'usare questo pattern è fondamentale decidere il giusto fattore di evaporazione e la frequenza di evaporazione, questi parametri dipendono dalla dinamicità dell'ambiente: un'evaporazione troppo veloce rischia di far perdere l'informazione mentre una troppo lenta rischia di dare troppa rilevanza ad informazioni obsolete mentre un alto fattore di evaporazione richiede meno memoria ma l'agente ha una conoscenza minore sul sistema.

L'evaporazione viene eseguita da agenti o da agenti strutturali con una certa frequenza eseguendo questa regola:

$$\begin{aligned} evaporation ::\langle L, C\rangle &\xrightarrow{r_{ev}} \langle L, C'\rangle \\ where\ C' &= \epsilon\,(C) \end{aligned}$$

dove ε(C) è la funzione che attribuisce una rilevanza decrescente nel tempo all'informazione contenuta in C.

L'evaporazione può essere eseguita da un agente, per aggiornare l'informazione posseduta localmente o da un agente infrastrutturale per aggiornare le informazioni presenti nell'ambiente, in questo caso è l'agente infrastrutturale che dialoga con l'host per far decadere la rilevanza dell'informazione che contiene, questo dialogo è possibile grazie ad una interfaccia fornita dall'host come illustrato in figura 1.7(c):



(a) Flusso di un agente (b) Agente (c) Agente - Host

Figura 1.7: (a) comportamento di un agente, (b) evaporazione locale, (c) evaporazione effettuata dall'host. Immagine tratta da [1]

Oltre ad essere un meccanismo di base per la creazione di pattern più complessi l'evaporazione è usata negli algoritmi di ottimizzazione dinamica come il già citato ACO oppure il *Quantum Swarm Optimisation Evaporation* (QSOE).

1.5.2 Pattern

In questa parte del capitolo si introdurrà il pattern come composizione di meccanismi, verrà descritto in particolare il pattern gradient, pattern fondamentale per lo sviluppo di questa tesi.

1.5.2.1 Gradient pattern

Il pattern gradient, conosciuto anche come campo computazionale, è una versione più evoluta del pattern spreading in cui l'informazione che viene propagata contiene anche l'indicazione della distanza della sorgente dell'informazione. Nel pattern gradient viene usato anche il pattern aggregation per unire diversi gradienti provenienti dalla stessa sorgente o per unire gradienti diversi. Questo pattern può essere applicato nel caso in cui debba essere mantenuta aggiornata l'informazione della distanza minima dalla sorgente.

L'informazione inizialmente presente in un nodo viene diffusa ed aggregata quando incontra nuova informazione al fine permettere agli agenti di avere informazioni di tipo globale sul sistema, come la distanza minima dalla sorgente di informazione, che altrimenti non sarebbe disponibile localmente. La distanza minima viene calcolata in fase di aggregazione grazie ad un operatore di filtro che mantiene solo la distanza più piccola fra tutte quelle disponibili.

È possibile usare anche il pattern evaporation per permettere al gradiente di adattarsi ai cambiamenti di topologia della rete: l'informazione viene diffusa periodicamente e le informazioni più vecchie vengono rese meno pertinenti per permettere all'agente di seguire sempre la strada più corta, in questo caso il gradiente viene definito attivo. Nella creazione di un gradiente attivo è fondamentale decidere in modo opportuno la frequenza di aggiornamento: alte frequenze rendono il gradiente molto reattivo ai cambiamenti di topologia ma rischiano di intasare ed appesantire il sistema mentre basse frequenze rischiano di fornire agli agenti informazioni sbagliate.

Le entità coinvolte nel pattern gradient sono agenti, host ed agenti infrastrutturali. Come nel pattern spreading, quando un gradiente viene creato, esso viene diffuso ai suoi vicini con regole di transizione che sono un caso particolare delle regole descritte nel paragrafo 1.5.1.1:

$$spreading ::\langle L,[D,C]\rangle \xrightarrow{r_{spr}} \langle L_k,[D+\Delta D,C]\rangle$$
$$where\ L_k = random\left(\{L_1,\ldots,L_n\}\right)$$
$$aggregation ::\langle L,[D_1,C]\rangle,\ldots,\langle L,[D_n,C]\rangle \xrightarrow{r_{aggr}} \langle L,[D',C]\rangle$$
$$where\ D' = min/max\left(\{D_1,\ldots,D_n\}\right)$$

dove D è un attributo che rappresenta la distanza dell'informazione dalla sorgente che l'ha generata. La regola di spreading in questo caso modifica l'informazione della distanza andandola ad aumentare di una distanza pari a quella fra il nodo ed il nodo vicino ogni volta che la tupla viene diffusa verso un certo vicino. La funzione random sceglie un nodo a caso fra tutti i vicini del nodo corrente, dal punto di vista spaziale il gradiente assumerà quindi la forma di un cono con il vertice rivolto verso il basso e posizionato nella sorgente. La regola di aggregazione fa in modo che se in un nodo sono presenti due tuple contenenti diverse distanze venga tenuta solo quella con distanza minima (o massima a seconda dei casi). Il comportamento degli agenti coinvolti nel pattern gradient è descritto nell'immagine 1.8:

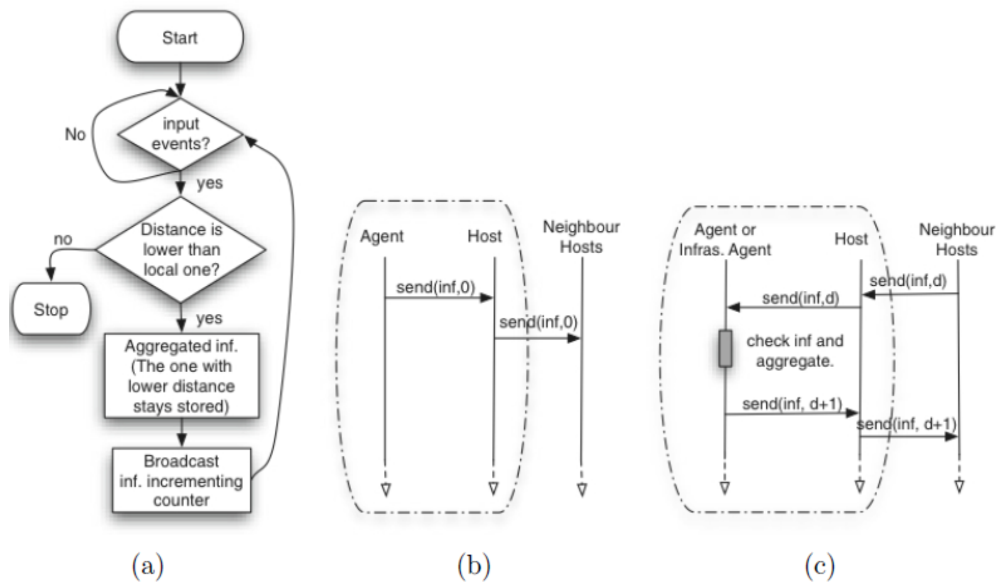


Figura 1.8: (a) comportamento di un agente, (b) inizializzazione di un agente, (c) interazione fra agente ed agente infrastrutturale. Immagine tratta da [1]

1.5.2.2 Digital pheromone pattern

Il digital pheromone pattern è un pattern di comunicazione indiretta in cui ogni agente deposita nell'host un feromone digitale che diffonde in gradiente nell'ambiente e persiste nel tempo per un certo periodo per poi evaporare. In questo modo anche gli agenti fuori dal range di comunicazione possono usufruire dell'informazione fornita dal feromone digitale.

Questo pattern è usato per coordinare agenti in ambienti su larga scala ed altamente dinamici, usando la comunicazione indiretta. Prende spunto dal feromone naturale usato dalle formiche ma nell'ambito dei sistemi computazionali è realizzato tramite combinazione del pattern gradient e del pattern evaporation, in pratica è un gradiente attivo in cui i feromoni sono responsabili della comunicazione indiretta mentre nel gradiente attivo classico la diffusione avviene da agente ad agente. I parametri chiave per il feromone digitale sono i parametri dei pattern atomici che lo compongono quindi rate e frequenza di evaporazione per l'evaporation pattern, distanza e frequenza di diffusione per lo spreading pattern che compone il gradient pattern.

Le entità coinvolte in questo pattern sono: agenti, gli unici che possono depositare feromone negli host, agenti infrastrutturali che applicano i meccanismi di diffusione, aggregazione ed evaporazione. Le regole di transizione che definiscono il pattern gradient sono simili a quelle del pattern gradient, con l'aggiunta di una regola per l'evaporazione:

$$
\begin{aligned}
spreading ::&\langle L, [PhV, C] \rangle \xrightarrow{r_{spr}} \langle L_k, [PhV - \Delta PhV, C] \rangle \\
&where\ L_k = random\left(\{L_1, \ldots, L_n\}\right) \\
aggregation ::&\langle L, [PhV_1, C] \rangle, \ldots, \langle L, [PhV_n, C] \rangle \xrightarrow{r_{aggr}} \langle L, [PhV_i, C] \rangle \\
&where\ PhV_i = max\left(\{PhV_1, \ldots, PhV_n\}\right) \\
evaporation ::&\langle L, [PhV, C] \rangle \xrightarrow{r_{ev}} \langle L, [PhV', C] \rangle \\
&where\ PhV' = PhV * Ev_{factor}
\end{aligned}
$$

In questo caso invece della distanza l'informazione aggiuntiva è quella della concentrazione di feromone in ogni nodo, allontanandosi dalla sorgente

diminuisce la concentrazione ed in ogni nodo verrà mantenuta solo la tupla contenente la concentrazione più alta, la terza regola indica come la concentrazione di feromone evapora grazie ad un parametro moltiplicativo compreso nell'intervallo [0..1].

Questo pattern, altamente scalabile e robusto, viene usato in applicazioni quali *Autonomus Coordination of Swarming* e Ant foraging pattern.

1.5.3 Pattern di alto livello

In questa ultima parte del Capitolo 1 vengono descritti i pattern di alto livello che sono in relazione al pattern Gradient, questi pattern complessi vengono realizzati tramite l'uso di pattern più semplici e meccanismi.

1.5.3.1 Ant foraging pattern

L'ant foraging pattern (ACO) è un pattern complesso di ispirazione biologica che ricalca il comportamento usato dalle formiche per trovare fonti di cibo, è fondamentalmente un problema di ricerca collettiva decentralizzata.

Questo pattern è una versione evoluta del feromone digitale descritto nel sotto-capitolo 1.5.2.2, ogni agente segue il gradiente con una certa probabilità che dipende dalla concentrazione del feromone presente oppure si muove in una direzione casuale, proprio come fanno le formiche in natura. In modo più formale ACO è definito dalle seguenti regole di transizione:

$$\begin{aligned} up_move ::& \langle L, [PhV_1, C] \rangle, \ldots, \langle L, [PhV_n, C] \rangle \xrightarrow{r_{umove}} \langle L_i, [PhV_i, C] \rangle \\ & where\ PhV_i = max\left(\{PhV_1, \ldots, PhV_n\}\right) \\ random_move ::& \langle L, C \rangle \xrightarrow{r_{rmove}} \langle L_i, C \rangle \\ & where\ L_i = random\left(\{L_1, \ldots, L_n\}\right) \end{aligned}$$

La prima regola modella un agente che decide di seguire la traccia del feromone mentre la seconda modella un agente che decide di muoversi in modo casuale.

Come già descritto per il feromone digitale anche in queste due regole è fondamentale il tuning dei rate per ottenere il comportamento desiderato.

Il comportamento di ogni agente coinvolto nell'Ant foraging pattern è descritto nell'immagine sottostante:

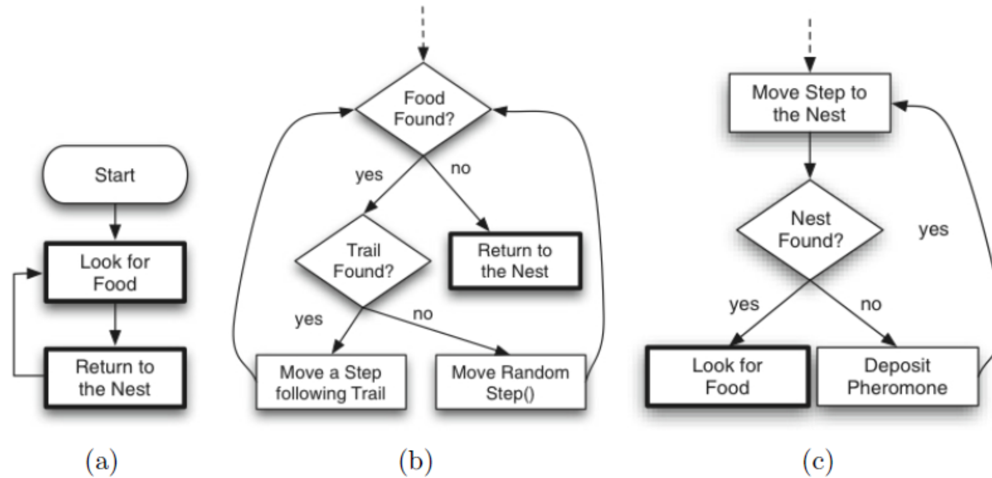


Figura 1.9: (a) Comportamento generale, (b) ricerca del cibo, (c) ritorno al nido. Immagine tratta da [1]

1.5.3.2 Chemiotaxis pattern

Il pattern chemiotaxis è un pattern di alto livello che permette di coordinare movimenti nei sistemi a larga scala grazie ad una estensione del pattern gradient. Ogni agente riesce ad identificare la direzione del gradiente e quindi muoversi di conseguenza, decidendo a seconda della necessità di compiere tre tipi di movimento: attrattivo, cioè muoversi verso la sorgente del gradiente, repulsivo, cioè allontanarsi dalla sorgente del gradiente, oppure muoversi in modo equipotenziale, cioè muoversi rispettando una certa distanza dalla sorgente. Questa regola di comportamento è così descritta:

$$move :: \langle L, [D_1, C] \rangle, \ldots, \langle L_n, [D_n, C] \rangle \xrightarrow{r_{move}} \langle L_i, [D_i, C] \rangle$$
$$where\ D_i = min/max/equal\left(\{D_1, \ldots, D_n\}\right)$$

Questo pattern trova applicazione nella coordinazione di robot mobili e nell'instradamento di messaggi necessari alla computazione pervasiva.

1.5.3.3 Morphogenesis pattern

Il pattern morphogenesis è un pattern complesso, estensione del pattern gradient, ispirato al processo biologico che permette alle cellule di un organismo in fase di formazione di posizionarsi al posto giusto. Questo pattern ha permesso di risolvere problemi in cui gli agenti devono prendere decisioni sulla in base al proprio ruolo o alla propria posizione spaziale.

Le entità coinvolte in questo pattern sono agenti, host ed agenti infrastrutturali. Si inizia con la diffusione di un certo numero di gradienti di morfogenesi, implementati secondo il pattern gradient, da parte di agenti specifici. Gli altri agenti valutano la loro posizione relativa rispetto alle sorgenti del gradiente in modo da poter adottare diversi ruoli o effettuare specifiche azioni a seconda di dove sono situati. La regola di transizione che modella questo comportamento è la seguente:

$$state_evolution :: \langle L, [D, State, C] \rangle \xrightarrow{r_{move}} \langle L, [D, State', C] \rangle$$
$$where\ State' = \pi\,(D)$$

dove $\pi(D)$ è una funzione che cambia le variabili di stato dell'agente a seconda delle informazioni che percepisce localmente. Il comportamento di un agente durante il morphogenesis pattern è mostrato in figura 1.10.

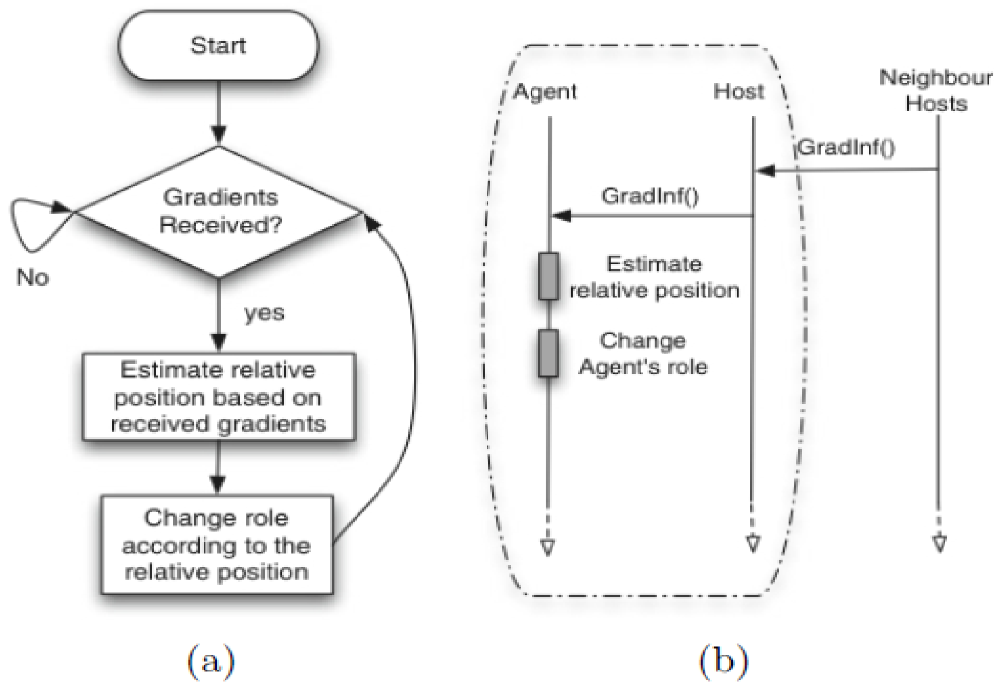


Figura 1.10: (a) comportamento di un agente, (b) interazione fra host ed agente. Immagine tratta da [1]

1.5.3.4 Quorum sensing pattern

Il quorum sensing pattern è un pattern complesso, usato per coordinare gli agenti al fine di prendere decisioni collettive in modo decentralizzato. L'obiettivo è quello di fornire una stima degli agenti nel sistema utilizzando solo interazioni locali dato che una certa decisione può essere presa solo se sono presenti almeno un certo numero di agenti. Viene realizzato estendendo il pattern gradient, il raggiungimento di una certa concentrazione di gradiente fa scattare la decisione o il comportamento collaborativo.

La regola di transizione che modella questo pattern è la stessa del chemiotaxis pattern, con la differenza che la funzione $\pi(D)$ è data da:

$$\pi(D) = \begin{cases} State & \text{if } D \leq threshold \\ State' & \text{if } D > \ threshold \end{cases}$$

Il comportamento degli agenti impiegati in questo pattern è mostrato nella figura sottostante:

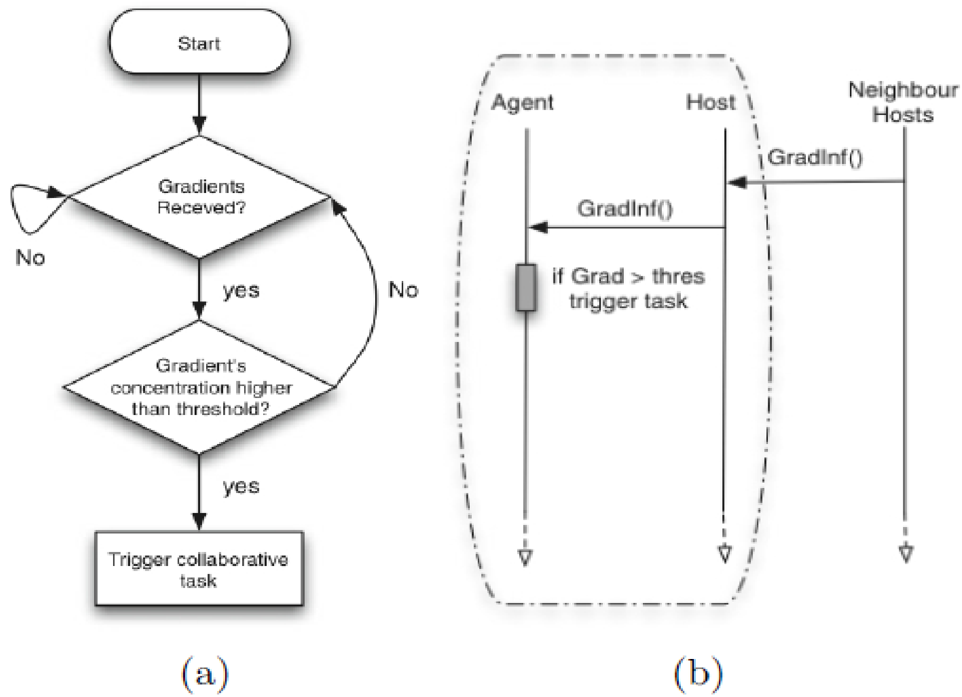


Figura 1.11: (a) comportamento dell'agente, (b) interazioni. Immagine tratta da [1]

Capitolo 2

Sistemi pervasivi

In questo capitolo si introduce il concetto di sistema pervasivo[2], cioè il tipo di sistema in cui i design pattern trovano applicazione e si descrive il framework di coordinazione SAPERE.

2.1 Definizione e requisiti

Un sistema pervasivo è un sistema computazione presente nell'ambiente quotidiano in cui viviamo, composto da individui di vario genere: persone fisiche, device mobili, sistemi software, sensori, sorgenti di conoscenza, informazioni, eventi ed altro ancora. Tutti questi individui agiscono autonomamente per raggiungere il loro scopo individuale ma sono anche regolamentati da leggi presenti nell'infrastruttura in cui sono situati. Queste infrastrutture possono essere utili solo se sono generali, aperte ed auto-organizzanti, cioè devono soddisfare i requisiti di situatedness, adattività e tolleranza[2].

La situatedness è la capacità dell'infrastruttura di interagire con agenti che sono collocati nello spazio (*spatial situatedness*) ed in preciso contesto sociale (*social situatedness*), adattando quindi il comportamento a seconda di ogni contesto. Questo requisito viene soddisfatto reificando dati ed eventi dal punto preciso dello spazio a cui essi appartengono e promuovendo interazioni di tipo locale.

L'adattività è la capacità del sistema pervasivo di continuare a funzionare a fronte di un cambiamento imprevisto dell'ambiente senza intervento umano o di un coordinatore centrale. Questo requisito viene soddisfatto tramite l'implementazione di regole di coordinazione a livello locale che permettono l'emergenza di proprietà globali del sistema.

La tolleranza è la proprietà del sistema di sostenere modelli aperti, i servizi offerti dal sistema non devono essere fissati a priori ma il loro numero può crescere grazie all'introduzione di nuovi servizi che vengono iniettati nel sistema. Questo requisito viene soddisfatto standardizzando il modo in cui gli individui manifestano la loro esistenza nel sistema ed il modo in cui questa manifestazione viene usata per attivare le interazioni.

Per soddisfare i requisiti sopra descritti è necessario che il sistema pervasivo sia auto-organizzante, cioè emergano modelli di interazione complessi dalle interazioni locali, regolate da semplici leggi, dei componenti del sistema. In questo modo la struttura del sistema diventa molto più flessibile e robusta ai cambiamenti del sistema e l'adattività ne diventa una caratteristica intrinseca.

I sistemi pervasivi si ispirano molto al modello biologico, intrinsecamente auto-organizzativo, in cui il comportamento globale emerge, tramite meccanismi come la stigmergia, dalle interazioni strettamente locali degli individui che sono guidati da leggi ambientali, nell'approccio usato in questa tesi queste leggi prendono il nome di *eco-law*.

Esempi di scenari applicativi per i sistemi pervasivi possono essere la fornitura di servizi su schermi adattativi, il controllo intelligente del traffico o l'*augmented reality*. Per realizzare questi scenari serve un approccio sistematico all'auto-organizzazione, che ne regoli i meccanismi, per venire incontro a questa esigenza sono stati definiti i design pattern, tra cui quelli descritti nel capitolo 1.5 e quelli sviluppati nel corso di questa tesi.

Tutti i pattern di questa tesi si basano su un noto meccanismo di auto-organizzazione che prende il nome di gradiente (o campo) computazionale. Il gradiente è una struttura dati distribuita basata sull'astrazione spaziale della distanza che viene progettato per fornire cammini di percorso minimo in sistemi pervasivi anche molto complessi ed articolati e per adattarsi dinamicamente a situazioni impreviste che comportano il cambio di topologia del sistema.

2.2 SAPERE: un framework per sistemi pervasivi

Il progetto SAPERE (*Self-adaptive Pervasive Service Ecosystem*) è nato per fornire un paradigma alternativo per progettare sistemi context-aware che fossero dotati di capacità di adattamento e self-awareness. Questo è stato possibile introducendo un substrato spaziale, rappresentato come un ecosistema di individui autonomi di vario tipo che si coordinano sulla base di leggi, chiamate eco-law. L'applicazione delle eco-law comporta l'evoluzione della popolazione degli individui presenti nel sistema.

2.2.1 Architettura e linguaggio

Il progetto SAPERE trae ispirazione dal modello chimico[2], l'idea di base è quella di modellare ogni entità in modo uniforme all'interno del framework. Queste entità, chiamate individui, possono essere persone fisiche, device mobili o servizi software. Per realizzare questa astrazione viene introdotto un substrato spaziale, mappato sulla struttura di rete pervasiva, formato da nodi che rappresentano una specifica area locale. Questo significa che ogni nodo sarà il luogo usato da individui presenti nella stessa area per scambiarsi informazioni.

I servizi a disposizione dell'infrastruttura si manifestano al sistema inserendo e gestendo negli spazi dei nodi delle apposite rappresentazioni

chiamate Live Semantic Annotation (d'ora in avanti LSA) che hanno la capacità di rappresentare la dinamicità dello scenario rappresentato e la mutevolezza delle entità del sistema.

L'aspetto di coordinazione del sistema sarà delegato a delle particolari regole, anche loro di ispirazione chimica, dette eco-law in grado di usare le LSA come reagenti per farle evolvere al fine di riflettere i cambiamenti del sistema e far emergere il pattern di coordinazione desiderato.

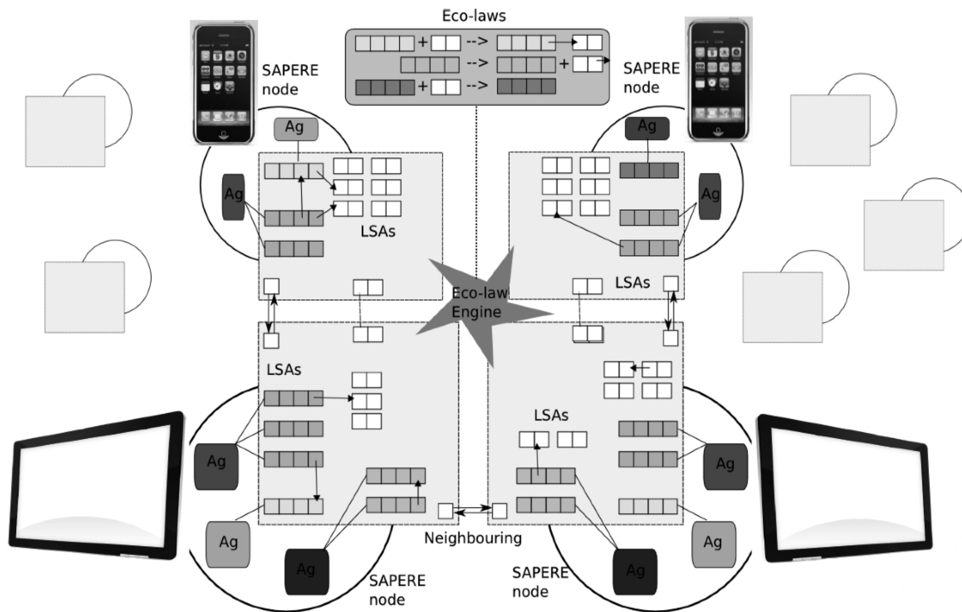


Figura 2.1: Architettura di un sistema pervasivo. Immagine tratta da [2]

Lo spazio condiviso è strutturato come una rete di LSA-Space, questi spazi sono componenti passivi simili agli spazi di tuple che forniscono un'interfaccia per la lettura, modifica, inserimento o rimozione delle LSA presenti al loro interno. Ogni LSA-Space è contenuto in un nodo ed ha il compito di memorizzare le LSA relative a quel nodo, inoltre ogni spazio è soggetto a particolari eco-law che regolano le reazioni fra le LSA, al fine di realizzare il comportamento desiderato. Le reazioni usano le LSA come

reagenti e vengono attivate da un meccanismo di pattern matching, l'attivazione di una reazione può portare ad una modifica di LSA, inserimento di nuove LSA o diffusione delle LSA verso i nodi vicini.

Una eco-law, in modo simile ad una reazione chimica, lavora sui pattern delle LSA. In questo contesto ogni LSA è rappresenta come una tupla, cioè una sequenza ordinata di valori tipati, quindi un LSA pattern non è altro che una LSA in cui si hanno delle variabili al posto di certi valori. Un a certa LSA L fa match con un pattern P se esiste una sostituzione di variabili tramite la quale è possibile applicare P dato L. Una eco-law è solitamente del tipo:

$$P_1, \ldots, P_n \xrightarrow{r} P'_1, \ldots, P'_n$$

- nel lato dei reagenti vengono specificati i pattern che devono fare match con le LSA estratte dal LSA-Space;
- nel lato dei prodotti vengono indicate le tuple che saranno inserite nel LSA-Space come risultato della reazione
- il rate *r* indica con quale frequenza viene attivata l'eco-law.

Il linguaggio viene poi esteso con alcuni concetti chiave fondamentali per il raggiungimento degli obiettivi di questo framework. Il concetto più importante è quello di pattern remoto per permettere l'interazione fra LSA-Space vicini, il pattern remoto può fare match con LSA presenti in LSA-Space remoti e viene indicato nel linguaggio con +P.

Un altro concetto importante è quello dell'inserimento di espressioni matematiche o variabili di sistema come argomento inseribile in un pattern, questo permette di applicare le eco-law su una gamma più vasta di LSA. Le variabili di sistema vengono usate per inferire informazioni sul contesto fornite dall'infrastruttura, nel linguaggio iniziano tutte con #, fra le più usate troviamo #T che è legata al tempo in cui l'eco-law viene attivata e #D che indica la distanza topologica fra lo spazio locale e lo spazio remoto.

Capitolo 3

Alchemist

Negli ultimi anni la simulazione di un sistema è stata introdotta già dalle prime fasi dello sviluppo del software, in modo da valutare subito l'andamento qualitativo delle dinamiche del sistema stesso. Questo ha reso la simulazione un approccio chiave soprattutto nei sistemi che devono presentare un comportamento emergente.

Lo scopo di questa tesi è proprio quello di simulare algoritmi di auto-organizzazione e per farlo è stato usato un framework chiamato Alchemist [3]. In questo capitolo verrà motivata questa scelta, verranno descritte le caratteristiche di Alchemist e verrà descritto il Domain Specific Language (DSL) usato in Alchemist per la modellazione degli algoritmi.

3.1 Motivazioni

La complessità è ovunque, sia in natura che nei moderni sistemi computazionali che supportano i sistemi pervasivi. Entrambi mostrano caratteristiche comuni come la situatedness, l'adattività e l'auto-organizzazione che permettono di reagire ai cambiamenti dell'ambiente in modo reattivo e senza la presenza di un coordinatore centrale ma grazie ad un comportamento emergente. È facile quindi intuire come la progettazione degli algoritmi presenti in questi sistemi si basi su due fattori chiave:

- Ispirazione ai sistemi biologici, sia in termini di pattern che di metafore. Questa tesi è focalizzata sul modello della computazione chimica.
- Simulazione: deve essere possibile effettuare analisi what-if prima dello sviluppo del sistema in modo da garantire il funzionamento corretto ed il giusto tuning dei parametri.

Alchemist, un framework di simulazione sviluppato dall' Ing. Danilo Pianini, all'interno del gruppo di ricerca della Seconda Facoltà di Ingegneria nell'ambito del progetto SAPERE, nasce proprio per colmare il gap esistente fra i due approcci sopra descritti ed è pensato in particolar modo per i sistemi computazionali di ispirazione chimica.

Alchemist estende il modello computazionale bio-chimico, di cui mantiene le alte performance, in modo da applicarlo a sistemi computazionali complessi. In particolare Alchemist è basato su una versione ottimizzata dello SSA di Gillespie, chiamato Next Reaction Method[7], opportunamente esteso per permettere la gestione di un ambiente dinamico (aggiunta/rimozione di reazione, entità, connessioni topologiche) in modo più generale possibile. Le alte performance, di cui si avrà una panoramica estesa nel prossimo capitolo, unite alla grande flessibilità e alla semplicità del DSL, lo rendono il framework più adatto per lo sviluppo di questa tesi.

3.2 Modello computazionale

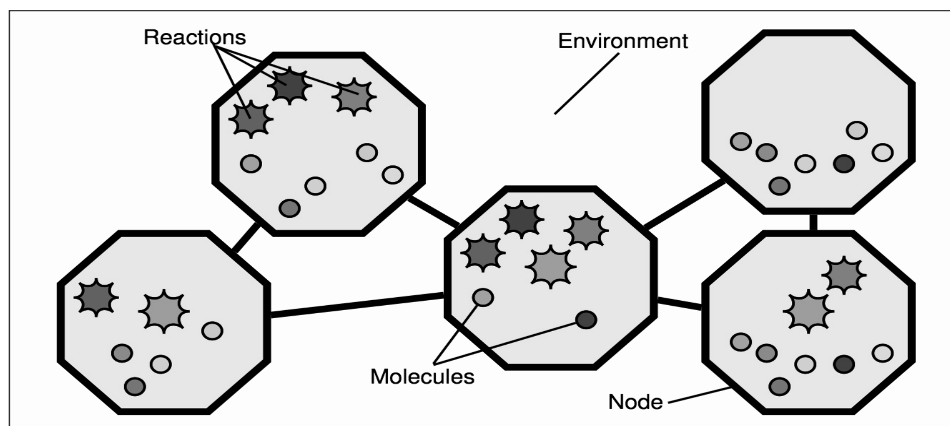


Figura 3.1: Modello computazionale di Alchemist. Immagine tratta da [5]

Alchemist ha una visione del mondo molto semplice, un ambiente è uno spazio multidimensionale, continuo o discreto, in grado di contenere nodi e responsabile di collegarli in base ad una regola. I nodi sono entità che contengono le molecole, ognuna dotata di una certa concentrazione, e possono essere programmate attraverso delle reazioni che cambiano nel tempo.

Il modello di reazione è rappresentato in figura 3.2: un insieme di condizioni sull'ambiente determinano se la reazione è eseguibile, un tasso che descrive quanto velocemente la reazione cambia in risposta ad una perturbazione ambientale, una distribuzione di probabilità per l'evento e un insieme di azioni che saranno effetto della reazione. Questo modello permette di definire quale tipo di distribuzione temporale viene usata per innescare le reazioni, è possibile quindi modellare e simulare sistemi basati sulle catene di Markov a tempo continuo (CTMC), per aggiungere trigger o per usare il tempo discreto basato sui “tick”.

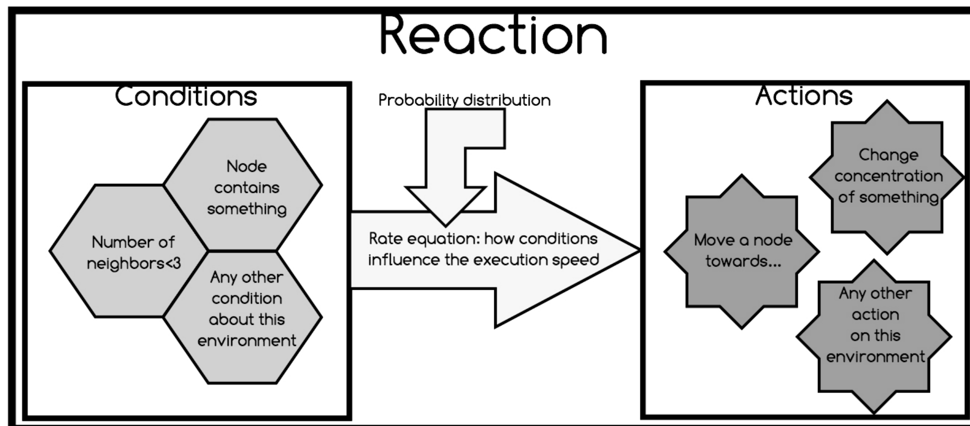


Figura 3.2: modello di una reazione. Immagine tratta da [5]

Prima di proseguire è necessario descrivere come i concetti di ambiente e nodo sono mappati in SAPERE.

L'ambiente, di solito continuo, collega i nodi fra loro e definisce il vicinato, questa definizione è di natura fisica cioè i vicini di un nodo sono tutti i nodi presenti entro un certo raggio dalla sua posizione, i vicini sono gli unici nodi in comunicazione diretta con il nodo. L'ambiente è anche responsabile della diffusione delle annotazioni tramite regole appropriate.

In accordo con la metafora bio-chimica i nodi sono contenitori di molecole, ogni nodo è regolato da un insieme di reazioni, applicabili su un sottoinsieme di molecole presenti, che possono modificare lo stato interno del nodo, ovvero la configurazione delle molecole. Le molecole sono un modello di dati, ognuno descritto da un valore di concentrazione e da una serie di proprietà, tali modelli prendono il nome di annotazioni all'interno di SAPERE.

Le reazioni qui descritte sono più complesse e generali delle reazioni chimiche composte solo da reagenti e prodotti, si considera un insieme di condizioni sullo stato del sistema che attivano un trigger responsabile dell'esecuzione di un insieme di azioni. Una condizione è generalmente una funzione che associa un valore booleano ad uno stato del sistema, ci possono

essere molti tipi diversi di condizioni, dalle più semplici alle più complesse. Se tutte le condizioni sono soddisfatte allora possono essere eseguite le azioni conseguenti sul sistema.

La velocità con cui le reazioni scattano è descritta da una funzione di *propensity*, dipendente dalla concentrazione dei reagenti, che permette una maggiore flessibilità in quanto è funzione del tasso della reazione, delle condizioni e dello stato dell'ambiente.

3.3 Architettura

Alchemist è stato progettato per essere completamente modulare ed estendibile, il modello può essere modificato senza andare a toccare in alcun modo il motore di simulazione.

Come si può notare in figura 3.3 Alchemist è composto da elementi comuni ad ogni scenario, indicati in figura con linee continue, e da componenti delimitati da linee tratteggiate che indicano estensioni specifiche che dovranno essere sviluppate secondo una ben precisa incarnazione. Questo è possibile grazie alla mancanza di restrizioni presenti sul genere di struttura dati che rappresenta la concentrazione delle molecole. Ad esempio se la concentrazione è definita come un numero intero, rappresentante il numero di molecole in un nodo, allora Alchemist si comporterà come un simulatore chimico stocastico mentre se la concentrazione viene definita come un tuple set, quindi le molecole saranno template di tuple, Alchemist diventerebbe un simulatore di per reti di spazi di tuple. Si ottiene una diversa incarnazione per ogni diversa definizione di concentrazione, in corrispondenza di ogni incarnazione può essere definito un insieme specifico di azioni, condizioni e reazioni.

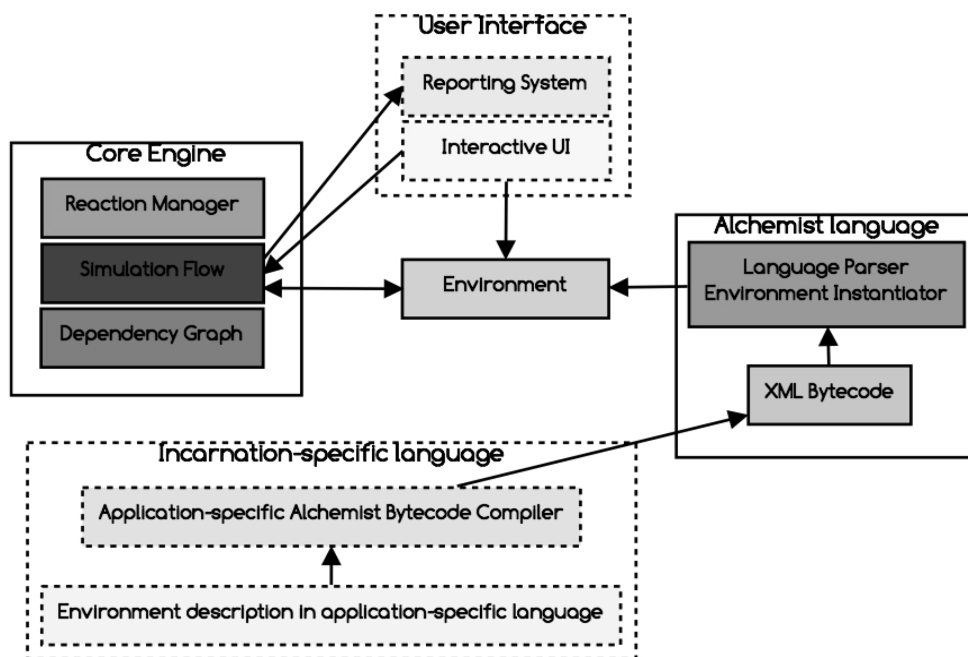


Figura 3.3: Architettura di Alchemist. Immagine tratta da [5]

3.4 Motore di simulazione

Come già detto nella sezione 3.1 Alchemist si basa sull'algoritmo SSA Next Reaction tramite l'uso di due importanti strutture dati chiamate Dependency Graph (DG) e Indexed Priority Queue (IPQ), sarà compito del Reaction Manager scegliere la prossima reazione da eseguire.

Il DG ha come compito quello di selezionare le reazioni che verranno aggiornate ad ogni step di simulazione, per fare ciò si appoggia ad una struttura dati chiamata Reaction Handler che lega all'entità di reazione utili dettagli implementativi. Come si può vedere in figura 3.4 il DG definisce le relazioni fra le condizioni e le azioni delle reazioni presenti nel sistema in modo da sapere subito quali reazioni aggiornare a fronte della modifica di una singola reazione. Analogamente, per ogni nuova reazione, verranno calcolate quelle che vengono influenzate e da quali viene influenzata, queste informazioni saranno memorizzate nel DG per evitare aggiornamenti inutili di reazioni indipendenti.

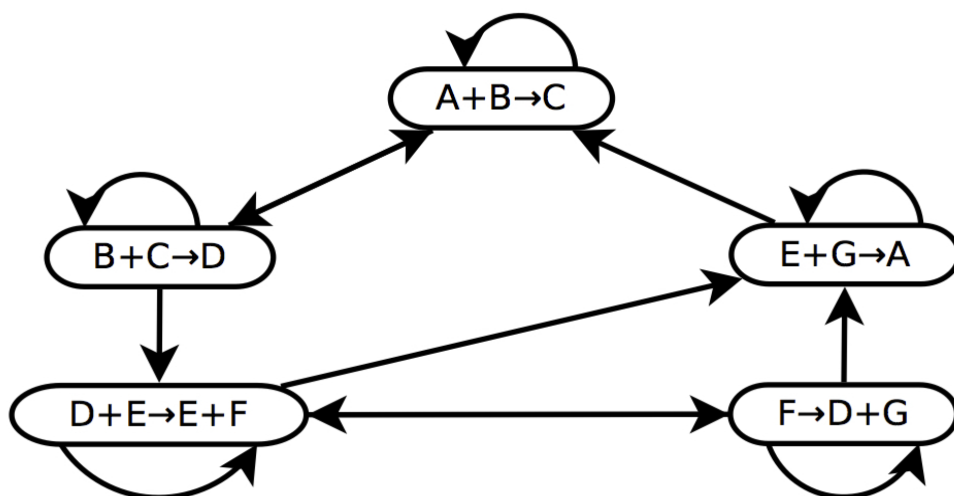


Figura 3.4: Esempio di Dependency Graph

L'IPQ è un albero binario di reazioni la cui caratteristica principale è quella di memorizzare in ogni nodo una reazione che ha un tempo presunto di esecuzione minore di ogni quello di tutti i suoi figli, questo significa che la prossima reazione da eseguire si trova sempre nel nodo radice dell'albero e il tempo di accesso risulta quindi costante. Un'altra proprietà importante dell'IPQ è quella di essere progettato per rimanere sempre bilanciato anche a fronte di inserimento o rimozione di nodi, questo è possibile tenendo contro del numero di figli presenti per ogni ramo dell'albero.

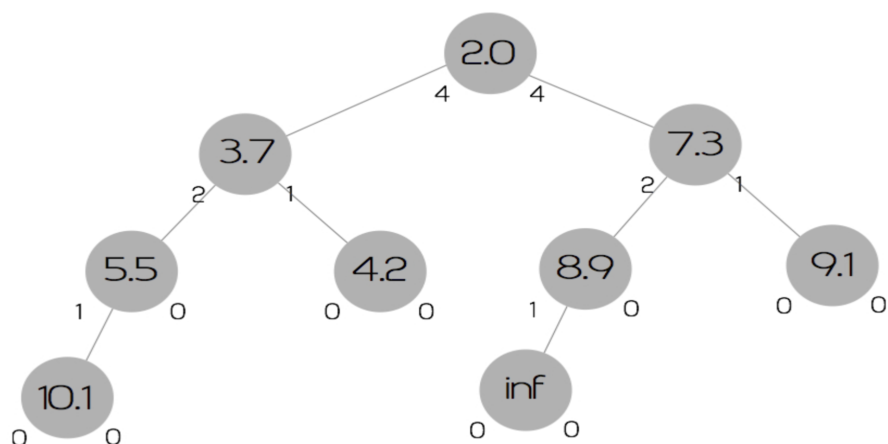


Figura 17: esempio di IPQ. Immagine tratta da [5]

3.5 Domain Specific Language

Il DSL usato in Alchemist è stato sviluppato nell'ambito del progetto SAPERE tramite la tecnologia Xtext quindi è disponibile un editor guidato dalla sintassi integrato in Eclipse, basterà creare in questo ambiente un file con estensione .alsap in cui scrivere la specifica e sarà compito del framework Xtext generare automaticamente il codice necessario per produrre il file XML che verrà interpretato.

Ogni file .alsap definite un ambiente, cioè una simulazione indipendente, all'interno di questo ambiente è possibile definire diversi elementi: ambiente, LSA, nodi e reazioni.

Ogni specifica deve iniziare con la keyword `environment` che può essere seguita da diversi parametri:

- `name`: parametro opzionale che definisce il nome dell'ambiente.
- `of type` *AnEnvironment* `params` "*some params*": per specificare un ambiente descritto nella classe Java *AnEnvironment* i cui parametri sono "*some params*". Se non specificato l'ambiente sarà uno spazio

continuo in cui i nodi sono collegati fra loro se la distanza che li separa è minore di una certa quantità passata come parametro.

- `with random seed` *N*: permette di specificare un particolare seed per la simulazione, utile per garantire la riproducibilità della simulazione. Se non specificato è un numero casuale.

Dopo la definizione dell'ambiente è necessario definire le LSA coinvolte, ogni LSA è indicata dalla parola chiave `lsa` seguita dal nome dell'LSA e dalla molecola che contiene. Ad esempio `lsa gradient <grad, Distance, 0>` dichiara una LSA di nome `gradient` che contiene la molecola `<grad, Distance, 0>`.

Adesso è possibile definire le eco-law, la sintassi corretta per definirle è del tipo `eco-law` *lawName* [*conditions*] -*r*- > [*action*]:

- `eco-law`: keyword per la dichiarazione di una eco-law.
- *lawName*: nome associato all'eco-law.
- [*conditions*]: le condizioni da soddisfare per far scattare la reazione, le molecole specificate fra le condizioni verranno consumate.
- [*actions*]: le azioni che verranno eseguite come conseguenza della reazione, le molecole presenti nelle azioni saranno inserite.
- *r*: è il rate della reazione, se non è specificato la reazione verrà eseguita ASAP cioè non appena le condizioni sono verificate.

È possibile inserire nelle reazioni dei caratteri speciali o delle keyword particolari con una semantica ben definita:

- +<*Mol*>: se usato nel lato sinistro della reazione significa che la molecola *Mol* dovrà essere presente in almeno un vicino del nodo in cui avviene la reazione e verrà consumata, se usata nel lato destro della reazione inserisce la molecola *Mol* in uno dei vicini del nodo in maniera random.

- *<*Mol*>: può essere usato solo nel lato destro della reazione, indica che la molecola *Mol* verrà inserita in tutti i nodi del vicinato.
- `#T, #D, #O`: sono caratteri speciali usabili solo nel lato destro delle reazioni che coinvolgono il vicinato, `#T` indica il tempo di simulazione in cui la reazione viene eseguita, `#D` indica la distanza topologica fra un nodo e i suoi vicini, `#O` è un vettore che punta al nodo corrente del vicinato.
- `agent` *AgentName* `params` "*some params*": usabile solo nel lato destro della reazione permette di caricare una classe Java di nome *AgentName*, che necessiterà dei parametri "*some params*", per definire un agente mobile, molto utile per eseguire azioni che non possono essere espresse tramite il DSL.

Infine è possibile definire i singoli nodi o un insieme di nodi con caratteristiche comuni tramite la keyword "`place` *N* `node`", dove *N* è il numero di nodi da inserire, seguita da:

- `at point(x,y)`: posiziona un singolo nodo nel punto (x,y) dello spazio
- `in rect(x,y,x',y')`: posiziona gli N>1 nodi in un intervallo rettangolare il cui punto più in basso a sinistra è (x,y) mentre il punto più in alto a destra è (x',y'). È possibile imporre, tramite la keyword "`interval` *L*", di posizionare i nodi intervallati da una distanza *L*, se non specificato i nodi vengono posizionati casualmente.
- `containing` *anLSA*: quale o quali LSA dovranno essere contenute nel nodo o nell'insieme di nodi.
- `with reaction`: quale o quali reazioni saranno presenti nel nodo o nell'insieme di nodi.

Questa panoramica del linguaggio, per quanto comprenda praticamente tutte le keyword usate nello sviluppo degli algoritmi che verranno presentati nei prossimi capitoli, non è completa, si faccia riferimento al manuale[5] per la descrizione completa.

Capitolo 4

Performance di Alchemist

Nel capitolo 3 si è motivata la scelta di Alchemist come framework di simulazione con le alte performance derivanti dal modello bio-chimico su cui è basato. In particolare Alchemist è costruito per gestire contemporaneamente un alto numero di gradienti senza degradare troppo le performance, in questo capitolo verranno presentare le analisi di performance effettuate sul simulatore che lo dimostrino. Le analisi di performance sono state effettuate sul gradiente YOUNGEST, uno dei gradienti più semplici e sul gradiente SAPERE, un gradiente ospitato in un agente. Le analisi effettuate serviranno anche a determinare le performance di ogni gradiente al fine di motivare la scelta del gradiente che verrà usato nello sviluppo del pattern di questa tesi.

4.1 Gradiente YOUNGEST

4.1.1 Descrizione del gradiente

Il gradiente YOUNGEST è stato uno dei primi gradienti sviluppati nell'ambito del progetto SAPERE, formalmente può essere descritto dalle seguenti LSA:

$$\langle \text{source}, Type, Time \rangle$$
$$\langle \text{grad}, Type, Distance, Time \rangle$$

La prima LSA rappresenta il nodo sorgente del gradiente mentre la seconda LSA rappresenta un nodo che porta al suo interno un gradiente, l'informazione sulla distanza dalla sorgente è contenuta nel campo *Distance*. Le eco-law responsabili della creazione e della diffusione del gradiente YOUNGEST sono le seguenti:

$$\begin{array}{rcl}
\langle \text{source}, Type, Time \rangle & \xrightarrow{r_{pump}} & \langle \text{source}, Type, Time + 1 \rangle\,, \\
 & & \langle \text{grad}, Type, 0, Time \rangle \\
\langle \text{grad}, Type, Distance, Time \rangle & \xrightarrow{r_{diff}} & \langle \text{grad}, Type, Distance, Time \rangle\,, \\
 & & * \langle \text{grad}, Type, Distance + \#D, Time \rangle \\
\langle \text{grad}, Type, Distance, Time \rangle\,, & \rightarrow & \langle \text{grad}, Type, Distance, Time \rangle \\
\langle \text{grad}, Type, Distance', Time' \rangle & & if\ Time' < Time \\
\langle \text{grad}, Type, Distance, Time \rangle\,, & \rightarrow & \langle \text{grad}, Type, Distance, Time \rangle \\
\langle \text{grad}, Type, Distance', Time \rangle & & if Distance < Distance'
\end{array}$$

La prima eco-law, detta eco-law di Pump, è responsabile della prima generazione del gradiente e della successiva rigenerazione del gradiente in modo che sia sempre reattivo ai cambiamenti topologici dell'ambiente. Il rate a cui effettuare questa reazione deve essere settato in modo che l'eco-law di Pump sia abbastanza frequente da garantire la reattività del gradiente ma una frequenza troppo elevata intasa il sistema e permette la formazione di glitch. Dalle prove effettuate si è ricavato che il valore ottimo per il rate di pump è 0.01.

La seconda eco-law, detta eco-law di Diffusione, è responsabile della diffusione del gradiente su tutti i nodi del sistema, dato il gradiente in un nodo

questa eco-law lo diffonde a tutti i vicini del nodo tramite l'operatore *, aumentando in modo opportuno la distanza. Anche il rate di diffusione ha un rate che deve essere accuratamente determinato per avere una stabilizzazione veloce del gradiente senza che però il sistema venga sovraccaricato. Dalle prove effettuate il rate ottimo è pari ad 1.

La terza eco-law, detta eco-law di Youngest, mantiene in un nodo sempre e solo la tupla più recente, in modo che il gradiente non venga calcolato sulla base di informazioni obsolete, per questo motivo viene eseguita ASAP.

La quarta eco-law, detta eco-law di Shortest, impedisce il formarsi di percorsi ciclici ed mantiene in ogni nodo solo la tupla con la distanza minima dal gradiente in modo da ottenere un gradiente consistente, per questo motivo viene eseguita ASAP.

4.1.2 Performance del gradiente

Dalla descrizione appena effettuata si possono estrarre gli elementi che influenzano la performance del gradiente:

- Numero di nodi nel sistema
- Rate di ogni eco-law
- Numero di campi per ogni LSA
- Numero di gradienti

Visto che ogni simulazione comporta anche degli agenti che risalgono il gradiente allora possiamo definire un ulteriore parametro per valutare le performance del gradiente e del simulatore, cioè:

- Numero di agenti presenti

Per effettuare queste analisi di performance si è preso uno scenario ben noto, composto da un ambiente in cui sono presenti 900 nodi disposti secondo una griglia regolare 30x30 in cui 100 agenti risalgono il gradiente partendo dal

punto di distanza massima dalla sorgente. Ad ogni analisi si sono mantenuti fissi tutti i parametri tranne quello da analizzare. Tutti i tempi indicati nelle tabelle sono in secondi.

Analisi di performance sul numero di nodi presenti nel sistema:

Numero di Nodi	Tempo Stabilizzazione	Tempo di Movimento senza Diffusione	Tempo di Movimento con Diffusione
100 (10x10)	0,3	0,9	1,1
400 (20x20)	1	2,3	3,8
900 (30x30)	2,2	4,3	9,9
1600 (40x40)	3,9	6,5	21,7
2500 (50x50)	9,2	12,2	46,5

Tabella 4.1: Analisi di performance sul numero di nodi presenti nel sistema

Al variare del numero dei nodi si è misurato quanto tempo impiegasse il gradiente a stabilizzarsi, cioè quando il valore corretto del gradiente arriva al nodo più lontano dalla sorgente, quanto tempo impiegassero gli agenti a muoversi su un gradiente che viene diffuso solo una volta sola (rate di pump pari a 0) e su un gradiente che invece viene diffuso periodicamente (rate di pump pari a 0.01). Dalla tabella sopra riportata si può concludere che la variazione del numero di nodi presenti nel sistema influisce in modo lineare sulle prestazioni del sistema, anche se in scenari molto grandi (come i 2500 nodi) subentrano effetti quadratici che le peggiorano sensibilmente.

Analisi di performance sul numero di agenti presenti nel sistema:

Numero di agenti	Tempo di Movimento senza Diffusione	Tempo di Movimento con Diffusione
1	*instant*	0,9
10	0,5	1,8
50	1,4	4,1
100	2,9	6,1
150	5,6	9
200	9,3	13,1
250	13,3	17,7
300	17,9	21,1
350	26,3	33,5
400	41	44
450	53	58
500	70	75
600	101	106
700	125	162
800	206	230

Tabella 4.2: Analisi di performance sul numero di agenti presenti nel sistema

Al variare del numero di agenti si può notare come il tempo necessario agli agenti per risalire il gradiente cresca in modo lineare in scenari con un numero ragionevole di agenti (scenari con più di 300 agenti sono raramente utilizzati) mentre in scenari con molti agenti la crescita del tempo segua un andamento più che lineare, questo dipende anche dal fatto che ogni agente è un thread autonomo e quindi si aggiungono i ritardi dovuti alla schedulazione di un numero così elevato di thread.

Analisi di performance sul numero di gradienti presenti:

Numero di gradienti	Tempo di Stabilizzazione	Tempo di Movimento con Diffusione
1	2,1	5,5
2	3,2	10,6
3	5,3	17,6
4	8,1	24,1
5	10,5	36,7
6	14	52,3
7	24,6	68
8	32,9	87
9	40,8	112
10	48,3	135

Tabella 4.3: Analisi di performance sul numero di gradienti presenti nel sistema

All'aumentare del numero di gradienti presenti nella simulazione le il tempo necessario alla stabilizzazione aumenta in quanto ci sono molte più reazioni da eseguire per diffondere i gradienti, così come aumenta il tempo di risalita in quanto gli agenti hanno molte più tuple da controllare. I tempi aumentano inizialmente in modo lineare (fino al sesto gradiente) per poi aumentare in modo più che lineare. Un'ulteriore analisi di performance per valutare l'impatto del numero di gradienti presenti è quella degli scenari simulabili: in questa analisi si va a determinare qual'è il numero massimo di agenti che possono essere presenti per avere una simulazione che sia, agli occhi dell'utilizzatore di Alchemist, fluida e veloce. Per lo scenario in questione questo comporta che il tempo necessario agli agenti per risalire il gradiente sia compreso fra 15 e 20 secondi.

Scenari simulabili con 1 gradiente:

Numero nodi	Numero agenti	Tempo di Movimento con Diffusione
10x10	300	7,5
10x10	600	15,9
20x20	300	11
20x20	350	21,3
30x30	200	13
30x30	250	17,4
40x40	100	12,4
40x40	150	17
50x50	10	14,2
50x50	50	18,3
50x50	100	24,8

Tabella 4.4: Scenari simulabili con 1 gradiente

Con un solo gradiente si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 600 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 350 agenti.
- In un ambiente con 900 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 1600 nodi possono esserci al massimo 150 agenti.
- In un ambiente con 2500 nodi possono esserci al massimo 50 agenti.

Scenari simulabili con 3 gradienti:

Numero nodi	Numero agenti	Tempo di Movimento con Diffusione
10x10	350	14,6
10x10	400	18,5
20x20	250	15,6
20x20	300	22,5
30x30	100	16,8
30x30	150	20,2
40x40	1	24,4

Tabella 4.5: Scenari simulabili con 3 gradienti

Con 3 gradienti si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 400 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 900 nodi possono esserci al massimo 150 agenti.
- Non esiste alcun scenario simulabile per ambienti di 1600 nodi.

Scenari simulabili con 5 gradienti:

Numero nodi	Numero agenti	Tempo di Movimento con Diffusione
10x10	350	14,9
20x20	200	17,2
30x30	1	26,4

Tabella 4.6: Scenari simulabili con 5 gradienti

Con 5 gradienti si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 350 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 200 agenti.
- Non esiste alcuno scenario simulabile per ambienti con 900 nodi o superiori

Scenari simulabili con 7 gradienti:

Numero nodi	Numero agenti	Tempo di Movimento con Diffusione
10x10	300	18,5
20x20	50	16,8

Tabella 4.7: Scenari simulabili con 7 gradienti

Con 7 gradienti si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 300 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 50 agenti.

Dall'analisi fatta è evidente che il numero di gradienti presenti nello stesso momento nell'ambiente sia un fattore determinante sulle performance del sistema, questo suggerisce che il gradiente YOUNGEST non è particolarmente adatto per lo sviluppo dei pattern, spesso composti da diversi gradienti combinati fra loro.

Analisi sull'aumento del numero di campi nelle LSA

Campi aggiunti	Tempo di Stabilizzazione	Tempo di Movimento con Diffusione
1	1,38	6,1
2	1,36	6,1
3	1,39	6,1
5	1,4	6,4
6	1,4	6,4
7	1,4	6,4

Tabella 4.8: Analisi sull'aumento del numero di campi nelle LSA

L'aumento del numero di campi delle LSA non comporta alcuna influenza sulle performance grazie al pattern matching di ispirazione chimica implementato nel motore di Alchemist.

4.2 Gradiente SAPERE

4.2.1 Descrizione del gradiente

Il gradiente SAPERE è un gradiente nato per sopperire alle scarse performance del gradiente YOUNGEST, è un gradiente ospitato in un agente (quindi è scritto in Java e non tramite eco-law) e sfrutta un approccio diverso rispetto a quello descritto nel sottocapitolo 3.1.1.

Nel gradiente YOUNGEST ogni nodo calcola la propria distanza dalla sorgente aggregando i valori che sono stati diffusi dal suo vicinato per poi diffondere il nuovo valore al proprio vicinato. Il gradiente viene aggiornato periodicamente tramite l'eco-law di pump e di diffusione.

Nel gradiente SAPERE ogni nodo calcola la propria distanza dalla sorgente andando a controllare i valori del vicinato e aggregandoli secondo la distanza minima al fine di computare il valore corretto del gradiente. Il gradiente viene tenuto vivo computando continuamente il valore aggiornato del gradiente in ogni nodo. La differenza di come viene mantenuto vivo il gradiente permette di eliminare i glitch derivanti dall'approccio utilizzato nel gradiente YOUNGEST.

4.2.2 Performance del gradiente

Per valutare quanto siano migliori le performance del gradiente SAPERE rispetto a quelle del gradiente YOUNGEST è necessario utilizzare lo stesso scenario di simulazione usato nel sottocapitolo 3.1.2, un ambiente composto da 900 nodi disposti su una griglia regolare 30x30 con 100 agenti che risalgono il gradiente dal punto di distanza massima fino alla sorgente.

Analisi del numero di nodi presenti nel sistema:

Numero di nodi	Tempo di Stabilizzazione del gradiente	Tempo di Movimento	Rapporto fra tempo di Stabilizzazione e di Movimento (%)
100 (10x10)	*instant*	0,2	
400 (20x20)	*instant*	0,9	
900 (30x30)	0,3	2	15
1600 (40x40)	0,42	4,1	10,24
2500 (50x50)	1	8,1	12,34
3600 (60x60)	1,5	13,1	11,45

Tabella 4.9: Analisi del numero di nodi presenti nel sistema

Come si può facilmente notare il numero di nodi presenti nel sistema influisce molto meno rispetto al precedente gradiente e la crescita del tempo di Movimento è lineare, senza presenza di effetti quadratici.

Analisi del numero di agenti presenti nel sistema:

Numero di agenti	Tempo di Stabilizzazione del gradiente	Tempo di Movimento	Rapporto fra tempo di Stabilizzazione e di Movimento (%)
10	*instant*	*instant*	
50	*instant*	0,3	
100	0,3	2	15
200	1,5	7,8	19,23
300	2,2	18,4	11,89
400	3	37,2	8,06
500	4,1	73,6	5,57

Tabella 4.10: Analisi del numero di agenti presenti nel sistema

Il numero di agenti presenti nel sistema influisce in modo lineare sul tempo di stabilizzazione del gradiente mentre iniziano a sentirsi gli effetti quadrati sul tempo di movimento quando il numero di agenti diventa sufficientemente elevato. In scenari reali, dove il numero di agenti non è troppo elevato, si può vedere come le prestazioni del gradiente SAPERE siano effettivamente migliori delle prestazioni del gradiente YOUNGEST.

Adesso è interessante analizzare gli scenari simulabili con questo gradiente per determinare se è adatto ad essere usato nello sviluppo di pattern più complessi.

Scenari simulabili con 1 gradiente:

Numero di nodi	Numero di agenti	Tempo di movimento
10x10	450	16
20x20	350	13,1
20x20	400	18,9
30x30	300	14,8
30x30	350	19,6
40x40	250	15,6
50x50	200	16,6

Tabella 4.11: Scenari simulabili con 1 gradiente

Con un solo gradiente si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 450 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 400 agenti.
- In un ambiente con 900 nodi possono esserci al massimo 350 agenti.
- In un ambiente con 1600 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 2500 nodi possono esserci al massimo 200 agenti.

Scenari simulabili con 3 gradienti:

Numero di nodi	Numero di agenti	Tempo di movimento
10x10	350	15,6
20x20	250	11,1
20x20	300	17,5
30x30	250	15,2
40x40	200	13,6
40x40	250	21,8
50x50	150	12,3
50x50	200	20,3

Tabella 4.12: Scenari simulabili con 3 gradienti

Con 3 gradienti si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 350 agenti.
- In un ambiente con 400 nodi possono esserci al massimo 300 agenti.
- In un ambiente con 900 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 1600 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 2500 nodi possono esserci al massimo 200 agenti.

Scenari simulabili con 5 gradienti:

Numero di nodi	Numero di agenti	Tempo di movimento
10x10	350	15,9
20x20	250	11,2
20x20	300	17,7
30x30	250	16,2
40x40	200	15,2
50x50	150	13,3
50x50	200	21

Tabella 4.13: Scenari simulabili con 5 gradienti

Con 5 gradienti si ha che:

- In un ambiente con 100 nodi possono esserci al massimo 350 agenti.

- In un ambiente con 400 nodi possono esserci al massimo 300 agenti.
- In un ambiente con 900 nodi possono esserci al massimo 250 agenti.
- In un ambiente con 1600 nodi possono esserci al massimo 200 agenti.
- In un ambiente con 2500 nodi possono esserci al massimo 200 agenti.

Gli scenari simulabili con 7 gradienti contemporaneamente sono gli stessi ottenibili con 5 gradienti, questo ci fa capire come il gradiente SAPERE sia molto più performante sotto questo aspetto rispetto al gradiente YOUNGEST e quindi , dando la possibilità di usare molti gradienti insieme senza degradare troppo le performance del simulatore, lo rende molto più adatto all'uso nello sviluppo di pattern.

4.3 Confronto fra il gradiente YOUNGEST ed il gradiente SAPERE

Basandoci sulle analisi sopra effettuate possiamo confrontare i le performance dei due gradienti per determinare quale sia il più adatto ad essere usato per sviluppare i pattern che saranno presentati nel prossimo capitolo.

I due gradienti possono essere confrontati secondo:

- Numero di agenti presenti nel sistema
- Numero di nodi presenti nel sistema
- Scenari simulabili.

Performance relative al numero di agenti nel sistema:

Numero di agenti	Tempo di Movimento gradiente YOUNGEST con Diffusione	Tempo di Movimento gradiente SAPERE	Variazione % del gradiente SAPERE rispetto al gradiente YOUNGEST
10	0,3	instant	/
50	4,1	0,3	-92,68
100	6,1	2	-67,21
200	13,1	7,8	-40,45
300	21,1	18,4	-12,79
400	44	37,2	-15,45
500	75	73,6	-1,86

Tabella 4.13: Confronto dei gradienti sulla base del numero di agenti

Performance relative al numero di nodi nel sistema:

Numero di nodi	Tempo di Movimento gradiente YOUNGEST con Diffusione	Tempo di Movimento gradiente SAPERE	Variazione % del gradiente SAPERE rispetto al gradiente YOUNGEST
100 (10x10)	1,1	0,2	-98,18
400 (20x20)	3,8	0,9	-76,31
900 (30x30)	9,9	2	-79,79
1600 (40x40)	21,7	4,1	-81,1
2500 (50x50)	46,5	8,1	-82,58

Tabella 4.14: Confronto dei gradienti sulla base del numero di nodi

Performance relative agli scenari simulabili con 1 gradiente:

Numero di nodi	Numero massimo di agenti per gradiente YOUNGEST	Numero massimo di agenti per gradiente SAPERE	Variazione % del numero massimo di agenti
100 (10x10)	600	450	-25
400 (20x20)	350	400	14,28
900 (30x30)	250	350	40
1600 (40x40)	150	250	66,67
2500 (50x50)	50	200	300

Tabella 4.15: Confronto dei gradienti sulla base

degli scenari simulabili con 1 gradiente

Performance relative agli scenari simulabili con 3 gradienti:

Numero di nodi	Numero massimo di agenti per gradiente YOUNGEST	Numero massimo di agenti per gradiente SAPERE	Variazione % del numero massimo di agenti
100 (10x10)	400	350	-14,28
400 (20x20)	250	300	20
900 (30x30)	150	250	66,67
1600 (40x40)	/	250	
2500 (50x50)	/	200	

Tabella 4.16: Confronto dei gradienti sulla base

degli scenari simulabili con 3 gradienti

Performance relative agli scenari simulabili con 5 gradienti:

Numero di nodi	Numero massimo di agenti per gradiente YOUNGEST	Numero massimo di agenti per gradiente SAPERE	Variazione % del numero massimo di agenti
100 (10x10)	350	350	0
400 (20x20)	200	300	50
900 (30x30)	/	250	/
1600 (40x40)	/	200	/
2500 (50x50)	/	200	/

Tabella 4.17: Confronto dei gradienti sulla base

degli scenari simulabili con 5 gradienti

Performance relative agli scenari simulabili con 7 gradienti:

Numero di nodi	Numero massimo di agenti per gradiente YOUNGEST	Numero massimo di agenti per gradiente SAPERE	Variazione % del numero massimo di agenti
100 (10x10)	350	350	0
400 (20x20)	50	300	500
900 (30x30)	/	250	/
1600 (40x40)	/	200	/
2500 (50x50)	/	200	/

Tabella 4.18: Confronto dei gradienti sulla base

degli scenari simulabili con 7 gradienti

Dai confronti effettuati è evidente che il gradiente SAPERE sia nettamente più performante del gradiente YOUNGEST, in particolar modo quando sono presenti più gradienti contemporaneamente nell'ambiente, requisito chiave per lo sviluppo di pattern complessi. Il gradiente YOUNGEST ha performance peggiori, a volte di un ordine di grandezza, rispetto al gradiente

SAPERE secondo tutti i parametri di confronto, motivo per cui il gradiente SAPERE è stato usato nello sviluppo dei pattern di questa tesi.

Capitolo 5

Pattern sviluppati

In questo capitolo verranno descritti i pattern di auto-organizzazione sviluppati nel corso di questa tesi, per ogni pattern verrà fornita una descrizione, una panoramica delle LSA e delle eco-law impiegate per realizzare il pattern, i vari scenari in cui è possibile applicare il pattern e i risultati delle simulazioni effettuate in Alchemist.

5.1 Segregation pattern

Il primo pattern sviluppato è stato chiamato Segregation Pattern, è un pattern piuttosto semplice che ha come scopo quello di dividere l'ambiente in porzioni tali per cui ogni nodo contenuto in una porzione si trova alla minore distanza possibile dalla sorgente di un gradiente. Più semplicemente in un ambiente in cui sono presenti diversi gradienti ogni nodo conterebbe tuple relative a tutti i gradienti, con l'applicazione di questo pattern ogni nodo conterrà solo la tupla relativa al gradiente a cui è più vicino alla sorgente.

5.1.1 Modello

Per prima cosa è stato necessario individuare i requisiti alla base del problema che si intende risolvere con il Segregation pattern:

- Topologia dell'ambiente arbitraria

- Numero di gradiente ignoto a priori e che può variare nel tempo
- Possibile mobilità dei nodi

La politica scelta per sviluppare questo pattern è stata molto semplice: ogni nodo controlla le tuple relative ai gradienti che contiene, se sono presenti due o più tuple relative a gradienti differenti solo quella con l'indicazione della distanza minore resterà nel nodo, le altre verranno consumate. Questo pattern è applicabile, ovviamente, solo in presenza di uno o più gradienti, la LSA usata per rappresentare il gradiente è:

```
lsa <grad, Type, Distance>
```

ovvero una tupla di tipo `grad` che ha due campi, uno chiamato `Type` che indica qual'è il gradiente di appartenenza, in modo da discriminare tuple di tipo grad provenienti da sorgenti differenti ed uno chiamato `Distance` che non è altro che l'indicazione della distanza dalla sorgente.

Grazie alla scelta di questa LSA è stato molto semplice realizzare il pattern che è composto dalla generazione di un gradiente a cui viene aggiunta una singola eco-law:

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, source,
gradient, 2, Distance+#D, null, 2000000, 1" [] --> []

eco-law seg
[<grad, T1, D1>, <grad, T2, def: D2>D1>] --> [<grad, T1, D1>]
```

Questa eco-law non fa altro che realizzare la strategia sopra descritta: consuma due tuple di tipo `grad` e viene reinserita solo quella con indicazione di distanza dalla sorgente minore. Il rate di questa eco-law è ovviamente ASAP in quanto si vuole garantire la massima reattività a cambiamenti topologici dell'ambiente.

Il modello del pattern è stato descritto direttamente tramite il DSL di Alchemist descritto nel capitolo 3.5, questa scelta è stata motivata dal fatto che si vuole essere il più espressivi possibile. Per la specifica completa del modello si veda l'Appendice A.

5.1.2 Scenari applicativi

Questo pattern non ha vere e proprie applicazioni se usato da solo ma può essere molto utile se combinato con altri pattern per realizzare pattern complessi, ne è un esempio la Alarm Sensor Network presentata nel sottocapitolo 5.2

5.1.3 Simulazione del pattern in Alchemist

Le simulazioni effettuate servono a dimostrare le caratteristiche di robustezza ed adattività del pattern, vengono presentati diversi scenari:

- Scenario1: è il più semplice, consiste in un ambiente composto da 900 nodi disposti regolarmente su una griglia 30x30 dove sono presenti già 5 sorgenti di gradiente immutabili nel tempo
- Scenario 2: simile allo scenario 1 ma le sorgenti dei gradienti compaiono e scompaiono in tempi differenti tramite apposite eco-law, serve a dimostrare la robustezza del pattern rispetto al numero di gradienti.
- Scenario 3: in questo scenario i nodi non sono disposti su una griglia regolare ma posizionati nell'ambiente in modo casuale, serve a dimostrare l'indipendenza del pattern rispetto alla topologia dell'ambiente al quale viene applicato.
- Scenario 4: in questo scenario le sorgenti possono muoversi di nodo in nodo.

5.1.3.1 Scenario 1

Situazione iniziale, i pallini neri più grossi indicano la presenza di una sorgente di gradiente nel nodo corrispondente:

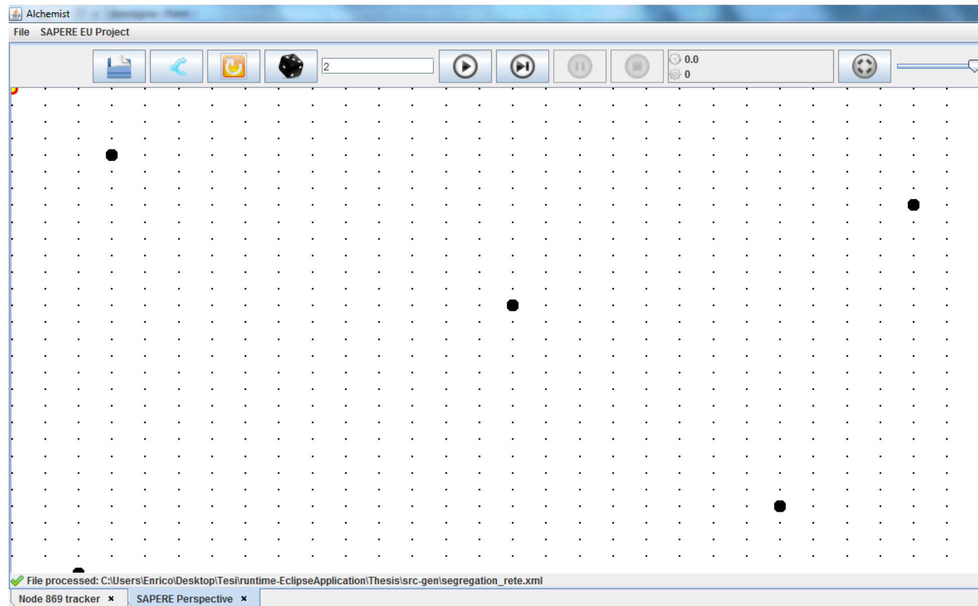


Figura 5.1: Segregation, Scenario 1, situazione iniziale

Situazione finale, ogni nodo contiene al suo interno solo la tupla relativa al gradiente più vicino, i gradienti sono distinguibili tra loro grazie ai diversi colori che gli sono stati assegnati:

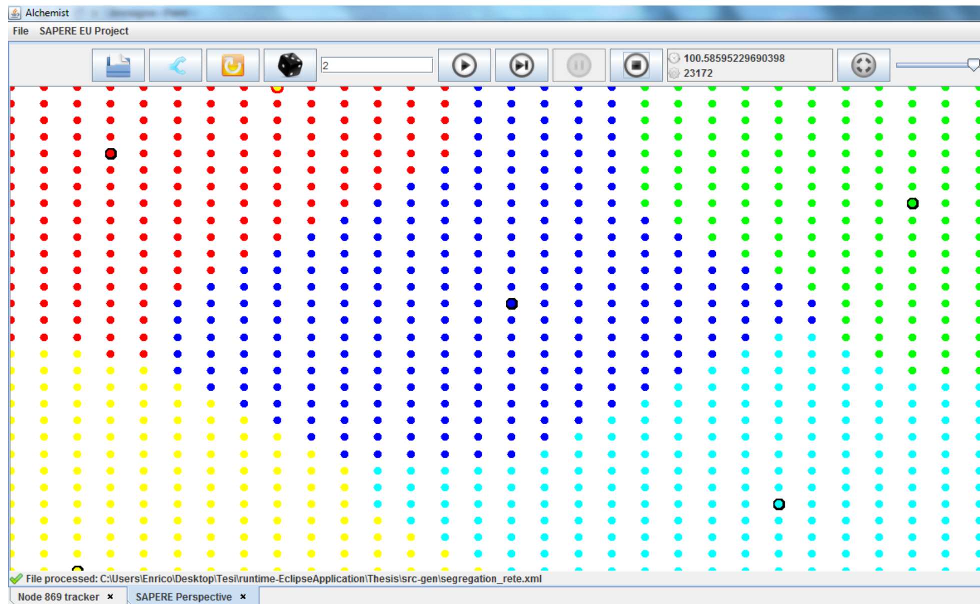


Figura 5.2: Segregation, Scenario 1, situazione finale

5.1.3.2 Scenario 2

Situazione iniziale, ancora nell'ambiente non è presente alcuna sorgente di gradiente:

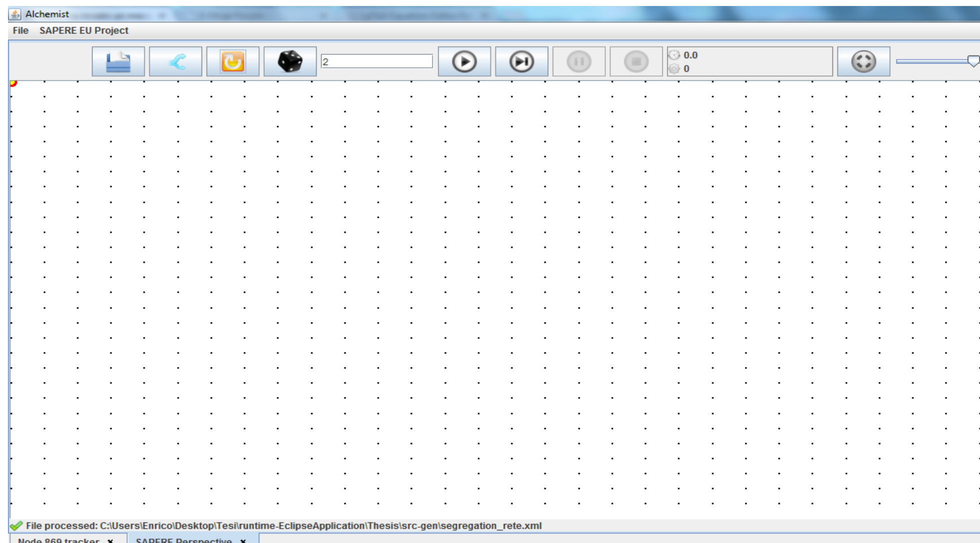


Figura 5.3: Segregation, Scenario 2, situazione iniziale

Comparsa della prima sorgente:

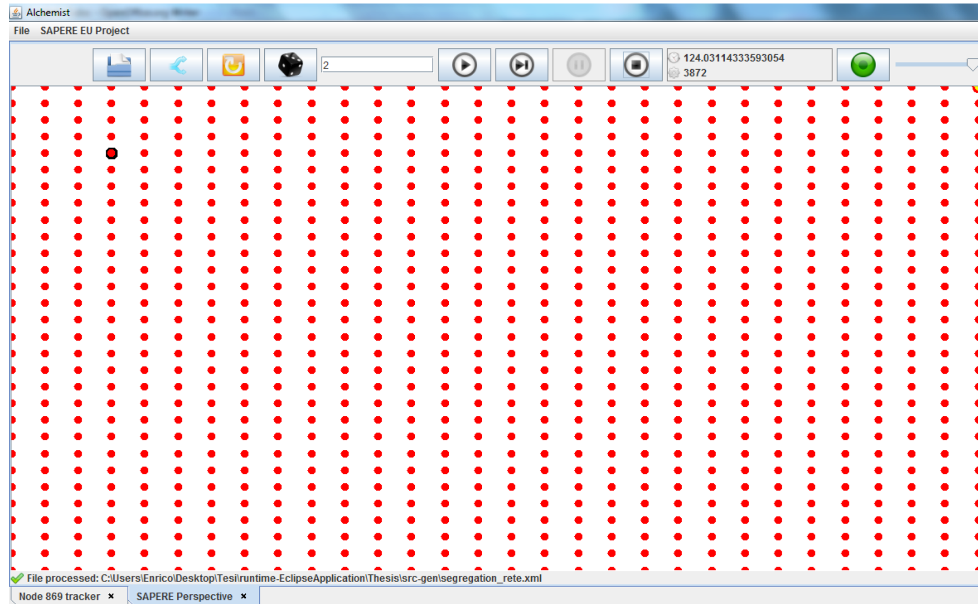


Figura 5.4: Segregation, Scenario 2, comparsa della prima sorgente

Comparsa della seconda sorgente:

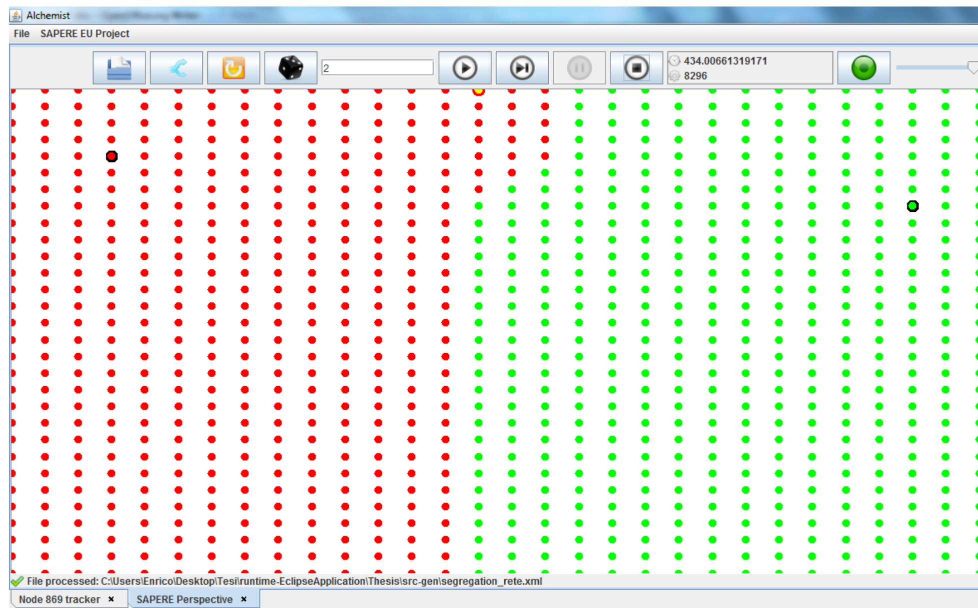


Figura 5.5: Segregation, Scenario 2, comparsa della seconda sorgente

Comparsa della terza sorgente:

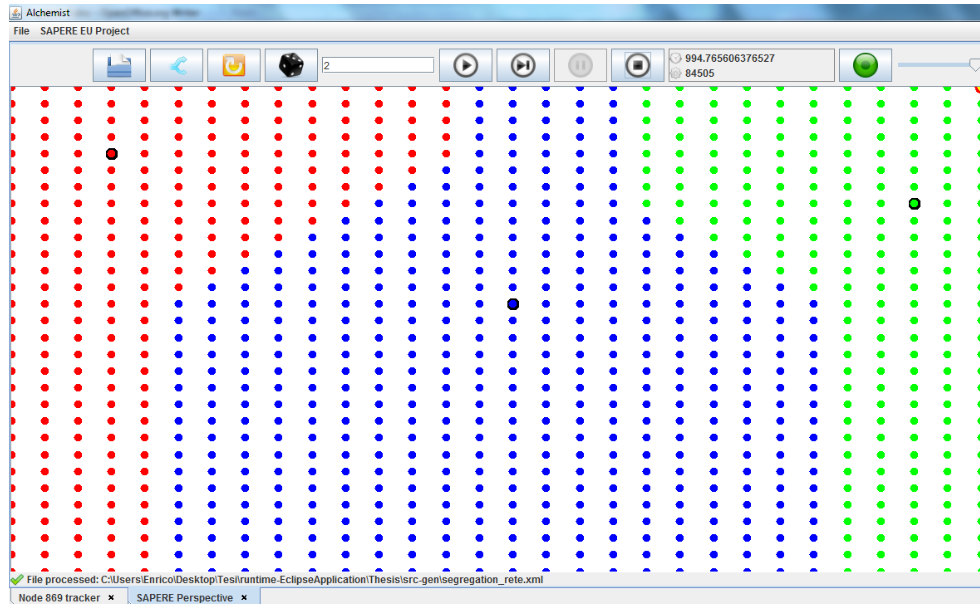


Figura 5.6: Segregation, Scenario 2, comparsa della terza sorgente

Comparsa della quarta sorgente:

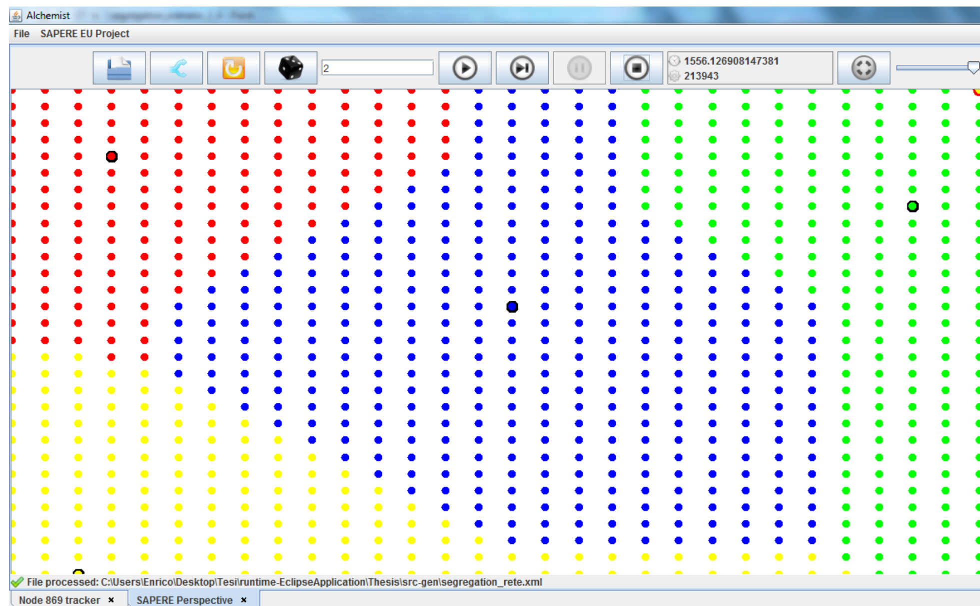


Figura 5.7: Segregation, Scenario 2, comparsa della quarta sorgente

Comparsa della quinta sorgente:

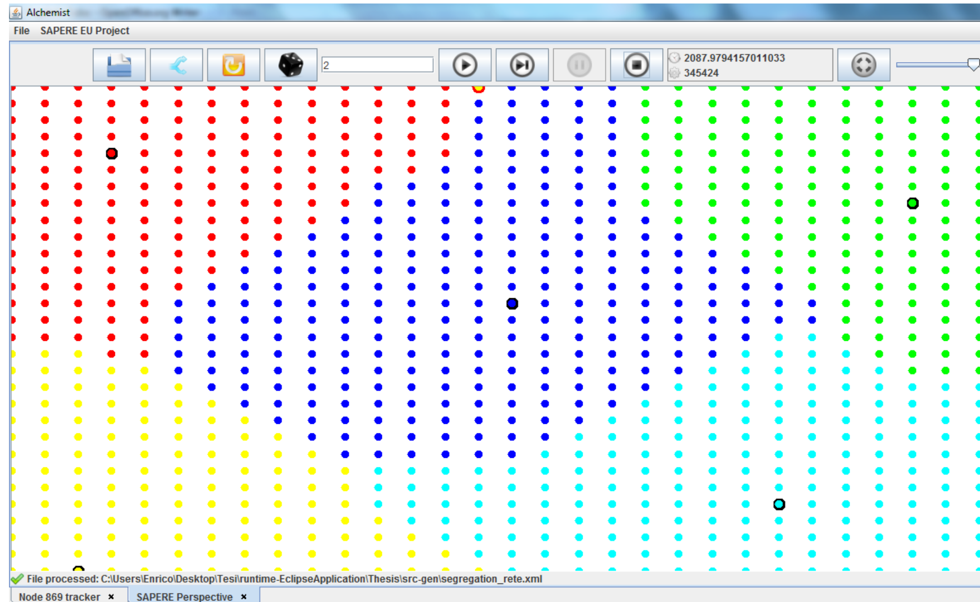


Figura 5.8: Segregation, Scenario 2, comparsa della quinta sorgente

Scomparsa della seconda sorgente:

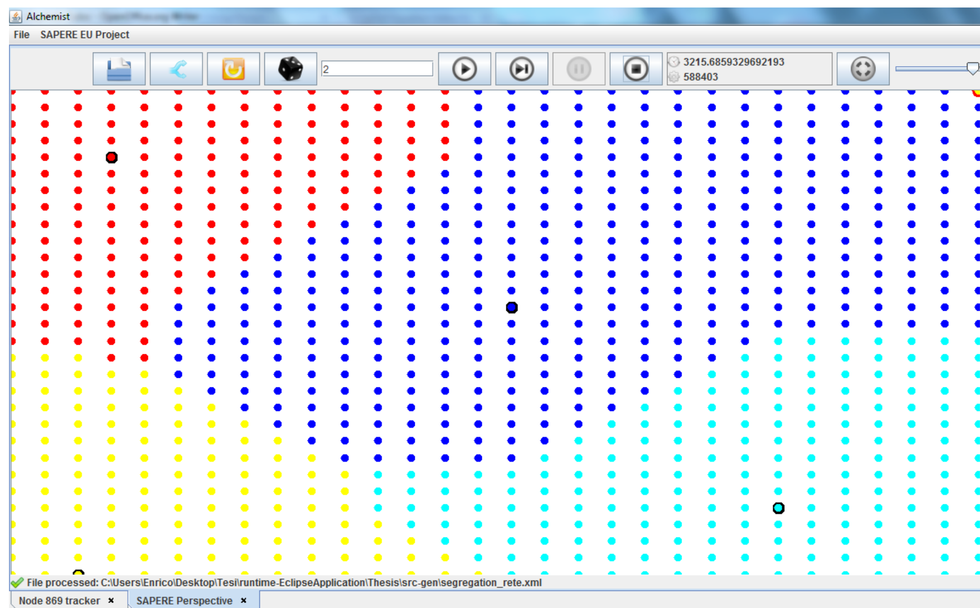


Figura 5.9: Segregation, Scenario 2, scomparsa della seconda sorgente

Scomparsa della terza sorgente:

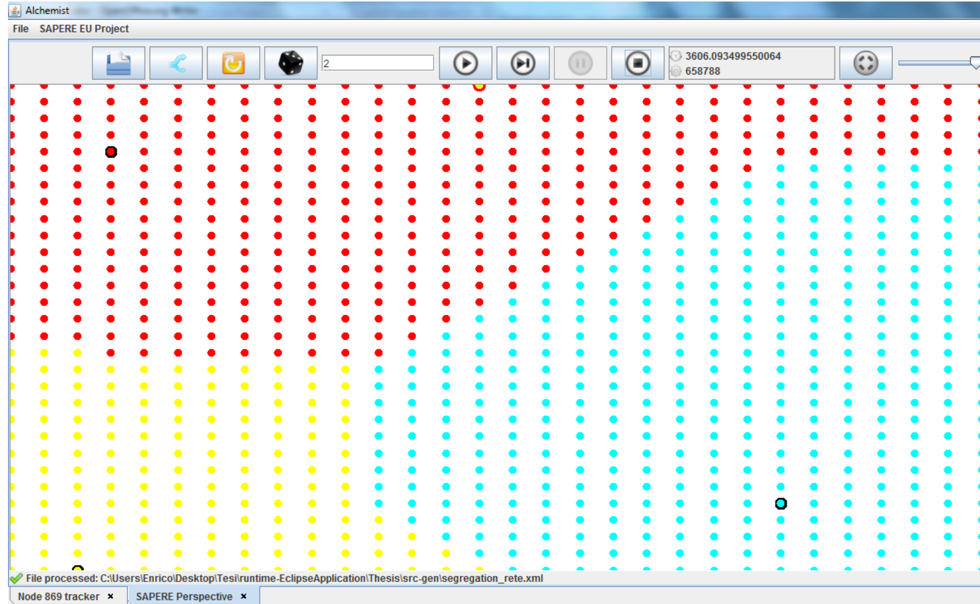


Figura 5.10: Segregation, Scenario 2, comparsa della terza sorgente

La simulazione conferma il comportamento atteso rispetto al numero variabile di sorgenti di gradiente presenti nell'ambiente.

5.1.3.3 Scenario 3

Situazione iniziale:

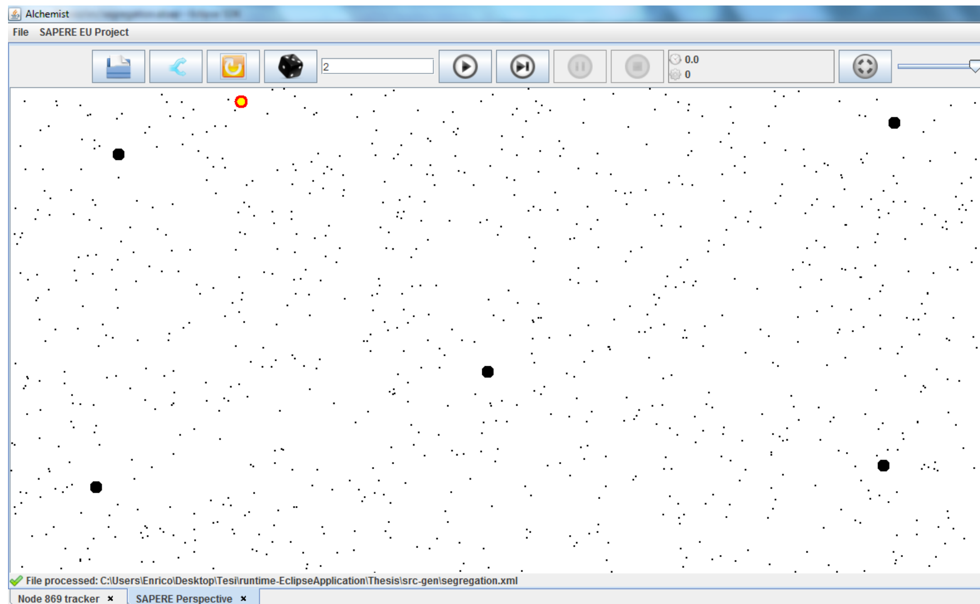


Figura 5.11: Segregation, Scenario 3, situazione iniziale.

Situazione finale:

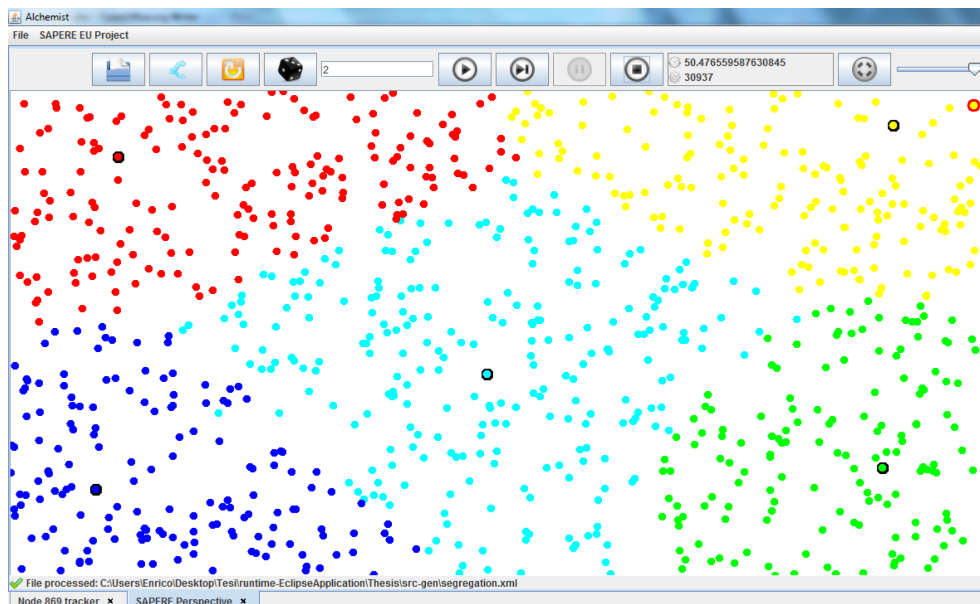


Figura 5.12: Segregation, Scenario 3, situazione finale.

Il pattern si dimostra indipendente dalla topologia dell'ambiente.

5.2 Alarm Sensor Network

Questo pattern nasce con l'idea di essere applicato su una rete di sensori al fine di identificare e segnalare valori anomali rilevati dalla rete stessa.

5.2.1 Modello

La rete di sensori su cui il pattern andrà ad agire è rappresentata come una griglia di nodi, ad ogni nodo corrisponde un sensore. L'idea di base è stata quella di generare, in corrispondenza di ogni valore errato, un gradiente con un raggio limitato, contare in ogni nodo quanti gradienti sono presenti e, se il numero di gradienti presenti è maggiore di una certa soglia, diffondere sulla rete un gradiente di allarme che può propagarsi secondo diverse politiche.

Le LSA utilizzate in questo modello sono:

```
lsa value <value, Val>
lsa sample <sample, Type, Dist>
lsa warning <warning, Type, Dist>
lsa count <count, Type, Num>
lsa alarm <alarm, Node, Time, Dist>
lsa alarm2 <alarm2, Node, Time, Dist>
```

La prima LSA modella il valore rilevato dal sensore, può essere un qualsiasi tipo di valore, come temperatura, pressione o corrente elettrica, il valore corrispondente è memorizzato nel campo `Val`. La seconda LSA è la tupla che rappresenta la sorgente del gradiente che verrà diffuso in corrispondenza di ogni valore errato rilevato dai sensori, l'LSA `warning` rappresenta il gradiente che verrà diffuso. L'LSA `count` è una tupla di appoggio per il conteggio del numero di gradienti in ogni nodo, le LSA `alarm` ed `alarm2` rappresentano rispettivamente la sorgente ed il gradiente di allarme che verrà diffuso sulla rete.

Per realizzare questo pattern sono state necessarie le seguenti eco-law:

```
eco-law detect
[<value, def: Val > 100>] --> [<sample, #NODE, 0>]

reaction SAPEREGradient params "ENV, NODE, RANDOM, sample,
warning, 2, Dist+#D, null, 5, 1" [] --> []

eco-law start_count
[<warning, Type, Dist>] -1-> [<warning, Type, Dist>, <count,
[Type;], 1>]

eco-law keep_count
[<count, List1, N>, <count, def: List2 hasnot [List1;], def:
N2>=N>] --> [<count, List2 add [List1;], N2+N>]

eco-law remove_copies
[<count, List1, N1>, <count, def: List2 has [List1;], def: N2 >
N1>] --> [<count, List2, N2>]

eco-law spread_alarm
[<count, List, def: Num>2>] --> [<alarm, #NODE, #T, 0>]

eco-law oldest_alarm
[<alarm, Node, T1, Dist>, <alarm, Node, def: T2>T1, Dist>] -->
[<alarm, Node, T1, Dist>]
```

Per prima cosa viene controllato in ogni nodo se il valore rilevato dal sensore è corretto o meno, in questo caso un valore errato si ha quando il valore rilevato è maggiore di 100, se è errato viene generata la tupla di origine del gradiente `warning` che contiene come informazione l'ID del nodo che ha rilevato il valore errato. Il gradiente generato non si diffonde su tutta la rete ma ha un raggio limitato, in questo caso 5 (penultimo paramentro della reaction che segue l'eco-law di `detect`). A questo punto si può iniziare a contare quanti

gradienti di warning sono presenti in ogni nodo, questo avviene con l'eco-law start_count che inserisce in ogni nodo in cui ho un gradiente warning anche la tupla di count che porta con sé l'indicazione relativa al gradiente a cui fa riferimento. Il rate di questa eco-law non è ASAP perché questa eco-law va eseguita periodicamente, se fosse eseguita appena possibile il sistema si bloccherebbe. Supponendo di avere tre gradienti di warning presenti in un stesso nodo avrò quindi tre tuple di tipo count, ognuna relativa ad un gradiente e con valore 1, è necessario quindi aggregare queste tuple per ottenere l'informazione del numero preciso di gradienti presenti nel nodo. L'aggregazione avviene con l'eco-law di keep_count: date due tuple di tipo count si considera quella che ha il valore Num più elevato (al limite uguale) e si controlla che nella sua lista dei gradienti contati non sia presente l'ID del gradiente presente nell'altra tupla, in caso positivo la tupla con conteggio più alto viene aggiornata (inserisco l'ID del gradiente nella lista ed aumento il conteggio di 1) mentre l'altra tupla viene consumata. Questo procedimento alla fine produce una tupla di tipo count, la cui lista conterrà tutti i gradienti presenti nel nodo ed il campo Num il loro numero, nel realizzare questo però verranno prodotte anche tuple count parziali, sarà compito dell'eco-law remove_copies rimuovere queste tuple, in modo da non intasare il sistema con delle informazioni inutili. Tutte le eco-law coinvolte nel processo di generazione della tupla di count e del relativo conteggio sono ASAP in modo da essere il più possibile reattive e fornire subito l'informazione richiesta.

L'eco-law spread_alarm controlla le tuple di count, non appena il valore dei gradienti conteggiato in ogni nodo supera una certa soglia (in questo caso 2) viene inserito nel nodo la tupla sorgente del gradiente di allarme nella quale è indicato l'ID del nodo della rete ed il tempo di sistema in cui il gradiente di allarme è stato generato. Sapere in che momento è stato generato il gradiente di allarme è fondamentale in quanto mi permette, tramite l'eco-law

`oldest_alarm` di non considerare tutti gli allarmi generati in quel nodo dopo il primo.

A questo punto, visto che nello stesso momento nell'ambiente possono esserci diversi gradienti di allarme contemporaneamente, sono state pensate tre diverse politiche di diffusione:

- Viene diffuso in tutto l'ambiente solo il primo gradiente di allarme generato, gli altri saranno "cannibalizzati" dal gradiente meno recente.
- Ogni gradiente di allarme generato non si diffonde in tutto l'ambiente ma solo entro un certo raggio R, entro questo raggio sarà sempre il gradiente meno recente a prevalere sugli altri.
- Ogni gradiente di allarme generato cerca di diffondersi su tutta la rete, entro un certo raggio R dal punto di origine prevale sempre il gradiente meno recente mentre se due nodi di origine di gradienti di allarme sono distanti fra loro più di R si dividono la rete sfruttando il pattern di Segregation descritto nel capitolo 5.1

Per realizzare la prima politica sono state necessarie le seguenti eco-law:

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 20000000, 1" [] --> []

eco-law oldest_grad
[<alarm2, Node1, T1, Dist1>, <alarm2, Node2, def: T2>T1, Dist2>]
--> [<alarm2, Node1, T1, Dist1>]

eco-law kill_source
[+<alarm2, Node1, T1, Dist1>, <alarm, def: Node2!=Node1, def:
T2>T1, Dist2> ] --> [+<alarm2, Node1, T1, Dist1>]
```

Viene generato un gradiente senza limitazione di distanza, quando questo gradiente incontra altri gradienti più recenti questi vengono consumati

(eco-law di `oldest_grad`), così come le loro sorgenti (eco-law di `kill_source`). Alla fine sulla rete di sensori si avrà un unico allarme, quello generato per primo.

Per realizzare la seconda politica si sono utilizzate le stesse eco-law della prima, con la sola differenza che il gradiente generato ha una limitazione sulla distanza massima alla quale si può diffondere:

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 7, 1" [] --> []
```

In questo caso si è scelto che il gradiente può esistere solo entro un'area di raggio 7 dalla sorgente.

Anche la terza politica è realizzata in modo simile alla prima:

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 20000000, 1" [] --> []

eco-law oldest_grad
[<alarm2, Node1, T1, def: Dist1 <=5>, <alarm2, Node2, def:
T2>T1, Dist2>] --> [<alarm2, Node1, T1, Dist1>]

eco-law shortest
[<alarm2, Node1, T1, def: Dist1 >5>, <alarm2, Node2, T2, def:
Dist2 >Dist1>] --> [<alarm2, Node1, T1,Dist1>]

eco-law kill_source
[+<alarm2, Node1, T1, Dist1>, <alarm, def: Node2!=Node1, def:
T2>T1, Dist2> ] --> [+<alarm2, Node1, T1, Dist1>]
```

Viene generato un gradiente senza limitazioni di distanza, l'eco-law di `oldest_grad` mi garantisce che, entro un raggio R dalla sorgente (5 in questo caso), solo il gradiente generato per primo sia presente mentre l'eco-law di `shortest` implementa la segregation in tutte le zone fuori dal raggio R deciso nella specifica.

Il modello completo è disponibile nell'appendice B

5.2.2 Scenari applicativi

Come facilmente intuibile questo pattern può essere applicato in reti di sensori, non essendo pensato per un sensore particolare ha un utilizzo molto flessibile, grazie anche alla possibilità di decidere tramite la specifica tutti i parametri.

5.2.3 Simulazione del pattern in Alchemist

Situazione iniziale, i pallini gialli rappresentano i sensori che stanno rilevando un valore errato:

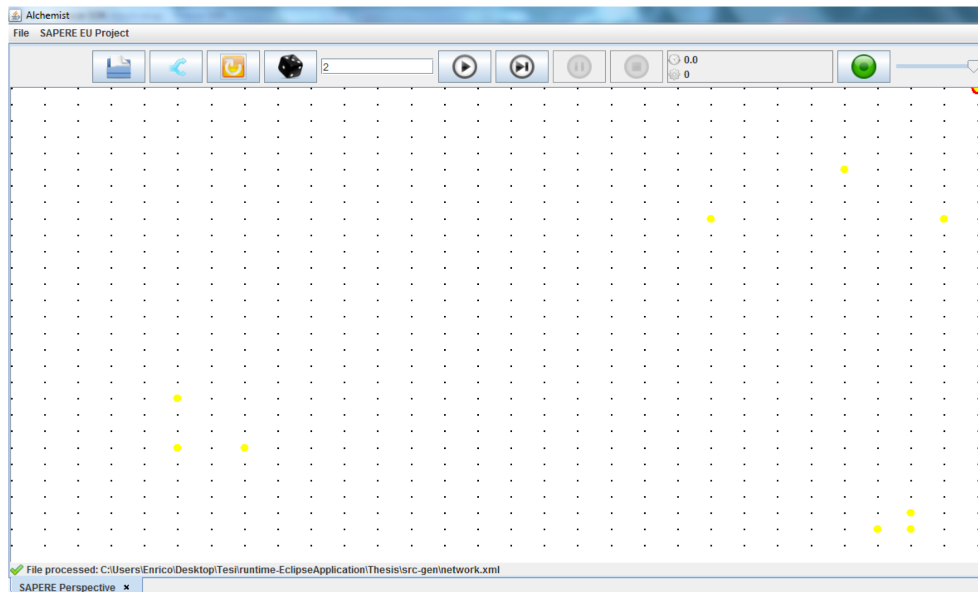


Figura 5.13: Alarm Sensor Network, situazione iniziale.

Vengono generati i gradienti di warning (in arancione) e quasi contemporaneamente si generano le sorgenti dei gradienti di allarme (in blu)

nelle zone in cui il conteggio dei gradienti supera la soglia ed i relativi gradienti di allarme (in rosso):

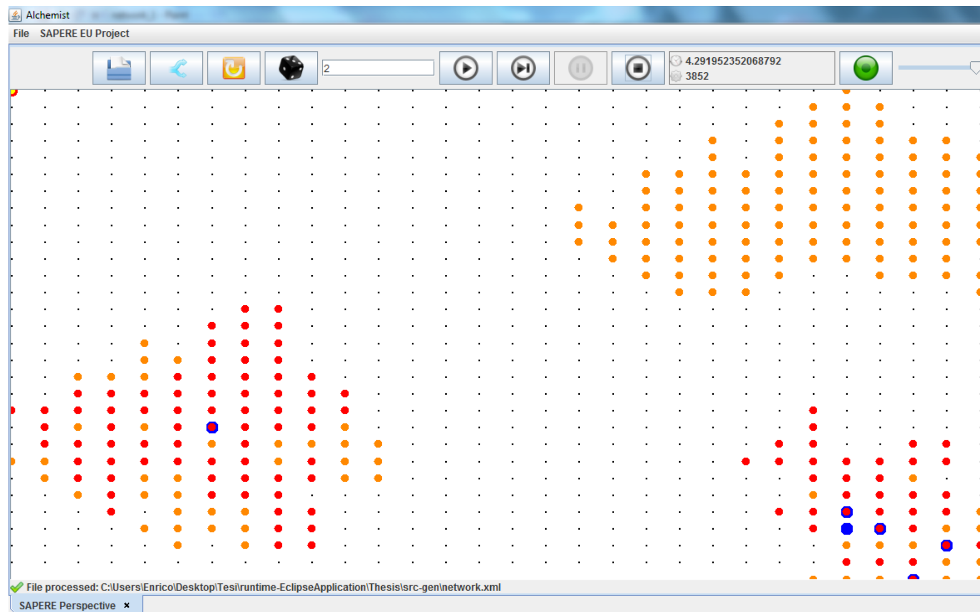


Figura 5.14: Alarm Sensor Network, generazione dei gradienti di warning.

L'allarme si diffonde secondo la prima politica:

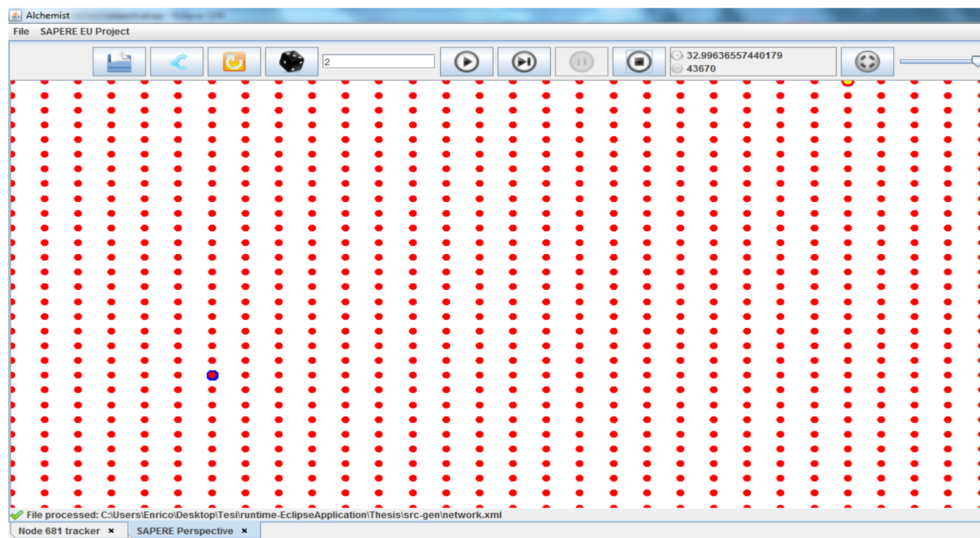


Figura 5.15: Alarm Sensor Network, situazione finale con la prima politica di diffusione

L'allarme si diffonde secondo la seconda politica:

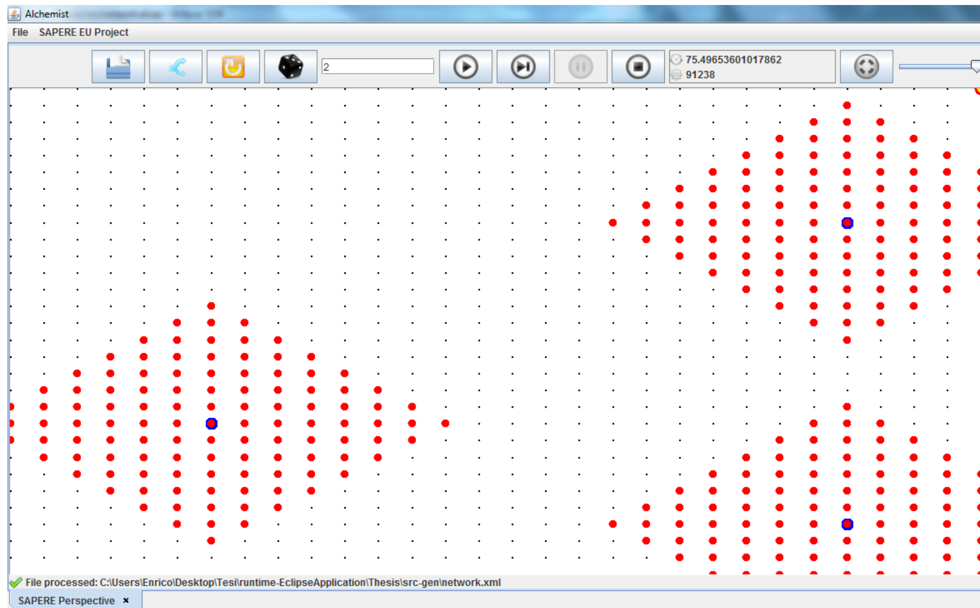


Figura 5.16: Alarm Sensor Network, situazione finale con la seconda politica di diffusione

L'allarme si diffonde secondo la terza politica:

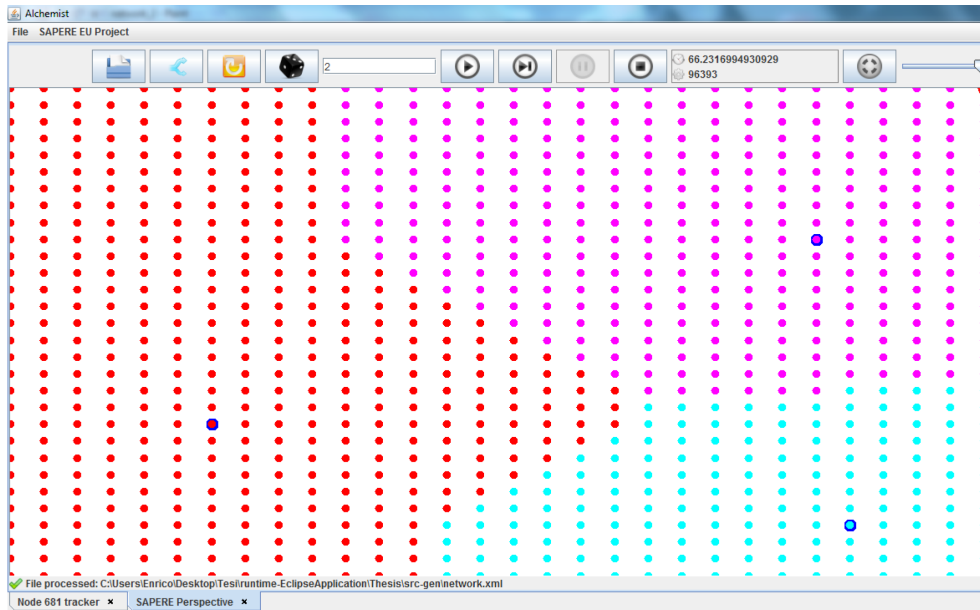


Figura 5.17: Alarm Sensor Network, situazione finale con la terza politica di diffusione

5.3 Gradiente con carico

Lo scopo di questo pattern è quello di indirizzare gli agenti presenti nell'ambiente verso le diverse sorgenti di gradiente in modo da distribuirli in modo più o meno uniforme fra le varie sorgenti.

5.3.1 Modello

Dato un ambiente con un numero di gradienti non noto a priori e variabile nel tempo si ha che, normalmente, gli agenti presenti si dirigono tutti verso la sorgente di gradiente più vicina. Per evitare questo e distribuire gli agenti si è adottata una politica di peggioramento della sorgente: ogni volta che un agente arriva alla sorgente, che in condizioni normali porta indicazione di distanza pari a 0, questa viene peggiorata, in modo da rendere il gradiente in oggetto meno attrattivo per gli agenti che ancora non sono arrivati alla sorgente e magari indirizzarli verso un'altra. Il peggioramento della sorgente consiste, in questo caso, nell'aumento dell'indicazione della distanza di un fattore 1 per ogni agente, il modello può essere però facilmente esteso per peggiorare la sorgente di un fattore K ogni volta che N agenti raggiungono la sorgente.

Le LSA usate in questo pattern sono:

```
lsa  source <source, Type, Distance>
lsa  gradient <grad, Type, Distance>
lsa  temp <temp, List, Counted>
lsa  person <person, Id>
```

Le prime due LSA servono per realizzare i gradienti presenti nell'ambiente, la tupla temp è presente in ogni nodo sorgente ed ha la funzione di tupla di appoggio per realizzare il calcolo del peggioramento della sorgente: il campo List, inizialmente vuoto, memorizza una lista di tutti gli ID dei nodi presenti nella sorgente mentre il campo Counted rappresenta il numero di nodi

presenti nella sorgente. La tupla person rappresenta un agente, identificato dal campo Id.

Le eco-law nei nodi agente:

```
eco-law moveperson
  []-1->[agent LsaAscendingAgent params "REACTION, ENV, NODE, gradient, 2"]

eco-law gen_id [<person, id>] --> [<person, #NODE>]
```

La prima eco-law dice all'agente di risalire il gradiente mentre la seconda serve a generare un identificativo univoco per l'agente in questione, questo è possibile grazie all'uso della keyword `#NODE`.

Le eco-law presenti nei nodi dell'ambiente sono:

```
eco-law peggiora_temp
[+<person, Id>, <temp, def: List hasnot [Id;], Count>] -->
[+<person, Id>, <temp, List add [Id;], Count+1> ]

eco-law refresh_source
[<source, Type, Distance>, <temp, List, Counted>] -1-> [<source, Type, Counted>, <temp, [], 0>]
```

La prima eco-law agisce sulla tupla di appoggio presente nella sorgente: se un agente di tipo `person` è vicino al nodo sorgente (l'agente `person` e la sorgente risiedono in due nodi diversi quindi si parla di vicinato anche quando sono sovrapposti, come nel caso della risalita completa del gradiente) e il suo ID non è ancora presente nella lista degli agenti arrivati alla sorgente allora il suo ID viene aggiunto alla lista e il conteggio dei nodi arrivati alla sorgente viene aumentato di 1.

La seconda eco-law realizza il peggioramento effettivo della sorgente: periodicamente viene controllata la tupla di appoggio ed il conteggio degli

agenti arrivati alla sorgente sarà il nuovo valore di partenza del gradiente in questione, in questo modo più agenti saranno presenti nella sorgente meno attrattivo sarà il gradiente. Ad ogni aggiornamento del valore di partenza della sorgente la tupla di appoggio viene azzerata (la lista viene svuotata e il conteggio degli agenti posto uguale a 0) per permettere un nuovo conteggio degli agenti, questo rende il pattern reattivo ai cambiamenti, senza questo aggiornamento non sarebbe possibile capire se un certo agente è ancora presente nella sorgente o meno. Per questo motivo il rate della prima eco-law deve essere ASAP, in modo da riflettere subito quanti agenti sono presenti nella sorgente, mentre l'aggiornamento deve avvenire periodicamente, se avvenisse ASAP bloccherebbe il sistema in quanto verrebbe sempre computata la tupla di appoggio e poi subito azzerata, dalle prove effettuate il rate ottimo di aggiornamento è pari ad 1.

Il modello completo è disponibile nell'Appendice C

5.3.2 Scenari applicativi

Questo pattern può essere usato per simulare tutti gli scenari in cui è necessario redistribuire il carico, ad esempio scenari di *bike sharing*: ogni sorgente di gradiente è un deposito per biciclette, l'arrivo dell'agente nella sorgente è il deposito della bicicletta, il peggioramento della sorgente serve per indirizzare le biciclette verso depositi più vuoti rispetto a depositi dove già sono presenti molte biciclette.

5.3.3 Simulazione del pattern in Alchemist

Situazione iniziale, due gruppi di agenti e 4 sorgenti di gradiente:

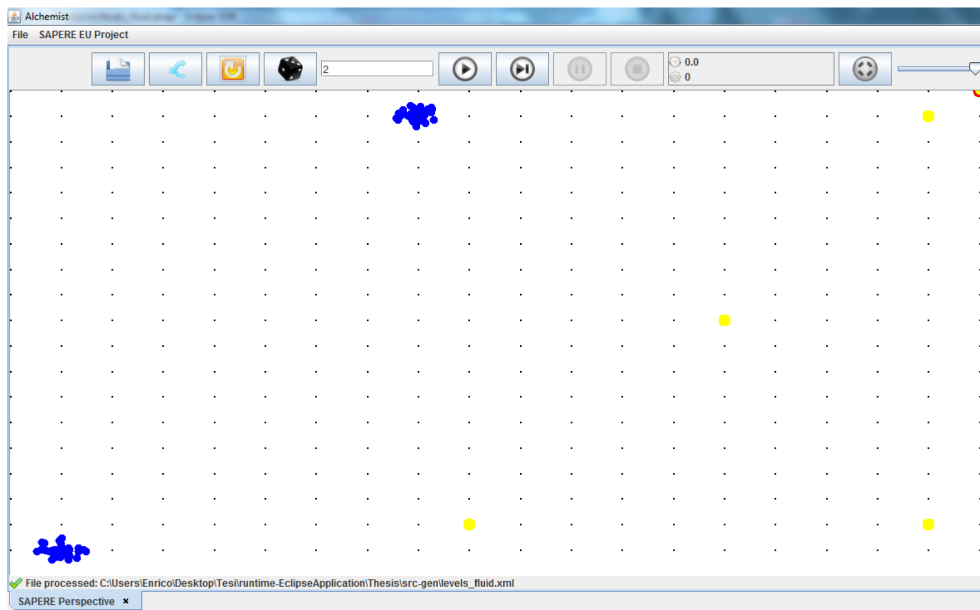


Figura 5.18: Gradiente con carico, situazione iniziale.

Gli agenti inizialmente erano tutti diretti verso la sorgente a loro più vicina ma il numero elevato indirizza altri agenti verso le sorgenti più lontane:

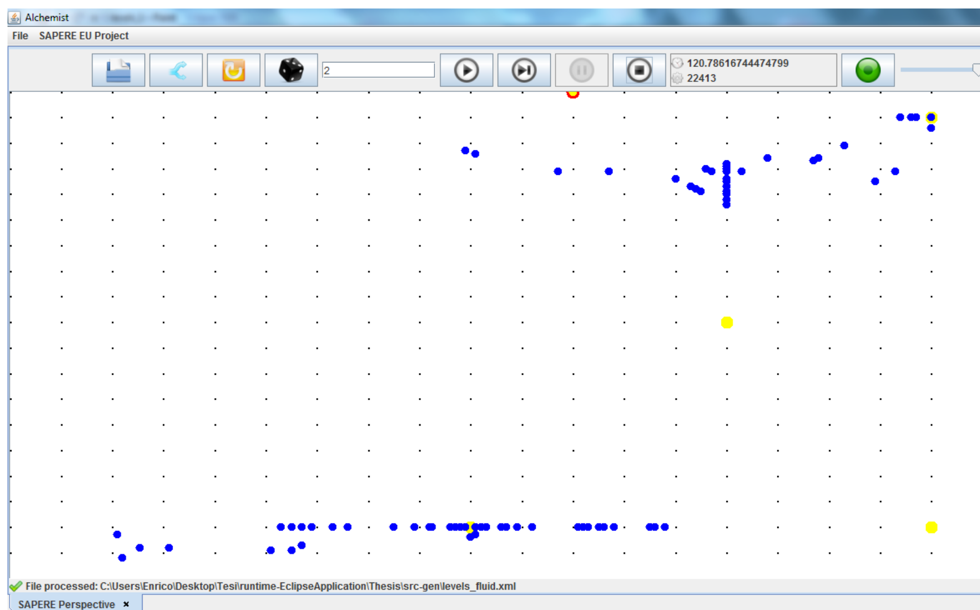


Figura 5.19: Gradiente con carico, situazione intermedia.

Dopo aver raggiunto le sorgenti più vicine fino a rendere poco attrattive gli agenti si distribuiscono verso le sorgenti rimaste:

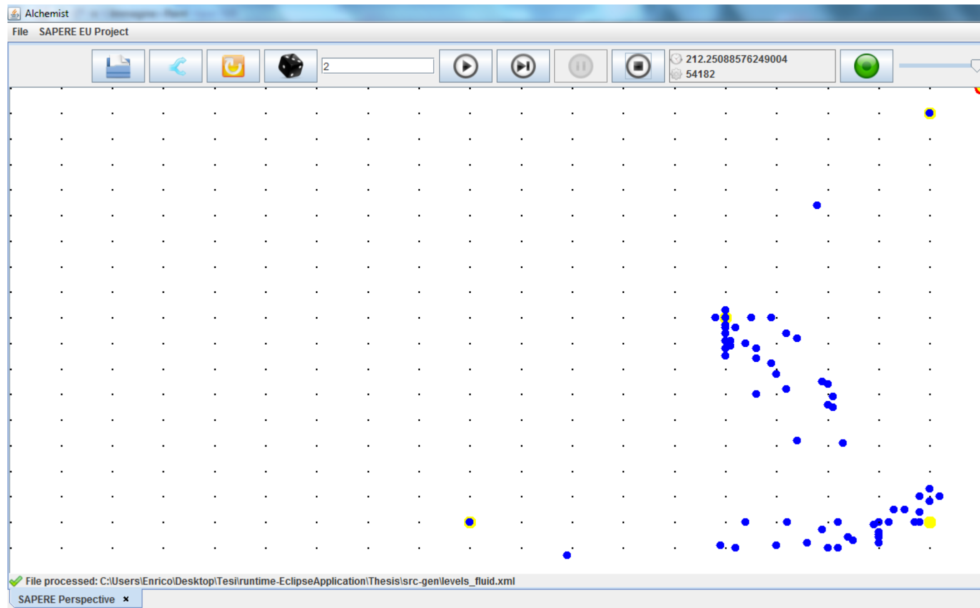


Figura 5.20: Gradiente con carico, saturazione delle sorgenti più vicine.

Anche le sorgenti più lontane sono diventate meno attrattive, gli agenti rimanenti sono distribuiti verso le sorgenti a loro più vicine:

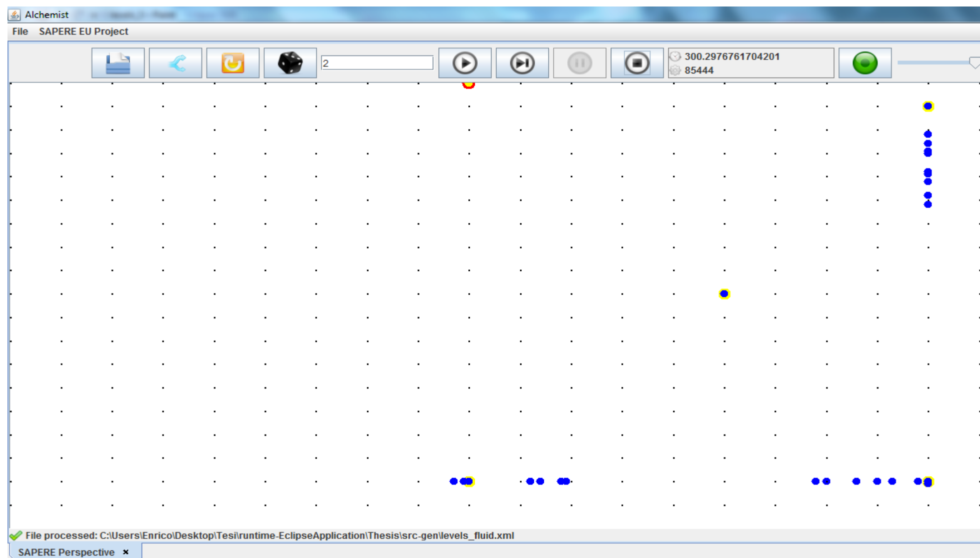


Figura 5.21: Gradiente con carico, verso l'equilibrio.

Equilibrio finale, gli agenti sono distribuiti in numero circa uguale in tutte le sorgenti:

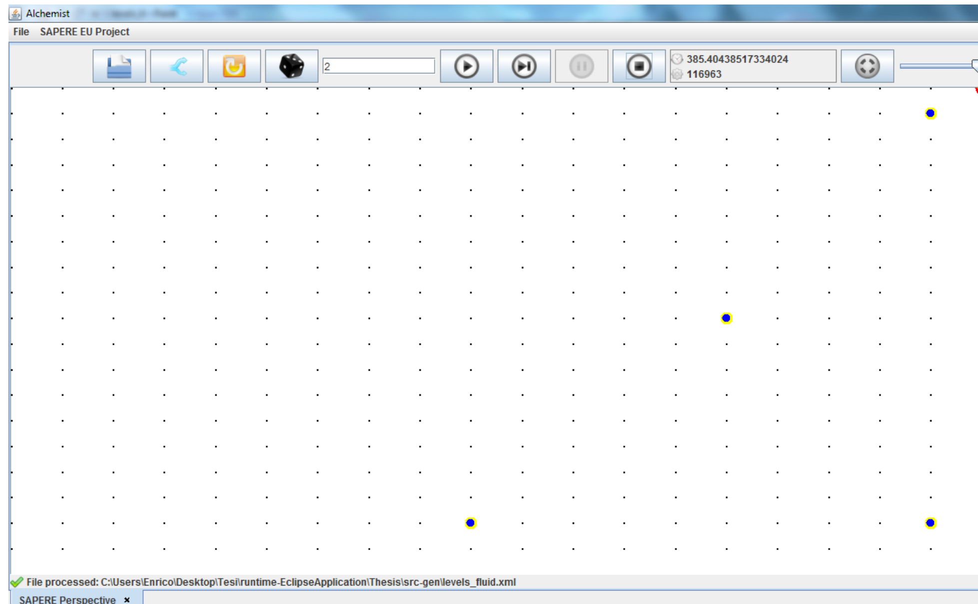


Figura 5.22: Gradiente con carico, equilibrio finale.

5.4 Traccia computazionale

Questo pattern ha come scopo quello di generare una traccia relativa al percorso che un agente seguirà nella risalita del gradiente. È stato sviluppato in due fasi, nella prima si è generata la traccia partendo da agenti fermi, mentre nella seconda fase ogni agente consuma la sua traccia fino a farla sparire completamente una volta che è stata raggiunta la sorgente.

5.4.1 Modello

Le LSA usate nello svulippo di questo pattern sono:

```
lsa  source <source, Type, Distance>
lsa  gradient <grad, Type, Distance>
 lsa person <person, Node, Tracked>
```

```
lsa trace <trace, Node, Dist, Tracked>
```

Le prime due LSA sono necessarie per implementare il gradiente che verrà risalito dagli agenti, l'LSA `person` rappresenta un agente: il campo `Node` è l'identificativo del nodo in cui risiede il gradiente mentre `Tracked` funziona come un flag per indicare se l'agente ha già generato la sua traccia o meno. L'LSA `trace` è una tupla che verrà inserita nei nodi che l'agente percorrerà, la sequenza di nodi che contengono questa tupla comporrà la traccia effettiva, il campo `Node` corrisponde all'identificato dell'agente che l'ha generata, `Dist` la distanza dell'agente dalla sorgente e `Tracked` è un flag che indica se un nodo è già stato analizzato o meno, in modo da non generare calcoli inutili.

Nella prima fase, quella della generazione della traccia con agenti fermi, oltre a generare il gradiente è stato necessario aggiunte due eco-law nei nodi che ospitano gli agenti:

```
eco-law
gen_id [<person, id, 1>] --> [<person, #NODE, Tracked-1>]

eco-law gen_track
[<person, Node, 0>, +<grad, Type, Dist>] --> [<person, Node,
Tracked-1>, +<grad, Type, Dist>, agent LsaTraceAgent params
"REACTION, ENV, NODE, gradient, 2, trace"]
```

La prima eco-law fa in modo che ad ogni agente venga assegnato in valore identificativo univoco tramite l'uso della variabile di sistema `#NODE`, collegata proprio all'ID del nodo. Questo deve essere fatto prima della generazione della traccia in modo che queste siano univoche per ogni agente. Una volta che viene generato l'ID viene settato il flag `Tracked` (valore iniziale 1) a 0 per segnalare che la traccia può essere effettivamente generata.

L'eco-law di generazione della traccia controlla tramite il flag `Tracked` relativo all'agente che questo sia pronto per generare la traccia e che nel suo vicinato sia presente il gradiente che deve risalire (agente e gradiente sono in nodi diversi), se entrambe queste condizioni sono verificate allora viene scelto il nodo del vicinato più vicino alla sorgente da cui partire a generare la traccia. Questa azione è stata implementata in Java in quanto il semplice uso dell'operatore + per inserire il primo step della traccia in un nodo qualsiasi del vicinato portava alla generazione di tracce che non corrispondevano al percorso che l'agente ha intenzione di seguire.

Una volta generato il primo step della traccia saranno presenti in tutti i nodi dell'ambiente (tranne quelli che ospitano gli agenti) le due eco-law responsabili alla diffusione della traccia:

```
eco-law spread
[<trace, Node, Dist, 0>] -10-> [<trace, Node, Dist, 1>, agent
LsaTraceAgent params "REACTION,ENV,NODE,gradient,2,trace"]

eco-law stop
[<trace, Node, Dist, 0>, <source, Type, Distance>] --> [<source,
Type, Distance>]
```

La prima eco-law controlla che nel nodo in questione la traccia non sia già stata calcolata, se questa condizione è soddisfatta usa la stessa azione descritta sopra per l'eco-law `gen_track` per inserire nel miglior nodo del vicinato il successivo step della traccia.

La seconda eco-law serve a fermare la diffusione della traccia quando questa raggiunge la sorgente del gradiente, per evitare che l'eco-law di `spread` continui a calcolare sempre il nodo sorgente come miglior nodo, questo porterebbe all'inserimento di tuple ridondanti che rallenterebbero il sistema. Proprio per questo motivo l'eco-law di `stop` ha un rate ASAP mentre quella di

`spread` ha un rate non ASAP, lo stop deve essere per forza schedulato prima di una eventuale diffusione della traccia nel nodo sorgente.

Nella seconda fase sono state inserite altre due eco-law nei nodi che ospitano gli agenti per farli risalire il gradiente e consumare la traccia relativa durante questa risalita:

```
eco-law moveperson
[]-1->[agent LsaAscendingAgent params "REACTION, ENV, NODE, gradient, 2"]

eco-law remove_track
[<person, Node, Tracked>, +<trace, Node, Dist, def:Flag=1>] -1->
[<person, Node, Tracked>]
```

La prima eco-law serve solo a far risalire il gradiente, la seconda controlla che nei nodi vicini a dove si sta muovendo l'agente sia presente una tupla traccia il suo identificativo sia lo stesso dell'agente, se sono uguali la tupla viene consumata in modo che la traccia sparisca dove passa l'agente.

Il modello completo è disponibile nell'Appendice D

5.4.2 Scenari applicativi

Questo pattern si pone come base per lo sviluppo di pattern più complessi, in questa versione la traccia porta con sé solo la distanza dell'agente dal nodo ma può portare altre informazioni utili, ad esempio il tempo impiegato dall'agente per percorrere la distanza fra diversi nodi. Questo pattern può essere utile come integrazione dell'Anticipative Gradient[6], le tracce generate dagli agenti portano informazioni temporali che permettono di individuare zone in cui ci sarà una alta concentrazione di agenti e permettere quindi di ridirigerli verso zone meno affollate.

5.4.3 Simulazione del pattern in Alchemist

Sono stati simulati due scenari: nel primo si vede l'applicazione del pattern in un ambiente in cui i nodi vengono disposti casualmente mentre nel secondo i nodi dell'ambiente sono disposti su una griglia regolare.

5.4.3.1 Scenario 1

Situazione iniziale, ancora non è presente alcun gradiente nell'ambiente:

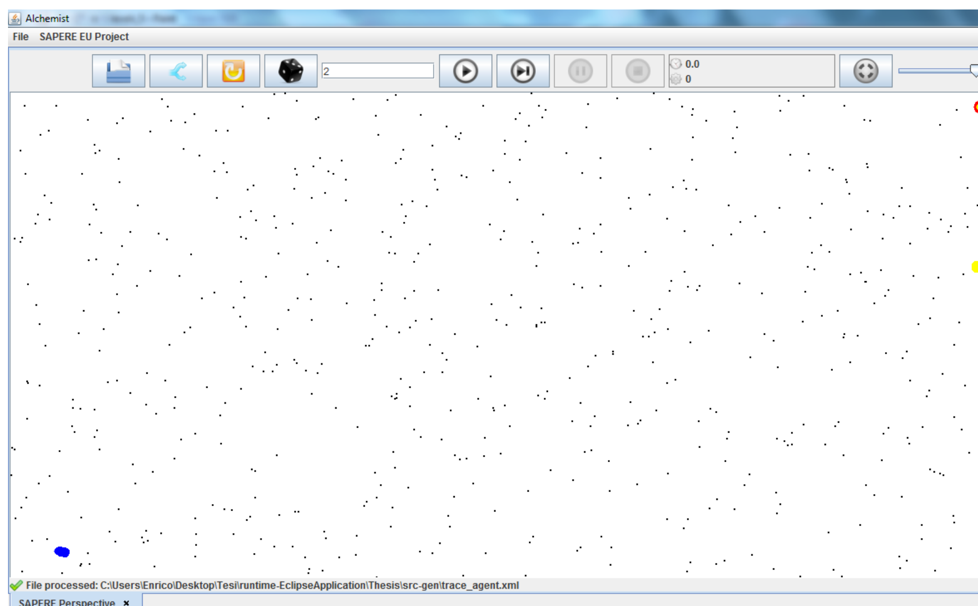


Figura 5.23: Traccia computazionale, situazione iniziale.

Il gradiente (in verde) viene diffuso nell'ambiente:

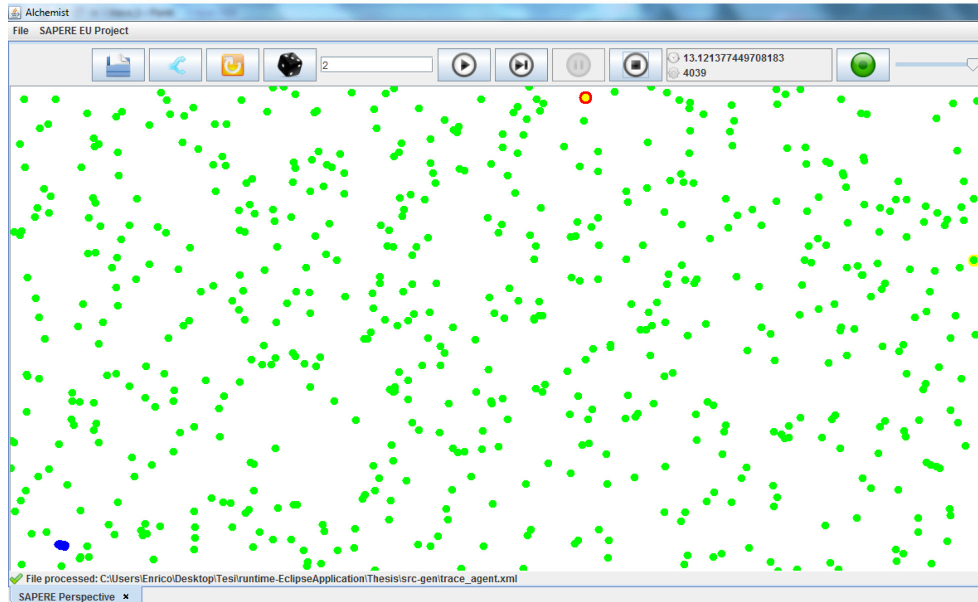


Figura 5.24: Gradiente con carico, generazione del gradiente.

Vengono generate le tracce (in rosso):

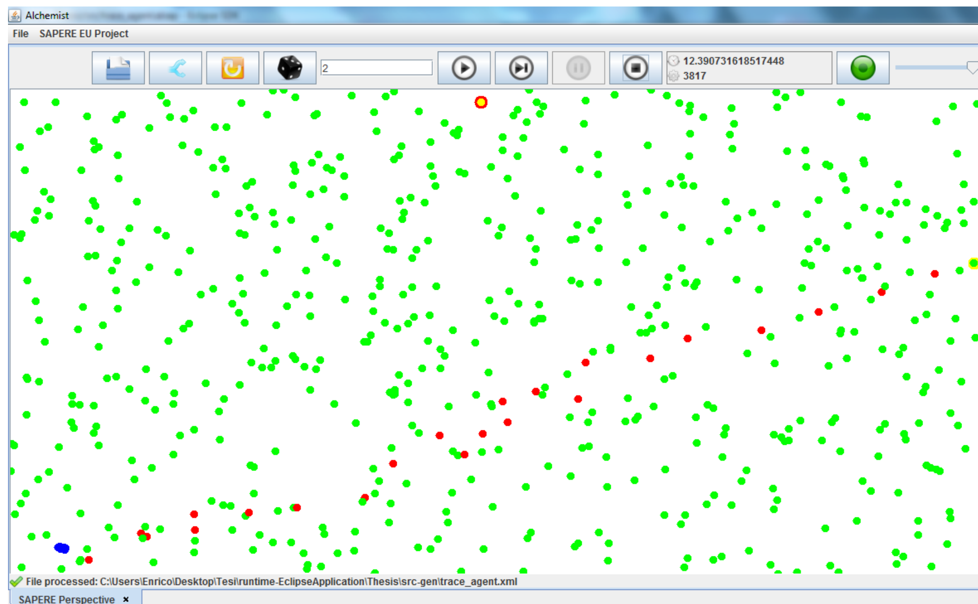


Figura 5.25: Gradiente con carico, generazione delle tracce.

Gli agenti (in blu) consumano le relative tracce:

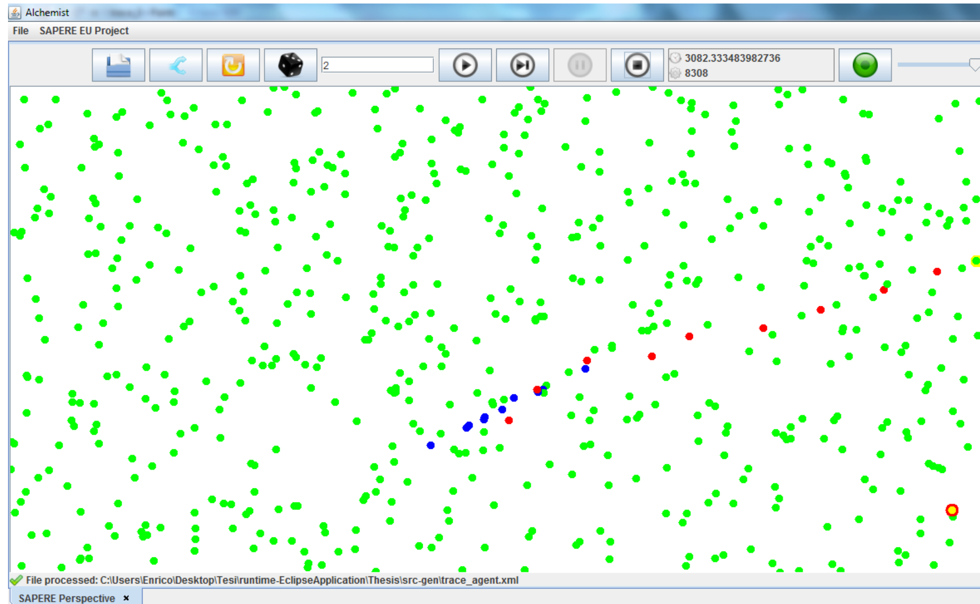


Figura 5.26: Gradiente con carico, gli agenti seguono la traccia, consumandola.

Una volta arrivati a destinazione le tracce sono sparite:

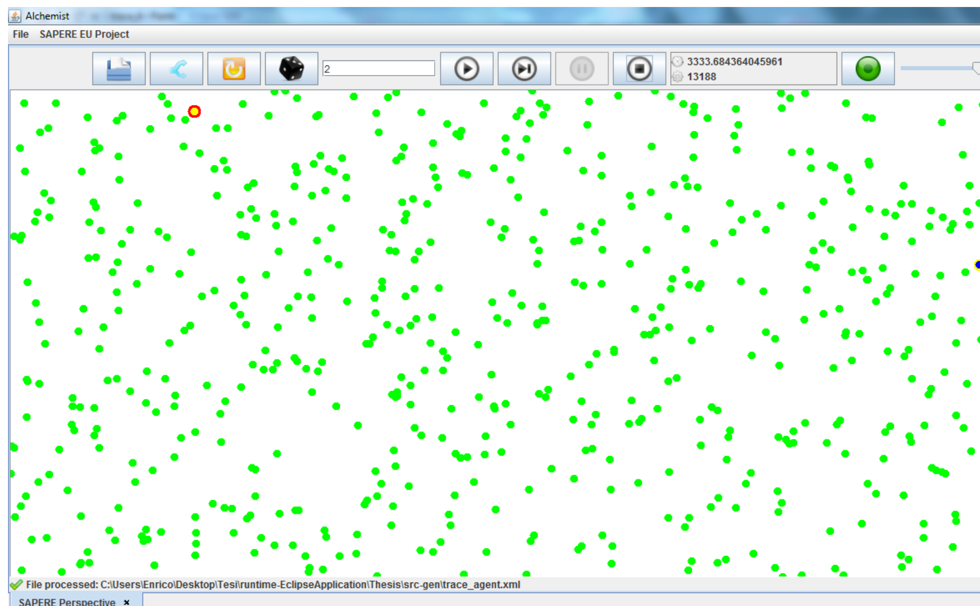


Figura 5.27: Gradiente con carico, situazione finale.

5.4.3.2 Scenario 2

Generazione della traccia:

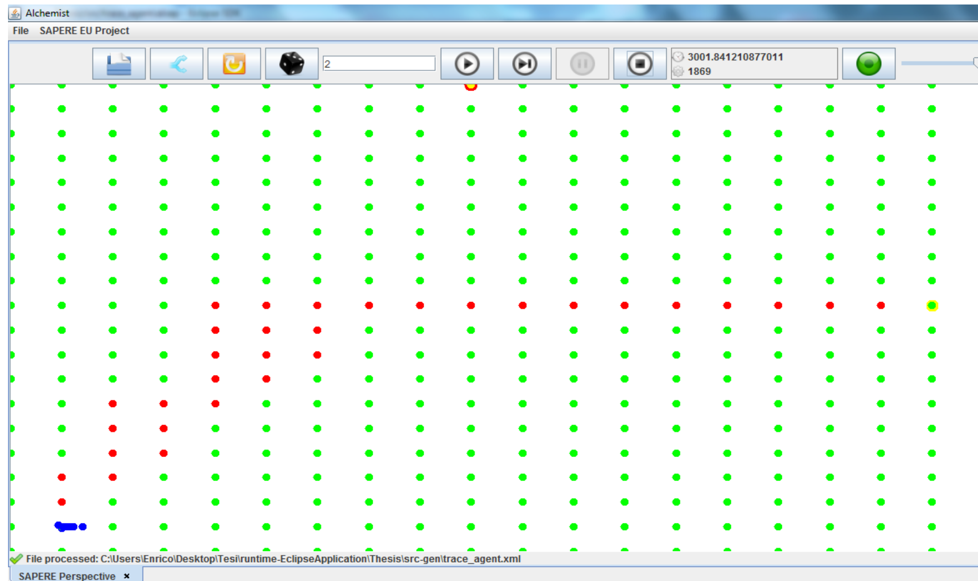


Figura 5.28: Gradiente con carico, generazione della traccia su percorsi equivalenti

La traccia non viene consumata:

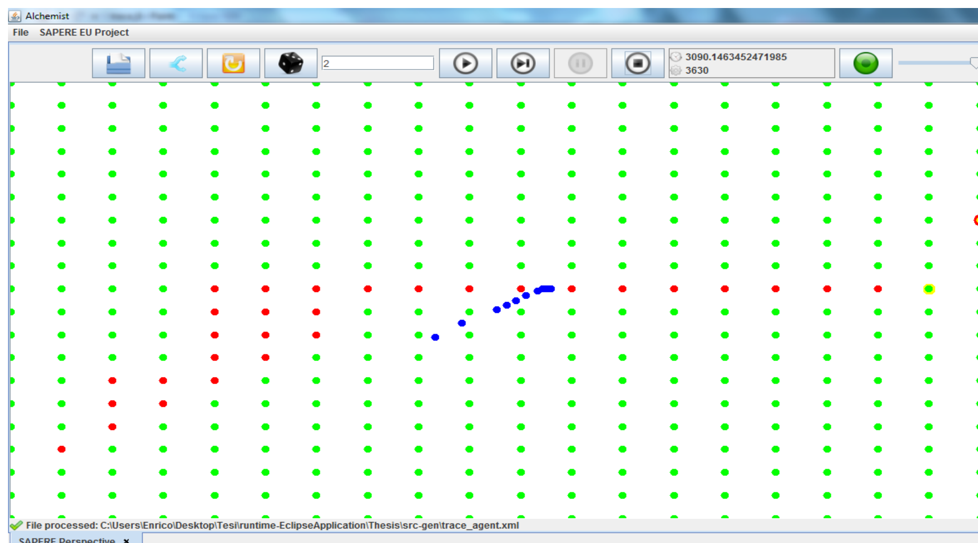


Figura 5.29: Gradiente con carico, gli agenti seguono un solo percorso

Rimangono tracce non consumate nell'ambiente:

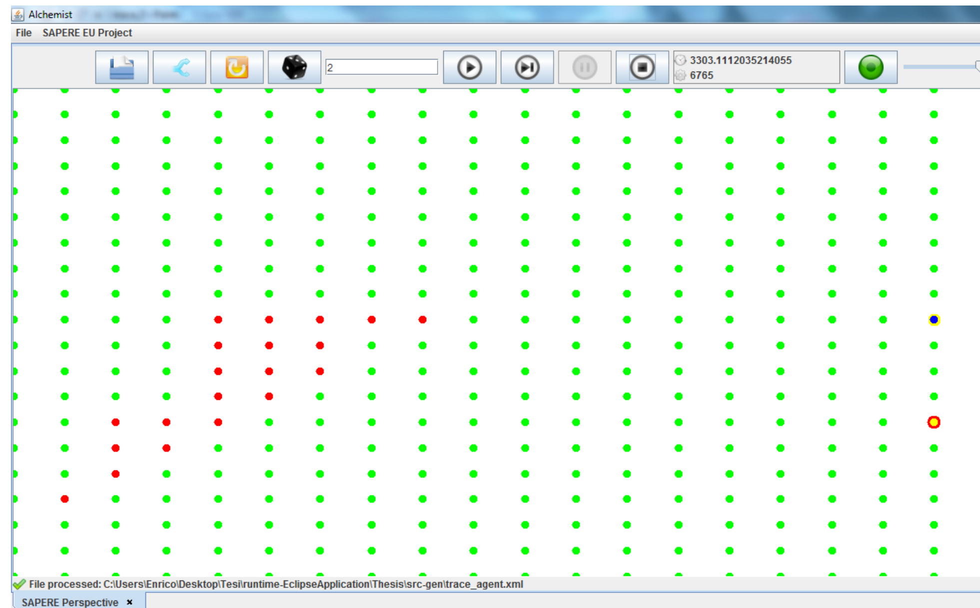


Figura 5.30: Gradiente con carico, tracce non consumate.

Lo scenario 2 evidenzia come questo pattern non sia robusto al 100% rispetto alla topologia dell'ambiente, quando sono presenti percorsi equivalenti non c'è modo di sapere quali di questi verranno scelti per la costruzione della traccia, rimangono così tracce in nodi in cui gli agenti non passano, problema che non si presenta nello scenario 1 in quanto la topologia con i nodi disposti a caso non comporta la creazione di percorsi equivalenti.

Capitolo 6

Conclusioni e sviluppi futuri

Lo scopo di questa tesi era sviluppare algoritmi di auto-organizzazione basati su gradiente computazionale che potessero servire come design pattern nella progettazione di sistemi pervasivi. Dai risultati illustrati nel capitolo 5 si può considerare raggiunto l'obiettivo. In questo capitolo vengono riassunti i contributi della tesi e proposti alcuni sviluppi futuri.

2.1 Riassunto e contributi

L'auto-organizzazione è una proprietà fondamentale per i moderni sistemi computazionali, in particolare per i sistemi pervasivi. In questa tesi si è sfruttato il gradiente computazionale per sviluppare algoritmi di auto-organizzazione che possano garantire ai sistemi pervasivi le caratteristiche di adattività, reattività e tolleranza richieste. Questi algoritmi sono stati modellati e simulati all'interno di un framework di simulazione bio-chimica chiamato Alchemist. Alchemist,sviluppato all'interno del progetto SAPERE, è basato sull'idea che ogni entità del sistema si manifesti tramite delle *Live Semantic Annotation* che evolvono secondo reazioni di ispirazione chimica chiamate *eco-law*.

Dato che lo sviluppo degli algoritmi si basa sul gradiente computazionale è stata eseguita un'approfondita analisi dei gradienti disponibili, cioè YOUNGEST e SAPERE, al fine di individuare il gradiente più adatto

all'uso in questa tesi. Il gradiente SAPERE si è dimostrato il migliore in quanto mantiene alte performance anche in presenza di un numero elevato di gradienti.

Sono stati sviluppati quattro pattern, ogni pattern ha seguito lo stesso iter di sviluppo iterativo composto da: modellazione sulla base dei design pattern esistenti, simulazione per analizzarne la dinamica in diversi scenari, validazione delle proprietà dell'algoritmo in ogni scenario e tuning dei parametri per determinarne la combinazione migliore.

Il risultato delle simulazioni effettuate nel capitolo 5 confermano, tranne in un caso particolare della Traccia computazionale, che tutti gli algoritmi sviluppati soddisfano le caratteristiche richieste di robustezza, generalità ed adattività.

2.2 Sviluppi futuri

Questa tesi è stata sviluppata in ambito di ricerca ed ha diversi possibili sviluppi futuri. Allo stato attuale Alchemist è un simulatore concorrente ed è ottimizzato ad un punto tale da non consentire ulteriori miglioramenti sulle performance, nonostante questo possono essere portati miglioramenti sotto altri aspetti, ad esempio trovare un modo più veloce ed intuitivo per la costruzione degli ambienti di simulazione.

Anche a livello di linguaggio Alchemist è migliorabile, il DSL non è in grado di esprimere tutto quello che si vorrebbe modellare, come dimostrato nello sviluppo del pattern di Traccia computazionale.

I pattern sviluppati in questa tesi possono essere ulteriormente migliorati, possono essere combinati con altri pattern per crearne di nuovi al fine di fornire sempre più soluzioni nella progettazione di sistemi pervasivi.

Bibliografia

[1] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluis Arcos. Description and composition of bio-inspired designpatterns: a complete overview. *Natural Computing*, May 2012. Online First.

[2] Mirko Viroli, Danilo Pianini, Sara Montagna, and Graeme Stevenson. Pervasiveecosystems: a coordination model based on semantic chemistry. In Sascha Ossowski, Paola Lecca, Chih-Cheng Hung, and Jiman Hong, editors, *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Riva del Garda, TN, Italy, 26-30 March 2012. ACM.

[3] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In Maria Ganzha, Leszek Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2011)*, pages 667-674, Szczecin,Poland, 18-21 September 2011. IEEE Computer Society Press.

[4] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 2013.

[5] Danilo Pianini. The Alchemist simulator full manual. Technical report, Universita di Bologna, May 2012.

[6] Sara Montagna, Danilo Pianini, and Mirko Viroli. Gradient-Based Self-Organisation Patterns of Anticipative Adaptation. *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, vol., no., pp.169,174, 10-14 Sept. 2012

[8] Michael A. Gibson and Jehoushua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry* A, 104(9):1876-1889, March 2000.

Appendice A

DSL per Segregation Pattern

A.1 DSL Scenario 1

```
environment -1 1.2
with random seed 2

lsa  source <source, Type, Distance>
lsa  target <source, target, 0>
lsa  target2 <source, target2, 0>
lsa  target3 <source, target3, 0>
lsa  target4 <source, target4, 0>
lsa  target5 <source, target5, 0>
lsa  gradient <grad, Type, Distance>

place 900 nodes in rect (0, 0, 29, 29) interval 1
containing
                  in point (3,25) starget
                  in point(27,22) starget2
                  in point(15,16) starget3
                  in point(2, 0) starget4
                  in point(23, 4) starget5

with reactions

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  source,
gradient, 2, Distance+#D,null, 2000000, 1" [] --> []

eco-law seg
[<grad, T1, D1>, <grad, T2, def: D2>D1>] --> [<grad, T1, D1>]
```

A.2 DSL Scenario 2

```
environment -1 1.2
with random seed 2

lsa  source <source, Type, Distance>
lsa  target <source, target, 0>
lsa  target2 <source, target2, 0>
```

```
lsa  target3 <source, target3, 0>
lsa  target4 <source, target4, 0>
lsa  target5 <source, target5, 0>

lsa  wait_target <wait, target>
lsa  wait_target2 <wait, target2>
lsa  wait_target3 <wait, target3>
lsa  wait_target4 <wait, target4>
lsa  wait_starget5 <wait, target5>

lsa  gradient <grad, Type, Distance>

place 900 nodes in rect (0, 0, 29, 29) interval 1
containing

                  in point (3,25) wait_target
                  in point(27,22) wait_target2
                  in point(15,16) wait_target3
                  in point(2, 0) wait_target4
                  in point(23, 4) wait_target5
with reactions

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  source,
gradient, 2, Distance+#D, null, 2000000, 1" [] --> []

eco-law seg
[<grad, T1, D1>, <grad, T2, def: D2>D1>] --> [<grad, T1, D1>]

eco-law add_poi_1 at time 100
[<wait, target>] --> [<source, target, 0>]

eco-law add_poi_2 at time 400
[<wait, target2>] --> [<source, target2, 0>]

eco-law add_poi_3 at time 900
[<wait, target3>] --> [<source, target3, 0>]

eco-law add_poi_4 at time 1400
[<wait, target4>] --> [<source, target4, 0>]

eco-law add_poi_5 at time 1900
[<wait, target5>] --> [<source, target5, 0>]

eco-law remove_poi2 at time 3100
[<source, target2, 0>] --> []

eco-law remove_poi3 at time 3300
[<source, target3, 0>] --> []
```

A.3 DSL Scenario 3

```
environment -1 2
with random seed 2

lsa  source <source, Type, Distance>
lsa  target <source, target, 0>
lsa  target2 <source, target2, 0>
lsa  target3 <source, target3, 0>
lsa  target4 <source, target4, 0>
lsa  target5 <source, target5, 0>

lsa  gradient <grad, Type, Distance>

place 900 nodes in rect (0, 0, 29, 29)
containing
                  in rect (3,25,1,1) target
                  in rect (26,6,1,1) target2
                  in rect (2,5,1,1) target3
                  in rect (26,26,1,1) target4
                  in rect (14,12,1,1) target5


with reactions

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  source,
gradient, 2, Distance+#D, null, 2000000, 1" [] --> []

eco-law seg
[<grad, T1, D1>, <grad, T2, def: D2>D1>] --> [<grad, T1, D1>]
```

Appendice B

DSL per Alarm Sensor Network

```
environment -1 1.2
with random seed 2

lsa value <value, Val>
lsa val <value, 750>
lsa val2 <value, 150>
lsa val3 <value, 300>
lsa val4 <value, 250>
lsa val5 <value, 150>
lsa val6 <value, 300>
lsa val41 <wait, 250>
lsa val51 <wait, 150>
lsa val61 <wait, 300>
lsa sample <sample, Type, Dist>
lsa warning <warning, Type, Dist>
lsa count <count, Type, Num>
lsa alarm <alarm, Node, Time, Dist>
lsa alarm2 <alarm2, Node, Time, Dist>

place 900 nodes in rect (0, 0, 29, 29) interval 1
containing
                  in point (21,21) val
                  in point (28,21) val2
                  in point (25,24) val3
                  in point (5,10) val
                  in point (5,7) val2
                  in point (7,7) val3
                  in point (26,2) val4
                  in point (27,2) val5
                  in point (27,3) val6

with reactions

eco-law detect
[<value, def: Val > 100>] --> [<sample, #NODE, 0>]

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  sample,
warning, 2, Dist+#D, null, 5, 1" [] --> []

eco-law start_count
[<warning,  Type,  Dist>]  -1->  [<warning,  Type,  Dist>,  <count,
[Type;], 1>]
```

```
eco-law keep_count
[<count, List1, N>, <count, def: List2 hasnot [List1;], def:
N2>=N>] --> [<count, List2 add [List1;], N2+N>]

eco-law remove_copies
[<count, List1, N1>, <count, def: List2 has [List1;], def: N2 >
N1>] --> [<count, List2, N2>]
eco-law spread_alarm
[<count, List, def: Num>2>] --> [<alarm, #NODE, #T, 0>]

eco-law oldest_alarm
[<alarm, Node, T1, Dist>, <alarm, Node, def: T2>T1, Dist>] -->
[<alarm, Node, T1, Dist>]
```

Politica 1

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 20000000, 1" [] --> []

eco-law oldest_grad
[<alarm2, Node1, T1, Dist1>, <alarm2, Node2, def: T2>T1, Dist2>]
--> [<alarm2, Node1, T1, Dist1>]

eco-law kill_source
[+<alarm2, Node1, T1, Dist1>, <alarm, def: Node2!=Node1, def:
T2>T1, Dist2> ] --> [+<alarm2, Node1, T1, Dist1>]
```

Politica 2

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 7, 1" [] --> []


eco-law oldest_grad
[<alarm2, Node1, T1, Dist1>, <alarm2, Node2, def: T2>T1, Dist2>]
--> [<alarm2, Node1, T1, Dist1>]

eco-law kill_source
[+<alarm2, Node1, T1, Dist1>, <alarm, def: Node2!=Node1, def:
T2>T1, Dist2> ] --> [+<alarm2, Node1, T1, Dist1>]
```

Politica 3

```
reaction SAPEREGradient params "ENV, NODE, RANDOM, alarm,
alarm2, 3, Dist+#D, null, 20000000, 1" [] --> []


eco-law oldest_grad
```

```
[<alarm2, Node1, T1, def: Dist1 <=5>, <alarm2, Node2, def:
T2>T1, Dist2>] --> [<alarm2, Node1, T1, Dist1>]

eco-law shortest
[<alarm2, Node1, T1, def: Dist1 >5>, <alarm2, Node2, T2, def:
Dist2 >Dist1>] --> [<alarm2, Node1, T1,Dist1>]

eco-law kill_source
[+<alarm2, Node1, T1, Dist1>, <alarm, def: Node2!=Node1, def:
T2>T1, Dist2> ] --> [+<alarm2, Node1, T1, Dist1>]
```

Appendice C

DSL per Gradiente con Carico

```
environment -1 1.2
with random seed 2

lsa  source <source, Type, Distance>
lsa  target <source, target, 0>
lsa  target2 <source, target2, 0>
lsa  target3 <source, target3, 0>
lsa  target4 <source, target4, 0>
lsa  gradient <grad, Type, Distance>
lsa  temp <temp, List, Counted>
lsa  person <person, Id>

place 400 nodes in rect (0, 0, 19, 19) interval 1

containing        in point (18,2) target
                  in point (18,2) <temp, [], 0>
                  in point (18,18) target2
                  in point (18,18) <temp, [], 0>
                  in point (9,18) target3
                  in point (9,18) <temp, [], 0>
                  in point (9,2) target4
                  in point (9,2) <temp, [], 0>
                  in point (14,10) target3
                  in point (14,10) <temp, [], 0>

with reactions

eco-law peggiora_temp
[+<person,  Id>,  <temp,  def:  List  hasnot  [Id;],  Count>]  -->
[+<person, Id>, <temp, List add [Id;], Count+1> ]

eco-law refresh_source
[<source, Type, Distance>, <temp, List, Counted>] -1-> [<source,
Type, Counted>, <temp, [], 0>]

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  source,
gradient, 2, Distance+#D, null, 2000000, 1" [] --> []

place 10 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law   moveperson   []-1->[agent   LsaAscendingAgent   params
"REACTION,ENV,NODE,gradient,2"]
```

```
eco-law gen_id [<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 20 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]
eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 40 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 60 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 2 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 80 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 2 nodes in circle(1,1,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 100 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]


place 10 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson []-1->[agent LsaAscendingAgent params
"REACTION,ENV,NODE,gradient,2"]
```

```
eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 20 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 40 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 10 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 60 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 2 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 80 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]

place 2 nodes in circle(8,18,0.5)
containing in all <person, id>
with reactions
eco-law moveperson at time 100 []-1->[agent LsaAscendingAgent
params "REACTION,ENV,NODE,gradient,2"]

eco-law gen_id
[<person, id>] --> [<person, #NODE>]
```

Appendice D

DSL per Traccia computazionale

```
environment -1 1.2
with random seed 2

lsa  source <source, Type, Distance>
lsa  target <source, target, 0>
lsa  gradient <grad, Type, Distance>
lsa person <person, Node, Tracked>
lsa trace <trace, Node, Dist, 0>

place 600 nodes in rect (0, 0, 19, 19)

containing  in rect (18,12,1,1) target

with reactions

reaction  SAPEREGradient  params  "ENV,  NODE,  RANDOM,  source,
gradient, 2, Distance+#D, null, 2000000, 1" [] --> []

eco-law spread
[<trace, Node, Dist, 0>] -10-> [<trace, Node, Dist, 1>, agent
LsaTraceAgent params "REACTION,ENV,NODE,gradient,2,trace"]

eco-law stop
[<trace, Node, Dist, 0>, <source, Type, Distance>] --> [<source,
Type, Distance>]

place 10 nodes in circle(1,1,0.1)
containing in all <person, id, 1>
with reactions
eco-law gen_id
[<person, id, def:Tracked=1>] --> [<person, #NODE, Tracked-1>]

eco-law gen_track
[<person,  Node,  def:Tracked=0>,  +<grad,  Type,  Dist>]  -->
[<person,  Node,  Tracked-1>,  +<grad,  Type,  Dist>,  agent
LsaTraceAgent params "REACTION,ENV,NODE,gradient,2,trace"]

eco-law moveperson at time 300
[]-1->[agent  LsaAscendingAgent  params  "REACTION,  ENV,  NODE,
gradient, 2"]

eco-law remove_track at time 300
```

```
[<person, Node, Tracked>, +<trace, Node, Dist, def:Flag=1>] -1->
[<person, Node, Tracked>]
```