ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Design e implementazione di un sistema di grid computing per il simulatore Alchemist

---

Relatore:

Prof. Mirko VIROLI

Correlatore:

Ing. Danilo PIANINI

Presentata da:

Matteo MAGNANI

Relazione finale in Programmazione ad Oggetti
Seconda sessione di laurea
Anno Accademico 2016/2017

Dedicato alla mia famiglia, che mi ha sempre sostenuto

Sommario

Alchemist è un simulatore ad eventi discreti in grado di eseguire simulazioni appartenenti a domini estremamente differenti, come simulazioni chimiche e movimento di pedoni. È frequente che venga impiegato per eseguire un grande numero di simulazioni contemporanee, che differiscono per pochi parametri di configurazione. Per questo motivo è stato considerato necessario sviluppare una parte dell'applicativo in grado di utilizzare tecniche di grid computing, in modo da distribuire questa enorme mole di lavoro. Questo documento fa una panoramica sulle attuali tecnologie per il grid computing identificando come tecnologia ideale Apache Ignite. A questo punto descrive nel dettaglio la progettazione e realizzazione vera e propria del sistema.

Introduzione

Alchemist è un simulatore ad eventi discreti in grado di eseguire simulazioni appartenenti a domini estremamente differenti, come simulazioni chimiche e movimento di pedoni. Questa grande versatilità del sistema è dovuta alla definizione di un metamodello estremamente astratto, che può poi essere implementato per rendere il sistema capace di eseguire simulazioni appartenenti ad un determinato dominio. Queste implementazioni sono definite all'interno del simulatore "incarnazioni" e l'utilizzo di una di esse viene esplicitamente dichiarato al momento della definizione di una simulazione. Questa definizione viene eseguita praticamente mediante dei file di configurazione.

Caratteristica peculiare della creazione di simulazioni è la possibilità di dichiarare all'interno di questi file delle variabili alle quali non corrisponde un valore univoco. È infatti possibile assegnare ad una o più variabili un set di valori possibili, per andare effettivamente a creare un insieme di simulazioni, discriminate dal valore delle stesse.

Il numero di simulazioni creabile a partire da una singola configurazione può risultare estremamente elevato; nel caso in cui si abbia più di una variabile multivalore, infatti, si ottiene un insieme di simulazione di grandezza pari alla cardinalità del prodotto cartesiano dei possibili valori che esse possono assumere.

Al momento l'esecuzione di queste simulazioni è esclusivamente locale e questo può comportare tempi di esecuzione estremamente lunghi, vista la grande durata che una singola simulazione può raggiungere. Per questo motivo è sopraggiunta la necessità di trovare una soluzione tecnologica in grado di accorciare i tempi di esecuzione di esperimenti dall'elevata complessità.

La soluzione a questo problema prestazionale è stata identificata nello sviluppo di una parte di grid computing per il simulatore, che permetta di rendere determinati calcolatori dei nodi pronti alla computazione, ai quali degli utilizzatori sono in grado di accedere per sfruttare le loro risorse nell'esecuzione dei set di simulazioni.

Il grid computing è una tecnologia, appartenente alla più grande famiglia della programmazione distribuita, specializzata nella coordinazione di risorse computazionali dislocate nello spazio e appartenenti a domini amministrativi diversi. Grazie alla coordinazione di queste risorse è possibile risolvere in maniera più rapida problemi dall'elevata complessità, andando a sfruttare una potenza computazionale difficile da

trovare su di un unico calcolatore. Questa tecnologia presenta però un insieme di criticità relative all'identificazione dei diversi nodi ed alla loro gestione, intesa come gestione del carico computazionale che ognuno di essi deve gestire per ottenere una computazione il più rapida possibile, e come reazione che il sistema deve avere al fallimento di uno di essi attualmente coinvolto nel calcolo. La gestione di tali problematiche è estremamente complessa ed ha portato negli anni allo sviluppo di diverse tecnologie che assistono lo sviluppo di applicazioni di grid computing.

Oggetto di questa tesi è la ricerca di una tecnologia per lo sviluppo di applicazioni grid utilizzabile nella piattaforma di sviluppo di Alchemist, ossia Java, e la relativa progettazione ed implementazione della stessa.

La parte iniziale del progetto è consistita in una lunga fase di documentazione sulle tecnologie attualmente presenti nel panorama commerciale, in modo da avere le basi per identificare quella più adatta per il soddisfacimento dei requisiti del progetto.

Successivamente è stata eseguita una modellazione del dominio in esame abbastanza approfondita e totalmente distaccata dalla tecnologia da utilizzare, in modo da poter essere di aiuto nella scelta della stessa. Inoltre una modellazione eseguita senza tener conto di una tecnologia specifica per l'implementazione ha permesso di ottenere un sistema finale nel quale un eventuale cambio tecnologico abbia il minor impatto possibile.

A questo punto è stata presa la decisione di sviluppare il sistema utilizzando il framework Apache Ignite. Questo è un framework per la creazione di sistemi data grid che offre funzionalità avanzate per lo sviluppo di applicazioni di grid computing. Le sue caratteristiche si sposano infatti perfettamente con i requisiti, come esaustivamente descritto nella sezione apposita della tesi.

Si è quindi proceduto con il design del sistema, che rifinisce il modello ed effettua le variazioni necessarie per l'utilizzo di Ignite, che sono risultate piuttosto limitate grazie al suo alto livello di astrazione.

Questa parte dell'applicativo è stata infine correttamente implementata, dando origine ad un sistema funzionante e testato. Alcune funzionalità sono però rimaste solamente modellate e non effettivamente sviluppate. La loro mancanza, dovuta all'eccessivo tempo che avrebbero richiesto per l'implementazione, fa perdere al sistema

una piccola parte della sua flessibilità, ma non preclude l'utilizzo pratico della parte di grid computing, come descritto nelle conclusioni.

La tesi presenta la seguente struttura:

- **Contesto**: Questa sezione introduttiva espone in maniera chiara e precisa i concetti alla base di questa tesi. In particolare contiene:
  - introduzione al concetto di simulazione e descrizione del simulatore Alchemist;
  - analisi del grid computing, criticità, e confronto con altri concetti di programmazione distribuita, come il cloud;
  - descrizione delle tecnologie Java più interessanti per lo sviluppo di applicazioni grid incontrate durante la fase di documentazione iniziale.
- **Contributo**: descrizione del lavoro svolto per la creazione della parte di grid computing all'interno di Alchemist. In particolare contiene:
  - descrizione particolareggiata dei requisiti;
  - modellazione del dominio;
  - motivazioni della scelta di Apache Ignite;
  - design.
- **Risultati, conclusioni e lavori futuri**: contiene le valutazioni finali sulla parte svolta, in particolare sono presenti:
  - commento ai test effettuati sull'applicazione;
  - descrizione delle parti ancora da sviluppare ed integrare nel sistema.

Indice

Capitolo 1

Contesto

1.1 Simulazione

Una simulazione è l'imitazione di un sistema o processo del mondo reale e della sua evoluzione nel tempo [4]. L'imitazione di un processo reale è definita "Modello", e consiste in una versione semplificata della realtà. Questa semplificazione è spesso indispensabile; un sistema reale può infatti avere diverse caratteristiche che rendono la sua esatta riproduzione difficoltosa ed a volte sconsigliabile, come:

- richiesta di tempi eccessivi;
- problemi nella riproduzione in ambienti controllati;
- pericolosità nella simulazione;
- richiesta di capacità tecniche oltre il disponibile.

Come si può evincere da queste definizioni una simulazione non è necessariamente qualcosa di digitale ed eseguibile su di un calcolatore, ma può essere qualunque procedimento che soddisfi i requisiti descritti in precedenza.

Per quanto concerne le simulazioni eseguite a livello digitale si può definire una classificazione che, in relazione alla loro gestione del tempo, è in grado di dividerle in due macro-gruppi: simulazioni Time-driven e simulazione DES (Discrete Event System).

Nelle prime il tempo avanza attraverso delle quantità discrete, dette tick, al termine di ognuna delle quali il modello assume un nuovo stato. Ogni evento di modifica

che il modello subisce all'interno dello stesso tick viene considerato simultaneo agli altri. Questa strategia di gestione del tempo risulta essere semplice da implementare, ma presenta una criticità non banale, ossia la definizione di un tick dalla dimensione adeguata. Prendendo dimensioni troppo piccole, infatti, si avrebbero degli stati del modello fra loro identici, in quanto nel breve lasso di tempo trascorso non avverrebbe alcun evento. Il calcolo di questi stati duplicati porterebbe ad avere uno spreco computazionale all'interno del simulatore, che si troverebbe ad effettuare calcoli sostanzialmente inutili. Al contrario assegnando al tick una quantità di tempo eccessivamente grande si andrebbero ad ottenere delle simulazioni molto meno precise, con un grande numero di eventi simultanei. La simulazione non è infatti in grado di ordinare nel tempo eventi che avvengono all'interno del medesimo tick.

Nelle simulazione DES, invece, l'avanzare del tempo della simulazione viene scandito dall'avvenire stesso degli eventi. Ognuno di essi, infatti, fa scorrere il tempo in avanti di un certo ammontare. Questo porta ad avere un insieme di eventi strettamente ordinati nel tempo in quanto nessuno di questi può essere simultaneo ad altri. Nel caso in cui due eventi siano affettivamente simultanei, la simulazione dovrà decidere quale dei due eseguire per primo.

1.1.1 Il simulatore Alchemist

Alchemist [17] è un simulatore DES che estende il modello computazionale di base delle reazioni chimiche in modo da renderlo applicabile ad un grande numero di situazioni complesse.

La natura delle applicazioni già implementate all'interno del simulatore è estremamente varia e comprende, ad esempio, simulazione di reazioni biochimiche [16] e del movimento di pedoni [5][20]. In particolare il tutto si basa su di una versione ottimizzata dell'algoritmo di Gillespie [13], chiamata Next Reaction Method [12], esteso in modo da poter lavorare con un modello mobile e dinamico, dove sia possibile aggiungere o rimuovere reazioni, dati e connessioni topologiche.

Il punto di forza del sistema è il meta-modello estremamente astratto, la cui effettiva realizzazione è demandata alle "incarnazioni", le quali rappresentano l'implementazione vera e propria delle diverse categorie di simulazioni eseguibili all'interno

del simulatore.

Il meta-modello

Come già esposto in precedenza Alchemist è stato sviluppato partendo da un'ispirazione prettamente chimica e le entità facenti parte del meta-modello hanno nomi riconducibili a tale dominio. Le entità in gioco, effettivamente implementate all'interno di ogni incarnazione, sono:

- **Molecola**: il nome di un dato;
- **Concentrazione**: il valore associato ad una molecola;
- **Nodo**: contenitore di molecole e reazioni;
- **Ambiente**: l'astrazione corrispondente allo spazio, è il contenitore dei nodi. È in grado di svolgere le seguenti funzioni:
  - restituire la posizione di un nodo
  - restituire la distanza che intercorre tra due nodi
  - in caso sia previsto, muovere i nodi
- **Regola di collegamento**: funzione relativa allo stato dell'ambiente che associa ad ogni nodo un vicinato;
- **Vicinato**: entità composta da un nodo centrale ed un set di nodi "vicini";
- **Reazione**: ogni evento che provoca cambiamento nell'ambiente. È composta da una lista di condizioni, una lista di azioni ed una distribuzione temporale. La frequenza con la quale una reazione avviene dipende da:
  - un parametro "rate" statico
  - il valore di ogni condizione
  - un'equazione che combina il "rate" con i valori delle condizioni e restituisce un "rate istantaneo"
  - la distribuzione temporale

- **Condizione**: una funzione che prendendo in ingresso lo stato corrente dell'ambiente restituisce un booleano ed un numero. Se il booleano è falso la reazione non può avvenire. Il numero di output può invece influenzare la velocità della reazione in funzione della reazione stessa e della sua distribuzione temporale;
- **Azione**: modella un cambiamento nell'ambiente.

FIGURA 1.1: Il modello di Alchemist

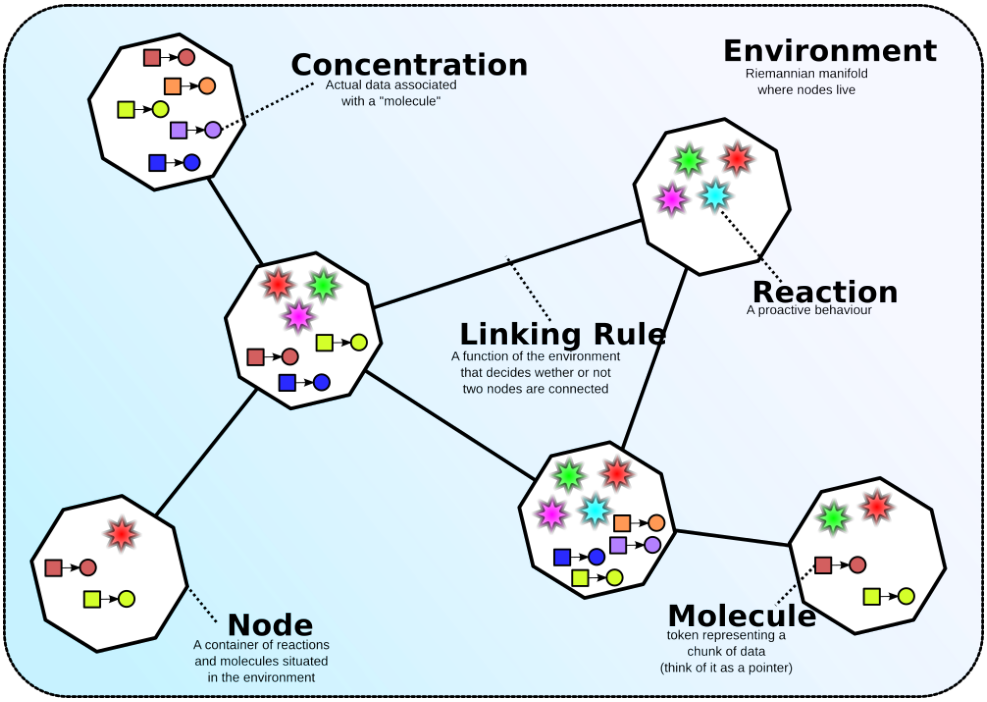
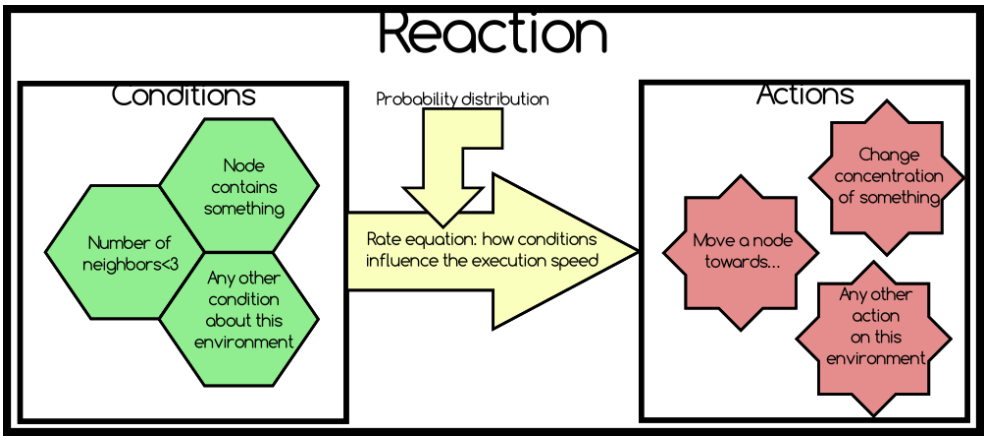


FIGURA 1.2: Schema relativo alle reazioni Alchemist



Configurazione di una simulazione

Le simulazioni da eseguire all'interno di Alchemist vengono configurate attraverso dei file YAML [6] con i quali è possibile definire l'incarnazione da utilizzare e tutte le entità che dovranno prendere parte al processo.

Caratteristica interessante per il contesto che si sta analizzando, ossia la distribuzione dell'esecuzione in un sistema grid, è la possibilità di utilizzare variabili all'interno dei file di configurazione. Queste variabili permettono di impostare determinate caratteristiche della simulazione con un set di valori invece che con un valore singolo. In questo modo da un'unica configurazione si andranno a creare una serie di simulazioni, una per ogni valore di variabile.

In caso di variabili multiple che possono assumere più di un valore, il simulatore eseguirà una simulazione per ogni possibile combinazione delle stesse, andando perciò a creare un grande numero di simulazioni, pari alla cardinalità del prodotto cartesiano dei possibili valori delle stesse.

1.2 Grid Computing

Il grid computing è una tecnica che consiste nel combinare una serie di computer, appartenenti a diversi domini amministrativi, per raggiungere uno scopo comune. Raggiunto lo scopo questi calcolatori possono rimanere disponibili oppure sparire, completamente o in parte [18].

Lo sviluppo di sistemi di questo tipo viene incontro ad applicazioni che necessitano di interagire con grandi moli di dati ed eseguire un grande numero di computazioni.

Per avere una descrizione più formale si può far riferimento alle parole di Ian Foster [10], che definisce una grid come un sistema che coordina un insieme di risorse non soggette a controllo centralizzato, utilizzando protocolli ed interfacce standard, aperte e general-purpose, per distribuire tipi di servizi non banali.

1.2.1 Terminologia

È innanzitutto importante definire alcuni termini tipici del grid computing, largamente utilizzati nelle sezioni successive, in modo da rendere l'intera analisi chiara e non

ambigua. I termini più importanti sono:

- **Task**: unità atomica che può essere programmata ed assegnata ad una risorsa;
- **Job**: insieme di task atomici che può essere assegnato ad una o più risorse;
- **Risorsa**: qualsiasi cosa che sia richiesta per portare a termine un'operazione. Esempi di risorsa sono i processori e gli hard disk. Spesso citata come "risorsa computazionale", per distinguerla dalle risorse delle applicazioni, come i file con i quali lavorano;
- **Nodo**: entità autonoma composta da una o più risorse;
- **Cluster**: insieme dei nodi che vanno a comporre la griglia, interconnessi e coordinati, e che vanno a formare un'unica entità per l'esecuzione di job;
- **Client**: nodo particolare del cluster che non partecipa all'esecuzione dei job e non viene coordinato con gli altri nodi. Sfrutta semplicemente il cluster per l'esecuzione di operazioni;
- **Topologia del cluster**: intesa come tipi di entità che lo vanno a comporre, come nodi e loro ruolo, client della griglia ed eventuali intermediari fra client e nodi stessi. Questo termine comprende anche la struttura di rete vera e propria, intesa come "chi è connesso direttamente a chi".

1.2.2 Le sfide del grid computing

Partendo dalle definizioni sopra citate si possono elencare le sfide principali che una tecnologia come quella in esame si trova a dover affrontare [22][18], e superare, per ottenere un consono grado di completezza ed usabilità:

- **Dinamicità**: le risorse in una griglia sono possedute e gestite da diversi soggetti, che possono entrare a far parte oppure abbandonare il sistema in qualunque momento;
- **Gestione**: per formare un insieme di risorse comune e sempre accessibile, sia in termini di potenza computazionale che di informazioni disponibili, è necessario

sviluppare un sistema di mantenimento comune, da coordinare con quelli locali, spesso complesso e gravoso;

- **Eterogeneità**: le risorse di una griglia sono spesso eterogenee ed è perciò necessario creare un framework in grado di gestire, e soprattutto organizzare, programmi data-intensive su larga scala in questo contesto;

- **Adattabilità**: all'interno di un sistema estremamente dinamico, come risulta essere dai punti precedenti quello in esame, il fallimento di una risorsa è assai frequente, e non deve perciò rappresentare un eccezione. Perciò è necessario sviluppare opportune strategie per sopperire a questi avvenimenti;

- **Design**: sono presenti vari problemi nel progettare software da eseguire in piattaforme di grid computing, come decomporre e distribuire l'elaborazione di elementi per poi ricostruire un risultato unico;

- **Programmazione**: la scrittura di software, vista la bassa relazione che i nodi hanno fra loro e la natura distribuita del sistema, risulta spesso più complessa.

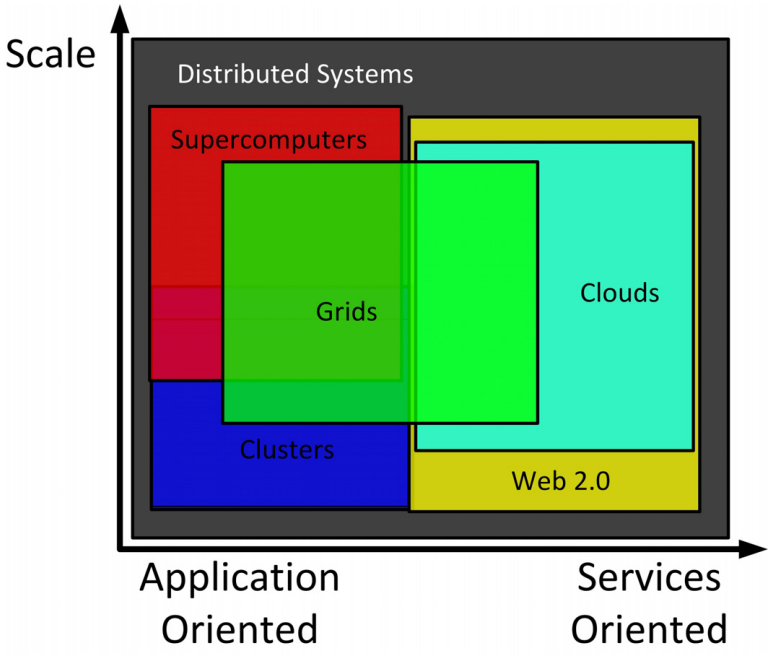
Bilanciamento del carico

Il bilanciamento del carico (Load Balancing) incarna i problemi di dinamicità, gestione ed eterogeneità sopra citati, e propone di adottare algoritmi, definiti appunto di Load Balancing, in grado di distribuire il lavoro su diversi terminali remoti in funzione delle risorse a loro disposizione [22].

Questi algoritmi sono divisi in due categorie principali: statici e dinamici. I primi decidono a quale calcolatore remoto assegnare un determinato task all'inizio della computazione, quando viene fatta una stima delle risorse necessarie per l'esecuzione dello stesso. I secondi, invece, sono in grado di migrare i task in attesa di essere eseguiti basandosi su informazioni non note a priori, ma emerse durante l'esecuzione. Questa categoria risulta particolarmente interessante in quanto, a differenza degli algoritmi statici, è in grado di gestire la dinamicità della griglia, utilizzando fin da subito eventuali nuove risorse che si sono unite.

1.2.3 Grid Computing, Cloud Computing e Programmazione Distribuita

FIGURA 1.3: Sovrapposizione fra le diverse tecnologie



Scopo di questa sezione è chiarire le differenze fra tre tecnologie simili e strettamente interconnesse come grid, cloud, e distributed programming. Questi tre concetti sono infatti per definizione simili e, per determinate caratteristiche ed applicazioni, presentano sovrapposizioni.

Un sistema distribuito [19] può essere semplicemente definito come una collezione di computer indipendenti che appaiono agli utenti come un unico sistema coerente.

Il cloud [7] è un tipo di sistema parallelo e distribuito che consiste in un insieme di computer interconnessi e virtualizzati, dinamicamente presentati come come una o più risorse unificate basate su di un accordo a livello di servizio. Il cloud è un modello per fornire un accesso di rete onnipresente, conveniente ed on-demand ad un insieme condiviso di risorse computazionali. Queste possono essere rapidamente fornite e poi rilasciate con un minimo sforzo di gestione. Il cloud fornisce tre tipi principali di servizio:

- **Software as a Service**: tipo di servizio dove gli utenti possono utilizzare un software ospitato dal servizio stesso, pagando solo per il tempo di utilizzo;

- **Platform as a Service**: fornisce un ambiente integrato di alto livello per design, realizzazione, test, distribuzione ed aggiornamento di applicazioni online;
- **Infrastructure as a Service**: fornisce agli utenti potenza computazionale, spazio d'archiviazione, connettività ed altre risorse computazionali per eseguire qualunque tipo di software, compresi sistemi operativi.

Da queste semplici definizioni è semplice notare come sia il grid che il cloud siano sistemi distribuiti, ma fra loro leggermente differenti [11]. Condividono il medesimo obbiettivo: ridurre il costo di calcolo aumentando flessibilità ed affidabilità, trasformando i computer, nati per lavorare in modo isolato, in qualcosa di coordinato ed accessibile da soggetti terzi. Risulta però differente la filosofia con cui lo fanno.

Un sistema grid è un sistema orientato all'applicazione che permette di interconnettere un insieme di risorse a propria disposizione per andare a creare una piattaforma in grado di eseguire job, appunto relativi all'applicazione, decomposti in task indipendenti. Quando un lavoro è in esecuzione alloca per se un sottoinsieme di queste risorse, attendendo il risultato, e le rilascia a computazione eseguita.

Un sistema cloud è invece orientato principalmente al servizio, che viene fornito ad una moltitudine di utenti. Gli utenti condividono perciò un medesimo insieme di risorse, volte all'esecuzione di una tipologia di lavoro unica. In caso vi sia un insieme di utenti tale da saturare le risorse disponibili, queste vengono automaticamente fatte scalare ed aumentate, in modo non trasparente agli utenti finali. Questa caratteristica fornisce un livello maggiore di dinamicità al cloud, e lo rende adatto per problematiche di dimensione nettamente maggiore e variabile rispetto al grid.

1.2.4 Data Grid

I sistemi data grid [8] nascono per gestire la combinazione di grandi moli di dati, utenti e risorse geograficamente distribuiti, e necessità di eseguire operazione computazionalmente complesse.

Per risolvere queste problematiche è stata creata una vera e propria estensione dei sistemi grid tradizionali, che offre funzioni specifiche per la memorizzazione e l'elaborazione di grandi moli di dati.

Questo nuovo tipo di tecnologie si basano su quattro principi fondamentali:

- **Mechanism neutrality**: indipendenza totale dalla tecnologia di basso livello utilizzata per la memorizzazione di dati e meta-dati, e per il loro movimento;
- **Policy neutrality**: le scelte di design con un forte impatto sulle performance devono essere esposte il più possibile all'utente e non essere incapsulate in delle "scatole nere". È quindi normale che la tecnologia utilizzata per spostare dati da un nodo all'altro non sia visibile, ma dev'essere decidibile la politica di replicazione da adottare, cioè in quali e quanti nodi avere i medesimi dati;
- **Compatibility with Grid infrastructure**: totale supporto a servizi di basso livello tipici del grid computing, come la gestione centralizzata delle risorse;
- **Uniformity of information infrastructure**: uniformità nella gestione dei dati e metadati con quanto garantito da un sistema grid per quel che riguarda le risorse. Questo significa adottare un sistema di memorizzazione che gestisca a pieno l'adattamento del sistema con le condizioni che variano a tempo di esecuzione.

Partendo da questi principi è possibile sviluppare un sistema completo ed affidabile, in grado di fornire una serie di servizi quali:

- **Storage systems and data access**: ogni nodo della griglia deve vedere l'insieme dei dati accessibili come unitario, distaccandosi totalmente dall'effettiva posizione e natura degli stessi, seguendo il principio di "Mechanism neutrality" sopra descritto. Il sistema deve fornire una serie di API per creare, distruggere, leggere e manipolare i dati astraendoli come file, intesi come unità atomiche di dati;
- **Metadata service**: il sistema deve consentire l'accesso ad informazioni relative ai file presenti in griglia, senza che sia necessario leggerli effettivamente. Metadati tipici sono nomi, dimensioni e contenuto dei file stessi.

1.2.5 Stato dell'arte

La ricerca in questo campo è stata, nell'ultimo decennio, estremamente attiva. Questo è dovuto al netto aumento delle necessità computazionali per l'esecuzione delle più svariate applicazione, come ad esempio l'elaborazione di grandi moli di dati. Negli

ultimi anni abbiamo assistito inoltre all'esplosione delle applicazioni cloud che, come abbiamo visto in precedenza, a volte sono parzialmente sovrapposte alla tecnologia grid.

In questa sezione si elencano alcuni progetti internazionali recenti, che utilizzano l'attuale stato dell'arte relativo alle tecnologie di grid computing [18]:

- **EGI-InSPIRE**[1]: progetto (Integrated Sustainable Pan- European Infrastructure for Researchers in Europe) iniziato nel 2010, finanziato dalla Commissione Europea, che ha come obbiettivo la creazione di un'infrastruttura grid comunitaria per portare beneficio alla ricerca in Europa.

- **MammoGrid** [21]: architettura service-oriented basata su di un'applicazione grid medica. Ha l'obbiettivo di aiutare medici radiologi nella ricerca sul cancro, analizzando immagini di mammografie.

- **DDGrid** [23]: (Drug Discovery Grid) progetto che ha l'obbiettivo di creare una piattaforma collaborativa per la scoperta di nuove droghe utilizzando tecnologie grid e p2p.

1.2.6 Tool Esistenti

Nel panorama Java, piattaforma di sviluppo di Alchemist, sono presenti vari framework con i quali è possibile progettare e sviluppare sistemi di grid computing. Alcuni di questi sono generici tool per la programmazione distribuita, mentre altri sono più specifici e dedicati alla distribuzione di semplici computazioni all'interno di un cluster di calcolatori. Di seguito vengono descritti i principali tool incontrati durante la fase di ricerca e documentazione per lo sviluppo del progetto.

Apache Storm

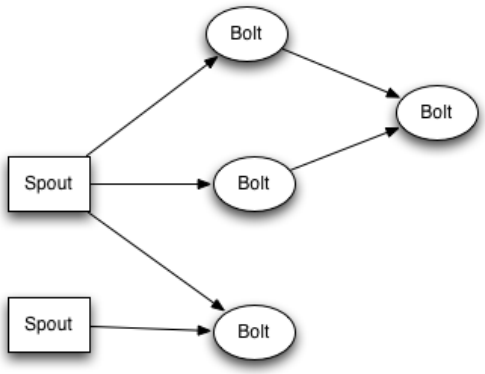
Apache Storm[2] è un framework di programmazione distribuita utilizzato principalmente per l'elaborazione in tempo reale di grandi stream di dati.

Il principio di funzionamento del sistema si basa su due entità principali, che partecipano alla formazione topologica del cluster: Spouts e Bolts. Gli Spouts sono

[1] *EGI-InSPIRE official website*. `https://wiki.egi.eu/wiki/EGI-InSPIRE:Main_Page`.
[2] *Apache Storm official website*. `http://storm.apache.org/index.html`.

responsabili della generazione di tuple che vengono inviate ai Bolts, vera entità di elaborazione. Questi eseguono operazioni sulle tuple, che vengono poi indirizzate verso nuovi elaboratori, oppure mandate all'unità demandata al salvataggio od utilizzo del risultato.

FIGURA 1.4: Topologia del cluster di Storm



Akka

Akka[3] è un framework di programmazione distribuita ad attori. Il modello ad attori, teorizzato per la prima volta nel 1973 [14], è un modello intrinsecamente concorrente e si basa sul semplice concetto secondo il quale "tutto è un attore". Un attore è un'entità autonoma che opera concorrentemente agli altri attori del sistema in modo del tutto asincrono.

Il modello si fonda su alcuni semplici paradigmi:

- Gli attori comunicano fra loro unicamente mediante lo scambio di messaggi asincroni;
- Ogni attore ha uno stato interno privato che può variare in seguito alla ricezione di un messaggio;
- Ogni attore può reagire ad un messaggio creando attori figli, terminandoli oppure inviando a sua volta altri messaggi.

[3] *Akka official website*. `https://akka.io`.

Akka implementa questo modello offrendo funzionalità relative a:

- Sistema di supervisione e monitoraggio di attori;
- Possibilità di monitoraggio continuo degli eventi relativi al cluster, come unione di nuovi attori, loro abbandono oppure loro terminazione improvvisa;
- Strategie per la distribuzione ed inoltro di messaggi;
- Strategie per l'identificazione di cluster attualmente attivi ed unione ad essi.

JPPF

JPPF[4] è un framework Java per il grid computing, che permette di eseguire applicazioni su di un grande numero di macchine in maniera semplice. Gli unici requisiti da soddisfare per poter sfruttare la distribuzione offerta sono:

- Dividere l'applicazione in parti indipendenti eseguibili in parallelo;
- Eseguire l'applicazione all'interno di una griglia JPPF.

Il framework si basa su di una topologia di rete costituita da tre soggetti:

- Client: applicazione che si unisce al cluster per eseguire un job, costituito da semplici task indipendenti;
- Server: riceve il job dal client e distribuisce i vari task ai nodi liberi. È responsabile dell'applicazione degli algoritmi di load balancing;
- Node: esegue i singoli task e ne restituisce il risultato.

Il sistema offre inoltre un insieme di funzionalità integrate che lo rendono molto interessante e comodo da utilizzare. Una di queste è il trasferimento delle classi e delle risorse richieste dai task in modo automatico; i nodi infatti sono in grado di richiedere automaticamente al client che ha lanciato il job eventuali risorse che non sono in grado di caricare con il class loader locale.

A livello meramente realizzativo costruire un task all'interno di Java è banale, basta infatti implementare la relativa interfaccia, ovviamente serializzabile per consentirne la distribuzione in remoto. All'interno di questo task, oltre al metodo da

[4] *JPPF official website*. `https://ignite.apache.org/`.

FIGURA 1.5: Topologia del cluster JPPF

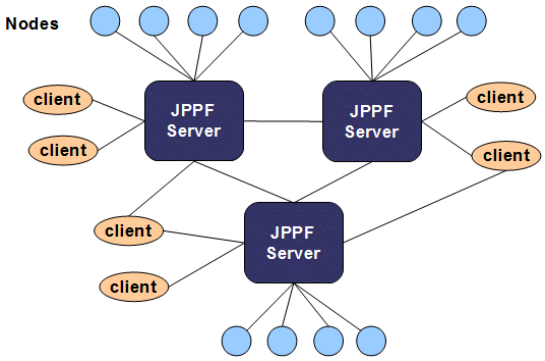
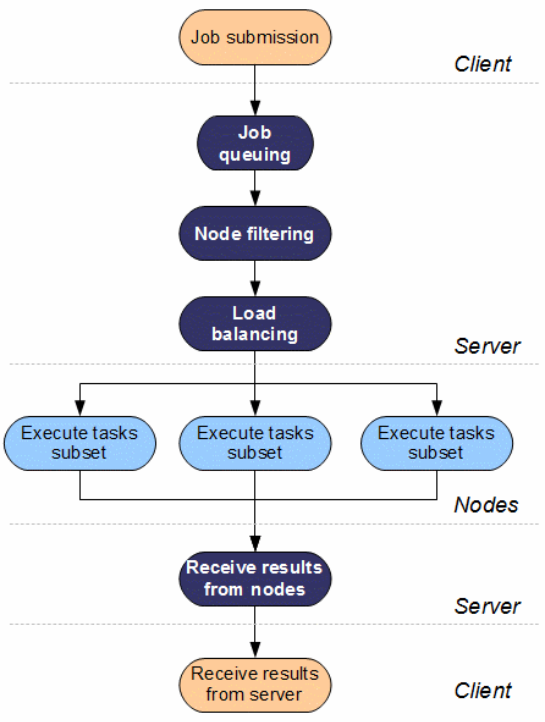


FIGURA 1.6: Distribuzione ed esecuzione di un job JPPF



eseguire, è possibile aggiungere informazioni direttamente come campi dell'oggetto. Questa metodologia di recapito delle informazioni risulta essere la più rapida, ma è possibile solo per dimensioni degli oggetti limitate, pari a pochi Mb.

Il recapito del risultato di una simulazione è automatico e consiste in una collezione di oggetti a discrezione dell'utilizzatore, uno per ogni task completato.

I meccanismi di fault tollerance e load balancing sono inoltre forniti già pronti per l'uso, con una riallocazione automatica dei task al momento del fallimento di un nodo e con l'implementazione di un buon numero di algoritmi differenti per la distribuzione efficiente del lavoro.

Apache Ignite

Apache Ignite[5] è un framework di computazione in-memory che permette di sviluppare applicazioni in contesti di data-grid.

Oltre alle caratteristiche rivolte alla gestione dei dati offre funzionalità estremamente avanzate per la creazione di applicazioni di grid computing. Per quel che riguarda la creazione dei job, e la distribuzione dei relativi task, il funzionamento è sostanzialmente identico a quanto offerto dal framework precedente. È sufficiente creare un insieme di task indipendenti fra loro, e che non presentino problemi di concorrenza, per permettere al sistema di distribuirli fra i calcolatori disponibili.

Anche la realizzazione effettiva dei task all'interno di Java è praticamente identica, con le stesse identiche valutazioni fatte in precedenza sui campi dell'oggetto utilizzati per il trasferimento delle informazioni e per la natura del risultato.

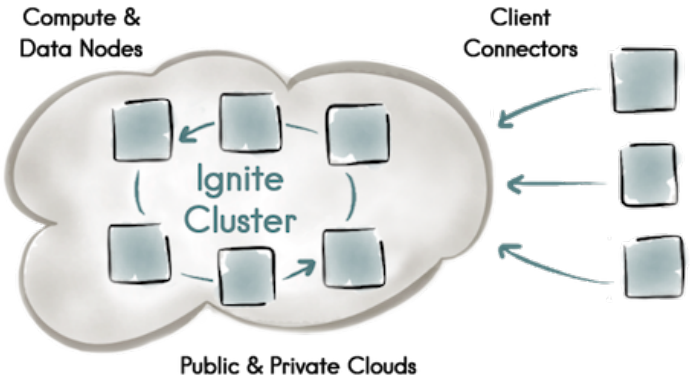
Se confrontato con JPPF l'organizzazione topografica dei nodi è differente e notevolmente semplificata. Ogni nodo si unisce infatti direttamente al cluster senza bisogno di passare attraverso un'entità server. Ogni nodo ha un pool di thread per l'esecuzione di task opportunamente configurabile, questo significa che un singolo nodo esegue contemporaneamente un numero di task pari alla dimensione di questo pool. I nodi sono inoltre semanticamente tutti uguali e differiscono esclusivamente per il loro stato interno, composto da:

- Ruolo: client o server;
- Attributi: coppie chiave valore definite all'avvio del nodo stesso;
- Metriche prestazionali: relative a caratteristiche come memoria RAM a disposizione e utilizzo attuale della cpu. Queste informazioni sono ovviamente dinamiche e cambiano a seconda del carico che il nodo sta attualmente gestendo.

Questo stato interno può essere acceduto da qualunque terminale del cluster; questo permette di fare processi di preselezione sui computer che dovranno andare ad eseguire il job. In particolare Ignite, partendo dall'entità che astrae il cluster, è in grado di creare dei gruppi dinamici in base a dei predicati riguardanti lo stato dei singoli nodi. Questa caratteristica è molto importante quando ci si trova ad avere griglie contenenti

[5] *Apache Ignite official website*. `http://www.jppf.org/`.

FIGURA 1.7: Organizzazione del cluster di Ignite



computer estremamente eterogenei a livello prestazionale, con alcuni di essi che non sarebbero in grado di eseguire certe operazioni, e soddisfa inoltre il problema della gestione della dinamicità. In caso un nodo si trovi a soddisfare tutti i predicati di un gruppo già esistente, a causa di un cambio delle sue metriche, oppure per una sua unione successiva alla griglia, il framework sarà infatti in grado di rilevarlo ed unirlo automaticamente.

La tolleranza ai guasti è gestibile con due strategie possibili già implementate, ossia con il fallimento del job in caso di caduta di un nodo, oppure con il tentativo di riallocazione dei task che erano in esecuzione su esso.

Per il load balancing sono invece disponibili diversi algoritmi già implementati, fra i quali sono degni di nota:

- **Random and Weighted Load Balancing**: i task vengono allocati prendendo nodi random all'interno del cluster o del gruppo, assegnandone un numero maggiore a quelli a cui è stato assegnato, al momento dell'avvio, un peso maggiore;

- **Job Stealing**: i task vengono divisi equamente fra i nodi del cluster o gruppo per poi essere riallocati dinamicamente nel caso uno o più nodi risultino sotto utilizzati. Questo significa che i nodi più veloci si troveranno ad eseguire più task, "rubandoli" ai nodi più lenti.

Per quanto riguarda la costruzione effettiva del cluster, il framework integra un grande numero di metodologie di rilevamento automatico di nodi già attivi, ed offre

anche un sistema per il passaggio p2p delle classi non presenti sui nodi, ma necessarie per l'esecuzione dei task e disponibili a chi ha lanciato il job. Questa caratteristica è leggermente meno avanzata rispetto al framework descritto in precedenza e non include l'invio in remoto di risorse differenti dalle semplici classi compilate.

Infine è bene notare come le caratteristiche derivanti dalla natura data grid del sistema permettano di accedere ad un grande numero di funzioni estremamente utili per condividere dati fra i nodi, come cache chiave-valore estremamente veloci e strutture dati condivise quali code e contatori.

Capitolo 2

Contributo

2.1 Requisiti

In questa sezione si vanno ad analizzare i requisiti funzionali e non del progetto in esame, ossia della distribuzione di diverse simulazioni Alchemist in un sistema di grid computing.

2.1.1 Requisiti funzionali

A livello funzionale il sistema dovrà essere in grado di eseguire in maniera distribuita le simulazioni multiple che vengono già eseguite in modalità "batch". In caso Alchemist venga eseguito in questa modalità con un file di configurazione contenente variabili multi valore, infatti, vengono già create, come descritto nella Sezione 1.1.1, una moltitudine di simulazioni. Queste, attualmente eseguite tutte in locale da un pool di thread, devono poter essere distribuite, passando al simulatore un nuovo parametro da riga di comando con le relative configurazioni necessarie. Le diverse simulazioni devono essere quindi eseguite in remoto in quanto tali e non deve essere fatto alcun lavoro di divisione dei processi che portano all'esecuzione di ognuna di esse.

È inoltre da notare come la modalità "batch" venga spesso utilizzata totalmente sprovvista di interfaccia grafica ed offra unicamente un log su console. È infatti possibile agganciarvi un'interfaccia grafica opzionale, ma questo non è richiesto se la computazione viene distribuita. L'output delle simulazioni è quindi notevolmente semplificato e consiste in un file contenente una serie di dati esportati. Questi dati, esplicitamente configurati, sono campionati con un certo intervallo.

L'insieme di simulazioni è caratterizzata quindi dallo stesso file di configurazione YAML, contenente le informazioni generali per la creazione, ma ognuna di esse sarà discriminata dalla altre da un diverso insieme di variabili di inizializzazione. All'interno del file possono essere inoltre definite dipendenze ad altri file esterni, come dati gps e definizione di funzioni. Le dipendenze spesso non sono dichiarate esplicitamente col percorso del file, ma questa caratteristica può essere opportunamente modificata per il funzionamento del sistema. Sostanzialmente, se un utente intende distribuire il lavoro, dovrà aggiungere alle configurazioni una sezione dove indica esplicitamente il percorso dei file di dipendenza.

Altro requisito non banale è la semplicità e flessibilità delle configurazioni per la creazione effettiva della griglia stessa. L'obbiettivo è infatti sfruttare per il calcolo computer di laboratorio, o comunque computer "di fortuna" connessi alla stessa rete. È perciò ovvio come la mancanza di un cluster stabile, e topologicamente ben definito a priori, porti la necessità di avere un avvio dei nodi ed un rilevamento di un eventuale sistema già in esecuzione i più rapidi e semplici possibile. In particolare è stato richiesto che la distribuzione base di Alchemist abbia un'opzione da riga di comando per l'esecuzione in modalità nodo, che accetti tutti i parametri di configurazione necessari.

2.1.2 Requisiti non funzionali

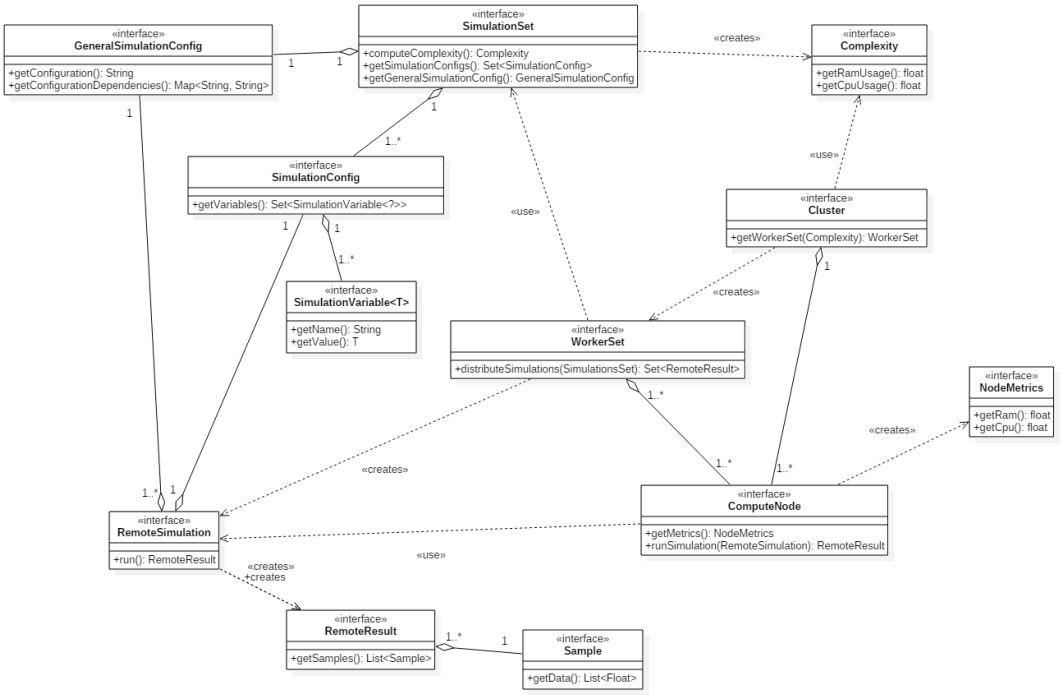
È importante notare come la natura del cluster sopra descritta possa portare con facilità ad una notevole eterogeneità delle macchine in gioco e ad una frequente variazione del loro numero. Le criticità, proprie del grid computing e descritte nella Sezione 1.2.2, relative a dinamicità, eterogeneità e adattabilità sono quindi estremamente importanti nel progetto ed andranno gestite con un occhio di riguardo. Le simulazioni Alchemist inoltre risultano spesso parecchio complesse e voraci di risorse. Per questo motivo ignorare l'eterogeneità delle macchine potrebbe portare ad un vero e proprio fallimento degli esperimenti, ad esempio per una richiesta di memoria eccessiva alla Java Virtual Machine.

2.2 Modello del dominio

Partendo dai requisiti precedentemente esposti è stato possibile estrapolare un modello generale del dominio, composto dalle sole interfacce delle entità in gioco e dai principali servizi che esse dovranno offrire.

È qui riportato uno schema contenente la modellazione dell'intero sistema, che verrà descritto per aree funzionali nelle sezioni successive.

FIGURA 2.1: UML del modello

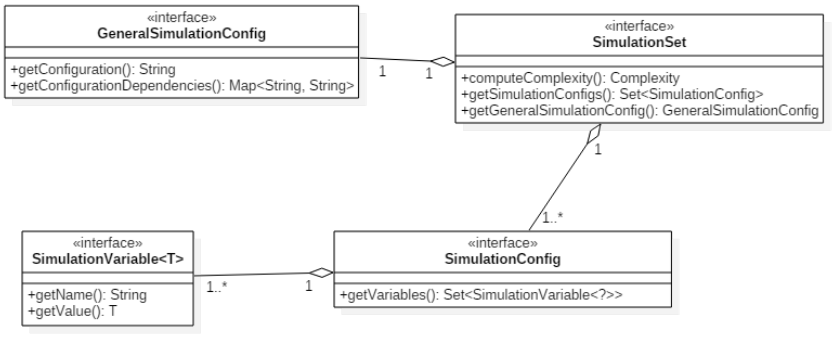


L'intera modellazione è stata ovviamente eseguita mantenendo totale distacco dal framework o tecnologia che sarebbe poi stata utilizzata per la progettazione ed implementazione effettiva.

2.2.1 Insieme di simulazioni e loro configurazione

La prima entità identificata è stata quella relativa all'insieme di simulazioni da eseguire, mappata nel `SimulationSet`. Questa dovrà contenere al suo interno tutti i parametri necessari per la creazione delle diverse simulazioni, ed ho perciò modellato due entità alle quali delegare questo compito: il `GeneralSimulationConfig` e

FIGURA 2.2: UML della modellazione relativa alle configurazioni delle simulazioni

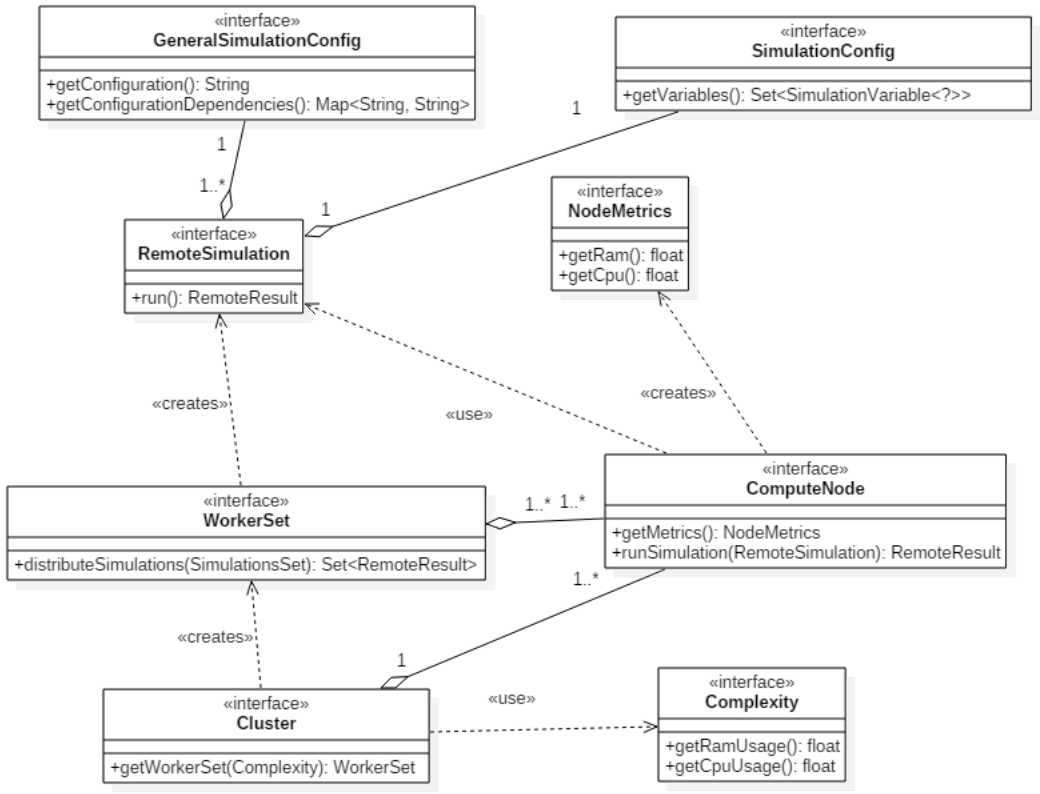


il `SimulationConfig`. Il primo dovrà essere in grado di fornire le informazioni generali di configurazione, comuni a tutte le simulazioni, ossia il file di configurazione principale e i relativi file di dipendenza elencati al suo interno. Il secondo dovrà invece contenere l'insieme di configurazioni che distinguono fra loro le diverse simulazioni, ossia l'insieme di variabili per l'inizializzazione. Come si può vedere dallo schema il `SimulationSet` conterrà infatti un unico `GeneralSimulationConfig` e molteplici `SimulationConfig`, uno per ogni simulazione da eseguire. L'unico servizio degno di nota offerto dal `SimulationSet` è quello di fornire una stima della complessità di una simulazione, corrispondente all'entità `Complexity`, fondamentale per distribuire le simulazioni alle sole macchine del cluster che hanno risorse sufficienti per portarle a termine. Come già discusso nella Sezione 2.1.1 l'eterogeneità del cluster nel quale il sistema si troverà a lavorare risulta essere un fattore di grande importanza, perciò una stima di complessità delle simulazioni è fondamentale, indipendentemente dalla tecnologia che verrà poi utilizzata per la distribuzione.

2.2.2 Cluster, nodi e simulazioni remote

A questo punto è stato necessario definire il cuore del sistema, ossia il cluster di macchine alle quali viene delegata la computazione. L'entità Cluster contiene al suo interno i vari nodi remoti ed è in grado di ricavarne un sottoinsieme filtrandoli in base ad una `Complexity`. Per rendere questo possibile i vari nodi possono fornire al Cluster, come si vede dallo schema, le loro metriche prestazionali. Questo rappresenta un'iniziale strategia di load balancing, che consiste nel selezionare i soli nodi in grado di

FIGURA 2.3: UML relativo alla modellazione di cluster, nodi e simulazioni

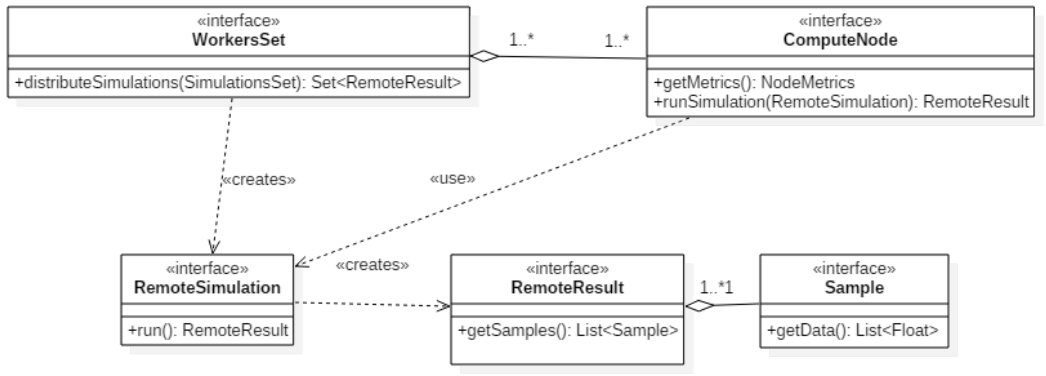


eseguire il lavoro.

Il `WorkerSet`, composto quindi anch'esso da nodi, è l'entità che si occupa direttamente di distribuire in remoto creazione ed esecuzione delle simulazioni. Partendo da un `SimulationSet`, infatti, si occupa di creare le simulazioni remote, contenenti tutti i parametri di configurazione corrispondenti, e di delegarne l'esecuzione ai nodi che contiene. È da notare come ogni simulazione contenga il `GeneralSimulationConfig`, che contiene parametri fondamentali per la configurazione di ognuna di esse, e un unico `SimulationConfig`, che contiene le variabili che discriminano l'esecuzione della singola simulazione.

Questa entità idealmente si dovrà occupare anche di mettere in atto strategie di load balancing, evitando di sovraccaricare i nodi prestazionalmente più deboli del gruppo.

FIGURA 2.4: UML relativo al risultato delle simulazioni



2.2.3 Risultato delle Simulazioni

Le ultime entità identificate, piuttosto banali, sono quelle relative al risultato delle diverse simulazioni remote. Il risultato di una simulazione è infatti un insieme di campionamenti (`Sample`) che contengono un insieme di metriche, proprio come descritto dai requisiti di progetto.

2.3 Scelta della tecnologia da utilizzare

Sono state eseguite varie ricerche che hanno portato ad analizzare alcuni framework, già introdotti nella Sezione 1.2.6. In questa sezione si andranno ad analizzare le caratteristiche di questi in relazione ai requisiti del progetto e si andranno a discutere le motivazioni che mi hanno portato alla scelta finale della tecnologia da utilizzare.

Apache Storm

L'organizzazione del sistema offerta, per quanto presenti ottime funzionalità di fault tollerance e load balancing presenti di default, si è rivelata davvero poco adatta al problema in esame, che prevede di eseguire sui dati lunghe simulazioni non scomponibili in step più piccoli, e non modifiche progressive.

Inoltre il load balancing di storm, essendo appunto nato per la distribuzione di piccole computazioni progressive, non tiene conto dell'eterogeneità delle risorse presenti nel cluster. Ogni Bolt, infatti, può essere eseguito indifferentemente su ogni calcolatore della griglia.

Akka

Akka, nonostante sia solitamente utilizzato per la risoluzione di problemi differenti da quello in esame, avrebbe permesso di sviluppare un sistema funzionale ad Alchemist. Il suo utilizzo avrebbe però presentato alcune criticità non banali:

- Necessità di sviluppare strategie per il load balancing basate sul semplice monitoraggio degli attori;
- Necessità di sviluppare tecniche di reazione al fallimento degli attori, che viene semplicemente notificato come evento del cluster, ma la cui gestione viene demandata al progettista.

JPPF

JPPF risulta essere un framework adatto al problema in esame. Punti a suo favore particolarmente interessanti sono:

- **Sistema di creazione dei task**: la creazione di task indipendenti non presenta alcun tipo di problema nel caso in esame. Le diverse simulazioni godono infatti di totale autonomia, ed è sufficiente far arrivare in remoto le configurazioni necessarie per riuscire ad eseguirle
- **Fault tollerance**: la riallocazione dei task assegnati precedentemente a nodi caduti è idonea all'applicazione da sviluppare;
- **Load balancing**: vari algoritmi di load balancing vengono forniti;
- **Resource loading**: il caricamento automatico delle risorse in p2p risulta estremamente comodo.

L'ultimo punto dell'elenco, per quanto positivo, presenta però diverse criticità. Il class loading automatico può infatti risultare problematico in un contesto come quello di Alchemist, che fa larghissimo uso della reflection. Anche se questa funzione non dovesse presentare problemi con la reflection, non bisognerebbe inoltre ignorare il fatto che uno scambio p2p su richiesta risulta essere sovente lento. Questo problema

prestazionale non è ignorabile nel sistema in esame, che a causa delle sue dimensioni utilizza un grande numero di classi durante l'esecuzione di una singola simulazione.

Questo problema è risolvibile includendo direttamente le classi richieste in server o nodi, evitando così le richieste al client. Questo non è però possibile con risorse diverse dalle classi, rappresentate in questo caso dai file di configurazione. Per ottimizzare il loro trasferimento sarebbe infatti necessario configurare all'interno di server e nodi meccanismi di caching dei file.

Una soluzione del genere risulterebbe conveniente in un sistema dove le dipendenze sono estremamente numerose ed identificarle singolarmente estremamente difficoltoso. Nelle simulazioni di Alchemist questo però non si verifica, e già nella definizione dei requisiti è stata espressa la possibilità di indicarli esplicitamente al momento della configurazione.

Altra caratteristica non ottimale di JPPF è l'organizzazione topologica del cluster, che presenta server e nodi che vanno propriamente configurati in un sistema di dipendenze. Questa organizzazione non è eccessivamente complessa, ma può sicuramente risultare macchinosa rispetto ad altre alternative, come l'organizzazione di Apache Ignite.

Apache Ignite

Apache Ignite è il framework sul quale è ricaduta la scelta finale per l'implementazione del sistema di grid computing. Esso infatti offre una serie di caratteristiche estremamente utili:

- **Creazione dei task**: la creazione dei task risulta estremamente comoda e sostanzialmente identica a quella descritta per JPPF;
- **Topologia del cluster**: il fatto che ogni macchina facente parte del cluster sia un semplice nodo che è in grado di unirsi direttamente al sistema, senza eventuali intermediari, soddisfa in pieno la caratteristica di semplicità ed immediatezza espressa nei requisiti;

- **Cache ottimizzate**: la natura rivolta all'elaborazione dei dati tipica di Ignite mette a sua disposizione delle cache estremamente veloci, ottime per trasmettere ai nodi i file di configurazione. Questi risultano infatti essere tipicamente pochi e di dimensioni non eccessive, ma comunque troppo pesanti per essere inseriti direttamente come campo del task;

- **Gruppi dinamici**: i gruppi creabili all'interno di Ignite permettono di filtrare i nodi in maniera molto più comoda ed esplicita rispetto alle tecniche di load balancing offerte da JPPF. Questa caratteristica sarà ottimamente sfruttata in fase di implementazione, come descritto in Sezione 2.4.3;

- **Load balancing**: gli algoritmi di load balancing offerti risultano adeguati per la gestione del carico di lavoro derivante dalla creazione delle simulazioni;

- **Fault tollerance**: la riallocazione automatica dei task precedentemente assegnati ai nodi caduti soddisfa pienamente i requisiti;

- **Scambio p2p delle classi**: questa caratteristica eredita i problemi discussi per JPPF.

2.4 Design

In questa sezione viene mostrata la realizzazione vera e propria del modello definito nella Sezione 2.2 e come questo sia evoluto alla luce dell'utilizzo di Apache Ignite e dei problemi riscontrati.

L'entry-point all'interno di Alchemist è la classe `AlchemistRunner`, dove viene effettivamente creato l'insieme di variabili necessario per l'inizializzazione delle diverse simulazioni. In questo punto è inoltre possibile ottenere tutte le informazioni relative ai file di dipendenza e alla configurazione generale della simulazione, gestite con le modalità descritte nella Sezione 2.4.2.

Dopo aver recuperato tali informazioni si procede con l'unione al cluster di Ignite (Sezione 2.4.3), per poi selezionare i nodi per l'esecuzione (Sezione 2.4.3) e lanciare effettivamente le simulazioni remote (Sezione 2.4.4).

Il sistema attende infine il risultato delle simulazioni e lo salva in locale (Sezione 2.4.5).

2.4.1 Insieme di simulazioni

FIGURA 2.5: UML dell'insieme di simulazioni e relative configurazioni

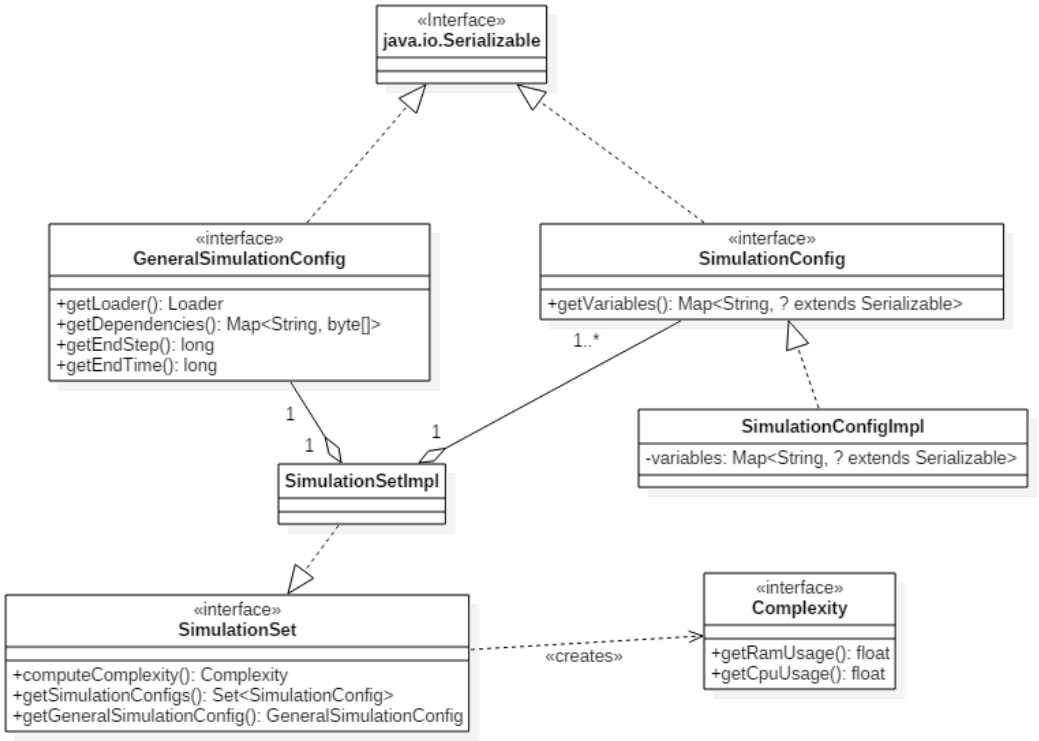
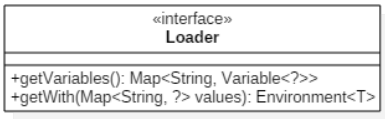


FIGURA 2.6: UML parziale del Loader di Alchemist



L'insieme di simulazioni, descritto dall'UML in Figura 2.5, ha subito cambiamenti piuttosto limitati. Il `GeneralSimulationConfig` ha visto la nascita di metodi relativi al tempo e allo step di fine simulazione, non considerati durante la modellazione precedente, ma indispensabili per la corretta esecuzione in remoto. Inoltre ora fornisce

direttamente il `Loader` di Alchemist, utilizzato per la creazione di tutte le simulazioni, al posto del contenuto del file di configurazione. La descrizione dettagliata delle realizzazioni dell'interfaccia è demandata alla sezione successiva.

Il `SimulationConfig` ha invece visto cambiare la logica di memorizzazione delle variabili di inizializzazione. Il set descritto nel modello è stato abbandonato in favore di una mappa, che contiene il nome della variabile come chiave ed il valore della variabile come valore.

Come si può vedere il valore possibile della variabile corrisponde a qualunque oggetto serializzabile, caratteristica fondamentale per la distribuzione in rete dello stesso. Questa caratteristica non era inizialmente presente all'interno di Alchemist, ma è stato possibile inserirla senza difficoltà, vista la serializzabilità nativa di tutti i tipi di variabili utilizzati. La scelta di questa nuova organizzazione delle variabili è stata presa a causa del suo già largo uso all'interno del simulatore, come si può vedere dal `Loader` di Figura 2.6. Non avrebbe infatti avuto alcun senso scegliere strutture dati differenti per poi convertirle al momento dell'utilizzo effettivo.

Infine entrambi gli oggetti a cui è demandata la memorizzazione delle informazioni sulle simulazioni implementano ora l'interfaccia `Serializable`, in modo da poter essere distribuiti liberamente in rete.

2.4.2 Configurazione locale e remota

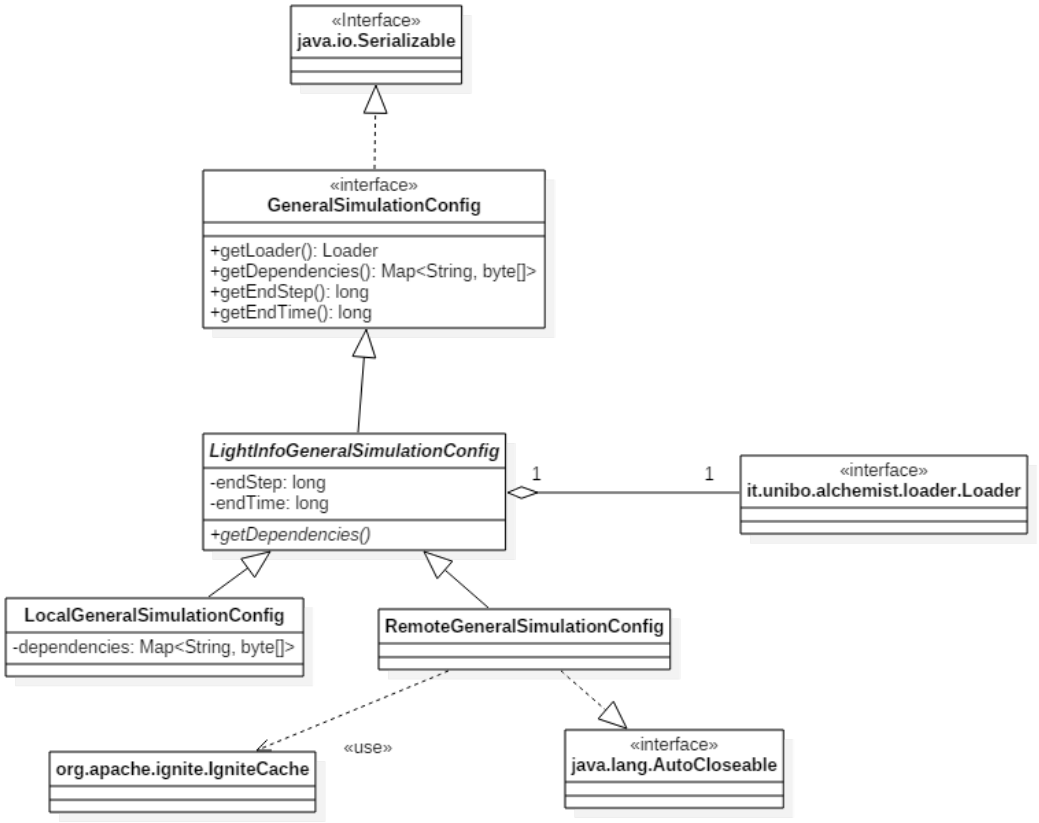
Le diverse realizzazioni del `GeneralSimulationConfig`(Figura 2.7) hanno dato origine ad una gerarchia ereditaria fortemente legata al meccanismo di distribuzione delle informazioni progettato con Ignite.

Il `LightInfoGeneralSimulationConfig` è una classe astratta contenente solamente le informazioni che, in termini di occupazione di memoria, risultano meno pesanti. Questo esclude da esse i file di dipendenza, che possono occupare anche diversi Mb.

Da questa classe ereditano due diverse implementazioni complete:

- **LocalGeneralSimulationConfig**: classe che memorizza semplicemente il contenuto dei file di dipendenza come oggetti locali, andando direttamente a leggerli sul file system;

FIGURA 2.7: UML relativo alle configurazioni generali locali e remote



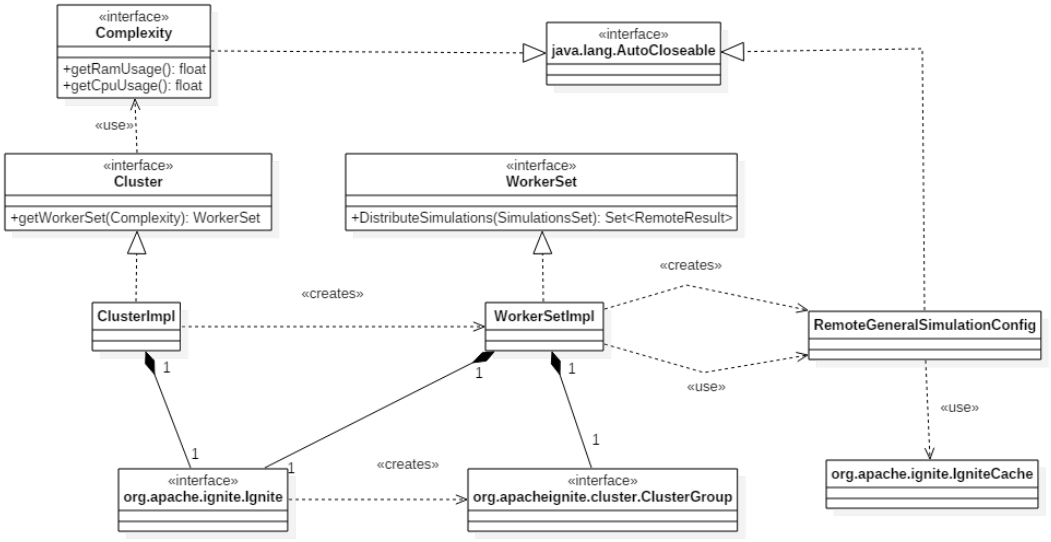
- **RemoteGeneralSimulationConfig**: classe che memorizza il contenuto dei file di dipendenza in modo da poter essere distribuiti. Come già descritto, infatti, Ignite supporta l'invio di oggetti direttamente inseriti nel task da eseguire solo per dimensioni inferiori a pochi Mb. Per questo motivo, vista la possibilità di avere file di dimensioni maggiori, questa classe utilizza le cache del framework per memorizzare il tutto e salva al suo interno solo la logica di reperimento.

L'implementazione locale è utilizzata all'interno dei client al momento della lettura del contenuto dei file, mentre quella remota, come verrà mostrato in seguito, viene utilizzata all'interno dei task remoti, ed è quindi effettivamente distribuita in rete.

Come si può vedere il `RemoteGeneralSimulationConfig` implementa anche l'interfaccia `AutoClosable`, in modo da poter essere utilizzata all'interno di un try-with-resources. Questo provvederà automaticamente all'eliminazione della cache del cluster chiamando il metodo "close".

2.4.3 Cluster e Nodi

FIGURA 2.8: UML di cluster e insiemi di nodi



Come si può notare dallo schema in Figura 2.8 l'astrazione del nodo è sparita dal design finale dell'applicazione. Questa scelta è stata presa per sfruttare i gruppi dinamici offerti da Ignite. Questi risultano infatti essere nativamente delle aggregazioni di nodi e variano come composizione nel tempo. Il framework offre anche un'astrazione relativa al nodo vero e proprio, ma sfruttarla avrebbe significato abbandonare l'utilizzo della dinamicità, che rappresenta una parte fondamentale delle strategie di load balancing utilizzate dal sistema.

Il load balancing utilizzato si basa infatti su due punti fondamentali:

- **Gruppi dinamici**: esattamente come progettato in fase di modellazione il `ClusterImpl` effettua vero e proprio load balancing, selezionando i soli nodi in grado di eseguire la computazione con successo. Questo lavoro di preselezione è, grazie alle tecnologie offerte, estremamente efficace, in quanto evolve all'evolvere del cluster stesso;

- **Job stealing**: l'algoritmo vero e proprio adottato è il job stealing, che è in grado di gestire ottimamente le differenze di potenza computazionale dei nodi appartenenti al gruppo. L'applicazione di questo algoritmo non viene attuata direttamente dal `WorkerSet` come pensato inizialmente, ma viene configurata

al momento dell'inizializzazione dei diversi nodi della griglia. Risulta quindi essere un utilizzo generico a livello di cluster, sfruttato di conseguenza anche dai gruppi, e quindi dai `WorkerSet`.

Il lavoro di selezione viene sempre effettuato partendo dall'oggetto `Complexity`, calcolato dal set di simulazioni.

Il `WorkerSet` si occupa anche della creazione della configurazione generale remota, che permette l'effettiva distribuzione dei file di dipendenza senza rischiare di superare la dimensione massima del task di Ignite. Utilizza inoltre sia l'istanza dell'oggetto `Ignite`, relativa al cluster, che quella di `ClusterGroup`, creata dal `ClusterImpl`. Questi due oggetti sono indispensabili per poter creare le cache del `RemoteGeneralSimulationConfig` e per distribuire le simulazioni che si andranno a creare.

2.4.4 Simulazioni remote

FIGURA 2.9: UML della gerarchia relativa alla simulazione remota

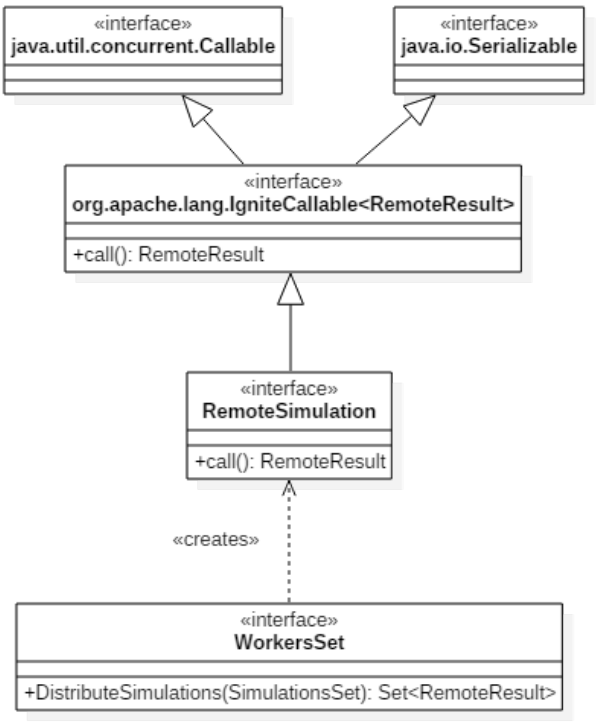
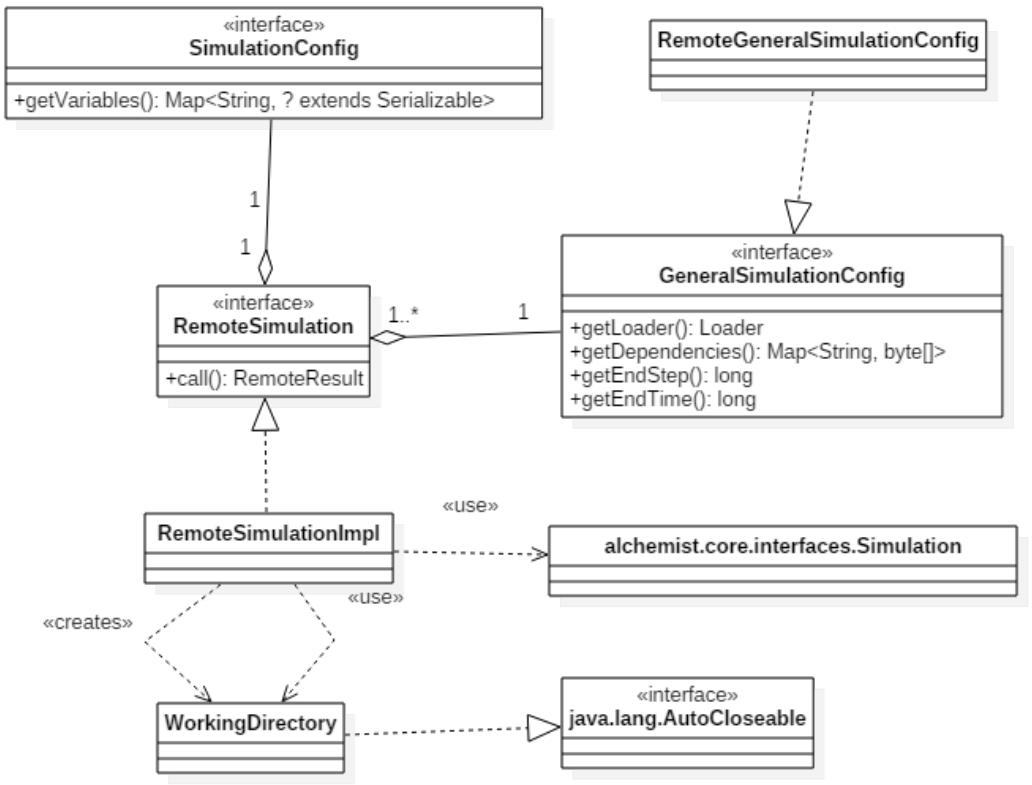


FIGURA 2.10: UML relativo alle dipendenze delle simulazioni remote



Le simulazioni remote definite in precedenza sono ora una specializzazione dell'interfaccia proprietaria di Ignite `IgniteCallable`(Figura 2.9), astrazione corrispondente al task, che può essere correttamente distribuita nella griglia dai `ClusterGroup`. Questa interfaccia a sua volta non è altro che la fusione delle interfacce native di Java `Callable` e `Serializable`. Quest'ultima, come abbiamo già visto più volte, è fondamentale per rendere qualunque oggetto trasportabile in remoto dal framework.

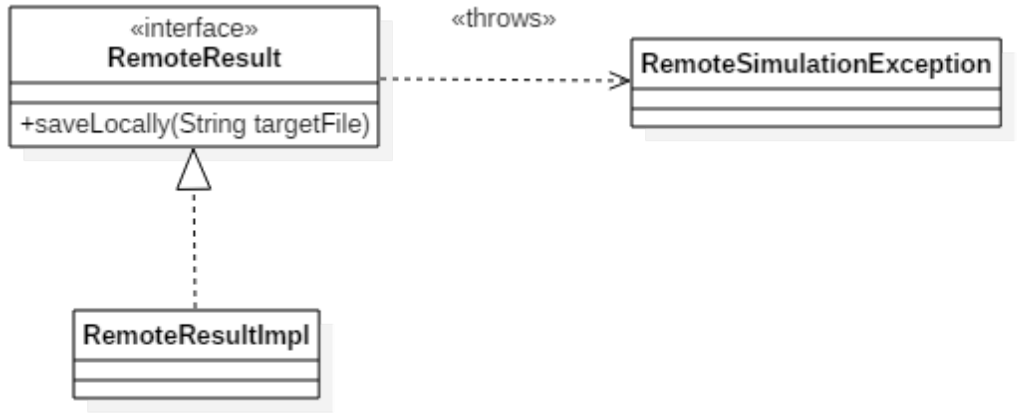
A livello funzionale (Figura 2.10) la simulazione ha al suo interno tutte le configurazioni necessarie all'esecuzione, che vengono trasmesse in remoto come campi interni all'`IgniteCallable`. Come abbiamo però già visto parte delle informazioni di configurazione generale sono in realtà salvate sulle cache di Ignite e il `RemoteGeneralSimulationConfig` presenta internamente la sola logica di reperimento. Partendo da queste informazioni va a creare la `Simulation` vera e propria di Alchemist e la esegue sul nodo.

Come si può vedere è indicata una dipendenza anche alla classe `WorkingDirectory`

che va a creare una cartella temporanea all'interno del sistema. Questa verrà poi utilizzata per salvare il contenuto dei file di dipendenza trasmessi ed aggiungerli al classpath. La simulazione richiede infatti di avere i file di dipendenza salvati in percorsi ben definiti, indicati nel file di configurazione principale. Questa funzionalità di Alchemist, estremamente radicata e difficile da variare, è risultata invariabile, ed è perciò stato necessario provvedere al salvataggio locale vero e proprio dei file nel sistema. Questa classe di utility implementa anche l'interfaccia `AutoClosable`, che le permette di essere automaticamente eliminata in caso sia utilizzata all'interno di un try-with-resources.

2.4.5 Risultato delle Simulazioni

FIGURA 2.11: UML relativo al risultato delle simulazioni



Per quanto concerne il risultato delle simulazioni sono stati effettuati diversi cambiamenti rispetto alla versione del modello. Il concetto di lista di `Sample` è infatti totalmente scomparso. Questa scelta deriva dal fatto che il comportamento di Alchemist, secondo il quale il risultato delle simulazioni viene eseguito su file, è risultato difficoltoso da modificare. Per questo motivo è stato deciso di continuare ad utilizzare il salvataggio su file anche in remoto e leggerne il contenuto a fine computazione, nonostante questo possa portare ad un leggero calo prestazionale dovuto alle operazioni di I/O. A questo punto ho deciso di fornire all'oggetto, al posto di un getter,

direttamente un metodo `saveLocally`, in grado di salvare il risultato su di un file dal percorso fornito.

Contestualmente alla creazione del risultato è stata creata anche un'eccezione relativa all'esecuzione remota della simulazione. Questa riporta informazioni riguardanti le eccezioni occorse in remoto, al nodo sul quale sono avvenute e al set di variabili che tale nodo aveva in carico. Questa strategia delega la notifica delle eccezioni proprie del funzionamento delle simulazioni Alchemist al client che ha lanciato il job, evitando che queste siano lanciate sui nodi remoti.

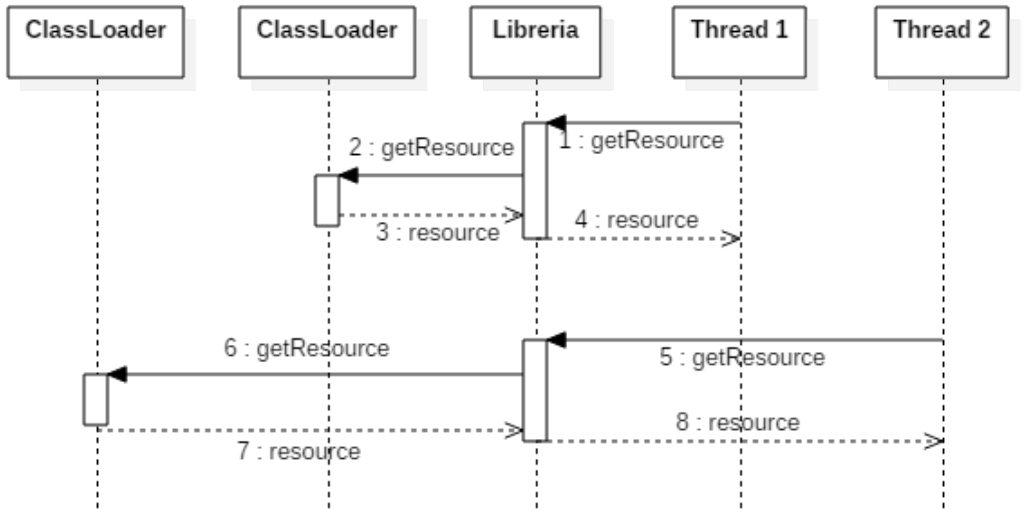
Ignite permette anche di adottare un approccio inverso; eventuali eccezioni remote non provocano infatti la caduta del nodo, ma portano ad una semplice notifica su di esso ed a un fallimento del job, notificato sul client. Le eccezioni di Ignite presentano però spesso poca chiarezza ed è stato perciò considerato migliore gestire esplicitamente eventuali situazioni impreviste.

È da notare come questo sistema si faccia carico solamente di eventuali eccezioni derivanti da un erroneo utilizzo delle simulazioni, e non di eventuali errori legati a cluster e funzionamento generico di Apache Ignite. Eventuali problemi di questo genere sono ancora totalmente gestiti dal framework, che presenta un livello di gestione delle diverse situazioni difficilmente riproducibile con eccezioni implementate ex-novo.

2.4.6 By-products

Un punto critico dello sviluppo del sistema è stato quello relativo all'aggiunta dinamica al classpath dei file di dipendenza delle simulazioni. In particolare i nodi remoti di Ignite eseguono più job senza essere riavviati, eventualmente anche in modo concorrente, con l'esecuzione contemporanea di task appartenenti a job diversi, e questo ha originato un problema di concorrenza. Nel caso in cui vengano eseguiti due job, infatti, può capitare che questi abbiano dei file di dipendenza differenti, ma che presentino il medesimo nome, come ad esempio due file dal nome "map". In questo caso un nodo, andando a ricercare suddetto file nel classpath, andrebbe a prendere la prima occorrenza trovata, e quindi potrebbe utilizzare una dipendenza non corretta. Questa funzionalità di gestione è stata inoltre ulteriormente complicata dalla richiesta di compatibilità sia con Java 8, attualmente utilizzato per lo sviluppo di Alchemist, che con

FIGURA 2.12: Schema relativo al funzionamento della libreria



Java 9. La nuova versione di Java, da poco rilasciata, varia infatti la gestione dei classpath con l'introduzione dei moduli, ed in particolare cambia l'implementazione del `ClassLoader` di sistema, al quale ora non è più possibile aggiungere dinamicamente percorsi nei quali cercare le risorse, come era possibile fare in passato.

Per risolvere il problema ho deciso di sviluppare una libreria in grado di caricare risorse in modo dipendente dal thread che lo richiede. Il suo funzionamento si basa su di una classe statica per il caricamento di risorse che utilizza internamente un `ThreadLocal`, in grado di restituire un differente `ClassLoader` per ogni thread che utilizzi suddetta classe. Questo `ClassLoader` personalizzato può essere inizializzato con un set di percorsi in cui cercare risorse, che viene aggiunto all'insieme appartenente al `ClassLoader` di default, e presenta inoltre la possibilità di aggiungere dinamicamente percorsi durante l'esecuzione.

Per permettere il corretto funzionamento del sistema la libreria viene ora utilizzata all'interno dell'intero simulatore, al posto del canonico caricamento delle risorse mediante il `ClassLoader` di sistema.

In questo modo il problema è stato risolto; quando una simulazione remota viene eseguita, infatti, viene lanciato un nuovo thread e viene aggiunto alla libreria il percorso della cartella temporanea di lavoro creata, contenente le dipendenze delle simulazioni. Ogni thread caricherà quindi le risorse corrette e non avrà all'interno

del `ClassLoader` che utilizza il percorso delle cartelle temporanee create dalle altre simulazioni.

Capitolo 3

Risultati, conclusioni e lavori futuri

Lo svolgimento del progetto ha portato con successo allo sviluppo di un sistema di grid computing per il simulatore Alchemist utilizzando il framework Apache Ignite. Questo sistema distribuisce in una griglia di nodi opportunamente configurati le simulazioni create durante la modalità "batch", nel caso in cui vengano utilizzate variabili dal valore multiplo. Alchemist presenta ora anche una modalità di esecuzione "nodo", che rende la macchina sulla quale è in esecuzione un componente di un cluster Ignite, pronto per eseguire task. Tutte queste funzionalità sono state integrate nel branch "develop" del sistema, in attesa di essere ulteriormente testate sull'esecuzione di esperimenti reali ed infine integrate in una versione ufficiale del simulatore.

3.1 Test del sistema

Sulla parte da me sviluppata non è stato possibile effettuare veri e propri benchmark per una serie di problematiche:

- **Disponibilità limitata di macchine**: per l'esecuzione di test sono stati messi a disposizione 3 server linux, la cui descrizione è demandata alla Sezione 3.1.1. Sebbene questo cluster permetta già di effettuare alcune valutazioni di prestazioni, test più significativi andranno svolti su sistemi con decine di nodi;
- **Natura delle macchine a mia disposizione**: i 3 server sono prestazionalmente estremamente eterogenei, il che rende complesso effettuare calcoli precisi sull'aumento delle prestazioni;

- **Tempo a disposizione**: oltre ai problemi legati alle risorse, è da notare come le fasi di ricerca, progettazione ed implementazione, con la relativa risoluzione dei vari problemi incontrati, abbiano lasciato poco spazio per l'esecuzione di benchmark molto sofisticati.

Nonostante questo le macchine sono state utilizzate per verificare il corretto funzionamento del sistema e hanno dato origine a dati che, per quanto limitati, fanno ben trasparire la convenienza di un sistema di grid computing. Su questi dati viene semplicemente calcolato il calo percentuale del tempo richiesto per l'esecuzione delle simulazioni in caso la computazione venga eseguita in maniera distribuita.

3.1.1 Descrizione delle macchine

I tre server linux a mia disposizione presentano le seguenti configurazione hardware per quanto riguarda CPU e RAM:

- **Server Andromeda**: processore Intel Core2 Quad Q9550 @ 2.83GHz 4 core 4 thread, RAM 8Gb;
- **Server Antares**: processore Intel Core i7-2720QM @ 2.20GHz 4 core 8 thread, RAM 8Gb;
- **Server Iris**: 2 x Intel Xeon X7350 @ 2.93GHz 4 core 4 thread, RAM 16Gb.

3.1.2 Esecuzione del test

Per l'esecuzione del test è stata utilizzata una configurazione dell'incarnazione di Alchemist chiamata "protelis", facendole generare prima 33 e poi 71 simulazioni, che differiscono per il valore di una sola variabile.

Per l'esecuzione locale dei due set di simulazioni sono stati utilizzati 4 thread su tutti e tre i server. Per la formazione della griglia Ignite, invece, sono stati configurati nodi sulle macchine "Iris" e "Antares", impostando per ognuna un numero di thread disponibili per l'esecuzione di task pari a 4.

La Tabella 3.1 presenta i tempi medi di esecuzione, sia per le esecuzioni locali, che per quelle distribuite. I tempi sono ottenuti facendo la media di 10 prove consecutive.

TABELLA 3.1: Dati relativi all'esecuzione delle simulazioni

Sim n	And	Ant	Ir	D	TA	TI	C%
33	575,69s	395,15s	578,52s	228,12s	20	13	42,27%
71	1113,17s	845,79s	1159,33s	494,14s	42	29	41,57%

Legenda: **Sim n** = numero di simulazioni, **And** = tempo di esecuzione su Andromeda **Ant** = tempo di esecuzione su Antares, **Ir** = tempo di esecuzione su Iris, **D** = tempo di esecuzione nella griglia Ignite, **TA** = numero di task eseguiti sul nodo Antares, **TI** = numero di task eseguiti sul nodo Iris, **C%** = calo percentuale

Come si può vedere dai dati il calo dei tempi richiesti per l'esecuzione delle simulazioni è pari a circa il 42%, calcolato prendendo come tempo di riferimento per l'esecuzione locale il migliore fra i tre a disposizione. Questo valore, considerando il passaggio da un'esecuzione a 4 thread per il caso locale ad una a 8 per il caso distribuito, ha un valore massimo teorico del 50%, dovuto al fatto che le entità deputate all'esecuzione di simulazioni, ossia i thread, raddoppiano. Il valore perfetto non potrà però mai essere raggiunto nel caso in esame.

Il server "Iris", i cui 4 thread si aggiungono a quelli di "Antares" nell'esecuzione distribuita, è infatti meno prestante del primo, come si può vedere dai tempi di esecuzione locali. Il tempo di esecuzione non potrà quindi dimezzarsi, come se le due macchine avessero processori egualmente potenti e si dividessero equamente il lavoro. Anche se quest'ultima supposizione fosse vera bisogna inoltre considerare che l'utilizzo di Ignite risulta essere un'ulteriore impedimento nel raggiungimento di un dimezzamento dei tempi. Il framework porta infatti con sé un'aggiunta ai tempi di pura computazione, relativa al trasferimento in remoto dei dati e all'unione effettiva del client alla griglia.

Alla luce di queste valutazioni l'incremento di prestazioni analizzabile dai dati disponibili è soddisfacente, ma, come descritto in precedenza, è necessario effettuare ulteriori test avendo più macchine a disposizione, in modo da verificare la presenza di eventuali cali di efficienza dovuti alla gestione di grandi numeri di nodi.

Infine è interessante notare come il job stealing, cardine del bilanciamento del carico da me adottato, funzioni come previsto, con un esecuzione media di task superiore per il server "Antares", più prestante, rispetto al server "Iris".

3.2 Lavori futuri

In questa sezione si vanno a descrivere le funzionalità del sistema lasciate incomplete per problemi di tempo a disposizione per lo sviluppo e che andrebbero integrate per ottenere un sistema completo.

3.2.1 Calcolo della complessità delle simulazioni

Una parte importante del sistema solamente modellata, ma non effettivamente implementata, è la stima della complessità delle simulazioni prima della loro distribuzione, citata nella Sezione 2.4.3. Questa funzione, come precedentemente descritto, è importante per evitare l'esecuzione di simulazioni su nodi che non sono in grado di eseguirle, ma presenta una certa complessità; il simulatore non è infatti in grado di valutare in modo banale quante risorse consuma una simulazione che sta eseguendo. Per questo motivo il sistema al momento non fa altro che distribuire le simulazioni su tutti i nodi disponibili. Questa mancanza non impedisce però totalmente l'utilizzo della parte di grid computing, che può essere infatti sfruttata con successo in cluster creati con sole macchine in grado di eseguire le simulazioni richieste.

3.2.2 Utilizzo di file di configurazione indipendenti da Ignite

Al momento per la configurazione di nodi e client vengono utilizzati i file di configurazione proprietari di Ignite, che consistono in documenti XML, mediante i quali è possibile definire varie proprietà. Questa caratteristica non intacca il funzionamento del sistema, ma sarebbe consono introdurre file di configurazione personalizzati che permettano di ridurre le opzioni personalizzabili nella creazione del cluster. Questa introduzione permetterebbe inoltre di distaccare ulteriormente il software dalla tecnologia utilizzata per l'implementazione vera e propria.

Ringraziamenti

Innanzitutto i miei ringraziamenti vanno al mio relatore, Prof. Mirko Viroli, e al mio correlatore, Ing. Danilo Pianini. È superfluo dire come senza la loro grande gentilezza e disponibilità questa tesi non avrebbe mai potuto vedere la luce.

In particolare è per me importante ringraziare l'Ing. Pianini, con il quale ho avuto modo di lavorare a stretto contatto, e grazie al quale ho potuto imparare molti aspetti della programmazione Java che non avevo avuto modo di esplorare in questi tre anni di percorso universitario.

Ringrazio poi la mia famiglia, ed in particolare i miei genitori e mia sorella, che in questi anni di studio e tensione mi sono stati sempre vicino.

Ringrazio i miei amici cesenati, che ho conosciuto per la maggior parte in concomitanza con l'inizio di questa avventura universitaria, e che in così pochi anni sono diventati una parte insostituibile della mia vita.

Ringrazio i miei compagni di corso, con i quali ho condiviso in questi anni le lunghe giornate di lezione e studio, e che grazie a loro sono sempre sembrate un po' meno dure. In particolare voglio citare Joseph e Giuseppe, miei ormai storici compagni di progetto; non penso avrei mai potuto trovare due matti migliori di voi. Preparatevi, ci aspettano altri due anni di nottate davanti al computer per finire progetti più grandi di noi.

Ringrazio inoltre tutti gli amici universitari che non sono strettamente miei compagni di corso, ma con i quali ho avuto modo di passare indimenticabili serate, di parlare delle nostre vite da studenti, e con i quali ho stretto delle vere e grandi amicizie.

Un ringraziamento particolare va al mio amico e fisioterapista Omar, che in questi anni mi ha rimesso letteralmente in piedi. Insieme a lui ringrazio tutti gli amici conosciuti in palestra, con i quali ho passato ore ed ore a parlare delle nostre vite, e tutti i compagni dell'"osteria", le nostre folli cene del martedì mi hanno aiutato in questo anno a smaltire stress e preoccupazione come nient'altro avrebbe potuto fare.

Bibliografia

[4] J. Banks et al. *Discrete-Event System Simulation*. 5ª ed. Prentice Hall, 2010. ISBN: 0136062121.

[5] Jacob Beal, Danilo Pianini e Mirko Viroli. «Aggregate programming for the internet of things». In: *Computer* 48.9 (2015), pp. 22–30.

[6] Oren Ben-Kiki, Clark Evans e Brian Ingerson. «YAML Ain't Markup Language (YAML™) Version 1.1». In: *yaml. org, Tech. Rep* (2005).

[7] Rajkumar Buyya et al. «Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility». In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[8] Ann Chervenak et al. «The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets». In: *Journal of network and computer applications* 23.3 (2000), pp. 187–200.

[10] Ian Foster. «What is the Grid? A Three Point Checklist». In: 1 (gen. 2002), pp. 32–36.

[11] Ian Foster et al. «Cloud computing and grid computing 360-degree compared». In: *Grid Computing Environments Workshop, 2008. GCE'08*. Ieee. 2008, pp. 1–10.

[12] Michael A Gibson e Jehoshua Bruck. «Efficient exact stochastic simulation of chemical systems with many species and many channels». In: *The journal of physical chemistry A* 104.9 (2000), pp. 1876–1889.

[13] Daniel T. Gillespie. «Exact stochastic simulation of coupled chemical reactions». In: *The Journal of Physical Chemistry* 81.25 (1977), pp. 2340–2361. DOI: 10.1021/j100540a008. eprint: http://dx.doi.org/10.1021/j100540a008. URL: http://dx.doi.org/10.1021/j100540a008.

[14] Carl Hewitt, Peter Bishop e Richard Steiger. «Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence». In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[16] Sara Montagna, Danilo Pianini e Mirko Viroli. «A model for drosophila melanogaster development from a single cell to stripe pattern formation». In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 1406–1412.

[17] Danilo Pianini, Sara Montagna e Mirko Viroli. «Chemical-oriented Simulation of Computational Systems with ALCHEMIST». In: *Journal of Simulation* (2013). ISSN: 1747-7778. DOI: 10.1057/jos.2012.27. URL: http://link.springer.com/10.1057/jos.2012.27.

[18] N. Sadashiv e S. M. D. Kumar. «Cluster, grid and cloud computing: A detailed comparison». In: *2011 6th International Conference on Computer Science Education (ICCSE)*. 2011, pp. 477–482. DOI: 10.1109/ICCSE.2011.6028683.

[19] Andrew S Tanenbaum e Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[20] Mirko Viroli et al. «A coordination model of pervasive service ecosystems». In: *Science of Computer Programming* 110 (2015), pp. 3–22.

[21] R Warren et al. «MammoGrid—a prototype distributed mammographic database for Europe». In: *Clinical radiology* 62.11 (2007), pp. 1044–1051.

[22] Belabbas Yagoubi e Yahya Slimani. «Dynamic load balancing strategy for grid computing». In: *Transactions on Engineering, Computing and Technology* 13 (2006), pp. 260–265.

[23] Wenju Zhang et al. «Drug discovery grid». In: *Proceedings of the UK e-Science All Hands Meeting*. 2005.