Università di Pisa
Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Simulatore per un servizio di consistenza su architettura Grid

Tesi di
Stefano Pellegrino

Relatori:

Prof. Andrea Domenici

Prof. Cinzia Bernardeschi

Laureando:

Stefano Pellegrino

Anno Accademico 2006/2007

Ai miei genitori

Ringraziamenti

Un ringraziamento particolare va a Gilda e a Giusy per l'affetto e l'aiuto morale che hanno saputo darmi, e per la pazienza avuta nei miei confronti non solo durante la stesura di questo lavoro, ma in tutta la durata della mia vita universitaria.

Un abbraccio va ai miei amici che non dimenticherò mai e che faranno sempre parte di me, e per i quali sarà sempre rivolto un pensiero particolare anche a distanza.

Un ringraziamento speciale va a Luca, che senza mai tirarsi indietro, mi ha aiutato in orari impossibili.

Infine ringrazio Il prof. Andrea Domenici per l'aiuto e i consigli forniti durante l'intero lavoro e l'Ing. Gianni Pucciani che mi ha sempre sostenuto, aiutandomi nei momenti di peggiore sconforto e difficoltà.

Indice

Elenco delle figure

Introduzione

Il Grid Computing rappresenta la nuova idea di architettura di calcolo parallelo, che sta destando un crescente interesse in tutti gli operatori del settore dell'Information Technology.

L'architettura Grid rappresenta una valida soluzione per il numero sempre maggiore di applicazioni che hanno l'esigenza di elaborare e gestire quantità enormi di dati eterogenei distribuiti su scala mondiale.

La gestione di grandi quantità di dati richiede però, la presenza di un meccanismo di replicazione che, per incrementare le prestazioni delle applicazioni, diminuisca i tempi di accesso ai dati stessi. Di conseguenza è necessario pensare a un servizio che garantisca la consistenza delle repliche dei dati.

Il servizio di consistenza studiato è il Replica Consistency Service CONStanza, che rappresenta una parte del ben più ampio progetto INFN Computational Grid.

CONStanza è un servizio di consistenza, che pur con le sue limitazioni, è stato implementato per le Grid reali. Questo lavoro nasce dalla considerazione che una Grid di questo tipo, se da un lato permette di testare l'effettivo funzionamento del servizio su esempi reali, dall'altro impone vincoli derivanti dalla sua topologia, con conseguente impossibilità di provare diverse configurazioni in molteplici scenari.

Una possibile soluzione è quella di simulare una Grid nella quale sia attivo un meccanismo di replicazione, su cui poi avviare il servizio di consistenza. Nel panorama Grid l'unico simulatore che ci permette flessibilità sia dal punto di vista topologico che dal punto di vista delle strategie di replicazione dei dati è OptorSim. L'integrazione di quest'ultimo con CONStanza è un modo per ottenere uno strumento idoneo per simulare un servizio di consistenza delle repliche in ambiente Grid.

Il fine non è proporre la miglior soluzione possibile, bensì studiare le varie problematiche connesse alla simulazione e fornire un prototipo di simulatore che risulti adatto allo scopo.

Diamo uno sguardo all'organizzazione dei vari capitoli.

Capitolo 1 si introduce il concetto di Grid e se ne descrive la sua architettura. Si effettua, inoltre, una panoramica del Grid Computing descrivendo da dove nasce e quali sono i problemi ancora aperti.

Capitolo 2 si tratta l'argomento della Grid Simulation. Quali sono i problemi e quali sono le soluzioni adottate nella simulazione di un ambiente Grid.

Capitolo 3 si affronta il problema della consistenza della repliche. In particolare si descrive il funzionamento del Replica Consistency Service CONStanza dal punto di vista architetturale.

Capitolo 4 si analizza l'ambiente di simulazione, ossia si vedrà come sia possibile simulare il funzionamento di una Grid. In dettaglio si analizzerà il funzionamento del simulatore Optorsim e il suo utilizzo al fine di ottenere diversi scenari di simulazione.

Capitolo 5 si descrive come mediante l'ausilio di Optorsim si possa realizzare un simulatore che permetta di testare la scalabilità di CONStanza all'interno di molteplici scenari di Grid simulate.

Capitolo 6 Si descrivono i dettagli implementativi che permettono ai due software di cooperare.

Capitolo 1

Grid Computing

Il Grid Computing rappresenta oggi l'ultima frontiera delle attività di ricerca nel campo delle architetture di calcolo parallelo. Tale fatto è testimoniato sia dal numero di progetti di ricerca ad esso dedicati, sia dal numero di grandi ditte commerciali di sistemi di calcolo che si sono dotate della loro *soluzione Grid*[1]. Il termine *Grid* infatti, rappresenta la formulazione particolarmente fortunata di una idea, quella della condivisione delle risorse di calcolo, che si è faticosamente sviluppata negli ultimi decenni in seguito alla rapida evoluzione delle tecnologie informatiche. Un'idea è scaturita dalla necessità di superare il problema della eterogeneità, sia hardware che software, al fine di fornire una immagine integrata del sistema.

1.1 Cenni storici

Il fenomeno della condivisione delle risorse di calcolo e la sua evoluzione si può dividere in quattro periodi, ognuno dei quali caratterizzato da diverse metodologie e architetture dedicate al supercalcolo.

Si parte dagli anni '70, nei quali si afferma il concetto di *"un computer per molti utenti"*. In questo periodo, il costo della singola risorsa di calcolo è tale da essere accessibile soltanto a grandi entità (militari e civili) , in grado di ammortizzare l'investimento solo attraverso l'uso contemporaneo della risorsa da parte di numerosi utenti (si tratta per lo più di risorse computazionali con un'architettura basata su soluzioni *ad hoc* sviluppate da ditte specializzate).

Tuttavia, alla fine degli anni '80, l'attenzione di alcuni gruppi di ricercatori

[1] **IBM Grid Computing**, http://www-1.ibm.com/grid/
Sun Solutions: Grid Computing, http://www.sun.com/solutions/infrastructure/grid/
Get on the Grid with Oracle 10g, http://www.oracle.com/events/grid/index.html

inizia a focalizzarsi sulla possibilità di sfruttare sia la sempre maggiore potenza di calcolo disponibile su una workstation, sia le tecnologie a basso costo che permettono l'interconnessione in rete locale di tali workstation, al fine di costruire macchine parallele virtuali [44] .

Gli anni '90, invece, introducono il concetto di *"molti calcolatori per singolo utente"*. La definitiva affermazione della legge di Moore [34, 41] nel campo dei Personal Computer comincia a rendere economicamente impraticabile ogni tentativo di sviluppare architetture di calcolo ad hoc. Contemporaneamente, in questi anni, si affermano anche tecnologie mainstream che, a fianco all'incredibile evoluzione delle CPU, rappresentano la seconda colonna portante dell'idea stessa di Grid: le reti di calcolatori ed internet.

1.1.1 Dal Cluster al Grid

Nella seconda metà degli anni '90 nasce, presso l'università Stanford, il progetto *Beowulf* [2] e inizia così l'ultima fase dell'evoluzione.

Beowulf rappresenta il primo progetto di architettura *Cluster* , cioè un tipo di architettura che permette di aggregare staticamente nodi di computazione dedicati (sia workstation che PC), interconnessi tramite un'interfaccia di rete di tipo standard. Questo progetto rappresenta un punto di svolta, poichè si assiste a un'inversione di tendenza da parte della ricerca, che sposta la sua attenzione dallo sviluppo dell'hardware a quello del software.

Di conseguenza, l'obiettivo dello sviluppo di sistemi di calcolo ad alte prestazioni, diventa la definizione e l'implementazione del cosiddetto *middleware*[2], ossia strati software che riescano a mascherare l'eterogeneità delle risorse di calcolo a chi utilizza il sistema. Il passo successivo è la nascita di numerosi progetti *metacomputing*[3]. Questi sistemi, oltre ad avere molte caratteristiche in comune con i sistemi Cluster, introducono alcuni concetti che si riveleranno fondamentali per l'architettura del Grid Computing, definendo modelli di sistemi formati dall'aggregazione dinamica di nodi computazionali non dedicati al metacomputer, interconnessi anche attraverso Internet e appartenenti a domini di sicurezza e gestione non omogenei.

La durata del metacomputing è stata breve (circa due anni) poichè tra l'inizio del 1998 [29] e la prima metà del 1999 [30], Ian Foster e Carl Kesselmann, i padri del progetto Globus [5] , pubblicano la loro visione anticipatrice

[2]si intende un insieme di programmi informatici che fungono da intermediari tra diverse applicazioni. Sono spesso utilizzati come supporto per applicazioni distribuite complesse. I software utilizzati possono anche essere più di uno

[3]Il termine metacomputing viene effettivamente coniato da L. Smarr, direttore NCSA, alla fine degli anni '80, tuttavia è solo nel 1997 che tale termine prende piede con il suo significato attuale

di una *"Griglia di distribuzione della potenza computazionale in cui, come i watt nella griglia di distribuzione dell'elettricità, la potenza computazionale può essere distribuita a chi ne fa richiesta senza badare alla sua provenienza"*. Questa futuristica visione ha un immediato successo e viene adottata dalla comunità scientifica sostituendo il concetto di metacomputing.

1.1.2 Le diverse visioni del Grid Computing

Da quando la ricerca dedicata ai sistemi di calcolo ha spostato la sua attenzione dall'hardware al software, hanno preso corpo diverse visioni del Grid Computing, essenzialmente derivate dal tipo di utilizzo a cui la Grid deve essere destinata.

Grid adibita al supercalcolo: per la comunità del supercalcolo, una Grid consiste nell'interconnessione di supercalcolatori situati in diversi centri, al fine di realizzare una struttura globale in grado di lanciare l'esecuzione di una applicazione su uno qualunque dei supercalcolatori appartenenti alla Grid. In realtà, attualmente, quasi tutte le applicazioni destinate al supercalcolo sono sostanzialmente le stesse che venivano scritte per le architetture cluster, con la differenza che, mentre quest'ultime utilizzavano risorse il più possibile omogenee tra loro, il sistema Grid, invece, utilizza risorse di tipo eterogeneo. Con questo tipo di struttura, le applicazioni, a meno di non accettare degradi prestazionali fortissimi, si devono modellare sulle caratteristiche del sistema di calcolo, allontanadosi così da uno dei concetti fondamentali del sistema Grid: la variazione dinamica delle risorse. Inoltre, pur essendo applicabile a una vasta tipologia di utilizzi, un'applicazione di questo tipo richiede uno sforzo di ingegnerizzazione o di reingegnerizzazione molto elevato, e spesso affrontabile solo da programmatori esperti nel campo del calcolo parallelo ad alte prestazioni. Questo è il motivo per cui solo la grande industria, oppure quegli enti che hanno a loro disposizione uno staff IT che si dedichi a queste problematiche, possono usufruire di tutte le soluzioni disponibili in questo ambito.

Grid basata su applicazioni Monte Carlo [45]: una seconda visione di Grid è quella adottata da progetti quali SETI@home [12] , Entropia [4] o AVAKI [1]. Caratteristica comune di questi progetti è l'esistenza di un centro di controllo della Grid che gestisce l'allocazione di parti dell'esecuzione globale dell'applicazione a nodi che possono essere ottenuti dinamica-

mente, o su base volontaria, oppure identificando i nodi disponibili all'interno di una intranet (tipicamente aziendale). Le applicazioni generate in questo modo si definiscono di tipo *Monte Carlo*, realizzate con un modello computazionale *Farmer-Worker*[4] ad accoppiamento ridotto. Questo tipo di soluzione permette sia la creazione di nuove applicazioni, sia il trasporto di applicazioni preesistenti verso piattaforme di tipo Grid, con un costo di reingegnerizzazione piuttosto basso. Tale soluzione, però, è adottabile soltanto per una specifica categoria di applicazioni (basate su simulazioni di tipo Montecarlo) o, più in generale, per quelle in cui è necessario eseguire una stessa sequenza di operazioni su un insieme enorme di dati, che possano essere ripartiti in sottoinsiemi eleborabili in modo indipendente. Questi vincoli sulla tipologia di applicazione fanno sì che essa non sia generalizzabile ad un ampio bacino di utenti, ma che sia principalmente adottata da programmi specifici, quali i test sui numeri primi,il progetto SETI, oppure per l'analisi del genoma umano.

Grid orientata ai servizi: una terza visione delle Grid è quella che utilizza la tecnologia *Web Services*. Tale visione si propone lo sviluppo di applicazioni capaci di sfruttare la tecnologia Grid Computing basata su servizi con interfacce standardizzate. Il modello definito dal progetto *Open Grid Service Architetture*(OGSA) [10] organizza una Grid in modo di permettere e coordinare l'uso di risorse condivise, a un insieme di individui e/o istituzioni riuniti in *"organizzazioni virtuale"*(VO). Una soluzione di questo genere, pur sacrificando una parte dell'ottimalità prestazionale raggiungibile solo con paradigmi di computazione parallela, permette di far accedere alla tecnologia Grid Computing un bacino di utenza che non necessariamente deve essere esperta delle problematiche della computazione parallela, ma che tuttavia in questo modo, può disporre di una capacità di calcolo superiore a quella che si potrebbe permettere.

[4]Il modello computazionale Farmer-Worker viene comunemente realizzato nel caso di applicazioni decomponibili in task (*Bag of Work*) tra di loro indipendenti, cioè che per la loro esecuzione non necessitano di comunicare con altri. Un Farmer genera un insieme di Bag of Work che vengono messi a disposizione dei Worker. Ogni Worker diponibile estrae un Bag of Work dall'insieme e lo esegue. Alla conclusione di questo task il Worker restituisce i risultati ottenuti al Farmer che, trovandosi ora nello stato ideale, si rimette in attesa.

1.2 Le organizzazioni virtuali

Il concetto di *Virtual Organization* (VO) nasce contestualmente a quello di Grid Computing, e rappresenta la visione di *utente*, inteso come gruppo di persone che partecipano allo sviluppo di uno o piu progetti, che usufruisce di una rete Grid.

Si definiscono VO, gruppi dinamici di individui e/o istituzioni a cui interessa non il mero scambio, ma l'accesso diretto a computers, software, dati, e altre risorse, al fine di implementare una vasta gamma di strategie adatte a risolvere i nuovi problemi che stanno emergendo nei campi dell'industria, della scienza, e dell'ingegneria [32] . Queste VO sono entità altamente dinamiche, sia per quanto riguarda il numero di partecipanti, la durata e la scala delle loro operazioni, che per quantità e modalità di condivisione delle risorse.

Ogni VO, inoltre, deve implemetare una politica che impone dei vincoli sull'utilizzo delle risorse, poichè ogni membro deve poter accedere in modo semplice e trasparente a tutte le risorse di una Grid e, allo stesso tempo, poter essere libero di stabilire cosa può essere fatto o no, con le proprie risorse.

L'implementazione di questi vincoli richiede, oltre ai meccanismi per esprimere le politiche di utilizzo di una particolare risorsa, altri, come *"VOMS"* [14] e *"LDAP"* [39] , utilizzati per stabilire le identità di chi accede alle risorse stesse e, in base a questo, determinare se una data operazione è autorizzata oppure no.

1.3 Architettura di una Grid

Quando, nel '90, gli sviluppatori e ricercatori si riunirono prima nel *Grid Forum* e successivamente nell'attuale *Global Grid Forum* [6] , definirono gli standard per questo tipo di tecnologia, indicando quali fossero le principali aree di interesse:

- Gestione della parte fisica (Hardware);
- Gestione del lavoro e delle prestazioni;
- Gestione dei sistemi ed automazione;
- Gestione dei servizi;
- Gestione dei Logical Resource;

- Servizi di Clustering;
- Servizi di connettività;
- Sicurezza;

Oggi si cerca di far convergere le diverse visioni di Grid verso una infrastruttura globale che coinvolge nello sviluppo varie organizzazioni dislocate su vari continenti.

Una Grid può essere vista come un insieme di protocolli, servizi ed API (*Application programming interface*) [32]. Questi tre componenti sono altamente correlati ed hanno bisogno l'uno dell'altro per poter svolgere la propria funzione. In effetti, la Grid fisicamente può essere considerata come una combinazione di computers con architetture eterogenee collegate tra di loro tramite una rete. Poichè una delle caratteristiche fondamentali della Grid è l'interoperabilità delle risorse, essa può essere garantita all'interno di una rete solo se si utilizzano protocolli comuni di comunicazione. Di conseguenza, una Grid è soprattutto un'architettura di protocolli che permettono agli utenti di negoziare, stabilire e controllare le risorse condivise. Potendo essere assimilata a un'architettura di protocolli, una Grid può essere visualizzata in maniera simile a quella dei modelli OSI o TCP/IP.

1.3.1 Visione stratificata di una Grid

L'architettura di una Grid può essere schematizzata in una struttura a livelli simile a quella illustrata nella figura 1.1.
In dettaglio questi livelli sono:

Livello fabric: comprende le componenti hardware del sistema, cioè le risorse il cui accesso è mediato da parte dei protocolli sovrastanti.

Livello dei protocolli: contiene le autenticazioni e le comunicazioni all'interno della Grid. In questo livello sono definiti i protocolli di base per la comunicazione semplice e sicura tra le risorse del livello fabric. Si cerca sempre di sfruttare protocolli già consolidati, sia per quanto riguarda la comunicazione, sia per quanto riguarda l'autenticazione e la protezione delle comunicazioni.

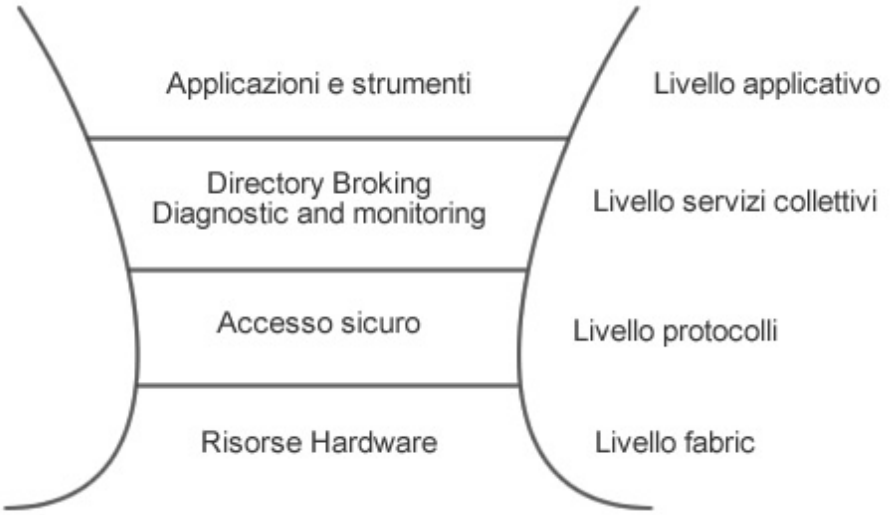

Figura 1.1: Architettura stratificata di una Grid

Livello di servizi collettivi: contiene i servizi, le API, che permetto la comunicazione tra applicazioni. Questo livello contiene protocolli e servizi che riguardano il sistema nella sua globalità e le interazioni tra i vari componenti. Si possono collocare in questo livello tutti i servizi che consentono di reperire le risorse in base a particolari condizioni e di effettuare operazioni di allocazione e controllo dello stato delle risorse.

Livello applicativo: contiene le applicazioni rivolte agli utenti. In questo livello sono presenti quelle applicazioni che permettono ad un ipotetico utente, inserito nel contesto di una VO, di vedere la Grid come un unico supercomputer.

Capitolo 2

Simulazione di una infrastruttura Grid

In questo capitolo si descriveranno quali sono le principali caratteristiche da modellare e implementare per ottenere una simulazione di una Computing Grid che si avvicini il più possibile ad un sistema reale. Si farà poi una panoramica dei i vari simulatori di Grid esistenti, affinché possano essere meglio comprese le scelte adottate in seguito.

2.1 Modello di ambiente di Grid

Una Grid permette l'aggregazione di risorse sia di calcolo distribuito che di dati tra loro eterogenei, contenuti in molteplici domini amministrativi [28].
Si crea così un grande sistema informatico integrato che fornisce un accesso diretto, multiplo e ridondante alle più disparate risorse. Le applicazioni che traggono maggior beneficio dalla loro esecuzione sulla Grid sono quelle che, oltre ad un certo grado di paralellismo, devono integrare dati attraverso numerose fonti.
Il workload di una applicazione contiene un insieme di task che devono essere eseguiti dalle risorse della Grid. Essi vengono inviati ai Resource Brokers, il cui scopo è quello di fungere da *gateway* per la Grid.

Generalmente, ci possono essere più Resource Brokers, ognuno dei quali possiede una propria politica di schedulazione, che indica come assegnare i vari tasks alle risorse disponibili sulla Grid. Nella figura 2.1 è rappresentato un ambiente Grid generalizzato.

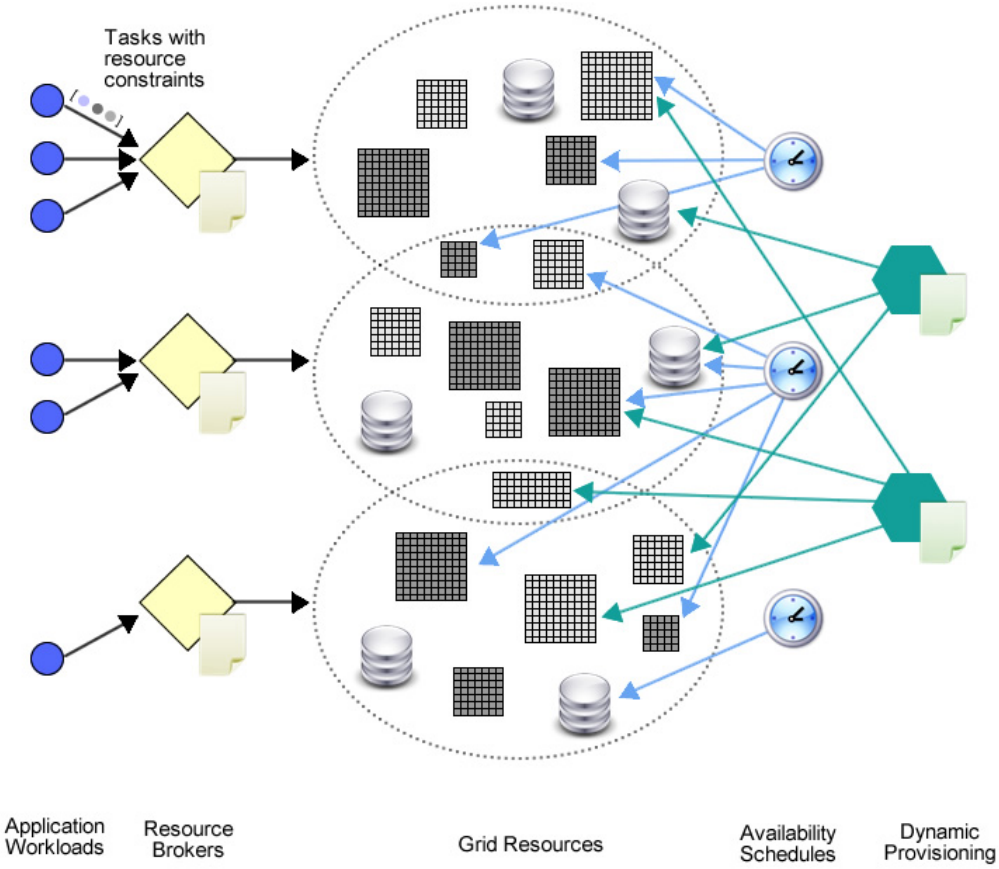

Figura 2.1: Flusso operativo in un ambiente Grid generalizzato

Le risorse possono essere di diverso tipo e totalmente indipendenti dalle piattaforme hardware e dai vari sistemi operativi. Generalmente, queste risorse vengono assegnate facendo riferimento alla loro disponibilità in base al tempo di schedulazione, disponibilità che può essere influenzata dalla cosiddetta *Dynamic Provisioning* [26].
Il *Dynamic Provisioning* è la possibilità di configurare automaticamente una risorsa, al fine di elaborare un task di un certo tipo (per esempio un Web Server può essere configurato come Application Server per elaborare alcuni tipi di transazioni effettuate on line). Il Dynamic Provisioning, mediante le sue politiche di schedulazione, cerca di determinare quando la risorsa potrebbe passare da uno stato all'altro, in modo che il task venga eseguito correttamente

(riferendosi all'esempio precedente, il provisioning rappresenta il passaggio dal web server all'application server).
Pertanto, essa rappresenta la capacità di una Grid di configurare automaticamente una risorsa in modo tale che l'esecuzione del task sia la migliore possibile [15].
Tenendo conto di queste caratteristiche, è opportuno analizzare in dettaglio i parametri che rappresentano meglio la Grid, soprattutto quelli che possono influenzare positivamente o negativamente una simulazione.

2.2 Caratteristiche di una simulazione

Un ambiente Grid possiede delle caratteristiche che una simulazione ben fatta deve sapere modellare e implementare nella maniera più coerente possibile [31].Le più significative sono:

Decomposizione dei job: ogni job appartenente ad un workload, durante la sua esecuzione, può richiedere l'accesso a più risorse. In questo caso, si indicherà con il termine *task* una richiesta fatta dal job ad una singola risorsa. Ogni richiesta deve essere strutturata in modo specifico a seconda della configurazione del server o del database.
Si prenda per esempio un job composto da 3 task:

1. il task uno che ottiene i dati da un database clienti
2. il task due che effettua un datamining sul server di calcolo
3. il task tre che aggiunge il risultato al database delle vendite

In questo esempio il job originale è stato scomposto in tre task, ciascuno dei quali effettua un azione specifica all'interno del nostro server o database.
L'importanza della decomposizione dei job dipende dal fatto che il punto di forza del progetto di una Grid è allocare un certo tipo di risorsa quando essa viene richiesta (ad esempio, un server di calcolo). Tutto ciò porta a trascurare l'effetto che una siffatta gestione delle richieste ha sulle altre risorse, nonostante queste ultime siano necessarie per i jobs in esecuzione.

Parallelizzazione dei tasks: un task appartenente ad un job (decomposto conformemente a quanto descritto al punto precedente) può essere parallelizzabile. Esistono due tipologie di parallelizzazione:

- al primo tipo appartengono gli *Embarrassingly parallel* task, i quali possono essere scissi in tante parti quante sono le risorse disponibili;
- al secondo tipo appartengono i task paralleli, che non tengono conto di tutte le risorse disponibili, ma fanno riferimento solo ad un numero di risorse definito.

Elaborazione multitask delle risorse: i processori, i database server, i link di rete, i data storage sono tutte risorse preemptive e multitasking. I task che devono accedere ad una risorsa, una volta avviati, vanno a finire nella coda dei processi da elaborare, la quale contiene tutti gli altri task che attendono l'accesso alla stessa risorsa.
Ognuno di questi task ottiene il diritto di accedere alle risorse per un intervallo di tempo predefinito. Trascorso tale intervallo, il task viene rimesso nella coda dei processi. Il multitasking, quindi, ha come risultato quello di cambiare il tempo di completamento di tutti i task che precedentemente avevano già richiesto l'accesso alla risorsa.
Una simulazione dettagliata del gestore dei task, all'interno di ogni singola risorsa, potrebbe richiedere molto tempo, tenendo conto del fatto che, di solito, un ambiente Grid consta di centinaia di risorse simulate e che le simulazioni possono durare anche una settimana. In realtà, il simulatore dovrebbe riuscire ad effettuare, nel momento in cui il nuovo task richiede l'accesso alla risorsa, una buona approssimazione del comportamento multitask, ricalcolando il tempo di completamento dei vari task che in quel momento attendono di utilizzare la risorsa.

Risorse eterogenee: le risorse di una Grid possono appartenere a diversi produttori con evidenti differenze sia a livello di piattaforme hardware che software. Di conseguenza, il tempo di esecuzione di un task su una risorsa qualsiasi dipende dalle prestazioni della medesima risorsa.
Il risultato è che, a seconda di quale task ci si trova ad eseguire, si possono avere prestazioni notevolmente lontane tra loro.

Schedulazione delle risorse: un simulatore Grid deve essere in grado di modellare le politiche di schedulazione usate dai Resource Broker, per determinare quali risorse debbano essere allocate, non appena arrivano le richieste da parte dei task. Le più importanti politiche sono:

- Load Leveling: il task viene assegnato alla risorsa meno utilizzata,

in modo da bilanciare il carico tra tutte le risorse compatibili con l'operazione;

- Greedy: il task viene assegnato alla risorsa che può completarlo il più velocemente possibile;
- Round Robin: il task è assegnato alla successiva risorsa compatibile, indicata in una sequenza precedentemente definita;
- Threshold Based: il task viene assegnato a una risorsa preferita fino a che non venga superato un certo parametro di soglia prestazionale.

I progettisti Grid possono usare la simulazione per valutare quali siano le scelte migliori atte ad aumentare le prestazioni della Grid stessa.

Provisioning delle risorse: l'abilità di provisioning delle risorse per la elaborazione di particolari tipi di task è una importante caratteristica da simulare.
Le politiche di provisioning possono essere basate su un certo calendario (per esempio, un dipartimento necessita di un e-mail server per un determinato periodo pomeridiano e di un datamining server per il resto della giornata) oppure su una politica più dinamica, che controlla sia gli arrivi dei workload che l'uso delle risorse e si comporta di conseguenza.
La simulazione di queste politiche di provisioning, in combinazione con le politiche di schedulazione delle risorse, permetterebbe a chi progetta una Grid di determinare se le rispettive politiche siano allineate ai loro parametri, fornendo in questo modo le prestazioni desiderate in termini di disponibilità delle risorse, workload throughput e di tempo di elaborazione.

Interfaccia Utente: i soggetti destinati ad usufruire della simulazione dovrebbero essere gli utenti non pratici di linguaggi di simulazione.
Di conseguenza, il simulatore, fornendo un ambiente grafico ed una interfaccia utente, potrebbe consentire, a chi progetta una Grid, di scrivere nuove politiche di schedulazione e provisioning per incorporarle in una simulazione.

2.3 Modello di simulatore

In questa sezione si approfondiranno i tratti generali di un simulatore Grid, chiarendo gli aspetti fondamentali della sua architettura. Si farà riferimento ad un simulatore generico, che soddisfi i requisiti esposti nel paragrafo precedente, e che corrisponda allo strumento ideale che progettisti e consulenti Grid vorrebbero avere a disposizione per analizzare l'infrastruttura IT della Grid stessa. Questa premessa si rende necessaria in quanto, come sarà in seguito esposto, i simulatori di Grid reali non ricalcano completamente il modello ideale, ma tendono invece a valorizzare alcuni aspetti della simulazione, trascurandone altri.

2.3.1 Progetto di un simulatore

Si prenderà in considerazione un sistema Grid suddiviso in tre parti, ciascuna della quali rappresenta un pezzo del flusso operazionale interamente eseguito su una Grid. La figura 2.2 mostra lo schema UML di una Grid simulata [16].

La parte della Grid relativa alla domanda di risorse si trova sulla sinistra e tiene conto, per quanto riguarda le applicazioni (1), dei requisiti dei task (2) che la compongono; invece per quanto riguarda i workloads, della statistica dei loro tempi di arrivo alla Grid.
La parte destra delle figura, invece, è formata da classi di risorse (5) valutate attraverso molteplici benchmark (6), le istanze di queste risorse (7) e la loro disponibilità (8).
Le due parti della figura sono collegate da uno o più Resource Brokers (9), che gestiscono l'esecuzione dei task sulla Grid, mediante degli schedulers (10) che decidono su quale risorsa un workload deve essere indirizzato per essere eseguito.
Diverse simulazioni possono essere realizzate separatamente per ogni scenario (11); tuttavia, per ognuna di esse, è necessario specificare, workload, istanze di risorse e Resource Brokers.
In un modello possono coesistere diversi scenari(12), ma perchè ciò sia possibile, essi devono avere in comune applicazioni e classi di risorse.

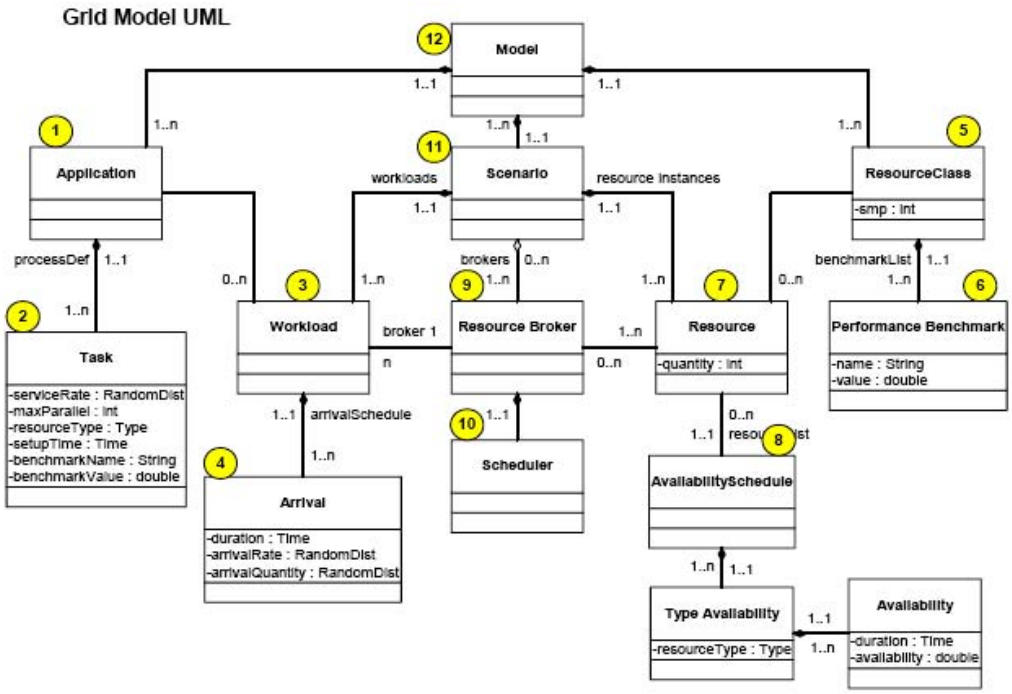

Figura 2.2: Modello UML di Grid per la simulazione

2.3.2 Simulatori Reali

Si descriveranno ora una serie di simulatori reali, appositamente progettati per simulare una Grid, nei quali i requisiti necessari per una corretta simulazione sono stati sviluppati a vari livelli. Ogni simulatore possiede pregi e difetti:

GridSim [7] è un simulatore scritto in linguaggio Java basato interamente sulla SimJava Library. Esso è stato progettato specificatamente per analizzare e confrontare le simulazioni dei vari algoritmi per la schedulazione delle risorse [36].
Le caratteristiche principali sono:

- La possibilità di modellare risorse eterogenee e multitasking all'interno delle griglie;
- Provisioning delle risorse basato su un calendario stabilito;
- La possibilità di modellare applicazioni parallele che girano sulla Grid;
- Schedulazione delle risorse.

Lo svantaggio principale è quello di implementare ogni Grid task come un thread separato, in esecuzione sulla Java Virtual Machine. Infatti, nell'affrontare simulazioni (tipicamente orientate alle esigenze di grandi organizzazioni) in cui sono presenti milioni di Grid task, questo tipo di implementazione rappresenta un limite per la scalabilità del software [42].
Inoltre il simulatore presuppone che l'utente possieda una certa capacità di programmazione, che consenta di sfruttare al massimo tutte le potenzialità del simulatore stesso (per esempio, si affida al programmatore il compito di effettuare l'implementazione della decomposizione dei job).

SimGrid [13] è un framework di simulazione, nato allo scopo di valutare gli algoritmi di schedulazione adottati da una Grid. Il modello delle risorse è eterogeneo e multitask. Il multitasking è implementato in shared mode, il che vuol dire che tutti i task attendono di essere eseguiti concorrentemente su una risorsa condivisa [40].
Il punti di forza della Sim Grid sono: la descrizione topologica della Grid Network e la simulazione del flusso di dati sulla rete in funzione della banda disponibile.

Tuttavia, Sim Grid non possiede una visione realistica del workload, poichè non provvede nè alla decomposizione dei job, nè assicura la caratteristica di parallelismo dei task.

Optorsim [11] è un simulatore scritto in Java concepito per valutare le prestazioni ottenute mediante iterazione delle politiche di schedulazione delle risorse con il provisioning dinamico [19]. Nel dettaglio, il provisioning dinamico viene effettuato nel contesto della replica dei dati nelle risorse di una Data-Grid.
Varie strategie di replicazione dei dati possono essere valutate in base alla prospettiva di massimizzare il job throughput, tuttavia il maggior peso della gestione dei dati va a discapito della capacità di calcolo, di solito presente su una Grid.
Le applicazioni sono considerate solo in base ai dati che esse richiedono.
Il modello di simulazione, inoltre, non considera l'eterogeneità delle risorse, parametro utile per ricavare informazioni sia sul tipo di risorsa che sulle sue prestazioni.

Capitolo 3

CONStanza: un Replica Consistency Service per Grid

In questo capitolo ci soffermeremo sul meccanismo di replicazione dei dati, analizzando soprattutto il problema della sincronizzazione e della consistenza tra repliche. Descriveremo inoltre CONStanza, un servizio di consistenza per le repliche di una Grid.

3.1 La consistenza dei dati

Una Data Grid è un'infrastruttura di calcolo che fornisce la capacità di memoria e di elaborazione alle applicazioni che trattano grandi quantità di dati.

Le Data Grids sfruttano il meccanismo delle repliche per ottenere migliori prestazioni ed affidabilità, immagazzinando copie dei dati su differenti nodi della Grid.
In questo contesto, le repliche rappresentano due o più istanze fisiche di uno stesso file logico. Un'eventuale operazione di modifica (update) su una di esse deve essere propagata a tutte le altre.

In generale, per la gestione dei dati, il middleware Grid assume che i files siano una singola unità di replica e che, una volta replicati, essi siano solamente read-only [38]. Se nella Grid però i dati sono solo read-only, essi sono sempre consistenti, poiché non è possibile effettuare aggiornamenti sulle repliche. Tuttavia esistono delle applicazioni che per la loro esecuzione hanno necessità di aggiornare spesso dei file la cui replica è presente in diversi nodi della Grid. Di conseguenza, se si considera possibile aggiornare una replica, si deve considerare anche il problema della sua consistenza.

Il problema della consistenza delle repliche è stato già analizzato sia nel

campo dei sistemi distribuiti che in quello dei database, ma le soluzioni conosciute sono soltanto parzialmente applicabili alle architetture Grid. Infatti, se prendiamo come esempio un Database Manager System (DBMS), esso ha controllo completo di tutti i dati che gestisce, dati che hanno un formato omogeneo. Al contrario, nelle Data Grid la manipolazione di dati avviene mediante i file-systems presenti in ogni nodo, che per mantenere la consistenza, non offrono meccanismi paragonabili a quelli delle transazioni su database [25]. Inoltre in una Data Grid sono presenti diversi tipi di dato che richiedono diversi tipi di consistenza, quindi un servizio, oltre a fornire varie politiche di replicazione, deve garantire la possibilità di usare qualsiasi metodo di memorizzazione.

In una Data Grid quindi, deve essere presente un Replica Manager (RM) che permetta agli utenti di creare, spostare, aggiornare e cancellare le repliche. I files sono copiati da un sito ad un altro e registrati in un catalogo. Quest'ultimo, insieme alle informazioni ausiliarie (per esempio gli indici) utilizzate per gestire i dati, deve essere anch'esso replicato e quindi mantenuto consistente.

Esistono due diversi approcci al problema della consistenza riassunti in altrettanti modelli.

Modello sincrono: tutte le repliche vengono aggiornate all'interno di un unica transazione.

Modello asincrono: alcune repliche per un certo periodo di tempo possono trovarsi in uno stato inconsistente.

3.1.1 Modello sincrono

Il modello di consistenza sincrono o *Eager Replication* ha come requisito fondamentale che tutte le operazioni di update sulle repliche avvengano in un unica transazione [35]. Per transazione si intende un'unità atomica di accesso a database che viene eseguita completamente o non eseguita per niente. Quindi, quando un utente modifica una replica di un dato, prima che essa sia accettata dal sistema, viene propagata a tutte le repliche dello stesso dato. In questo modo abbiamo la certezza che, in ogni momento, tutte le repliche sono perfettamente *sincronizzate*.

In un DBMS distribuito il modello sincrono può essere realizzato estendendo il meccanismo di lock a tutte le repliche. Questo metodo, però, non fornisce una soluzione molto flessibile e difficilmente può essere utilizzato da una Data Grid. Infatti, in tale ambiente, si deve avere la possibilità di specificare livelli di consistenza diversi a seconda del tipo di dato da sincronizzare: alcuni tipi di dati possono essere momentaneamente non sincronizzati (bassa consistenza) e

altri tipi devono essere sempre sincronizzati (alta consistenza).

3.1.2 Modello asincrono

Un ambiente Grid, in particolare una Data Grid, gestisce una enorme quantità di dati nell'ordine dei petabytes. Mantenere tutti questi dati sincronizzati non è affatto semplice. Il modello di consistenza asincrono o *Lazy Replication* permette una temporanea inconsistenza di alcune repliche [35]. La fase di aggiornamento della singola replica e quella di propagazione dell'aggiornamento avvengono all'interno di due transazioni separate.

Il modello asincrono permette tre diversi tipi di approcci per risolvere il problema della consistenza [43]:

Primary-copy approach. Per ogni dato esiste una sola "master replica" mentre tutte le altre repliche sono considerate copie secondarie. I cambiamenti possono essere fatti solo sulla copia primaria. Le richieste di update effettuate su una copia secondaria sono inviate alla copia primaria che fa l'update e propaga i cambiamenti a tutte le copie secondarie. Questo metodo fornisce un alto grado di consistenza dei dati e soprattutto un maggior riscontro prestazionale rispetto al modello sincrono, poiché il lock del file avviene su un unica copia (quella primaria) e non su tutte quelle presenti nel sistema.

Epidemic approach. I cambiamenti possono essere fatti su ogni replica, un processo separato confronta le versioni delle varie repliche e invia i cambiamenti alle repliche più vecchie in maniera asincrona. Gli aggiornamenti sono eseguiti sempre prima localmente e poi i vari nodi comunicano tra loro per scambiarsi le operazioni utili all'update. Il grado di consistenza dei questo metodo può essere molto basso, quindi non si possono escludere eventuali anomalie nei database.

Subscription and relatively independent sites. Simile all'Epidemic approach, la sua politica è basata sul fatto che un nodo ha come interesse principale quello di possedere il dato, e non quello di garantirne la consistenza. Rispettando questa politica, quando su un nodo si effettua un'operazione di update su un file, questa è notificata soltanto ad alcuni dei nodi della Grid. Quelli che non l'hanno ricevuta e in seguito richiederanno il file aggiornato, potranno recuperare i cambiamenti da

altri nodi della Grid precedentemente aggiornati. Questo metodo concede piu flessibilità riguardo alla consistenza dei dati e la quasi totale indipendenza dei nodi della Grid. Infatti, un nodo può decidere se e quali dati aggiornare in base alle proprie esigenze.

3.2 Un servizio di consistenza su una Grid

Le sezioni che seguono sono tratte da [21, 23] e riguardano la progettazione e l'architettura software RCS con particolare riferimento a CONStanza.

3.2.1 La consistenza in un ambiente Grid

Un generico ambiente Grid è costituito da due tipi di nodi: *Computing Element* (CE) che fornisce la capacità di computazione, *Storage Element* (SE) che fornisce la capacità di memoria. Il middleware Grid fornisce alcune importanti funzionalità come la schedulazione dei job, l'allocazione delle risorse e la gestione di dati, inclusi il *Replica Management Service* (RMS) ed il *Replica Consistency Service* (RCS).

Il RMS cerca di replicare i files in modo da migliorare il tempo e la disponibilità di accesso. Quindi, per ogni job in esecuzione su un dato CE, esso cerca di determinare quale sia la migliore replica a cui accedere in caso di bisogno.

Ogni file in una Grid è caratterizzato da un nome *logical file name* (LFN), al quale vengono associate le relative repliche. Queste ultime sono mappate negli Storage Elements. Si definisce *physical file name* (PFN) una serie di informazioni necessarie per accedere ad una data replica: queste informazioni includono l'indirizzo del SE, il pathname del suo filesystem e possibilmente un relativo protocollo di accesso (per esempio, GridFTP).

La proprietà di consistenza per un insieme di repliche, rappresentata in figura 3.1, può essere definita: *" ...si consideri una qualsiasi coppia di repliche appartenenti allo stesso file logico, il loro contenuto deve risultare uguale"* [20].

Il modello illustrato precedentemente si può estendere al concetto più generale di *dataset*. Un dataset può essere sia un *flat file* (non strutturato) che un *structured dataset* (strutturato). Il primo ha una struttura interna che, ai fini della replicazione e consistenza, puo essere trascurata. Il secondo, invece, è gestito tramite operazioni ad alto livello, (ad esemio di tipo SQL su un database relazionale) la cui struttura interna è fondamentale.

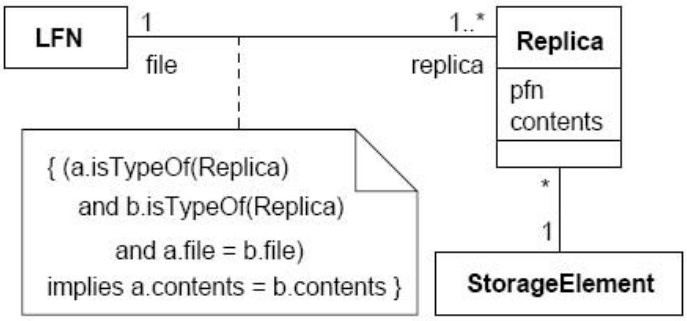

Figura 3.1: Modello UML di files e repliche

3.2.2 CONStanza e la sua architettura

Un *Replica Consistency Service* (RCS) è un servizio Grid che deve garantire la consistenza dei dati replicati. Questo servizio comprende vari schemi di replicazione, dovrebbe poter garantire vari livelli di consistenza per diverse applicazioni utente. Una architettura base per un RCS deve possedere le seguenti funzionalità:

- Un interfaccia client per il servizio.
- Un meccanismo di update dei file, responsabile di applicare gli aggiornamenti alla singola replica.
- Un protocollo di propagazione dell'update, responsabile della propagazione dell'update a tutte le repliche sulla Grid.

In questa sezione descriveremo CONStanza, un un prototipo di RCS sviluppato all'interno dell'INFNGrid project [9].

CONStanza [22] è un RCS accessibile come un Web Service, che può essere configurato in base a varie politiche di update. I componenti dell'architettura sono:

Global Replica Consistency Service (GRCS) punto di ingresso per le richieste dell'aggiornamento e componente principale dell'interfaccia verso gli end users.

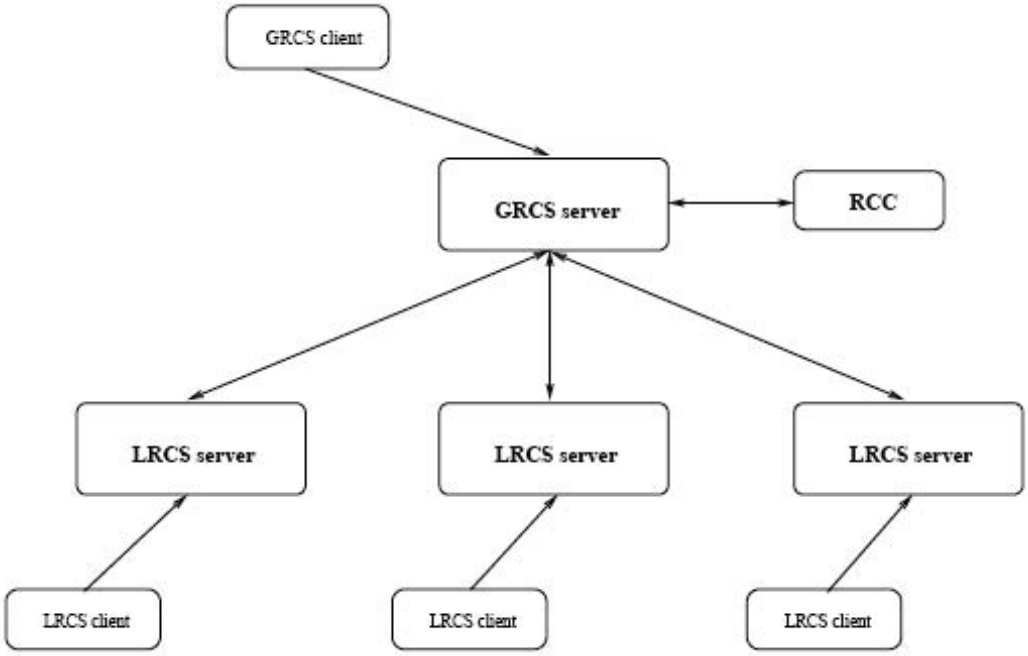

Figura 3.2: Architettura RCS CONStanza

Local Replica Consistency Service (LRCS) gestisce e controlla le repliche locali, eseguendo quando richiesto le operazioni di update. Possiede un catalogo interno per la gestione dei metadati utili all'aggiornamento della copia locale. CONStanza attualmente supporta una politica single master, cioè un unico LRCS ha la master replica che può essere aggiornata dagli utenti. Tutte le altre repliche, ossia quelle secondarie, vengono sincronizzate dall'RCS.

Replica Consistency Catalogue (RCC) memorizza le informazioni su tutte le repliche, per esempio il loro stato e dove si trovano all'interno della Grid.

La figura 3.2 mostra l'interazione tra i vari componenti.

Nello scenario più tipico, un utente deve prima di tutto aggiornare la replica master e poi, inviare il comando di update dalla linea di comando al RCS che si incarica di eseguire l'aggiornamento. Il sistema inoltre suppone che i dati

siano già replicati in alcuni nodi della Grid, la cui posizione è memorizzata nel catalogo di consistenza delle repliche (RCC). Questo è il classico approccio *single master* che permette un accesso generalizzato in lettura, ma limita e centralizza quello in scrittura.

Tuttavia, lo scenario che a noi interessa è quello del *Database Replication* (figura 3.3) in cui è previsto un ulteriore componente: il **Log File Watcher**. In pratica quello che viene svolto sulla linea di comando dall'utente dello scenario precedente, viene automaticamente eseguito da questo nuovo componente. Il Log File Watcher, quindi, controlla periodicamente se è stato effettuato un'update sulla replica master. Una volta verificato questo, esso estrae gli ultimi cambiamenti dal file di log del database e li inserisce in un *updatefile* che viene inviato a tutte le altre repliche. I cambiamenti sulle repliche secondarie vengono applicate da un ulteriore componente, il **DBUpdater**.

L'intero processo può essere schematizzato in tre fasi principali:

Estrazione del Log e notifica Il master LRCS mediante il Log File Watcher controlla periodicamente il file di log del database per rilevare eventuali aggiornamenti. In base a questi ultimi, crea un updateFile che in seguito dovrà essere utilizzato dagli slave LRCS per l'aggiornamento delle proprie repliche. Non appena l'updateFile è pronto, viene inviata una notifica al GRCS.

Update Propagation Una volta ricevuta la notifica di update, il GRCS contatta tutti gli slave LRCS che localmente possiedono una replica interessata dall'update, fornendo la posizione dell'updateFile all'interno della Grid.

Log Transfer & Application Quando uno slave LRCS riceve la notifica di un'update, esso recupera mediante GridFTP l'updateFile e, tramite il proprio DBUpdater, applica le modifiche.

3.2.3 Implementazione

Un prototipo avanzato per il RCS è stato progettato ed implementato in C++ (GCC 3.2.x su GNU/Linux), utilizzando gSOAP 2.7 per la comunicazione fra i componenti del RCS. La figura 3.4 mostra i principali sottosistemi del servizio.

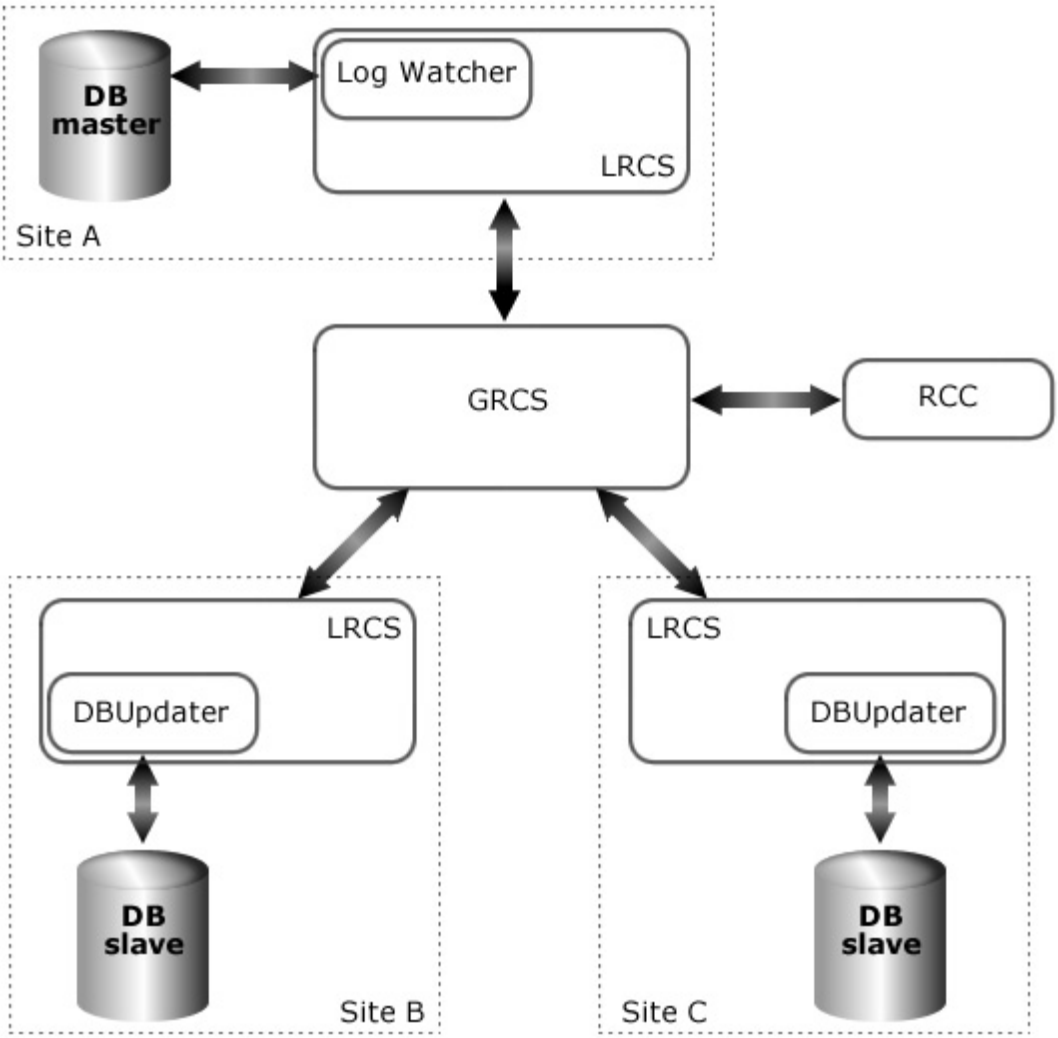

Figura 3.3: Architettura CONStanza in un Database scenario

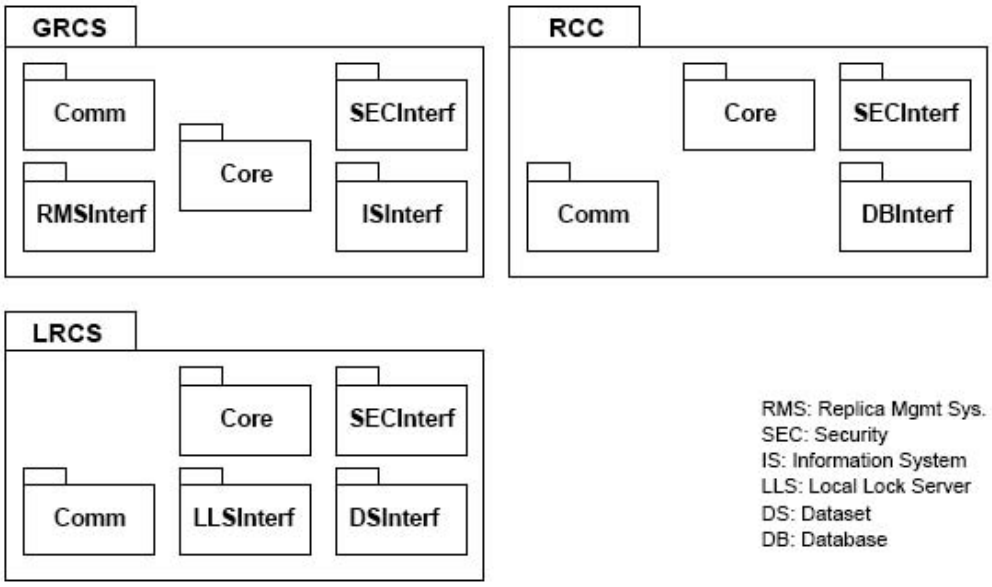

Figura 3.4: Sottosistemi software di CONStanza

Ogni sottosistema ha un package **Core** in cui sono implementate le funzionalità di sottosistema, e un package **Comm** che supporta sia la comunicazione fra i sottosistemi che fra il RCS ed i relativi client. Il modulo **DSInterf** applica gli aggiornamenti ai differenti tipi di dataset.

Nel proseguo di questo lavoro, modificheremo il package core di GRCS e LRCS per meglio adattarlo ai nostri scopi. La modifica comunque sarà minima, preservando in questo modo l'architettura originaria dell'intero sistema.

Capitolo 4

OptorSim: Simulazione dinamica di una Grid

In questo capitolo analizzeremo il funzionamento di una Grid simulata. Inoltre cercheremo di capire come sfruttare questa simulazione per studiare gli algoritmi di replicazione dinamica.

4.1 Ambiente di simulazione

Un' architettura semplificata di Griglia può essere vista come un insieme di siti ognuno dei quali contiene al massimo una unità Computing Element (CE) ed al più una unità Storage Element (SE). Questa architettura è descritta in figura 4.1.

Il Resource Broker (RB) è il componente che si incarica di schedulare i job forniti dall'applicazione utente su un CE. Il job in esecuzione su un CE, nella cui descrizione è presente una lista di "Logical File Name" (LFN) usata per effettuare le proprie elaborazioni, accederà ai dati (file) del proprio SE o di un SE remoto.

Il Replica Manager (RM) si incarica di gestire l'accesso ai dati, e tramite il Replica Optimizer decide quando e dove creare o rimuovere repliche. Il Replica Catalog è usato per richiedere informazioni sulle repliche, ed è implementato tramite una tabella di LFN e PFN ("Physical File Name") corrispondenti. Ricevuta la richiesta di esecuzione di un job tramite l'interfaccia di programmazione, il RB cerca un CE idle con caratteristiche opportune (frequenza di esecuzione dei job, tipi di job che può eseguire) e gli invia la richiesta di esecuzione del job.

Quando un job richiede l'accesso ad un file, il LFN viene usato per reperire,

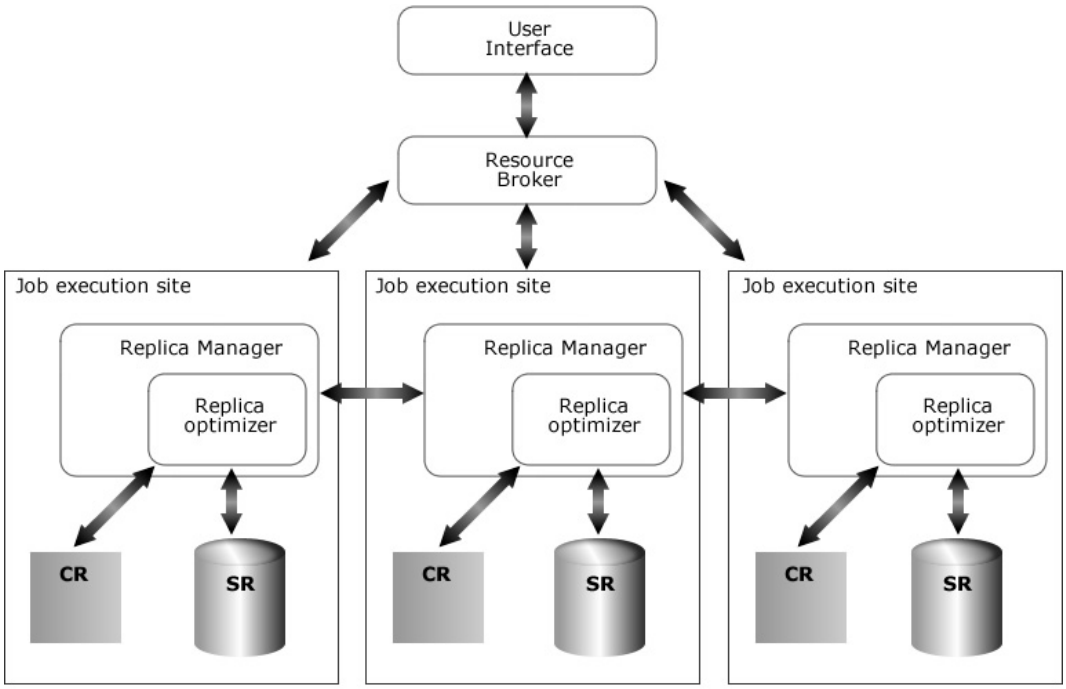

Figura 4.1: Architettura semplificata di una Grid

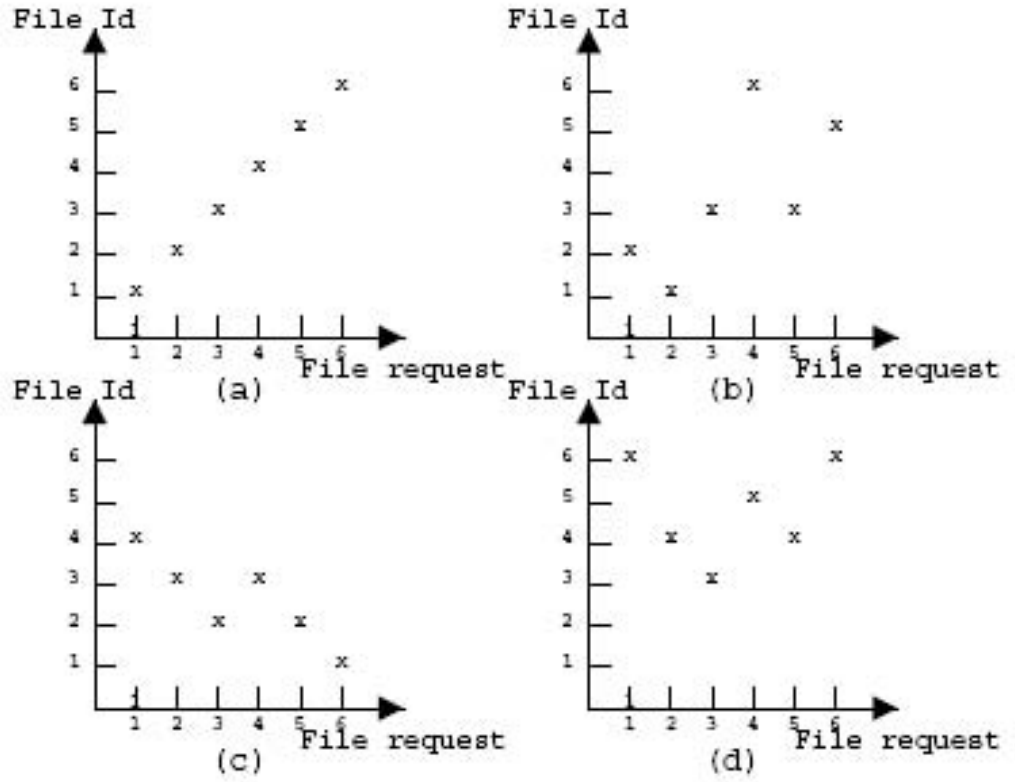

Figura 4.2: Esempi di access patterns: (a) sequenziale, (b) random, (c) random a passo unitario, (d) random a passo Gaussiano

tramite la funzione di ottimizzazione *getBestFile()*, la replica migliore (best replica).

L'ordine di accesso alla lista di LFN può essere scelto tra vari modelli: sequenziale , random, random a passo unitario e random a passo Gaussiano (figura 4.2).

La *getBestFile()* è una chiamata che può bloccare l'esecuzione del job. Questa funzione seleziona la best replica e decide se effettuare una copia sullo SE locale, oppure se conviene eseguire delle operazioni di I/O remoto. Attualmente OptorSim simula soltanto la parte del Job che riguarda l'accesso ai dati e non quella relativa alla elaborazione degli stessi.

4.2 Algoritmi di ottimizzazione delle repliche

Gli algoritmi di ottimizzazione servono per migliorare l'accesso ai dati da parte dei job. Essi si distinguono principalmente in base alla scelta relativa alla creazione di nuove repliche. Infatti, dopo la chiama al getBestFile(), viene deciso di creare una replica di un dato file sullo SE locale. Può succedere, tuttavia, che lo SE sia pieno, di conseguenza una replica dovrà esser rimossa per far posto alla nuova.

OptorSim utilizza a tal fine alcuni di questi algoritmi:

no replication: il posizionamento delle repliche viene deciso all'inizio e rimane invariato per tutto il corso della simulazione. Nell'ipotesi in cui la richiesta di accesso sia relativa ad un file la cui best replica si trova su un SE remoto, a questa si accederà in maniera remota, senza effettuare nessuna copia locale.

unconditional replication: questo algoritmo esegue sempre la replicazione del file nel sito in cui verrà elaborato dal job. Ne esistono due varianti:

- **oldest file deleted**, la replica che eventualmente dovrà essere rimossa sarà la più vecchia all'interno dello SE
- **least accessed file deleted** la replica rimossa sarà invece quella a cui si è eseguito un accesso da più tempo;

economic model: qui i file sono visti come dei beni di consumo. Questi vengono comprati dai CE per svolgere i loro job, e dagli SE per fare degli investimenti. L'obiettivo dei CE è quello di minimizzare i costi, quello degli SE di massimizzare il profitto.

4.3 Files di configurazione

In OptorSim si fa uso principalmente di tre file di configurazione:

1. Il primo è il "file di configurazione della Grid", all'interno del quale i dati sono organizzati in una matrice. Ogni riga contiene: il numero di worker nodes, il numero di SE e la sua grandezza. Se queste prime tre colonne contengono solo zeri allora la riga corrisponde ad un nodo di tipo network router.

2. Il secondo è il "file di configurazione dei job", dove troviamo informazioni relative ai file presenti nella Grid e ai job che verranno eseguiti durante

la simulazione. Esso è logicamente suddiviso in quattro parti. La prima parte riguarda i file presenti nella Grid: nome (LFN unico),grandezza (in MB) e un indice che viene usato nell'economic model e che indica la correlazione tra file e job. Nella seconda parte sono indicati quali LFN dei file vengono richiesti da ogni job. La terza parte indica su quali nodi verranno eseguiti i vari job. La quarta riguarda invece la probabilità di esecuzione di ogni job.

3. Il terzo file di configurazione, `parameters.conf`, è quello principale che, oltre a contenere l'indicazione dei precedenti due file, include:

   - il numero di job complessivamente eseguiti, il tipo di politica adottata dallo scheduler, scelto in modo casuale o tramite una chiamata alla getAccessCosts();
   - il numero di accessi ai file eseguiti dai job;
   - altri parametri come la grandezza della coda dei job, tempo impiegato dai CE per elaborare un file;
   - la modalità di visualizzazione dei risultati.

Si rimanda alla guida [18] del programma per la descrizione dettagliata di tutti i parametri. Si riportano di seguito alcuni esempi di file di configurazione.

File simple_grid.conf

```
# A simple network configuration based on 10 sites, two of which
have CEs.
# no of CEs, no of SEs, SE sizes, site vs site bandwidth
#
 0 1 10000 0. 0. 1000. 0. 0. 0. 0. 0. 0. 0.
 0 1 10000 0. 0. 1000. 0. 0. 0. 0. 0. 0. 0.
 1 1 10000 1000. 1000. 0. 1000. 1000. 0. 0. 0. 0. 0.
 0 1 10000 0. 0. 1000. 0. 0. 0. 1000. 0. 0. 0.
 0 1 10000 0. 0. 1000. 0. 0. 0. 0. 0. 0. 0.
 0 1 10000 0. 0. 0. 0. 0. 0. 0. 1000. 0. 0.
 0 1 10000 0. 0. 0. 1000. 0. 0. 0. 1000. 0. 0.
 1 1 10000 0. 0. 0. 0. 0. 1000. 1000. 0. 1000. 1000.
 0 1 10000 0. 0. 0. 0. 0. 0. 0. 1000. 0. 0.
 0 1 10000 0. 0. 0. 0. 0. 0. 0. 1000. 0. 0.
```

File simple_job.conf

```
# Simple jobs for the 2 CEs in the simple network
#
# File Table
#
\begin{filetable}
File1 1000 1
File2 1000 2
File3 1000 3
File4 1000 4
File5 1000 5
File6 1000 6
File7 1000 7
File8 1000 8
File9 1000 9
DBmaster 1000 10
\end
#
# Job Table
# A job name and a list of files needed.
#
\begin{jobtable}
job1 File1 File2 File3 File4 File5 File6 File7 File8
job2 File9
LFWatch DBmaster
\end
#
# CE Schedule Table
# CE site id, jobs it will run
#
\begin{cescheduletable}
2 job1 LFWatch
7 job2
\end
#
# The probability each job runs
#
\begin{jobselectionprobability}
job1 0.8
job2 0.2
```

```
LFWatch 0.3
\end{jobselectionprobability}
```

File parameters.conf

```
#
# This file contains all the parameters required for OptorSim
# using the Properties class to store this information.
#
grid.configuration.file = examples/simple_grid.conf
job.configuration.file = examples/simple_job.conf

number.jobs = 10
#
# The choice of the scheduling algorithms for the RB is:
# (1) random
# (2) queue length
# (3) access cost for current job
# (4) access cost for current job + all queued jobs
scheduler = 4
#
# The categories of users available are:
# (1) Simple - wait job delay between submitting jobs
# (2) Random - wait uniform random time between 0 and 2 * job delay
# (3) CMS DC04 Users
users = 1
#
# The choice of optimisers is:
# (1) SimpleOptimiser - no replication.
# (2) LruOptimiser - always replicates, deleting least recently
#        created file.
# (3) LfuOptimiser - always replicates, deleting least frequently
#        accessed file.
# (4) EcoModelOptimiser - replicates when eco-model says yes, deleting
#        least valuable file.
# (5) EcoModelOptimizer Zipf-like distribution
optimiser = 3
#
# automatically multiplied by scale factor
```

```
#
dt = 1000000
#
# The choice of access pattern generators is:
# (1) SequentialAccessGenerator - Files are accessed in order.
# (2) RandomAccessGenerator - Files are accessed using a flat random
#       distribution.
# (3) RandomWalkUnitaryAccessGenerator - Files are accessed using a
#       unitary random walk.
# (4) RandomWalkGaussianAccessGenerator - Files are accessed using a
#       Gaussian random walk.
# (5) RandomZipfAccessGenerator - Files are accessed using a
#        Zipf distribution
#
access.pattern.generator = 1
# Shape parameter for Zipf-like distribution > 0
shape = 0.85
#
# The job set fraction is the number of files accessed per job.
#
job.set.fraction = 1
#
# The initial file distribution is either "random" in which case the
# master files are distributed randomly to SEs or numbers separated
# commas (e.g. 8,10,15) giving the sites where all the masters
# should be evenly distributed.
# (in edg testbed site 8 is CERN)
#
initial.file.distribution = 2
#
# Do we want to fill all the SEs with replicas?
#
fill.all.sites = no
#
# The scale factor is used to speed up the simulation by reducing the
# file sizes. If it is too small the timing will be inaccurate.
#
scale.factor = 1
#
# The job delay is the interval in ms between the RB submitting each job.
```

```
# Automatically multiplied by scale factor
#
job.delay = 25000
#
# The random seed for deciding which jobs are chosen can be random or
# fixed.
#
random.seed = no
#
# The maximum queue size is the maximum number of jobs the JobHandler
# will keep in its queue.
#
max.queue.size = 200
#
# The time (in ms) it takes each file to be processed (this is
# divided by the number of worker nodes on the site.)
# Automatically multiplied by scale factor
#
file.process.time = 100000
#
##################
# Auction stuff #
##################
#
auction.flag = no
# actually the number of sites contacted
hop.count = 50
timeout = 500
timeout.reduction.factor = 0.4
#
# Outputs auction information to auction.log. Can slow the
# simulation down a little bit.
#
auction.log = yes
#
#############################
# BandwidthReader stuff     #
#############################
# flag to switch background traffic on or off
background.bandwidth = no
```

```
#
# The directory in which your background bandwidth data files are stored
data.directory = examples/bw_data/edg_testbed/
#
# The datafile to use when no other background data are available.
# For EDG, lyon_to_cern_ave.numbers is good; for UK dl_to_ncl_ave.numbers
is good. default.background = lyon_to_cern_ave.numbers
#
# The time of day used as starting point. Should be in hours, with minutes after
# a the decimal point e.g. 22.5 for 22:30, and must be on the hour or half-hour.
time.of.day = 0.0
#
##############
# GUI stuff #
##############
#
# Options to use the GUI and histogram browser
#
gui = no
histogram.browser = no
#
# The file with the map information
#
map.info = examples/gui/europe.coords
#
###############
# Statistics #
###############
#
# Level of statistics to be printed out at the end of the simulation
# (1) None
# (2) Simple - only stats for the whole grid
# (3) Full - full stats for all elements on all sites
#
statistics = 3
#
###############
# Time Model #
###############
#
```

```
# use advanced grid time (yes) or not (no)
#
time.advance = yes
#
# end
#
```

4.4 La Graphical User Interface

OptorSim possiede una Graphical User Interface (GUI), illustrata in figura 4.3 che permette all'utente di controllare la simulazione e visualizzare i risultati. Se il parametro **gui** nel file di configurazione e settato a `yes` l'intera simulazione può essere visualizzata mediante una GUI window formata da tre pannelli:

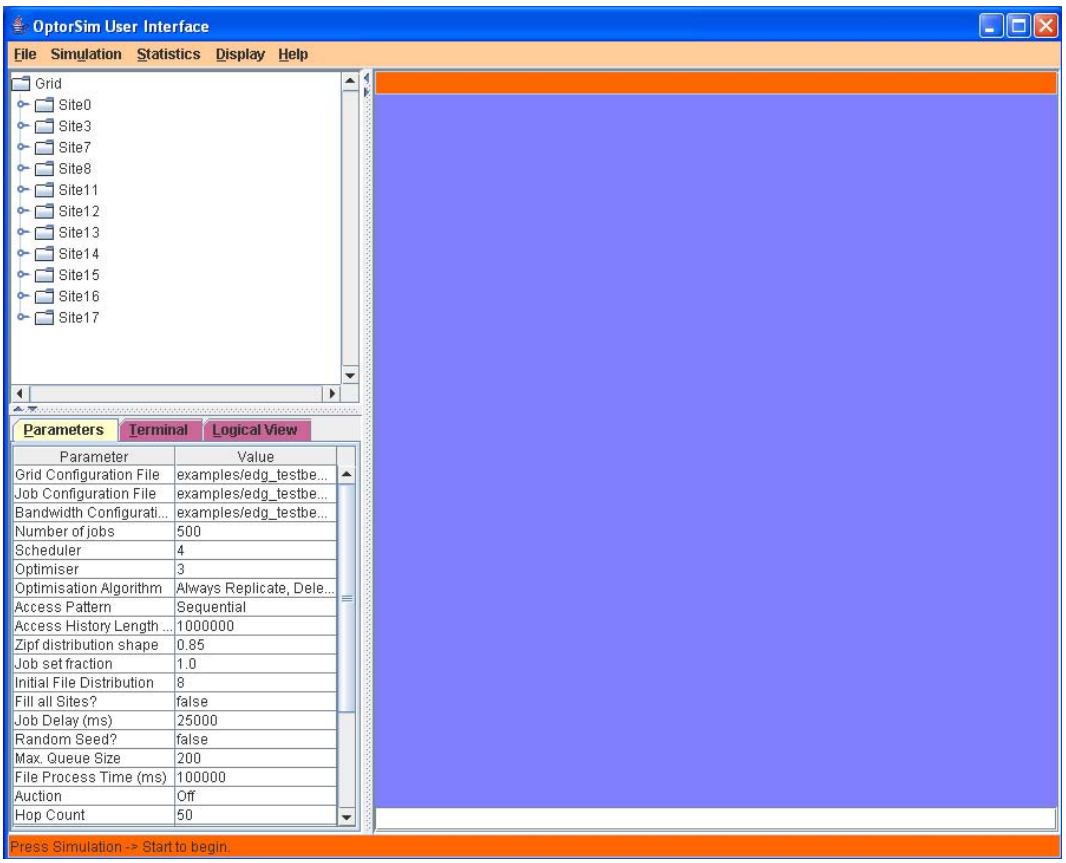

Figura 4.3: Screenshot della GUI di Optorsim

Hierarchy pane. Fornisce una visione gerarchica della Grid visualizzando i sottolivelli CE/SE.

Statistics pane. Contiene un insieme di tabelle che sintetizzano con dei grafici tutti i parametri statistici relativi a vari nodi. Inoltre contiene una sintesi statistica del corrente stato della simulazione.

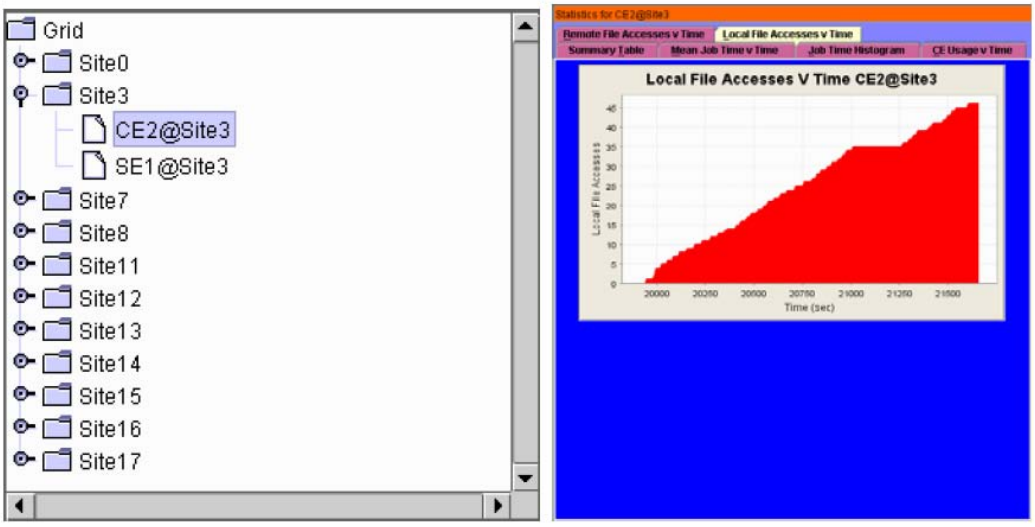

Figura 4.4: Hierarchy pane (sinistra)e Statistics pane (destra)

Information pane. Contiene tre tabelle:

- **Paremeters**, che mostra i parametri in base ai quali verra svolta la simulazione.
- **Terminal**, che mostra lo standard output della simulazione.
- **Logical View**, che mostra tutti i trasferimenti dei file da un nodo all'altro.

4.5 Output del simulatore

OptorSim fornisce le statistiche in base alle opzioni specificate nel file di configurazione. Se si sceglie la modalità `full` vengono rappresentati tutti i parametri statistici a disposizione della simulazione. Questi parametri vengono divisi in quattro categorie ad ognuna delle quali è associato un elemento della simulazione:

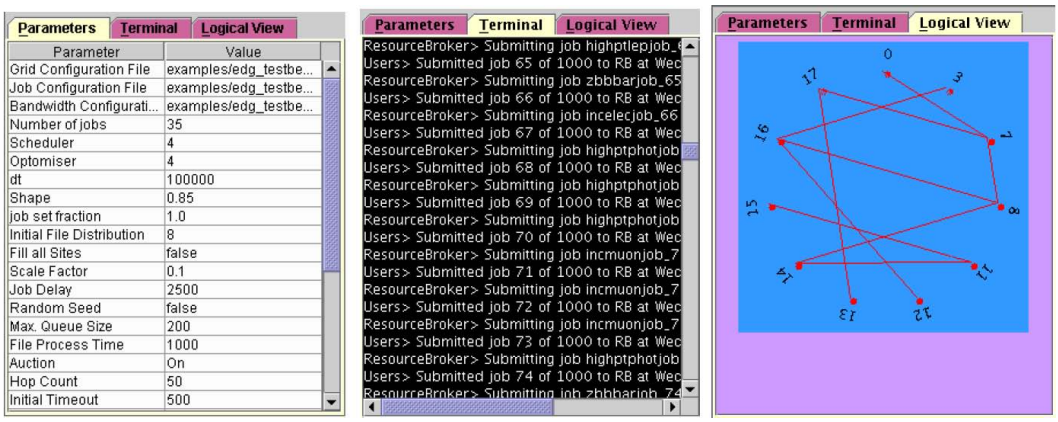

Figura 4.5: Paremeters (sinistra), Terminal (centro) e Logical View (destra)

- **Whole Grid**, contiene le statistiche della Grid vista nella sua totalità;
- **Grid Sites**, contiene le statistiche reletive al singolo nodo della Grid;
- **Storage Elements**, contiene le statistiche relative ai SE;
- **Computing Elements**, contiene le statistiche relative ai CE;

La descrizione dettagliata degli output di OptorSim è contenuta nella guida [18].

Capitolo 5

Descrizione del simulatore

In questo capitolo illustreremo le fasi della creazione di un simulatore per il servizio di consistenza derivato dall'integrazione tra i due software descritti nei capitoli precedenti: CONStanza e OptorSim.

Descriveremo inoltre in che modo essi interagiscono al fine di ottenere una simulazione coerente di un reale servizio di consistenza delle repliche.

5.1 Introduzione

Un aspetto fondamentale delle Data Grid è quello relativo alla gestione dei dati replicati. Il meccanismo di replicazione dei dati, infatti, ha come effetto un aumento notevole delle prestazioni, oltre a una maggiore tolleranza ai guasti.

In un ambiente Grid, ma in generale in qualsiasi ambiente distribuito, esistono delle applicazioni che per il loro funzionamento richiedono prima una modifica di un dato esistente e poi, il conseguente aggiornamento della sue repliche. Si presenta così il problema della consistenza delle repliche, che deve essere necessariamente risolto da un servizio di consistenza che si occupi di tale aspetto.

CONStanza è uno dei pochi software (se non l'unico) che attualmente si occupa di fornire un servizio di consistenza in un ambiente Grid. Il servizio, essendo un prototipo implementato per una piccola Grid reale (pochi nodi), è particolarmente soggetto ai vincoli derivanti dalla topologia della Grid su cui sarà in esecuzione. Una Grid reale con pochi nodi, infatti, se da un lato permette di testare l'effettivo funzionamento del servizio su esempi reali, dall'altro è limitante soprattutto per quanto riguarda i test relativi alla scalabilità del servizio stesso. Per sopperire a questa carenza si è pensato di integrare CONStanza con un simulatore Grid già esistente, in modo da poter variare

le configurazioni della Grid su cui andrà in esecuzione, e per permettere test sulla scalabilità del software.

Il simulatore Grid che meglio si adatta alle nostre esigenze è OptorSim. Questo simulatore, infatti, è quello che possiede la migliore gestione delle repliche tra quelli presenti attualmente nel panorama Grid. Esso ci permette flessibilità sia dal punto di vista topologico, che dal punto di vista delle strategie di replicazione dei dati, il tutto finalizzato alla massimizzazione del job throughput.

Lo scopo, quindi, è quello di ottenere un simulatore che ci permetta di effettuare test sulla scalabilità del servizio di consistenza e verificare nel contempo l'efficienza dei vari metodi di replicazione.

5.2 CONStanza e il servizio di consistenza

In questa sezione descriveremo le modifiche effettuate al servizio di consistenza CONStanza, affinché esso possa essere integrato con il simulatore Grid OptorSim. Queste modifiche sono mirate a non stravolgere completamente il software, anzi, a mantenerlo il più possibile conforme all'originale, in modo da essere preservata la sua efficienza in un ambiente Grid reale.

5.2.1 Architettura

In breve, riferendoci al capitolo 3, i componenti dell'architettura di CONStanza sono:

- Global Replica Consistency Service (GRCS)
- Local Replica Consistency Service (LRCS)
- Replica Consistency Catalog (RCC)

CONStanza è stato progettato per essere un Replica Consistency Service (RCS) accessibile come Web Service. In questo contesto i primi due componenti svolgono un ruolo fondamentale: quello di essere dei veri e propri server, in esecuzione in diversi nodi della Grid. Essi permettono sia la comunicazione tra i vari nodi, che l'implementazione delle differenti politiche di update delle repliche. Poiché CONStanza è un RCS implementato per una Grid reale, GRCS e LRCS sono visti come processi in esecuzione su diversi nodi.

Lo scenario a cui faremo riferimento, anch'esso dettagliatamente esposto nel capitolo 3, è quello di replicazione di database, il cui processo di sincronizzazione può essere suddiviso in tre fasi:

- Estrazione del Log e notifica
- Update Propagation
- Log Transfer & Application

Questo tipo di architettura, realizzata per funzionare su una Grid reale, oltre a dover essere adattata ad una Grid simulata fornita da OptorSim, deve rispettare una caratteristica fondamentale: la scalabilità. Si deve distinguere però, tra "scalabilità del servizio di consistenza" e "scalabilità del simulatore del servizio di consistenza".

5.2.2 La scalabilità del servizio di consistenza

Per un servizio di consistenza, la scalabilità può essere vista come la capacità del servizio stesso di mantenere adeguati livelli di *Quality of Service* (QoS) al variare del numero delle repliche da gestire. Il concetto di Qualità del Servizio è ampio e viene utilizzato in numerosi campi applicativi molto differenti fra loro: valutazione dell' usabilità delle interfacce grafiche, definizione di standard qualitativi per le prestazioni delle reti, prestazioni di processori. In ognuno di questi settori esiste un concetto preciso di QoS.

In generale possiamo dire che le specifiche di QoS condividono le seguenti caratteristiche:

- Dipendono dal dominio applicativo.
- Possiedono un formato appositamente progettato tenendo conto del particolare impiego.
- Necessitano di essere tradotte a livello applicativo dal linguaggio, a livello di sistema in parametri di QoS interpretabili dal middleware sottostante.

Ecco alcune definizioni di qualità del servizio presenti in letteratura:

"*La qualità del servizio è la prestazione del sistema percepita e le impressioni di utilizzo dell'applicazione dal punto di vista dell'utente* [33]"

"*Il QoS è il grado di conformità del servizio fornito rispetto all'accordo stipulato fra l'utente ed il fornitore dello stesso* [27]"

Queste definizioni sono del tutto generali e non danno indicazioni su come ottenere la qualità desiderata, ma pongono l'attenzione sul fatto che la qualità è qualcosa che deve poter essere percepita e misurata con le metriche specifiche del dominio di applicazione. A livello utente vengono valutati principalmente gli aspetti direttamente percepibili delle applicazioni (es. tempo di risposta, usabilità etc.), a tutti i livelli sottostanti, i criteri di valutazione della qualità del servizio sono specifici delle risorse coinvolte.

In [37], per esempio, si fa riferimento a un modello QoS per la consistenza, dove si indicano alcuni parametri di cui si deve tener conto:

- Timeliness of response;
- Consistency of delivered data;

Entrambi i parametri sono particolamente utili per stabilire i tradeoffs per un servizio di consistenza.

5.2.3 La scalabilità del simulatore

Una workstation possiede risorse computazionali limitate, come ad esempio memoria e tempo di elaborazione, che spesso limitano gli elementi che i progettisti possono simulare. Il termine scalabilità si riferisce in generale *alla capacità di un sistema di "crescere" o "decrescere" (aumentare o diminuire di scala) in funzione delle necessità e delle disponibilità*[1].

Nel nostro caso per scalabilità si intende la possibilita di eseguire a parità di potenza con lo stesso software simulazioni sempre più grosse (aumentare il numero dei nodi).

Considerando che la simulazione avvenga su un unica workstation, implementare GRCS e i vari LRCS come processi rappresenta una soluzione infelice. Infatti, se per GRCS basta un solo processo, il numero dei processi relativi ai LRCS è riconducibile al numero di repliche che devono essere aggiornate durante l'operazione di update. In generale questo numero potrebbe essere elevatissimo. Avviare tanti processi quante sono le repliche porterebbe inevitabilmente ad una immediata saturazione delle risorse del sistema. Per questo motivo è stata modificata l'implementazione dei server, trasformando i processi originali in thread, che notoriamente sono meno dispendiosi dal punto di vista delle risorse.

[1] http://it.wikipedia.org/

Quindi il vantaggio della nuova implementazione riguarda soprattutto la rappresentazione dei LRCS che, di conseguenza, necessitano per la loro elaborazione di una minore quantità di risorse.

5.2.4 La sincronizzazione dei database

Il processo di sincronizzazione dei database, come descritto precedentemente, si basa su l'interazione tra RCS e la Grid sulla quale viene eseguito. Nel nostro caso la Grid in questione è quella fornita e simulata da OptorSim. Di conseguenza, tutte le operazioni di trasferimento di informazioni e di dati che vengono effettuate da CONStanza, devono avere per la loro esecuzione un corrispettivo su OptorSim. Ciò implica una modifica di OptorSim (che verrà trattata in seguito) e una suddivisione dei compiti da parte dei due software nel processo di sincronizzazione. Quest'ultimo può essere diviso in varie parti:

1. Estrazione del Log e notifica;
2. Update Propagation;
3. Log Transfer;
4. Log Application;

I punti 2 e 4 sono di competenza di CONStanza, il resto, invece, di OptorSim. Si noti che le operazioni che deve effettuare CONStanza sono tra loro disgiunte e, quindi, possono essere eseguite singolarmente sul server corrispondente. Infatti, sia "Update Propagation" che la "Log Application", eseguite rispettivamente su GRCS e LRCS, potrebbero essere considerate come delle operazioni che un qualsiasi client del sevizio possa utilizzare.

5.3 OptorSim e la simulazione della Grid

Da quanto visto precedentemente, il compito di OptorSim è quello di fornire una Grid simulata su cui far girare il servizio di consistenza. Inoltre, nello scenario specifico della "sincronizzazione dei database", esso si incarica di simulare due funzionalità:

- Estrazione del Log e notifica;
- Log Transfer;

Ognuna di queste funzionalità deve essere implementata direttamente in OptorSim, poiché si trova ad interagire direttamente con la simulazione della Grid.

Nella simulazione ogni Computing Elements (CE) è rappresentato da un thread. La sottomissione dei job ad un CE è gestita da un altro thread: il Resource Broker (RB). Un diagramma del flusso di esecuzione è descritto in figura 5.1.

Il RB si assicura che su ogni CE sia in esecuzione un job, cercando sempre di bilanciare la distribuzione dei job su tutti i CE. Non appena il RB trova un CE in stato "idle", seleziona un job da mandare in esecuzione secondo la politica del CE. Su un CE può girare al più un job e, non appena finito quello in esecuzione, il RB gliene assegnerà un altro.

5.3.1 Estrazione del Log e notifica

Nella descrizione del servizio di consistenza (Capitolo 3), nel caso di database scenario, si è introdotto un nuovo componente il **Log File Watcher**, il cui compito è:

- Controllo periodico della master replica;
- Creazione dell'updateFile;
- Notifica dell'operazione di update;

L'implementazione di questo componente in OptorSim è stata fatta in modo da rispettare tutti e tre i punti sopra elencati. Il Log File Watcher, in questa implementazione, viene rappresentato come un particolare job che va in esecuzione periodicamente su un CE prestabilito. Per comprendere meglio il suo funzionamento, partiremo dalla una descrizione generica di un job per poi evidenziare le differenze.

Il RB invia i job ai vari CE secondo la politica specificata dal parametro "`users`" nel file di configurazione generale (`parameter.conf`). In base ad un ulteriore parametro, il "`job.delay`" stabilisce la scansione temporale in base alla quale avviene la schedulazione dei job, ossia ogni quanto tempo un job viene schedulato. Le caratteristiche di ogni job sono contenute nel file di configurazione dei job, che rappresenta anch'esso un parametro della simulazione.

Il Log File Watcher può essere rappresentato quasi come un generico job eseguito su un unico nodo (su cui è posizionato il LRCS master) e che per la

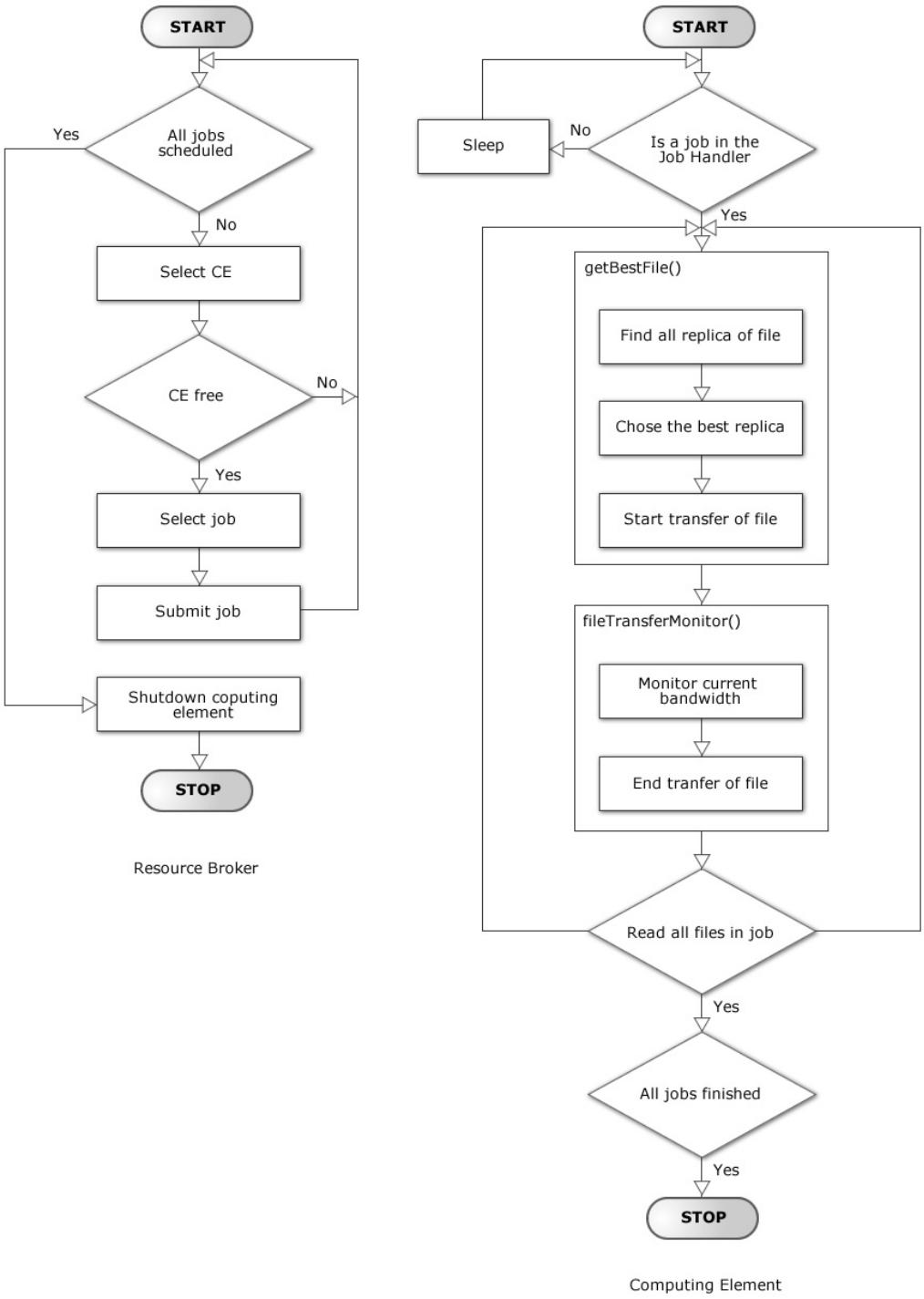

Figura 5.1: Diagramma del flusso di esecuzione di RB e CE

sua esecuzione richiede un solo e unico file (updateFile). La differenza rispetto ad un normale job sta nella sua probabilità di esecuzione.

Infatti, il Log File Watcher non ha una probabilità di esecuzione ma, al contrario, viene eseguito sempre con un intervallo fisso di " **N *** `job.delay` ", dove **N** indica il numero di job in esecuzione sul CE.

Quindi il numero che appare accanto al Log File Watcher (LFWatch) nel file di configurazione dei job non rappresenta la probabilità di esecuzione, bensì la *"probabilità che sia avvenuta una scrittura sulla master replica"*.

porzione del file di configurazione dei job

```
#
# The probability each job runs
#
\begin{jobselectionprobability}
job1 0.8
job2 0.2
LFWatch 0.3
\end{jobselectionprobability}
```

OptorSim permette l'accesso ai files solo in lettura e non in scrittura. Ma, affinché un servizio di consistenza possa aver senso, sarebbe necessario l'acceso ai files anche in scrittura. Per ovviare al problema si è pensato di considerare che la scrittura e la conseguente "Estrazione del Log" avvenga al di fuori del simulatore, e che il file a cui deve accedere il Log File Watcher sia in realtà updateFile già costruito. Il vantaggio è di avere un file pronto a essere trasferito di cui possiamo anche impostare la dimensione.

Quindi, quando il Log File Watcher va in esecuzione, simula (mediante una distribuzione di probabilità uniforme) un avvenuto acceso in scrittura. A questo punto esso invia la notifica di update ad un server **"RCSim"**, che dopo una comunicazione veloce con CONStanza avvia l'operazione di "update propagation".

5.3.2 Il server RCSim

RCSim è un server multithread per la gestione della notifica dell'update effettuata dal Log File Watcher. Per ogni notifica che riceve, esso avvia un thread "updateDB" il quale si occupa di effettuare la comunicazione con CONStanza e predisporre i parametri per l'update propagation.

La comunicazione con CONStanza avviene tramite la sua interfaccia Web service, mediante un metodo java che invoca la chiamata di update su GRCS. Una volta avviato l'update, si aspetta che l'operazione vada a buon fine, per poi determinare quali siano i nodi della Grid su cui girano gli LRCS secondari. Questi nodi sono indicati nel file di configurazione dei job in maniera particolare.

porzione del file di configurazione dei job

```
#
# CE Schedule Table
# CE site id, jobs it will run
#
\begin{cescheduletable}
0  highptphotjob zbbbarjob
3  highptphotjob zbbbarjob LRCS
7  highptphotjob zbbbarjob LFWatch
11 highptphotjob zbbbarjob
12 highptphotjob zbbbarjob LRCS
13 incelecjob incmuonjob highptphotjob zbbbarjob
14 incelecjob incmuonjob highptphotjob zbbbarjob
15 incmuonjob highptphotjob zbbbarjob
16 highptphotjob zbbbarjob
17 incmuonjob highptphotjob zbbbarjob
\end
```

Come si può notare sul nodo 3 e sul nodo 12 è presente un job `LRCS`.

Esso non è un vero e proprio job, poiché in realtà rappresenta un modo per impostare come LRCS secondari i corrispondenti nodi della Grid. Si noti, inoltre, che la presenza del job `LFWatch` indica su quale nodo gira il GRCS e quindi LRCS primario. Così, in maniera molto intuitiva, si riesce a stabilire, e di conseguenza impostare, l'ubicazione di GRCS e dei vari LRCS all'interno della Grid.

Stabiliti gli LRCS secondari, mediante un opportuna struttura di sincronizzazione, si segnala ai nodi interessati che, non appena possibile, c'è una scrittura pendente da effettuare. A questo punto sarà compito del nodo richiedere l'updateFile per effettuare l'aggiornamento.

5.3.3 Log Transfer

L'aspetto relativo al Log Transfer riguarda il trasferimento dell'updateFile da un nodo all'altro della Grid. Come descritto precedentemente, una volta che un nodo su cui gira un LRCS secondario si accorge che ci sono delle scritture pendenti deve avviare il trasferimento dell'updateFile dal sito principale. Di conseguenza, ogni volta che un CE relativo a un LRCS secondario ritorna nello stato idle, verifica se ci sono scritture pendenti e, prima di elaborare il job successivo, effettua il trasferimento.

Il trasferimento viene effettuato mediante un metodo messo a disposizione nativamente da OptorSim:

```
simulateRemoteIO( DataFile remoteFile, float fraction).
```

Questo metodo simula il trasferimento di un file (`remoteFile`), o frazione del file stesso (`fraction`), dal nodo in cui si trova al nodo che ha effettuato la richiesta. Esso, essendo implementato nativamente, è soggetto a tutti i rilevamenti statistici effettuati da OptorSim, con il vantaggio di ottenere output molto precisi.

Capitolo 6

Architettura del simulatore

6.1 Comunicazione tra OptorSim e CONStanza

CONStanza è un prototipo di Replica Consistency Service, accessibile come Web Service, che gestisce la consistenza di dati replicati in un ambiente Grid. In generale un Web Service gestisce le cosiddette interazioni program-to-program, implementando un modello comune di comunicazione da programma a programma basato su standard esistenti come HTTP, Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI). La sua funzione è permettere alle applicazioni un'integrazione più rapida e facile e soprattutto in maniera meno costosa. Un Web Service può essere definito come un'interfaccia che descrive una serie di operazioni, accessibili attraverso una rete mediante messaggi XML. Alla base della comunicazione dei Web Service c'è SOAP, un protocollo basato su XML.

SOAP è un meccanismo semplice e leggero per lo scambio di dati strutturati fra applicazioni di rete. Il protocollo SOAP definisce la struttura dei messaggi, l'insieme delle regole di codifica per esprimere istanze di tipi di dati, e la convenzione per rappresentare le chiamate a procedure remote (RPCs) e le risposte. SOAP può essere implementato su altri protocolli di rete quali HTTP, SMTP, FTP e altri.

CONStanza è realizzato utilizzando il toolkit gSOAP basato sul protocollo SOAP. Il toolkit di sviluppo per Web Service gSOAP [8] fornisce un collegamento fra XML e linguaggio C/C++ che facilita lo sviluppo di Web Service SOAP/XML in C/C++ ed anche lo sviluppo d'applicazioni client di un Web Service in C/C++. Questo toolkit si basa su tecnologie di compilazione che fanno corrispondere messaggi XML a definizioni C/C++, in modo che l'inter-

operabilità SOAP/XML sia ottenuta con un'API semplice che permette allo sviluppatore di concentrarsi sulla logica dell'applicazione.

Un client di CONStanza ha a disposizione delle chiamate che, tramite protocollo SOAP, interagiscono con il servizio di consistenza. Seguendo questa linea, per implementare la comunicazione tra CONStanza e OptorSim, si sono utilizzate chiamate già implementate nell'interfaccia client nel caso di GRCS server e se ne sono aggiunte di nuove nel caso di LRCS server.

OptorSim comunica direttamente con CONStanza in due casi specifici: Notifica dell'update e Log application.

Nel primo caso OptorSim effettua una chiamata ad una funzione client di GRCS già esistente.

```
rcs-update-db [-h <host> -p <port> -v <vo> -d] -l <lfn> -u <url>
-q <quorum> -b
```

Questa funzione avvia l'operazione di update sul GRCS server. Tenendo conto che ci troviamo in un ambiente localhost, gli argomenti più importanti della funzione sono quelli relativi alle opzioni -u -b. La prima opzione ha come argomento l'URL del' updateFile che deve essere trasferito in locale. Originariamente esso doveva contenere il prefisso "gridftp://" ma, come vedremo in seguito, non ce ne sarà più bisogno poiché si farà riferimento al filesystem locale. L'opzione -b indica se la procedura è bloccante oppure no.

Nel secondo caso OptorSim effettua una chiamata alla funzione client di LRCS implementata *ad hoc* per i nostri scopi.

```
lrcs-update-DBreplica [-h <host> -p <port> -v <vo> -d] -l <lfn> -u
<url>
```

Questa funzione avvia l'operazione di update sul LRCS server. Essa originariamente è stata implementata per essere una funzione interna alla procedura di update. Infatti, la chiamata avviene dopo il trasferimento dell'updateFile in locale con un "`lrcs->updateDBReplica(...)`". Ma poiché di questo trasferimento se ne occupa OptorSim, creare una funzione esterna alla procedura ci consente di richiamarla direttamente spezzando la procedura stessa.

I parametri di entrambe le funzioni vengono gestiti direttamente da OptorSim (si tenga conto che comunque si parla di localhost). Ogni chiamata avviene sfruttando la classe *Runtime* messa a disposizione da Java. Per esempio prendiamo la `lrcs-update-DBreplica(...)`

```
Runtime rt = Runtime.getRuntime();
try{
    Process cmd = rt.exec(lrcs-update-DBreplica -d -l VOMS -u VOMS2");
    try{
        int rc = cmd.waitFor();
        System.out.println("Update LRCS completo");
        }
    catch (InterruptedException e){
    System.out.println("Errore: update LRCS"+e)}
    }
catch (IOException e){
 System.out.println("Errore I\O: update LRCS"+e);}
```

Si avvia l'esecuzione del comando e poi si aspetta la sua terminazione (waitFor()) prima di proseguire. In caso di terminazione non corretta oppure timeout elevato si genera un'interruzione intercettata dal `catch` più interno. Invece, in caso di impossibilità di eseguire il comando, viene generata un eccezione di tipo I/O intercettata dal `catch` più esterno. Come si può notare questa struttura annidata delle eccezioni permette di gestire agevolmente tutte le condizioni di errore.

6.2 Modifiche relative a CONStanza

Il Log File Watcher non appena verificata una scrittura sul log file effettua la chiamata "`grcs->updateDB(...)`" per eseguire la procedura di update.

Una volta avviato il comando `updateDB(...)` il flusso di operazioni effettuate da CONStanza può essere diviso in tre parti:

1. Tutte le operazioni eseguite prima del trasferimento del updateFile
2. Il trasferimento vero e proprio del file
3. Aggiornamento delle repliche

Poiché il trasferimento del file deve avvenire su una Grid simulata il punto 2 è competenza di OptorSim. Nel corso della sua esecuzione, `updateDB(...)`, crea un `UpdateOperation` che in seguito avvia una routine per eseguire l'update delle repliche. Le modifiche apportate a CONStanza sono principalmente a livello di UpdateOperation e quindi concentrate nel file `UpdateOperation.cc`.

La funzione modificata è:

```
void* upd_routine_asynch(void* threadarg)
```

Questa è la routine che si incarica prima di cercare su RCC tutti gli LRCS che hanno una replica secondaria da aggiornare e poi di reperire il loro indirizzo per effettuare la chiamata di `updateDB(...)` su tutte le repliche. Questa seconda parte non viene più eseguita all'interno di questa routine. Infatti la parte relativa alla notifica ai LRCS è stata implementata da Optorsim, mentre l'aggiornamento delle repliche è stata implementata nell'interfaccia client di LRCS (lrcs-update-DBreplica).

La porzione di codice non eseguito di cui si è parlato precedentemente è:

```
// for (j = pmy_data->replicas.begin(); j != pmy_data->replicas.end(); ++j)
//     {rep = *j;
//      //get the LRCS stub
// if ((ret = pmy_data->pOp->getLRCSStubFromRep(rep, pmy_data->pOp->pRCC_,
//                     lrcsaddr, lrcs))) {
//          cerr << "UpdOp " << pmy_data->pOp->id_ << ":"
//               << "  error, in getLRCSStubFromRep() " << endl;
//          pthread_exit(NULL);
//      }
//      //call LRCS update replica method
//         #ifdef BENCHMARK
//             Timer queryTime("time");
//          unsigned long currentTime = 0; // benchmarks
//          currentTime = queryTime.Start();
//         #endif
//      if (pmy_data->pOp->fileFlag_)  // files
//          ret = lrcs->updateFileReplica(pmy_data->ldataset,
//                      rep, pmy_data->update, res);
//      else                // DB's
//          ret = lrcs->updateDBReplica(pmy_data->ldataset,
//                      pmy_data->update, res);
```

6.3 Modifiche relative a OptorSim

Mentre per CONStanza le modifiche sono essenzialmente concentrate su un unico file, per OptorSim le cose cambiano radicalmente. Possiamo fissare tre punti fondamentali nei quali i cambiamenti hanno maggior rilevanza:

- RCSim Server
- RCScontainer
- modifiche CE

6.3.1 RCSim

RCSim è un server multithread implementato principalmente per gestire la comunicazione con CONStanza. Quando inizia la simulazione, RCSim, è uno dei primi thread ad andare in esecuzione. Una volta avviato resta in attesa di una notifica di scrittura da parte di un CE. Non appena la notifica viene effettuata, essa viene servita da un ulteriore thread `updateDB` che si occupa sia della comunicazione con CONStanza nei modi precedentemente descritti, sia di effettuare la propagazione dell'update ai vari LRCS secondari. Descriviamo di seguito il metodo *run()*:

```
Runtime rt = Runtime.getRuntime();
try{
    Process cmd = rt.exec("/opt/lcg/bin/rcs-update-db -d -l VOMS
    -u VOMS2 -b");
    try{
        int rc = cmd.waitFor();
        System.out.println("Update GRCS completo");
    }
    catch (InterruptedException e){}
}
catch (IOException e){
    System.out.println("Errore: update GRCS"+ e);
}
GridContainer gc = GridContainer.getInstance();
for(int i=0; i<gc._LRCS.size(); i++ ){
    Integer s = (Integer) gc._LRCS.get(i);
    int lrcs = s.intValue();
    if (!(_sharedCell.setWriteSite(lrcs)))
        System.out.println("AVVIA UPDATE "+lrcs+"\n");
}
```

Dopo aver effettuato la chiamata a CONStanza il metodo `updateDB` scorre un vettore dinamico "`_LRCS`" che contiene l'identificativo numerico del CE

su cui è in esecuzione un LRCS secondario. Una volta individuata l'identità del CE con il metodo `setWriteSite(...)` si comunica a tale CE che c'è da effettuare il trasferimento dell'updateFile in locale per poi aggiornare le repliche.

6.3.2 RCScontainer

RCScontainer è una risorsa condivisa che permette l'iterazione e la sincronizzazione tra thread.

In figura 6.1 si illustra il diagramma delle classi.
Soffermiamoci sul metodo:

```
boolean setWriteSite(int siteIndex)
```

Questo metodo è quello che provvede all'update propagation. Esso va a modificare un vettore "`int[] _site`" di dimensione pari al numero di nodi della Grid. L'identificativo numerico di ogni nodo è associato all'indice del vettore, mentre il valore rappresenta il numero di trasferimenti ancora da eseguire.

6.3.3 Modifiche del CE

Il CE rappresenta la centrale di elaborazione della Grid simulata. Ogni job viene schedulato e inviato a un CE per essere elaborato e ricavarne le statistiche. La classe che implementa il CE in OptorSim è *SimpleComputigElement.java*. Nella nostra architettura il CE ha due importanti funzioni:

1. Inviare la notifica di update.

2. Effettuare il trasferimento e aggiornare le repliche

Per quanto riguarda il primo punto il CE si limita a verificare quando va in esecuzione il LFWatcher, e in base alla probabilità di scrittura, inviare la notifica al server RCSim.

```
......
if (job.LogFWatcher()){
    String _typeOfAccess="r";
    int winterval = (int)(job.writeProbability() * 100);
    Random _random = new Random();
    int _rw = _random.nextInt(101);
    if (_rw <= winterval){
```

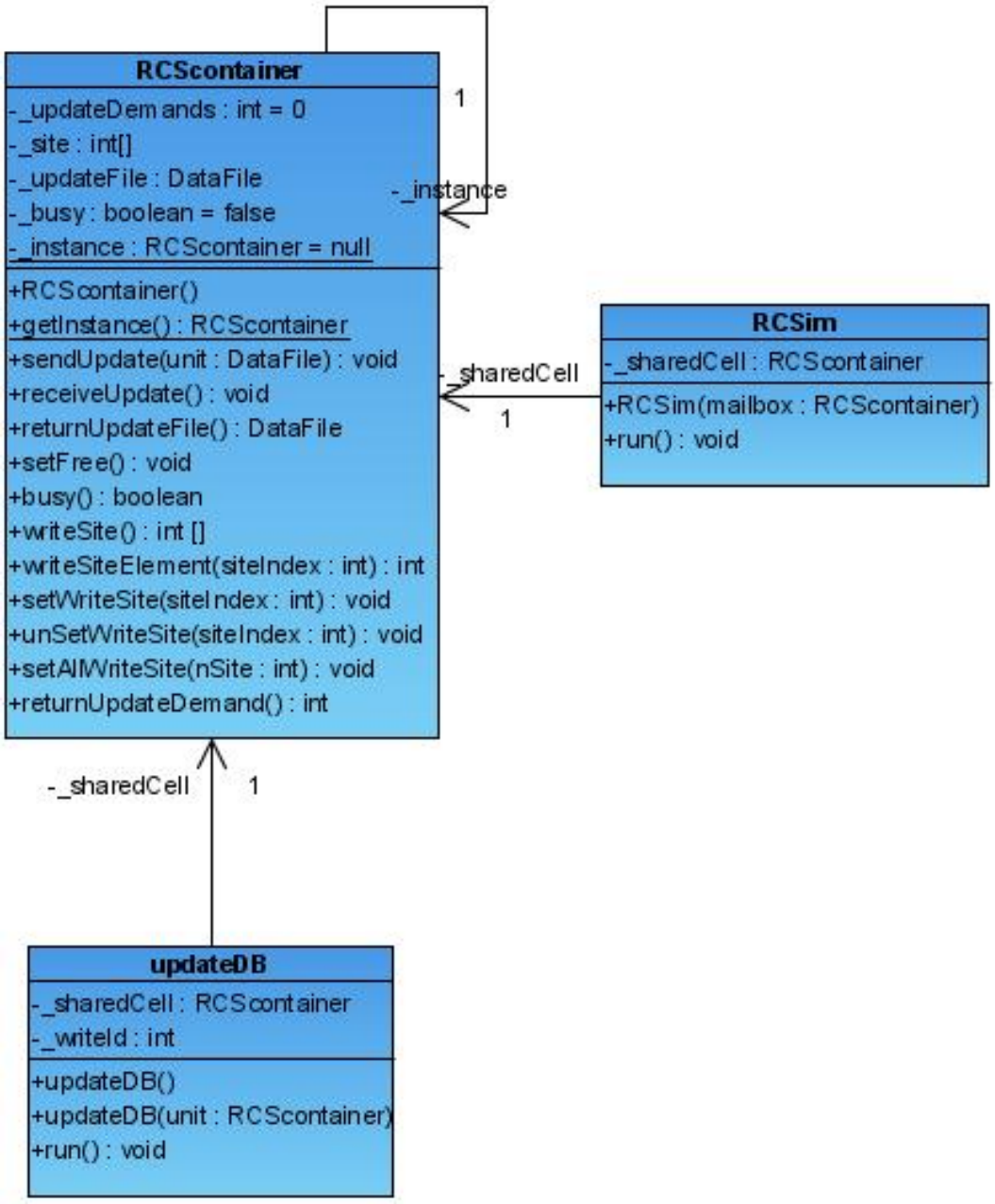

Figura 6.1: Diagramma delle classi RCSim

```
        _typeOfAccess = "w";
        System.out.println('\n'+_ceName+">"+"  LogFile modified..."+'\n' );
    }
    if (_typeOfAccess == "w") {
        //write operation
        files[0].isMaster();
        synchronized (_sharedCell){
        _sharedCell.sendUpdate(files[0]);
    }
......
```

Per il secondo punto il CE va a leggere nel vettore `_site` di RCScontainer. Accede all'elemento il cui indice è indicato dal proprio identificativo numerico e verifica se è maggiore di zero. In tal caso effettua l'operazione di scrittura e poi quella di update della replica.

```
......
if(_sharedCell.writeSite()!=null){
    if(_sharedCell.writeSiteElement(_site.exposeIndex())>0){
        DataFile logFile = _sharedCell.returnUpdateFile();
        simulateRemoteIO( logFile,1);
        _remoteReads++;
        Runtime rt = Runtime.getRuntime();
            try{
                Process cmd = rt.exec("/opt/lcg/bin/lrcs-update-DBreplica
                -d -l VOMS -u VOMS2");
                }
        ...................
        ...................
        synchronized (_sharedCell){
            sharedCell.unSetWriteSite(_site.exposeIndex());
        }
        _writeIndex++;
        System.out.println('\n'+_ceName+">"+"
        write complete_"+_writeIndex+'\n');
     }
}
......
```

Come si può notare il trasferimento viene fatto mediante la `simulateRemoteIO(..)` di questo se ne tiene traccia nelle statistiche incrementando il valore delle letture remote. Infine si decrementa il vettore `_site` per segnalare che una delle scritture pendenti è stata eseguita. A questo punto vediamo come è cambiato il diagramma di flusso di un generico job che utilizza il simulatore (Figura 6.2).

6.4 Compilazione e installazione

Da questa sezione in poi, per evitare confusione, chiameremo RCSim_1.0 il simulatore ottenuto dall' integrazione di OptorSim e CONStanza. Per RCSim_1.0 la compilazione così come l'installazione prevede lo stesso procedimento adottato per compilare e installare separatamente i due software. Per i dettagli della compilazione e l'installazione CONStanza e OptorSim faremo riferimento ai rispettivi manuali [21, 11].

Si parte scompattando il file `RCSim_1.0.tar.gz`, a questo punto ho a disposizione due cartelle: una contiene il codice modificato di OptorSim, l'altro di CONStanza. Si continua compilando e installando il software separatamente.

6.4.1 CONStanza

La versione di CONStanza su cui abbiamo lavorato è la rel-0.12. Si entra nella directory `rcs-0.12` e si procede alla compilazione.

Compilazione

Per la compilazione sono necessari i seguenti software esterni:

MySQL 4.1.10 o superiori disponibile al sito
`http://dev.mysql.com/downloads/mysql/5.0.html`

MySQL C++ API 1.7.40 disponibile al sito
`http://tangentsoft.net/mysql++/`

gsoap_linux_2.7.6 disponibile al sito gSOAP
`http://www.cs.fsu.edu/ engelen/soap.html`

LCG 2.7 CE il RCS software è stato testato su una distribuzione SLC3 con una parziale installazione del software LCG i cui pacchetti necessari sono contenuti in [21]

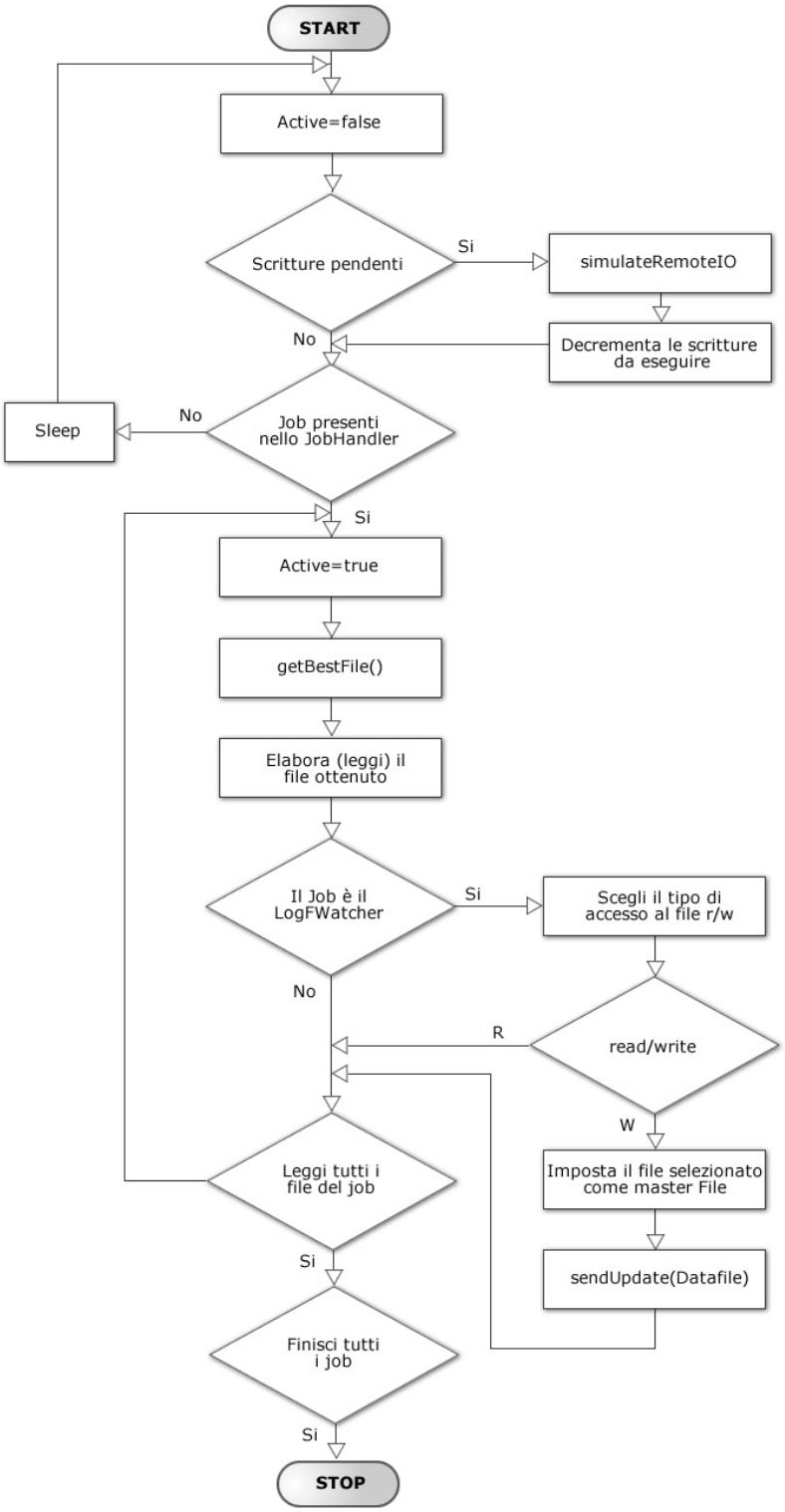

Figura 6.2: Diagramma del flusso di esecuzione di un generico job

Installazione

L'installazione di CONStanza avviene come un qualsiasi software autoconfigurante. La procedura si riassume nei seguenti passaggi:

1. `./autogen.sh`, per creare i file di configurazione.
2. `./configure`, per configurare CONStanza nel tuo sistema.
3. `make`, per compilare il programma.
4. `make install`, per installare il binario.
5. `make clean`, per ripulire la directory da librerie e file eseguibili.
6. `make clean-tree`, per ripulire e ripristinare il CVS tree.

Per ulteriori dettagli sull'installazione si faccia riferimento a [21].

6.4.2 OptorSim

OptorSim per la sua esecuzione necessita di una *Java Virtual Machine*. Il Java Runtime Environment è disponibile al sito:

`http://www.java.com/it/download/index.jsp`.

Una volta installato l'ambiente Java si entra nella directory optorsim-2.0 modificando lo user path includendo la cartella /bin che contiene i vari script utilizzati da OptorSim:

```
$ export PATH=$PATH:$PWD/bin
```

A questo punto si è pronti a lanciare l 'esecuzione di Optorsim con il comando:

```
$ optorsim.sh [parameters file]
```

Per ulteriori dettagli sull'installazione si faccia riferimento a [11].

Una volta compilato e installato RCSim_1.0 è pronto all'esecuzione.

6.5 Configurazione

La configurazione di RCSim_1.0 (come nella sezione precedente) dipende dalla configurazione di entrambi i software che lo compongono.

6.5.1 CONStanza

Per quanto riguarda CONStanza innanzitutto si deve configurare MySQL concedendo all'utente, su tutte le tabelle dei database che andremo a adoperare, tutti i privilegi necessari per potervi accedere [21].

Ipotizzando un utente 'rcs' identificato dalla password 'constanza' il comando per esempio può essere:

```
   mysql> grant all on RCC.* to‘rcs’@‘localhost’
identified by ‘constanza’
```

Dopodiché si può avviare il servizio con un apposito script:

```
$ ./service.sh [numero di LRCS secondari da avviare]
```

Questo script avvia un thread relativo al GRCS server e tanti thread relativi a LRCS server quanto il numero indicato come parametro dello script. La parte CONStanza si trova così pronta all'uso con tutti i thread operativi.

6.5.2 OptorSim

La parte di configurazione OptorSim è quella che gestisce l'ubicazione all'interno della Grid dei vari GRCS e LRCS server utilizzati per il funzionamento del sevizio di consistenza. Il file da modificare per ottenere diverse configurazioni è il file di configurazione di job.

Nel configurare il nostro RCSim_1.0 dobbiamo scegliere :

1. quale CE fungera da GRCS server e quindi LRCS master server.

2. quali CE fungeranno da LRCS secondary server.

Nel primo caso basta inserire nella tabella di schedulazione dei job, in corrispondenza del CE che vogliamo far diventare master server, il job **LFWatcher**.

```
#
# CE Schedule Table
# CE site id, jobs it will run
#
\begin{cescheduletable}
......
7  highptphotjob zbbbarjob LFWatch
.....
\end{cescheduletable}
```

L'inserimento del **LFWatcher** fa diventare il CE7 master. Il **LFWatcher** deve essere unico poiché CONStanza per ora supporta solo la modalità single master, in caso contrario si potrebbero verificare malfunzionamenti a livello di replicazione di dati.

Il secondo caso è molto simile al primo, in quanto, al posto del job **LFWatcher** si utilizza **LRCS**.

```
#
# CE Schedule Table
# CE site id, jobs it will run
#
\begin{cescheduletable}
......
3  highptphotjob zbbbarjob LRCS
.....
12 highptphotjob zbbbarjob LRCS
.....
\end{cescheduletable}
```

Il job **LRCS** in corrispondenza del CE3 e del CE12, ci indica che su questi nodi gira in LRCS secondario. A differenza del **LFWatcher** per i job **LRCS** non ci sono limitazioni. Essi possono essere eseguiti anche su tutti i CE se questi vengono considerati server LRCS secondari.

In questo modo RCSim_1.0 possiede dei server veri e propri (forniti da CONStanza) localizzati su una Grid simulata (fornita da OptorSim).

6.6 Avviare una simulazione

Non appena configurato RCSim_1.0 possiamo avviare una simulazione in qualsiasi momento usando i comandi di Optorsim. Infatti l'integrazione è avvenuta pienamente e una volta avviata la simulazione il servizio di consistenza sarà incluso. Naturalmente vi è la possibilità di usare anche la GUI per avviare la simulazione con il vantaggio di avere a disposizione un'interfaccia grafica.

6.6.1 Output

Gli output del simulatore RCSim_1.0 sono nello stesso formato di quelli di OptorSim. Se non si usa la GUI tutte le informazioni e statistiche vengono stampate sullo standard output.

Se invece su usa la GUI alla fine della simulazione un utente deve salvare i risultati scegliendo la voce `Produce Summary` dal menu `File`.

I risultati, sotto forma di grafici `jpeg`, vengono inseriti in una directory chiamata `AllGraphics@sim_completed` all'interno della propria directory di lavoro.

Il file `Summary@sim_completed.html` possiede un link per ciascuno dei grafici. Le figure di seguito mostrano i grafici principali.

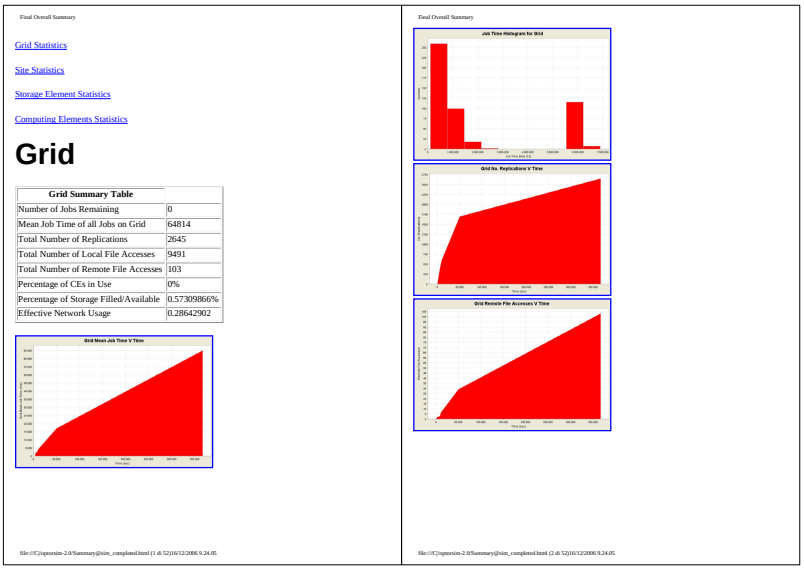

Figura 6.3: Esempio di statistiche sulla Grid

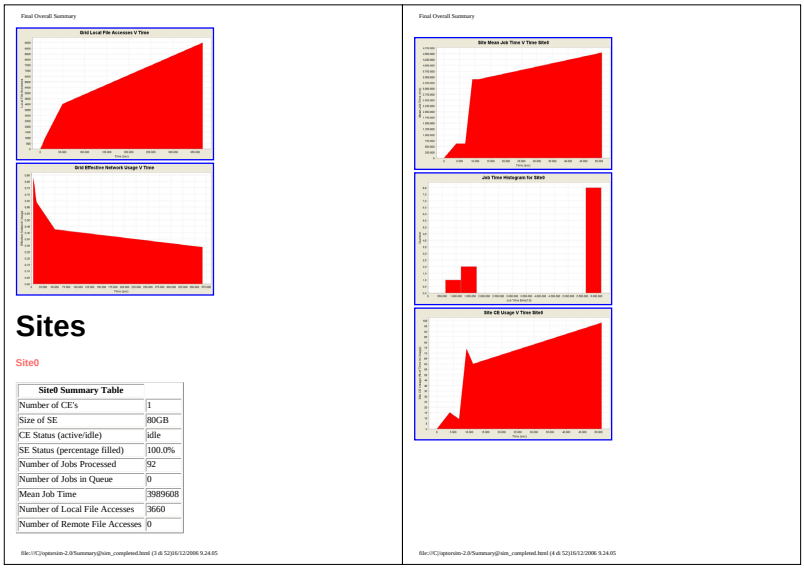

Figura 6.4: Esempio di statistiche sui siti

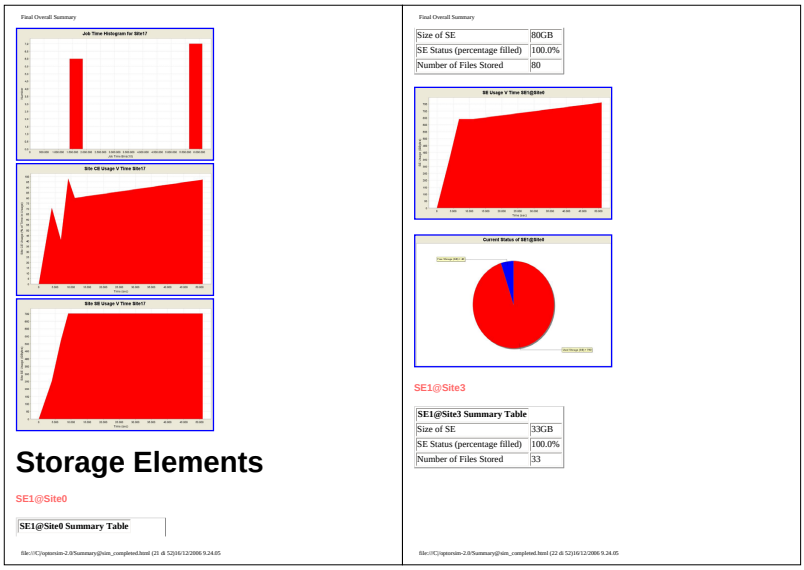

Figura 6.5: Esempio di statistiche su un SE

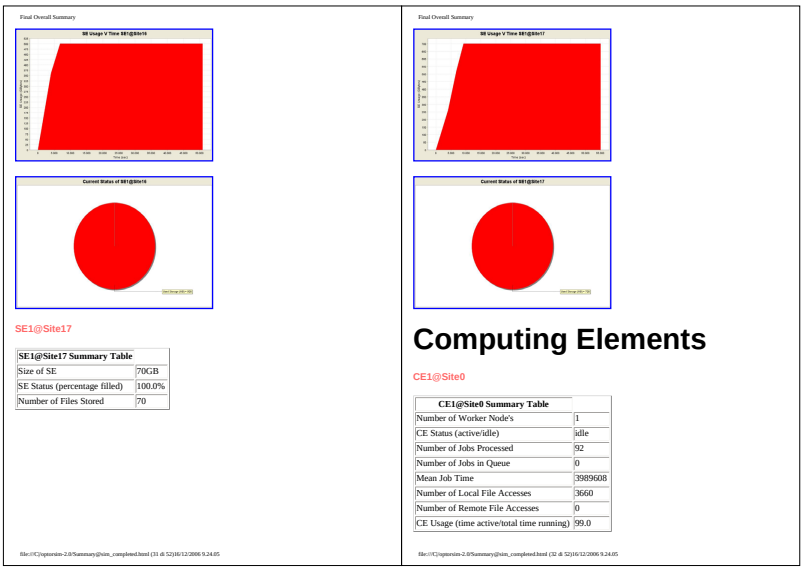

Figura 6.6: Esempio di statistiche su un CE (parte prima)

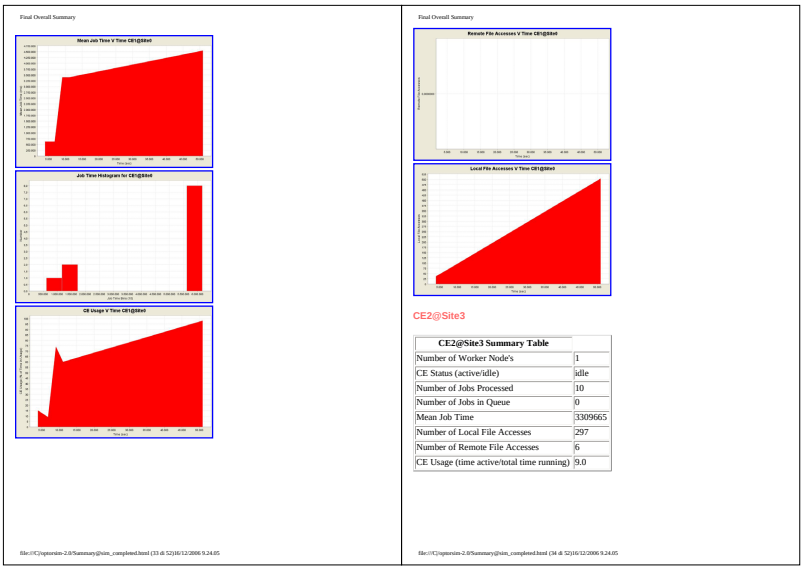

Figura 6.7: Esempio di statistiche su un CE (parte seconda)

Conclusioni

Questo lavoro mi ha permesso di approfondire sia le problematiche relative alla simulazione di un ambiente Grid, che alla replicazione dei dati in tale ambiente.

Prima abbiamo analizzato i problemi generali della simulazione di un ambiente Grid soffermandoci sugli aspetti fondamentali da tenere in considerazione per una simulazione.

Poi abbiamo descritto i due software che compongono il simulatore, ossia CONStanza e OptorSim, delineandone architettura e funzionamento per comprendere meglio come potessero interagire tra loro.

Nell'ultimi due capitoli ci siamo concentrati sulla struttura del simulatore stesso, specificando come siano stati integrati CONStanza e OptorSim, integrazione rivelatasi difficoltosa per i diversi linguaggi di programmazione utilizzati per l'implementazione dei due softwares, ossia C++ e Java. Inoltre riferendoci allo scenario specifico di sincronizzazione di database abbiamo visto come servizio di consistenza di una Grid reale, possa essere modificato in modo da essere eseguito su una Grid simulata.

Infine abbiamo ottenuto un simulatore in grado di valutare la scalabilità del servizio di consistenza mettendo a disposizione diverse configurazioni di Grid simulate.

Il simulatore che abbiamo realizzato è un prototipo che potrà essere migliorato e ampliato.

Per ora, l'architettura di CONStanza, implementa soltanto il meccanismo di update basato su un modello asincrono con approccio single master. Prevedendo lo sviluppo di altri protocolli di consistenza all'interno del progetto CONStanza, una estensione del simulatore potrebbe essere uno strumento utile per valutare il loro impatto su diverse configurazioni di Grid.

Un'altro argomento interessate è quello della replicazione dei dati. OptorSim implementa diverse strategie di replicazione dei dati. Sfruttando questa caratteristica, per ogni protocollo di consistenza, si potrebbero valutare le migliori strategie di replicazione al fine di ottenere la migliore combinazione possibile per massimizzare le prestazioni della Grid.

In sintesi questo lavoro ha fornito la base per un simulatore che potrà svilupparsi insieme al progetto CONStanza. La possibilità di integrare nel prototipo del simulatore ogni nuova funzionalità di CONStanza permette di ottenere una simulazione sempre più evoluta di un servizio di consistenza.

Bibliografia

[1] AVAKI website, http://www.avaki.com

[2] Beowulf website, http://www.beowulf.org/

[3] http://www.pi.infn.it/constanza/

[4] Entropia website, http://www.entropia.com

[5] Globus website, http://www.globus.org/

[6] Global Grid Forum website, http://www.gridforum.org

[7] GridSim website, http://www.gridbus.org/gridsim/

[8] The toolkit gSOAP site, http://www.cs.fsu.edu/ engelen/soap.html

[9] INFNGrid project web site. http://grid.infn.it

[10] Sito del progetto OGSA, http://www.globus.org/ogsa/

[11] Optorsim website,
http://edg-wp2.web.cern.ch/edg-wp2/optimization/optorsim.html

[12] SETI@home website, http://setiathome.ssl.berkeley.edu/

[13] SimGrid website, http://simgrid.gforge.inria.fr/

[14] VOMS website, http://www.egrid.it/doc/sysadm/voms

[15] Appleby, K., S. B. Calo, J. R. Giles, and K. W. Lee. 2004. Policy based automated provisioning. *IBM Systems Journal***43**: 121-135.

[16] Bagchi, S. 2005. Simulation Of Grid Computing Infrastructure: Challenges And Solutions. *In Proceedings of the 2005 Winter Simulation Conference* ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 1773-1778.

[17] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. Simulation of dynamic grid replication strategies in optorsim, 2002.

[18] Cameron, D. G., R. Carvajal-Schiaffino, J. Ferguson, A. P. Millar, C. Nicholson, K. Stockinger, and F. Zini. OptorSim v2.0 installation and user guide. Technical report, 2004.

[19] Cameron, D. G., R. Carvajal-Schiaffino, A. P. Millar, C. Nicholson, K. Stockinger, and F. Zini. 2003. Evaluating scheduling and replica optimisation strategies in OptorSim. *Proceedings of the Fourth International Workshop on Grid Computing* (GRID '03).

[20] D. Cameron, J. Casey, L. Guy, P. Kunszt, S. Lemaitre, G. McCance, H. Stockinger, K. Stockinger, G. Andronico, W. Bell, I. Ben-Akiva, D. Bosio, R. Chytracek, A. Domenici, F. Donno,W. Hoschek, E. Laure, L. Lucio, P. Millar, L. Salconi, B. Segal, M. Silander. Replica Management in the EU DataGrid Project, International Journal of Grid Computing, Springer Science+Business Media B.V., Formerly Kluwer Academic Publishers B.V., 2(4):341-351 (2004).

[21] A. Domenici, F.Donno, G. Pucciani, H. Stockinger, and K. Stockinger. CONStanza RCS User Guide. 2005

[22] A. Domenici, F.Donno, G. Pucciani, H. Stockinger, and K. Stockinger. 2004. Replica consistency in a Data Grid. *Proceedings of the IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Tsukuba, Japan, December 1-5, 2003

[23] A. Domenici, F.Donno, G. Pucciani, H. Stockinger, and K. Stockinger. 2004. CONStanza: Data Replication with Relaxed Consistency. *Technical Report INFN / TC06 / 6*

[24] DAIS Working Group, Services for Data Access and Data Processing on Grids, GFD-I.14, Global Grid Forum (2003).

[25] D. Düllmann, W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, K. Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid, *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)* (San Francisco, California, August 7-9, 2001).

[26] Eun-Kyu Byun, Jae-Wan Jang, Wook Jung, Jin-Soo Kim. 2005. A dynamic grid services deployment mechanism for on-demand resource provisioning. *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid* (CCGrid'05) Volume 2, 863-870.

[27] EURESCOM 806-GI: *A Common Framework for QoS/Network Performance in a Multi-Provider Environment.*

[28] Foster, I., C. Kesselman, J. M. Nick, and S. Tuecke. 2002. Grid services for distributed systemintegration. *IEEEComputer* June 2002: 37-46.

[29] Foster I., Kesselman C.: The Globus Project: A Status Report, Proceedings of the Seventh Heterogeneous Computing Workshop. *IEEE Computer Society Press*, March 1998, p. 4-19.

[30] Foster I., Kesselman C., (eds.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, 1999.

[31] Foster I., Kesselman C., Globus: A Metacomputing Infrastructure Toolkit.*Intl J. Supercomputer Applications*, 11(2):115-128, 1997.

[32] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[33] Franken L.J.N., *Quality of Service Management: a Model-Based Approach.* Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1996

[34] Gelsinger P., Gargini P., Parker G., Yu A.: Microprocessors circa 2000. *IEEE Spectrum*, Ottobre, 1989.

[35] Gianni Pucciani, Tesi di Laurea. *Simulazione di modelli di consistenza per file replicati su sistemi Grid*, 2003.

[36] Howell, F. and R. McNab. 1998. SimJava: A discrete event simulation library for Java. In *Proceedings of the 1998 International Conference on Web-Based Modeling and Simulation*, 51-56. The Society for Computer Simulation International, San Diego, CA.

[37] Krishnamurthy S., Sanders W.H., Cukier, M. Performance evaluation of a QoS-aware framework for providing tunable consistency and timeliness, *Quality of Service, 2002. Tenth IEEE International Workshop on* 214-223.

[38] P. Kunszt, E. Laure, H. Stockinger, K. Stockinger. Advanced Replica Management with Reptor. In *5th Int. Conf. on Parallel Processing and Applied Mathematics* (Czestochowa, Poland, September 7-10, 2003).

[39] LDAP Directories Explained: An Introduction and Analysis
By Brian Arkills. Published by Addison Wesley Professional. Series: Independent Technology Guides. 2003.

[40] Legrand, A., L. Marchal, and H. Casanova. 2003. Scheduling distributed applications: *The SimGrid simulation framework. Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* (CCGRID.03).

[41] Moore G.: *Cramming more components onto integrated circuits.* Disponibile on-line, Electronics, Aprile, 1965.

[42] Phatanapherom, S. and V. Kachitvichyanukul. 2003. Fast simulation model for grid scheduling using HyperSim. In *Proceedings of the 2003 Winter Simulation Conference* ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 1494-1500.

[43] Heinz Stockinger. Distributed Database Management Systems and the Data Grid. *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 17-20, 2001.

[44] Sunderam V.S., PVM: A Framework for Parallel Distributed Computing, Concurrency, Practice and Experience, 1990.

[45] Yaohang Li, Michael Mascagni. 2002. Grid-Based Monte Carlo Application. *Proceedings of the Third International Workshop on Grid Computing* , 13-24